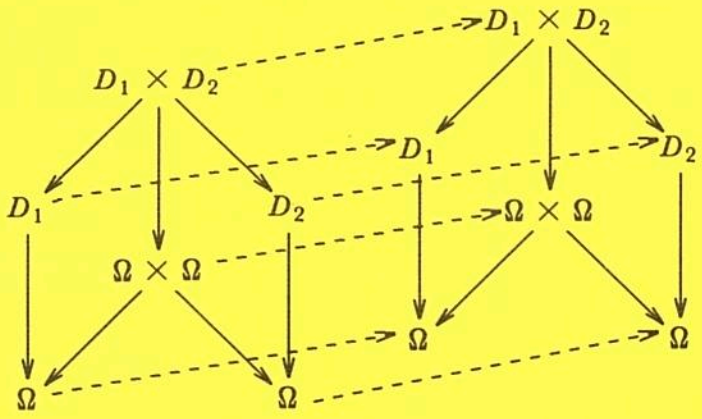
On Complex Objects



Arno Siebes

On Complex Objects

On Complex Objects

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof.dr.ir.J.H.A. de Smit,
volgens besluit van het College van Dekanen
in het openbaar te verdedigen
op vrijdag 9 maart 1990 te 14.00 uur.

door

Arnoldus Paulus Johannes Maria Siebes

geboren op 27 juni 1958

te Gouda

Promotor:	Prof. Dr. P.M.G. Apers
Ass. promotor:	Dr. M.L. Kersten
Referent:	Dr. H. Balsters

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Siebes, Arnoldus Paulus Johannes Maria

On complex objects / Arnoldus Paulus Johannes Maria
Siebes. - [S.l. : s.n.]. - Ill.
Proefschrift Enschede. - Met lit. opg.
ISBN 90-9003271-1
SISO 520 UDC 681.3(O43.3)
Trefw.: informatica

To Inge

CONTENTS

Preface

Writing a thesis serves two purposes. Its first purpose is not unlike the masterpiece that a medieval novice had to produce to prove his mastership to the members of the guild. Thus, it should not be considered a masterpiece in the modern sense but, rather, an exhibition of acquired skill. Its second purpose is that it gives the novice four years to get thoroughly acquainted with a particular field of knowledge. For it is well known that the best way to learn a subject is to do research in it.

The writing of this thesis agrees with the above description but, rather than studying one field, the author studied two, viz., database theory and category theory. The knowledge the author gained in the latter field is applied to the research in the former field. The database theory in this thesis focusses on the subject of complex objects, a topic in object-oriented database research, while the use of category theory is limited to its basic notions and results.

The advantage for the database field is that category theory shows that one can build an elegant and strong theory by adhering strictly to a few simple principles. The advantage for category theory is perhaps that it is applied in this thesis, because this illustrates that it is not the general abstract nonsense it is sometimes held to be.

Of course, not all readers will share the author's interests. Readers more interested in database theory than in category theory may skip the last part of Chapter 3, and Chapter 7 in its entirety. This means, however, that they will not be able to read the formal semantics of Flock, the formalism developed in this thesis. On the other hand, all concepts in this thesis are first developed intuitively, often using set theory, before they are being formalised using category theory. So, the disadvantage is less then it may appear.

To those readers who prefer category theory over database theory, we have little to offer. All the category theory introduced in this thesis can be found elsewhere in the standard literature. We merely treat those topics needed for the development of Flock. Their only solace is, perhaps, that this thesis gives an example how their favorite topic applied in yet another field.

Acknowledgements

Although a thesis is meant as a proficiency test to the student, there are many people I have to thank, for I would not have succeeded without their continuing support.

First of all I have to thank Martin Kersten for taking me aboard his project even though I knew next-to-nothing about database theory. Moreover, I am grateful that he allowed me the freedom to pick those topics I was interested in, even though they did not match his favorite topics. His enthusiasm and unabated trust that eventually something would come out of this struggle has been a constant support throughout these years. Similarly, I am grateful to Peter Apers, for agreeing to be my promotor, although the topic was even further from his research interests.

I thank Lambert Meertens for the many discussions we have had through the years. The topics in these discussions ranged from theoretical physics, in which we are both interested amateurs, through Objectivism and other Usenet extravagances, to work reported in this thesis. Through these discussions he has taught me many things on the nature of research and proofs. Moreover, he has always been willing to lend me a hand in my continuing struggles with troff.

I owe many thanks to the members of the DAISY Project and the inter-university Study Group on Object-Oriented Databases, especially the sub-group on type theories for Object-Oriented Databases. Our discussions have taught me many important notions and problems in this field. Similarly, I learned much from the CWI Study Group on Category Theory.

I happily acknowledge my debts to the Amoeba group, and especially Sape Mullender, for supporting the writing of this thesis both with hardware and software.

On a more social level, I owe many thanks to the members of the AA department and CWI in general, for the pleasant atmosphere that stimulated both pure research and applied research in the products of breweries around the world. Special thanks are due to Frank van Dijk for puttting up with my tempers, good or bad, for over three years.

Finally, I owe many thanks to my family and friends. Their continuing moral support has made my life a pleasant one; research would not have been possible otherwise.

Gratitude is too small a word for thanking Inge, her presence and support were indispensable for the work reported here. Her ability to make our house an enjoyable place despite a demanding job, helped me through the last few hectic months. To her I dedicate this thesis.

Chapter 1

Introduction

The concept of *complex object*, the subject of this thesis, arises whenever a data model has more than one type constructor and it supports inductive type definitions; a complex object is a constant of a thus constructed type. Using the dictionary, we† can give a less technical description of complex objects. Combining the entries for *complex* and *object* we may construct the following description for a complex object:

> *Something composed of several parts, that can be known or perceived by the mind.*

This description is close to the common usage of the term in Computer Science.

As indicated by the technical description, complex objects do not appear in the, by now, usual relational databases. In fact, complex objects are often found in association with *object-oriented databases*. The area of object-oriented databases is an example of the continuing effort of the database research community to define new, more powerful, data models. While the relational data model is well on its way to conquer the business world, the database research community is well on its way to define new data models. This thesis presents yet another attempt in this search for a new model. The motivation and the goals of this thesis are given succinctly below. The rest of this chapter is an extensive discussion of this motivation and these goals.

The motivation for this ongoing search for new data models can be found in theoretical and practical deficiencies of the relational model. At the time of

† The use of 'we' in this thesis is not intended as *pluralis majestatis*; we merely follow the convention in scientific literature. Moreover, it expresses that many of the ideas used in this thesis are inspired by discussions with the people already mentioned in the acknowledgements. The fact that there is only one author indicates that the contents is the responsibility of only one person.

writing, there are two main directions: object-oriented data models and deductive databases. These two directions are largely complementary in the deficiencies they try to solve. Therefore, it is worthwhile to strive for a *unified model*, that unites the strong points of both directions.

The concept of 'complex object' can be seen as one of the strong points of object-oriented data models. Moreover, some notion of complex object also appears in deductive data models. So, complex objects clearly belong in such a unified data model. The goal of this thesis is the development of a formalism of complex objects that satisfies the requirements of both approaches. The formalism developed in this thesis is called *Flock* which abbreviates *Formalisme pour Les Objects Complex*. We have chosen this name, because a Complex Object Language is already defined by a group of French researchers (Abiteboul and Grumbach (1987)). Moreover, the word itself symbolises a database rather well.

The definition of a formalism consists of two parts, viz. syntax and semantics. For the semantics, we have chosen an unconventional vehicle, namely Category Theory. The motivations for this choice are:

1) The theory is constructive; this implies that the semantics of Flock are adequate as a specification of an implementation.

2) The theory is fundamental; this means that we need only a few concepts, but these concepts are strong enough to develop, e.g. the Set Theory and the Logic Theory, we need in this thesis. As a result, we do not have to use an amalgamation of different mathematical theories to provide Flock with semantics.

3) The theory is graph-oriented; this gives a natural translation of our concept of a *type graph*. Moreover, the type constructors of Flock have a natural counterpart in Category Theory.

The drawback of the choice for Category Theory is reflected in the size of this thesis. As it is a relatively unknown field, we have to introduce many of its concepts and theorems along the way. However, we feel that the advantages outweigh the disadvantages.

The rest of this thesis is an elaboration of the short description given above. Our first concern is the question 'why new data models?'

Similar to Bancilhon (1988), we answer this question by going back to the basic problem addressed in the area of database research, which we describe in Section 1. In the second section, we give a short introduction to the relational data model, as well as a brief overview of results in relational database theory on data modelling. Its purpose is three-fold. First, this section argues the inadequacy of the relational data model. Secondly, it serves as an introduction to the uninitiated reader. Not so much as a crash course in database theory, but as an explanation of some of the problems addressed in this field. Finally, the results presented in this section, provide a measure for Flock to be considered a more powerful data model. For the initiated reader, this section contains nothing new.

Further evidence of the inadequacy of the relational data model is given in the third section. In this section, we give a brief overview of the area of *information systems design*. This field is mainly concerned with methodologies for database design and comprises much knowledge distilled from the actual design of databases. Consequently, this is an area that poses many requirements on data models. These requirements can be used to appraise the relational data model; or any data model, for that matter.

In the fourth section, we turn our attention to the two predominant research directions mentioned above, viz. object-oriented data models and deductive databases. We briefly describe and compare both areas (an extensive description is the subject of Chapter 2). In the fifth section, we use the comparison of these two approaches to derive some requirements for a *unified* approach, i.e. for Flock. In the sixth, and final, section, we formulate the thesis of our study in terms of the requirements of the fifth section. Moreover, we give a summary of our main results in an outline of the rest of this dissertation.

1.1 Databases as models

In the introduction to this chapter, we mentioned that the search for new data models is motivated by deficiencies of current models. In this context, a deficiency means that the current data models are not sufficient for the purposes of a data model. So, to illustrate the deficiencies, we should describe the purposes and the functionality of a data model. In this section, we try to give such a description by returning to the basic question that initiated database research. More in particular, we seek an answer to the question what should be described in a database.

One might say that automated data processing began when organisations started to transfer their manual operations to computers. The computers offered economical, high speed accurate data processing. This transfer was more or less successful, however, problems arose concerning data availability. Each program had its own data in a specific format on a file. This implied that a lot of data was replicated a number of times in different files, often in different formats. In turn, this caused that if there was a change in the data, all these different representations had to be updated. Moreover, it became difficult for a user 'to obtain, integrate, or transform the available data' for various uses (Sibley (1976)).

The integration of all the available data gave rise to the concept of a database management system. So, the subject *Databases* of computer science has grown out of the following problem:

The efficient management of large amounts of persistent, reliable, and shared data.

In this formulation, 'large' means too big to fit in main memory; 'persistent' means that the data persists from one session to another, regardless of the state of

the underlying hardware. 'Reliable' means recoverable in case of a hardware or software failure. 'Sharable' means that several users should be able to access the data in an orderly manner. 'Efficient' means that both storage size and response time should be acceptable. 'Management', finally, means prescriptive control. As an aside, note that the 'largeness-problem' is gradually disappearing, as the decrease in the price of memory motivates the building of main memory database machines; see e.g. (Kersten *et al.* (1988)).

Sharing data by different users in different sessions is probably *the* feature that distinguishes a database management system from other large programming problems. If only because this problem strongly influences the architecture of a database management system. This is witnessed by the influential ANSI/SPARC report (Tsichritzis and Klug (1977)) in which two important aspects of data sharing are recognised:

1) If users and application programs use the physical layout of the data, any change in this layout requires a major update in all applications that use this data. Because the knowledge of the data organisation and the access techniques is built into the program. Such (physical) data dependence is obviously a non-desirable property that should be avoided as much as possible. This leads to the concept of *data independence*, the DataBase Management System (DBMS) should provide uniform access to the data, and in no way reflect the actual data organisation and the actual access method. The result is that application programs become largely immune to changes in organisation and access methods, which may e.g. occur for efficiency reasons.

2) The second aspect is that no user or application program is interested in (or has privileges to access) all data. In fact, it is possible for two users to view the same data in two radically different ways. Thus, although the database itself contains all the data in an unambiguous way, it should be able to reflect the actual view a user requires on that data. This leads to the concept of a *user view*, which is a representation of the database's contents, constructed from the actual database, tailored towards the wishes and privileges of the user (or application program), such that manipulation and queries on such a user view can be translated to manipulation and queries on the actual database contents.

For a DBMS to provide for these two aspects, the ANSI/SPARC report divides the architecture of a DBMS into three levels:

1) the internal level,

2) the conceptual level,

3) the external level.

The internal level is the one closest to physical storage. This is the level that is concerned with the way the data is actually stored, i.e. it is the level of the DBMS that 'knows' the various ways of data organisation and access methods. However, it does not know anything about physical details, such as blocks or sequencing on disks, and neither does it know about direct access to the raw data; this knowledge belongs to the access methods for the associated file structure. It does know the definition of the storage structure, i.e. which files there are, what their structure is and what their associated access methods are. This description is known as the internal view or the internal schema. The access method can be seen as a mapping from the internal level to the actual physical level. This mapping belongs to the operating system and it is of no concern to us here.

The external level is the one closest to the user. This is the level that is concerned with the way the data is viewed by the users, i.e. the level that contains the descriptions of the various user views or *external* views. Moreover, it provides the database designer with a Data Definition Language (DDL) and the user with a Data Manipulation Language (DML). The user view is written in the DDL, while its data is manipulated via the DML. Of course there has to be a mapping from the external level to the conceptual level, which is introduced below.

At the intermediate level, ANSI/SPARC defines the conceptual level. While the external level is concerned with the individual user views, the conceptual level is concerned with the unified view on the data. In particular, while each user view is an abstract description of a portion of the database, the conceptual *schema* is an abstract description of the database in its entirety. By definition, all user views are derived from the conceptual schema. The conceptual schema has in principle its own (conceptual) DDL and (conceptual) DML. In practice, however, these are often identical to their counterparts at the external level. Similar to the external schema/conceptual schema mapping, there has to be a conceptual schema/internal schema mapping, which is also introduced below.

Before we introduce the two mappings connecting the three levels, note that the three levels of a DBMS consist of schemas. These schemas, together with the mappings are themselves a database! This database is often known as the data dictionary or DD/DS (where DS stands for data storage). Considering the data dictionary as a database leads in turn to a new data dictionary; a higher level dictionary. So, naively, the idea of the dictionary as a database, leads recursively to an infinite number of databases. In practice, this problem is solved by simply not considering the data dictionary as a database, but as a hard-wired component in the DBMS. This is a very unsatisfying solution, if only, because the data dictionary is also subject to change, albeit at a slower rate than the 'first-order' database, to reflect changes in the outside world. A more satisfying solution to this problem is the *self-descriptive database*, see e.g. (Mark (1985)). A further treatment of this subject is beyond the scope of this thesis.

The *conceptual schema/internal schema* mapping defines how the conceptual data is mapped to the 'stored' data and vice versa, and how manipulation of the

conceptual data is transformed to manipulation of the stored data. If the storage schema changes, e.g. for efficiency reasons, this mapping has to change accordingly to reflect these changes; i.e. the conceptual schema is data independent. Similarly, the *external schema/conceptual schema* mapping defines the correspondence between the external schemas and the conceptual schema, and translates manipulations on the external level to manipulations of the conceptual level. Moreover, if the conceptual level changes, this mapping should change accordingly; i.e. the external schemas should only change if they really have to.

The conceptual level is the crucial level of the database architecture with regard to the sharing of data. A conceptual schema compromises a unique description of the information content of the data in the database. All user views (or external schema's) are derived from this conceptual schema. Thus if the users want to share data, they should agree on the conceptual schema. In other words: the conceptual schema reflects the common understanding of the data in the database, and, therefore, the conceptual schema should represent the 'real world' concepts as accurately as possible. Only then can the database represents the real world as accurately as possible. But how do we represent the fundamental concepts of the real world that are necessary to structure the data, and, perhaps even more important, what are these fundamental concepts?

In the early 1980's, ISO TC97/SC5/WG3 defined the contents and concepts of a conceptual schema (van Griethuysen (Ed.) (1984)). As a basic notion they used the concept of an *entity*; they rephrased the definition of an entity as follows:

> *Any concrete or abstract thing of interest, including associations among things.*

The second important notion is that of the *Universe of Discourse* (UoD); informally, the UoD is the set of those things and happenings to which the common understanding of the represented data refers. Or, connecting the concept of UoD's with databases: a database describes a part of the real world, this part of the real world is the UoD. Using the entity concept, a more succinct description of the UoD is given as follow:

> *All entities of interest that have been, are, or ever might be.*

The conceptual schema is then defined relative to a UoD as:

> *The description of the possible states of affairs of the UoD, including the classifications, rules, laws, etc. of the UoD.*

Moreover, they give two general principles for the conceptual schema that at all times should be observed:

100 % PRINCIPLE:

> *All relevant general static and dynamic aspects, i.e. all rules, laws, etc., of the UoD should be described in the conceptual schema. The information system cannot be held responsible for not meeting those described elsewhere, including in particular those in application programs.*

CONCEPTUALISATION PRINCIPLE:

> *A conceptual schema should only include conceptually relevant aspects, both static and dynamic, of the UoD; thus excluding all aspects of (internal or external) data representation, physical data organisation and access, as well as all aspects of particular external user representation such as message formats, data structures, et cetera.*

Their following observation is that the definition of a conceptual schema can be made more precise using *propositions*. In their terminology, a proposition is a conceivable state of affairs concerning entities about which it is possible to assert or deny that such a state of affairs holds for those entities. A database reflects the UoD at a certain point in time, i.e. it reflects an *entity world* which is a possible collection of entities that are perceived together. A collection of propositions each of which holds for a given entity world is called a *proposition world*. For a given UoD, there is a universe of all possible entity worlds, each of which is described by a proposition world. Some propositions will be asserted to hold for all the entity worlds and are therefore part of all possible proposition worlds. Such propositions are called *necessary propositions*. The conceptual schema can now be seen as:

> *The representation of the consistent collection of necessary propositions that hold for a UoD.*

Using the definitions given above, the content of the database itself (called the information base in the aforementioned report) can be defined as well:

> *The representation of the collection of the propositions, other than the necessary propositions, that hold for a specific entity world. This collection should be consistent, both in itself and joined with the conceptual schema.*

Consequently the information base together with the conceptual schema describe all propositions relevant for a specific entity world; i.e. they form a proposition world for that entity world.

The following step is that we need a formalism in which both the conceptual schema and the information base can be described and manipulated. Such a formalism is often known as a database- or data-model. In other words, a data model is a formalism in which the conceptual schema and the information base

can be described.

The predominant database model used in practice at the time of writing is the relational database model proposed by Codd (Codd (1970)). In the next section we describe this model.

1.2 Relational database theory

In the previous section, we defined the kind of models that should be formulated in a data model. In this section, we give an overview of the currently most important data model, viz. the relational data model. This section is split in two subsections. The first subsection is a proper introduction to the relational data model; it covers the three main aspects, viz. structure, integrity, and manipulation.

In the second subsection, we show some of the results of relational database theory pertaining data modelling. Moreover, we show some deficiencies of the relational model in this area from a theoretical point of view. In the next section, we illustrate deficiencies from a practical point of view. As an aside, note that many of the concepts and problems that we introduce in this section will reappear later in this thesis.

1.2.1 The relational data model

The structure of a relational database is described by a *relational schema rs* over a set of attributes $A = \{A_1, \ldots, A_n\}$. The relational schema rs is a set of *relation schemas* $\{r_1, \ldots, r_n\}$. Each relation schema corresponds to an entity type or a relation type and is defined as a named subset of A; e.g. $r_i = \{A_{i,1}, \ldots, A_{i,m}\} \subseteq A$. The attributes A_i are associated with a *domain*, a set of values, denoted by D_i. This association can be extended to the relation schemas using the Cartesian product as follows:

$$D_{r_i} = \prod_{A_j \in r_i} D_j$$

The set D_{r_i} is often called the domain of r_i. A *relation* R_i over the schema r_i, also known as an extension of r_i, is a subset of D_{r_i}; an element of R_i is often called a *tuple*. Each tuple corresponds to an entity of the entity type corresponding with the relation schema.

Relations can be depicted graphically as tables as follows:

A_1	A_2	A_3
$v_{1,1}$	$v_{1,2}$	$v_{1,3}$
$v_{2,1}$	$v_{2,2}$	$v_{2,3}$
$v_{3,1}$	$v_{3,2}$	$v_{3,3}$

The first row ennumerates the attribute names of the relation schema r_i, while

the other rows are the tuples t_j in the relation R_i; each $v_{j,k}$ corresponds to the value of t_j for attribute A_k. The value of t_j for attribute A_k is often denoted by $t_j[A_k]$; more generally, if k is a subset of the attribute set of a relation schema r, and t is a tuple for r, $t[k]$ denotes the restriction of t to k.

As mentioned before, a tuple corresponds to an entity. Often not all information of an entity is known from the start. As an example, it is feasible that an employee is hired before it is known to which department she will be assigned. In such a case, *Null-values* are often used. In the literature, there are many different Null-values with different interpretations. The Null-value we are introducing should be interpreted as: 'information not yet known'. To introduce it formally: assume that each domain D_i has a distinguished element $Null_i$ which is interpreted as an unspecified value.

The second component of the relational model consists of integrity rules. Integrity rules are means to reduce the possible extensions of a set of relation schemas, and thus to make a more accurate representation of the real world. the basic integrity rules are entity integrity rule and the referential integrity rule. These two rules are general, they should hold for every relational database. Other integrity rules, or *constraints* have been defined, but for each relation schema, it should be specified which of these constraints have to be satisfied. This more general notion of integrity rule is introduced later in this section in more detail.

The entity integrity rule is based on the notion of a *primary key*. To introduce this notion, recall that a relation is defined as a set of tuples, i.e. all tuples in a relation are mutually different. In other words, a tuple is uniquely identified by its values. Often, a subset of the attributes can be specified which will already identify tuples uniquely. Such a set of attributes is called a *key* of the relation schema. More formally:
Let $r = \{A_1, \ldots, A_n\}$, then $k = \{A_1, \ldots, A_m\}$ is a key of r if for any relation R over r and for any two tuples $t_1, t_2 \in R$ we have:

$$(\forall A_j \in k\colon t_1[A_j] = t_2[A_j]) \textit{ iff } t_1 = t_2$$

Clearly, the keys have to be designated as such by the database designer. Moreover, a relation schema may have more than one key. In such a case, one of the keys has to be chosen as *primary key*; the others are known as *alternate keys*.

From the definition of a primary key, we see that it uniquely identifies a tuple in the relation. Each tuple in the relation corresponds to an entity. It would correspond *uniquely* with an entity iff all information about the primary key is known, i.e. if there are no Null-values in the primary key. This is the requirement made by the entity integrity rule:

no attribute of a primary key may accept a Null-value.

Let k be the primary key of a relation r then we know from the entity integrity rule that for each tuple t for r, $t[k]$ uniquely identifies an entity. Suppose, we have another relation schema s such that $k \subseteq s$. Moreover, let t_1 be a

tuple for r and t_2 a tuple for s. If $t_1[k] = t_2[k]$ we may conclude that t_2 specifies additional information on the entity uniquely identified by t_1. In such a case, $k \subseteq s$ can be seen as some kind of a *pointer* from s to r. If we take this pointer idea seriously, we have at least to ensure that for each tuple t_2 in the extension of s there is a tuple t_1 in the extension of r such that $t_1[k] = t_2[k]$. This pointer mechanism is formalised in the *referential integrity rule* using the notion of a *foreign key*. This is a set of attributes of one relation that is required to match those of a primary key in some other relation. The referential integrity rule now states:

> *either a foreign key is completely Null, or there exists a tuple in a second relation such that its primary key is exactly the foreign key of the tuple in the first relation.*

Note that this definition of the rule is not undisputed. A discussion of the relative merits of the various definitions is, however, beyond the scope of this short introduction; the interested reader is referred to e.g. (Date (1981a)). Summarising the integrity component of the relational model, we may say that the entity integrity rule ensures that the primary key serves to identify uniquely real world objects, foreign keys are used to denote relationships of this real world object, and the integrity rule states that we can only know relationships of real world objects we know.

The third component of the relational model consist of a manipulation language: *relational algebra*. The algebra consists a a set of operators which all take one or more relations as their input and return a relation as their output. A relation is defined as a set, hence one would expect the usual set operators in the relational algebra. And indeed, the traditional set operators *union*, *intersection*, *difference*, and *Cartesian product*, modified straightforwardly to handle relations instead of sets are part of the algebra. This set of traditional operators is extended with some special operators: *select*, *project*, and *join*; we describe these new operators briefly.

The first special operator is *select*. A select statement contains a *qualification*, and given a relation as input it returns *that* part of the relation that satisfies the *qualification*. The qualification is a Boolean combination of simple comparisons. In turn, a simple comparison uses a scalar comparison operator, such as $=$ or $<$, to compare either the value of two attributes of a tuple or the value of one attribute with a constant. An example of a qualification would be:

$$(t[A_1] < t[A_2]) \wedge t[A_3] = 45454$$

Clearly, the result of a select on a relation R over a relation schema r is a new relation R' over r; moreover $R' \subseteq R$.

Selection returns a horizontal 'slice' of the relation, contrasting, *projection* yields a vertical slice of the relation. Given a relation and a subset of the attributes of a relation, it returns a relation whose attributes are exactly the specified subset and whose extension consists of the set of all tuples in the original relation restricted to the specified attribute set. More formally:

Let r be a relation schema and $k \subseteq r$, for a relation R over r, the projection of r on k, denoted by $\pi_k(R)$ is defined as:

$$\pi_k(R) = \{t[k] \mid t \in R\}$$

Hence, the result of the projection on k is a *relation* over the relation schema k.

The third special operator is the *join*, which is not really a new operator, but a convenient expression for a concatenation of the previously defined operators. There are various different definitions of a join operator, the one we introduce is often called the *natural join*. The (natural) join takes two relations as input and outputs one new relation as follows:
Let R and S be relations over the relation schemas r respectively s:

1) Construct the relation $R \times S$
2) Select all tuples t in $R \times S$ such that $(\pi_R(t))[r \cap s] = (\pi_S(t))[r \cap s]$ in the relation X
3) Project X on $r \cup s$ into the relation $R \bowtie S$ over the relation schema $r \cup s$; this relation is the *join* of R and S.

As an aside, note that sometimes a fourth special operator, called *divide*, is defined. Like the join, it can be expressed in the other operators, however, unlike the join it is an operator that is scarcely used. Therefore, we do not define it, the interested reader is referred to e.g. (Date (1981b)).

From this introduction to the relational database model and the literature quoted, we can conclude that it is based on well-defined concepts (relations etc.) and it has well-defined operators. Moreover, the operators take relations as their input and they yield relations as their output; the relational model is closed†. This is a common situation in mathematics, e.g. a group consists of a set of elements, together with an operation, and the group is closed under the operation. As in mathematics, the closure property of the relational model enabled the development of a *theory* of relational databases. To illustrate, we sketch one of its first results, a theorem on query languages. Therefore, we introduce two query languages for the relational model. The first query language is based on the relational algebra, the second is called the *relational calculus*. As an aside, note that the query language defined for Flock later in this thesis is also a *calculus*. Hence, the introduction of the relational calculus in this section serves two purposes. First, it is used to present an example result of relational database theory. And secondly, more important, it simplifies a comparison between Flock and the relational data model.

To define the first query language and explain this notion, observe that the relational algebra is a way to create new tables from existing tables. We can make this statement a bit more precise as follows:

†This justifies the qualification *algebra* for the relational algebra.

Let R and S be relations over the relation schemas r respectively s. Moreover, let t be a relation schema such that $t \subseteq r \cup s$. Using R and S and the relational algebra operators we can construct relations T over t. The result of the construction depends on:

1) the combination of operators used,
2) the conditions used in the selections (if any) and
3) the attribute sets used in the projections.

To give an example consider e.g. the relational schema given by the relation schemas

$$r_1 = \{A, B, C\}, r_2 = \{B, C, D\}.$$

Two possible constructions for the relation schema $r_3 = \{A,D\}$ are:

1) $\pi_{AD}(r_1 \bowtie r_2)$;
2) $\pi_{AD}(\pi_{AC}(r_1)) \bowtie \pi_{CD}(r_2))$.

The application of these constructions to tables R_i for the r_i will result, in general, in different tables R_3 for r_3. Hence, the user may manipulate the relation schemas formally, and the result of this formal manipulation applied to actual relations yields a new relation that only depends on the *contents* of the input relations. Such a mechanism is known as a query language. From this discussion we see that the relational algebra forms a query language.

The relational algebra contains the usual set theoretic operations, except the *comprehension principle*. This comprehension principle is the heart of the *relational calculus*. More precisely, a relational calculus expression is an expression of the form:

$$\{t \mid \psi(t)\}$$

As usual, the result is a set of tuples (a relation), such that all the tuples in the set satisfy $\psi(t)$. The formula $\psi(t)$ has t as its only free variable and is built in the usual way using the connectors $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\exists$ and $\forall$ from the following types of atoms:

1) $R(s)$, where R is a relation name and s is a tuple variable, denoting s belongs to R.
2) $x[i]\,\theta\,y[j]$, where x and y are tuple variables, as above, $x[i]$ is the i^{th} component of x and similarly $y[j]$ is the j^{th} component of y and θ is a value comparison.
3) $x[i]\,\theta\,c$, where x is a tuple variable, c is a constant of the appropriate domain and θ is again a value comparison.

Finally, calculus queries have to be *safe*, i.e. the result of a query should always be finite; no matter what the actual content of the database is. To decide whether or not a calculus expression is safe is known to be a hard problem. This may be witnessed by the fact that there is no known syntactic restriction on the relational calculus which would produce exactly the safe queries. Of course, there are restrictions that produce only safe queries, e.g. if there is no $\neg$ in the formula, the query is safe. But all these restrictions disallow some safe queries, e.g. our example restriction disallows set difference, which is encoded by $\psi(t) = R(t) \wedge \neg S(t)$.

We have now defined two query languages. A natural question is now whether these two languages are equivalent in the sense that a query Q in one language may be translated in a query Q' in the other language such that Q and Q' yield the same result on the same input. One of the earliest results of relational database theory confirms this equivalence:

THEOREM 1.1
The safe relational calculus and the relational algebra are equivalent.

PROOF
see e.g. (Maier (1983)) □

It is far beyond the scope of this short introduction to give a detailed account of the theory of relational databases, the interested reader is referred to (Maier (1983); Ullman (1988); Ullman (1989)). However, in the next subsection, we present some results of the theory pertaining to data modelling, because this will reveal the weakness of the relational model and so justify the development of more advanced data models.

1.2.2 Database theory

A database is a collection of relations, one for each relation schema in the relational schema. The tuples in a relation correspond uniquely, by the entity integrity rule, to entities in the UoD. In this sense, the database can be seen as a abstract description of the UoD. Usually, not every possible database faithfully reflects a possible state of the UoD. In fact, we have already introduced the entity integrity rule and the referential integrity rule. It soon turned out that keys alone do not allow for an adequate description of a UoD as there may be more specific constraints in the UoD, e.g a *person* can have only one *age*. This gave rise to dependency theory, subsequently normalisation theory, and the more general question: 'what is a good relational database'. In a nutshell, the dependencies are defined as a mechanism to express integrity constraints, normalisation is a mechanism to resolve *anomalies* caused by dependencies, and the question of a good relational database arose when it turned out that normalisation was not always possible.

It is not completely true that normalisation was introduced to resolve anomalies caused by dependencies. The first normal form has nothing to do what so ever with anomalies, it simply states that all attribute domains have to be *atomic*. Atomic means that the elements of a domain are not supposed to have any structure such as e.g. a set. From the point of view of the relational model, this point of view is reasonable, for what is the use of a set as the value for an attribute if the relational algebra does not allow you to manipulate this value set-theoretically. In fact, from a mathematical point of view the first normal form is superfluous as it simply states that the smallest *objects* in relational database theory are abstract objects. In group theory, the elements of a group are also seen as abstract objects; i.e. their internal structure is not considered. However, the first normal form assumption or better the lack of manipulation power is one of the shortcomings of the relational model, as we will see later in this section.

The first dependency, the *functional dependency*, was already introduced by Codd (Codd (1972)). The definition of a functional dependency is a generalisation of the key. A key is a set of attributes such that the rest of the attributes are completely determined by the key. This is generalised by narrowing 'the rest of the attributes' to another set of attributes. In other words, a functional dependency $a \rightarrow b$, on sets of attributes a and b of a relation schema r, means that whenever two tuples of a relation R over r agree on a, they also agree on b. A functional dependency is full if the there is no $a' \subseteq a$ such that $a' \rightarrow b$ also holds, and it is called partial otherwise. So, each attribute in the relation schema is functionally dependent on the key. If functional dependencies are defined for a relations schema r, update anomalies may occur. For an easier description call an attribute that is in some key an *prime*-attribute, and a *non-prime* attribute otherwise:

1) Suppose that k is the primary key of r and that there exist an $a \in k$ and a $b \in r - k$ such that $a \rightarrow b$. So, the functional dependency $k \rightarrow b$ is only partial. This implies that a relation R over r will have redundant information, each time tuples agree on the a-value, they have to agree on the b-value also. Consequently, the modification of one tuple may cause the modification of a whole set of tuples. If, on the other hand, b would have been fully dependent on k this cascading could not happen as there would be no redundant information. Therefore, the second normal form requires that the relation schema is in first normal form and, moreover, that each non-key attribute is fully dependent on the primary key.

2) Suppose that k is the primary key of r and that a and b are both non-key attributes and thus fully dependent on k. Moreover, suppose we have the functional dependency $a \rightarrow b$; b is called *transitively* dependent on k, as we have $k \rightarrow a \rightarrow b$. Clearly, although a relation R over r will be in second normal form, it may still have redundant information as a certain combination of a and b values may occur more than once in the relation. As above,

this redundancy may cause cascading modifications. Therefore, the third normal form requires that a relation schema is not only in second normal form but also that no non-key attribute is transitively dependent on the primary key.

For each relation schema r in first normal form, there is a set of relation schemas $\{r_1, \ldots, r_n\}$, which are all in third normal form, such that

1) $r_i \subseteq r$,

2) let R be a relation over r, then $R = \pi_{r_1}(R) \bowtie \cdots \bowtie \pi_{r_n}(R)$.

Such a decomposition of r in the r_i is called lossless. The decomposition depends, of course, on the keys, but given a set of keys, some keys might imply others In general, a set of functional dependencies may logically imply other dependencies. Thus two sets of keys might be equivalent in the sense that one set implies the other and vice versa. Moreover, a set might contain redundant dependencies as an element might be implied already by the others. Obviously, this has repercussions on the decomposition of the relation. In 1974, Armstrong gave a sound and complete axiom set for functional dependencies (Armstrong (1974)). This axiom system can be used to derive all functional dependencies that are implied by a given set of dependencies. This *complete* set of functional dependencies can be derived from a number of its subsets. A subset that implies the complete set is called a *cover*. It should be obvious that a minimal cover is to be preferred for decomposition. However it is known that the computation of a minimal cover is NP-complete. Hence, a decomposition constructed with a minimal cover, into third normal form is a computationally expensive procedure.

Let $r = ABC$ be a relation schema, if either $A \rightarrow B$ or $A \rightarrow C$, r would be decomposed into $r_1 = AB$ and $r_2 = AC$ to achieve third normal form. Moreover, we know that for each relation R, $R = \pi_{AB}(R) \bowtie \pi_{AC}(R)$. A natural question is then if the reversed implication also holds, i.e., if:

$$(\forall R: R = \pi_{AB}(R) \bowtie \pi_{AC}(R)) \Rightarrow (A \rightarrow B \vee A \rightarrow C).$$

In 1977, Fagin showed this implication is false (Fagin (1977)). This situation became known as a *multivalued-dependency* (mvd) †. So, given a relation schema $r = ABC$, the multivalued dependency $A \rightarrow\!\!\rightarrow B$ (or equivalently $A \rightarrow\!\!\rightarrow C$ is defined as $(\forall R,\ R = \pi_{AB}(R) \bowtie \pi_{AC}(R))$. Note that this definition implies that the functional dependency $A \rightarrow B$ is a special case of the multivalued dependency $A \rightarrow\!\!\rightarrow B$.

The definition of a multivalued dependency forbids the relation

$$R = \{(a_1, b_1, c_1), (a_1, b_1, c_2), (a_1, b_2, c_1)\}$$

† In practice, multivalued dependencies are often used to model sets in a relational database. This trick has to be used as the first normal form explicitly forbids attributes to be sets.

as an extension of r. The smallest extension that satisfies the above mvd and is a superset of R is given by:

$$R' = \{(a_1, b_1, c_1), (a_1, b_1, c_2), (a_1, b_2, c_1), (a_1, b_2, c_2)\}$$

So, tuples may not be simply deleted from R'. Clearly, deletions become much simpler if we consider the two subrelations AB and AC of R that are indicated by the mvd. This is essentially what the fourth normal form requires us to do. A relation schema r is in fourth normal form if whenever a mvd $A \rightarrow\rightarrow B$ is defined on r, then all attributes in r are functionally dependent on A (note that fourth normal form implies third normal form). If there is only one multivalued dependency defined on a relation schema r, it is easy to decompose r in two relation schemas that are in fourth normal form: AB and AC are in fourth normal form.

If there is more than one mvd defined on a relation schema r, the situation is more complicated. The first step is similar as the first step for functional dependencies. Similar to fd's, mvd's may imply other mvd's, and again an axiom set is known. Moreover, there is an axiom set for sets of dependencies consisting of fd's and mvd's, i.e. all fd's and mvd's that are logically entailed by this set can be found using the axiom set; see (Beeri, Fagin, and Howard (1977)). Not surprisingly, the computation of a minimal cover for both functional dependencies and multivalued dependencies was also shown to be NP-complete. Much worse, however, is that there are sets of mvd's that resist a good decomposition. These sets suffer either from the *intersection anomaly* or the *split left-hand side anomaly* or both:

1) A full set of multivalued dependencies M is said to suffer from the intersection anomaly iff there are left-hand sides X and Y in M such that $(X \cap Y) \rightarrow\rightarrow (Dep(X) \cap Dep(Y))$ † is not derivable from M. The trouble of this anomaly is clearly that there is no unique relation containing the attributes in $(Dep(X) \cap Dep(Y))$.

2) A full set of multivalued dependencies M is said to suffer from the split left hand side anomaly iff there are two left hand sides X and Y in M, such that for distinct V and W in $Dep(X)$, both $V \cap Y \neq \emptyset$ and $W \cap Y \neq \emptyset$. During normalisation, the dependency having X as its lhs would split Y over various relations, while the dependency with Y as its lhs would keep the attributes of Y together.

A possible solution to this problem would be to extend the set of mvd's in such a way that the semantics are not seriously altered; note that these new mvd's cannot be entailed by the existing set. Some partial results of this approach are known in the literature, see (Beeri and Kifer (1986)). However, such solutions will always alter the semantics of the relation schema. Therefore, the existence

† $Dep(X)$ are all the multivalued dependencies with X as their lhs.

of these anomalies became one of the results that lead to the question: 'what is good database design?' However, before we turn our attention to results concerning this question, we first give some more results from dependency theory. If only, because these new results stressed the importance of the question on good database design.

The *join-dependency* is a logical generalisation of a multivalued dependency. A multivalued dependency states that a relation can be decomposed in two components without losing information. A natural extension is then of course the non-loss decomposition in more than two components. This dependency is known as the join dependency (jd) (Aho, Beeri, and Ullman (1979)). Given this new dependency, a new normal may be defined, the fifth normal form. Although this normal form exists, the situation is even worse as it was for multivalued dependencies, because there is no sound and complete axiom system for jd's. A sound set is known but its completeness is unlikely, see (Sciore (1982)). Moreover, the *fifth* normal form is again not always achievable.

Superficially, a join dependency appears to be a shorthand for a set of multivalued dependency. It is e.g., easy to see that for the relation schema $r = ABCD$, the join dependency $R = \pi_{AB}(R) \bowtie \pi_{BC}(R) \bowtie \pi_{CD}(R)$ is equivalent to the set $\{B \rightarrow\rightarrow A,\ C \rightarrow\rightarrow D\}$. So, it is a natural question to ask whether or not each jd is equivalent to a set of mvd's. The answer to this question is negative. In the proof of this fact, the *hypergraph*†. defined by a jd is used. For a jd $R = \pi_{r_1}(R) \bowtie \cdots \bowtie \pi_{r_n}$ on a relation schema r, this hypergraph is defined by:

1) The nodes are the attributes in r.

2) The sets of attributes r_i in the jd are the hyperedges.

If this hypergraph is acyclic (which is defined using a reduction schema) the join dependency is equivalent to a set of multivalued dependencies (Beeri *et al.* (1981)), otherwise it is not.

As mentioned above, there is no known axiom system for join dependencies. This is clearly a undesirable situation with respect to database modelling. When the design of such a system turned out to be a very hard problem, a number of researchers defined yet more general dependencies and tried to design axiom systems for such classes, in the hope that this could serve as a guide for the class of join dependencies. Independently, two papers proposed the most general class of dependencies, viz.: (Fagin (1982)) in which *embedded implicational dependencies* were defined and (Yannakakis and Papadimitriou (1981)) in which algebraic dependencies were defined. These dependencies form the largest class in the sense that it exactly represents the *domain-independent* dependencies; i.e. dependencies that remain valid under permutations of the domain. In the latter paper it is proved

† A hypergraph is a pair (N,E), where N is a finite set of nodes and $E \subseteq P(N)$ is a finite set of hyperedges.

that the two types of dependencies are equivalent, and moreover a complete axiomatisation for this class is given. As the algebraic dependencies are easier to understand, we introduce this class:
Let r be a relation schema with attributes $a(r) = \{A,B,\ldots,Z\}$, then $\bar{r}$ is a relation schema with $a(\bar{r}) = \{A_1,B_1,\ldots,Z_1,A_2,B_2,\ldots,Z_2,\cdots\}$. Moreover, each relation $\bar{R}$ over $\bar{r}$ is constructed from a relation R over r as follows:

$$\bar{R} = \{(t;t;\cdots)\,|\,t \in R\}.$$

So a tuple in $\bar{R}$ consists of an infinite number of copies of the *same* tuple t in R. An algebraic dependency over R is then defined as an expression:

$$\phi_1(\bar{R}) \subseteq \phi_2(\bar{R}),$$

in which the ϕ_i are project-join expressions.
An example of such a dependency is the well-known functional dependency $A \rightarrow B$ which is expressed algebraically as

$$\pi_{A_1B_1}(\bar{R}) \bowtie \pi_{A_1B_2}(\bar{R}) \subseteq \pi_{A_1B_1B_2}(\bar{R}).$$

The class of algebraic dependencies is rather large, so one is led to wonder if all these dependencies are useful in practice? So again we are facing the question: 'what database schemas are good?' Probably the first attempt to define good database schemas is the *Universal Relation* assumption. Simplified, this assumption states that attributes have a meaning of their own (disembodied of any particular relation) and thus have the same meaning in all relations they appear in. Moreover, the database schemas are considered to be projections of one Universal schema, in which null values are, of course, allowed. This assumption was not uncontroversial when it was first posed but became respectable after an influential paper of Mendelzon (Mendelzon (1984)). The most important aspect of the universal relation assumption for a user is that it is no longer necessary to know the relation schemas of the database. Instead, the user simply uses the attributes appropriate for the query without referring to the relation schemas the attributes belong to. This approach runs the risk of being ambiguous, as there may be more than one way to construct a relation schema r_m from given relation schemas $r_1,\ldots,r_n$. Therefore, such a *Universal Relation* has to be equipped with *window functions*; one for each set of attributes. The window function of a subset x of attributes contains the details of how a relation X over x should be constructed from the various relations in the relational schema, (Maier, Rozenshtein, and Warren (1986)). So, window functions give the UR assumption its full power. In a sense, the universal relation assumption can be seen as complementary to normalisation, normalisation *splits* relations to remove the possibility of update anomalies, while the universal relation is the *unification* of normalised relations to simplify querying. So, although the universal relation assumption makes a *good user-interface*, it does not tell us what good database design is.

Another view on good database design was put forward in an influential paper was written by Sciore (Sciore (1981)), in which the author argues that an 'every day' database only needs a set of functional dependencies and a single join dependency. Note that this thesis poses new problems for database theory, e.g. the problem when a set of jd's is equivalent to one jd; This question was posed by Paredaens in (Biskup (1987)). Circumstantial evidence for Sciore's thesis can be found in the fact that there is no natural example in the database literature that needs e.g. a set of multivalued dependencies that suffers from the split left-hand side anomaly. Moreover, it is well-known that acyclic schemas have desirable properties with regard to e.g. query processing. In fact, there are now four different types of acyclicity all with their own desirable properties, see (Fagin (1983)). So, although Sciore's thesis is pragmatic rather than based on theoretical insights, schemas satisfying the thesis are clearly desirable over those that don't. However, it is difficult to say how natural the requirements for the different forms of acyclicity are.

Summarising, we might say that the theory of relational database theory is impressive, but it cannot answer the question on good database design conclusively. In fact, the only way to resolve the viability of Sciore's thesis is by turning our attention to the field of *data modelling* or *information systems design*; the subject of the next section.

1.3 Information systems design

In the previous section, we gave an introduction to the relational data model. Moreover, we illustrated some deficiencies of this model from a theoretical point of view. In this section, we take a more practical view on database modelling. This switch in view point is made for two reasons. First, practical considerations may shed some light on the theoretical problems of the previous section. Secondly, real-world applications may highlight completely different shortcomings of this data model.

The area of *data modelling* or *information systems design* is much more an art than a theory. Mostly it is not built on a solid mathematical foundation, but rather encompasses the knowledge gained in designing *real world* databases. Therefore, it is reasonable to expect this area to 'proof' Sciore's thesis. One could say that this area has been founded by the the 'Entity Relationship' (ER) paper of Chen (Chen (1976)), and the 'Aggregation/Generalisation' paper of Smith and Smith (Smith and Smith (1977)). We describe both papers briefly.

Although not the first paper on data modelling, a fairly similar paper is (Schmid and Swenson (1975)), Chen's paper made data modelling known to the database community. In the ER model, the world is considered to exist of entities and relationships. Entities may be regular entities or weak entities, weak entities are existence dependent on some regular entities, whereas regular entities exist in their own right. Relationships are classified by their type (i.e. using weak and regular entities). Next, Chen defines a set of integrity rules (e.g.

limiting the number of times an entity can participate in a relationship) and informally defines a set of operators. An important aspect of this model is the deletion of an entity cascades to the deletion of corresponding relationships.

A weakness of the original proposal was that a relationship was not considered to be an entity, and consequently, a relationship can not participate in further relationships. However, after the original publication, it has gained considerable interest and a huge amount of extended and related data models have been proposed, most of which do not share this weakness; see e.g., (Chen (Ed.) (1980)). The popularity of the Entity-Relationship model and its derivatives is probably best illustrated by the fact that every commercial approach to database design uses one of these models. This is perhaps largely due to the graphical description of the UoD that the model derives, this facilitates the discussion between the database designer and the (future) user. Moreover, for several 'Entity-Relationship like' data models, methods have been developed to map the derived model of the UoD to a relational schema in a semi-automatic way. As far as can be concluded from the literature, none of the relational schemas that are derived in this way seem to suffer from the anomalies we discovered†. As the commercial database design methods are tailored towards business-type applications, as opposed to e.g. CAD-databases, we may conclude that Sciore's thesis is supported by practice.

In their paper (Smith and Smith (1977)), Smith and Smith introduce the concept of data abstraction. Abstraction means that certain details are deliberately suppressed; it allows the user to concentrate on relevant aspects and ignore the irrelevant ones. Smith and Smith distinguish two kinds of abstraction: *aggregation* and *generalisation*. Aggregation is the form of abstraction in which a relationship between objects is considered as a higher level object with the lower level details suppressed. As an example, an association can be regarded as a relationship among the participating entities (these participants are visible) or as an object in its own right (the participants are not visible).

Generalisation is the form of abstraction in which a set of similar objects is considered as a higher level object, again with the lower level details suppressed. In a sense, we can distinguish two kinds of generalisations:

1) A generalisation from a specific object of a certain type to the set of all objects of that type; the differences between the objects are then ignored. An example is the generalisation from a *specific employee* to the set of all *employees*.

2) A generalisation of objects of a different type to an object of a common (higher) type; in general, the properties of the common type are those properties that are common in all types. As an example, both engineers and secretaries are employees.

†This conclusion is an illustration of the so called *closed world assumption*, the author is unaware of publications claiming the contrary, and therefore assumes that the claim is true.

The first type of generalisation is in fact the basis for relational model; the abstraction is made each time a relation schema is introduced. The second type of generalisations are not directly supported; one may mimic such a generalisation by having foreign keys in both the *engineer* table and the *secretary* table that are derived from the *employee* table. However, such a solution is only partial. One could say that this paper was among the first to describe a *semantic* deficiency of the relational model.

In recent years, one sees the advent of so-called semantic data models in this area. The adjective 'semantic' does not imply that the older models did not involve semantics, it only implies that they capture more of the intended meaning of the user. In fact, we have already seen a paper (Potter and Trueblood (1988)) referring to 'Hyper-semantic data models', meaning models that capture even more meaning. This is of course a regrettable inflation of names . All the more, as the purpose of a data model is to describe the UoD as accurately as possible. However, as it is by now more or less a standard adjective we will adhere to standard terminology.

The basic problem with the Entity-Relationship like data models is that they provide a single construct to model relationships. This construct is to model a variety of relationships, varying from *parent-child* relationships in hierarchical databases to a *grades* relation between *students* and *courses*. It is obvious that using one and the same construction will result in the loss of the exact meaning of the relationship. Semantic data models are data models that provide more constructs for expressing different kinds of relationships. The first two such constructs, already introduced, are aggregation and generalisation as defined by Smith and Smith. Other examples are:

1) Classification, which allows specific instances to be considered as a higher level object.

2) Membership, which allows several object types to be combined into one higher level object type.

Besides abstraction, other mechanisms are discovered that are needed for an accurate description of a UoD. The first is the notion of *derived data*. Often, the value of one or more attributes can be calculated from the values of other attributes; e.g. the amount of taxes someone has to pay depends on her income and marital status. Clearly, a functional dependency would lose the information of how the relationship between the attributes may be calculated. Therefore, some semantic models allow such functions between, sets of, attributes to be modelled explicitly; see (Potter and Trueblood (1988)) for further references.

An extension of derived data is *inferred data*. The main difference is that now instead of attributes, complete extensions are constructed from given extensions. The *classical* example for inferred data seems to be the *ancestor* problem: given the *parent-child* relation and a rule explaining the relation between parents and

ancestors, infer the *ancestors*. This observation led some semantic data models to allow for the representation of such rules; see (Potter and Trueblood (1988)) for further references.

So far, the proposed extensions focussed only on the static aspects of the UoD. A further extension of the semantic data models is the incorporation of a description of the dynamic aspects of the UoD†. In ER-like models, the description of the UoD is still static. Of course, when such a static description of the UoD is developed, it is often complemented with a separate dynamic description known under such names as a *functional schema*. However, in recent years it is recognised that the static and dynamic description of the UoD are intertwined. This led to the development of semantic data models, in which both are integrated; see (Potter and Trueblood (1988)) for further references. This extension makes it possible to express *temporal constraints*. These are constraints expressing that certain events have to take place concurrently or sequentially. And in turn make it possible to abstract again, by abstracting such concurrent (or sequential) objects into a new higher level object.

It should be clear by now that it is not possible to translate these constructs faithfully to a relational database schema. If only because this data model has only one relation structure and can only be used to describe the static aspects of the UoD. To illustrate this deficiency further, note that the relational query languages are not powerful enough to compute the transitive closure of a relation (Aho and Ullman (1979)). This means that given a *parent-child* relation, the query languages are not sufficient to compute the *ancestor* relation in one query. So, the relational model cannot be used to model inferred data.

Recall, that we turned our attention to the area of data modelling in the hope to find evidence to support Sciore's thesis. However, after some initial support for this thesis, it quickly turned out that in the actual modelling phase one needs concepts that are not supported by the relational model. Moreover, the outcome of such a modelling phase can only be translated into a relational schema at the cost of the loss of a much of the information present in the design. So, we may conclude this section by observing that, the area of data modelling forces our attention back to database theory to look for data models that support the needs expressed in practice.

1.4 Object-oriented and deductive databases

To resolve the deficiencies of the relational data model, quite some research is invested in the definition of new, more powerful, data models. At the time of writing, two directions seem to prevail, viz. object-oriented databases and deductive databases; the latter are also known as knowledge bases. In the rest of this

†Note that in the ISO report, dynamic aspects of the UoD were required to be modelled explicitly. It is however only recent that this aspect is generally thought to belong to the database instead of the application programs.

section, we give a short description of both areas and compare the two areas.

1.4.1 Object oriented databases

Object-oriented databases can be seen as the more or less formal counterpart of semantic data models†. More or less, as there is as of yet no theory of object-oriented database systems. In fact, it is our aim that this thesis may prove to be a step in the development of such a theory. There are three main motivations for this approach:

1) the needs expressed in the advanced semantical data models,
2) new application areas, and
3) the impedance mismatch.

The first reason has been illustrated at length above. The second is closely related with the first. The need for database technology was first felt by those implementing business-type applications. Therefore, the attention of the database research community has been focussed on business type applications for a long time. However, new application areas such as CAD and office automation come forward with new requirements. To illustrate, in computer graphics one would like to store matrices and vectors such that they can be applied, searched, and manipulated. In CAD/CAM, one would like to store solid geometric objects such that the object can be translated, rotated, and transformed easily (Kemper and Wallrath (1987)).

For the third motivation, note that relational systems suffer from the impedance mismatch (Bancilhon (1988)): they are suited for ad hoc query mode, but not for application development. This stems largely from the fact that relational systems are *set-at-the-time*, whereas programming languages are *tuple-at-the-time*. Moreover, the computational models are quite different †. Object-oriented databases are sometimes seen as the merging of object-oriented programming languages and databases. Clearly, this would resolve the impedance mismatch.

Although there exists a steadily increasing amount of literature on object oriented databases, with subjects varying from formalisms for complex objects to actual implementations of object-oriented database management systems, it is still rather difficult to give a precise definition of an object-oriented DBMS. The main reason for this problem is the fact that there is little consensus on what constitutes an object-oriented system, let alone an object-oriented DBMS. However, considering the literature as indicative for the ingredients of such a system,

†In fact, the border between these two areas is rather fuzzy. Much of the research which we classify as 'object-oriented database research' is only concerned with of a formal definition of a complex object model. Broadly speaking, we classify formalisms as belonging to the area of object oriented databases and the more 'intuitive' models as semantic data models. As an example, in the next chapter, we treat IFO as an object-oriented data model, while it is defined as a formal semantical data model!

complex objects, inheritance, object identity and, perhaps to a lesser extend, the possibility to store both programs and data emerge as the main aspects of object-oriented databases. Note that these concepts are also among those classified mandatory in the recently published (Atkinson *et al.* (1989)). The authors of this Manifesto have identified more concepts as mandatory, however, those concepts are not relevant for the material presented in this thesis. These concepts we identified above can be described as follows:

Complex Objects
Simplified, complex objects can be likened to tuples of a relation that is not necessarily in first normal form. In the relational model, there is only one type constructor, viz. the Cartesian product; i.e. each relation schema represents the Cartesian product over its attributes. New application areas indicate that this single construction is not enough to capture the semantics of their data adequately. Therefore, object-oriented data bases offer a richer set of type constructors. This set often includes, in addition to the product, the disjoint union and a power type construction. Moreover, the formalisms often allow for more or less inductive type constructions using these operators. Elements of a thus constructed type are known as complex objects.

Inheritance
From a formalists point of view, inheritance is a reflection of the process of inductive type definition. From the point of view of information systems design, inheritance is the formalisation of ISA-relations. To illustrate, suppose that in the UoD we are describing we can distinguish *persons* and *employees*. Moreover, suppose that each *employee*-entity is also a *person*-entity. In data modelling terms, this is described by the ISA relation, *employee ISA person*. In the type structure of an object-oriented database it implies that *employees inherit* from *persons*. Inheritance means that all properties of the type *person* are also applicable to the type *employee*.

Object identity
Object identity captures the fact that although the representation of a real world entity in the database may change, the entity itself (Das Ding an sic) does not change. Moreover, the identity of an object distinguishes it from all other objects that were, are, or might become, part of the database. Rather than depicting a set of attributes as the unique identifier of an object, such as the key concept in the relational model, the identity is seen as a property that is distinct from all the other properties an object might have. Moreover, this identity is immutable for any action on the database.

Storing programs
The ability to store both data and programs not only ensures that the database describes both the dynamic and static aspects of the UoD as accurately as possible, but it also allows for *data-structure hiding* and software reuse. Often

classes are defined, which is a combination of a data type and a set of basic *access functions*. Objects of a certain class can only be manipulated through the basic access functions. This hides the particular type from the user and thus allows changes in the type structure without any effect on the application programs. In turn, this allows for highly modularised software. Objects representing dynamic aspects of the UoD are often called *dynamic objects* in this thesis.

1.4.2 Deductive databases

The first step toward knowledge bases is the logical reformulation of the relational data model. There are two options for such a reformulation, a database can be seen as a model and as a theory. In the area of knowledge bases, the former is predominant. An advantage of a logical reformulation is that it simplifies the definition and the semantics, of e.g. incomplete information. Incomplete information is a generalisation of Null-values. Null-values denote that a particular value is not yet known. Incomplete information denotes that the value of an attribute is not exactly known, but some information on the value is already known. As an example, one may know that the colour of a car is either *red* or *blue*. The formalisation of incomplete information in turn, allows for a sound definition of the notion of dependencies for databases that could contain incomplete information. Another advantage is that it is straightforward to generalise relational databases in such a reformulation. The extension studied within the area of knowledge base systems, is the addition of *rules* to database. A rule is a first order formula of the form:

$$p_1(\vec{x}) \wedge \cdots \wedge p_n(\vec{x}) \rightarrow q(\vec{x})$$

The advantage of allowing such rules is two-fold:

1) If such a rule is added to the database, the extension of q does not have to be stored as its contents can be inferred from the extensions of the p_i. Not only storage space is saved, but it is also ensured that the extension of q is consistent with the extensions of the p_i and the rule.

2) Allowing such rules in the query language enhances the expressiveness of the query language considerably. For example, the transitive closure of the binary relation $p(X, Y)$ is described by the following two rules:

$$p(X, Y) \rightarrow q(X, Y).$$

$$p(X, Z) \wedge q(Z, Y) \rightarrow q(X, Y).$$

It is well known that this query can not be formulated in the relational algebra.

Knowledge bases, in general, do not support the concept of an object identity, therefore this area may be classified as *value oriented* data models.

1.4.3 Object oriented databases versus Deductive databases

Research is invested in both object-oriented and deductive databases because both directions promise to resolve, some of, the deficiencies of the relational model. This causes some controversy on which is the best approach. In two successive years, this issue has been raised at the PODS conference, viz. by Ullman (Ullman (1987)) who argues for the value oriented direction, and by Bancilhon (Bancilhon (1988)) who argues for the object-oriented direction. Remarkable, both researchers more or less agree on the strong and weak points of both directions:

Ullman
Ullman sees as advantages of value-oriented systems that they encourage the use of declarative programming, the ease of composing queries as the type of input and output match, and finally that they do not restrict the use of the database to predefined access paths. He identifies three major drawbacks of value-oriented systems. First, their schemas are, like in the relational model, subject to anomalies. Secondly, because queries can use equality on arbitrary collections of fields, query optimisation is necessary. For query optimisation, the query language must be sufficiently stylized, which makes integration with a general-purpose language difficult. And finally that queries (and thus view definitions) return copies of the data, rather than the data itself, view updates are hard to define.

As strong points of the object-oriented systems, he identifies the object identity, the 'built-in' paths that have to be used by queries avoid the need of optimisation. Moreover, the same language can be used for all purposes and finally, view definitions return objects, so the semantics of updates is unambiguous. As weak points he identifies that declarative programming is hard to integrate with object-oriented systems. The fact that query composition is only possible if it is allowed by the types. And finally that querying is only allowed using the built-in paths.

As a result, he predicts that value oriented systems will eventually proof to be better than object-oriented systems for the same reasons that relational systems are better than hierarchic and network systems (in his view these systems were object-oriented!). Although, again, object-oriented systems will be first on the market as they are easier to implement.

Bancilhon
Bancilhon, sees as major advantages of object-oriented systems, that they support complex objects, object identity, extensibility, storing of both programs and data, and typing and inheritance, overriding and late binding. As prime drawbacks he notes the lack of a formal model and consequently the lack of simplicity, the lack of a declarative ad-hoc query language, and finally the lack of a relational interface, as most existing databases are relational. He predicts that object-oriented is the winning approach, but he identifies these drawbacks as major

research issues. Another important research issue he identifies is the need for a good formalism to handle sets. Moreover, he sees a contradiction in the data-hiding of encapsulation and the data request of queries.

A Comparison
As the two opponents predict a different winning approach, the reasons for calling a point strong differ greatly. For example, Ullman sees the object structure as a strong point, because it provides access to the data via 'built-in' paths and, thus, greatly reduces the need for query optimisation. Whereas for Bancilhon, complex objects allow for an intuitive representation of complex objects in a UoD. Although Ullman sees advantages in object structure, he identifies it also as the cause of limited access of the data, which he considers to be a major disadvantage.

1.5 A unified approach

Rather than joining one of the two camps, we observe the two approaches are concerned with more or less complementary deficiencies of the relational data model. Therefore, we believe that a better solution is found by providing the best of both worlds, i.e. by unifying the two approaches. Essentially, such a unification should combine the strong points of both approaches. However, the weak points of an approach are strongly related to its strong points. Therefore, we examine some of these weak points in some more detail. The first considered is the limited access offered in the object oriented approach, caused by the object structure. The second point is the lack of a declarative query language for object-oriented databases. The third point we briefly look into is the fact that only deductive databases allow the storage of knowledge. The fourth point is the storage of dynamic objects. An the final point is whether or not a unified approach should support object identity. After these discussions, we formulate the requirements a unified approach should satisfy.

Object Structure
The limited access to the data in the object-oriented approach, is caused primarily by the structure of the complex objects. This structure is used to model the semantics of the UoD explicitly. Through this, it protects the user from semantic errors. In this sense, it can be compared to the notion of typing in programming languages, which has also been a hot issue for quite some time. Reynolds, in his introductory section of (Reynolds (1984)) says: 'those in favour of untyped languages claim that typed languages preclude a variety of powerful programming techniques and are verbose to the point of unintelligibility; while the other side claims that untyped languages preclude compile time error checking, are prone to 'semantic' errors and are succinct to the point of unintelligibility'. He follows with the remark that: 'from the theorists point of view they are both right and their arguments are a motivation for seeking type systems that

are more flexible and succinct'.

To illustrate the semantic errors that users, but also database designers, may make, note that the following program, which groups states that property p holds for a set if it holds for all the members of the set:

$$p(\{X\}) \leftarrow p(X),$$

is a valid LDL-program! Of course, this program is rejected, but only because it is not stratified. But it is well-known that such expressions lead to the well-known paradoxes of naive Set Theory. These paradoxes in turn, were the stimulus for the development of Type Theory in Set Theory.

To illustrate the problems of fixed access paths to the data, recall that one of the main disadvantages of the hierarchical data model is that it presents a fixed view on the data and that queries expressing this fixed view are much easier to formulate than queries that express another view. Notorious examples are provided by many to many relationships, such as the ornithologist who keeps a database of which birds she has spotted in which area. By choosing the bird-records as children of the area-records, it is complicated to ask for the areas in which a specific bird has been spotted (Date (1981b)).

So, complex object structures can be used to represent the data intuitively and their typing mechanism precludes the user from making semantic errors. However, their fixed access paths may complicate queries unnecessarily. In line with Reynolds solution for typed programming languages, we should solve this problem by defining object structures with more flexible access paths, i.e. *flexible complex object structures*:

For the ornithologist database, the problem is that one is forced to choose between two *equivalent* representations. After a choice has been made, the representation is fixed. The solution of the relational model to this problem lies in a third, again equivalent, representation; namely a relation with *bird* and *area* as its attributes. The choice between these three representations is based on:

1) what is the most intuitive representation;

2 what representation is the simplest for most usage?

Clearly, a formalism which does not demand such a choice by providing all these equivalent representations at the same time, is easier to use and thus a valuable goal for research in data modelling. This observation is provides a motivation for the development of Flock.

Ad-hoc query languages

The second relative weak point of the object-oriented approach lies in the lack of a declarative ad-hoc query language. This is often attributed to the rigidity of its type structures. However, the structure does not have to preclude a declarative query language. Evidence for this claim may be found in the query languages defined for *LDM* (Kuper (1985)) which strongly resemble their relational

counterparts. Further support may be found in *The Calculus of Complex Objects* (Bancilhon and Khoshafian (1986)), in which rules are supported up to simple recursion. Moreover, FQL shows that a functional data model is highly suited to support declarative query languages. So, if the type structure is defined functionally, there seems to be no obstacle to define a declarative query language.

Knowledge
Thirdly, in deductive databases knowledge, or rules, can also be stored in the database. Ullman did not raise this point†, perhaps because databases that contain such rules are often called knowledge bases. However, it has always been of major importance in the database area to keep the representation of a UoD as small as possible. If only, because data redundance makes it more difficult to keep the database consistent. Moreover, such *stored queries* make the database easier accessible for the ad hoc user.

Dynamic Objects
Knowledge bases generally do not support the storage of the dynamic aspects of the UoD in the form of dynamic objects. One could argue that most of these aspects can be encoded in rules, but this implies that these aspects are no longer typed. Moreover, the stored data can be modified, while the rules can not be changed. In our opinion, one of the advantages of the object oriented approach is that the dynamic aspects of a UoD form an integral part of the database, and can thus be modified when appropriate.

Object Identity
Finally, a major difference seems to be that object identity plays a central role in the object-oriented approach, while this notion is almost inconsistent with the value oriented approach. So, if our unified approach supports objet identity, it may be categorised as an object-oriented data model rather than an unified data model. On the other hand, if we do not support object identity, our approach could be categorised as value oriented. Before we try to untangle this Gordian knot, note that, implementations of, the prototypical value oriented data model, i.e. the relational model support a sort of object identity in various disguises:

1) A foreign key can be seen as a formalised pointer, expressing that two tuples represent information about the same real world object.

2) If a relation R is logically decomposed in the relations during normalisation in say R_1 and R_2, i.e. R is the join of R_1 and R_2, the deletion of a tuple of R can be translated in various modifications of the tables R_1 and R_2. In fact, this problem is an instance of the more general view update problem. In various systems, such as system R, this is solved using the concept of a tuple identifier.

† Bancilhon compared the object-oriented approach with the relational data model.

3) The semantics of SQL is based on multi-sets rather than the set-semantics of the relational model. One reason for this deviation is that otherwise aggregate functions do not yield the expected result

One could say that this last disguise of object identity is caused by a mistake in the definition of SQL. However, the first two examples can not be brushed aside that easily. More in particular, the lack of convincing results on the view update problem seem to indicate that views do not really fit into the value oriented approach. So, again the Gordian knot is cut rather than untangled: a unified approach should support object identity. However, this support should not result in a non-declarative query language. Moreover, the identity should convey semantical information to the user. Often, object identity is seen as some machine generated number that is unique to the object and immutable during its lifetime (Khoshafian and Copeland (1986)). Such a notion of identity is usable in a formal setting. However, it is unintelligible for a user of the database. It can not be expected that a user knows the identity of any object in the database, i.e. from the users point of view the identity can not be used to distinguish between two objects that happen to have the same *value* at a certain moment in time. Moreover, it is not clear what should happen if during the lifetime of the database, two objects merge into one object or vice versa, one object 'splits' into two objects. In the former case, the problem is which of the two identities prevails, and in the latter the question is which of the two objects gets a new identity. In our opinion, object identity not only implies that the answer to a query is the object itself rather than a copy of its state, but it should also aid the user to retract the object she wants. Moreover, the notion of identity should allow for the representation of the merging and splitting of objects.

Requirements for unification

The discussions above leads to the conclusion that each unifying data model has to contain a formalism for complex objects such that:

1) The type system should support the use of *equivalent* types in the sense of the discussion above.

2) The type system should supports (multiple) inheritance both for static and dynamic objects.

3) The set of type constructors should at least contain the Cartesian product, the disjoint union and a power type construction. Moreover, it should contain a construction that allows for the modelling of the dynamic aspects of the UoD.

4) Power types imply that 'values' may be sets of less structured 'values', so clearly the formalism should allow for the set-theoretic manipulations of such set-values.

5) The formalism supports object identity as discussed above.

6) The formalism supports incomplete information.

7) The formalism supports the use of rules, both in the database and in queries.

8) The query language should be declarative.

1.6 The Thesis

Our aim in this dissertation is to develop a formalism for complex objects, called Flock, that satisfies these requirements. The definition of a formalism consists of two parts, viz. syntax and semantics. For the semantics, we have chosen an unconventional vehicle, namely Category Theory. The motivations for this choice are:

1) The theory is constructive; this implies that the semantics of Flock are adequate as a specification of an implementation.

2) The theory is fundamental; this means that we need only a few concepts, but these concepts are strong enough to develop, e.g. the Set Theory and the Logic Theory, we need in this thesis. As a result, we do not have to use an amalgamation of different mathematical theories to provide Flock with semantics.

3) The theory is graph-oriented; this gives a natural translation of our concept of a *type graph*. Moreover, the type constructors of Flock have a natural counterpart in Category Theory.

The drawback of the choice for Category Theory is reflected in the size of this thesis. As it is a relatively unknown field, we have to introduce many of its concepts and theorems along the way. However, we feel that the advantages outweigh the disadvantages.

The main results of this thesis are summarised by the following outline.

1.6.1 Chapter 2

Developing a new formalism implies that one is not satisfied with the already existing formalisms. To justify this opinion, the second chapter of this thesis is devoted to related work. The chapter starts with the object oriented direction. A difficulty encountered in this field is that there is no real consensus on what constitutes an object oriented database. Only recently, an influential group of researchers published the so-called Object-Oriented Manifesto (Atkinson *et al.* (1989)) in which they supply a list of requirements such a system should meet.

The direct consequence of this lack of consensus is that there exists a plethora of formalisms for complex objects. So, a complete overview is outside the scope of this thesis. Therefore, we have chosen five formalisms, which in our view

represent the major trends in this area. The five formalisms are:

1) The 'calculus for complex objects', defined by Bancilhon and Khoshafian (Bancilhon and Khoshafian (1986));
2) The IFO semantical database model, defined by Abiteboul and Hull (Abiteboul and Hull (1987); Abiteboul and Hull (1987));
3) The Logical Data model defined by Kuper (Kuper (1985));
4) The ψ-calculus defined by Ait-Kaci (Ait-Kaci (1984));
5) The O_2 data model from GIP Altair (Lecluse and Richard (1989))

The second section is a short introduction to the area of deductive databases. This area is much better understood and therefore, a shorter introduction is possible. In this section, two subjects are highlighted, viz. the logical reformulation of relational database theory and the, so-called, knowledge bases.

The first subject, which is introduced following (Reiter (1984)), reformulates relational database theory either as a model or a logical theory. This reformulation is done because the logical semantics provide a strong grip on problems such as incomplete information.

The second subject, knowledge base systems, is introduced by means of an example, viz. LDL (Naqvi and Tsur (1989)). There are two main motivations for this area:

1) The relational query languages are not Turing complete. Although this is on purpose, as this has proved instrumental in query optimisation, it is felt that the choice is too restrictive. Many reasonable queries, such as the transitive closure of a relation, are not expressible as a relational query. The desire for more expressive query languages which are still suitable for query optimisation is one of the motivations for knowledge base systems.
2) The other motivation is the incorporation of rules in a database. If data can be derived from other data in the database, only the rule to derive this data should be represented. The obvious reasons are that redundancy poses update problems and that redundancy increases the necessary memory. A nice side effect of storing the rules is that it enables reasoning over the data in the database.

Both motivations are highlighted in our description.

The third and final section is devoted to unifying proposals described in the literature. Just as we were not the first to discover the semantical inadequacy of the relational model, we were not the first to note the complementarity of object oriented and deductive databases. The actual unifying formalisms depend, of course, on the starting point of the researchers. One may start with a deductive database language and embed a formalism for complex objects. An example of

this approach is LDL with ψ-terms (Beeri, Nasr, and Tsur (1988)). Or one could unify an existing logic programming language and an existing formalism for complex objects on a basis of equality. An example of this approach is Log In (Ait-Kaci and Nasr (1986a)). Or, one could define a logical data language which has constructs for complex objects. An example of this approach is COL (Abiteboul and Grumbach (1987)). Finally, one could also start with a formalism for complex objects and extend this into a logical programming language. This is the approach used in O-logic (Kifer and Wu (1989)). As before, this selection of formalisms reflects the knowledge and the bias of the author. We give a brief overview of these models in chronological order.

Whereas the first two sections of this chapter can be seen as an introduction to the areas of object-oriented and deductive databases, the last section obviously describes formalisms that are closely related with Flock. Therefore, we evaluate these formalisms against our list of requirements. Not surprisingly, we find none of these formalisms adequate (the lack of surprise is of course due to the fact that this thesis is actually written). Moreover, we argue that none of these formalisms is particularly suitable to be extended to meet our requirements.

The first motivation to dismiss some of these formalisms is that we believe that the complex object should be the central concept; as we are interested in complex objects. The remaining formalisms already have a rather complicated semantics, this makes it awkward to add concepts as subtyping or type equivalences. If one succeeds, the resulting semantics will most probably be rather opaque.

1.6.2 Chapter 3

The first two chapters were introductory in nature, the third chapter finally, is the starting point of the development of Flock. This chapter starts with the introduction of the concept of a *type graph*. Usually, the types of a language are defined by a set of basic types and a set of type constructions. However, we make two observations on the relational model that inspire the notion of a type graph. These observations are:

1) The relational operators may be used ad libertum on the tables of a schema. However, it is not guaranteed that the relation schema of the result of such a manipulation describes anything remotely familiar in the UoD.

2) Even if the resulting schema is meaningful, there may be several different ways to construct a table over this schema. In general, the different constructions will lead to different result tables. This means that there is semantics hidden in the way the table is constructed. Some constructions may be meaningful, while others are senseless.

Now, a type graph can be likened to a description of the meaningful relation schemas and the meaningful conversions between these relation schemas. The main difference is that rather than relation schemas, we allow arbitrary types and that instead of relational constructions, we allow arbitrary functions for the conversion.

After this motivation for a type graph, we show how its directed graph nature can be used to define well known concepts such as specialisation and generalisation. Moreover, exploiting the functional nature of the directed arcs of the type graph, we define the notion of *equivalent* or *synonymous* types. These concepts are central to Flock, as can be seen by requirements Flock has to meet.

After the introduction of the concept of a type-graph, the introduction to Flock begins with the definition of its type structure. This is the subject of the rest of this chapter. As is obvious from the beginning of this chapter, the type structure of Flock will be a type graph.

Usually, in the definition of a language a set of basic types, such as *Bool*, are pre-defined. In the definition of Flock, however, we make an exception to this rule. This means that we do not define a basic type graph, but rather let the database designer define this basic type graph herself. The reason is that in Flock the basic types represent the 'smallest' observable types in a UoD. Clearly, these 'atoms' cannot be defined uniquely over all UoD's. Note that this does not imply that these basic types cannot be complex types in some other language, but that these complex types cannot be destructed within the formalism.

Flock is provided with six type constructors, viz.:

1) Products, this construction can be likened to the array construction in conventional languages.

2) Co-products, this construction can be likened to the disjoint union provided in some languages.

3) Exponents, this constructs a function type. It will proof to be instrumental in the definition of methods in Flock.

4) Equalisers, this construction does not have a counterpart in conventional languages. Its interpretation can best be sketched as a selection.

5) Power types, this is the usual construction that allows attributes to be tables in NF^2 data models.

6) Co-equalisers, this is perhaps the most difficult operator. Its main use lies in the fact that it allows us to construct *equivalence types*. Hence, it is instrumental to meet our requirements.

Note that these constructions do not only add nodes to the type-graph, but may also add new arcs.

All the constructions are introduced accompanied by their set-theoretic interpretation. These set-theoretic interpretations can be used as the informal

semantics of the constructions. To facilitate the step towards the formal semantics of the constructions, a universal property is proven for each of the constructors. Moreover, to introduce the reader to categorical reasoning, these universal properties are used to proof simple theorems. Most of these theorems pertain to type equivalence.

This chapter ends with a short introduction to category theory. Which is used to define the formal semantics of the type constructions of Flock.

1.6.3 Chapter 4

In the fourth chapter, we construct types whose elements are exactly the valid extensions of a relation schema over which a set of algebraic dependencies are defined. This exercise serves two purposes:

1) It illustrates the use of the constructions introduced in the third chapter.

2) It allows a comparison between 'relational semantics' and Flock-semantics

In the first part of this chapter, we show how the relational constructions *join* and *projection* can be defined within Flock. Using these constructions, the construction of the required types becomes straightforward. This means that Flock is at least as expressive as the relational data model with regard to the representation of real world semantics.

However, in Chapter 1, we have seen that dependencies may cause update anomalies. The equivalent Flock-types constructed in the first part are just as intuitive as their relational counterparts, i.e. although the elements of a type are sets of tuples, we cannot simply insert or delete a tuple and remain in the same type. Therefore, the remainder of this chapter is devoted to more intuitive translations. In other words, given a relation schema r with a set of dependencies D, we want to find a type t, such that:

1) The elements of t correspond to the extensions of r that satisfy D and vice versa.

2) The update semantics of t are natural.

The latter requirement is of course, rather vague. In this chapter, we have reformulated it as stating that the elements of t should consist of independent components. Independence roughly means that a modification of one component does not affect any of the other components.

This notion of independence suggests that the horizontal decompositions of relations over r satisfying D can be used to find such a natural translation. In the second part of this chapter, we show that indeed, horizontal decomposition leads to such a natural translation under the assumption that we have a natural translation for functional dependencies and provided that D is restricted to conflict-free sets of multivalued dependencies.

The latter requirement shows that our requirement for natural translations coincides with the requirements for normalisation; provided we find a natural translation for functional dependencies. The functional dependencies reappear in Chapter 10.

Clearly, different criteria for natural update semantics will lead to different classes of permitted dependencies. A natural candidate would seem to be to allow both horizontal and vertical decompositions. We end this chapter by a short introduction in the theory of horizontal and vertical decompositions into independent components.

1.6.4 Chapter 5

In Chapter 3, we have shown how a type graph can be defined. In this chapter, we take the next step towards the definition of complex objects, viz. we define structure types and structured objects. The structured objects are complex objects for which we have subtyping and type-equivalence. Both concepts are defined using the terminology for type graphs developed in the first part of Chapter 3. More specifically, subtyping is defined using the specialisation/generalisation hierarchy, while type equivalence is defined using synonymity. Roughly spoken, structure types are defined as subgraphs of the type graph, while structured objects are elements of such a type.

The remainder of this chapter is devoted to the definition and study of subtyping and type equivalences. Again, roughly spoken, subtyping is defined using a subgraph relationship, i.e. $s < t$ if the graph of t is a subgraph of the graph of s. The ratio behind this choice is that each object of type s can be transformed into an object of type t by 'forgetting' certain nodes and edges.

This definition of subtyping implies that many structure types are in a subtyping relation because of the way in which the type graph is constructed. In a theorem, we exhibit some general subtyping rules, induced by the type constructions of Flock. To compare our notion of subtyping, we reformulate the famous subtyping rules of Cardelli (Cardelli (1988)), in Flock. It is then an easy corollary that each of these rules is induced by a type construction. In other words, Cardelli's rules are implied by our definition of subtyping.

Type equivalence is simply defined using isomorphisms or synonymous types. There are two cases of type equivalence: either the database designer explicitly gives the isomorphism or two types are equivalent by construction. The latter case is interesting from a theoretical point of view. Moreover it greatly enhances the 'flexibility' of the system. Some examples of types that are equivalent construction are already given in Chapter 3. In this chapter, we provide some more examples.

In the ideal situation, the system can decide whether two types are equivalent by construction or not. In other words, one would like to have a sound and complete axiom system that can be used to proof equivalence. The last part of this chapter is devoted to this, well-known, problem. It turned out to be an extremely difficult problem. Therefore, we cannot give a complete axiom system.

Instead we give axiom systems for special cases in which the number of type constructions is limited.

1.6.5 Chapter 6

Structured objects and structure types as defined in the previous chapter are not particularly adequate to represent methods and method types. In fact, it is well known that anti-monotony in Cardelli's typing rule for exponential types 'blocks' the straightforward solution for method inheritance. As our subtyping rules entail Cardelli's rules, we cannot use this straightforward solution either.

So, the inheritance of methods requires that we find another way of looking at subtyping. As this is a semantical problem, the reader will not be surprised that the solution towards this problem is found using Category Theory. More in particular, we show that our definition of specialisation can still be used, but we have to define our semantics in a different category; for category theoreticians, the alternative semantics are placed in $C^{\rightarrow}$ rather than in C. Moreover, we show that method types can be seen as the usual graph-structure in these alternative semantics.

The object-oriented ideal is that methods are inherited automatically. In other words, one would like that methods defined for some type s are automatically extended to all subtypes t of s. It is shown that in general this is in general impossible. Consider e.g. the types *employee* and *manager*, such that *manager* is a subtype of *employee*. Let *promote* be a method that promotes an *employee* to a *manager*. As managers are *employees*, this method should also be applicable to *managers*, however, it is not a priori clear what the result-type of this application should be; let alone what the result itself should be.

However, we show that in restricted cases automatic inheritance is possible. The restrictions are:

1) Only products are used in the type constructions.

2) The methods preserve the type of the objects.

In this case, a method can be inherited by assuming it acts as the identity on other components. To continue our example, suppose that a manager is defined as an employee with a budget, i.e. the type is *employee* $\times$ *budget*. The method *raise-salary* is inherited as the function *raise-salary* $\times$ id_{budget}.

Implicit in the discussion above, is the assumption that the structure types are known at the time that the methods and their types are defined. Therefore, we use the term 'second-order' types when we formalise the description above in the last part of this chapter. This formalisation is rather complicated as we have to project our alternative semantics back to the usual semantics. The reason for this projection is that methods should act on structured objects; hence, these concepts should be defined in the same category. The last part of the formalisation is devoted to the semantics of the application of methods on structured objects

1.6.6 Chapter 7

Now that we have defined structured objects, methods and the application of structured objects, it is worthwhile to define a query language for structured objects. To give semantics to the query language, however, we need set-theoretic manipulations. Therefore, we need the intermediate Chapter 7. This chapter develops Logic and Set Theory in a categorical framework. All the material in this chapter is well-known in the realm of Category Theory. In other words, we do not present any new material in this chapter. Rather we introduce those concepts that we need in the remainder of this thesis.

The main points the reader should have learned after reading this chapter or, indeed, may take for granted, are:

1) The logical operations can be embedded naturely in Category Theory. Although the logic does not have to be classical. In particular, we show that the law of the excluded middle does not have to hold.

2) Using the logic, Set Theory can be developed in a categorical framework. Clearly, if the logic is non-classical, neither is the Set Theory.

These main points imply that we can incorporate set-theoretic and Logical manipulations in the syntax of Flock.

1.6.7 Chapter 8

Using the results of the last chapter, we define a query language, called SOQL, for databases consisting of structured objects. We start developing a query language before the definition of complex objects for two reasons:

1) As we have less concepts to deal with, the semantics of this query language are easier to define than those of a full-fledged query language.

2) Complex objects are built modularly from structured objects, similarly the query language for complex objects can be built modularly from SOQL

SOQL is a calculus defined along the usual lines. The highlights of this calculus are:

1) It is a typed language.

2) Set values can be manipulated using the set-theoretic operations.

3) Queries may be recursive, provided they are stratified.

As an aside, note that we do not support a *grouping*-operator in the query language as some other formalisms do. The reason for this omission is simply that the equivalence relation on classes (the types of complex objects), as defined

in Chapter 10, automatically entails grouping.

The semantics of non-recursive queries are defined along the usual lines. For recursive queries, however, we resort to a trick. Rather than developing the formal semantics of fixed points or unification within our categorical framework, we resort to an operational semantics given in (Abiteboul and Beeri (1988)). The motivation for these operational semantics is that a formal definition would require yet another extensive subset of Category theory.

This chapter is finished with a short note on how the semantics of recursive queries can be exploited to store rules in the database. The drawback of this solution is that the rules have to be invoked explicitly in a query. The solution to this problem can be given as soon as we have an object identity.

1.6.8 Chapter 9

In the real world, it often happens that one wants to store some information, although it is not yet complete. In database terminology, such information is known as incomplete information. The formalism we developed so far in this thesis cannot be used to represent incomplete information. Therefore, we develop the concept of *is*-objects and *is*-types from structured objects and structure types in this chapter.

Naively spoken, one could use a Set-theoretic encoding of incomplete information: simply group all the possible values in a set, and use this set as the value. Although this model is conceptually sound, it has some drawbacks if domains are large, let alone if they are infinite. Therefore, we use functions rather than sets, however, these functions can be seen as abstract representations of sets.

This functional representation is close to the representation of methods. So, one might expect that incompleteness causes similar problems as methods did. However, we show that a careful definition of *is*-types ensures that such problems do not appear.

Incomplete information puts an extra burden on the definition of methods. For example suppose that we have a method that assigns a new colour to cars and suppose that all colours are mapped to red, except red, which is mapped to blue. Now suppose that we have a car of which we know that it is not red. What colour does it have after the method has been applied? Clearly, we want the answer to be red rather than not blue. This means that the method should 'know' how to deal with incompleteness.

Finally, we extend the query language SOQL such that it can query databases consisting of *is*-objects. For structured objects, there were two possibilities, either an object belongs to the result of a query or it does not. For *is*-objects, there is the possibility of doubt. To illustrate this doubt, suppose that we know that a car is either red or green. Moreover, suppose that we query for cars that are either red or blue. Does our car belong to the result or not? To enable the user to specify exactly what she considers an answer, the query language is extended such that logical manipulations of the incompleteness is possible.

1.6.9 Chapter 10

In this chapter, we will merge the various threads of this thesis, such as dependencies, type equivalences, methods, objects, and queries. The result of this merging is the definition of *classes* and, finally, *complex objects*. Before we can begin this merging process, we first need to introduce the concept of an *object identity*; one of the main characteristics of object oriented database systems (Atkinson *et al.* (1989)), and the last requirement left on our list.

Usually, object identity is seen as a surrogate, i.e. a machine defined number, that is immutable to changes. In the first section, we give another definition of object identity based on the history of an object. The result is that object identity is defined as a set of expressions over the methods defined with the database. So, for the definition of methods, we assumed that the set of types was fixed, now we assume that the set of methods is fixed. Special attention is given to the question of how methods should be applied to such identities. The result of this analysis is yet again, that the method definition should be extended such that it also encodes this information. Moreover, we show how this definition of identity supports in a natural way the modelling of object sharing and *HAS-A*-relationships in general.

In the second subsection, we define constraints on object identities, by defining grammars the expression in an identity should satisfy. Moreover, we show how such constraints can be used to model both structural and temporal data dependencies in the database in a natural way.

In the fourth section, we extend the definitions of the third section to classes and complex objects. The construction of classes from identity types is similar to the construction of structure types from entity types. This implies that again type equivalence plays a major role in this definition. The grammars used to restrict the possible identities of a type, play a crucial role in these equivalences. Different from entity types, it is easy to show that the question whether two (restricted) entity types are equivalent is undecidable. This means that we can only offer some examples.

The examples are based on a grammatical formulation of the functional dependency. This is achieved by observing that a functional dependency can be seen as a dynamical dependency. Note that this result together with the results of Chapter 4 mean that we can finally give a natural translation for relation schemas with sets of functional dependencies and conflict free sets of multivalued dependencies. The fact that we can finally model multivalued dependencies induce the examples: we simply use the inference axioms for multivalued dependencies.

The lack of a formal semantics for classes means that we cannot give formal semantics to their query language. So rather than defining a new query language in the last section of this chapter, we simply illustrate four aspects of such a query language, viz. value oriented queries, HAS-A relations, grammars and historical data.

1.6.10 Chapter 11

In the last chapter of this thesis, we summarise the main results from this thesis and evaluate Flock against our list of requirements. Moreover, we discuss some directions for future research.

Chapter 2

Related formalisms

In Chapter 1, we have seen that there are three major threads in research in new database models, viz. the object-oriented databases, deductive databases, and the unified approach; this last approach tries to unify object-oriented and deductive databases. In this chapter, we give a brief overview of these three approaches as a reference for Flock. As there are three threads, there are three main sections.

The first section is devoted to the object-oriented database approach. In the previous chapter, we already mentioned that the consensus on what an object oriented database system should be is only slowly growing. Therefore, there are many rivaling proposals, especially for formalisms for complex objects. In this section, we give an overview of some of them.

Deductive databases are the topic of the second section. In this area there is much more consensus. Therefore, we limit ourselves to the description of just one of the systems. Moreover, we limit this description its main features. This means that we do not discuss the important topic of recursive query optimisation, which forms prehaps the body of the research performed in this area. The reason for this limitation is simply that the topic of this thesis is complex objects rather than recursive query optimisation.

In the third section, we give an overview of research on a formalism for a unified model. Clearly, the lack of consensus on a complex object model reflects directly in the diversity of proposals in the unified area. So, as in Section 2.1, we give an overview of some of the proposals in this area.

In the fourth and final section we formulate our conclusions.

2.1 Complex object formalisms

As mentioned in the introduction to this chapter, there are many formalisms for complex objects proposed in the literature. Moreover, the differences between the various proposals is considerable. The main reason for this diversity of proposals is probably the only slowly growing consensus on the definition of an object-oriented database system. This lack of consensus makes it difficult to give an overview of this area. Because the fact that the various proposals differ considerably implies that we cannot simply present one such formalism as prototype of this field. On the other hand, it is not feasible to present all of the proposals. Therefore, we present a selection. Obviously, any such selection reflects both the knowledge and the bias of the author, however, we believe that the following list covers the most important directions in the proposals:

1) The 'calculus for complex objects' defined by Bancilhon and Khoshafian (Bancilhon and Khoshafian (1986));
2) The IFO semantical database model defined by Abiteboul and Hull (Abiteboul and Hull (1987));
3) The Logical Data model defined by Kuper (Kuper (1985));
4) The ψ-calculus defined by Ait-Kaci (Ait-Kaci (1984));
5) The O_2 data model from GIP Altair (Lecluse and Richard (1989)).

The fact that IFO is added to this list is perhaps the strongest reflection of the bias of the author. This formalism is proposed as a semantical database model rather than a complex object model. However, in our view these two fields are closely related, particularly in the case of IFO.

2.1.1 A Calculus for Complex Objects

In (Bancilhon and Khoshafian (1986)), Bancilhon and Khoshafian define a general notion of objects. Such an object can be built from atomic objects by applying the *tuple* and the *set* constructor inductively to the set of atomic types. These atomic types are the standard integers, floats, strings, and booleans completed by the two special objects: $\bot$, called *Top* (the inconsistent object), and $\top$, called *Bottom* (the undefined object). For the tuple constructor, it is assumed that there is a countable set of *attribute names*, which can be recognised unambiguously from any other object in the system.
Given the objects $O_1, \ldots, O_n$ and the (distinct) attribute names $a_1, \ldots, a_n$, then:

1) $O = [a_1{:}O_1, \ldots, a_n{:}O_n]$ is a (tuple) object. Moreover, $O.a_i$ denotes the projection of O on the a_i attribute. If $a_i \notin \{a_1, \ldots, a_n\}$, then $O.a_i = \top$.

2) $O = \{O_1, \ldots, O_n\}$ is a (set) object.

Examples are [*name*:*peter*, *age*:25], {*peter*, *john*, *mary*} and [*name*:*peter*, *children*:{*max*, *susan*}].

Now that we have introduced the objects, we can proceed with the introduction of the calculus on these objects. Intuitively, this calculus can be used to combine the information represented by the objects. Technically, this is achieved by defining a lattice structure on the set of objects.

As usual, the first step towards the definition of this lattice structure is the definition of a partial order on the objects. The canonical choice for this order is obviously given by:

1) $O \leqslant O$

2) Every object is a subobject of $\bot$ and $\top$ is a subobject of every object.

3) If O_1 and O_2 are tuple objects, then $O_1 \leqslant O_2$, iff $O_1.a \leqslant O_2.a$ for all attributes a.

4) If O_1 and O_2 are set objects, then $O_1 \leqslant O_2$, iff every element of O_1 is a sub-object of an element of O_2.

However, this does not have to define a partial order. In fact, it is possible that two different objects represent the same information. For example, $\{O_1, O_2, \top\}$ represents the same information as $\{O_1, O_2\}$. Therefore, *reduced objects* are introduced using the $\leqslant$ relation defined above. The definition is again straightforward, the atomic objects are reduced, a tuple object O is reduced iff $O.a$ is reduced for all attributes a and a set O is reduced iff it does not contain two elements O_1 and O_2, such that $O_1 \leqslant O_2$.

It is easy to see that $\leqslant$ is a partial order on reduced objects. Moreover, if we define the union and intersection of (reduced) objects as indicated by the following examples, the reduced objects form a lattice:

$$[a:1, b:2] \cup [b:2, c:3] = [a:1, b:2, c:3];$$

$$\{1, 2\} \cap \{2, 3\} = \{2\};$$

$$[a:1, b:2] \cup \{2, 3\} = \bot.$$

The lattice structure of the 'object-world' can be used to define queries on the objects. As usual, this query language is defined using WELL-FORMED FORMULAE (wff). These wff's are introduced, much in the same way as the introduction of objects. So, variables and atomic objects are well-formed formulae, given the wff's $w_1, \ldots, w_n$ and the (distinct) attribute names $a_1, \ldots, a_n$, $[a_1:w_1, \ldots, a_n:w_n]$ is a wff, and $\{w_1, \ldots, w_n\}$ is a wff.

Given a wff E with variables $X_1, \ldots, X_n$, an assignment σ of objects $O_1, \ldots, O_n$ to $X_1, \ldots, X_n$ is denoted by $E(\sigma)$. The interpretation of E with respect to an object O is then defined as:

$$E(O) = \cup \{E(\sigma) \mid \sigma\colon E(\sigma) \leqslant O\}.$$

The query language defined above is rather limited, e.g. joins cannot be expressed in this language. The reason for this limitation is that each $E(\sigma)$ is a subobject of O, and the union of subobjects is again a subobject. So, $E(O)$ is a subobject of O. Therefore, the authors proceed to define *rules* in the calculus. Rules are given by pairs $<\psi, \phi>$ of wff's. Usually, such a rule is denoted by $\psi(X_1) :- \phi(X_2)$, in which X_1 and X_2 are vectors of variables. The idea of such a rule is that the tail selects while the head transforms. In other words, the effect of a rule r on an object O is then defined as:

$$r(O) = \cup \{\psi(\sigma) \mid \sigma\colon \phi(\sigma) \leqslant O\}$$

As an example, the rule:

$$[R:\{[A:X,\ D:Z]\}] :- [R_1:\{[A:X,\ B:Y]\},\ R_2:\{[C:Y,\ D:Z]\}]$$

joins the relations R_1 (over schema AB) and R_2 (over schema CD) on the condition that $B = C$ and assigns the result to R after projection on A and D.

The generalisation of the effect of one rule on an object to the effect of a set of rules on an object is defined using a closure. An object O is closed under a rule r iff $r(O) \leqslant O$. Given an object O and a set of rules R, there is a unique minimal object, $R^*(O)$, which contains O and is closed under R. The effect of a set of rules on an object is now defined as this unique minimal object. Note that sets of rules may be recursive, e.g. in a database of the form:

$$[\textit{family}:\{[\textit{name}: \cdots,\ \textit{children}:\{[\textit{name}: \cdots]\}]\}]$$

the set of descendents of Abraham is expressed by the set of rules:

$$[\textit{doa}:\{\textit{abraham}\}].$$

$$[\textit{doa}:\{X\}] :- [\textit{family}:\{[\textit{name}:Y,\ \textit{children}:\{[\textit{name}:X]\}]\}, \textit{doa}:\{Y\}].$$

In 1988, an extension of this model is defined by Bunemann, Davidson, and Waters (1988). In this paper, the notion of complex objects is extended such that it provides a denotation for incomplete tuples and partially described sets. Note that the incompleteness described in this paper is meant to enhance the query language, not to denote incompletely specified objects.

The incompleteness is formalised by 'sandwiching' set-values between *complete* and *consistent* descriptions. This is done by extending the definition of complex objects, especially set-objects are replaced by pairs of set-objects of the old definition; i.e. if $A_1, \ldots, A_n, B_1, \ldots, B_n$ are complex objects, then, subject to consistency restrictions given below, $(\{A_1, \ldots, A_n\}, \{B_1, \ldots, B_n\})$ is a complex object; objects of this form are called sandwiches. The first component is also called the *complete* constraint and the second component is also called the

consistent constraint. The ordering on complex objects is extended, to handle sandwiches, for $O = (A,B)$ and $O' = (A',B')$, $O \leqslant O'$ iff:

$$(\forall\, a \in A,\ \exists\, a' \in A' \colon a \leqslant a') \wedge (\forall\, b' \in B',\ \exists\, b \in B \colon b \leqslant b').$$

Thus the ordering on the first component is the ordering on set-objects as given above, the ordering on the second component is more or less its dual†.

The consistency criteria on the construction of sandwiches are that A and B should be reduced and there should be a set C, such that $\forall\, c \in C,\ \exists\, a \in A : a \leqslant c$ and $\forall\, b \in B,\ \exists\, c \in C : b \leqslant c$. So, consistency means that there exist objects between the sandwich-halves.

Concluding, we can say that the calculus of complex objects offers a formalism for complex objects that supports both incomplete information and a limited form of recursion. So, the reader might wonder why we the calculus is not classified as an unified formalism. However, we feel that both the recursion and the incomplete information are treated in a rather ad-hoc way. The recursion is only introduced by means of an example, while the incompleteness is only meant for queries.

From this introduction, it should be clear that the calculus does not satisfy our requirements. The most important point that is lacking is a clear definition of types. This means that if we want to extend the calculus to a formalism that meets our requirements, we have to start with the definition of types. In turn, this means that we have to begin at the very beginning. But then we might as well define a completely new formalism.

2.1.2 IFO

IFO (Abiteboul and Hull (1987)) is defined as a formal semantic data model, not as a complex object model. However, objects are one of the three main components of this model. Therefore, it seems reasonable to include this model in this survey.

IFO consists of three structural components. The first component is the representation of objects (as indicated above). The second component is used to represent(functional) relationships between objects. Objects with functional relationships are called fragments. The third component finally, can be used to model ISA relationships between fragments. We will discuss all three components briefly below. Note, that although (Abiteboul and Hull (1987)) gives a formal definition of IFO, we will only discuss it informally. The reason for this informality is that the formal description is rather complex. So a formal description that is intelligible to those who do not already know IFO would too large for this survey.

As indicated above, the first component of IFO can be used to define types. In IFO, as in the calculus of abstract objects, (complex) objects are defined

†The ordering on the first component is the Smyth power domain, while the ordering on the second component is the Hoare power domain.

recursively using constructors. An important difference, however, is that IFO uses the type concept. There are three kinds of atomic types. The first kind are the usual types, called the *printable* types. The set of printable types is denoted by P and contains e.g. strings and natural numbers. The second kind are *abstract* types, this set is denoted by A. An abstract type is a type that correspond to objects in the real world that have no underlying structure; although abstract types may have attributes. An example of such an abstract type is *person*. The final kind of types are the free types subtypes. The third atomic type are *free* types, this set is denoted by F. The difference between free types and abstract types is that the former is intuitively connected with an ISA-relationship. The distinction is made, as some rules in IFO need this distinction.

From these atomic types, new types may be defined inductively using the set and the tuple constructor. So, given objects $O_1, \ldots, O_n$, the tuple object $[O_1, \ldots, O_n]$ and the set object $\{O_1, \ldots, O_n\}$ can be constructed. IFO is a graph-oriented formalism, hence, these constructions can be rendered graphically; set types are denoted by $\odot$ and tuple types by $\otimes$. To depict the relation between a newly constructed type and its constituants, an arc is drawn from the new type to each of its constituants. So, if A denotes a type whose objects are sets of objects of type B, then we have an edge from A to B. It is assumed that each type has a unique root type.

With each object type we can associate a domain in the usual way. Usually, instances of a type are defined as elements of the domain of a type. In IFO, however, instances are defined as a finite subset of the domain. The elements of a domain are called objects.

The following component of IFO is called fragments. Fragments are built by defining functional relationships between object types. Such functional relationships are depicted in the graph by directed edges. If we want e.g. to model a functional relationship from *student* to *grade*, this is achieved by drawing an arrow from the student type to the grade type. Functions may not be defined between arbitrary nodes in types, but:

1) the head of the function edge is always the root of some type;

2) the tail of such an edge is either a root type, or the child of a root type provided this root type is a $\odot$ type.

Note that semantically these functions edges are not mapped to functions, but to partial functions. In other words, these edges can be used to define both real functions and functional dependencies.

The last construct of IFO are schemas. These schemas are built from fragments by defining specialisation and generalisation edges between fragments. Both kinds of edges are used to define ISA relationships. Intuitively, specialisation can be used to define possible roles of an object. For example, we might have a specialisation edge from *employee* to *person* to denote that each *employee* is a *person*; in this case, *student* is a role of *person*.

Generalisations are used to represent situations in which distinct types are combined to form new virtual types. For example, a *vehicle*-type may be built from a *car*- and a *boat*-type. Note, it is assumed that such a virtual type is completely covered by its subtypes.

These specialisation and generalisation edges can not be added at random to the IFO graph. Due to the semantics of these edges, some restrictions have to be satisfied. It is e.g. forbidden to have a cycle of edges in the graph.

Concluding, we can say that IFO is a rich model in that it supports a wide variety of constructs. Moreover, it has a sound formal basis which can be used to analyse IFO graphs. Clearly, IFO does not satisfy our requirements. Then again, it was not the purpose of the definition of IFO to define a unified formalism in our sense. In principle, we could try to extend IFO such that this extended version satisfies our requirements. However, the semantics of IFO as it is are already quite complicated; one of the reasons for this difficulty is the different semantics that have to be assigned to the different kinds of edges. Extending IFO would imply that the semantics of IFO would get even more complicated, a not very inviting prospect. A final weak point is in our eyes that no distinction is made in the relation of a $\odot$ node with its constituant and a $\otimes$ node with its constituants. However, mathematically, the relations between their domains is rather different.

2.1.3 The Logical Data model

As in the two models introduced above, LDM (Kuper (1985)) has constructs to create new types from existing ones. However, LDM differs in two aspects from these two models. First, it adds a layer of indirection at each application of a construct, e.g. each member of a product will be a tuple of pointers to other objects rather than the objects themselves. Secondly, it introduces the first calculus and algebra for complex objects which are essentially complete, in the sense of capturing the expressive power of the database logic query language. We will only survey the object constructions of LDM.

The LDM objects are built from the basic types (there is only one basic type in the paper, but that is no restriction) using the $\odot$, $\otimes$ and the $\oplus$ † constructors, which have the same semantics as in the previous models, with the addition that $\oplus$ denotes the disjoint union in LDM. The type constructions in LDM are also different in that complete schemas (which are graphs) are defined in one step. However, for all practical purposes we might as well assume that we construct the types first. As a schema is comparable with a type.

The types are then defined inductively, with the exception that for a $\oplus$ construction, all constituants have to be different. In the graph representation of the type, we have an arc from a type to its constituants. Note that this implies (similar to IFO) that no distinction is made in the relation of e.g. a $\oplus$ node with

† In the paper, pictorial symbols are used, but for consistency with the other models, we choose for the corresponding algebraic symbols.

its constituants and a $\otimes$ node with its constituants.

Given a schema S, instances have to be defined. As remarked above, LDM uses an indirection at every node, therefore an instance consists of two parts, an assignment of a set of *l-values* to every node (this is the pointer) and the assignment of an object called its *r-value* to each *l-value*. It is supposed that there exists a set A that supplies an infinite amount of 2l-values. Note that schemas may be cyclic, and hence instances cannot be constructed recursively as done in the two models above. Therefore, an instance is defined directly as a tuple $I = <l,r>$ such that:

1) l assigns disjoint sets of l-values to every node in S;

2) r is a function on the l-values in $\cup_{v \in V} l(v)$ consistent with the type of v. For example, if v has a $\otimes$-type with constituants v_i, then a $p \in l(v)$ is mapped to a tuple $(p_1, \ldots, p_n)$, such that $p_i \in l(v_i)$.

A finite instance is now an instance such that $l(v)$ is finite for every node v. The comparison of r-values is straightforward, but we should also be able to compare l-values , i.e. do they have the same r-value; this is done using similarity; two nodes are similar if they have the same type and have the same value

After the definition of schemas and their instances, the paper proceeds to define both a query algebra and a query calculus. Moreover, it is proved that just as in the relational model, the safe calculus and the algebra are equivalent. However, we will not introduce these languages for two reasons. First, the introduction would have to be technical rather than descriptive. And secondly, perhaps more important, we will not choose LDM as our vehicle.

The motivation for the rejection of LDM is as follows. Similar to the two earlier formalisms, LDM does not meet our requirements. So, if we decide to make LDM as our vehicle, we have to extend it. However, some of the choices made in the design of LDM seem to make such an extension rather awkward. First of all, we have seen that, similar to IFO, the relations between types and there constituants are always 'is-built from' relationships, while these constructions are mathematically completely different. Secondly, the indirections are an interesting idea of modelling object sharing, however, we believe that sharing should be optional. Finally, the semantics of LDM are rather complex, as can be witnessed by the fact that instances cannot be built inductively but have to be given at once. Extending LDM would complicate these semantics even further; a not very inviting prospect.

2.1.4 ψ-calculus

The motivation for the ψ-calculus (Ait-Kaci (1984)) is not so much to define a formalism for object-oriented databases, but to define a formalism for semantic networks. This can e.g. be seen from the fact that there is no set-mechanism in ψ-calculus. However, we give a quick overview of the calculus for three reasons:

1) it is a highly original approach to typing;

2) the advent of *knowledge*-base systems and *expert*-database systems shows that the 'gap' between artificial intelligence and databases is small;

3) the calculus has been used to unify the deductive and the object-oriented approach (Beeri, Nasr, and Tsur (1988)).

A broader introduction to the ψ-calculus can be found in (Ait-Kaci and Nasr (1986a)) and a more technical introduction can be found in (Ait-Kaci (1984); Ait-Kaci (1986)). The main difference between the ψ-calculus and the other formalisms is that it does not construct new types out of other types using constructors. Rather, it allows to define *subtypes* of given types by prescribing properties the objects in the subtype should have.
As an example, if the type *person* is defined, then

$$person\,(id \Rightarrow name\,(first \Rightarrow integer\,;\ last \Rightarrow string))$$

describes the subtype of all persons that have a name (a tuple) that consists of a first and a last name.

Tagging may be used to imply that certain entries have to be the same, e.g.

$$person\,(id \Rightarrow name\,(first \Rightarrow integer\,;\ last \Rightarrow X\colon string)\,;$$
$$father \Rightarrow person\,(id \Rightarrow name\,(last \Rightarrow X))).$$

describes the sub-type of *person*, who have a name as above, and a father who has the same last name.

Formally, the ψ terms are built using three kinds of symbols:

1) A signature Σ of type constructor symbols, which is a partially ordered set of symbols containing two special symbols *Top* and *Bottom*. Type symbols denote sets of objects, and their ordering denotes set inclusion. More specifically, *Top* denotes all possible objects and *Bottom* denotes the empty set. Every ψ term has at least one such symbol, called its root. Moreover, in the other formalisms we have seen, the *constants* where seen as part of a domain of a certain type. In ψ-calculus, they are seen as part of Σ, i.e. as a types.

2) A set A of attribute or access function symbols. These can be seen as record field labels. Each attribute denotes a function *in intenso* from the root type to the type of an associated sub-ψ-term. Concatenation of attributes symbols denotes function composition. Strings of A terms (elements of A^*) are called the ψ-term addresses. The *domain* of a ψ-term is the subset of A^* of the addresses of the ψ-term.

3) A set R of reference tag symbols. These tags are used as *coreference* constraints among paths of attribute symbols, which indicate that the corresponding function compositions denote the same function; i.e. that they

point to the same sub-ψ-term. Reference tags can be viewed as types variables. Hence they must consistently refer to the same structure. To avoid redundancy, the type of a reference tag has to be specified only once.

Using these three sets, a well-formed ψ-term (wft) can be defined as a triple $<\Delta,\kappa,\psi>$ such that:

1) Δ, the ψ-term domain is a regular set of finite strings from A^* closed under the prefix operation.

2) κ, the coreference relation, is an equivalence relation on Δ with a finite number of classes. When two addresses corefer, any pair of addresses in the domain obtained from them by further concatenation on the right must also corefer.

3) ψ is a type function extending the partial function on A^* defined from the coreference classes Δ/κ to Σ, by associating *Top* to all strings in A^* which are not in Δ.

An example of a wft is:

person (*id* $\Rightarrow$ *name* (*first* $\Rightarrow$ *integer* ; *last* $\Rightarrow$ X : *string*);

father $\Rightarrow$ *person* (*id* $\Rightarrow$ *name* (*last* $\Rightarrow$ X))).

The partial order on the type signature Σ is extended to the set of wft's in such a way as to reflect the set-inclusion interpretation. Thus t_1 is a subtype of t_2 if the root symbol of t_1 is a subtype in Σ of t_2. Furthermore, all the attribute labels of t_2 should also be attribute labels of t_1, and their wft's in t_1 should be subtypes of their corresponding wft's in t_2 and finally, all coreference constraints binding in t_2 must also be binding in t_1. Extending our example, if *student*$<$*person* in Σ, then the following wft is a subtype of the wft given above:

student (*id* $\Rightarrow$ *name* (*first* $\Rightarrow$ *integer* ; *last* $\Rightarrow$ X : *string*);

livesat $\Rightarrow$ *address* (*city* $\Rightarrow$ *cityname*);

father $\Rightarrow$ *person* (*id* $\Rightarrow$ *name* (*last* $\Rightarrow$ X))).

If the signature Σ is such that a greatest lower bound (lowest upper bound) exist for each pair of type symbols, with respect to the partial ordering on Σ, then the greatest lower bound (lowest upper bound) also exists for the extended wft ordering. In the unification algorithms for ψ-terms, the greatest lower-bound for types is used. For example, if in the type lattice, *teenager* is the *glb* of *adult* and *child*, then the ψ terms:

child (*knows* $\Rightarrow$ X : *person* (*knows* $\Rightarrow$ *queen* ; *hates* $\Rightarrow$ Y : *monarch*);

hates $\Rightarrow$ *child* (*knows* $\Rightarrow$ Y ; *likes* $\Rightarrow$ *wicked_queen*);

likes $\Rightarrow$ X).

$$adult\,(knows \Rightarrow adult\,(knows \Rightarrow witch);$$
$$hates \Rightarrow person\,(knows \Rightarrow X\colon monarch\,;\ likes \Rightarrow X))$$

have as their *glb*:

$$teenager\,(knows \Rightarrow X\colon adult\,(knows \Rightarrow wicked\_queen\,;\ hates \Rightarrow Y\colon wicked\_queen);$$
$$hates \Rightarrow child\,(knows \Rightarrow Y\,;\ likes \Rightarrow Y);$$
$$likes \Rightarrow X)$$

This unification is of course not possible if Σ is not a lower semi-lattice (i.e. if greatest lower bounds do not have to exist). This problem is solved, by enriching the ψ-term syntax with disjunctive ψ-terms. A disjunctive ψ-term is a set of incomparable ψ-terms, $\{t_1, \ldots, t_n\}$, where the t_i are (possibly disjunctive) ψ-terms. The subsumption ordering is extended such that:

$$D_1 < D_2 \Leftrightarrow (\forall t_1 \in D_1, \exists t_2 \in D_2\colon t_1 < t_2)$$

The formal semantics of ψ-terms is algebraic, based on lattice theory. The semantics of the calculus is operationally type structure rewriting, which can be formalised by a fixed point semantics. However, a survey of these aspects is beyond the scope of this chapter.

From the survey above, it is obvious that the ψ-calculus does not satisfy our requirements. However, in the third section of this chapter we will see that it can be extended to a formalism that is closer to these requirements. Still, we do not chose the ψ-calculus as a vehicle for our formalism. The reason for this rejection is that all constants are treated as types. This might be adequate for AI purposes (for which the ψ-calculus was defined), but not for database purposes. As the number of constants in a typical database application is rather large.

2.1.5 O_2

O_2, (Lecluse and Richard (1989)) is rather different from the other formalisms proposed in this section, as it is developed as a full-fledged database DBMS rather than a complex object formalism. It is not the only object-oriented database system in development other example are *Hybrid* (Nierstrasz (1985)) and *GemStone* (Maier *et al.* (1986)). However, it differs in that it is based on a well defined formalism. We only describe its complex objects model.

In O_2, there are two ways of structuring data. Types are recursively constructed using atomic types such as integers, floats, strings and class names and the set list and tuple constructor. Instances of types are called values. An example of a type is the following expression:

tuple (name: string,
country : string,
population : float
monuments : ***set***(Monument)***)***

A class describes the structure and the behaviour of a set of objects. The structural part of a class is a type as defined above. An example of the structural part of a class is given by the expression:

add class City
    type tuple (name: string,
        country : string,
        population : float
        monuments : ***set***(Monument))

Instances of a class are objects, they have an internal identifier and a value of the associated type. Objects are encapsulated, i.e. their value cannot be manipulated directly, but only by means of the *methods* associated with the class.

A method is a piece of code that is attached to a specific class, which can be applied to all objects in a class. An example of a method is given by:

add method increase-population (amount : integer)
 in class City
body

The body is defined in some programming language. It is interesting to note that O_2 is not a complete programming language. Rather it is a schema, in which the database programming language is filled in and a programming language may be embedded in the rest of the schema. This embedding yields the final language. An example of such a language is CO_2 in which the language C is embedded.

The classes in O_2 form a hierarchy, based on subtyping and inheritance. The subtyping, is a relation on the type components of classes. While inheritance is used to built the actual hierarchy. For subtyping, a Cardelli like (Cardelli (1988)) mechanism is adopted. We describe Cardelli's mechanism in more detail in Chapter 5, for the moment an example is suffices:

tuple (name: string,
    country : string,
    population : float
    airport : string
    monuments : ***set***(Monument))

is a subtype of the type

tuple (name: string,
    country : string,
    population : float
    monuments : ***set***(Monument))

Based on this subtyping, inheritance works as follows:

add class Big_city ***inherits*** City
type tuple (***airport*** : string)

Has the effect that the type of Big_city is given by:

tuple (name: string,
country : string,
population : float
airport : string
monuments : ***set***(Monument)***)***

The other effect of this definition is that each method associated with City is also applicable to any object in Big-city; although these methods may be defined differently (with the same name) for big cities. As an aside note that the example above is single inheritance but multiple inheritance is also allowed.

O_2 is a complete object-oriented DBMS. So, it covers more than we could cover in such a short survey. We already mentioned above that its complex object part is close to Cardelli's definition of multiple inheritance. In a later part of this thesis, we will show that Flock is a generalisation of this mechanism.

2.2 Deductive databases

The bond between logic and databases is an old one. Indeed the relational calculus relies on a first order language. The use of logic in the theory of databases is, however, far larger, see for instance (Gallaire and Minker (Eds.) (1978); Gallaire, Minker, and Nicolas (Eds.) (1981); Gallaire, Minker, and Nicolas (Eds.) (1984); Jacobs (1985); Minker (Ed.) (1988)). Broadly speaking, we can divide the area in two directions†:

1) The reformulation of the relational model (or even the hierarchical or network model) as a logical formalism to gain a better understanding of and to provide a solution for problems such as Null values, indefinite information, negative information, and view updates.

2) Knowledge bases, this is an extension of relational database (logically reformulated) so that knowledge in the form of rules can be stored in the database as well.

† Of course, this is gross generalisation as these two directions share many problems and solutions. However, it greatly simplifies that task of an informal introduction.

In the first two subsections, we give a short introduction to both directions. In last subsection we formulate some conclusions with regard to deductive databases.

2.2.1 Logical reformulations

There are two ways to describe a relational database in a logical formalism: as a model and as a theory, the description of both follows that given by Reiter (1984). In the model theoretic version, the relational schema is extended to a first order language with finitely many constants (the domains of the relations †), a distinguished equality predicate, and a distinguished set of unary predicates called the simple types (these are the attributes of the relational schema). The database (i.e. an extension of the relational schema) is then considered as a model for this first order language. Queries are answered relative to this model. Negation in queries is solved using *negation as failure*, or using the *closed world assumption*; the assumption being made is that if a positive literal cannot be proved, its negation is true. These two techniques are not equivalent (Sheperdson (1984)). Moreover, their basic assumption seems to be questionable. If only because it assumes that the information in the database is complete. This model theoretic version of relational databases is very close to Codd's original definition and in his article Reiter argues that it has the same deficiences, such as problems with Null values, and that a relational database should be regarded as a first order theory.

To turn the model in a first order theory, think of an interpretation I as being specified by a set of ground atomic formulae, and view this set as a first order theory T. Next, add to T the domain closure axiom, i.e. for an interpretation I with domain $c_1, \ldots, c_n$ add the axiom $(x)[=(x,c_1) \vee ,\ldots, \vee =(x,c_n)]$. Then add to T for any pair of distinct constants c_1 and c_2 the unique name axioms:$\neg=(c_1,c_2)$. To turn the equality predicate in the 'standard' equality, add equality axioms specifying the reflexivity, commutativity, and transitivity of equality and the principle of substitution of equal terms. Finally, add for each predicate the completion axiom which states which *tuples* of constants can be used in that predicate. The only model of T is now exactly I and instead of truth in the model, provability in the theory can be used.

Reiter shows that by suitable changes to the axioms added to T, such a theory can be used to represent Null values and disjunctive information. And that by adding more wff's such a theory can be used to represent events, ISA hierarchies, and aggregates.

† It is assumed that the domain elements of an interpretation are named using the constants of the language.

2.2.2 Knowledge bases

In the introduction, we have seen that the relational algebra cannot be used to compute the transitive closure of a relation. One way to repair this deficiency, while remaining declarative is to interface a relational dbms with a logic programming language such as Prolog; for a more extensive survey of query languages more expressive than the relational algebra see (Chandra (1988)). Moreover, the need arose, to store such rules in the database as well. These two reasons can be seen as the motivation for *knowledge* bases This is a large area, and we will not give a complete survey. Rather we give a description of LDL to show the general direction of this field (Naqvi and Tsur (1989)).

An LDL program consists of a set of (Horn) clauses, just as Prolog. A fact is a rule with an empty body. A traditional database is just a set of facts, the EDB (extensional database) in LDL. The EDB is extended with rules, the IDB (intensional database), these rules can be used to derive new facts. Simple LDL programs have only positive predicates and recursion. There are three (equivalent) semantics for such a program:

1) Declarative semantics; broadly speaking, this means: treat the program as a formula of a logic and define the notion of a minimal model for that formula.

2) Bottom up semantics; this can be paraphrased as: compute the minimal model of 1), using fix point constructions.

3) Top down evaluation; in this approach, the program is a formula of a logic. Take a query and constructively proof that it is a consequence of this formula; the substitutions used provide the answer to the query.

As it is possible that during the construction of a minimal model one of the intermediate models is infinite, the notion of safety is introduced (as in the theory of relational databases). Basically, a program is safe if every variable in the head of a rule also occurs in the body of that rule.

The first extension is the addition of evaluable functions and predicates. An evaluable function or predicate is realised if all its arguments are ground terms of the appropriate domain of the LDL Universe and a substitution for a formulae is admissible if it realises an evaluable function or predicate in that formulae. An example of such an evaluable function is the standard function *cons*, of type $elem \times list \rightarrow list$, which allows for the construction of lists in LDL. More general, evaluable functions may use $X \ @ \ Y$, for $X, Y \in N$ and $@ \in \{+, -, *, /\}$. Examples of evaluable predicates include $=$ and $\neq$, the use of equality and evaluable functions might result in infinite results. This will not occur if all variables occurring as an argument of $=$ or $\neq$ are covered with regard to $=$ respectively $\neq$; i.e. if the variables also occur in a non-evaluable function or predicate in the same body. Besides $=$ and $\neq$ all the normal relational comparisons are defined for ground terms.

If an LDL program contains negation, the program should be stratified (otherwise it would not admit a unique minimal model). For two predicates p and q define $p \geqslant q$ if there is a rule of the form $p \leftarrow ..., q, ...$ and $p > q$ if there is a rule $p \leftarrow ..., \neg q, ...$ If in the transitive closure of the so defined partial ordering $p > p$ occurs, the program does not admit stratification, otherwise it does. Using variables in negated predicates might again yield infinite models, thus these variables should be covered with regard to negation; i.e. such a variable should occur in the same body in a positive predicate.

LDL also admits sets, sets can be constructed in two ways, using *scons* and using grouping. The evaluable function *scons* of type $elem \times set-term \rightarrow set-term$ allows for the construction of enumerated sets. Grouping† takes the following form:

$$p(t_1, \ldots, t_n, \{Y\}) \leftarrow body(\bar{X}, Y).$$

Its meaning is as follows, let $\bar{Z}$ be all the variables occurring in the t_i and $\bar{X}$ all the variables occurring in the body except for Y. Construct a relation R by evaluating the body. Next, partition R horizontally for each distinct combination of values in $\bar{Z}$. Then group all the Y-values in each partition in a set. The derived relation p has a tuple for each distinct partition of the relation R; the first column of p has a distinct combination of the values of $t_1, \ldots, t_n$ and the second column has the corresponding grouped set. Again there might not be a unique minimal model, and therefore a program using grouping has to admit stratification. Which is defined by extending the partial ordering given above by $p > q$ if there is a rule $p(..., \{ \cdots \}) \leftarrow ..., q, ...$

We end our description of LDL by the remark that LDL is a real database language by admitting the insertion and deletion of predicates. This is done via update predicates, which are predicates of the form $+p(t_1, \ldots, t_n)$ or $-p(t_1, \ldots, t_n)$, where p is an n-ary base relation and the t_i are ground. Intuitively $+p(t)$ means insert t in p and $-p(t)$ means delete t from p. Using these basic constructs more interesting updates can be written.

2.2.3 Conclusions

In this section, we presented a short survey of the area of deductive databases. Obviously, there is much more to this subject then we could cover within the scope of this section. The main point the reader should have learned from this survey is this area is rather successful solving the deficiencies of the relational model they planned to solve.

Considering this success, one might be tempted to use a deductive data model as the starting point of a unified model. This temptation is strengthened by the observation that the *grouping* operator gives this area a limited notion of complex objects. This could mean that we would only have to have to extend this limited notion.

† Grouping is very similar to the NFNF *nest* operator.

However, in Chapter 1, we have already mentioned that one of the weak points of this area is that it uses an untyped formalism. In our opinion, the typing is central to the notion of complex objects. So, we need a typing schema before we can even think complex objects. As the goal of this thesis is the definition of a formalism for complex objects, this observation means that we should start with the definition of a typing schema before we do anything else. Hence, the deductive data models should not be used as a starting point.

Of course, if we want our formalism to meet our requirements, we should add the 'logic' features. At that point, we shall not attempt to reinvent the wheel, but incorporate the knowledge acquired in the field of deductive databases.

2.3 Unifying proposals

Both object-oriented databases and deductive databases solve deficiencies of the relational model. It should therefore not come as a surprise that there are proposals for formalisms that have the best of both worlds. In fact, one might be tempted to classify e.g. the calculus of complex objects already as such a unifying proposal. However, although recursive queries can be expressed in this formalism, such knowledge can not be seen as a part of the database. Moreover, we have already seen that the extend of the recursive queries is far from clear. For the similar reason LDM is not classified as a unifying proposal.

The actual unifying formalism depend of course on the starting point of the researchers. One may start with a deductive database language and embed a formalism for complex objects. An example of this approach is LDL with ψ-terms (Beeri, Nasr, and Tsur (1988)). Or one could unify an existing logic programming language and an existing formalism for complex objects on a basis of equality. An example of this approach is Log In (Ait-Kaci and Nasr (1986a)). Or, one could define a logical data language which has constructs for complex objects. An example of this approach is COL (Abiteboul and Grumbach (1987)). Finally, one could also start with a formalism for complex objects and extend this into a logical programming language. This is the approach used in O-logic (Kifer and Wu (1989)). As before, this selection of formalisms reflects the knowledge and the bias of the author. We give a brief overview of these models in chronological order.

2.3.1 COL

COL (Complex Object Language) is a logic programming approach to complex objects, based on recursive rewrite rules (Abiteboul and Grumbach (1987)). It is an extension of Datalog, with complex objects built using tuple and set constructors. More formal:

A set of atomic types is assumed, moreover, with each type t a *domain* is associated denoted by $dom(t)$. If $T_1, \ldots, T_n$ are types, then so are:

1) $[T_1, \ldots, T_n]$, a *tuple-type* with domain:

$$dom(T) = \{[a_1, \ldots, a_n] \mid \forall i,\ a_i \in dom(T_i)\};$$

2) $T = \{T_1, \ldots, T_n\}$, a *set-type* with domain

$$dom(T) = \{\{a_1, \ldots, a_m\} \mid m \geqslant 0,\ \forall i\ \exists j,\ a_i \in dom(T_j)\}.$$

The predicate $\in$ belongs to the language, and hence e.g. intersection can be defined by the following rule:

$$x \in \cap(X,Y) \leftarrow x \in X,\ x \in Y.$$

The language L of the underlying logic of COL is now based on a typed alphabet containing:

a) typed constants and variables;

b) $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\exists$, and $\forall$;

c) typed equality, $=_T$, and membership, $\in_{T,S}$, symbols;

d) type predicate symbols;

e) Typed function symbols of three kinds: data functions, tuple functions and set functions.

The tuple and set functions are used to denote the tuple and set constructs we introduced above. The data functions are set-valued functions.

From this alphabet, COL is defined as follows.
Constants and variables are terms, and if $t_1, \ldots, t_n$ are terms and F is an n-ary function symbol, then $F(t_1, \ldots, t_n)$ is a term. A term is called closed if it contains neither variables nor function symbols.
For an n-ary predicate symbol R and terms $t_1, \ldots, t_n$, $R(t_1, \ldots, t_n)$, $t_1 = t_2$ and $t_1 \in t_2$ are positive literals (if well-typed). And if ψ is a positive literal, then $\neg\psi$ is a negative literal. Atoms are literals of the form $R(t_1, \ldots, t_n)$ or $F(t_1, \ldots, t_n)$, if the $t_1, \ldots, t_n$ are closed, the atom is called closed.
Next, well-formed formulae are defined from the literals in the usual way. A *rule* is the defined as an expression of the form:

$$A \leftarrow L_1, \ldots, L_n.$$

In which the body, $L_1, \ldots, L_n$ is a conjunction of literals and the head A is an atom. Moreover, a program is defined as a finite set of rules.

The semantics of a COL program is defined using minimal models. Of course, a COL-program does not have to have a minimal model, e.g.

$$1 \in F,\ p(F),\ q(2)$$

$$q(1) \leftarrow p(\{1\})$$

has $\{1 \in F,\ p(\{1\}),\ q(1),\ q(2)\}$ and $\{1 \in F,\ 2 \in F,\ p(\{1,2\}),\ q(2)\}$ as incomparable minimal models. As usual, again, this problem is solved with a notion of *stratification*. Moreover, it is shown that these models can be computed as fixpoints.

From this short introduction, it is obvious that COL is a unified data model. However, it does not satisfy our requirements. It is clearly visible that Datalog has been the inspiration of COL. The main objective seem to have been to define complex objects staying as close to Datalog as possible. This is perhaps best visible in the typing schema. Types are introduced as predicates rather than as a distinct notion. This implies that typing is reduced to a run-time notion rather than a compile-time notion. Moreover, if we would add subtyping, each use of this mechanism would yield a set of inferences.

Rather than extending COL, we would prefer to ue a formalism in which typing is *the* primitive notion.

2.3.2 Log In

Log In (Ait-Kaci and Nasr (1986a)) is simply Prolog where first-order constructor terms have been replaced by ψ-terms. Its operational semantics is the immediate adaptation of Prolog's SLD-resolution. Elaborating, first order terms are translated to ψ-terms as follows:

$$f(t_1,\ldots,t_n) \Rightarrow f(1 \Rightarrow t_1, \ldots, n \Rightarrow t_n)$$

Secondly, instead of the ordinary unification used in Prolog, ψ-term unification is used. The only difference with the ψ-calculus is that the ψ-term unification has to be adapted to allow for undoing coreference merging and type coercion upon backtracking. Consider the following example program:

Let Σ be defined as follows:

$$\begin{aligned}
&\mathit{student} < \mathit{person}.\\
&\{\mathit{peter},\ \mathit{paul},\ \mathit{mary}\} < \mathit{student}.\\
&\{\mathit{goodgrade},\ \mathit{badgrade}\} < \mathit{grade}.\\
&\{a\ b\} < \mathit{goodgrade}.\\
&\{c\ ,\ d,\ f\} < \mathit{badgrade}.
\end{aligned}$$

Suppose, we have the following rules:

$$\begin{aligned}
&\mathit{likes}(X{:}\mathit{person},\ X).\\
&\mathit{likes}(\mathit{peter},\ \mathit{mary}).\\
&\mathit{likes}(\mathit{person},\ \mathit{goodthing}).
\end{aligned}$$

And the following facts:

$$\mathit{got}(\mathit{peter},\ c).$$

got(*paul*, *f*).
got(*mary*, *a*).

Finally, let the following rules explain when a person is happy:

happy(*X*:*person*):− *likes*(*X*, *Y*), *got*(*X*, *Y*).
happy(*X*:*person*):− *likes*(*X*, *Y*), *got*(*X*, *goodthing*).

Then the query ?− *happy*(*X*)., which is an abbreviation of ?− *happy*(*X*: ⊥)., unifies with the head of the first rule for *happy*, by coercing $X:\bot$ to *X*:*person*. This yields the new resolvant:

likes(*X*:*person*, *Y*), *got*(*X*, *Y*).

Next, *likes*(*X*:*person*, *Y*) unifies with the head of the first rule for *likes*, merging coreference $Y:\bot$ to *X*:*person*. This yields the resolvant:

got(*X*:*person*, *person*).

As there is neither such fact nor an applicable rule, the system should backtrack, and undo the coreference merging of $Y:\bot$ to *X*:*person*. The reader may care to simulate the system and find as answers to the query *mary* (twice) and *peter*.

In the paper, the authors only give an operational semantics of Log In, by actually giving the SLD-like algorithm. However, as it is an adaptation of the ψ-term unification, it is probably not too difficult to give a formal semantics in the line of the formal semantics of the ψ-calculus.

Log In, and thus ψ-calculus, is part of an even larger attempt to unify logic (called relational), functional, and object-oriented programming called LIFE. As we have already introduced part of this language, and it is the first such attempt the author is aware of, we include a brief overview of the complete language (Ait-Kaci and Lincoln (1988); Ait-Kaci and Lincoln (1988)).

The authors compare LIFE to a molecule. The atoms of the molecule are the λ-calculus (computing with functions), the π-calculus (computing with relations) and the ψ-calculus (computing with types). The bonds are the $\lambda\pi$-calculus (Le Fun), the $\pi\psi$-calculus (Log In) and the $\psi\lambda$-calculus (FOOL). The molecule emerging form this is LIFE (a $\lambda\pi\psi$-calculus). We describe these components below to give an overview of life.

The functional and the relational (= logic programming) components of life are straightforward. The functional component is a 'standard' functional programming language based on the λ-calculus. Standard implies that, although the λ-calculus is computationally complete, the functional component contains a built in set of constants, such as integer arithmetic constants and functions, boolean constants, equality on ground terms and notably a constant for conditional expressions such as *if* − *then* − *else*. Contrasting with some other functional languages, no reserved word (like *rec*) is used to explicitly denote recursion. The relational component is a logic programming language for which e.g. Prolog

can be used. We have already introduced the type component, i.e. the ψ-calculus.

The first bond, Log In, is described above, so we only have to describe the other two bonds. FOOL is simply a pattern-oriented functional language where (again) first order constructor terms have been relaced by ψ-terms, with type definition. Its operational semantics is immediate. The ψ-term subsumption ordering replaces the first-order matching ordering on constructor terms. In particular, disjunctive patterns may be used. The arbitrary richness of the user-defined partial ordering on types allows for highly generic functions.

Le Fun extends a logic programming language by generalising first order terms by inclusion of *applicative expressions* augmented with logical variables. The purpose is to allow *interpreted* functional expressions to participate as arguments in logical expressions†. We will give the informal syntax of Le Fun, as can be found in (Ait-Kaci and Nasr (1986b)). A Le Fun term is one of the following:

1) Variables, represented by capitalised identifiers;

2) Identifiers, represented starting with a lower case letter;

3) Abstractions, of the form $\lambda X.e$, X a variable and e a term;

4) Applications, of the form $e(e_1, \ldots, e_n)$, e and e_i terms.

As always in functional languages, left-associativity, infix notation and currying of applications are assumed. The special applications of the form $c(e_1, \ldots, e_n)$, where c is an identifier known to be a constructor symbol (see below) are called constructions.

A Le Fun program consists of a sequence of equations and clauses. An equation is of the form $f = - e$ where f is an identifier called an interpreted symbol and e is a term. If e is an abstraction of the form $\lambda X_1 \cdots X_n e'$, $f(X_1, \ldots, X_n) = e'$ may also be used. A clause is exactly a Prolog clause, with the exception that Le Fun terms are used, where first-order terms are used in Prolog. Such literals, which constitute Le Fun clauses, are called Le Fun literals.

The lexical distinction between constructor and interpreted symbols is simply that a constructor is any identifier that does not appear at the left-hand side of an equation. For those, fixed arity is assumed. Any construction with root constructor of arity n must have exactly n arguments. If it has more, it is ill-formed, if it has less, it is an abstraction. Indeed, if c is an *n-ary* constructor, $c(t_1, \ldots, t_k)$, $k<n$, means the term $\lambda X_1 \cdots X_{n-k}.c(e_1, \ldots, e_k, X_1, \ldots, X_{n-k})$, where the X_i do not occur free in the e_j. In a clause, the logical variables are those that are not in any λ scope. These are the only variables that can be instantiated by unification.

Finally, given a Le Fun program, a query is a sequence of Le Fun literals. If only a function evaluation is desired, a query of the form $X = f(e_1, \ldots, e_n)$

† We have met this idea already in LDL.

will provide the value of evaluating the given functional expression as X's binding.

The operational semantics of Le Fun uses *Residuations*, it may happen that a function application is not ready for reduction or that some expression components are still uninstantiated. In such a case, the unification is delayed until the operands are ready; i.e. until further variable instantiations make it possible to reduce unificands containing applicative expressions. Such a unification can be seen as a residual equation that has to be verified as opposed to solve, to confirm eventual success. In case of success, a residuation is simply discarded; if it fails it triggers backtracking.

Finally, we are nearing LIFE. A basic (i.e. not disjunctive) ψ-term structure expresses only typed equational constraints on objects. Now, with FOOL and Log In arbitrary functional and relational constraints on ψ-terms can be specified. In LIFE, a basic ψ-term denotes a functional application in FOOL's sense if its root symbol is a defined function. Thus a functional expression is either a ψ-term or a conjunction of ψ-terms denoted by $t_1 : \cdots : t_n$. This is how functional dependency constraints are expressed in a ψ-term in LIFE. Unifying such a ψ-term is as the standard unification of ψ-terms modulo residuation of functional expressions that cannot be unified yet. As for relational constraints on objects in LIFE, a ψ-term t may be followed by a *such that* clause consisting of the logical conjunction of literals $l_1, \ldots, l_n$. This is written as $t \mid l_1, \ldots, l_n$. Unification of such relationally constrained terms is done modulo proving the conjoined constraints.

Summarising, we can say that Log In, and most certainly LIFE, is a unified data model. However, the drawbacks for databases are the same as those of the ψ-calculus. The constants are still treated as types rather than as elements of some set. As said before, this might be appropriate for artificial intelligence but not for databases.

2.3.3 LDL with ψ-terms

We already noted that the ψ-calculus does not allow sets in the complex objects, neither does Prolog. So, Log In is not really apt to be used as a a database language. On the other hand, LDL does allow for the construction of sets. Hence, an integration of LDL and ψ-calculus could lead to a unifying formalism. However, the integration proposed in (Beeri, Nasr, and Tsur (1988)), is based on a restricted form of LDL, which does not (yet?) include e.g. *grouping*.

Before we describe the embedding of ψ-terms in LDL, note that LDL already supports a restricted form of complex objects. For example, for the relation schema *emp*(*firstname*, *lastname*, *job*, *education*), we may have the entry *emp*(*jim*, *neat*, *vp*, *degree*(*ms*, *english*, *school*(*harvard*, *ma*), 1981). In this case, *education* is a complex term. The embedding of ψ-terms thus extends the notion of complex objects in LDL.

The syntax of LDL-ψ, is rather similar to Log In, e.g. a *happy* clause given above in Log In, would be written in LDL as:

$$happy(X) \leftarrow likes(X{:}person,\ Y),\ got(X,Y).$$

The semantics, however, is completely different. Recall, that the semantics of LDL-programs are described by minimal models. The same is true for LDL-ψ. However, a provision has to be made for the fact that in ψ-calculus the constants are seen as part of the type structure. Therefore, a universe is built, which differs from the ordinary LDL-universes.
In plain terms, this universe is the collection of all finite or infinite terms, in which the leaves are either values or tags. The key in the formal definition of this universe is the notion of a basic term:

$$t(value \Rightarrow v)$$

Using this notion, it is indifferent whether constants are seen as part of the type structure or not.

The semantics is then defined relative to this universe using ψ-term unification.

As we have seen in the introduction of LDL, it is designed for database applications, as it has primitives for *insert* and *delete*. These primitives are also defined for LDL-ψ, e.g.

$$X + sex \leftarrow X{:}person$$

gives all the *persons* in the database the *sex*-attribute,

$$X(+sex \Rightarrow male) \leftarrow X{:}person(name \Rightarrow john)$$

gives all the *persons* in the database with *name* = *john*, the *sex*-attribute with value *male*.

In general, objects can be added to the database, using rules that have variables in the head, that do not appear in the body. However, the interpretation of such a rule:

$$f(X,\ Y,\ Z) \equiv h(X,\ Y) \leftarrow body(X,\ Z)$$

is $\forall X\ \exists Y\ \forall Z\ f(X,\ Y,\ Z)$. Hence, an object is added to the database, if their is no other object that would allow the rule to succeed. Of course, the drawback of such a 'hard-wired' interpretation is that the user has to be very careful if she intents a different interpretation.

The advantage of LDL-ψ over Log In is basically that LDL-ψ has a set construction. However, LDL-ψ shares the disadvantage that constants are treated as types. Hence, LDL-ψ is rejected on the same grounds as Log In.

2.3.4 O-logic

O-logic is originally defined by D. Maier. However, in the original version sets were not included. There are two revisions of this logic, viz C-logic defined by Chen and Warren (Chen and Warren (1989)), and one, again called O-logic defined by Kifer and Wu (Kifer and Wu (1989)). As the latter contains the

former as a proper subset and we consider sets an important construction for object-oriented databases, we review the O-logic of Kifer and Wu†.

O-logic has some similarities with ψ-calculus, however, it is also radically different. O-logic, is the only formalism in which the object identity plays a significant role. Moreover, the identity is seen as something more than a machine generated unique number. An example O-term is:

$$empl : john[name \rightarrow 'john']$$

In this term *empl* denotes the *class* the object belongs to, *john* is the identity of the object, *name* is a functional label and *'john'* is a basic object. Note that in this example, the object identity is very simple, however, they can be much more complicated and useful.
The alphabet of an O-logic is defined to consist of:

1) a set O of basic objects, with two basic objects: the nil-object denoted by $\top$ and the meaningless object denoted by $\bot$;
2) a set F of function symbols called object constructors;
3) a set $L = L_{\#} \cup L_{*}$ of labels, where $L_{\#}$ is the possibly infinite set of functional labels, and L_{*} the possibly infinite set of set valued labels;
4) a possibly infinite lattice Σ of class names, with two special elements called *all* and *none*;
5) an infinite set V of object variables;
6) the usual logical symbols such as $\forall$, $\vee$ ...

The object identities, called id-terms, are terms, built of object constructors (F), basic objects (O), and object variables (V). An example is:

$$f(a, g(X, b))).$$

The set O^{*} of all ground id-terms plays a role comparable to the Herbrand Universe in classical logic. The elements of O^{*} should be perceived as the objects themselves. In contrast, complex O-terms are assertions about various properties of objects represented by id-terms. Note that this resembles the way in which ψ-terms make assertions about objects in a certain type.

The *O-terms* are defined as follows:

1) a simple O-term is of the form $p:T$, where T is an id-term, and p is a class†;
2) a complex O-term is of the form:

†Again a choice that reflects the bias of the author.
†Note that is notation is exactly the reversed of the usual notation of $o:t$, denoting that object o has type t.

$$p:T[flab_1 \rightarrow t_1,\ldots,flab_n \rightarrow t_n,$$
$$slab_1 \rightarrow \{s_{1,1}, \ldots, s_{1,m_1}\}, \ldots, slab_k \rightarrow \{s_{k,1}, \ldots, s_{k,m_k}\}],$$

with T an id-term, p a class, t_i and $s_{i,j}$ O-terms, the labels $flab_i$ are functional and the labels $slab_j$ are set valued.

Examples of O-terms are:

$$pers:peter[name \rightarrow string:'peter',\ hobby \rightarrow \{game:chess,\ game:bridge\}]$$
$$path:reach(X,\ Y)[start \rightarrow X,\ end \rightarrow Y]$$

All O-terms are O-formulae, O-formulae are obtained from simpler formulae using the logical connectors and quantifiers.

The semantics of O-logic is rather complicated, and we will not discuss it, save the remark that it is based on *models*. Therefore, it makes sense to define a database D as a set of O-formulae, and to define that $D \models \phi$ if ϕ is true in all models of D. Moreover, the authors prove that a variant of the Herbrand Theorem holds for O-logic:

THEOREM

A finite set of O-logic formulae S is inconsistent iff a finite subset of its ground instances is inconsistent.

This theorem is, as in ordinary logic programming, the basis for a sound and complete resolution-based proof procedure.

In our overview of LDL-ψ, we noted that the interpretation whether or not an object should be entered in the database for variables that appear in the head but not in the body is 'hard-wired' in the language. In contrast, in O-logic, the user can convey her intention using the id-term. As an example, compare the two rule pairs:

$$path:add(E,\ nil)[start \rightarrow X,\ end \rightarrow Y] \leftarrow$$
$$\quad edge:E[start \rightarrow node:X,\ end \rightarrow node:Y]$$
$$path:add(E,\ P)[start \rightarrow X,\ end \rightarrow Y] \leftarrow$$
$$\quad edge:E[start \rightarrow node:X,\ end \rightarrow node:Z],$$
$$\quad path:P[start \rightarrow node:Z,\ end \rightarrow node:Y]$$

$$path:reach(X,\ Y)[start \rightarrow X,\ end \rightarrow Y] \leftarrow$$
$$\quad edge:E[start \rightarrow node:X,\ end \rightarrow node:Y]$$
$$path:reach(X,\ Y)[start \rightarrow X,\ end \rightarrow Y] \leftarrow$$
$$\quad edge:E[start \rightarrow node:X,\ end \rightarrow node:Z],$$
$$\quad path:P[start \rightarrow node:Z,\ end \rightarrow node:Y]$$

The first pair results in an object for every path that exists between X and Y, as the path is part of the id-term. While the second pair yields at most one object

for every pair of nodes, as the id-term is defined over X and Y.

Of all models we introduced in this section, O-logic is closest to our list of requirements. The main drawback of O-logic is that it has no clear typing schema. Adding a fulfledged typing schema to O-logic will be a rather difficult task, as the semantics of O-logic is already quite complex. So, for similar reasons as COL, we do not choose O-logic as our starting point.

2.4 Conclusions

The first two sections of this chapter can be seen as an introduction to the areas of object-oriented and deductive databases. In the first section, we describe a few formalisms for complex objects, while deductive databases are the subject of the second section. The third section surveys some unified formalisms. We argue that none of these formalisms is particularly suitable to be extended to meet our requirements.

The first motivation to dismiss some of these formalisms is that we believe that the complex object should be the central concept; as we are interested in complex objects. The remaining formalisms already have a rather complicated semantics, this makes it awkward to add concepts as subtyping or type equivalences. If one succeeds, the resulting semantics will most probably be rather opaque. The most important lesson that the reader should have learned from this chapter is that the type concept should be the central issue in a complex object formalism. Therefore, we start the development of Flock with the definition of its typing mechanism in the next chapter.

Chapter 3

The type graph

To simplify the discussion between the database designer and the, future, user, graphical representations of the UoD are often used. Clearly, the closer the representation technique is to the data model, the more useful the graphical representation is to the database designer. In the previous chapter, we have seen that many formalisms for complex objects support a graphical representation of the type of complex objects. These graphical representations are often *component diagrams*, each node in the graph is connected to its components. So, such connections denote only a *component-of* relationship. In other words, these diagrams do not distinguish between e.g. the connection of a record with its components and a variant-record with its components.

Flock is no exception to the general trend, in that there exists a graphical representation of the type of complex objects. It differs, however, in that this representation is *structural*, i.e. the graph represents the structure of the type. Moreover, the arcs in the graph denote *functions* rather than component-of connections. In fact, the graph structure of types in Flock is the foundation of the type structure of Flock, rather than a convenient representation. The distinction in type structure between Flock and that of other formalisms can be formulated as follows:

> Similar to programming languages, the types in the usual formalisms are built from a *set* of basic types, such as *Bool* and a set of type constructors such as *record*. The type constructors can be used to add new types to a set of basic types.
>
> In Flock, we start with a *graph* of basic types and a set of type constructors. These type constructors can be used to add new nodes (types) and arcs (functions) to the basic type graph.

In the first section of this chapter, we motivate our unconventional choice for a type graph rather than a set of types. Moreover, we show how the usual definitions of generalisation and specialisation can be generalised to this graph structure.

The graph structure of Flock allows us to use constructions from *Category Theory* as the constructors of Flock. In the second section, we introduce these constructors. The definition of each construction is preceded by the definition of the corresponding construction in Set Theory. These Set Theoretical definitions can be seen as a naive semantics for the constructions in Flock, which are only introduced *syntactically* in this section. The formal semantics of Flock, based on Category Theory, are defined in section four of this chapter. The reason why the syntactical and formal semantical definition of the constructions is spread over two sections is twofold:

1) This thesis is on complex objects rather than Category Theory, and it is not assumed that the reader is fluent in Category Theory. Therefore, an early emphasis on the Categorical Semantics would conceal the simplicity of the constructions.

2) The *semantics* of Flock can be found using Category Theory, the formalism itself is *not* Category Theory. Moreover, the naive Set Theoretical Semantics give the reader an important intuition about the more abstract Categorical Semantics.

The introduction of each of the constructors is accompanied by the proof of some simple properties of this constructor as well as an example construction. In the fourth and final section, we introduce the categorical counterparts of our constructors. Moreover, we show that each of the constructors satisfies the definition of its categorical counterpart.

Before we make this first excursion into Category Theory, we illustrate the type constructors of Flock using the well-known *CAR*-example of (van Griethuysen (Ed.) (1984)), in the third section.

3.1 A type graph

In the previous chapter, we noted that the type system is a central part of a formalism for complex objects. Therefore, we start the development of Flock with the introduction of its type system. Usually, such a type system has a set of types. In Flock, however, we have a set of types. In this section, we motivate this unconventional choice and define what a type graph is. Moreover, we show how the usual definitions of specialisation and generalisation can be generalised to the setting of a type graph.

Our unconventional choice for a type graph rather than a set of types is motivated by the following two observations concerning the relational model:

1) Given a relation schema R over a set of attributes A, a relation schema R_B over a set of attributes $B \subseteq A$ can be constructed by projecting the relation R on the attributes in B. When this projection is applied to a table r_A for R_A the result is a table r_B for R_B. More generally, let a relational schema be given by a set of relation schemas R_i, defined over some attribute set A; i.e. $R_i \subseteq A$. By applying the relational operators formally on the R_i, each subset of A can be constructed as a new relation schema R_j. By applying the same sequence of operations to tables r_i for the schemas R_i, a table r_j for the schema R_j is constructed. So, tables of a given schema can be converted to tables of another schema. However, not every subset of A necessarily represents a meaningful chunk of information. Recall that the relation schemas in a relational database represent entity types in the UoD under consideration. It is unlikely that each set of attributes represents such an entity type. As an example consider a relational schema representing a UoD in which, e.g. *cars*, *books*, *freezers*, and *employees* are considered as entities. So, the set of attributes A, might very well contain elements as *average speed*, *number of pages*, *lowest achievable temperature*, and *employee number*. However, it is rather unlikely that the relation schema:

$$R = \{\textit{average speed}, \textit{number of pages}, \textit{lowest achievable temperature}, \textit{employee number}\}$$

has any natural semantical interpretation.

2) There may be more than one possible construction from a new relation schema from a given collection of relation schemas. Consider e.g. the relational schema given by the relation schemas

$$R_1 = \{A, B, C\}, R_2 = \{B, C, D\}.$$

Two possible constructions for the relation schema $R_3 = \{A,D\}$ are:

$$\pi_{AD}(R_1 \bowtie R_2)$$

$$\pi_{AD}(\pi_{AC}(R_1) \bowtie \pi_{CD}(R_2)).$$

The application of these constructions to tables r_1 for the R_1 and r_2 for R_2 will result, in general, in different tables r_3 for R_3. Hence the semantical interpretation of R_3 depends on the construction used. Put differently, the semantical relation between the pair (R_1, R_2) and R_3 depends on the actual construction used.

These two observations imply that a relational schema would be a more accurate description of the UoD if it would contain both the definition of all

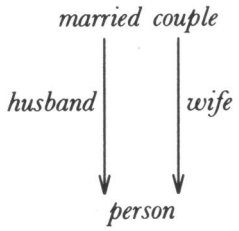
meaningful relation schemas and the permissible conversions between various, sets of, relation schemas. Note that these conversions do not have to be restricted to *pure* relational algebra expressions, because most commercial relational database management systems allow some minimal set of extra functions, called aggregate functions, to be a part of this conversion process. Moreover, by being more liberal in the conversions that are allowed, we also cover the notion of a *derived attribute*. Such an attribute a does not have to be stored explicitly, as the value of a tuple t for that attribute a can be calculated from the values of t for the other attributes. Derived attributes are not only useful for storage saving, but also convey extra information concerning the UoD; viz. a causal relationship among attributes.

The conclusion we draw from this discussion is that an adequate representation of a UoD consists of the types as well as the permissible conversions between (sets of) types. In fact, we may restrict ourselves to conversions between two types, e.g.:

> Let the conversion f have the types t_1 and t_2 as *input-parameters* and t_3 and t_4 as *output-parameters*. Then we may introduce new types t_5 and t_6 such that t_5 represents the $<t_1, t_2>$-pair and t_6 the $<t_3, t_4>$-pair.

Similarly, we can always use one, new, type representing the input parameters and one, new, type representing the output parameters. Moreover, the conversions themselves should not be restricted to expressions in some algebra but rather be allowed to be any, computable, function (as each tighter restriction would put restrictions on the representable derived attributes).

Graphically, we might depict the situation as a graph, in which the (directed) edges represent the possible conversions. Note that there might be more than one edge between two nodes as there might be more than one 'semantical relation' between two relation schemas. Therefore it will be useful to label the edges with a name that conveys the intended meaning of the transformation. In the graph below, we have the types *married-couple* and *person* and the two *projections husband* and *wife*, with their obvious interpretation:



The directed graph representation as we sketched above is called a *typegraph*, denoted by *TG*. The nodes of such a graph are called entity types, the edges are computable functions, called *semantical transformations*. Before we define the notion of a type graph more precisely, we first discuss the nodes and the edges in more detail. We start with the entity types.

An entity type may be likened to a relation schema; although a *type* is more general. An entity type consists of a name and a domain; the structure of such a domain may be more complicated than the Cartesian product structure used for relations.

In data modelling terms, an entity type represents a meaningful collection of 'real world entities'; such a meaningful collection is often called a natural class. The actual mechanism by which it is decided to which natural classes a real world entity belongs is of no concern in this thesis; possibly there is some predicate associated with each class and the real world entities satisfying the predicate are automatically considered to be a member of the class. Note that under this interpretation, a real world entity might be a member of more than one natural class. This turns out to be a very useful property and therefore we assume that real world entities may belong to more than one class.

An entity is now a representation of a real world entity for a given class†. Hence, it is defined as a value of a given type, i.e. as a $<type,value>$-pair. The fact that a real world entity may belong to more than one natural class, means that there may be more than one representing entity for a given real world entity; moreover, these representing entities may have different types.

Above we have described the nature of entity types, we now proceed by describing the function of the edges of our typegraph.

The underlying assumption of our type graph is that the functions (edges) transform one representation of a real world entity into another representation of that same real world entity. This deserves to be spelled out:
A function $f: e_1 \rightarrow e_2$, between the entity types e_1 and e_2 describes how an entity of type e_1 can be transformed (coerced) to an entity of type e_2. Given an entity (e_1, v_1) representing a real world entity o, $f(e_1, v_1) = (e_2, f(v_1))$ is also a representation of o. The fact that there might be more than one function between two entity types, implies that a real world entity may have more than one representation for a given entity type!

To illustrate this, perhaps puzzling, observation we continue our example from above, i.e. consider the entity types *marriage* and *person*, where the domain of *marriage* is e.g.

$$D_{marriage} = D_{person} \times D_{person} \times D_{date}.$$

Now, we can give two transformations from *marriage* to *person*, viz. *husband*, which is the projection on the first *person* component and *wife*, which is the projection on the second *person* component. Hence, a real world entity of the natural class *marriage* is also represented by two persons. To illustrate this even further, let $D_{person} = name$, i.e. a *person* is defined by her name. Then the *marriage* $<marriage, <john, paula, 25/4/89>>$ is represented by the persons $<person, john>$ and $<person, paula>$.

†Usually, the word entity is used both to denote an 'object' in a UoD and a representation of this world entity' whenever we mean an entity in the UoD.

Note at this point that the different representations convey information about the same real world entity; so, the functions *husband* and *wife* represent an *ISA* relation rather than a *HAS-A* relation which implies sharing. Indeed, the intention of the type graph *TG* is that (connected) subgraphs may serve as complex object types. Therefore, we assume that all real world entities have their representations for connected subgraphs of *TG*. Sharing or *HAS-A* relationships between complex objects is one of the subjects treated in Chapter 10.

Having established that the edges, the functions, in *TG* transform representations of one type into representations of another type. This suggests a generalisation of the definition of a semantical transformation from edges to compositions of edges, for we have the following simple result:

PROPOSITION 3.1

Let $f: e_1 \rightarrow e_2$ and $g: e_2 \rightarrow e_3$ be two edges in *TG*, then $g \circ f: e_1 \rightarrow e_3$ transforms an e_1 representation of a real world entity into an e_3 representation of the same real world entity.

PROOF

Let (e_1, v_1) be an entity representing the real world entity o_1. Then $(e_2, f(v_1))$ is another entity representing o_1, as f is a semantical transformation. Now g is also a semantical transformation, hence g transforms a representation of a real world entity o_2 of type e_2 into a representation of o_2 of type e_3. So, g transforms the representation $(e_2, f(v_1))$ of o_1 into the representation $(e_3, g(f(v_1))) = g \circ f(e_1, v_1)$ of o_1. □

With each entity type in the graph, at least one semantical transformation can associated, viz. the identity function. This transformation transforms a representation into itself. Note, that this assumption on *TG* uses the fact that we posed no limits on the nature of the directed graph; more specifically, we did not assume that *TG* is acyclic.

Summarising the assumptions and conventions we made about *TG*, we get:

DEFINITION 3.2

A type graph, denoted by *TG* is a labeled, directed graph in which:

1) The nodes denote entity types, which are $<$*name*, *domain*$>$-pairs.
2) The labels along the edges are functions, whose input type is the type of the source node of the edge and whose output type is the type of the target node of the edge.

The paths in the graph denote semantical transformations. Each semantical transformation is the function constructed by the composition of all the labels along the path. Entities here are $<$*type*, *value*$>$-pairs, such that the value is an element of the domain of the type.

Perhaps superfluously, we want to stress again, that there might be more than one edge between two nodes.

Before we define the notions such as *specialisation* in the context of a typegraph, we give one more example of such a typegraph. In the examples we gave before the edges were simple projections, but we already indicated that an edge may represent an arbitrary functions. To give an example of such an arbitrary function, suppose that an *employee* is represented by a record that has a *name*-field, an *age*-field, and a *salary*-field, i.e. we have the entity type

$$<employee,\ D_{name} \times D_{age} \times D_{salary}>.$$

In general, the information given in such a record is enough to compute the taxes an employee has to pay. So, if we also have the entity type:

$$<taxes,\ D_{taxes}>,$$

We can define the function *taxes-due:employee* $\rightarrow$ *taxes*, which computes the taxes an employee should pay. Now, one could argue that the relationship between *employee* and *taxes* should not be called an ISA relationship. However, for us *ISA*-relationships are a consequence of *specialisation* and *generalisation* in the type graph; which we define below. We return to this problem after the definition of these concepts.

One of the meanings of the word 'special' is 'readily distinguishable from the others in the same category'. Using this interpretation as the basis for *specialisation*, we see that one natural class is a specialisation of another iff the former is a subset of the latter. Entity types are representations of such natural classes: each entity of a given type is a representation of a real world entity in the associated natural class. The functions in *TG* are semantical transformations between the entity types. Let $f: e_1 \rightarrow e_2$ be such a transformation and let o be a real world entity in the natural class associated with e_1, represented by the entity $<e_1, v_1>$. Then $<e_2, f(v_1)>$ is a different representation of the *same* real world entity o. Hence, the natural class associated with e_1 is a subset of the natural class associated with e_2. This leads us to the following definition:

DEFINITION 3.3

Let e_1 and e_2 be entity types in *TG*, then e_1 is called a *specialisation* of e_2 in *TG* iff there is a transformation $f: e_1 \rightarrow e_2$ in *TG*.

Similarly we may define *generalisation*, this concept has an interpretation that is more or less dual to specialisation, viz. 'concerned with universal rather than particular aspects'. So, one natural class is a generalisation of another iff the former is a superset of the latter. By the same discussion as given above, this observation yields the following definition:

DEFINITION 3.4

Let e_1 and e_2 be entity types in *TG*, then e_1 is called a *generalisation* of e_2 in *TG* iff there is a transformation $f: e_2 \rightarrow e_1$ in *TG*.

Further motivation for these definitions may be found in the fact that application of a function at most preserves the amount of information present; i.e. it is not always possible to reconstruct the original from the image. So, our definitions of specialisation/generalisation satisfies the requirement that at the more specialised type more information is known about a real world entity than at the more general type. In other words, *specialisation* and *generalisation* as we defined them are properties of *representations* of real world entities.

If we now look at the examples we have given before, we see that *married-couple* is a specialisation of *person* and *employee* a specialisation of *taxes*. The reason in both cases is the possibility to derive representations:

1) given an entity of type *married-couple*, we can derive to entities of type *person*;

2) given an entity of type *employee*, we can derive an entity of type *taxes*.

Note that the duality in the nature of specialisation and generalisation is expressed by 'reversing the arrow'. This coincides with the categorical notion of duality which we will encounter again in the next section. However, note that although these notions may be dual, this does not imply that for a fixed entity type e the set of its generalisations, say G_e and the set of all its specialisations, say S_e are complementary sets. If only for the fact that there may be entity types that are neither generalisations nor specialisations of e. However, we do have that if $e_2 \in G_{e_1}$, then $S_{e_1} \subseteq S_{e_2}$, because composition of functions yields that specialisations of e_1 are also specialisations of e_2.

Let e_1 be an entity type and let S_{e_1} and G_{e_1} be as defined above. An entity type e_2 is a member of the intersection of S_{e_1} and G_{e_1} iff there are transformations: $f: e_1 \to e_2$ and $g: e_2 \to e_1$ From our discussion above, we see that in such a case, e_1 and e_2 represent the same class of real world objects. This situation is closely connected to the concept of *synonyms*. In every day life, two words are synonymous if they have the same or nearly the same meaning in some or all senses. In a formalism, one tends to be more strict, so for us synonymous means 'the same meaning in all senses'. This could be translated to the type graph by defining two entity types synonymous if they represent the same class of real world objects. Under that definition e_1 and e_2, as given above, would be synonyms. However, this definition is not strong enough for our purposes. As an illustration, consider the following example:

> Let e_1 be the type $<person, D_{person}>$ and let e_2 be the type $<person-pair, D_{person} \times D_{person}>$, i.e. each e_2 entity represents a pair of persons. Moreover, define $g: e_2 \to e_1$ to be the projection on the first component of the pair, i.e.
>
> $$g(<person-pair, (John, Mary)>) = <person, John>.$$
>
> And finally, define $f: e_1 \to e_2$ as the *diagonalisation function*, i.e.
>
> $$f(<person, John>) = <person-pair, (John, John)>.$$

Clearly, we would not want to consider *person* and *person-pair* as synonymous concepts.

This example may seem to be far-fetched, however it illustrates our point. To avoid such counter-intuitive 'synonyms' we put an extra requirement on the two functions. Note that in the example above, $g \circ f = id$, but $f \circ g \neq id$. So, if we require both compositions to be the identity, our example is no longer an illustration of 'synonymous'. Moreover, this extra requirement ensures that there is a one-one correspondence between the entities of each type, which seems to be a reasonable requirement for synonymity. Therefore, we have the following definition:

DEFINITION 3.5

Two entity types e_1 and e_2 are *synonymous* iff there are transformations $f: e_1 \rightarrow e_2$ and $g: e_2 \rightarrow e_1$ such that $f \circ g = id_{e_2}$ and $g \circ f = id_{e_1}$.

If we compare our definitions of specialisation/generalisation with concepts from ER-like models, the main vehicle for information systems modelling, we see that they coincide with the use of an ISA-hierarchy. In these models an *ISA* relationship such as e_1 *ISA* e_2 means that each real world entity represented by e_1 is also represented by e_2. In other words, it means that the natural class represented by e_1 is a subset of the natural class represented by e_2. When we define our complex objects, the reader can verify that such a complex object really contains all representations of a real world entity.

Finally, comparing the graph structure of *TG* with the graph structure of the complex objects of the formalisms we introduced in the second chapter, we see that the edges of *TG* are functions acting on values rather than pointer such as in LDM, IFO, or the calculus of complex objects. Moreover, in contrast with IFO, we only need one kind of arrow to define concepts such as specialisation and generalisation. In turn, this implies that we also do not have to give explicit rules stating which compositions of arrows are meaningful and which are not. Rather, each well typed combination is meaningful. Moreover, it means that we do not restrict ISA-hierarchies to hierarchies explicitly named as such, but also consider the 'subset behaviour' as a generalisation.

3.2 Type constructions

In the previous section, we introduced the use of a type graph, rather than a set of types. In this section we show how to construct such a type graph. We assume that we start with a directed graph of basic types with functions between them. These basic types can be compared to the abstract types of the IFO-model, i.e. they are a kind of objects that have no underlying structure, at least relative to the point of view of the database designer. So, such a basic type may very well be constructed from primitive types like *Int* and *Bool*, but from the

point of view of the model they are indecomposable. Hence, we make no assumptions whether or not such a basic type is *printable*, a distinction IFO makes, because these assumptions have no impact on the complex objects we will define. This basic type graph can be extended with new types and new functions using the six type constructors introduced below; viz. products, co-products, exponents, equalisers, power types, and co-equalisers Moreover, at any time the database designer may define functions, that cannot be obtained by constructions on the type graph.

The types are the entity types of the previous section, hence they are $<$*name*, *domain*$>$-pairs. The names of entity types serve two functions, first of all they serve as an easy reference to the entity types and, secondly, if two entity types have the same domain, they allow us to distinguish between the two types. In the semantics of Flock however, the names of the entity types play no role whatsoever; for the semantics only the domains are important. Therefore, the type-constructors only use the domain of entity types and construct domains for new entity types. So, the database designer should provide meaningful names for the newly created entity types. In this thesis, we use a default naming schema, both for nodes and edges. It is up to the database designer to supply a meaningful name.

In this section, the constructors are introduced set-theoretically. These definitions can be seen as the informal semantics of the constructors, the formal semantics are given in the last section of this chapter. The fact that the type constructors are defined as if types are only sets has the advantage that we may use the constructors on the extensions of entity types, which are intuitively also sets. Due to this 'trick' essentially all concepts that are needed in Flock are defined using the six constructions introduced in this section.

3.2.1 The product construction

The first type constructor is the well-known product, also known as the Cartesian product, direct product, aggregation et cetera, which is the basis for the relational model. In Flock, given the entity types A and B it adds the node $A \times B$ to the type graph, as well as the arcs $\pi_A: A \times B \to A$ and $\pi_B: A \times B \to B$. In set-theoretic terms, the product is defined as:

DEFINITION 3.6

Let A and B be sets, then:

1) the product of A and B is defined by:

$$A \times B = \{(a,b) \mid a \in A, b \in B\};$$

2) the projection functions $\pi_A: A \times B \to A$ and $\pi_B: A \times B \to B$ are defined by

$$\pi_A((a,b)) = a \; ; \; \pi_B((a,b)) = b.$$

The product of sets satisfies a universal property:

LEMMA 3.7

Let A, B, and C be sets and let $f: C \to A$ and $g: C \to B$ be functions, then there exists a unique function $(f, g): C \to A \times B$ such that $\pi_A \circ (f, g) = f$ and $\pi_B \circ (f, g) = g$.

PROOF

Define $(f, g)(c) = (f(c), g(c))$, then we have:

$$\pi_A \circ (f, g)(c) = \pi_A \circ (f(c), g(c)) = f(c).$$

Similarly we have that $\pi_B \circ (f, g) = g$. To see, that (f, g) is the only function satisfying the constraints, let p be another such function. Let $p(c) = (y, z)$. The requirement that $\pi_A \circ p = f$ implies that $y = f(c)$. Similarly, we have that $z = g(c)$. □

Such a property is called universal because it is a statement about all sets. This lemma may seem a mathematical triviality, however, it will prove to be the basis for our semantics in the last section of this chapter. As such, it will in turn prove to be fundamental in the treatment of equivalent types. Moreover, this mathematical triviality is in general not respected in programming languages: In his Turing Award Lecture, Floyd (Floyd (1987)) uses the following predator-prey system as an example:

$$W' = f(W, R)$$
$$R' = g(W, R)$$

He notes that in many (imperative) programming languages a temporary variable is needed to calculate the next state of this system.
The universal property of the product, however, implies that we may model this predator-prey system with only *one* function, viz.:

$$(W', R') = (f, g)(W, R)$$

Finally, note that this universal property can be read as a statement on semantical transformations in the same vein as the predator-prey system above. In this context, it should be read as stating:

> Given semantical transformations $f: e_1 \to e_2$ and $g: e_1 \to e_3$, there is a semantical transformation $(f, g): e_1 \to (e_2 \times e_3)$ (if this latter is an entity type).

In our opinion, such a property is 'to be expected', therefore, the universal property of products is 'hard-wired' into the definition:

DEFINITION 3.8

Let $<e_1, D_1>$ and $<e_2, D_2>$ be entity types in the type graph *TG*. The expression:

$$type\ e_3 = e_1 \times e_2\ [f: e_3 \to e_1, g: e_3 \to e_2].$$

adds a new node $<e_3, D_3>$ and two arcs to *TG* with:

$$D_3 = D_1 \times D_2$$

and

$$f = \pi_{e_1}: e_3 \to e_1; g = \pi_{e_2}: e_3 \to e_2$$

Moreover, whenever a node (entity type) e_4 is connected to e_1 and e_2 with transformations $k: e_4 \to e_1$ and $l: e_4 \to e_2$, respectively it is connected by the arc $(k, l): e_4 \to e_3$.

The definition should be interpreted as follows: execution of the statement adds the node e_3 to the type graph, together with the edges f and g. Moreover, for each e_4 that has transformations $k: e_4 \to e_1$ and $l: e_4 \to e_2$, a transformation $(k, l): e_4 \to e_3$ is added. This latter transformation should always be added, i.e. even if the type e_4 is constructed after e_3.

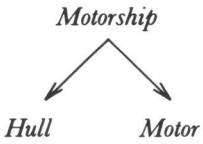
Note that the definition of the product of entity types requires that meaningful names are given both to the new node and to the two new arcs. In this thesis, we will often use the default names $e_1 \times e_2$ for the new node and π_{e_1} and π_{e_2} for the new arcs. Moreover, if an entity type, say e_1, is a factor of more than one other entity type, we will use superscripts to clarify which projection is meant; e.g.:
Let $e_4 = e_1 \times e_2$ and $e_5 = e_1 \times e_3$, then $\pi_{e_1}^{e_4}$ denotes the projection $e_4 \to e_1$ and similarly, $\pi_{e_1}^{e_5}$ denotes the projection $e_5 \to e_1$.
As a final remark on this definition of product, note that we use the $\times$-sign rather than the usual array notation. The reason for this deviation from standard programming languages notations is two-fold:

1) It greatly simplifies the mapping from Flock to its categorical semantics.
2) This notation is more succinct than the standard array notation and this enhances its usability in the 'formula manipulation' necessary to prove the equivalence of two types.

To give an example of the product, suppose that we have the basic types *Hull* and *Motor* then we can define the type *Motorship* by the expression:

$$type\ Motorship = Hull \times Motor$$
$$[hull: Motorship \to Hull,\ motor: Motorship \to Motor]$$

Graphically, this yields:



An easy consequence of the universal property of the set-product is the following corollary, in which $\equiv$ denotes that the two sets are isomorphic:

LEMMA 3.9

Let A, B, and C be sets, then:

1) $A \times B \equiv B \times A$

2) $A \times (B \times C) \equiv (A \times B) \times C$

PROOF

1) The universal property gives us the functions:

$$(\pi_A^{B \times A}, \pi_B^{B \times A}): B \times A \to A \times B;$$
$$(\pi_B^{A \times B}, \pi_A^{A \times B}): A \times B \to B \times A.$$

Composition yields the function:

$$(\pi_B^{A \times B}, \pi_A^{A \times B}) \circ (\pi_A^{B \times A}, \pi_B^{B \times A}): B \times A \to B \times A.$$

Moreover, we have by the universal property:

$$\pi_A^{B \times A} \circ (\pi_B^{A \times B}, \pi_A^{A \times B}) \circ (\pi_A^{B \times A}, \pi_B^{B \times A})$$
$$= \pi_A^{B \times A} \circ (\pi_A^{B \times A}, \pi_B^{B \times A}) = \pi_A^{B \times A}.$$

and similarly:

$$\pi_B^{B \times A} \circ (\pi_B^{A \times B}, \pi_A^{A \times B}) \circ (\pi_A^{B \times A}, \pi_B^{B \times A}) = \pi_B^{B \times A}.$$

However, again by the universal property, the function with these properties is unique and $id_{B \times A}$ already satisfies them. Hence

$$(\pi_B^{A \times B}, \pi_A^{A \times B}) \circ (\pi_A^{B \times A}, \pi_B^{B \times A}) = id_{B \times A}.$$

Similarly, we have that:

$$(\pi_A^{B \times A}, \pi_B^{B \times A}) \circ (\pi_B^{A \times B}, \pi_A^{A \times B}) = id_{A \times B}.$$

2) Similar as 1). □

Since the universal property of the set-product is 'hard-wired' in the definition of the product of entity types and will be the basis of the definition of the semantics, these simple results carry over:

THEOREM 3.10

Let e_1, e_2 and e_3 be entity types in *TG*, then:

1) $e_1 \times e_2$ and $e_2 \times e_1$ are synonymous entity types, i.e. $e_1 \times e_2 \equiv e_2 \times e_1$

2) $e_1 \times (e_2 \times e_3)$ and $(e_1 \times e_2) \times e_3$ are synonymous entity types.

PROOF

In the last section of this chapter, we will see that the categorical product satisfies the universal property. In the proof of the lemma, only this property is used. Therefore, the result carries over. □

This theorem may seem trivial, however, it is noteworthy for a number of reasons. The first reason is that in the proof of the lemma (and hence of the theorem) only the universal property of the product is used. This makes this proof illustrative for category theory, as well as for many of the proofs in this thesis. The second reason is that this theorem gives two examples of entity types that are *automatically* equivalent; this topic is investigated much deeper in Chapter 5.

The first item guarantees that if we use equivalence types rather than the entity types themselves the product does not depend on the order of the 'attributes', similar to the relational model.

The final reason is that the second item allows us to write simply $e_1 \times e_2 \times e_3$, as the associativity makes this expression unambiguous. Moreover, it is easy to see that the universal property of the product is inherited by all finite products. Simply place brackets in the expression and apply the universal property of the product and note that the associativity implies that the so constructed function is well-defined. Hence, we have:

COROLLARY 3.11

Let $e_1, \ldots, e_n$, $n \geqslant 2$ and e be entity types in *TG*, and let $f_i: e \to e_i$ be transformations, then there exists a unique transformation

$$(f_1, \ldots, f_n): e \to e_1 \times \cdots \times e_n$$

such that $\pi_{e_i} \circ (f_1, \ldots, f_n) = f_i$. □

Now that we have defined finite products with more than two components, it is natural to consider what a product with one or even zero components looks like. The universal property of products is our starting point.

Let e_1 be an entity type, we denote the product of e_1 by $\times(e_1)$. The universal property translates to: for each set e_2 with a transformation $f: e_2 \to e_1$ there exists a unique function $(f): e_2 \to \times(e_1)$ such that $\pi_{e_1} \circ (f) = f$.
It is easy to see that e_1 itself, with id_{e_1} as its projection satisfies this requirement. Moreover, it seems to be the obvious choice. Therefore, we define the product of one entity type as the entity type itself.

To translate the universal property in the case of zero entity types we first examine the set product of zero sets:
Let Y denote the product of zero sets, the universal property of the product translates to:
For each set B, there exists a unique function $f : B \rightarrow Y$.
It is easy to see that each singleton set satisfies the requirement placed on Y. Moreover, if a set satisfies the requirement, it has to be a singleton. Hence, all the solutions for Y are isomorphic. In categorical terms, an object with this universal property is called a terminal object, often denoted by 1. The single inhabitant of the set 1 is denoted by *. The unique function $A \rightarrow 1$ is often denoted by !, or $!_A$ if ambiguities could arise.
If we translate this result to Flock, we get that the product of zero entity types is the entity type $<one, 1>$. It turns out that such a type is useful in Flock, just as *Top* and *Bottum* in other formalisms. Therefore, we assume that the type $<one, 1>$ is a node in the basic type graph. To keep the notation simple, we will use the notation 1 rather than $<one, 1>$; which of the two denotations of the symbol 1 is used will be clear from the context. We get the following definition:

DEFINITION 3.12

The basic type graph contains the entity type 1, which is defined by the pair:

$$1 = <one, 1>$$

Moreover, from each node e in the type graph *TG*, there is a unique function:

$$!: e \rightarrow 1$$

The universal property implies that each other entity type in the basic type graph, even in the complete type graph, is a specialisation of 1, i.e. 1 is the most general type in the type graph. The fact that 1 has only one inhabitant, and is the most general type in the type graph can be paraphrased (or justified) as follows:

> IF WE DO NOT CONSIDER ANY PROPERTIES, THE ENTITIES IN THE UoD CAN NOT BE DISTINGUISHED; HENCE THE UoD IS SEEN AS ONE ENTITY.

The fact that we use $\times$ to denote products suggests that we can calculate with types. This suggestion is enhanced by using the symbol 1 for the terminal object of *TG*. The following lemma illustrates that these suggestive notations are not misleading:

LEMMA 3.13

Let e be an entity type, then e and $e \times 1$ are synonymous.

PROOF

Again, we prove this fact for sets using only the universal properties. The fact that these properties also hold in the categorical semantics ensures that the result carries over. So, we have to prove that for the sets A and 1 the required equivalence holds, i.e. that $A \equiv A \times 1$:
With the sets 1, A, and $A \times 1$, we have the following functions:

1) $\pi_A: A \times 1 \to A$.
2) $!_{A \times 1} = \pi_1: A \times 1 \to 1$, note that these two functions are the same by the universal property of the terminal object.
3) $(id_A, !_A): A \to A \times 1$.

By definition we have that $\pi_A \circ (id_A, !_A) = id_A$. To proof that the reverse also holds, i.e. that: $(id_A, !_A) \circ \pi_A = id_{A \times 1}$, note that:

1) $\pi_A \circ (id_A, !_A) \circ \pi_A = \pi_A$
2) $\pi_1 \circ (id_A, !_A) \circ \pi_A = !_A \circ \pi_A$. And by the universal property of the terminal object, this latter term is equal to $!_{A \times 1}$. □

More on this 'calculus of types' may be found in Chapter 5.

Finally, above, we only defined products for the nodes in the type graph, this definition can easily be extended to the edges as well. For if $f: e_1 \to e_3$ and $g: e_2 \to e_4$, we have a function $e_1 \times e_2 \to e_3 \times e_4$ which outputs $(f(a), g(b))$ for an input (a, b). This set-theoretic construction can be rephrased using the universal property of the product as follows:
Let $f: e_1 \to e_3$ and $g: e_2 \to e_3$ be transformations. The entity type $e_1 \times e_2$ has the transformations π_{e_1} and π_{e_2}, composition of these transformations with f and g respectively yields the transformations:

$$f \circ \pi_{e_1}: e_1 \times e_2 \to e_3,$$

$$g \circ \pi_{e_2}: e_1 \times e_2 \to e_4,$$

Hence, by the universal property of the product, we have:

$$(f \circ \pi_{e_1}, g \circ \pi_{e_2}): e_1 \times e_2 \to e_3 \times e_4.$$

The function $(f \circ \pi_{e_1}, g \circ \pi_{e_2})$ is often denoted by $f \times g$. It is easy to see that this function is exactly the one introduced Set-theoretically above. This discussion gives us the following definition:

DEFINITION 3.14

Let e_1, e_2, e_3, e_4, $e_1 \times e_2$, and $e_3 \times e_4$ be entity types in *TG*. Moreover, let $f: e_1 \to e_3$ and $g: e_2 \to e_3$ be transformations, then:

$$f \times g = (f \circ \pi_{e_1}, g \circ \pi_{e_2}): e_1 \times e_2 \to e_3 \times e_4.$$

Note that this definition is only a syntactical convention; we do not have to 'hardwire' the existence of this transformation in Flock, as it is already guaranteed to exist by the universal property of the product.

3.2.2 The co-product construction

The second constructor is called the co-product, also known as the direct sum and the disjoint union. In Flock, this construct adds the node $A + B$ to the type graph, together with the arcs $i_A: A \rightarrow A + B$ and $i_B: B \rightarrow A + B$. So, it is a construction dual to the product in the sense that for co-products the components have arcs *to* the new type rather than vice versa; i.e. in a product the arrow is $\pi_A: A \times B \rightarrow A$, while for co-products the arrow is $i_A: A \rightarrow A + B$.

As with products, we first recall the set-theoretic definition of the co-product (disjoint union). To define the co-product in set-theoretic terms, we assume an extra set, the index set, say $I = \{0,1\}$. The co-product is then defined as a subset of $(A \times I) \cup (B \times I)$ as follows:

DEFINITION 3.15

Let A and B be sets, and let the index set $I = \{0, 1\}$:

1) The co-product of A and B is defined by:

$$A + B = \{(a, 0) \mid a \in A\} \cup \{(b, 1) \mid b \in B\}.$$

2) The injection functions $i_A: A \rightarrow A + B$ and $i_B: B \rightarrow A + B$ are defined by:

$$i_A(a) = (a, 0); \; i_B(b) = (b, 1).$$

As a technical aside, note that the choice of the index used in this definition of the co-product is not essential.

Like the set-product, the set-co-product has a universal property:

LEMMA 3.16

Let A, B and C be sets and let $f: A \rightarrow C$ and $g: B \rightarrow C$ be functions, then there exists a unique function $[f, g]: A + B \rightarrow C$ such that $[f, g] \circ i_A = f$ and $[f, g] \circ i_B = g$.

PROOF

Define $[f, g]((a, 0)) = f(a)$ and $[f, g]((b, 1)) = g(b)$, then we have $[f, g] \circ i_{A(a)} = [f, g]((a, 0)) = f(a)$; similarly, we have that $[f, g] \circ i_B = g$. To see, that $[f, g]$ is the only function satisfying the constraints, let p be another such function. Let $p((a, 0)) = y$. The requirement that $[f, g] \circ i_A = f$ implies that $y = f(a)$. Similarly, we have for $p((b, 1)) = z$ that $z = g(b)$. □

This lemma illustrates the duality between products and co-products even clearer, the major difference between the two universal properties is that the direction of all functions, and therefore all compositions, is reversed. In fact, the proof is the same modulo reversal of function direction. This is exactly the notion of duality in Category Theory. It is a well-known principle in this area that given a proof of some theorem, the dual theorem is also true. Moreover, the proof of the dual theorem is the dual of the proof of the 'normal' theorem. We will refrain from using this principle in this section as a spelled out proof may give the reader more insight in the constructors of Flock †.

Again, the universal property of co-products will be instrumental both for the semantics of Flock and for use of equivalence types. Therefore, the universal property of co-products will be part of the definition of co-products for entity types. But before we give this definition, note that the universal property is the mathematical formulation of a well-known programming languages construction: The property states that if we have a function $f: A \rightarrow C$ and a function $g: B \rightarrow C$, we have a function $[f, g]: A + B \rightarrow C$. A computer scientist would write this function like:

```
case x in
    A : f(x);
    B : g(x);
esac
```

Using the set-theoretic definition of the co-product as well as its universal property, we get the following definition of co-products in Flock:

DEFINITION 3.17

Let $<e_1, D_1>$ and $<e_2, D_2>$ be entity types in the type graph *TG*. The expression:

$$type\ e_3 = e_1 + e_2\ [f: e_1 \rightarrow e_3, g: e_2 \rightarrow e_3].$$

adds the new node $<e_3, D_3>$ and the two arcs f and g to *TG* with:

$$D_3 = D_1 + D_2$$

and

$$f = i_{e_1}: e_1 \rightarrow e_3; g = i_{e_2}: e_2 \rightarrow e_3$$

Moreover, whenever e_1 and e_2 are connected to the same entity type e_3 by transformations $k: e_1 \rightarrow e_4$ and $l: e_2 \rightarrow e_4$, then e_3 is also connected to e_4 by the transformation: $[k, l]: e_3 \rightarrow e_4$.

Note that similar to the remark we made following the definition of products, the addition of $[k, l]$ to the type graph should always occur, even if e_4 is constructed after e_3.

†Usually, this is seen as an exercise left to the reader; however, one cannot expect the reader to be more industrious than the author.

As for products, the definition of co-products requires that meaningful names are given both to the new node and to the two new edges. Again, we will use the default names $e_1 + e_2$ and i_{e_1} and i_{e_2}.

The notation of co-products is again a deviation from the standard notation in programming languages. The reasons are the same as for products, i.e. the mapping to the semantics is easier and the succinct notation is more suitable for 'formula manipulation'. In fact, we showed in the previous subsection that the suggestive notations are not misleading. So, given the $\times$ and the $+$ notation, one would expect the following theorem to hold (in which $\equiv$ denotes type equivalence):

THEOREM 3.18

Let e_1, e_2 and e_3 be entity types, then:

$$(e_1 + e_2) \times e_3 \equiv e_1 \times e_3 + e_2 \times e_3$$

Indeed, this theorem holds, however, its proof is postponed to chapter 5. In the meantime, however, a comparison of our notation with the usual:

$$[a : \mathit{union}[b : t_1, c : t_2], d : t_3] \equiv \mathit{union}[[b : t_1, d : t_3], [c : t_2, d : t_3]]$$

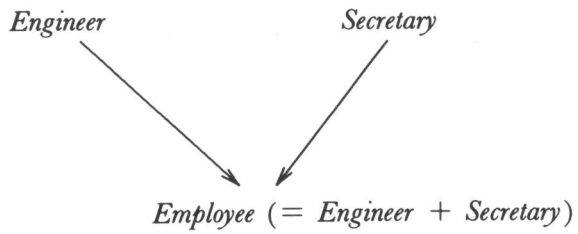
should suffice to illustrate why we think our notation is more adequate for a calculus of types.

Comparing this definition of the co-product with the use of the disjoint union in other formalisms such as LDM and the calculus of complex objects shows a major difference. In those formalisms, there is always an arrow from the *relation* to its *components*. In Flock, however, all edges are functions. Hence, where LDM would have an arrow $A + B \rightarrow A$, we have a *function* $i_A : A \rightarrow A + B$. We believe that in keeping the semantics of the constructors as close as possible to their mathematical intuition, makes Flock is easier to understand.

To illustrate this constructor, suppose that a company has employees and each employee is either a secretary or an engineer. So, we have the sets *Employee*, *Secretary*, and *Engineer*. The constraint says that the *Employee* is in one-one correspondence with the disjoint union of *Secretary* and *Engineer*; i.e.

$$\begin{aligned} \mathit{type}\ \mathit{Employee} = {} & \mathit{Secretary} + \mathit{Engineer} \\ & [\mathit{is-emp} : \mathit{Secretary} \rightarrow \mathit{Employee}, \\ & \mathit{is-emp} : \mathit{Engineer} \rightarrow \mathit{Employee}]. \end{aligned}$$

The effect of this type definition on the type graph can be depicted graphically as follows:



As for the product, the universal property of the set-co-product has some simple consequences, given in the next lemma:

LEMMA 3.19

Let A, B and C be sets, then:

1) $A + B \equiv B + A$

2) $A + (B + C) \equiv (A + B) + C$

PROOF

1) The universal property gives us the functions:

$$[i^A_{A+B}, i^B_{A+B}]: B + A \to A + B;$$
$$[i^A_{B+A}, i^B_{B+A}]: A + B \to B + A.$$

Composition yields the function:

$$[i^A_{B+A}, i^B_{B+A}] \circ [i^A_{A+B}, i^B_{A+B}]: B + A \to B + A.$$

Moreover, we have by the universal property:

$$[i^A_{B+A}, i^B_{B+A}] \circ [i^A_{A+B}, i^B_{A+B}] \circ i^A_{B+A}$$
$$[i^A_{B+A}, i^B_{B+A}] \circ i^A_{A+B} = i^A_{B+A}$$

and similarly:

$$[i^A_{B+A}, i^B_{B+A}] \circ [i^A_{A+B}, i^B_{A+B}] \circ i^B_{B+A} = i_{B+A^B}.$$

However, by the universal property, the function with these properties is unique and id_{B+A} already satisfies them. Hence

$$[i^A_{B+A}, i^B_{B+A}] \circ [i^A_{A+B}, i^B_{A+B}] = id_{B+A}.$$

Similarly we have that:

$$[i^A_{A+B}, i^B_{A+B}] \circ [i^A_{B+A}, i^B_{B+A}] = id_{A+B}.$$

2) Similar as 1). □

Note that this proof again illustrates the duality between products and co-

products, it is the same proof as for products modulo arrow-reversing. Again, this result simply carries over to the type graph:

THEOREM 3.20

Let e_1, e_2, and e_3 be entity types in *TG*, then:

1) $e_1 + e_2$ and $e_2 + e_1$ are synonymous entity types,

2) $e_1 + (e_2 + e_3)$ and $(e_1 + e_2) + e_3$ are synonymous entity types.

PROOF

Again, this lemma only uses the universal property of the co-product. This universal property is again the basis of the semantics as we will see at the end of this chapter. Therefore, the result carries over. □

Again, this theorem not only serves as an illustration of equivalent types, because the second item enables, again, the usage of a co-product with more than two components. Indeed, we may simply write $e_1 + e_2 + e_3$ as the associativity makes this expression unambiguous. Moreover, it is again easy to see that the universal property of the co-product is inherited by 'variable length' co-products: place brackets in the expression, and use the universal property of the co-product. The associativity guarantees that the resulting function is well-defined. Hence, we get:

COROLLARY 3.21

Let $e_1, \ldots, e_n$, $n \geqslant 2$ and e be entity types in *TG*, and let $f_i: e_i \to e$ be transformations in TG function, then there exists a unique transformation

$$[f_1, \ldots, f_n]: e_1 + \cdots + e_n \to e$$

such that $[f_1, \ldots, f_n] \circ i_i = f_i$. □

So, similar to products, we have finite co-products. This makes it again worthwhile to investigate co-products of on type and of zero types set-theoretically. An then, if possible, define such co-products in terms of Flock.

Let A be a set, and denote by $+(A)$, the co-product of A.The universal property translates to:

for each set B with a function $f: A \to B$ there exists a unique function $[f]: +(A) \to B$ such that $[f] \circ i_A = f$.

It is easy to see that A itself, with id_A as its injection satisfies this requirement, moreover, it seems to be the obvious choice. Therefore, we define the co-product of one set as the set itself. Hence, the co-product of one entity type is defined as the entity type itself.

To translate the universal property in the case of zero sets, we need to give the co-product a name, say Y. The property then becomes:
For each set B, there exists a unique function $f: Y \to B$.

It is easy to see that the empty set satisfies the requirement placed on Y. Moreover, if a set satisfies the requirement, it has to be empty. In categorical terms, an object with this universal property is called an initial object, often denoted by 0. The unique function from 0 to an object A is often denoted by $!_A$ or ! if no confusion may arise.

If we translate this to Flock, we get that the co-product of zero entity types is the entity type $<zero, 0>$. It turns out that such a type is useful in Flock, just as *Top* and *Bottum* in other formalisms and 1 introduced in Flock above. Therefore, we assume that the type $<zero, 0>$ is a node in the basic type graph. In fact, we shall often refer to this entity type as simply 0. As an aside, note that 0 and 1 can be characterised set-theoretically as $\varnothing$ and $\{\varnothing\}$ respectively. We will use this characterisation later in this thesis.

The universal property implies that each other entity type in the basic type graph, even in the complete type graph, is a generalisation of 1, i.e. 0 is the most specialised type in the type graph. The fact that 0 has no inhabitants and is the most specialised type in the type graph can be paraphrased (or justified) as follows:

NO OBJECT HAS ALL POSSIBLE PROPERTIES.

It is relatively easy to see that the suggestive use of 0 is not misleading, as we have:

LEMMA 3.22

Let e be an entity type, then e and $e + 0$ are equivalent.

PROOF

Again, it is enough to prove that for a set A, $A \equiv A + 0$ holds:
For the sets 0, A and $A + 0$ we have the following functions:

1) $i_A: A \rightarrow A + 0$.
2) $i_0 = !_{A+0}: 0 \rightarrow A + 0$, note that these functions are the same by the universal property of 0.
3) $[id_A, !_A]: A + 0 \rightarrow A$.

By definition, we have that $[id_A, !_A] \circ i_A = id_A$. To proof that $i_A \circ [id_A, !_A] = id_{A+0}$, note that:

1) $i_A \circ [id_A, !_A] \circ i_A = i_A$
2) $i_A \circ [id_A, !_A] \circ i_0 = i_A \circ !_A = !_{A+0}$

Hence, by the universal property, $i_A \circ [id_A, !_A] = id_{A+0}$ holds. □

As with the product, we can also define a co-product on the transformations. Let $f: e_1 \rightarrow e_3$ and $g: e_2 \rightarrow e_4$ be transformations. Then we have the transformations:

$$i_{e_3} \circ f: e_1 \rightarrow e_3 + e_4,$$

$$i_{e_4} \circ g: e_2 \rightarrow e_3 + e_4.$$

Hence, we have the transformation:

$$[i_{e_3} \circ f, i_{e_4} \circ g]: e_1 + e_2 \rightarrow e_3 + e_4$$

This leads to the following *notational* definition:

DEFINITION 3.23

Let $f: e_1 \rightarrow e_3$ and $g: e_2 \rightarrow e_4$ be transformations. The transformation $f + g: e_1 + e_2 \rightarrow e_3 + e_4$ is defined by:

$$f + g = [i_{e_3} \circ f, i_{e_4} \circ g]$$

3.2.3 The exponent construction

The third construction is called exponentiation. This construction is rather different from the first two constructions, if only, because it is not inspired by a well-known set-theoretic construction. Moreover, none of the formalisms for complex objects introduced in the second chapter has an analogous construction. Therefore, we start this subsection with a motivation for the construction instead of with a definition.

The paths in the type graph are functions. These functions are *static* in the sense that although they transform one representation of a real world object into another representation, these transformations do *not* reflect a change in the UoD. Rather, these transformations translate between different views on the same real world entity. Spelled out:

> If a real world entity r is represented by the entities $<e_1, d_1>$ and $<e_2, d_2>$, this should be interpreted as meaning that looking through e_1-glasses r appears as d_1 and looking through e_2-glasses r appears as d_2. If their is a transformation $f: e_1 \rightarrow e_2$, then f denotes the way in which the e_2 representation can be derived from the e_1 representation.

So, the functions in the type graph are part of the static representation of the UoD. In the UoD, we do not only find *static entities* such as persons, but also *dynamic entities* such as *raise* or *hire*. Clearly, we also need functions to model such dynamic entities. The type of such entities will be *functional*. It is not enough if some of the basic types are functional, as for the new types that are constructed new dynamic entities are needed. To illustrate this, suppose that *person* is a basic entity type, and we construct a new entity type *employee*, which is a specialisation of *person*. Moreover, suppose that we want to *hire* new employees, this will be a function *person* $\rightarrow$ *employee*. As *employee* is not a basic type, it is not very likely

that there is a basic entity type representing the functions *person* → *employee*.

The upshot of this discussion is that we need a construction that given two entity types e_1 and e_2 yields the entity type e_3 that represents all possible dynamical entities (functions) $e_1 \to e_2$. This is the role of *exponentiation*. Set-theoretically, it is defined as follows:

DEFINITION 3.24

Let A and B be sets, the set B^A is defined by:

$$B^A = \{f \mid f: A \to B\}.$$

So, B^A is the set of *all* set-functions $A \to B$. Before we define exponentiation for entity types, we first investigate B^A a bit further.

Given dynamical entities, i.e. entities of an exponential type, we have to be able to apply these dynamic entities on other entities to model events in the UoD. Suppose we have the entity *John* of type *employee* and the entity *raise* of type $employee^{employee}$. If we want to give *John* a raise, we have to evaluate *raise* with respect to *John*. Clearly, if f is a function of type B^A, we need an element of A to evaluate it. Hence, the *evaluation* is a function:

$$eval: B^A \times A \to B.$$

Set-theoretically, it is defined as follows:

DEFINITION 3.25

Let A and B be sets, the function $eval: B^A \times A \to B$ is defined by:

$$eval(f, a) = f(a)$$

The function *eval* is of course well known from functional programming languages such as *LISP*. In such languages, functions may be partially evaluated in a process called *currying*.

For example, let $add: N \times N \to N$ be the well-known natural numbers addition. By currying, we can construct the function add_3 from add with $add_3: N \to N$. The evaluation of add_3 is defined by:

$$eval(add_3, 4) = eval(add, (3, 4)) = 7$$

In general currying constructs a function $\hat{f}: A \to B^C$ from a function $f: A \times C \to B$. This construction is in fact the universal property of *exponentiation* and *eval*:

THEOREM 3.26

Let A, B, and C be sets, and let $g: C \times A \to B$ be a function. Then there is a unique function $\hat{g}: C \to B^A$ such that:

$$eval \circ (\hat{g} \times id_A) = g.$$

PROOF

For a given $c \in C$, define $g_c: A \to B$ as

$$g_c(a) = g((c,a))$$

Now, $\hat{g}$ can be defined by $\hat{g}(c) = g_c$. For any $(c, a) \in C \times A$ we have:

$$eval((\hat{g}(c), a)) = g_c(a) = g((c, a))$$

Let p be another function that satisfies the requirements of the theorem, then $p(c)$ has to be function such that $p(c)(a) = g((c, a))$. Hence $p(c) = g_c$. □

To illustrate this universal property, suppose that we have the function *raise* : *employee* × *amount* → *employee*. If we specify the amount a of the raise, we can curry *raise* to $raise_a$, which is evaluated as defined above, i.e. $eval(raise_a(John)) = raise(John, a)$.

Another illustration of the universal property is given by the following lemma:

LEMMA 3.27

Let A, B, C, and D be sets with functions $f: C \to A$ and $g: B \to D$. Then there exists the function:

$$(f \circ eval \circ (g \times id))^{\#}: C^D \to A^B$$

For notational convenience we write $\hat{function}^{\#}$ rather than $\hat{function}$; we will sometimes refer to this function as $exp(f, g)$.

PROOF

Informally, the argument is that given a function $h \in C^D$, i.e. $h: D \to C$, we have $f \circ h \circ g: B \to A$. This argument can be formalised using the universal property of exponentiation. Namely, $g: B \to D$ and $id: C^D \to C^D$ imply:

$$g \times id: C^D \times B \to C^D \times D.$$

Composition with $eval: C^D \times D \to C$ and $f: C \to A$ yields

$$f \circ eval \circ (g \times id): C^D \times B \to A.$$

Hence by the universal property of exponentiation, we have a function

$$(f \circ eval \circ (g \times id))^{\#}: C^D \to A^B. \quad \square$$

To give an example of this lemma, suppose we have the nodes *employee*, *manager*, and *director* in the type graph with the obvious functions between them. If we instantiate A = *manager*, $B = D$ = *employee*, and C = *director*, the lemma tells us that the promotion from an employee to a director is a specialisation of the promotion from an employee to a manager.

Clearly, the *eval* function should not be added to the type graph as it is used to model changes in the UoD rather than connecting two *simultaneous* views on the same real world entity. As the universal property of exponentiation depends on *eval*, this property can not be part of the definition of the exponent construction. Both *eval* and the universal property belong to the semantics of Flock. The function

$$(f \circ eval \circ (g \times id))^{\#}: C^D \rightarrow A^B$$

does belong to the type graph to the type graph, as it shows how the hierarchy of 'static' types translates to dynamic types. Moreover, the example above clearly illustrates the use of this lemma.

Hence, we have the following definition of exponentiation of entity types:

DEFINITION 3.28

Let $<e_1, D_1>$ and $<e_2, D_2>$ be entity types in the type graph *TG*. The expression:

$$type\ e_3 = e_1^{e_2}$$

adds the new node $<e_3, D_3>$ to *TG* with:

$$D_3 = D_1^{D_2}$$

Moreover, iff there are also entity types e_3, e_4 and $e_3^{e_4}$ with edges $f: e_3: \rightarrow e_1$ and $g: e_2 \rightarrow e_4$. It also adds the edge

$$exp(f, g): e_3^{e_4} \rightarrow e_1^{e_2}$$

Note that exponentiation does not have to add new edges except of course the standard edges $!: 0 \rightarrow e_3$ and $!: e_3 \rightarrow 1$.

Exponentiation may be new to complex object formalisms, it is, of course, well-known in the area of type theory for programming languages; see e.g. (Cardelli (1988)). In such theories, $e_1^{e_2}$ is often denoted as $e_2 \rightarrow e_1$. So, again we deviate from 'standard' notation, and again it is to allow easier manipulation. Moreover, again, the suggestive notation is not misleading:

Let f be a function $f: A + B \rightarrow C$, then f induces the two functions $f \circ i_A: A \rightarrow C$ and $f \circ i_B: B \rightarrow C$. Moreover, two functions $g: A \rightarrow C$ and $h: B \rightarrow C$ induce a function $[g, h]: A + B \rightarrow C$.

This observation suggests the following theorem, which will be proved in Chapter 5:

THEOREM 3.29

Let e_1, e_2, and e_3 be entity types, then:

$$e_1^{e_2 + e_3} \equiv e_1^{e_2} \times e_1^{e_3}.$$

Both the edges in *TG* and the elements of an exponential type are functions, the universal property of exponentiation can be used to show a connection between these two 'classes' of functions. To do this, we first have to define an element of a type (i.e. an entity) in Flock:
For any set A, there is a correspondence between the elements of A and the functions from 1 to A: for each $a \in A$ we can define the function f_a: $1 \rightarrow A$, such that $f_a(*) = a$. Moreover, for each function g: $1 \rightarrow A$ there is an $a \in A$ such that $g(*) = a$, i.e. $g = f_a$. This gives us the following definition:

DEFINITION 3.30
Let e be an entity type, an entity of type e is a function f: $1 \rightarrow e$.

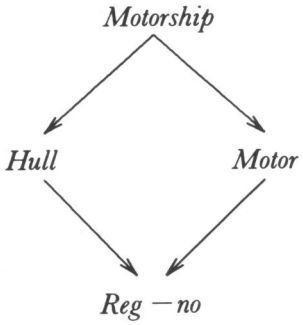
Note that entities are not a part of the type graph.

Let f be an edge of the type graph, i.e. f: $A \rightarrow B$ is in the type graph. Then we have that $f \circ \pi_A$: $1 \times A \rightarrow B$. Hence, by the universal property of exponentiation, we have a unique function $\hat{f}$: $1 \rightarrow B^A$ such that $eval \circ \hat{f} \times id_A = f \circ \pi_A$. So $\hat{f}$ has the same effect on an element of a as f has, moreover, $\hat{f}$ is an element of B^A in the sense defined above. As a final aside note that $\hat{f}$ is often called the *name* of f.

3.2.4 The equaliser construction

The fourth type construction is again rather different from the first three. Whereas product, coproduct, and exponentiation used nodes in the type graph to construct new nodes and new edges, the equalising construction uses *paths* in the type graph to construct new nodes and new edges. Again, this is a construction that is not common in formalisms for complex objects, only the ψ-calculus has a similar construction. Moreover, it is neither based on a well-known set-theoretic construction. Therefore, we start with a motivation.

There may be more than one path in the type graph, connecting two entity types. In principle, these paths will represent different semantical transformations, i.e. different functions. As an example, suppose that the *motorships* of the product example are to be registered. Moreover, suppose that this is done by registering the *hull* and the *motor*, i.e. *Reg-no* becomes a supertype of both *Hull* and *Motor*. This can be depicted graphically as follows:



As a *motorship* is just a (*hull*, *motor*)-pair, it is not necessary that the *hull* and the *motor* of such a *motorship* have the same *registration number*. For the sake of the argument, suppose that a *motorship* is only considered registered if the *hull* and the *motor* have the same *registration number*. Clearly, the type *motorship* is not the adequate type for registered motorships, we need to construct the type of the (*hull*, *motor*)-pairs in which both the *hull* and the *motor* have the same registration number.

This is were the *equaliser*-construction comes in. Set-theoretically, it is defined as follows:

DEFINITION 3.31

Let A and B be two sets, and let f and g be two *parallel* functions (functions with the same source and target), $f, g: A \to B$, then:

$$C = \{a \in A \mid f(a) = g(a)\}$$

So, the equaliser generates the largest subset of A on which f and g agree. In other words, it forces f and g to be equal; hence, its name. As an aside note that the equaliser is a selection mechanism; in that capacity it will prove to be the heart of selections in the query language of Flock.

As C is defined as a subset of A, we have an injection $i: C \to A$, defined by $i(c) = c$. This function satisfies the equation:

$$f \circ i = g \circ i$$

This injection equation is the basis for the universal property of the equaliser:

THEOREM 3.32

Let A, B, and C be sets, with parallel functions $f, g: A \to B$ such that C is the equaliser of f and g and $i: C \to A$ its injection. For any set D with a function $h: D \to A$ such that $f \circ h = g \circ h$, there is a unique function $k: D \to C$ such that $h = i \circ k$.

PROOF

Note that for a $d \in D$ $f(h(d)) = g(h(d)$, so $h(d) \in C$. Hence we can

define the function $k: D \rightarrow C$ as being equal to k! Moreover, this is the only function that satisfies the requirement, for we must have that $i(k(c)) = h(c)$ which translates to $k(c) = h(c)$ as i is an injection. □

Note that this universal property is yet another way to state that the equaliser is the largest subset of A on which f and g agree. Just as the universal property of the product and the co-product, this universal property may add functions to the type graph. Again, these functions seem to denote logical semantical transformations. Therefore, we add the property to the definition of the equaliser in *TG*:

DEFINITION 3.33

Let $<e_1, D_1>$ and $<e_2, D_2>$ be two entity types in the type graph *TG*. Moreover, let $f, g: e_1 \rightarrow e_2$ be two parallel semantical transformations in *TG*. The expression:

$$type\ e_3 = eq(f, g)\ [i: e_3 \rightarrow e_1]$$

adds a new node $<e_3, D_3>$ and a new edge $i: e_3 \rightarrow e_1$ to the type graph *TG*, with

$$D_3 = eq(f, g: D_1 \rightarrow D_2)$$

$$i(d) = d$$

Moreover, whenever there is a node e_4 with a semantical transformation $h: e_4 \rightarrow e_1$ such that $f \circ h = g \circ h$, then it is connected by a semantical transformation $k: e_4 \rightarrow e_3$ such that $h = i \circ k$.

As an illustration of this construction, we continue our example of registered motorships from the beginning of this subsection. The registered ships are those for which the *Motor* and *Hull* have the same *Reg-no*, i.e. we can construct this type as the equaliser of the parallel functions:

$$reg-no \circ \pi_{Hull},\ reg-no \circ \pi_{Motor}: Motorship \rightarrow Reg-no.$$

Hence, we define the type by:

$$type\ Registered\ ship = eq(reg-no \circ \pi_{Hull},\ reg-no \circ \pi_{Motor})$$
$$[i: Registered\ ship \rightarrow Motorship]$$

Note that we use i both for co-products and equalisers, the reason is that in both cases, the function is the canonical injection. We expect that it will be no cause for confusion as the context will clarify the intended function. If we need to distinguish between two equalisers, we will use a sub-script, such as $i_{f,g}$.

The equaliser is defined to find the largest subset on which two functions agree. As there may be more than two semantical transformations between two entity types, the equaliser may seem to be a restricted operator. However, using the equaliser, we can construct subsets on which an arbitrary number of functions agree. To see how this works, consider the following example:

Let A and B be two sets with three parallel functions $f, g, h: A \to B$. To construct the subset of A on which f and g and h agree, we can use different strategies, e.g.:

1) Let C denote the equaliser of f and g, with its injection $i_{f,g}: C \to A$. Then we have the two functions:

$$f \circ i_{f,g} \ (= g \circ i_{f,g}),\ h \circ i_{f,g}: C \to B$$

Let D be the equaliser of these two functions, i.e. $D = eq(f \circ i_{f,g}, h \circ i_{f,g})$, with the injection $j: D \to C$. Clearly, D is a subset of A, with injection $k = i_{f,g} \circ j: D \to A$. Moreover, f, g and h agree on D, i.e. $f \circ k = g \circ k = h \circ k$.

2) Let C' denote the equaliser of f and h, with its injection $i_{f,h}: C' \to A$. Again we have the two functions:

$$f \circ i_{f,h} \ (= h \circ i_{f,h}),\ g \circ i_{f,h}: C \to B$$

Now, let D' be the equaliser of these two functions with the injection $j': D' \to C$. Again, D' is a subset of A, with injection $k' = i_{f,h} \circ j': D' \to A$. Moreover, f, g and h agree on D', i.e. $f \circ k' = g \circ k' = h \circ k'$.

Set-theoretically, it is easy to see that D and D' are actually the same subset of A:

$$\begin{aligned} x \in D &\Leftrightarrow (x \in C \subseteq A \wedge f(x) = h(x)) \\ &\Leftrightarrow (f(x) = g(x) = h(x)) \\ &\Leftrightarrow (x \in C' \subseteq A \wedge f(x) = g(x)) \Leftrightarrow x \in D' \end{aligned}$$

Moreover, as both D and D' are constructed as equalisers, it is obvious that the universal property of an equaliser of two functions carries over to the equaliser of three functions. Hence, we may write $eq(f_1,\ldots,f_n)$ without being ambiguous.

Using the universal property of equalisers rather than set-theoretic arguments, we can not prove that D and D' are the same set. However, we can prove that D and D' are isomorphic sets. In the last section of this chapter, we shall see that in the *semantics* of Flock, this suffices. As an illustration of the use of the universal property of the equaliser, we give the proof:

LEMMA 3.34

D and D' as constructed above are isomorphic.

PROOF

We have already seen that the following equations hold:

1) $f \circ k = g \circ k = h \circ k$

2) $f \circ k' = g \circ k' = h \circ k'$

As C is the equaliser of f and g, the second equation and the universal property of C gives us:

$$\exists l\colon D' \to C\colon i_{f,g} \circ l = k'$$

Similarly, we have:

$$\exists l'\colon D \to C\colon i_{f,h} \circ l' = k$$

But then,

$$f \circ i_{f,g} \circ l = f \circ k' = h \circ k' = h \circ i_{f,g} \circ l$$

As D is the equaliser of $f \circ i_{f,g}$ and $h \circ i_{f,g}$, its universal property gives:

$$\exists m\colon D' \to D\colon j \circ m = l$$

Similarly, we have that

$$\exists m'\colon D \to D'\colon j' \circ m' = l'$$

Next, we have to prove that $m \circ m' = id_D$ and that $m' \circ m = id_{D'}$. To proof this, note that:

$$k \circ m \circ m' = i_{f,g} \circ j \circ m \circ m' = i_{f,g} \circ l \circ m' =$$

$$k' \circ m' = i_{f,h} \circ j' \circ m' = i_{f,h} \circ l' = k$$

However, by the universal property of D, there is exactly one function $p\colon D \to D$ with the property $k \circ p = p$. Clearly, id_D satisfies this equation and hence $m \circ m' = id_D$. Similarly, $m' \circ m = id_{D'}$ holds. □

We end this subsection, by giving an example of type equivalences in which equalisers occur. The proof of this example may be found in Chapter 5:

THEOREM 3.35

Let $f_1, g_1\colon e_1 \to e_2$ and $f_2, g_2\colon e_3 \to e_4$ be semantical transformations in *TG*. Moreover, let the entity types e_5, e_6, and e_7 be defined by:

$$type\ e_5 = eq(f_1, g_1)\ [i_1\colon e_5 \to e_1]$$

$$type\ e_6 = eq(f_2, g_2)\ [i_2\colon e_6 \to e_3]$$

$$type\ e_7 = eq(f_1 \times f_2, g_1 \times g_2)\ [i_3\colon e_7 \to e_1 \times e_3]$$

Then $e_7 \equiv e_5 \times e_6$ and $i_3 = i_1 \times i_2$.

In other words this theorem states that the product construction distributes over the equaliser construction.

3.2.5 The power type construction

The fifth construction, the power type construction, occurs in almost any formalism for complex objects. Indeed, it is the construction that was used to extend the relational data model to the so-called NF2-data model. In such models, this construction is often referred to as *grouping*. The reason why we did not introduce this construction earlier is that we need the equaliser for the universal property of this construction.

In set theory, the power set of a set is defined as follows:

DEFINITION 3.36

Let A be a set, its power set $P(A)$ is defined by:

$$P(A) = \{B \mid B \subseteq A\}$$

In terms of the relational model, the power set construction enables an attribute value to be a *set* of values. Such set-valued attributes are e.g. used for teams or sets of data gathered in a particular experiment.

Obviously, there is a function $s: A \to P(A)$; s is simply defined by $s(a) = \{a\}$. Clearly, this function belongs in the type graph; it states that if both the type e and the type $P(e)$ are known, then each e-entity v is also a *set* of e-entities (a $P(e)$-entity), viz. the set containing only v. Spelled out this observation states that an *employee* is also a *set of employees*.

For the following observation, let $f: A \to B$ then f induces a function $f^*: P(A) \to P(B)$ defined by:

$$f^*(\alpha) = \{f(a) \mid a \in \alpha\}$$

As an illustration, let *pers*: *employee* $\to$ *person* represent the 'person'-view on an employee, then $pers^*$: $P(emp) \to P(pers)$ allows a set of *employees* to be seen as a set of *persons*.

These observations lead us to the following definition for power types:

DEFINITION 3.37

Let $<e_1, D_1>$ be an entity type in *TG*. The expression:

$$type\ e_2 = P(e_1)\ [s: e_1 \to e_2]$$

adds the node $<e_2, P(D_1)>$ and the semantical transformation $s: e_1 \to e_2$ to the type graph.

Moreover, if $f: e_1 \to e_2$ is a semantical transformation in the type graph and both $P(e_1)$ and $P(e_2)$ are in *TG*, then so is the semantical transformation: $f^*: P(e_1) \to P(e_2)$.

The mapping from functions f to functions f^* give the following easy result on equivalences of power sets:

LEMMA 3.38

Let A and B be sets with functions $f: A \to B$ and $g: B \to A$ such that $g \circ f = id_A$ and $f \circ g = id_B$, then $g^* \circ f^* = id_{P(A)}$ and $f^* \circ g^* = id_{P(B)}$.

PROOF

Obvious from the definition. □

Given the semantics of the power type construction in section four of this chapter, it is easy to see that this result carries over to entity types. As for the other constructions, the equivalence $P(e_1) \equiv P(e_2)$ if $e_1 \equiv e_2$, is proven using the universal property of this construction. However, we have not yet shown this universal property. The reason for this is two-fold:

1) As with exponentiation, this universal property belongs solely to the semantics of Flock and does not find its way into the syntax.

2) Stating and proving this universal property requires quite some extra work. We considered it appropriate to postpone these complications till after the formal definition of the power type construction.

To state the universal property of power types, we need the *membership*-relation between A and $P(A)$, denoted by ϵ_A. This membership relation can be defined set-theoretically as follows:

DEFINITION 3.39

The membership relation, denoted by ϵ_A, between A and $P(A)$ is defined by:

$$\epsilon_A = \{(U,a) \mid U \in P(A) \wedge a \in U\}.$$

As $\epsilon_A \subseteq P(A) \times A$, we have the canonical injection $\in: \epsilon_A \to P(A) \times A$, defined by $\in(U, a) = (U, a)$. The membership relation ϵ_A has a universal property among all relations in which A participates. To show this universal property, let B be a set, and let R be a relation on $B \times A$, i.e. there is an injective function $r: R \to B \times A$:

1) We can define a function $f: B \to P(A)$ as follows:

$$f_R(b) = \{a \in A \mid (b, a) \in r(R)\}$$

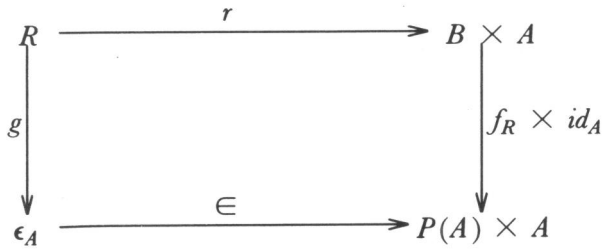
So, $f(b)$ is the, possibly empty, set of all a's related to b by R.

2) The function f_R is easily extended to a function from $B \times A$ to $P(A) \times A$, by 'multiplying' with id_A:

$$f_R \times id_A: B \times A \to P(A) \times A$$

3) Let $c \in R$ with $r(c) = (b, a)$, then $f_R \times id_A(b, a) = (U, a)$ for some $U \subseteq A$. Clearly, $a \in U$ as U is the set of all a's related with B. Hence, there is a $d \in \epsilon_A$ such that $\in(d) = (U, a)$. So, we have a function $g: R \to \epsilon_A$ with $g(c) = d$.

4) From the definitions of the functions, it is easy to see that $\in \circ g = f_R \circ r$. Graphically, we get the following diagram in which the two paths *commute*, i.e. both paths represent the same function:



5) This diagram satisfies a universal property. To illustrate this property, let E be a set, with two functions $k: E \to B \times A$ and $l: E \to \epsilon_A$ such that $f_R \circ k = \in \circ l$. So, for each $e \in E$, $k(e) = (b, a)$ such that $l(e) = f_R(b, a) = (U, a)$. In particular, this implies that if $k(e) = (b, a)$, then a and b are related by R. To see this, suppose that a and B were not related, then $f_R \times id_A(b, a) = (\varnothing, a)$ by the definition of f_R. But then, there is no $d \in \epsilon_A$, such that $\in(d) = f_R(b, a)$ as the empty set has no elements. Hence, if $k(e) = (b, a)$, then a and b are related by R; in other words, there is a $c \in R$ with $r(c) = (b, a)$. So, we can define a function $m: E \to R$, with $m(e) = c$. Moreover, from the definitions of the functions, it is easy to see that $g \circ m = l$ and $r \circ m = k$.

The universal property of ϵ_A we discovered above is a special instance of a construction called the *pullback*. The definition of this construction allows a more elegant formulation of this universal property. Therefore, we give a brief introduction below.

The pullback is not really a new construction; rather, it is the combination of two previously introduced constructions, viz. the product and the equaliser:
Let A, B and C be sets, with functions $f: A \to C$ and $g: B \to C$. As both A and B have a function into C, $A \times B$ will have two functions into C, viz.:

$$f \circ \pi_A, g \circ \pi_B: A \times B \to C$$

As these two functions are parallel, we can construct their equaliser E. A pair (a, b) with $a \in A$ and $b \in B$ belongs to E iff $f(a) = g(b)$. Moreover, if $i: E \to A \times B$ denotes the canonical injection, then $g' = \pi_A \circ i: E \to A$, $f' = \pi_B \circ i: E \to B$ and $f \circ g' = f \circ \pi_A \circ i = g \circ \pi_B \circ ig \circ f'$. The set E with the functions f' and g' is called the pullback of f and g.

The pullback is defined as a combination of the product and the equaliser, this yields a universal property of the pullback, which is also a combination of the universal properties of the product and the equaliser:

LEMMA 3.40

Let A, B, C, and E be sets with functions $f: A \to C$, $g: B \to C$, $f': E \to B$ and $g': E \to A$ such that E with f' and g' is the pullback of f and g. Then for every set X with functions $m: X \to A$ and $n: X \to B$ such that $f \circ m = g \circ n$ there exists a unique function $k: X \to D$ such that $g' \circ k = m$ and $f' \circ k = n$.

PROOF

Note that E is simply the set:

$$E = \{(x, y) \in A \times B \mid f(x) = g(y)\},$$

and $f'((x, y)) = y$ and $g'((x, y)) = x$. Now, X has functions m and n with $m: X \to A$ and $n: X \to B$, hence, by the universal property of the product, there is a unique function $p: X \to A \times B$ such that $\pi_A \circ p = m$ and $\pi_B \circ p = n$. So, as $f \circ m = g \circ n$, we have $f \circ \pi_A \circ p = g \circ \pi_B \circ p$. As E is the equaliser of $f \circ \pi_A$ and $g \circ \pi_B$, we have by the universal property a unique function $k: X \to D$ such that $i \circ k = p$. Hence, we have that: $f' \circ k = \pi_B \circ i \circ k = \pi_B \circ p = n$. Similarly we have that $g' \circ k = m$. □

Using the notion of an equaliser and its universal property, we can now simply state the universal property discovered above:

THEOREM 3.41

Let $m: R \to A \times B$ be an injective function. Then there exists unique functions $f: B \to P(A)$ and $g: R \to \epsilon_A$ such that R with m and g is isomorphic to the pullback of $id_a \times f$ and $\in$.

The proof of this theorem follows the lines of our informal discussion above:

PROOF

First, define $f: B \to P(A)$ and $g: R \to \epsilon_A$ as follows:

$$f(b) = \{a \mid \exists x \in R: m(x) = (b,a)\}$$

$$g(x) = (f(b), \pi_A \circ m(x))$$

Next, $id_a \times f: B \times A \to A \times P(A)$, and thus the pullback of $id_a \times f$ and $\in$ is the set:

$$D = \{(x,y) \in \epsilon_A \times (B \times A) \mid \in(x) = (id_A \times f(y)\} =$$

$$\{((a_1, A), (a_2,b)) \mid a_1 \in A, \wedge (a_1, A) = (a_2, f(b))\}$$

with the functions $m: D \to \epsilon_A$ defined by $m(((a_1, A), (a_2,b))) = (a_1, A)$

and $n: D \rightarrow A \times B$ defined by $n(((a_1, A), (a_2,b))) = (a_2,b)$.
Note that $D = \{((a,A),(a,b)) \mid a \in A \wedge A = f(b)\}$, which in turn implies that $D \equiv \{(a, b) \mid a \in f(b)\} = R$. □

The fact that R is isomorphic to the pullback does not decrease the strength of the universal property, for if there is a unique function to the pullback there is also a unique function to a set isomorphic to the pullback.

Both the introduction of the function $s: e \rightarrow P(e)$ and the step from $f: e_1 \rightarrow e_2$ to $f^*: P(e_1) \rightarrow P(e_2)$ are motivated above by Set-theoretic observations. As indicated before, the semantics of Flock are not set-theoretic but category-theoretic. In these semantics the universal properties of the constructions play an important role. As an illustration of the use of the universal property of ϵ_A, we construct s. The construction of f^* is a bit more complicated and will be done in Section 4:
The function $id_{A \times A}: A \times A \rightarrow A \times A$ is clearly injective. Hence, by the universal property of ϵ_A, there are unique functions $g: A \times A \rightarrow \epsilon_A$ and $f_{id_{A \times A}}: A \rightarrow P(A)$, such that $A \times A$ with g and $id_{A \times A}$ forms the pullback of $f_{id_{A \times A}} \times id_A$ and $\in$. It is easy to see that s as defined above is exactly $f_{id_{A \times A}} \times id_A$.

For the concluding remark of this subsection, note that the introduction of power types in Flock implies that we also have to introduce Set Theory in Flock. Because if entities can be sets of other entities, one expects that say the intersection of two such set-entities can be constructed. The development of Set Theory within Category Theory is the topic of Chapter 5. An important role of this recreation of Set Theory will be played by the membership relation defined above; albeit its categorical definition will be used.

3.2.6 The co-equaliser construction

The sixth, and final, construction is the co-equaliser construction. The co-equaliser is probably the most abstract and the least intuitive of the six. Moreover, it is neither a well-known Set-theoretic construction nor a common construction in complex object formalisms. Therefore, the definition of the co-equaliser both Set-theoretically and the Flock definition is preceded by an informal discussion concerning the usage of the co-equaliser in the construction of *equivalence types*.

As already explained in the first chapter, equivalence types are introduced in Flock so that the database designer does not have to choose between equivalent representations of the same data. In the terminology of the type graph, two equivalent types e_1 and e_2 are two types, for which there are semantical transformations $f: e_1 \rightarrow e_2$ and $g: e_2 \rightarrow e_1$ such that $g \circ f = id_{e_1}$ and $f \circ g = id_{e_2}$. In the equivalence type of e_1 and e_2, denoted by say $[e_1]$, the entities of type e_1 and e_2 that are related by these functions are identified.

To investigate this situation, let A and B be sets, and let $f: A \to B$ and $g: B \to A$ be functions, such that $g \circ f = id_A$ and $f \circ g = id_B$. We now want to define a set C, such that for each $a \in A$, a and $f(a)$ are represented by the same element $[a]$ in C. Note that if a and $f(a)$ are identified, then for all $b \in B$, b and $g(b)$ are also identified.

As usual, this identification can be made using an equivalence relation. As we want to identify elements of A with elements of B, we need an equivalence relation on $A + B$, i.e. the equivalence relation R is a subset of $(A + B) \times (A + B)$. The set we want to construct is now $(A + B)/R$. Informally, R can be defined by saying $(a, b) \in R \Leftrightarrow f(a) = b \Leftrightarrow g(b) = a$. To define R properly, we have to extend f and g to functions f^* and g^* on $A + B$. Set-theoretically, this is done as follows:

$$f^*(x) = \begin{cases} (f(y),1) & \text{if } x = (y, 0) \\ (y, 1) & \text{if } x = (y, 1) \end{cases}$$

Similarly, we can extend g as follows:

$$g^*(y) = \begin{cases} (y, 0) & \text{if } x = (y, 0) \\ (g(x),0) & \text{if } x = (y, 1) \end{cases}$$

As an aside, note that using the universal property of the co-product, these functions can be defined by:

$$f^* = [f, id_B]$$
$$g^* = [id_A, g]$$

To define R, recall that a and $f(a)$ have to be identified, so:

$$(x, y) \in R \Leftrightarrow (x = y \vee f^*(x) = y \vee g^*(x) = y)$$

LEMMA 3.42

R is an equivalence relation.

PROOF

The first condition ensures that R is reflexive. To see that R is symmetric, let $(x, y) \in R$, then:

1) If $x = y$, then clearly $(y, x) \in R$.
2) If $x \neq y$ and $f^*(x) = y$, then $g^*(y) = g^*(f^*(x)) = x$. Hence, $(y, x) \in R$.
3) If $x \neq y$ and $g^*(x) = y$, then $f^*(y) = f^*(g^*(x)) = x$. Hence, $(y, x) \in R$.

Finally, we have to show that R is transitive, therefore, let $(x, y), (y, z) \in R$:

1) If $x = y$ or $x = z$ or $y = z$, then clearly $(x, z) \in R$.

2) Let $x \neq y$ and $x \neq z$ and $y \neq z$ and let $f^*(x) = y$. Then $g^*(y) = x$ and as $x \neq z$, we have $f^*(y) = z$. So, $f^*(f^*(x)) = z$, however, for all $c \in A + B$ we have that $f^*(f^*(c)) = f^*(c)$. Hence, $y = z$, which is a contradiction.

3) Similarly, the assumption that $x \neq y$ and $x \neq z$ and $y \neq z$ and $g^*(x) = y$ leads to a contradiction. □

Note that the reflexivity and the symmetry of R allows us to give a different characterisation of R, viz.:

$$R = \{(c_1, c_2) \mid c_1 = c_2 \vee (\exists d: f^*(d) = c_1 \wedge g^*(d) = c_2) \vee (\exists d: g^*(d) = c_1 \wedge f^*(d) = c_2)\}$$

The set C in which we are interested is now simply the set $(A + B)/R$. So, to construct equivalence types, we have to be able to construct a set modulo some equivalence relation; this is what the co-equaliser will do. The definition of the co-equaliser requires, however, a slightly more general setting than two equivalent sets (types). First of all, note that we had two parallel functions above, viz $f^*, g^*: A + B \to A + B$ and the constructed set $(A + B)/R$. The following observation is that there is a canonical function $h_R: A + B \to (A + B)/R$, with $h_R(c) = [c]$; i.e. h_R maps each element of $A + B$ to its equivalence class under R. So, graphically we have:

$$(A + B) \underset{g^*}{\overset{f^*}{\rightrightarrows}} (A + B) \xrightarrow{h_R} (A + B)/R$$

Note that this is the *dual* picture of the equaliser, i.e. the arrows are 'turned around'. But there is even more, as we have:

LEMMA 3.43
$h_R \circ f^* = h_R \circ g^*$.

PROOF
Let $c \in A + B$, we have to prove that $(f^*(c), g^*(c)) \in R$.

1) If $c = (x, 0)$, then $g^*(c) = c$. So, in this case we have that $f^*(g^*(c)) = f^*(c)$.

2) If $c = (x, 1)$, then $f^*(c) = c$. So, in this case we have that $g^*(f^*(c)) = g^*(c)$.

So, in both cases $(f^*(c), g^*(c)) \in R$. □

So, not only the picture is dualised, but also the equation.

To make the co-equaliser the proper dual of the equaliser, it should work on any two parallel functions rather than on the special f^* and g^* used above:
Let P and Q be two sets, and let $k, l: P \to Q$ be two parallel functions. Similar to the construction above, k and l define a relation r on B:

$$r = \{(q_1, q_2) \mid q_1 = q_2 \vee (\exists p \in P: q_1 = k(p) \wedge q_2 = l(p)) \vee (\exists p \in P: q_1 = l(p) \wedge q_2 = k(p))\}$$

However, in contrast with the construction above, r is not necessarily an equivalence relation. It is reflexive and symmetric, but it does not have to be transitive. Therefore, we construct the following relation:

$$R(x,y) \Leftrightarrow r(x,y) \vee R(y,x) \mid (\exists z \in B: R(x,z) \wedge R(z,y)))$$

Clearly, R is an equivalence relation. Moreover, R is the smallest equivalence relation covering r. Similar as above, let $S = Q/R$ and $m: Q \to S$ be the function $m(q) = [q]$; where $[q]$ denotes the equivalence class of q under R. Graphically, we have:

$$P \underset{l}{\overset{k}{\rightrightarrows}} Q \xrightarrow{m} S = Q/R$$

The construction of S and m as above, is the set-theoretic definition of the co-equaliser construction:

DEFINITION 3.44

Let A and B be sets, and let $f, g: A \to B$ be two parallel functions. The co-equaliser of f and g, denoted by $co\text{-}eq(f, g)$ is defined by the set:

$$co\text{-}eq(f, g) = B/R$$

and the function:

$$h: B \to co\text{-}eq(f, g)$$

With R as defined above.

Before we define the co-equaliser in Flock, we first give the universal property of this construction in Set Theory. It depends on the following simple result:

LEMMA 3.45

Let f, g and h be as above, then $f \circ h = g \circ h$.

PROOF

Let $a \in A$, then $(f(a), g(a)) \in r$. Hence, $(f(a),g(a)) \in R$. So, by definition, $h(f(a)) = [f(a)] = [g(a)] = h(g(a))$. □

So, the co-equaliser is the dual of the equaliser. By now, the dualisation principle should become clear to the reader. Hence, it will not come as a surprise to the reader that the statement of the universal property is simply derived by reversing the functions in the universal property of the equaliser:

THEOREM 3.46

Let A, B, and D be sets, with parallel functions $f, g: A \rightarrow B$, and a function $p: B \rightarrow D$ such that $p \circ f = p \circ g$. Then, there exists a unique function $k: co\text{-}eq(f, g) \rightarrow D$ such that $k \circ h = p$ with h the canonical function.

PROOF

From our 'derivation' of the definition of the co-equaliser, we immediately see that if $p \circ f = p \circ g$ then $p(b_1) = p(b_2)$ if $R(b_1,b_2)$. Hence, p is constant on the equivalence classes of R, and thus p defines a function $k: co\text{-}eq(f, g) \rightarrow D$ with $k([b]) = p(b)$. □

Similar to the universal property of say the equaliser, this universal property gives a natural semantical transformation. Therefore, the universal property is again hard-wired into the definition of the co-equaliser in Flock:

DEFINITION 3.47

Let $f, g: e_1 \rightarrow e_2$ be two parallel semantical transformations in the type graph. The expression:

$$type\ e_3 = co\text{-}eq(f, g)\ [h: e_2 \rightarrow e_3]$$

adds the node $<e_3, D_3>$ with $D_3 = co\text{-}eq(f, g)$ and the canonical function $h: e_2 \rightarrow e_3$ to the type graph.

Moreover, for any node e_4 with a semantical transformation $l: e_2 \rightarrow e_4$ with $l \circ f = l \circ g$ there is a unique semantical transformation $k: e_3 \rightarrow e_4$ such that $k \circ h = l$.

As already mentioned in the beginning of this subsection, the main importance of the co-equaliser construction lies in its use in the construction of equivalence types. Indeed, we see now other intuitive use of equivalence classes in data modelling. Therefore, the reader is referred to Chapter 5 to see examples using the co-equaliser construction.

3.3 The type graph

Now that all the type constructions are defined, we can give the expression to define a type graph in Flock:

DEFINITION 3.48

A basic type graph is defined by the expression:

$$\begin{aligned} &\textit{basic typegraph}(\textit{name}) = \\ &\qquad \textit{nodes} = \{\textit{entity type}\} \\ &\qquad \textit{edges} = \{\textit{semantical transformations}\} \end{aligned}$$

A type graph is defined by a basic type graph, a stratified set of type constructions and a set of user defined functions i.e.:

$$\begin{aligned} &\textit{type graph}(\textit{name}) = \\ &\qquad \textit{basic type graph}(\textit{name}) \\ &\qquad \textit{types} = \{\textit{typeconstructions}\} \\ &\qquad \textit{edges} = \{\textit{semantical transformation} \\ &\qquad\qquad f : e_1 \rightarrow e_2 \; [f = \Lambda]\} \end{aligned}$$

In which Λ denotes the function expression.
Stratification means that we can give a linear order on the entity types, such that if e_1 is used in the construction of e_2, then e_1 is smaller in the order.

As an aside note that to keep the semantics of Flock reasonably simple, we allow only constants and a restricted set of other functions as user defined functions (see the end of this chapter). The second observation we want to make is that the basic type graph consists of at least the entity types 1 and 0.

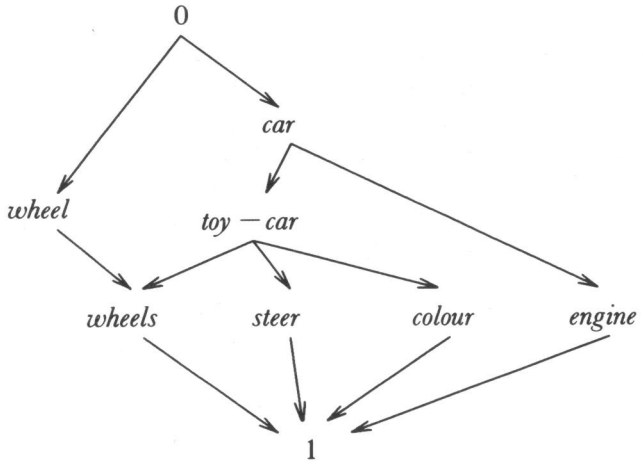
We illustrate this definition of a type graph with an example:
First we define the basic type graph (a bit sloppy as we should give the domains as well):

$$\begin{aligned} &\textit{basic type graph}(B) = \\ &\qquad \textit{nodes} = \{\textit{wheel}, \textit{steer}, \textit{engine}, \textit{colour}, 1, 0\} \\ &\qquad \textit{edges} = \{ \\ &\qquad 0 \rightarrow \textit{wheel}, \; 0 \rightarrow \textit{steer}, \; 0 \rightarrow \textit{engine}, \; 0 \rightarrow \textit{colour}, \\ &\qquad 0 \rightarrow 1, \; \textit{wheel} \rightarrow 1, \; \textit{steer} \rightarrow 1, \; \textit{engine} \rightarrow 1, \\ &\qquad \textit{colour} \rightarrow 1\} \end{aligned}$$

Now we can define the type graph as follows:

$$
\begin{aligned}
&\mathit{typegraph}(TG) = \\
&\qquad \mathit{basic\ type\ graph}(B) \\
&\qquad \mathit{types} = \{ \\
&\qquad \mathit{type\ wheels} = P(\mathit{wheel}) \\
&\qquad \mathit{type\ toy-car} = \mathit{wheels} \times \mathit{steer\ colour} \\
&\qquad \mathit{type\ car} = \mathit{wheels} \times \mathit{steer} \times \mathit{colour} \times \mathit{engine}\}
\end{aligned}
$$

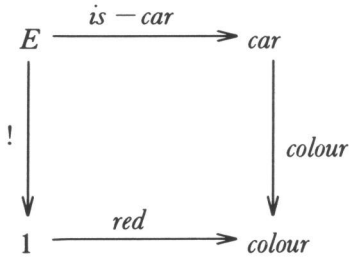
The type graph that results from this definition can be shown graphically as follows; note that we do not draw composed edges separately:



Note that the set of edges in the type graph does not depend on the order of the construction of the various entity types. Moreover, note that in the terminology of the first section of this chapter we can say that *car ISA toy − car*. This can be justified by the observation that each car has *wheels*, *steer*, and a *colour* which are the attributes of a *toy-car*.

Often the class of *red cars* is seen as a specialisation of the class of *car*. Indeed, one would like to construct the type *red cars* from the type *car* such that there is a function *red cars* $\rightarrow$ *car*; this function should map each *red-car* to its associated *car*. In other words, this functions should be the *inclusion function* from *red cars* into *car*. Using a user defined-function in the type graph, we can actually construct the type in Flock:

Let *red*: $1 \rightarrow$ *colour* be the 'constant' function with *red*(*) = *red*. Moreover, let *colour*: *cars* $\rightarrow$ *colour* be the function that assigns a colour to a car. Let E be the *pullback* of *red* and *colour* with functions *is-car*: $E \rightarrow$ *car* and !: $E \rightarrow 1$. Then E is the subset of *car* $\times$ 1 consisting of those elements for which *red* $\circ$! = *colour* $\circ$ *is* − *car*. So, as *car* $\times$ $1 \equiv$ *car*, we have that E is the set of cars for which *red* $\circ$! = *colour* $\circ$ *is* − *car*, i.e. cars which are red. Graphically, we get:



3.4 The semantics

In this section we define the semantics of Flock. These semantics are defined using category theory. This choice has the drawback that although category theory is applied more and more in computer science, see e.g., (Manes (Ed.) (1974); Pitt *et al.* (1985); Pitt, Poigné, and Rydeheard (Eds.) (1987); Rydeheard and Burstall (1988)), it can not yet be considered mainstream Computer Science knowledge. Moreover, there are only few applications of Category Theory in database theory (Maibaum (1977); Rissanen (1977); Sernadas, Sernadas, and Ehrich (1987)). This implies that we have to give an introduction to Category Theory. This introduction is the main part of this section. Clearly, the main topic of this thesis should be Flock and not Category Theory. Therefore, we only introduce the concepts and the results that are needed in the development of Flock. For a real introduction, the interested reader is referred to e.g. S. Mac Lane (Lane (1971)), or R. Goldblatt (Goldblatt (1979)). Moreover, if the proof of a result is rather technical, the reader is often referred to the literature. Indeed, proofs are only given if they help the user in building intuition on the semantics of Flock.

3.4.1 Categories

Obviously, the first definition in an introduction on Category Theory has to be the definition of a *category*. Informally, a category is a collection of objects, together with a collection of arrows between objects that satisfies some conditions. Alternatively, a category can be seen as a directed graph that satisfies some conditions on the edges, the nodes are called objects and the edges are called arrows. To use this alternative definition, we first recall the definition of a directed graph:

DEFINITION 3.49

A directed graph is a 4-tuple (N, E, s, t), in which N is a set of *nodes*, E a set of edges and s and t functions $s: E \to N$ and $t: E \to N$. $s(e)$ is called the *source* of an edge e and $t(e)$ its *target*.

This leads to the following definition of a Category:

DEFINITION 3.50

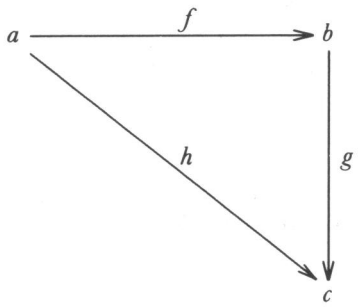
A category C is a directed graph (O, A, s, t), whose nodes are called *C-objects* (the set O) and whose edges are called *C-arrows*, *C-morphisms*, or simply *functions* (the set A), together with an operation assigning to each pair (g, f) of C-arrows with $s(g) = t(f)$, a C-arrow $g \circ f: s(f) \rightarrow t(g)$ called the composite of f and g, and an assignment to each C-object a of a C-arrow $id_a:a \rightarrow a$ called the identity arrow on a, which are such that the following two laws hold:

1 Associative law: Let f, g, h be three C-arrows, such that $t(f) = s(g)$ and $t(g) = s(h)$, then $h \circ (g \circ f) = (h \circ g) \circ f$ and

2 Identity law: Let $f:a \rightarrow b$ be a C-arrow, then $id_b \circ f = f$ and $f \circ id_a = f$.

An example of a category is *SET*, whose objects are sets and whose arrows are set-mappings. *SET* is a large category, its collection of objects (and arrows) is a class not a set. An example of a small category (i.e. the collections of objects and arrows are sets) is *One*, which has one object and one arrow.

We have defined a category as a special kind of directed graph, and indeed many definitions of properties of categories are stated as 'commutating diagrams' and proofs are presented as 'diagram chasing'. A diagram is a graph whose nodes are labelled with objects from the category and whose nodes are labelled with arrows from the category, such that the source and target nodes of an edge are labelled with the source and the target of the arrow that labels the edge. The commutation of a diagram means that two paths through the diagram with the same source and the same target are equal, e.g. the commutation of



means that $g \circ f = h$.

So, a direct benefit of using Category Theory in database design is the graphical visualisation as found in ER-like models is an integral part of the formalism. However, category theory can also be presented in an algebraic fashion. This has the advantage that it is more suitable for manipulation. Therefore, we will use both alternatives in this thesis.

As a first introduction to the way in which concepts are defined in category theory we give the categorical definitions of *surjective*, now called *epic* and injective, now called *monic*. Such definitions are generally made using a *universal property* of the corresponding concept in Set Theory. The universal property of surjectivity and injectivity are given by the following fact:

FACT 3.51
Let B and C be sets, and let $f: B \to C$ be a function. f is surjective iff for any set D and any pair of parallel functions (i.e. functions with the same source and the same category) $g, h: C \to D$, $g \circ f = h \circ f$ implies that $g = h$.

FACT 3.52
Let B and C be sets, and let $f: B \to C$ be a function. f is injective iff for any set A and any pair of parallel functions $g, h: A \to B$, $f \circ g = f \circ h$ implies that $g = h$.

These universal properties are turned into definitions as follows:

DEFINITION 3.53
Let $f: a \to b$ be an arrow between the objects a and b:

1) f is epic, iff for any object c and any pair of parallel functions $g, h: b \to c$, $g \circ f = h \circ f$ implies that $g = h$;
2) f is monic iff for any object c and any pair of parallel functions $g, h: c \to a$, $f \circ g = f \circ h$ implies that $g = h$.

Such definitions by universal property are quite common in category theory. Indeed, this is the prime reason why we proved universal properties for all our constructions. The definition of their categorical counterparts all rely on such universal properties as we will see in the next subsections.

3.4.2 Limits

The categorical analogues of the product and the equaliser of Flock, also called product and equaliser, are special examples of so called *limits*. Limits are defined using *cones*:

DEFINITION 3.54
Let D be a diagram in a category C, a *cone* for D is a C-object c, together with an arrow $f_i: c \to d_i$ for each object d_i in D, such that for every arrow $g: d_i \to d_j$ in D, $g \circ f_i = f_j$.

Consider the following three examples:

1) Let D be the empty diagram, a D-cone is then simply a C-object, as there are no arrows or objects in D.

2) Let D be the diagram consisting of two objects a and b, a D-cone is then a C-object c together with two arrows $f_a\colon c \to a$ and $f_b\colon c \to b$. As D has no arrows, we do not have to worry about commuting paths.

3) A more intricate example is given by the diagram D consisting of two objects a and b and two functions $f\colon a \to b$ and $g\colon a \to b$. Graphically:

$$a \underset{g}{\overset{f}{\rightrightarrows}} b$$

A D-cone is an object c with two functions $f_a\colon c \to a$ and $f_b\colon c \to b$ such that $f \circ f_a = f_b$ and $g \circ f_a = f_b$; i.e. $f \circ f_a = g \circ f_a$. Thus we might as well say that a D-cone is an object c with a function $h\colon c \to a$ such that $f \circ h = g \circ h$. Graphically:

$$c \xrightarrow{h} a \underset{g}{\overset{f}{\rightrightarrows}} b$$

A *limit* of a diagram D is a special kind of cone on D, viz. a cone on D that satisfies a universal property among all cones on D:

DEFINITION 3.55

A D-cone $(c, \{f_i\}_{d_i \in D})$ is a limiting cone, *limit*, for a diagram D iff for each cone $(c', \{f'_i\}_{d_i \in D})$ there is a unique arrow $f\colon c' \to c$ such that $f_i \circ f = f'_i$ for all $d_i \in D$

Continuing the examples given above, we get:

1) A limit for the empty diagram D is a C object c that has a unique arrow to each C-object. So, the terminal object 1 is a limit of the empty diagram.

2) For the diagram D consisting of two objects a and b is a C-object c with two arrows $f_a\colon c \to a$ and $f_b\colon c \to b$ such that for every C-object c' with arrows $f'_a\colon c \to a$ and $f'_b\colon c \to b$ there exists a unique arrow f with $f_a \circ f = f'_a$ and $f_b \circ f = f'_b$. Hence, the product of two sets A and B is a limit of the diagram consisting of A and B.

3) Finally, consider the diagram D consisting of two objects a and b and two functions $f\colon a \to b$ and $g\colon a \to b$. A limit for D is a D-cone, consisting of a C-object c together with an arrow $h\colon c \to a$, such that for each other D-cone, $(c', h'\colon c' \to a)$ there is a unique arrow $k\colon c' \to c$ such that $h \circ k = h'$. So, the equaliser of f and g is a limit of the diagram consisting of f and g.

Note that we say *a* limit, this is done as a diagram may have more than one limit, however, these limits are isomorphic:

PROPOSITION 3.56
If $(c, \{f_i\}_{d_i \in D})$ and $(c', \{f'_i\}_{d_i \in D})$ are both limits for a diagram D, then there exists an isomorphism $f: c \to c'$.

PROOF
Both c and c' are limits of D, hence, there are unique arrows $f: c \to c'$ and $f': c' \to c$ such that the appropriate arrows commute. Hence, $f' \circ f: c \to c$ is the unique mapping from the cone $(c, \{f_i\}_{d_i \in D})$ to the limit $(c, \{f_i\}_{d_i \in D})$. Hence, $f' \circ f = id_c$. Similarly $f \circ f' = id_{c'}$. □

The isomorphisms between two limits of the same diagram make the universal property a strong enough basis for our constructions.

Note that it is not necessary that a limit exists in a category. Therefore, we have to make the existence of the limit part of the definition of the categorical counterparts of our constructions:

DEFINITION 3.57
Let C be a category.

1) C has a terminal object if it has a limit for the empty diagram.
2) C has (binary) products if it has a limit for any diagram consisting of two objects.
3) C has equalisers if it has a limit for any diagram consisting of two parallel arrows.

From the introduction of the constructions in Flock, we now have the following easy consequence:

COROLLARY 3.58
The category *SET* has a terminal object, binary products and equalisers. □

As an aside, note that the fact that limits are determined up to isomorphism allows us to use the set-theoretic definitions of the constructions in the semantics of Flock later in this section.

In section two, we have seen that the canonical injection of the equaliser of two functions is injective in *SET*. Indeed, this function is injective by definition. As an illustration of the use of universal properties in a proof, we show that in a general category, this function is monic:

LEMMA 3.59

Let $i: e \rightarrow A$ be the equaliser of $f,g: a \rightarrow b$, then i is monic.

PROOF

To use the definition of monic, let $j,l: c \rightarrow e$ with $i \circ j = i \circ l$, and define $h: c \rightarrow a$ by $h = i \circ j$. Then we have that:

$$f \circ h = (f \circ i) \circ j = (g \circ i) \circ j = g \circ h$$

Hence by the universal property of equalisers, we have a unique function $k:C \rightarrow E$ such that $i \circ k = h$. But $h = i \circ j$. Hence, $k = j$. Moreover, we also have that $i \circ j = i \circ l$. So, $k = l$ and thus $j = l$. □

Products and equalisers are only special examples of limits. Therefore, it may seem that we artificially restricted the set of constructions supported by Flock. However, products and equalisers are enough to construct all other limits if we are willing to restrict us to limits of finite diagrams. This seems a reasonable assumption as the number of entity types and semantical transformations recognised in a UoD will usually be finite:

THEOREM 3.60

A category C that has a terminal object, binary products, and equalisers has all finite limits.

The proof of this theorem is constructive; i.e. it prescribes an algorithm which constructs limits for finite diagrams. We use the term binary product in the theorem instead of product, as we generalise the product construction to an arbitrary (finite) number of objects:

DEFINITION 3.61

Let D be the diagram consisting of n objects, the limit of D is called the product of these n elements

Note that this definition of $a \times b \times c$ is consistent with our earlier use, in which it meant e.g. $(a \times b) \times c$.

As a first step toward the proof of the theorem, we show that a category having binary products and a terminal object has general products:

LEMMA 3.62

If a category C has a terminal object and binary products, it has arbitrary products.

PROOF

The proof of this lemma is a simple induction proof:

1) If $D = \varnothing$, the limit of D is the terminal object of C.

2) If $D \neq \varnothing$, there is a node d in D, let D' be the diagram $D - \{d\}$. The induction hypothesis is that D' has a limit, consisting of an object a together with arrows $\pi_i: a \to d_i$ for all $d_i \in D$. Let b be the binary product of a and d, together with the arrows $\pi_a: b \to a$ and $\pi_d: b \to d$. Then b together with the arrows $\pi_d: b \to d$ and $\pi_i \circ \pi_d: b \to d_i$ is a cone for D.
To proof that this cone is the limit of D, let $(c, \{f_i\} \cup \{f\})$ be another cone of D, with $f_i: c \to d_i$ and $f: c \to d$. As $(a, \{d_i\})$ is the limit of D', there is an unique arrow $g: c \to a$ such that $\pi_i \circ g = f_i$ for all i. Hence, we have an arrow $c \to a$ and a arrow $c \to d$, but as $(b, \{\pi_a, \pi_d\})$ is the limit of the diagram $\{a,d\}$, there is a unique arrow $h: c \to b$ such that $\pi_a \circ h = g$ and $\pi_d \circ h = f$. Thus, $\pi_i \circ \pi_a \circ h = \pi_i \circ g = f_i$, and we only have to proof that h is unique. But this follows directly from the uniqueness of g and h.

For the rest of the proof of the theorem, we can use finite products and equalisers. In fact, we use the equalisers to account for the edges in the diagram D:

LEMMA 3.63
A category C that has finite products and equalisers has all finite limits.

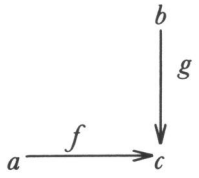
PROOF
This is again a proof by induction, now on the number of arrows in D:

1) Let D be a diagram without edges, then the limit of D is the product of the elements of D.

2) D has at least one edge $g: d_i \to d_j$, let D' be the diagram defined by $D' = D - g$. The induction hypothesis is that D' has a limit $(a, \{f_i\})$ with $f_i: a \to d_i$. Then we have the parallel arrows $f_j, g \circ f_i: a \to d_j$. Let (e,h) be the equaliser of this diagram. Then $(e, \{f_i \circ h\})$ is a cone on D.
To proof that it is a limit, let (b,k_i), with $k_i: b \to d_i)$ be another cone of D. As it is a cone on D, it is certainly a cone on D', thus there exists a unique arrow $l: b \to a$ such that $k_i = f_i \circ l$. Thus in particular $f_j \circ l = g \circ f_i \circ l$. Now (e,h) is defined as the equaliser of $f_j, g \circ f_i: a \to d_j$, and hence there is a unique arrow $m: b \to e$ such that $l = h \circ m$. Thus $k_i = f_i \circ h \circ m$. The uniqueness of m follows from the uniqueness of m and h. □

The combination of the lemmata yields the proof of the theorem.

So, the set of constructions of Flock is rich enough to construct each finite limit. In fact, we have already seen such a constructed limit, viz. the pullback of f and g, which is the limit of:

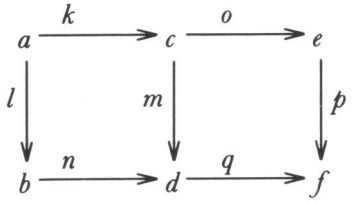


Indeed, the universal property we derived for the Set-theoretic definition of the pullback in Section 2 of this chapter is exactly the universal property of the limit of this diagram.

We have already seen in Section 2 that the pullback plays an important role in the Set-theoretic formulation of the universal property of the power type construction. Later in this section, we shall see that this is also true for the categorical definition of *power objects*. Due to this dependency, we will need some properties of pullbacks to proof properties. We give these properties without a proof, the interested reader is referred to e.g. (Goldblatt (1979)):

PULLBACK LEMMA (3.64)

If a diagram of the form



commutes, then:

1) If the two inner rectangles are pullbacks, then the outer rectangle is a pullback.

2) If the outer rectangle and the right hand rectangle are pullbacks, then the left-hand rectangle is also a pullback.

3.4.3 Co-limits

In section two of this chapter, we have already seen that the notion of a co-products is the dual notion of a product. Similarly, co-equalisers are dual to equalisers. This observation also holds for their categorical counterparts, which are examples of *co-limits*. Not surprisingly, the notion of a co-limit is the dual of the notion of a limit. Therefore, co-limits are defined using *co-cones*:

DEFINITION 3.65

Let D be a diagram in a category C, a *co-cone* for D is a C-object c, together with an arrow f_i: $d_i \rightarrow c$ for each object d_i in D, such that for every arrow $g: d_i \rightarrow d_j$ in D, $f_j \circ g = f_i$.

As examples we dualise the examples we gave for cones:

1) Let D be the empty diagram, a D-co-cone is then simply a C-object, as there are no arrows or objects in D.

2) Let D be the diagram consisting of two objects a and b, a D-co-cone is then a C-object c together with two arrows $f_a: a \to c$ and $f_b: b \to c$.

3) Let D be the diagram consisting of two objects a and b and two functions $f: a \to b$ and $g: a \to b$. A D-co-cone is an object c with two functions $f_a: a \to c$ and $f_b: b \to c$ such that $f_b \circ f = f_a$ and $f_b \circ g = f_a$; i.e. $f_b \circ f = f_b \circ g$. Thus, in analogy with equalisers, we might as well say that a D-co-cone is an object c with a function $h: b \to c$ such that $h \circ f = h \circ g$. Graphically we have:

$$a \underset{g}{\overset{f}{\rightrightarrows}} b \xrightarrow{h} c$$

Now we can define co-limits of a diagram D as co-cones that satisfy a universal property. The nature of this universal property is dual to the nature of the universal properties satisfied by limits. As always in this thesis duality means reversal of the arrows. Therefore, we have the following definition:

DEFINITION 3.66

A D-co-cone $(c, \{f_i\}_{d_i \in D})$ is a co-limiting cone, co-limit, for a diagram D iff for each $(c', \{f'_i\}_{d_i \in D})$ there is a unique arrow $f: c \to c'$ such that $f \circ f'_i = f_i$ for all $d_i \in D$.

Continuing the examples given above (and thus dualising the limit examples):

1) a co-limit for the empty diagram D is a C object c that gets a unique arrow from each C-object. So, the initial object 0 is a limit of the empty diagram.

2) For the diagram D consisting of two objects a and b it is a C-object c with two arrows $f_a: a \to c$ and $f_b: b \to c$ such that for every C-object c' with arrows $f'_a: a \to c$ and $f'_b: b \to c$ there exists a unique arrow $f: c \to c'$ with $f \circ f'_a = f'_a$ and $f \circ f'_b = f_b$. So, the co-product of two objects is a co-limit of the diagram consisting of these two objects.

3) For the diagram D consisting of two objects a and b and two functions $f: a \to b$ and $g: a \to b$. A co-limit for D is a D-co-cone, consisting of a C-object c together with an arrow $h: b \to c$, such that for each other D-co-cone, $(c', h': b \to c')$ there is a unique arrow $k: c \to c'$ such that $k \circ h = h'$. So, the co-equaliser of f and g is a co-limit of the diagram defined by f and g.

As with limits, co-limits are defined uniquely up to isomorphism

PROPOSITION 3.67
If $(c, \{f_i\}_{d_i \in D})$ and $(c', \{f'_i\}_{d_i \in D})$ are both co-limits for a diagram D, then there exists an isomorphism $f : c \to c'$.

PROOF
By the duality principle. □

Again, it is not necessary that a co-limit exists in a category. Therefore, we have to make the existence of the co-limit part of the definition of the categorical counterparts of our constructions:

DEFINITION 3.68
Let C be a category.

1) C has an initial object if it has a co-limit for the empty diagram
2) C has (binary) co-products if it has a co-limit for any diagram consisting of two objects.
3) C has co-equalisers if it has a limit for any diagram consisting of two parallel arrows.

In the previous subsection, we proved that equalisers are monic, by dualising this result, we get that co-equalisers are epic:

COROLLARY 3.69
Let $h : b \to c$ be the co-equaliser of f, $g : a \to b$, then h is epic. □

As with limits, the constructions in section 2 of this chapter give us the following easy consequence:

COROLLARY 3.70
The category *SET* has an initial object, binary co-products and co-equalisers. □

In the previous subsection, we proved that products and equalisers suffice to construct each finite limit. By dualising, we get the analogous result for finite co-limits:

THEOREM 3.71
A category C that has an initial object, binary co-products and co-equalisers has all finite co-limits.

We only give the important steps in the proof, i.e. we define general co-limits

and state the dual of the lemmata. The proof may be constructed by dualising the proofs in the previous subsection:

DEFINITION 3.72
Let D be the diagram consisting of n objects, the co-limit of D is called the co-product of these n elements.

LEMMA 3.73
If a category C has an initial object and binary co-products, it has arbitrary co-products.

LEMMA 3.74
A category C that has finite co-products and co-equalisers has all finite co-limits.

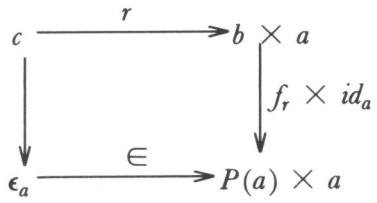
A category C that has all finite limits is called finitely complete, and dually, a category C that has all finite co-limits is called finitely co-complete. A category that is both finitely complete and co-complete is called finitely bi-complete. The results of the last two sections, give us that *SET* is an example of such a finitely bi-complete category.

3.4.4 Exponentiation and power objects

In the previous two subsections, we have seen the categorical counterparts of four of our Flock constructions. In this section, we define the categorical counterparts of our last two constructions, viz. exponentiation and power objects. Other than the previous definitions, these constructions are not examples of a more general construction, rather, their definition is simply the universal property we discovered for the Set-theoretic counterpart. Hence, we get the following definition:

DEFINITION 3.75
Let C be a category, such that C has products

1) C has exponentiation if for any two C-objects a and b there is a C-object b^a and a C-arrow $ev: b^a \times a \to b$ such that for any C-object c and C-arrow $f: c \times a \to b$ there is a unique C-arrow $\hat{f}: c \to b^a$ such that $ev \circ (g \times id_a) = f$;
2) C has power objects, if for any C-object a there are C-objects $P(a)$ and ϵ_a and a C-arrow $\in: \epsilon_a \to p(a) \times a$, such that for any C-objects b and c with a monic C-arrow $r: c \to b \times a$ there is a unique C-arrow $f_r: b \to P(a)$ such that the following diagram is a pullback diagram:



A category that is finitely bi-complete, has exponentiation and has power objects is called a *topos*. So, an easy consequence of the second section is:

COROLLARY 3.76
The category *SET* is a topos. □

Clearly, the constructions defined in Flock imply that any category that is chosen for the semantics of Flock has to be a topos. The reason for this choice is that a *topos* has certain properties which are rather useful for the development of e.g. a query language for Flock as we shall see in chapters five and seven. In fact, we could have achieved this fact with viewer constructions in Flock, as the following theorem is well-known in Category Theory:

THEOREM 3.77
A category C that is finitely complete and has power objects has exponentiation and is finitely co-complete.

PROOF
See (Wraith (1975)).

Note that this theorem implies that our set of constructions in Flock is redundant, as the co-product and the co-equaliser could be expressed in terms of the others. However, we refrained from an over-zealous use of Occam's razor, as it would imply that we would have had a rather unintuitive definition of a type constructor so simple as the co-product. Moreover, to understand this theorem, a profound knowledge of category theory is needed; a so far reaching introduction should surely mention co-limits.

As an aside note that a type graph is always a category but it does not have to be a topos. As the type graph contains e.g. only those products for which a natural class exists in the UoD. Moreover, this observation holds not only for all products, but in general for all finite limits, co-limits, exponentiation and power objects. Therefore, we give the semantics by embedding the type graph in a topos.

Before we turn our attention to these semantics, we still have to show that a function $f:a \rightarrow b$ implies a function $f^*: P(a) \rightarrow P(b)$. As this is a rather complex construction, we perform the construction in *SET* and use some Set-

theoretic arguments:
Let $f: A \to B$ be a set function. Clearly, we want to use the universal property of the power object construction. Hence, we want a monic function $r: C \to P(A) \times B$. Now combining f with the functions: $\in: mo_A \to P(A) \times A$ and $id_{P(A)}$: $P(A) \to P(A)$ yields the function:

$$g = (id_{P(A)} \times f) \circ \in: \epsilon_A \to P(A) \times B$$

As f does not have to be monic (injective), neither has g. However, in Set Theory, we have the following fact:

LEMMA 3.78
Each set-function $k: X \to Z$ can be *factorised* into two functions $l: X \to Y$ and $m: Y \to Z$ such that l is surjective, m is injective and $m \circ l = k$.

PROOF
Let $Y = k(X) \subseteq Z$ and $l = k$ and m is the injection from Y into Z. □

So, let q, r be the surjective/injective split of g above, i.e. r is an injective function $m: C \to P(A) \times B$, where $C = r(\epsilon_A)$. Now we can use the universal property of the power object construction. This yields a function $f_r: P(A) \to P(B)$. To see that f_r is the function f^* we wanted to construct, note that by definition:

$$\begin{aligned} f_r(\alpha) &= \{b \mid (\alpha, b) \in r(C)\} \\ &= \{b \mid \exists\, a \in \alpha: f(a) = b\} \end{aligned}$$

So, we constructed the right function.
To repeat this construction categorically, we need the following fact:

LEMMA 3.79
In any topos C, each C-arrow $f: a \to c$ has an epic/monic factorisation, i.e. there exist C-arrows $g: a \to b$ and $h: b \to c$ with g epic and h monic, such that $h \circ g = f$.

PROOF
See e.g. (Goldblatt (1979)). □

Using this fact, the construction sketched above may be performed in any topos. Hence, in each topos an arrow $f; a \to b$ implies an arrow $f^*: P(a) \to P(b)$.

3.4.5 The semantics

Each construction of Flock has been given an intuitive semantics with their definition. In this section, and more specifically, this subsection we give the formal semantics of the constructions using the categorical concepts introduced above. The reason why we prefer categorical semantics above the intuitive set-

theoretic semantics is that these more abstract semantics are more easily extended to the complex objects of Flock. Other reasons for the choice of category theory were given in Chapter 1.

The semantics are given by embedding the type graph into a topos T; when we have to be more specific on T, we will assume that T is *SET*. The query language of Flock may serve as an example of this situation, it does not have the comparisons $<$ and $>$ because sets do not necessarily have a, partial, order. However, most of the semantics do not rely on a particular topos and so it is straightforward to adopt Flock to, say, partially ordered sets.

The mapping $Sem: TG \rightarrow T$ is built inductively, copying the inductive structure of TG itself. The induction starts with the basic type graph B, by mapping the basic types $<e, D>$ to D and the arrows $f: e_1 \rightarrow e_2$ to the corresponding arrows $f: D_1 \rightarrow D_2$. Note that this induction basis assumes that all the domains of the basic types are objects in the same topos T. In the induction step, we then define $Sem(e_1 \times e_2) = Sem(e_1) \times Sem(e_2)$ et cetera. This gives the following definition for *Sem*:

DEFINITION 3.80

Let TG be a type graph and T a topos, such that the domains of the basic type graphs are objects in T and the arcs in the basic type graph correspond to arrows in T. The *semantics* of TG are given by the functor *Sem*, defined inductively as follows:

1) If $<e, D> \in B$, then $Sem(<e, D>) = D$.
2) If $f: <e_1, D_1> \rightarrow <e_2, D_2> \in B$, then $Sem(f) = (f: D_1 \rightarrow D_2)$
3) Let δ be a diagram in TG, then:

   $Sem(lim\delta) = \lim(Sem(\delta))$

   $Sem(co-lim\delta) = co-lim(Sem(\delta))$

4) For e_1 and e_2 in TG, $Sem(e_1^{e_2}) = Sem(e_1)^{Sem(e_2)}$.
5) For $e \in TG$, $Sem(P(e)) = P(Sem(e))$.
6) For an edge e, $Sem(e)$ is the corresponding arrow in T, e.g. $Sem(f, g) = (Sem(f), Sem(g))$.

So, the constructions in Flock are mapped to their categorical counterparts. This implies in particular that if $T = SET$, the intuitive semantics of each of the constructions denotes the same set as the formal semantics.

An important observation is that 6) implies that each of the user defined functions in the type graph has to be mapped to its categorical counterparts. The only functions we know in T are the constants and functions whose existence can be proven by the topos properties of T. Clearly this limits the use of user defined functions, but at the same time it keeps the semantics reasonable.

The observation above can be rephrased more constructively as follows:

> Each user defined function should first be constructed in T before it can be used.

That this limitation is not as severe as it may appear can be seen in the last section of Chapter 5, where we briefly remark on the construction of primitive recursive functions.

Now that we have formally defined the semantics of the entity types, we can define the formal semantics of an entity. Relative to the type graph, entities are defined as $<$*type*, *value*$>$-pairs. Moreover, set-theoretically, it is assumed that the *value* is an element of the domain of the type. So, to define the semantics of an entity, we have to define an 'element of an object' categorically. This definition is based on the following set-theoretic observation:
For any set A, there is a correspondence between the elements of A and the functions from 1 to A. For each $a \in A$ we can define the function $f_a\colon 1 \to A$, such that $f_a(*) = a$. Moreover, for each function $g\colon 1 \to A$ there is an $a \in A$ such that $g(*) = a$, i.e. $g = f_a$. This gives us the following definition:

DEFINITION 3.81

> Let e be an entity type and $<e, v>$ an entity of type e. The semantics of $<e, v>$, $Sem(<e, v>)$ is given by the function $f\colon 1 \to Sem(e)$ with $f(*) = v$.

Given these semantics, it is easy to see that the following equivalences are indeed true, as claimed in the previous section:

1) $e_1 \times e_2 \equiv e_2 \times e_1$
2) $e_1 \times (e_2 \times e_3) \equiv (e_1 \times e_2) \times e_3$
3) $e_1 \times 1 \equiv e_1$
4) $e_1 + e_2 \equiv e_2 + e_1$
5) $e_1 + (e_2 + e_3) \equiv (e_1 + e_2) + e_3$
6) $e_1 + 0 \equiv e_1$

3.5 Conclusions

This chapter starts with the introduction of the concept of a *type graph*. After giving the motivation for using type graphs rather than a set of types, we show how its directed graph nature can be used to define well known concepts such as specialisation and generalisation. Moreover, exploiting the functional nature of the directed arcs of the type graph, we define the notion of *equivalent* or *synonymous*

types. These concepts are central to *Flock*, as can be seen by requirements *Flock* has to meet.

After the introduction of a type graph, we define how a type graph can be defined in Flock. Usually, in the definition of a language a set of basic types, such as *Bool*, are pre-defined. In the definition of Flock however, we make an exception to this rule. This means that we do not define a basic type graph, but rather let the database designer define this basic type graph. The reason is that in Flock the basic types represent the 'smallest' observable types in a UoD. Clearly, these 'atoms' cannot be defined uniquely over all UoD's. Note that this does not mean that these basic types cannot be complex types in some other language, but that these complex types cannot be destructed within the formalism.

Flock is provided with six type constructors, viz.:

1) Products; this construction can be likened to the array construction in conventional languages.

2) Co-products; this construction can be likened to the disjoint union provided in some languages.

3) Exponents; this constructs a function type. It will proof to be instrumental in the definition of methods in Flock.

4) Equalisers; this construction does not have a counterpart in conventional languages. Its interpretation can best be sketched as a selection.

5) Power types; this is the usual construction that allows attributes to be tables in NF^2 data models.

6) Co-equalisers; this is perhaps the most difficult operator. Its main use lies in the fact that it allows us to construct *equivalence types*. Hence, it is instrumental to meet our requirements.

Note that these constructions do not only add nodes to the type-graph, but may also add new arcs.

All the constructions are introduced accompanied by their set-theoretic interpretation. These set-theoretic interpretations can be used as the informal semantics of the constructions. To facilitate the step towards the formal semantics of the constructions, a universal property is proven for each of the constructors. Moreover, to introduce the reader to categorical reasoning, these universal properties are used to proof simple theorems. Most of these theorems pertain to type equivalence.

This chapter ends with a short introduction to category theory. Which is used to define the formal semantics of the type constructions of Flock.

Chapter 4

Algebraic Dependencies

In relational database theory, an extension of a relation is a subset of the domain D of a relation, i.e. a member of $P(D)$. A dependency on a relation determines a subset X of $P(D)$, the elements of X are those extensions that satisfy the dependency (also called valid extensions). Sets of dependencies can thus be used to exclude unnatural extensions from the database. In this sense, dependencies add semantical information to a relation scheme; in fact, this is the primary means to add semantics to a relational database.

As dependencies can be seen as rules to exclude extensions that can not represent a state of the UoD, they can be seen as semantical additions to the type information.

For example, let R be a relation schema over $\{A_1, A_2\}$. The functional dependency $A_1 \rightarrow A_2$ implies that the domain of R is not really $P(D_{A_1} \times D_{A_2})$, but rather some subset $X \subseteq P(D_{A_1} \times D_{A_2})$ such that X contains exactly those elements that satisfy the functional dependency.

With this interpretation of dependencies, it becomes an interesting question whether all the relational types (i.e. relation schemas with sets of dependencies) can be constructed using Flock's type constructions. That is, let M be a set of dependencies on a relation schema R, can we construct a type τ, such that the elements of τ are exactly the valid extensions of R? This question is the topic of this chapter. The function of this chapter in the thesis is twofold:

1) It illustrates the use of the type constructions of Flock.

2) If the answer to the central question is affirmative, then we show that the 'semantical' expressivity of Flock is strictly larger than that of the relational model; a claim that is often made with regard to object oriented database systems.

In the first section of this chapter, we show that this question can indeed be answered affirmatively for *algebraic dependencies*. In the construction, only the universal properties of the type constructions of Flock are used.

Although the constructions of the first section translate 'relational semantics by dependencies' to Flock type constructions, it is no natural translation. For example, the elements of the translations would suffer from similar update anomalies as their relational counterparts. In the second section, we show how a restricted class of dependencies may be translated such that the Flock types capture the intuitive semantics of these dependencies. These translations will play an important role in the introduction of dependencies in Flock in Chapter 10.

In the final section of this chapter, we summarise and give conclusions.

4.1 Algebraic dependencies and types

In this section, we show that we can translate a relation schema with dependencies to an equivalent Flock type. The construction of the equivalent Flock type is spread over three subsections. In the first subsection, we give a better formulation of the problem in terms of algebraic dependencies after some preliminary definitions. Moreover, we show how the problem can be solved under the assumption that each project-join expression can be seen as a function between two Flock types. The next two subsections are used to show that for each project-join expression, there is an equivalent function implied by the universal properties of the Flock constructions.

4.1.1 Problem and solution under assumption

The problem we want to solve in this chapter is if we can translate a relation schema with dependencies to a Flock type. To answer this question, we should of course specify the class of dependencies we are willing to consider. Clearly, the larger this class, the more interesting the translation becomes. Therefore, we define the class of dependencies to be the class of *algebraic dependencies*. This is the largest reasonable class of dependencies as it coincides with the class of domain-independent dependencies, c.f., (Yannakakis and Papadimitriou (1981)). Before we can define this class, we need a preliminary definition:

DEFINITION 4.1

Let R be a relation-schema with attributes $a(R) = \{A, B, \ldots, Z\}$, then $\overline{R}$ is a relation-schema with

$$a(\overline{R}) = \{A_1, B_1, \ldots, Z_1, A_2, B_2, \ldots, Z_2, \cdots\}.$$

Moreover, for a relation $\overline{r}$ over $\overline{R}$, there is a relation r over r such that

$$\overline{r} = \{(t; t; \cdots) | t \in r\}.$$

So, a tuple in $\bar{R}$ consists of an infinite number of copies of *the same* tuple in R. Using the $\bar{R}$ notation, we can define algebraic dependencies as follows:

DEFINITION 4.2

An algebraic dependency over R is an expression:

$$\phi_1(\bar{R}) \subseteq \phi_2(\bar{R})$$

with the ϕ_i project-join expressions.

Examples of such algebraic dependencies are, of course, provided by the usual 'tuple-generating' dependencies such as multivalued dependencies and join dependencies. A more interesting example is probably provided by the well-known functional dependency $A \rightarrow B$ which is expressed algebraically as

$$\pi_{A_1B_1}(\bar{R}) \bowtie \pi_{A_1B_2}(\bar{R}) \subseteq \pi_{A_1B_1B_2}(\bar{R})$$

The following step we have to make before we can restate the problem is to define *relations* and *extensions* in Flock:
Commonly, a relation R is defined over a set of attributes, say $\{A_1, \ldots, A_n\}$, and the domain of R is defined by $D_R = D_{A_1} \times \cdots \times D_{A_n}$. We do not have attributes in our model, but we do have the Cartesian product, hence we can define relation types as follows:

DEFINITION 4.3

An entity type e is a *relation type* if $e = e_1 \times \cdots \times e_n$.

Note that to mimic the relational data model faithfully, we should require that the e_i are basic entity types. However, this restriction plays no role in our constructions, and therefore we refrain from making it. To stay close to the relational notation, we will often use names like A fot the 'attributes' of the 'relation. Moreover, we will often refer to the relation type $r = A_1 \times \cdots \times A_n$ by the name $A_1 \cdots A_n$.
An extension of a relation is commonly seen as a subset of the domain of the relation; hence we can define:

DEFINITION 4.4

Let r be a relation type, then $P(r)$ is called an *extension* type. An extension R of r is a member of $P(r)$. †

So, the set of all extensions of r satisfying some dependency is a subset of $P(r)$. Such subsets can be denoted by injective functions: $i: E \rightarrow P(r)$; it is this functional characterisation of subsets we will use in the rest of this chapter. The

†Note that we use r to denote a relation type rather than R which was used to denote a relation schema. This choice was made to keep close the conventions on type names. Therefore, we use R for extensions.

problem can now be rephrased as follows:

> Given a relation schema r and a set of algebraic dependencies D on r, can we construct the subset S of $P(r)$ such that:
> $$x \in S \Leftrightarrow x \text{ satisfies all } d \in D$$

To answer this question, note that in 'relational' terms, each project join operation transforms relations over some schema r_1 into relations over a schema r_2. For example, let $r_1 = ABCD$ be a relation schema and $\phi = \pi_{AB}(R) \bowtie \pi_{BC}(R)$ a project-join expression, then a relation R over r_1 is transformed into the relation $\pi_{AB}(R) \bowtie \pi_{BC}(R)$ over $r_2 = ABC$. This means that ϕ can be seen as a function $\phi\colon P(r_1) \to P(r_2)$. In general, each project join expression can of course be seen as such a function. This observation will prove to be crucial in the construction of the required type. However, there is one small caveat: in algebraic dependencies the project-join expressions are defined on $\overline{r}$, rather than on r. Clearly, $\overline{r}$ is not an Flock type, as it is defined using an infinite product and Flock only knows finite products. Still, in the rest of this subsection, we assume that the project-join expressions from algebraic dependencies can be seen as functions $P(r_1) \to P(r_2)$ in which both $P(r_1)$ and $P(r_2)$ are Flock types. In the next subsection, we will show that finite products indeed suffice, and in the third subsection, we show how ϕ as a function is implied by universal properties.

Let d be an algebraic dependency on the relation type (schema) r_1, i.e. $d = \phi_1(r_1) \subseteq \phi_2(r_1)$. Because of the subset requirement, both project-join expressions considered as functions have to have the same target, i.e. $\phi_1, \phi_2\colon P(r_1) \to P(r_2)$ for some relation type r_2. If the dependency d would be of the form $d = \phi_1(r_1) = \phi_2(r_1)$, then clearly the equaliser of ϕ_1 and ϕ_2, as functions, would give the subtype of $P(r_1)$ of all extensions satisfying the dependency. However, in general a subset requirement is required to hold, and therefore we have to make a somewhat more complicated construction:

Let $f, g\colon P(e) \to P(d)$, we want to have the subtype of $P(e)$ consisting of all elements for which $f(x) \subseteq g(x)$. Now, the statement $f(x) \subseteq g(x)$ implies that $f(x) = g(x) \cap f(x)$. So, if the intersection can be seen as a function $\cap_{P(d)}\colon P(d) \times P(d) \to P(d)$, we want those x's for which $f(x) = \cap_{P(d)}(f(x), g(x))$, which is an application of the equaliser.

The function $\cap_{P(d)}$ can be constructed using the universal property of the power type construction. For $\Theta, \Upsilon \in P(d)$:

$$\cap_{P(d)}(\Theta, \Upsilon) = \{\theta \mid \theta \in \Theta \wedge \theta \in \Upsilon\}.$$

Now, consider $\epsilon_d \times \epsilon_d$, this type has the injective function:

$$\in \times \in : \epsilon_d \times \epsilon_d \to P(d) \times d \times P(d) \times d.$$

Moreover, $(\Theta, \theta, \Upsilon, \upsilon) \in \in \times \in (\epsilon_d \times \epsilon_d)$ iff $\theta \in \Theta$ and $\upsilon \in \Upsilon$. Let π_1 and π_2 denote the two projections from $P(d) \times d \times P(d) \times d$ to d. Now, let $i\colon E \to \epsilon_d \times \epsilon_d$ be the equaliser of $\pi_1 \circ (\in \times \in)$ and $\pi_2 \circ (\in \times \in)$. Then E

'is' the set of $(\Theta, \theta, \Upsilon, \upsilon)$ tuples for which $\theta = \upsilon$. Moreover, $(\in \times \in) \circ i$ is an injective function. In fact, we have an injective function $j: E \to P(d) \times P(d) \times d$ as both d-components in an E-element are the same. Hence, the universal property of power types give us a function: $\cap_{P(d)}: P(d) \times P(d) \to P(d)$ with:

$$\begin{aligned} \cap_{P(d)}(\Theta, \Upsilon) &= \{\theta \mid (\Theta, \Upsilon, \theta) \in E\} \\ &= \{\theta \mid \theta \in \Theta \wedge \theta \in \Upsilon\} \end{aligned}$$

Hence, $\cap_{P(d)}$ is the intersection function.

Summarising, we have the following construction:

1) We start with the functions: $\phi_1, \phi_2: P(r_1) \to P(r_2)$;
2) these imply the function: $(\phi_1, \phi_2): P(r_1) \to P(r_2) \times P(r_2)$;
3) composition with the intersection function on $P(r_2)$ yields the function: $\cap_{P(r_2)} \circ (\phi_1, \phi_2): P(r_1) \to P(r_2)$;
4) we have seen above that elements of $P(r_2)$ satisfy the dependency d if $\phi_1(x) = \phi_1(x) \cap \phi_2(x)$. Hence, the type we are interested in is the equaliser of these two functions:

$$E = eq(\phi_1, \cap_{P(r_2)} \circ (\phi_1, \phi_2)).$$

Hence, given one algebraic dependency d on r, we can construct the subtype of $P(r)$ of all extensions satisfying d. The following step is a subtype of $P(r)$ of the extensions satisfying several dependencies:

Let d_1 and d_2 be two algebraic dependencies on r and let E_1 and E_2 be the subtypes of r_1 satisfying d_1 and d_2 respectively. Clearly the intersection of E_1 and E_2 consists of all extensions satisfying both d_1 and d_2. However, we cannot use $\cap_{P(r)}$ to construct this intersection as $i_1: E_1 \to P(r)$ and $i_2: E_2 \to P(r)$ have a different source. Therefore, we need another construction to get this intersection:

Let $\Theta \in E_1$ and $\Upsilon \in E_2$, Θ and Υ 'are' the same extension of r if $i_1(\Theta) = i_2(\Upsilon)$. So, let P be the pullback of i_1 and i_2, then P consists of pairs (Θ, Υ) for which $i_1(\Theta) = i_2(\Upsilon)$. So, the image of P under $i_1 \circ \pi_1$ is the intersection of E_1 and E_2. Hence, if we construct the subtype E_3 as the epic/monic factorisation of $i_1 \circ \pi_1$, then E_3 is the subtype of $P(r)$ of all extensions satisfying both d_1 and d_2.

We have already seen that the intersection function $\cap_{P(r)}$ can be constructed using universal properties, so to show that each type as constructed above can be constructed using only universal properties we only have to show how to construct the ϕ_i functions with these properties. This is the subject of the following two subsections.

4.1.2 Finite products suffice

In the previous subsection we made the assumption that an algebraic dependency was defined on r rather than on $\bar{r}$. The reason for this is clearly that $\bar{r}$ is a subtype of the *infinite product* of r with itself and we can only construct finite products. However, this is not a serious drawback, as in each project-join expression only a finite number of a attributes from this infinite relation schema will be used. Hence, for each algebraic dependency a finite relation schema will suffice. We can make this more precise using the set of attributes of $\bar{r}$ used in a project-join expression; we denote this set by $s(\phi, r)$. The definition of $s(\phi, r)$ is as follows:

DEFINITION 4.5

Let r be a relation schema with $a(r) = \{A, B, \ldots, Z\}$ and let ϕ be a project-join expression defined on $\bar{r}$, then the set of source attributes of ϕ, denoted by $s(\phi, r)$ is defined inductively by:

1) If $\phi(\bar{R}) = \bar{R}$, then $s(\phi, r) = \varnothing$.

2) If $\phi(\bar{R}) = \pi_{A_1 \cdots A_{n_A} \cdots Z_1 \cdots Z_{n_Z}}(\phi_1(\bar{R}))$, then
$s(\phi) = \{A_1 \cdots A_{n_A} \cdots Z_1 \cdots Z_{n_Z}\} \cup s(\phi_1, r)$.

3) If $\phi(\bar{R}) = \phi_1(\bar{R}) \bowtie \phi_2(\bar{R})$, then $s(\phi) = s(\phi_1, r) \cup s(\phi_2, r)$.

As usual, a set of attributes is a relation schema, so we may consider $s(\phi, r)$ to be a relation type. Moreover, each extension R of r determines a relation by doubling the attribute values of a tuple as often as needed:
Let $t = (t_A, \ldots, t_Z) \in R$ then $s(\phi, r)(t)$ is the tuple with $s(\phi, r)(t)_{A_i} = t_A, \ldots, s(\phi, r)(t)_{Z_j} = t_A$.
As an example, let $r = AB$ and $s(\phi, r) = A_1A_2B_1$ then the image of a tuple $t = (t_A, t_B)$ is the tuple $s(\phi, r)(t) = (t_A, t_A, t_B)$. By the definition of power types, the function $s(\phi, r)$ implies a function: $s(\phi, r)^*: P(r) \to P(s(\phi, r))$.

Note that the expression ϕ can be applied to $s(\phi, r)(R)$, as all the attributes needed to calculate ϕ are present in the schema $s(\phi, r)$. Given this observation, we have the following simple but important lemma:

LEMMA 4.6

Let r be a relation schema and $d \equiv \phi_1(\bar{r}) \subseteq \phi_2(\bar{r})$ an algebraic dependency on r. A relation R satisfies d iff :

$$\phi_1(s(\phi_1, r)(R)) \subseteq \phi_2(s(\phi_2, r)(R)).$$

PROOF

Clearly, $\phi_i(\bar{R}) = \phi_i(s(\phi_i, r)(R))$. Hence the lemma holds. □

This lemma ultimately implies that we do not need an infinite product, as observed above, but that the finite product over $s(\phi_i, r)$ suffices. Therefore, we will use this relation type as the source type of a project-join expression ϕ:

DEFINITION 4.7

Let ϕ be a project-join expression defined on a relation type r, then the source of ϕ is the relation type $s(\phi, r)$.

The target type of a project-join expression is much easier to define, it is simply the subset of the source type specified by the 'outer' projections and joins. For example, let ϕ be the expression:

$$\phi = \pi_{AB}(\phi_1(\overline{r})) \bowtie \pi_{BC}(\phi_2(\overline{r}))$$

then the target type of ϕ is ABC. So, we get the following definition:

DEFINITION 4.8

Let ϕ be a projection expression on a relation schema r, the target type of ϕ, denoted by $t(\phi, r)$ is defined inductively by:

1) If $\phi = \pi_{A_1 \cdots A_n}(\phi_1(\overline{R}))$, then $t(\phi, r) = A_1 \cdots A_n$

2) If $\phi = \phi_1(r) \bowtie \phi_2(r)$, then $t(\phi, r) = t(\phi_1, r) \cup t(\phi_2, r)$.

The following lemma shows that the name *target type* is not misleading:

LEMMA 4.9

Let t be a tuple over r, then $\phi(\overline{t})$ is a tuple over $t(\phi, r)$.

PROOF

Let $\phi(\overline{t})$ be a tuple over the relation type e. Clearly, the attribute set of e has to include the attribute set of $t(\phi, r)$ as ϕ supplies a value for these attributes. If K_i is an attribute of $\overline{r}$ which is not in $t(\phi, r)$, ϕ may use values for K_i internally, but these values are projected out along the computation. □

Combining the source and the target type of a project-join expression ϕ, we get the following lemma:

LEMMA 4.10

A project-join expression ϕ on a relation type $\overline{r}$ can be seen as a function ϕ: $P(s(\phi, r)) \rightarrow P(t(\phi, r))$.

PROOF

An element of $P(s(\phi, r))$ is an extension over $s(\phi, r)$. We have already seen

that ϕ can be applied to extensions of this relation type. Moreover, a tuple of type $s(\phi, r)$, will be mapped to a tuple over $t(\phi, r)$ by ϕ. □

The function ϕ of this lemma is not yet of the right form to allow us to apply the construction of the previous subsection. To apply this construction, we have to compose ϕ with the function $s(\phi, r)^*$ yields the function:

$$\Phi = \phi \circ s(\phi, r)^* : P(r) \rightarrow P(t(\phi, r))$$

In the remainder of this section, we show how the function $s(\phi, r)^*$ can be constructed from the universal properties, and in the next subsection, we show how the function ϕ can be constructed.

In fact, we only have to show how the function $s(\phi, r)$ can be constructed. From the set-theoretic definition of this function, it should be clear that it is enough if we show how one of the attributes of a relation r can be doubled.

Let $r = ABC$ be a relation type. By the universal property of the product construction, there is a unique function $(id_A, id_A) = \Delta : A \rightarrow A \times A$ with $\pi_1 \circ \Delta = id_A$ and $\pi_2 \circ \Delta = id_A$. So, the function Δ maps a value a to the pair (a, a). Again by the universal property of the product, we get:

$$(\Delta, id_B, id_C) : ABC \rightarrow AABC$$

By definition, this function satisfies our requirements. Hence, each $s(\phi, r)$-function can be constructed. So, by the universal property of the power type construction, we get the function $s(\phi, r)^*$.

As a final remark note, that given the function $s(\phi, r)^*$, we only have to create the function $\phi : P(r_1) \rightarrow P(r_2)$ as we do not have to consider infinite products. Moreover, for the construction of the function ϕ it does not matter whether two attributes always have the same value. This construction is the subject of the next subsection.

4.1.3 Project join expressions as functions

The final step in the construction of a type equivalent with a relation schema with a set of algebraic dependencies, we have to construct the function $\phi : P(s(\phi, r)) \rightarrow P(t(\phi, r))$. Each such ϕ is built from projections and joins. So, if we can find functions for these operators, we are ready.

The projection operator is the easiest of the two. For if $r = A \times B$, then by definition we have the projection functions $\pi_A : r \rightarrow A$ and $\pi_B : r \rightarrow B$. Moreover, we have already seen that these functions induce the functions $\pi_A^* : P(r) \rightarrow P(A)$ and $\pi_B^* : P(r) \rightarrow P(B)$. Clearly, these last two functions are the *relational* projections we need in our translation.

For the join, there are three cases depending on the overlap in the attribute sets on both sides of the join:

1) $AB \bowtie CD$;

2) $AB \bowtie BC$;

3) $AB \bowtie B$.

In the remainder of this subsection, we will analyse these three cases. Note that there are in fact only two cases as item 3 is a special case of item 2, we only added it for clarity.

For case 1), we need a function $f: P(AB) \times P(CD) \to P(ABCD)$ with:

$$f(\Theta, \Upsilon) = \{(a, b, c, d) \mid (a, b) \in \Theta \wedge (c, d) \in \Upsilon\}.$$

The universal property of ϵ_{ABCD} gives us this function if we have a relation R on $P(AB) \times P(CD) \times ABCD$ with:

$$(\Theta, \Upsilon, a, b, c, d) \in R \Leftrightarrow (a, b) \in \Theta \wedge (c, d) \in \Upsilon.$$

Now $(a, b) \in \Theta$ is encoded in ϵ_{AB} and $(c, d) \in \Upsilon$ is encoded in ϵ_{CD}. Hence the relation R is simply given by the monic function:

$$\in \times \in : \epsilon_{AB} \times \epsilon_{CD} \to P(AB) \times P(CD) \times ABCD.$$

For case 2), we need a function $f: P(AB) \times P(BC) \to P(ABC)$ with

$$f(\Theta, \Upsilon) = \{(a, b, c) \mid (a, b) \in \Theta \wedge (b, c) \in \Upsilon\}.$$

As above, we want to get this function from the universal property of ϵ_{ABC}. So, we need a relation R on $P(AB) \times P(BC) \times ABC$ with:

$$(\Theta, \Upsilon, a, b, c) \in R \Leftrightarrow (a, b) \in \Theta \wedge (b, c) \in \Upsilon.$$

Again, $(a, b) \in \Theta$ is encoded in ϵ_{AB} and $(b, c) \in \Upsilon$ is encoded in ϵ_{BC}. But now the function,

$$\in \times \in : \epsilon_{AB} \times \epsilon_{BC} \to P(AB) \times P(BC) \times AB \times BC$$

does not yet yield this relation, as the two B-components have to be equal. Moreover, we need a relation on $P(AB) \times P(BC) \times ABC$ rather than on $P(AB) \times P(BC) \times AB \times BC$. The first of these problems is solved by taking the equaliser of the two projections on B, i.e. let

$$\begin{aligned} i: E &\to \epsilon_{AB} \times \epsilon_{BC} \\ &= eq(\pi_{1,B} \circ \in \times \in, \pi_{2,B} \circ \in \times \in) \end{aligned}$$

Then, for each element of E, the two B-components coincide. Moreover, set-theoretically, it is obvious that E has an injection into $P(AB) \times P(BC) \times ABC$. However, to keep this section within our categorical semantics we have to be a bit more careful. Therefore, let g be the function:

$$g = \pi_{1,\, P(AB) \times P(BC) \times ABC} \circ \in \times \in \circ i$$

Moreover, let the pair $g_1: E \to R$ and $g_2: R \to P(AB) \times P(BC) \times ABC$ be

the epic/monic factorisation of g. Then $g_2: R \to P(AB) \times P(BC) \times ABC$ denotes the relation:

$$(\Theta, \Upsilon, a, b, c) \in g_2(R) \Leftrightarrow (a, b) \in \Theta \wedge (b, c) \in \Upsilon.$$

Hence, f is the function implied by the universal property of ϵ_{ABC} and g_2.

As already said above, the third case is simply a special case of the second the construction of f in second case can be transformed to a construction of f in the third case by removing all C's in the construction.

Now that we know how to compute both joins and projections, each project-join expression on finite relations can be computed. Moreover, given the previous subsection, this means that we can compute each project-join expression used in an algebraic dependency as a function ϕ: $P(r_1) \to P(r_2)$. But than the results of the first subsection give us that for each relation schema r with a set of algebraic dependencies D an Flock type E can be constructed such that E consists of all extensions of r that satisfy D:

THEOREM 4.11

Let r be a relation schema with a set of algebraic dependencies D on r. Then there exists an Flock type E such that E consists of all extensions of r that satisfy D. Moreover, all user-defined functions used in the construction of such a type are implied by the universal properties of the type constructions. □

This theorem is the solution to the problem posed at the beginning of this chapter. Although the answer is affirmative, it is not very satisfying. For the solution thus offered for translating 'relational semantics' to complex object semantics is conceptually flawed, as we have neither that:

1) entities can be seen as elements of an extension, as manipulation with such elements will in general not result in a valid extension. As an example, in the extension $\{(a, b_1, c_1), (a, b_1, c_2), (a, b_2, c_1), (a, b_2, c_2)\}$ for the relation type $R = ABC$ with the multivalued dependency $A \to\to B \mid C$, we may not delete or modify one of the four tuples, and neither may we insert only the tuple (a, b_1, c_3).

2) nor that valid extensions are the conceptual level for manipulating the extension of such a relation. Continuing our example above, another valid extension of R is $\{(a, b_2, c_2), (a, b_2, c_3), (a, b_3, c_2), (a, b_3, c_3)\}$. However, the union of these two extensions is not a valid extension.

In the next section, we will show that for a limited set of dependencies a more natural translation may exists.

4.2 A better translation

In the previous section, we have seen that relational real world semantics can be translated to Flock real world semantics. However, we have also seen that this translation is not very helpful. In this section, we show that a better translation exists for a restricted set of dependencies iff we have a satisfying solution for *functional dependencies*. The Flock solution for functional dependencies is only given in Chapter 10, when we return to constraints in Flock. So, for the remainder of this section, we simply assume that such a translation exists.

This section is divided into three subsections. In the first subsection, we sketch a translation for non-anomalous sets of multivalued dependencies or, equivalently, for acyclic join dependencies. Moreover, we show that this translation cannot work for general sets of multivalued dependencies or for general join dependencies. In the next subsection, we prove that our method sketched in the first subsection is correct. In the third and final subsection, we briefly look at possible generalisations of our method.

4.2.1 The translation and its intuition

In applications, the multivalued dependency is often used to model sets in the relational model. Recall that an mvd $A \rightarrow\rightarrow B \mid C$ on a relation $r = ABC$ can be understood as saying that in an extension R of r there is a subset B_a of B and one subset C_a of C associated with each $a \in \pi^*_A(R)$. Moreover, the set $\{(a, b_i, c_j) \mid b_i \in B_a \wedge c_j \in C_a\}$ equals the set $\pi^*_{BC}(\sigma_{A=a}(R))$. So, an extension of ABC can be seen as an extension of $AP(B)P(C)$. Moreover, with each a-value, exactly one $P(B)$-value and one $P(C)$-value is associated, i.e. we have the *functional dependencies* $A \rightarrow P(B)$ and $A \rightarrow P(C)$. As said in the introduction of this subsection, we assume that we can express such functional dependencies in Flock. To stress this assumption, we will use the notation $P(A\{B\}_A\{C\}_A)$ as the type of all elements of $P(AP(B)P(C))$ that satisfy these dependencies.

It should be clear that such a type represents the extensions of R for which the multi-valued dependency holds. Such a representation of an extension satisfying a multivalued dependency is in fact a decomposition of the extension based on a structure that is recognised. One could argue that such a representation is in fact better than the one we constructed in the previous subsection as it conveys more semantics. Moreover, it helps in understanding which actions on the database will be sound in respect to the dependencies. However, this approach works fine for multivalued dependencies and more generally acyclic join dependencies, but breaks down for general sets of multivalued dependencies and acyclic join dependencies.

To explain how this method works and why this 'breakdown' occurs, we have to determine what a 'good structure', i.e. a structure allowing such a decomposition is (Hodges (1987)):
Ideally, larger extensions are built from the smaller ones by adding pieces of

certain fixed kinds; then extensions are determined up to isomorphism by how many pieces of each kind were added. In a relation without dependencies, there is one kind of pieces, viz. tuples. If the relation includes, however, a multivalued dependency, then the insertion of one tuple potentially generates a set of tuples that should also be included to satisfy the dependency.

Thus, we would like to decompose the extension in (maximal) independent sets, under a suitable notion of independence, such that insertion leads to either:

a) adding elements to an independent set;

b) adding a new independent set;

c) or both.

Thus, 'independence' should imply that if two sets X_1 and X_2 are independent, an insertion in X_1 neither relies on X_2 nor does it affect X_2; i.e. X_1, X_2, and $X_1 \cup X_2$ are valid extensions and when an update in the X_1 part of $X_1 \cup X_2$ is performed, the validness of the new extension $X_1 \cup X_2$ is independent of its X_2 part. Moreover, let $Ext_1(e)$ and $Ext_2(e)$ be two extensions of e such that $Ext_1(e) \subseteq Ext_2(e)$ and X_1 and X_2 are two independent components in E_1. Then there are also two independent components Y_1 and Y_2 in $Ext_2(e)$ such that $X_1 \subseteq Y_1$ and $X_2 \subseteq Y_2$.

This notion of a good structure is the basis for the alternative representation of extensions satisfying a multivalued dependency:
Let $r = \{ABC\}$ with the multivalued dependency $A \twoheadrightarrow B$ and two valid extensions of r called R_1 and R_2. Then $R_1 \cup R_2$ is a valid extension of r iff $\pi_A(R_1) \cap \pi_A(R_2) = \varnothing$. So, an independent component of an extension of R has exactly one A value. Moreover, such an A value is uniquely associated with a set of B values that does not depend on the C values.

This decomposition method also works for sets of multivalued dependencies that suffer neither from the intersection anomaly nor from the split left hand side anomaly. This fact will be proved in the next subsection, in this subsection, we will merely illustrate by means of a large example. For more details, see (Siebes and Kersten (1988)).

Let, in some relation R, the two multivalued dependencies be given by $X_1 \twoheadrightarrow Y_1 \mid Z_1$ and $X_2 \twoheadrightarrow Y_2 \mid Z_2$. We have the following complete set of cases:

1) $X_2 \subseteq Z_1, Y_2 \subseteq Z_1$.

2) $X_2 \subseteq Y_1, Y_2 \subseteq Z_1$.

3) $X_2 \subseteq Y_1, Y_2 \cap Y_1 \neq \varnothing, Y_2 \cap Z_1 \neq \varnothing$.

4) $X_2 \subseteq X_1, Y_2 = Y_1$.

5) $X_2 \subseteq X_1, Y_2 \cap Y_1 \neq \varnothing, Y_2 \cap Z_1 \neq \varnothing$.

To analyse these cases by example, we use the relation schema $r = \{ABCDE\}$ and the first mvd as $A \rightarrow\rightarrow BC \mid DE$. The results can be found in Table 1, we will discuss two of them in detail.

case	mvd	type
1	$D \rightarrow\rightarrow E$	$R \in P(A\{BC\}_A\{E\}_D)$
2a	$C \rightarrow\rightarrow E;$	$R \in P(ABC\{D\}_A\{E\}_{C,A})$
2b	$C \rightarrow\rightarrow DE$	$R \in P(ABC\{DE\}_{A,C})$
2c	$BC \rightarrow\rightarrow E$	$R \in P(ABC\{D\}_A\{E\}_{BC,A})$
2d	$BC \rightarrow\rightarrow DE$	$R \in P(ABC\{DE\}_{A,BC})$
3a	$B \rightarrow\rightarrow CE \mid D$	$R \in P(AB\{C\}_B\{D\}_A\{E\}_{A,B})$
3b	$B \rightarrow\rightarrow CDE$	$R \in P(ABC\{C\}_B\{DE\}_{A,B})$
4	$A \rightarrow\rightarrow B \mid CDE$	$R \in P(A\{B\}_A\{C\}_A\{DE\}_A)$
5	$AB \rightarrow\rightarrow CD \mid E$	$R \in P(A\{BC\}_A\{D\}_A\{E\}_A)$

Table 4.1 Two MVD's

The result of case 1 is obvious, because in normalisation theory the multivalued dependencies $A \rightarrow\rightarrow BC \mid DE$ and $D \rightarrow\rightarrow E$ on r yield as decomposition for each extension R: $R = \pi_{ABC}(R) \bowtie \pi_{AD}(R) \bowtie \pi_{DE}(R)$

To enhance the reader's feeling, we give the results of all the possibilities in the cases 2 and 3. To illustrate the table, we will analyse 2a in detail:
Let R_1 and R_2 be two legal extensions of r under the two multivalued dependencies, $R_1 \cup R_2$ is a legal extension iff:

1) $\pi_A(R_1) \cap \pi_A(R_2) = \varnothing$

2) $\pi_C(R_1) \cap \pi_C(R_2) = \varnothing$

Then, let $\pi_A(R_1) = a_1$, $\pi_A(R_2) = a_2$, $\pi_C(R_1) = \{c_1,c_2\}$, $\pi_C(R_2) = \{c_2,c_3\}$ and let R be the smallest legal extension such that $R_1 \subseteq R$ and $R_2 \subseteq R$ then $\pi_E(\sigma_{a_1,c_1}(R)) = \pi_E(\sigma_{a_2,c_2}(R))$, et cetera.

Next, let $\pi_C(R_1) = c_1$, $\pi_C(R_2) = c_2$, $\pi_A(R_1) = \{a_1,a_2\}$, $\pi_A(R_2) = \{a_2,a_3\}$ and let R be the smallest legal extension such that $R_1 \subseteq R$ and $R_2 \subseteq R$ then $\pi_E(\sigma_{a_1,c_1}(R)) = \pi_E(\sigma_{a_3,c_2}(R))$, et cetera.

Finally, let $\pi_A(R_1) \cap \pi_A(R_2) = \{a\}$, and let R be the smallest legal extension such that $R_1 \subseteq R$ and $R_2 \subseteq R$ then $\pi_D(\sigma_a(R)) = \pi_D(\sigma_a(R_1)) \cup \pi_D(\sigma_a(R_2))$. So a constructive form of R is $P(ABC\{D\}_A\{E\}_{A,C})$. The other results in the table are derived analogous.

To see that such a decomposition is generally impossible for sets of multivalued dependencies, consider the relation $r = BVPC$ with the multivalued dependencies $BV \rightarrow\rightarrow P \mid C$ and $PC \rightarrow\rightarrow B \mid V$. Note that both left-hand-sides are split (this example is taken from (Beeri and Kifer (1986))).

To unravel the structure of the extensions of r, let R_1 and R_2 be two disjoint valid extensions of r. They can be independent components iff $R_1 \cup R_2$ is also a valid extension. Straightforward calculation yields that $R_1 \cup R_2$ is valid if:

1) $\pi_{BV}(R_1) \cap \pi_{BV}(R_2) = \varnothing$
2) $\pi_{PC}(R_1) \cap \pi_{PC}(R_2) = \varnothing$

Let R_1 and R_2 satisfy these conditions. The insertion of a tuple t such that $\pi_{BV}(R_1) \cap \pi_{BV}(t) \neq \varnothing$ and $\pi_{PC}(t) \cap \pi_{PC}(R_2) \neq \varnothing$, causes a partition of $R_1 \cup R_2 \cup \{t\}$ such that neither R_1 nor R_2 is a subset of one of the new components; thus violating one of our requirements of a good structure.

In a similar vein it can be shown that relations with a cyclic join dependency do not have a good decomposition. We will start with an example:
Let $r = ABC$, and let the join dependency $\pi_{AB} \bowtie \pi_{BC} \bowtie \pi_{CA}$ be defined on r. To unravel the structure of the extensions of r, let R_1 and R_2 be two disjoint valid extensions of R. They can be independent components iff $R_1 \cup R_2$ is also a valid extension. Straightforward calculation yields that $R_1 \cup R_2$ is valid if:

1) $\pi_{AB}(R_1) \cap \bigcup_{c \in \pi_C(R_2)} \pi_A\sigma_c(R_2) \times \pi_B\sigma_c(R_2) = \varnothing$
2) $\pi_{AB}(R_2) \cap \bigcup_{c \in \pi_C(R_1)} \pi_A\sigma_c(R_1) \times \pi_B\sigma_c(R_1) = \varnothing$
3) $\pi_{BC}(R_1) \cap \bigcup_{a \in \pi_A(R_2)} \pi_B\sigma_a(R_2) \times \pi_C\sigma_a(R_2) = \varnothing$
4) $\pi_{BC}(R_2) \cap \bigcup_{a \in \pi_A(R_1)} \pi_B\sigma_a(R_1) \times \pi_C\sigma_a(R_1) = \varnothing$
5) $\pi_{AC}(R_1) \cap \bigcup_{b \in \pi_B(R_2)} \pi_A\sigma_b(R_2) \times \pi_C\sigma_b(R_2) = \varnothing$
6) $\pi_{AC}(R_2) \cap \bigcup_{b \in \pi_B(R_1)} \pi_A\sigma_b(R_1) \times \pi_C\sigma_b(R_1) = \varnothing$

These conditions are intuitive, using the following equality:

$$\pi_{AB}(R) \bowtie \pi_{BC}(R) \bowtie \pi_{CA}(R) =$$
$$\pi_{AB}(R) \bowtie \pi_{BC}(R) \cap \pi_{BC}(R) \bowtie \pi_{CA}(R) \cap \pi_{AC}(R) \bowtie \pi_{CA}(R).$$

This intuition can be generalised:

DEFINITION 4.12
A join dependency is called simple cyclic if its hypergraph (the components are the hyper-edges) consists of exactly one cycle.

An easy result on simple cyclic join dependencies is:

LEMMA 4.13
Every simple cyclic join dependency is equal to the intersection of a set of acyclic join dependencies □

Moreover, we have that

LEMMA 4.14
$(R_1 \cap R_2) \bowtie R_3 = (R_1 \bowtie R_3) \cap (R_2 \bowtie R_3)$ □

This leads to the following theorem:

THEOREM 4.15
Every join dependency can be written as the intersection of a set of acyclic join dependencies. □

Thus, for every join dependency, the requirements for components to be independent will be similar to the ones in our simple example. As with sets of multivalued dependencies suffering from an anomaly, the requirements show that the decomposition of the extensions in independent components cannot lead to a good structure.

4.2.2 Proving the method correct

To prove that the method sketched above is correct, we need a different characterisation of the join, or better of project-join expressions, than used in the previous subsection, i.e. a characterisation based on grouping and its inverse operation, called *ungrouping*. As these constructions are rather complicated, we first proof the method correct for a relation schema r with a single multivalued dependency. Then we proof the correctness of the method for acyclic join dependencies. Note that this implies that the method is also correct for anomaly-free sets of multivalued dependencies.

4.2.2.1 Multivalued dependencies

Let $r = ABC$ and let R be an extension of R. The project-join expression $\phi = \pi_{AB}(R) \bowtie \pi_{AC}(R)$ can be calculated by associating each value $a \in \pi_A(R)$ with the set β_a of values $b \in \pi_B(R)$ for which $(a, b) \in \pi_{AB}(R)$. Then the result of the join is given by the set:

$$\{(a, b, c) \mid (a, c) \in \pi_{BC}(R) \wedge b \in \beta_a\}.$$

In this calculation, two functions are important:

1) $g(b, a, r)$: $P(r) \rightarrow P(A \times P(B) \times C)$.

2) $u(b, P(A \times P(B) \times C))$: $P(A \times P(B) \times C) \rightarrow P(ABC)$.

The function $g(b, a, r)$ associates each (a, c)-pair with the set β_a, i.e. the function *groupes*. And the function $u(b, P(A \times P(B) \times C))$ first maps a tuple (a, β, c) to the set of tuples $\{(a, b, c) \mid b \in \beta\}$ and then takes the union of all these sets, i.e. the function *ungroupes*.

So, we can construct the function ϕ if we can construct these two functions. We first construct the grouping and then the ungrouping:

4.2.2.1.1 Grouping

To explain the constructions needed for the grouping function, we start with the simplest example.
For the relation type AB, the grouping 'by A' can be defined set-theoretically by:

$$g(a, X) = \{(a, \beta) \mid \beta = \{b \mid (a, b) \in X\}\}.$$

To construct a function from $P(AB) \to P(AP(B))$ by the universal property of power types, we need a relation R on $P(AB) \times A \times P(B)$. More specifically, to construct g, we need a relation R with:

$$(X, a, \beta) \in R \Leftrightarrow \beta = \{b \mid (a, b) \in X\}.$$

The first step toward a construction of this relation, is the function f_{AB} that maps (X, a) to β:

1) The definition of β, $\{b \mid (a, b) \in X\}$, refers to the type ϵ_{AB}, as this types consists of tuples $(X, (a, b))$ with $(a, b) \in X$.
Moreover, $\in : \epsilon_{AB} \to P(AB) \times AB$ is injective. Hence, the universal property of the power type construction yields a function $f_{AB}: P(AB) \times A \to P(B)$ with:

$$\begin{aligned} f_{AB}(X, a) &= \{b \mid (X, a, b) \in \epsilon_{AB}\} \\ &= \{b \mid (a, b) \in X\}. \end{aligned}$$

Combining f_{AB} with the function $\pi_{P(A) \times A}$ and $\in$ gives the function:

$$k = (f_{AB}, id_{P(A) \times A}) \circ \pi_{P(A) \times A} \circ \in : \epsilon_{AB} \to P(AB) \times A \times P(B).$$

Moreover, $k(X, a, b) = (X, a, \beta)$, with β as defined above. However, k does not have to be monic, therefore the second step in the construction of R is the epic/monic factorisation of k:

2) Let $s: P(AB) \times AB \to R$, $r: R \to P(AB) \times A \times P(B)$ be the epic/monic factorisation of $(f_{AB}, \pi_{P(A) \times A})$. Then,

$$(X, a, \beta) \in R \Leftrightarrow \beta = \{b \mid (a, b) \in X\}.$$

As indicated above, g is the function $g: P(AB) \to P(AP(B))$ induced by the monic function r and the universal property of $\epsilon_{AP(B)}$.

To generalise the construction of g given above, note that the generic grouping function is given by $g(b, a, r)$ for $r = ABC$, in which each attribute may be replaced by a set of attributes:
Similar to the construction above, the first step is the construction of a function f_{AB}: $P(ABC) \times A \to P(B)$ that maps a pair (X, a) to the set β with $\beta = \{b \mid \exists c: (a, b, c) \in X\}$, i.e. β is the set of all b's that are related to a. To get this function from the universal property of ϵ_B, we need a relation R on $P(ABC) \times AB$ with:

$$(X, ab) \in R \Leftrightarrow \exists c: (a, b, c) \in X$$
$$\Leftrightarrow \exists c: (X, abc) \in \epsilon_{ABC}.$$

So, R can be constructed from ϵ_{ABC}:

1) Let k denote the function

$$k = \pi_{P(ABC) \times AB} \circ \in : \epsilon_{ABC} \to P(ABC) \times AB.$$

Then $k(X, a, b, c) = (X, a, b)$ with $(a, b) \in \pi^*_{AB}(X)$, i.e. $k(x) \in R$. Moreover, whenever $y \in R$ then there exists an $x \in \epsilon_{ABC}$ such that $y = k(x)$. So R is given by the epic/monic factorisation of k:

$$k_1: \epsilon_{ABC} \to R,$$
$$k_2: R \to P(ABC) \times AB.$$

Finally, f_{AB} is the function induced by k_2 and the universal property of ϵ_{AB}.

As in the simple case above, g is constructed by a relation S on $P(ABC) \times AC \times P(B)$ such that:

$$(X, a, c, Y) \in S \Leftrightarrow ((a, c) \in \pi^*_{AC}(X) \wedge Y = f_{AB}(a))$$

Again, this relation can easily be constructed from ϵ_{ABC}:

2) Consider the function l;

$$l = (f_{AB}, id_{P(ABC) \times AC}) \circ \pi_{P(ABC) \times AC} \circ \in :$$
$$\epsilon_{ABC} \to P(ABC) \times AC \times P(B).$$

For $x \in \epsilon_{ABC}$, $l(x) = (X, a, c, Y)$ with $(a, c) \in \pi^*_{AC}(X)$ and $Y = f_{AB}(a)$. Moreover, whenever $(X, a, c, Y) \in P(ABC) \times AC \times P(B)$ with $(a, c) \in \pi^*_{AC}(X)$ and $Y = f_{AB}(a)$, then there is an $x \in \epsilon_{ABC}$ such that $(X, a, c, Y) = l(x)$. So, S is given by the epic/monic factorisation of l:

$$l_1: \epsilon_{ABC} \to S,$$
$$l_2: S \to P(ABC) \times AC \times P(B).$$

Finally, the function $g(b, a, r)$ is induced by the universal property of $\epsilon_{P(AP(B)C)}$ and l_2.

4.2.2.1.2 Ungrouping

We define the ungrouping function for a relation on $r = A \times P(B) \times C$ in which A, B and C may be replaced by sets of attributes. To get a function $u: P(A \times P(B) \times C) \to P(ABC)$ by the universal property of ϵ_{ABC}, we need a relation on $P(A \times P(B) \times C) \times ABC$. More specifically, to get u, we need the relation R with:

$$(X, a, b, c) \in R \Leftrightarrow \exists \beta: (a, \beta, c) \in X \wedge b \in \beta.$$

The information that $(a, \beta, c) \in X$ is encoded in $\epsilon_{A \times P(B) \times C}$ and the information that $b \in \beta$ is encoded in ϵ_B. Hence, we will need $\epsilon_{A \times P(B) \times C} \times \epsilon_B$ to construct R. For this type, we have the function:

$$\in \times \in: \epsilon_{A \times P(B) \times C} \times \epsilon_B \to$$
$$P(A \times P(B) \times C) \times A \times P(B) \times C \times P(B) \times B.$$

Clearly, to construct R, we only need pairs which map to the same $P(B)$-value. Hence, the first step is the construction of the equaliser of the two projections on P(B):

1) Let $i: E \to \epsilon_{A \times P(B) \times C} \times \epsilon_B$ be the equaliser of the functions:

$$\pi_{1, P(B)} \circ (\in \times \in): \epsilon_{A \times P(B) \times C} \times \epsilon_B \to P(B),$$
$$\pi_{2, P(B)} \circ (\in \times \in): \epsilon_{A \times P(B) \times C} \times \epsilon_B \to P(B).$$

As we need a relation on $P(A \times P(B) \times C) \times ABC$ rather than on $P(A \times P(B) \times C) \times A \times P(B) \times C \times P(B) \times B$, we have to project the $P(B)$ components away. This is the second step:

2) Define the function:

$$j = \pi_{P(A \times P(B) \times C) \times ABC} \circ (\in \times \in) \circ i: E \to P(A \times P(B) \times C) \times ABC$$

Let $x \in E$, then $j(x) = (X, a, b, c)$ such that there is a $\beta \in P(B)$ with $(a, \beta, c) \in X$ and $b \in \beta$. Moreover, for all tuples (X, a, b, c) in $P(A \times P(B) \times C) \times ABC$ with this property, there is an $x \in E$ with $j(x) = (X, a, b, c)$. So, the third step is the epic/monic factorisation of j, which will yield the required relation S:

3) Let:

$$m_1: E \to S,$$
$$m_2: S \to P(A \times P(B) \times C) \times ABC,$$

be the epic monic/factorisation of j. The ungrouping function $u(b)$ is the function induced by m_2 and the universal property of ϵ_{ABC}.

4.2.2.1.3 Constructing ϕ

Now that we have defined the grouping and the ungrouping functions, the function $\phi \equiv (\pi_{AB} \bowtie \pi_{BC})$: $P(r) \rightarrow P(r)$, with $r = ABC$, is easily constructed. In fact, can give various different functions that encode ϕ, we give three of these functions:

THEOREM 4.16

The following three functions are equivalent:

1) $f_1 = u(b) \circ g(a, b, r)$: $P(r) \rightarrow P(ABC)$,

2) $f_2 = u(c) \circ g(a, c, r)$: $P(r) \rightarrow P(ABC)$,

3) $f_3 = u(b) \circ u(c) \circ g(a, c, AP(B)C) \circ g(a, b, r)$: $P(r) \rightarrow P(r)$.

Moreover, each of these functions encodes $(\pi_{AB} \bowtie \pi_{BC})$.

PROOF

This is an easy consequence of the definition of u and g. □

Now, from the previous section, we know that the subset of $P(ABC)$ of all the extensions that satisfy our multivalued dependency is the pullback of f_j and $id_{P(ABC)}$, in which each of the f_j may be used. Moreover, we have that $i: E \rightarrow P(ABC)$ denotes this subtype. Clearly, E is isomorphic to $g(a, b, r) \circ i(E)$. Hence, E is isomorphic with a subset D of $P(AP(B)C)$. Clearly, if $d \in D$, then the dependency $A \rightarrow P(B)$ holds over d. So, let $d \in P(AP(B)C)$ such that this dependency holds over d, we have to proof that $d \in D$. Therefore, let $e = u(b)(d)$. Then $f_1(e) = u(b) \circ g(a, b, r) \circ u(b)(d)$. As $A \rightarrow P(B)$ holds for d, we have $g(a, b, r) \circ u(b)(d) = d$. Hence, $f_1(e) = e$. But then $d \in D$. So, $D = A\{B\}_A C$, and thus $E \equiv A\{B\}_A C$. Similarly, we have that $E \equiv AB\{C\}_A$ and $E \equiv A\{B\}_A\{C\}_A$. So, for a relation schema with one multivalued dependency the method yields a correct result.

4.2.2.2 Acyclic join dependencies

The proof for general acyclic join dependencies follows the same pattern as the proof for multivalued dependencies. The basic idea of the proof is that an acyclic join can be computed by first doing all the necessary grouping followed by the ungrouping steps. As above, this causes as for multivalued dependencies that the type constructed by iteratively applying the groupings is isomorphic to the subtype of $P(r)$ of all extensions of R that satisfy the acyclic join dependencies.

The proof that the methods computes the required acyclic join is rather tedious. It is inductive on the structure of the hypergraph of the join dependency. Rather than giving the complete proof, we supply the difficult pattern in this proof:

This difficult pattern can be seen as a join dependency of the form:

$$(\pi_{AB} \bowtie \pi_{AC} \bowtie \pi_{AD} \bowtie \pi_{AE}) \bowtie \pi_{AF}$$
$$\bowtie (\pi_{FG} \bowtie \pi_{FH} \bowtie \pi_{FI} \bowtie \pi_{FJ}) \bowtie \pi_{FK}$$
$$\bowtie (\pi_{KL} \bowtie \pi_{KM} \bowtie \pi_{KN} \bowtie \pi_{KO})$$

As with the construction of a project-join dependency in the first section of this chapter, we built this join step by step:

1) π_{AB} means that we have to group the B's per A, i.e.

$$P(ABC \cdots O) \rightarrow P(\{A\}_B BC \cdots O)$$

2) The next step is grouping $\{A\}_B C$ per B:

$$P(\{A\}_B BCD \cdots O) \rightarrow P(\{\{A\}_B C\}_B BD \cdots O).$$

As we already grouped the A's per B, this latter type is clearly isomorphic to:

$$P(\{\{A\}_B C\}_B BD \cdots O) \equiv P(\{A\}_B \{C\}_B BD \cdots O).$$

Similarly, we take π_{AD} and π_{AE} into account. This gives us the mappings:

$$P(\{A\}_B \{C\}_B BDEFG \cdots O)$$
$$\rightarrow P(\{A\}_B \{C\}_B \{D\}_B BEFG \cdots O)$$
$$\rightarrow P(\{A\}_B \{C\}_B \{D\}_B \{E\}_B BFG \cdots O).$$

3) Similarly, we use the F and K subpart of the expression, i.e.:

$$P(\{A\}_B \{C\}_B \{D\}_B \{E\}_B BFGHI \cdots O)$$
$$\rightarrow P(\{A\}_B .. \{E\}_B B\{G\}_F FHI..O)$$
$$\rightarrow \cdots$$
$$\rightarrow P(\{A\}_B \cdots \{E\}_B B\{G\}_F \cdots \{J\}_F FKLMNO)$$
$$\rightarrow \cdots$$
$$\rightarrow P(\{A\}_B \cdots \{E\}_B B\{G\}_F \cdots \{J\}_F \{L\}_K \cdots \{O\}_K K)$$

4) In the final step, we have to take the two projections π_{AF} and π_{FK} into account. This is done by grouping $\{A\}_B .. \{E\}_B B$ and $\{L\}_K \cdots \{O\}_K K$ per F, in any of the two possible orders. this finally yields:

$$P(\{\{A\}_B .. \{E\}_B B\}_F \{G\}_F \cdots \{J\}_F \{\{L\}_K \cdots \{O\}_K K\}_F F)$$

Let $(\alpha, \beta, \ldots, \iota, \kappa, .., \omega, f)$ be an element of the image of some extension R of r, moreover, let $a \in \alpha, \ldots, o \in \omega$. It is easy to check that then $((a, b) \in \pi_{AB}(R), \ldots, (k, o) \in \pi_{KO}$. The reverse is also easy to check.

In the example above, we had four 'loose' edges in each of the 'central' nodes *A*, *F* and *K*. Clearly, the number of these loose edges does not matter in the construction above. The inductive proof is now based on the observation that the acyclic join is built from such components.

This informal proof sketch gives us that acyclic join dependencies on relation schemas can be modelled structurally using grouping iff we know how to model the functional dependencies used in this representation. In Chapter 10, we show how functional dependencies are captured in Flock using a dynamical characterisation of functional dependencies. The joint result of this subsection and Chapter 10 is then that acyclic join dependencies are modelled partly structural and partly dynamical.

4.2.3 A generalisation

In the first section, we have seen that a relational schema with a set of algebraic dependencies can be translated to an equivalent Flock type. In the previous subsection, we have seen that if the set of dependencies is either an anomaly free set of multivalued dependencies or if it consists of one acyclic join dependency, a better translation exists. This better translation is reached by a horizontal decomposition of a valid extension. An interesting question is then whether a different decomposition procedure would allow a larger class of dependencies.

In principle, this question is beyond the scope of our thesis. Our goal is the definition of a complex object formalism that satisfies the requirements we set in the first chapter. This chapter is meant as an illustration of type constructions in Flock. Moreover, we have already shown in the first section that the relational semantics can be translated to Flock semantics. The question if a better translation exists is in fact a reformulation of the question what the 'real world semantics' of the large class of algebraic dependencies is; a question that already popped up in Chapter 1. Clearly, this is a question that belongs to relational database theory rather than to Flock. The final reason for claiming this question outside our scope is that a theory of decompositions of relations is a subject for a thesis itself.

However, we briefly look into possible generalisations of our decomposition method, that might be useful. In principle, there are two ways to generalise our method:

1) Our horizontal decomposition forces each of the horizontal components to have the same structure. An obvious generalisation is to loosen this constraint and allow different structures for different components.

2) We only consider horizontal decomposition. So, another generalisation is to consider both horizontal and vertical decompositions.

These two generalisations are the subject of the rest of this subsection.

4.2.3.1 Generalised horizontal decomposition

The horizontal decomposition of our method forces the different horizontal components to have the same structure. Another horizontal decomposition method, based on afunctional dependencies, proposed by De Bra and Paredaens in (De Bra and Paredaens (1983); De Bra and Paredaens (1984); De Bra (1986)). This method is based on the assumption that there may be exceptions to a dependency. Rather than a strict constraint such a dependency is a *goal*. We give a brief description to this approach, the interested reader is referred to (Paredaens *et al.* (1989)).

To motivate the afunctional dependencies, we give an example based on (Paredaens *et al.* (1989)).
Often, a guest in a hotel stays in one room, has one arrival date and one departure date and has one bill to pay. So, it seems wise to make the guest the key of 'his' tuple. However, it may occur that a guest has more accompanying persons than will fit in one room. In this case, more than one room has to be assigned to the guest and he has more than one bill to pay. Moreover, the arrival and departure dates for the different rooms may be different. This is an exception to the usual rule.

The solution proposed by De Bra and Paredaens is to split the relation r in two subrelations r_1 and r_2, such that all guests in r_1 have one room, while all the guests in r_2 have more than one room. So, in r_1 the functional dependency *guest* $\rightarrow r_1$ holds. Clearly, one only wants to enter tuples in r_2 if this is strictly needed, i.e. there should be no tuple with a unique guest attribute in an extension of r_2. Moreover, no *guest* should have tuples in both r_1 and r_2. The first requirement is formalised using *afunctional dependencies*:

DEFINITION 4.17

A set of attributes Y afunctionally depends on a set of attributes X, if for every tuple t_1 in an extension a tuple t_2 exists such that $t_1[X] = t_2[X]$ and $t_1[Y] \neq t_2[Y]$.

A sound and complete axiom system for functional and afunctional dependencies exists as well as normalisation algorithms. The normalisation, which in fact decomposes relations both horizontally and vertically ensures the second requirement, i.e. that a *guest*-value may appear in only one of the horizontal components.

If we compare this decomposition with our decomposition, we see that the components under this approach do not have to be independent following our criteria. For if a guest originally reserves one room, this information will be added in r_1. However, if the guest later reserves a second room, the first tuple will be moved to r_2. So, by weakening the independence definition potentially a larger class of dependencies allow for a natural translation to Flock types. Certainly, this seems a promising approach.

To conclude this short description, we describe how we would model the example in Flock:
For simplicity, we assume that we only keep track of the guests and the rooms they should pay for. The 'normal' guests only pay for one room, hence this can be modelled by $guestname \times room$ with the functional dependency $guestname \rightarrow room$. The 'exceptional' guest pays for several rooms, i.e. a set of rooms, i.e. the type $guestname \times P(room)$, this time with the functional dependency $guestname \rightarrow P(room)$. The function $s : room \rightarrow P(room)$, which maps a room to the singleton set with this room as its only inhabitant, induces that $guestname \times room$ can be seen as a specialisation of $guestname \times P(room)$. Hence, we would introduce the type $guest = guestname \times P(room)$ and its specialisation $regular\text{-}guest = guestname \times room$, each with the appropriate functional dependency (in fact, we shall see in Chapter 10 that the fd on *guest* is inherited by *regular-guest*). Graphically, this can be pictured as:

$$\begin{array}{ccc} regular-guest & = & guest-name \times room \\ & & \downarrow id_{guest-name} \times s \\ guest & = & guest-name \times P(room) \end{array}$$

An exceptional guest is then a guest who is not a regular guest.

4.2.3.2 Horizontal and vertical decomposition

Vertical decompositions of relations are 'dangerous' in the sense that in the definition of the relational schema the attributes are placed together, to capture the semantics of the real world. To include the additional semantics specified by a set of algebraic dependencies we are now prepared to break this bond. So, vertical decompositions may be made in intermediate steps, but in the ultimate translation of the relation type r with a set of dependencies d to an Flock type r',all the attributes should again be present in r'.

Vertical decompositions are of course well known in relational database theory. Normalisation Theory is a theory of vertical decomposition. However, it is well known that, say, cyclic join dependencies do not allow for normalisation. Therefore, we turn our attention to decomposition theory as developed for the view update problem. In a sense, this is ironic, as one of the advantages of the object identity of object oriented databases is that the view update problem does not appear.

Anyway, a view V on a relational schema R is itself a relational schema, which is defined by a query Q on R. The view update problem is: given an update u on V, give an update u' on R, such that $Q(u'(r)) = u(Q(r))$ for all valid extensions r of R. For obvious reasons, this 'new' update should be

minimal, such that if the view is not modified, the stored relation is not modified either. It was recognised by Bancilhon and Spyratos that this problem may be solved by decomposing a relation in independent components, see (Bancilhon and Spyratos (1981a); Bancilhon and Spyratos (1981b)). This direction was taken up again by Hegner (Hegner (1983); Hegner (1984)), and by Gottlob et al. (Gottlob, Paolini, and Zicari (1988)). In this subsection, we give a brief overview on the decompositions used in these three approaches. More information can be found in the forthcoming monograph by S. Hegner (Hegner (1990)).

All these approaches rely on lattice theory. Recall that a lattice L is a set L with a partial ordering $\leqslant$ for which each pair (x, y) has a least upper bound, the join: x $\vee$ y, and a greatest lower bound, the meet: x $\wedge$ y. L is usually represented as an algebra (L, $\vee$, $\wedge$) as these operations completely determine $\leqslant$. A lattice is distributive if it satisfies x $\wedge$ (y $\vee$ z) = (x $\wedge$ y) $\vee$ (x $\wedge$ z). It has universal bounds if it has a greatest and a smallest element, 1 and 0 respectively. Two elements are complementary if their join is 1 and their meet is 0. An element x is an atom if x $\neq$ 0 and $0 \leqslant y \leqslant x$ then y = 0 or y = x.
Now fix a lattice L with universal bounds. Then a decomposition is a subset X such that

1) $\vee$ X = 1.
2) for any partition $\{X_1, X_2\}$ of X, $(\vee X_1) \wedge (\vee X_2) = 0$.

If X and Y are decompositions, then Y is a refinement of X if every element of X is the join of elements of Y. A decomposition X is maximal if for every refinement Y of X, Y = X. X is ultimate if it is a refinement of every decomposition. The facts of distributive lattices that are important for the theory decompositions are:

1) If x has a complement, then it is unique, and it is denoted by $\bar{x}$.
2) Any two decompositions have a common refinement, therefore maximal decompositions are ultimate.
3) The set of all elements of L which have a complement forms a distributive sublattice of L, denoted as Comp(L).
4) If L also has the property that its component algebra is finite, then the atoms of the component algebra form an ultimate decomposition.

To illustrate how lattice theory is used to decompose relational databases, we briefly describe the constant complement approach of Bancilhon and Spyratos (Bancilhon and Spyratos (1981b)). In this approach, the lattice is built on the space of database mappings.
Given a database schema $<D \mid C>$, in which C denotes the constraints on the schema and a database variable D', any mapping from the state space of $<D \mid C>$ to the state space of D' without constraints, i.e.

$$f: S<D \mid C> \rightarrow S<D' \mid \lambda>$$

with λ denote the empty set of constraints is a database mapping. The set of all database mappings on $S<D \mid C>$ is denoted by MAP. The lattice structure is built on MAP. As an aside, note that $f(S<D \mid C>)$ is a subset of $S<D' \mid \lambda>$ and can thus be described by a predicate C' such that $f(S<D \mid C>) = S(<D', C'>$.

To define a lattice structure, we need a partial order; this order is based on a dominance relation:

DEFINITION 4.18

Let $f, g \in$ MAP, we say that f dominates g, $f \geqslant g$ iff

$$f(D_{t_1}) = f(D_{t_2}) \rightarrow g(D_{t_1}) = g(D_{t_2}) \; \forall \; D_{t_1}, D_{t_2} \in S.$$

f and g are equivalent, $f \equiv g$ if $f \geqslant g$ and $g \geqslant f$.

The following step toward the lattice structure is the definition of a top and a bottom:

DEFINITION 4.19

Let $D_{t_0} \in S<D \mid C>$ be a fixed database value, define:

1_S: $S<D \mid C> \rightarrow S<D \mid \lambda>$ such that $1_S(D_t) = D_t$, for all $D_t \in S<D \mid C>$

0_S: $S<D \mid C> \rightarrow S<D \mid \lambda>$ such that $0_S(D_t) = D_{t_0}$, for all $D_t \in S<D \mid C>$

The final step in the definition of a lattice structure is the definition of the join and the meet. However, it turns out that the meet of two database mappings is computationally rather complex, therefore the authors try to define a notion of complement that depends only on the join. This notion of a complement can be paraphrased as saying:

> For two database mappings f and g, f is a complement of g if f distinguishes the database values that g identifies and vice versa, i.e. (f, g) is an injective function.

Note that we have to say a complement as this definition does not force a unique complement. One way to repair this situation would be to define *the* compliment as the minimum of all possible compliments. However, there may be more than one incomparable minimal complement. Moreover, it is shown by Cosmadakis and Papadimitriou in (Cosmadakis and Papadimitriou (1983)) that even in very simple cases, finding a smallest complement is an NP-complete problem.

However, for decomposition it does not really matter if complements are minimal. It is enough to define a pair (f, g) to be a decomposition of a database

iff $(f, g)^{-1}$ exists. Clearly a decomposition is better when the components are independent in some sense. One would anticipate a definition of independence that can be paraphrased as:

> two database mappings f and g are independent, if knowing the value of f does not affect our knowledge of the value of g, and vice versa.

This definition of independence is however so restrictive that almost no database would allow for a decomposition. Therefore, a less restrictive definition of independence is defined called weak independence. As a special case, the authors prove that the usual database decomposition is a weakly independent decomposition.

4.3 Conclusions

In this chapter, we construct types whose elements are exactly the valid extensions of a relation schema over which a set of algebraic dependencies are defined. This exercise serves two purposes:

1) It illustrates the use of the constructions introduced in the third chapter.
2) It allows a comparison between 'relational semantics' and *Floc*-semantics

In the first part of this chapter, we show how the relational constructions *join* and *projection* can be defined within *Floc*. Using these constructions, the construction of the required types is straightforward. This means that *Floc* is at least as expressive as the relational data model with regard to the representation of real world semantics.

However, in Chapter 1, we have seen that dependencies may cause update anomalies. The equivalent *Floc*-types constructed in the first part are just as intuitive as their relational counterparts, i.e. although the elements of a type are sets of tuples, we cannot simply insert or delete a tuple and remain in the same type. Therefore, the remainder of this chapter is devoted to more intuitive translations. In other words given a relation schema r with a set of dependencies D, we want to find a type t, such that:

1) The elements of t correspond to the extensions of r that satisfy D and vice versa.
2) The update semantics of t are natural.

The latter requirement is of course, rather vague. In this chapter, we have reformulated it as stating that the elements of t should consist of independent components. Independence means roughly that a modification of one component does not affect any of the other components.

This notion of independence suggests that the horizontal decompositions of relations over r satisfying D can be used to find such a natural translation. In

the second part of this chapter, we show that indeed, horizontal decomposition leads to a natural translation:

1) under the assumption that we have a natural translation for functional dependencies and
2) provided that D is restricted to conflict-free sets of multivalued dependencies.

The latter requirement shows that our criterium for natural translations coincides with the requirements for normalisation; provided we find a natural translation for functional dependencies. The functional dependencies reappear in Chapter 10.

Clearly different criteria for natural update semantics will lead to different classes of permitted dependencies. A natural candidate would seem to be to allow both horizontal and vertical decompositions. We end this chapter by a short introduction in the theory of horizontal and vertical decompositions into independent components.

Chapter 5

Complex structures

In the third chapter we have introduced the notion of a type graph is. Moreover, we defined how a typegraph can be defined in Floc, using its type constructions. Recall that a real world entity can be represented by several entities of different types. Moreover, these different representing entities are connected through the edges of the typegraph; because the edges represent semantical transformations that transform one representation of an entity in another representation of the same entity. In this chapter, we pack the different representations of a real world entity into one *abstract* object. The resulting unique representation of the real world object is called a *structured object.* Such objects are called structured objects rather than complex objects, as they do not yet meet the requirements we specified in Chapter 1. For example, structured objects do not have an identity, nor can they be used to represent partial information.

Strucured objects have a type, called a structure type. In the first section of this chapter, we define structure types and structured objects formally. An important role in the definition of structure types is played by the equivalence relation on entity types. The semantics of these new concepts are given in the second section. In the third section, we define a pre-order on structure types. Moreover, we show that this pre-order defines a subtyping relation that subsumes Cardelli's well-known subtyping mechanism (Balsters and Fokkinga (1990); Cardelli (1988)). As mentioned above, equivalent entity types play an important role in the definition of structure types. Therefore, in the fourth and final section we look at isomorphisms between Flock's entity types that are canonical in some well-defined sense.

As a motivation for the work reported in this chapter, note that the type equivalence used in the definition of structure types and the pre-order defined on these structure types give, e.g., Flock's query language its flexibility. To be more

precise, in Section 2, we show how the pre-order defines a subtyping. Subtyping in this context means that if e is an expression of type t_1, denoted by $e: t_1$, and t_1 is a subtype of t_2, denoted by $t_1 \leqslant t_2$ then $e: t_2$ also holds. The use of type equivalence will induce that if $e: t_1$ and $t_1 \equiv t_2$ than $e: t_2$ also holds. The effect is that the user does not have to know the type of an object precisely, an approximation suffices.

5.1 Structure types

In chapter 3 we have seen that entity type t is an abstract description of a natural class n in the UoD. An entity e of type t is a representation of some real world entity r in the class n (note that t may be a representation of several different real world entities in n, but the uniqueness problem will be solved by the object identity in Chapter 10). As a real world entity may be in more than one natural class, a real world entity may be represented by more then one entity of possibly different types. Each of the representations of the real world entity describe a different view. If one such view can be transformed in another view, this is represented by an edge in the type graph.

The fact that each real world entity may be represented for a number of nodes in the type graph can be reformulated as: each real world entity r implies a subgraph S_r of the type graph TG. The nodes in S_r are the entity types that describe a natural class of which r is a member. The edges of S_r are the edges in TG between two nodes which are both in S_r. This subgraph S_r can be seen as a form for the representation of r, all its representations are given by consistently (consistent with respect to the semantical transformations) specifying values for each of the nodes in the graph.

Clearly, different real world entities may imply the same subgraph of TG. This means that the representations of these real world entities is given by filling in the same form, i.e. such a subgraph belongs to a class of entities that are similar in this sense. As before, this notion is captured as a *type*. This type is called the *structure type* of a real world entity r because it conveys the structure of all the representations of r. Moreover, each real world entity r is associated with one such a subgraph and the 'filled-out form' can be seen as *the* representation of r. Such a filled-in form is called a structured object. So, each real world entity is represented by exactly one structured object. It is not called a complex object, as one structured object may be the representation of the several real world entities. The concept of an object identity is needed to make the representation unique. Moreover, we want complex objects that can represent incomplete information, a subject that will be discussed in Chapter 9.

The rest of this section is a formal definition within the framework of Flock of the concepts we introduced informally above, viz. structure types and structured objects. Before we can give these definitions, however, a number of remarks on structure graphs have to be made.

For the first remark, let S_r be the subgraph induced by the real world entity r. Moreover, let t_1 and t_2 be two nodes in S_r. Then *all* edges that exist in *TG* between t_1 and t_2 have to be present in S_r. The reason for this requirement is that the meaning of an edge f in *TG* is as follows: if $<t_1, v>$ is a representation of r, then $<t_2, f(v)>$ is also a representation r. As already said, these edges can be likened to ISA-relations. Clearly leaving out such an edge would yield an unfaithful representation of the UoD.

The second remark is an assumption concerning structure types. We have seen above that each real world entity r induces a subgraph S_r of *TG*. Clearly, there is no need that this subgraph is connected. However, the theory we are developing would be much easier if each S_r is a connected subgraph. Clearly, a theory that supposes that each S_r is a connected subgraph is easily extended to a theory in which each S_r is a union of a collection of maximal connected subgraphs. So, it is not that this assumption makes Flock mathematically easier, but it makes reading and writing this thesis much easier. Moreover, one could say that the various disconnected components are related as they describe information on the same real world object. Hence, there should be an entity type describing this relation. This 'new' entity type would in fact connect the loose components to one connected subgraph. Therefore, we assume that each S_r is connected and, consequently, structure types will be defined as connected subgraphs

The third and final remark is also an assumption on the nature of an S_r. For the moment, suppose that *TG* is acyclic, then of course each S_r is acyclic. Each acyclic connected directed graph has one or more 'root nodes', i.e. nodes that have only outgoing edges. If we supply values for all the root nodes, values for the other nodes in the graph can be derived automatically using the semantical transformations. In fact, it can always be arranged that representations of r for the root nodes automatically induce all representations of r. Now suppose that all representations of r are induced by the representations for the root nodes t_1 and t_2. This means that r is fully described by a $(<t_1, v_1>, <t_2, v_2>)$-pair. So, by putting $t_1 \times t_2$ 'on top of' t_1 and t_2 we can give S_r a unique root such that a representation for that unique root induces all representations of r.

So, if *TG* is acyclic, we may as well assume that S_r has a unique root and that all representations of r are induced by its representation for this unique root type. This implies that all we have to know of a real world entity is *one* representation, this special representation together with *TG* yields all representations. Obviously, this is much easier to handle than a set of representations that have to be kept consistent with each other.

If *TG* contains cycles, a subgraph does not have to have root nodes. However, there will always be a subset I of the nodes of S_r, such that specifying representations for the nodes in I induces all the representations of r. Clearly, I may contain redundant information, i.e. there may be nodes t_1 and t_2 in I, for which there is a functional transformation $f: t_1 \rightarrow t_2$ in the type graph. In that case t_2 is redundant, because specifying a representation for t_1 induces a representation for t_2.

Therefore, let M be a minimal (minimal with regard to the subset behaviour) subset of the nodes of S_r such that representations for M induce all representations for r. This means that r is completely specified by a tuple of values, one value for each node in M. As TG is finite, M is finite and hence we can construct the entity type t as the product type of the elements of M. Moreover, r is completely specified by a value for t. So, again we have an entity type in which all representations of r are encoded. However, there is a difference with the acyclic case. If the type graph is acyclic, there is a unique entity type t that can serve as the root type of S_r; slightly abusing language, we will call t the root type of S_r. If the type graph contains cycles, however, there may be more than one entity type that can serve as the root of S_r as M is not necessarily unique. If e.g. S_r consists of two synonymous entity types t_1 and t_2 (and thus also contains the invertible semantical transformations between them) then both the sets $\{t_1\}$ and $\{t_2\}$ are minimal and completely specify S_r.

So, in general there is a choice in denoting one type as the root type of S_r. To discuss this choice a bit further, let S_r consist of two entity types t_1 and t_2 with edges $f: t_1 \rightarrow t_2$ and $g: t_2 \rightarrow t_1$. Clearly, we have two choices for the root of S_r, viz. t_1 and t_2. Moreover, we can distinguish two cases in this choice:

1) f and g are each others inverses and thus t_1 and t_2 are synonymous;
2) f and g are not each others inverses.

As the root type is, in a sense, the most important type of S_r, denoting one entity type as being the root type gives some information on the UoD. In fact, if C denotes the class of all real world entities with the same S it is very well possible that for some of them t_1 seems appropriate as root type while for others t_2 is a better choice (except if t_1 and t_2 are synonymous, then the choice between either one will be arbitrary).

If we want to identify the structure types with their root type, such an arbitrary choice will always be unnatural to part of the future users. If t_1 is chosen, those that prefer t_2 have to remember that t_2 is derived from t_1. Clearly, this would greatly reduce the use of synonymous entity types in a type graph. Therefore, we should not have to choose between either t_1 or t_2 as root, but choose both as root. There are two ways to achieve this without violating our wish of having a unique root type:

1) Choose either one of the synonymous candidates and make the conversions between them automatically, i.e. we let the designer make an arbitrary choice but hide the choice from the user.
2) Hide the choice not only from the user but also from the designer, by making her define the root up to equivalence, i.e. by using equivalence types as roots.

Clearly, the second option is preferable, not only from a design point of view, but also from the theoretical point of view. Somewhere in the formalism it should be made explicit that synonymous entity types can be used interchangeable. The earlier we accomplish this, the easier it will be.

So, to make the choice of a root type for an S_r less arbitrary, we should 'collapse' all sets of equivalent types into equivalence types. In Chapter 3 we have informally defined equivalence types and shown how the co-equaliser can be used to construct the equivalence type for two equivalent entity types. We will now generalise this construction to n synonymous entity types and then use this construction to give a formal definition of an equivalence type.

For two entity types t_1 and t_2 with functions $f: t_1 \to t_2$ and $g: t_2 \to t_1$ such that $f = g^{-1}$, the construction of $[t_1]$ went as follows:
Consider the functions:

$$f^* = [f, id_{t_2}]: t_1 + t_2 \to t_1 + t_2$$

$$g^* = [id_{t_1}, g]: t_1 + t_2 \to t_1 + t_2$$

The co-equaliser of f^*, g^* is the object $[t_1] = (t_1 + t_2)/R$ in which R is the equivalence relation:

$$(x, y) \in R \Leftrightarrow x = y \vee f^*(x) = y \vee g^*(x) = y$$

To generalise this construction, let $\{t_1, \ldots, t_n\}$ be a set of entity types together with functions $f_j^i: t_i \to t_j$ for each pair $(i, j) \in I \times I$ such that $f_i^j \circ f_j^i = id_{t_i}$ and $f_i^j \circ f_j^k = f_i^k$.
Now, we want to identify $x \in t_i$ and $y \in t_j$ if $f_j^i(x) = y$, or equivalently $f_i^j(y) = x$. As above, the f_i^j induce functions $t_1 + \cdots + t_n \to t_1 + \cdots + t_n$ similar as above, i.e.:

$$g_i = [f_i^1, \ldots, f_i^{j-1}, id_{t_i}, f_i^{j+1}, \ldots, f_i^n]$$

The relation R on $t_1 + \cdots + t_n$ we are looking for can now be given as:

$$(a, b) \in R \Leftrightarrow a = b \vee g_1(a) = b \vee \cdots \vee g_n(a) = b$$

For, let $x \in t_i$, $y \in t_j$, and let $a = i(x)$ and $b = i(y)$, then $f_j^i(x) = y$ iff $g_j(a) = b$.
Then, the entity type $[t]$ we want has to identify elements x and y of $t_1 + \cdots + t_n$ exactly if $(x, y) \in R$. Hence, $[t]$ is the co-equaliser of $g_1, \ldots, g_n$, as it is straightforward to verify that R is an equivalence relation.

This discussion leads us to the following definition:

DEFINITION 5.1

Let $\{t_1, \ldots, t_n\}$ be a maximal set of synonymous entity types, then the entity type $[\{t_1, \ldots, t_n\}]$, called an *equivalence entity type*, is defined as the co-equaliser of the $g_1, \ldots, g_n$ as defined above.

We will often write $[t]$ for an equivalence entity type, its meaning is the equivalence entity type of all entity types that are equivalent to t. In particular this means that if $t_1 \equiv t_2$, then $[t_1] = [t_2]$.

Let $t_1 \equiv t_2$ and $t_3 \equiv t_4$, a function $f: t_1 \to t_3$ induces a function $f': t_2 \to t_4$. In fact, it is easy to see that f induces a function $[f]: [t_1] \to [t_3]$. We use the notation $[f]$ as there may be functions $f_1: t_1 \to t_3$ and $f_2: t_3 \to t_4$ that induce the same function $[t_1] \to [t_3]$. Note that if $f = g \circ h$, then $[f] = [g] \circ [h]$. Categorically, this can be expressed by saying that $[\_]$ is a functor. But more importantly, for us it means that we can consistently replace each node t and each function F in TG by $[t]$ and $[f]$ respectively; the resulting graph is called the equivalence type graph:

DEFINITION 5.2

Let TG be a type graph, its *equivalence typegraph*, denoted by ETG is constructed by mapping each entity type t to its equivalence type $[t]$ and each function $f: t_1 \to t_2$ to the function $[f]: [t_1] \to [t_2]$.

Similar to TG, each real world entity r induces a subgraph S_r of ETG. The advantage of ETG over TG is that it does not contain cycles of synonymous entity types. Hence, in choosing a root for such a subgraph, we do not have to choose between synonyms. Of course, one still has to choose between non-synonymous entity types. But in our opinion, now the two options have different real world semantics. To illustrate this, let t_1 and t_2 be two entity types with functions $f: t_1 \to t_2$ and $g: t_2 \to t_1$ which are not each others inverses. Suppose we choose t_1 as root, than a value v for t_1 induces the value $f(v)$ for t_2, which in turn induces $g(f(v))$ for t_1 et cetera. So, we get the following sets of values:

$$t_1: \{(fg)^n(v) \mid n \geqslant 0\}$$

$$t_2: \{(fg)^n(f(v)) \mid n \geqslant 0\}$$

If we would have chosen t_2 as root, and had started with $f(v)$ as first value for t_2, we would have obtained the sets:

$$t_1: \{(fg)^n(v) \mid n \geqslant 1\}$$

$$t_2: \{(fg)^n(f(v)) \mid n \geqslant 0\}$$

Clearly, the sets $\{(fg)^n(v) \mid n \geqslant 0\}$ and $\{(fg)^n(v) \mid n \geqslant 1\}$ do not have to coincide if f and g are not each others inverses. In fact, as neither f nor g have to be surjective, it may very well occur that choosing say t_1 as root may imply that some reasonable values for t_2 can never occur. Hence, we feel that the choice makes a semantical difference.

So, we cannot hide the choice in such cycles, the only way out would be to forbid cycles with non-equivalent entity types. However, this would greatly complicate the definition of Flock. Consider for example the entity types 1 and $1 + t$, by definition we have the functions:

$$i_1 : 1 \rightarrow 1 + t$$

$$!_{1+t} : 1 + t \rightarrow 1$$

Clearly, these two functions are not each others inverses. So, if we want to forbid such non-isomorphism cycles we have to forbid some constructions. In fact, we would have to rule out too much, e.g. the grouping and ungrouping functions defined in the previous chapter are in general not each others inverses. So, forbidding non-isomorphism cycles would seriously degenerate Flock' simplicity as well as its expressiveness. Therefore, we do not make this choice.

To summarise the long discussion of this section, we get that structure types represent connected subgraphs of *ETG* with one root and if $[t]$ is reachable from the root in *ETG* it should be present in this subgraph. In other words, a structure type is completely specified by one entity type in *ETG*. In fact, it is completely specified by an entity type in *TG* as *TG* determines *ETG*. So, we get the following definition:

DEFINITION 5.3

Let *TG* be an entity type and t an entity type in *TG*, the structure type induced by t is defined by the expression:

$$\textit{structure type name} = ([t], TG)$$

$[t]$ is called the root entity type, or simply root, of the structure type *name*.

In the rest of this thesis, we will use the name $s_{[t]}$ for the name of the structure type with root $[t]$. Moreover, we will assume that the type graph is known, i.e. we will not specify the type graph. As an aside, note that we do not have to specify the complete equivalence class of t, one representative, viz., it is enough to determine which equivalence class is meant.

The informal semantics of this expression are of course that $s_{[t]}$ points to the subgraph of *ETG* consisting of all paths from $[t]$. The formal semantics will be given in the next section. A structured object is a 'filled-out' structure type, i.e. a value should be given for each node in the graph, moreover, these values have to be consistent with the transformations in the graph. So, from the informal semantics we already see that it is specified by a value for the root type, i.e. an equivalence class of values. So, we get the following definition of a structured object:

DEFINITION 5.4

A structured object is a pair $<s_{[t]},[v]>$ in which $s_{[t]}$ is a structure type and $<[t],[v]>$ is an entity of type $[t]$.

Note that it is enough to specify the equivalence class of values by one of its representatives, i.e. v is a value such that $<t_2, v>$ is an entity of type t_2, and t_1 and t_2 are equivalent.

As an aside, note that what we call structured objects coincides with the definition of complex objects in many other formalisms. We have already explained in the introduction of this chapter that structured objects are not yet complex objects as they lack both an identity and the possibility to represent incomplete information. However, structured objects are the foundation on which complex objects will be defined. Therefore, a query language for complex objects will be built on a part that can be called a query language for structured objects. To simplify the introduction of the query language for complex objects, we will define a separate query language for structured objects in Chapter 7. Besides readability of this thesis, this 'primitive' query language facilitates a comparison between Flock and other formalisms.

5.2 The semantics

In chapter 3, we gave the semantics of the type graph in a category C. In these semantics, the entity types are mapped to objects in the category C. So, as structure types are conceptually graphs, we cannot map structure types to objects of C rather they should be mapped to subgraphs of C. Hence, we need a new category whose objects are graphs in C. Furthermore, as each structure type has exactly one root, the objects in this new category should be graphs with one root, i.e. graphs with a distinguished node such that all other nodes are reachable from this special node r. Finally, the structured objects are completely determined by their value for the root type, hence functions on a structured object only need this value. Hence, a function between two structure types is specified by a function between their root types. So, the functions in the new category are basically functions in C. This gives us the following definition:

DEFINITION 5.5

Let C be a category, the category $FSub(C)$ is constructed from C as follows:

1) The objects of $FSub(C)$ are the finite subcategories of C that have an initial object.

2) Let o_1 and o_2 be two $FSub(C)$-objects with initial objects r_1 and r_2. An $FSub(C)$-arrow between o_1 and o_2 is a C-arrow from r_1 to r_2.

It is easy to see that $FSub(C)$ is indeed a category, i.e.

LEMMA 5.6

$FSub(C)$ as defined above is a category

PROOF

The arrows of C and $FSub(C)$ coincide. □

By 'inheriting' the arrows of C, $FSub(C)$ not only becomes a category, but if C has limits, then so does $FSub(C)$. Similarly, it inherits *exponents* and *powerobjects*. Hence if C is a topos, then so is $FSub(C)$:

THEOREM 5.7

If C is a topos, then so is $FSub(C)$.

PROOF

The proof is straightforward, as an illustration we show how $FSub(C)$ inherits its binary products from C:

Let o_1 and o_2 be objects in $FSub(C)$ with initial objects r_1 and r_2 respectively. Then $r_1 \times_C r_2$ with its identity arrow is a $FSub(C)$-object, called $o_1 \times o_2$. Let o_3 be another $FSub(C)$ object, with root r_3 and $FSub(C)$-arrows $f: o_3 \to o_1$ and $g: o_3 \to o_2$. The functions f and g are inherited from functions $f: r_3 \to r_1$ and $g: r_3 \to r_2$. Hence, by the universal property of $r_1 \times_C r_2$, there is a function $(f, g): r_3 \to r_1 \times_C r_2$ such that $\pi_1 \circ (f, g) = f$ and $\pi_2 \circ (f, g) = g$. Hence, there is a $FSub(C)$-arrow $(f, g): o_3 \to o_1 \times o_2$ with $\pi_1 \circ (f, g) = f$ and $\pi_2 \circ (f, g) = g$. It is easy to see that (f, g) is unique with respect to this property. □

Note that this theorem implies that we could also define the constructions of Flock on structured types. Moreover, if we see structured types as subgraphs of *TG*, the proof of the theorem shows that it does not matter whether if we e.g. first multiply the root types and then take the structure type induced by this new type or directly multiply the structure types. So, adding constructions to this new layer in Flock does not extend its expressivity. Therefore we refrain from adding structure type constructions.

The definition of the semantics of a structure type is given in two steps. In the first step we determine which subgraph of *ETG* is induced by the node $[t]$ and, after that, this subgraph is mapped to the appropriate subcategory of C, using the mapping *Sem* of chapter 3.

For the first step, we need the subgraph of *ETG* of all nodes that are reachable from a node $[t]$. This subgraph can be defined inductively as follows:

DEFINITION 5.8

Let $[t]$ be a node in *TG*, and let $\{[f_1]: [t] \to [t_1], \ldots, [f_n]: [t] \to [t_n]\}$ be the set of all edges in *ETG* which start at $[t]$. The subgraph generated by $[t]$, denoted by $SG([t])$ is defined as:

$$Nodes(SG([t])) = \{[t]\} \cup nodes(SG([t_1])) \cup \cdots \cup nodes(SG([t_n]))$$

$$Edges(SG([t])) = \{[f_1], \ldots, [f_n]\} \cup Edges(SG([t_1])) \cup \cdots \cup Edges(SG([t_n])).$$

All nodes and all edges in *ETG* are constructed by means of Flock

constructionsfrom the nodes and edges of *TG*. So, we can use the mapping *Sem* of Chapter 3 to map them to *C*. Clearly, this has the effect that $SG([t])$ is almost mapped to a subcategory of *C*; almost as we have to add the identity arrows on the nodes to make it a real subcategory. So, we use the mapping Sem_S, which is defined by:

DEFINITION 5.9

Let $[t]$ be a node in *ETG*, the semantics of the structure type $s_{[t]}$, denoted by $Sem_S(s_{[t]})$ is given by the following object in $Fsub(C)$:

$$Nodes(Sem_S(s_{[t]})) = \{Sem(n) \mid n \in Nodes(SG([t]))\},$$

$$Edges(Sem_S(s_{[t]})) = \{id_n \mid n \in Nodes(Sem_S(s_{[t]}))\} \cup \{Sem(f) \mid f \in Edges(SG([t]))\}.$$

It is straightforward to verify that $Sem_S(s_{[t]})$ is indeed a subcategory of *C* and thus an object in $FSub(C)$.

Given the semantics of a structure type, the semantics of a structured object are given analogous to the semantics of an entity, i.e.

DEFINITION 5.10

Let $<s_{[t]}, [v]>$ be a structured object, its semantics, denoted by $Sem_S(<s_{[t]}, [v]>)$ is given by the $FSub(C)$-arrow:

$$[v]: 1 \rightarrow Sem_S([s_t])$$

implied by the *C*-arrow:

$$[v]: 1 \rightarrow Sem([t])$$

which gives the semantics of the entity $<[t], [v]>$.

So, similar to the entities, structured objects are *constants*, and again these constants are mapped semantically to functions with the initial object as their source.

5.3 Subtyping

In Chapter 3, we defined an entity type t_1 to be a specialisation of an entity type t_2 iff there is a semantical transformation $f: t_1 \rightarrow t_2$ in *TG*. This is easily extended to structure types, we could define one structure type $s_{[r_1]}$ to be a specialisation of a structure type $s_{[r_2]}$ iff there is a semantical transformation: $[f]: [r_1] \rightarrow [r_2]$ in *ETG*. However, we prefer a 'semantical' definition, i.e using $Sem_S(s_{[r_i]})$. Moreover, we will call $s_{[r_1]}$ a subtype of $s_{[r_2]}$ rather than a specialisation, although we will use the two words interchangeably:

DEFINITION 5.11

Let $s_{[r_1]}$ and $s_{[r_2]}$ be structure types, $s_{[r_1]}$ is called a subtype of $s_{[r_2]}$, denoted by $s_{[r_1]} \leqslant s_{[r_2]}$, if $Sem_S(s_{[r_2]})$ is a subcategory of $Sem_S(s_{[r_1]})$.

This definition is consistent with our specialisation definition, as we have:

LEMMA 5.12

If r_1 is a specialisation of r_2, then $s_{[r_1]} \leqslant s_{[r_2]}$.

PROOF

By definition, the fact that r_1 is a specialisation of r_2 means that there is an arrow $f: r_1 \to r_2$ in TG. Hence, there is an arrow $[f]: [r_1] \to [r_2]$ in ETG, but this implies by definition that $SG(s_{[r_2]}) \subseteq SG(s_{[r_1]})$. So, $Sem_S(s_{[r_2]})$ is a subcategory of $Sem_S(s_{[r_1]})$. □

As an aside, note that subtyping as used in the theory of programming languages means that if t_1 is a subtype of t_2, and e is an expression of type t_1, then e also has type t_2. Expressions are not yet defined, all we have are values. However, the lemma above yields that if r_1 is a specialisation of r_2, than we may 'coerce' a value of type $s_{[r_1]}$ into a value of type $s_{[r_2]}$.

Note that the reverse implication of the lemma above is not true. The notable exception is that for each structure type s, $s \leqslant s$ now holds, while it would not necessarily hold if the 'functional' definition had been chosen; because id_t does not have to be in the type graph. In fact, we have:

LEMMA 5.13

Let $s_{[r_1]} \leqslant s_{[r_2]}$ then either $[r_1] = [r_2]$ or there is a semantical transformation $f: r_1 \to r_2$ in TG.

PROOF

$s_{[r_1]} \leqslant s_{[r_2]}$ holds if $Sem_S(s_{[r_2]})$ is a subcategory of $Sem_S(s_{[r_1]})$. But the only new arrows added are the id_n. □

This lemma is called the functional characterisation of $\leqslant$. Note that it implies that we only have to look at the root types in TG to check whether two structure types are in a subtyping relation. Before we look at some of the consequences of this observation, we first show that $\leqslant$ is a pre-order, as the subcategory relation is transitive:

LEMMA 5.14

The relation $\leqslant$ on structure types is a pre-order.

PROOF

The relation is both reflexive and transitive as we have seen above. □

However, $\leqslant$ is not a partial order because it is not anti-symmetric, i.e. $s_1 \leqslant s_2$ and $s_2 \leqslant s_1$ does not imply $s_1 = s_2$. Because the root types of s_1 and s_2 may be in a non-isomorphism cycle in *TG*. As an illustration of this fact, consider the types 1 and $t + 1$, then we have the functions $!: t + 1: \rightarrow 1$ and $i: 1 \rightarrow 1 + t$. So, although we have natural candidates, viz. s_0 and s_1 for a top and a bottom, we do not and will not have a type lattice. However, the fact that for each structure type t, $s_o \leqslant t$ and $t \leqslant s_1$ will prove to be sufficient.

The definition of the subtyping relation yields an easy algorithm to check whether two structure types are in a subtyping relation; simply check their graph. However, we have seen above that we only have to look at the root types in *TG* to see a subtyping relation. This observation gives us the following easy to proof result:

THEOREM 5.15

Let $s_{[r_1]}$, $s_{[r_2]}$, $s_{[r_3]}$, and $s_{[r_4]}$ be structure types, then:

1) $s_{[r_1 \times r_2]} \leqslant s_{[r_1]}$ and $s_{[r_1 \times r_2]} \leqslant s_{[r_2]}$
2) If $s_{[r_3]} \leqslant s_{[r_1]}$ and $s_{[r_3]} \leqslant s_{[r_2]}$ then $s_{[r_3]} \leqslant s_{[r_1 \times r_2]}$
3) $s_{[r_1]} \leqslant s_{[r_1 + r_2]}$ and $s_{[r_2]} \leqslant s_{[r_1 + r_2]}$
4) If $s_{[r_1]} \leqslant s_{[r_3]}$ and $s_{[r_2]} \leqslant s_{[r_3]}$ then $s_{[r_1 + r_2]} \leqslant s_{[r_3]}$
5) $s_{[r_3^{r_4}]} \leqslant s_{[r_1^{r_2}]}$ if $s_{[r_3]} \leqslant s_{[r_1]}$ and $s_{[r_2]} \leqslant s_{[r_4]}$.
6) if $s_{[r_1]} \leqslant s_{[r_2]}$ then $s_{[P(r_1)]} \leqslant s_{[P(r_2)]}$.

PROOF

This follows directly from the universal properties of the constructions. □

Note that we cannot completely characterise the subtyping relation in this way, as the subtyping may be forced by user defined functions.

To illustrate this theorem, we give an example of item 5). Consider the structure types *employee*,*director*, *secretary*, and *manager*, with the obvious subtyping *director* $\leqslant$ *manager* $\leqslant$ *secretary* $\leqslant$ *employee*. Then the theorem says informally that $secretary^{manager} \leqslant employee^{director}$. So, for example the degradation of a manager to a secretary is a specialisation of the degradation of a director to an employee.

At this point, it is illustrative to compare the partial order Cardelli (Balsters and Fokkinga (1990); Cardelli (1988)) uses with our order. His system has the three type constructors product, co-product, and exponentiation, they are called record, variant, and function type respectively. Each of these constructions has a different rule in the order relation. The reader is warned that Cardelli's definition is on the level of what we would call entity types. Paraphrasing Cardelli's definition in our formalism we get:

DEFINITION 5.16
The partial order of Cardelli, denoted by $\leqslant_{Car}$ is defined as follows:

1) $A \times B \times C \leqslant_{Car} A \times B$
2) $A \leqslant_{Car} B \Rightarrow A \times C \leqslant_{Car} B \times C$
3) $A + B \leqslant_{Car} A + B + C$
4) $A \leqslant_{Car} B \Rightarrow A + C \leqslant_{Car} B + C$
5) $C \leqslant_{Car} A \wedge B \leqslant_{Car} D \Rightarrow C^D \leqslant_{Car} A^B$

An easy consequence of the theorem above is:

COROLLARY 5.17
If $A \leqslant_{Car} B$ then $s_{[A]} \leqslant s_{[B]}$

So, the rules Cardelli defined for subtyping are for us simply a consequence of our semantics, i.e. we did not have a choice. Moreover, for Cardelli, the types A, $<a:A>$ (a record with one component of type A labeled by a) and $[a:A]$ (a variant with one component of type A labeled by a) are incomparable, while in Flock, a sum or a product of one component is the same as the original type. As nothing comes for free, the reader should not be surprised that our more general definition has a drawback. In Cardelli's system, the subtyping is completely determined by the rules given above. As said before, we cannot give such a complete characterisation as we cannot characterise the user defined functions. As an aside, note that Cardelli's system has been enriched with power types by Balsters and de Vreeze (1990). It is interesting to note that the subtyping rule for power types chosen in this work is exactly consequence 6) above, i.e. $A \leqslant B \Rightarrow P(A) \leqslant P(B)$; again a rule that is also true in Flock.

The fact that Cardelli's subtyping rules are equally valid in Flock, suggest that Cardelli's system can be givem categorical semantics similar to the semantics of Flock. This is, however, outside the scope of this thesis. But, we do want to point out that there do exist categorical semantics for this subtyping system. In (Bruce and Longo (1988)), the authors provide such semantics. Their approach is, however, rather different from ours, and a comparison is again outside the scope of this thesis.

5.4 Type equivalences

The equivalence type $[t]$ represents all entity types t_i in TG such that $t \equiv t_i$. This means that the database designer has to define all the t_i that she considers 'reasonable' alternatives explicitly. In particular, this means that she has to add all kinds of obvious alternatives. For example, the intuitive semantics would force

her to add both $t_1 \times t_2$ and $t_2 \times t_1$ to achieve that both forms may be used. Clearly, this greatly reduces the power obtained by using equivalence types. One would like that all the obvious equivalences are taken into account automatically. That is $[t]$ not only represents all entity types in *TG* that are equivalent to t, but also entity types that are not in *TG*, but are obviously equivalent.

Before we can add such a feature to Flock, we have to clarify what is intended by the concept 'obviously equivalent'. Clearly, we do not want each isomorphism to be taken into account. For example, it is well-known that for the natural numbers N and $N \times N$ are isomorphic sets. This isomorphism will, in general, have no obvious semantics in the UoD that is going to be modelled.

So, what isomorphisms are we going to take into account? In other words, which isomorphisms are obvious? Or, in a more mathematical formulation, what isomorphisms are canonical? Clearly, any choice made will be ad-hoc. The solution we choose depends on the semantics of Flock, i.e. category theory. We define canonical equivalent by:

DEFINITION 5.18

Two entity types t_1 and t_2 are canonically equivalent iff the isomorphism can be constructed from the universal properties of the type constructions of Flock.

Using the semantics of the type constructions in Flock, this definition can be rephrased as follows:

LEMMA 5.19

Two entity types t_1 and t_2 are canonically equivalent if for any topos C and for any assignment from the basic type graph B in C, $Sem(t_1) = Sem(t_2)$ holds.

PROOF

All that is given for a topos is that all our constructions are well defined. Moreover, categorically these constructions are defined by the universal properties. □

So, to be able to check whether two entity types are canonically equivalent, we have to be able to decide whether two constructions in a topos are isomorphic. This is the topic of this section. In the first sub-section we give a number of important examples and the second sub-section, we consider the question whether we can give a finite axiom system that can be used to decide whether two constructions in a topos are isomorphic.

5.4.1 Examples

Perhaps the most important property a schema for automatic equivalences should have is that one may substitute equals for equals in a construction. In the case of the product, co-product, exponentiation and power types, it is straightforward what is meant by substituting equals for equals. The proof that it may indeed be done is given in the following theorem:

THEOREM 5.20

Let T be a topos, and let A, B, C and D be objects in this topos, then:

1) if $(A \equiv C \wedge B \equiv D) \vee (A \equiv D \wedge B \equiv D)$, then $A \times B \equiv C \times D$
2) if $(A \equiv C \wedge B \equiv D) \vee (A \equiv D \wedge B \equiv D)$, then $A + B \equiv C + D$
3) if $A \equiv C \wedge B \equiv D$ then $A^B \equiv C^D$
4) if $A \equiv B$ then $P(A) \equiv P(B)$ and $\epsilon_A \equiv \epsilon_B$.

PROOF

1) Let $f: A \to C$ and $g: B \to D$ be the isomorphisms, then by the universal property of the product, we have the functions:

$$h = (f \circ \pi_A, g \circ \pi_B),$$
$$k = (f^{-1} \circ \pi_A, g^{-1} \circ \pi_B),$$

together with the equations:

$$\pi_C \circ h = f \circ \pi_A, \pi_D \circ h = g \circ \pi_B,$$
$$\pi_A \circ k = f^{-1} \circ \pi_C, \pi_B \circ k = g^{-1} \circ \pi_D.$$

So, $\pi_A \circ k \circ h = f^{-1} \circ \pi_C \circ h = f^{-1} \circ f \circ \pi_A = \pi_A$, and similarly $\pi_B \circ k \circ h = \pi_B$. Hence, by the universal property of the product, $k \circ h = id_{A \times B}$. The proof that $h \circ k = id_{C \times D}$ is analogous.

2) By the duality principle and 1)

3) Let $f: A \to C$ and $g: B \to D$ be the isomorphisms, then:

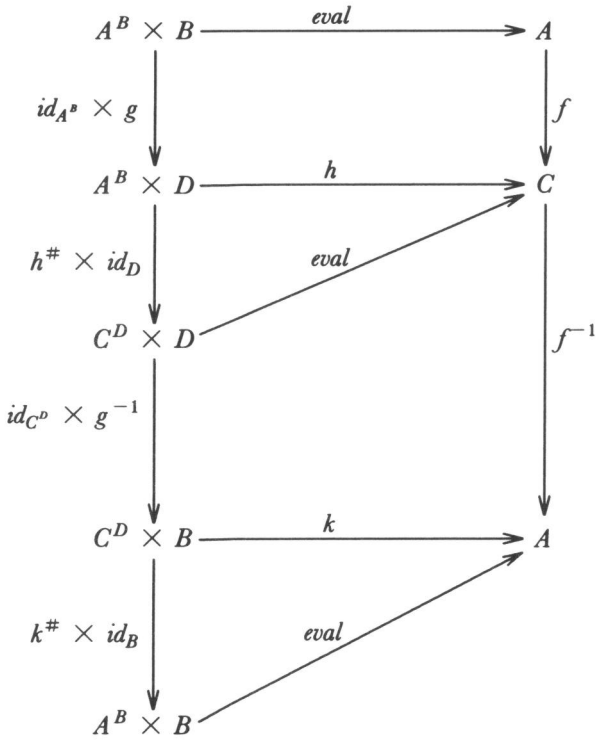
$$h = f \circ eval \circ (id_{A^B} \times g^{-1}): A^B \times D \to C$$

Hence, by the universal property of exponentiation, we have a unique function $h^{\#}: A^B \to C^D$ such that $eval \circ (h^{\#} \times id_D) = h$. Similarly, we have the function

$$k = f^{-1} \circ eval \circ (id_{C^D} \times g): C^D \times B \to A$$

and thus the unique $k^{\#}: C^D \to A^B$ with $eval \circ (k^{\#} \times id_B) = k$. Consider

the following diagram:



The following equations hold:

a) $f \circ eval = h \circ (id_{A^B} \times g)$ by the definition of h;

b) $eval \circ (h^\# \times id_D) = h$, as seen above;

c) $f^{-1} \circ eval = k \circ (id_{C^D} \times g^{-1})$ by the definition of k;

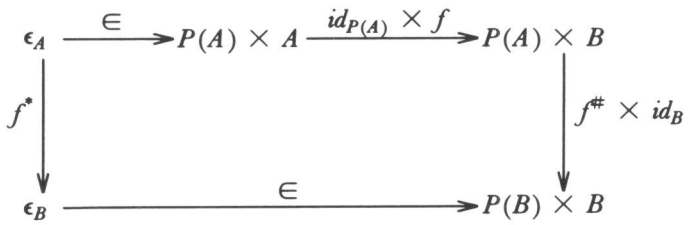
d) $eval \circ (k^\# \times id_B) = k$ as seen above.

Hence, the diagram commutes and $eval = eval \circ ((k^\# \circ h^\#) \times id_B)$. So, by the universal property of exponentiation we have $k^\# \circ h^\# = id_{A^B}$. Similarly, $h^\# \circ k^\# = id_{C^D}$ holds.

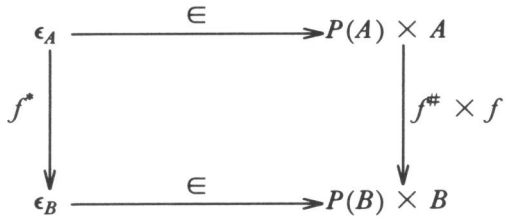
4) Let $f: A \to B$ be the isomorphism, composition with "$\in$" yields the injective (monic) function:

$$(f \times id_{P(A)}) \circ \in : \epsilon_A \to P(A) \times B$$

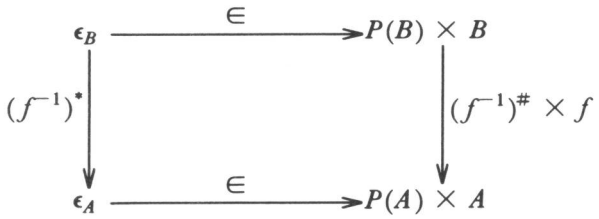
By the universal property of the power object, we get the pullback diagram:



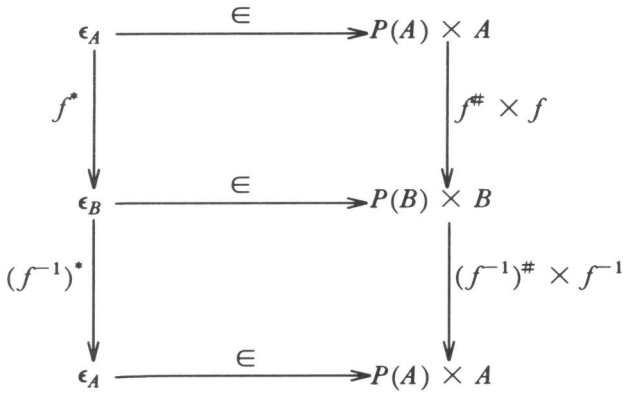
Note that we abbreviated $((f \times id_{P(A)}) \circ \in)^*$ to f^* and $((f \times id_{P(A)}) \circ \in)^\#$ to $f^\#$. As $A \equiv B$, the following diagram is also a pullback diagram:



Similarly, we get the following pullback diagram:



The composition of these two pullback diagrams yields another pullback diagram:



So, by the universal property of power types, $(f^{-1})^* \circ f^* = id_{\epsilon_A}$ and $(f^{-1})^{\#} \circ f^{\#} = id_{P(A)}$. Similarly, one proves that $f^* \circ (f^{-1})^* = id_{\epsilon_B}$ and $f^{\#} \circ (f^{-1})^{\#} = id_{P(B)}$ □

As an aside, note that all examples in this lemma are examples in which $A \leqslant B \wedge B \leqslant A \Rightarrow A \equiv B$ holds.

For the equaliser and the co-equaliser it is perhaps less obvious what is meant by substituting equals for equals, as they are defined to act on functions rather than on objects. However, if $f: A \to B$ and $g: C \to D$ are isomorphisms, and $k, l: B \to C$ are parallel functions, A and B, respectively C and D, can be considered as equals one may want to substitute. This leads to the following theorem:

THEOREM 5.21

Let T be a topos, and let A, B, C and D be objects in this topos and let f, g, k, l be arrows in this topos with $f: A \to B$ and $g: C \to D$ are isomorphisms, and $k, l: B \to C$ parallel, then:

1) $eq(k, l) \equiv eq(g \circ k \circ f, g \circ l \circ f)$
2) $co\text{-}eq(k, l) \equiv co\text{-}eq(g \circ k \circ f, g \circ l \circ f)$

PROOF

1) Let $i: E \to B$ be the equaliser of k and l and let $i': E' \to A$ be the equaliser of $g \circ k \circ f$ and $g \circ l \circ f$. Then, as g is an isomorphism, we have that $f \circ i' \circ l = f \circ i' \circ k$. Hence, by the universal property of E, there is a unique function $k: E' \to E$ such that $i \circ k' = f \circ i'$. Similarly, we have a unique function $k': E \to E'$ such that $i' \circ k' = f^{-1} \circ i$. So, as f is an isomorphism, $i' = i' \circ k' \circ k$ and $i = i \circ k \circ k'$. Hence, by the universal properties of E and E' we have that $k \circ k' = id_E$ and $k' \circ k = id_{E'}$.
2) By the duality principle and 1) □

So, in all constructions, equals may be substituted for equals.

More interesting examples can be given if more than one construction is used in an object; i.e. examples that show the 'interaction' between two constructions. The following theorem gives some such examples:

THEOREM 5.22

Let T be a topos, and let A, A_1, B_2, C_1 and C_2 be T objects, then:

1) Let $f_1, g_1: B_1 \to C_1$ and $f_2, g_2: B_2 \to C_2$ be two pairs of parallel arrows with $h_i: A_i \to B_i$ the equaliser of f_i and g_i. Then:

$eq(f_1, f_2) \times eq(g_1, g_2) \equiv eq(f_1 \times f_2, g_1 \times g_2)$.
Where $eq(f_1,f_2) \times eq(g_1,g_2)$ is the set $(A_1 \times A_2)$ together with the arrow $h_1 \times h_2 \colon A_1 \times A_2 \to B_1 \times B_2$, induced uniquely by the h_i.

2) Let $f_1,g_1 \colon A_1 \to B_1$ and $f_2,g_2 \colon A_2 \to B_2$ be two pairs of parallel arrows with $h_i \colon B_i \to C_i$ the co-equaliser of f_i and g_i. Then:
$co\text{-}eq(f_1 + f_2, g_1 + g_2) \equiv co\text{-}eq(f_1 + f_2) + co\text{-}eq(g_1 + g_2)$.
Where $co\text{-}eq(f_1,f_2) + co\text{-}eq(g_1,g_2)$ is the set $(C_1 + C_2)$ together with the arrow $h_1 + h_2 \colon B_1 + B_2 \to C_1 + C_2$ induced uniquely by the h_i.

3) $eq(\pi_1,\pi_2)(A \times A) \equiv A$. With the π_i the two projections of $A \times A$ to A.

4) $co\text{-}eq(i_1,i_2)(A + A) \equiv A$. Withe the i_A, the two injections from A in $A + A$.

PROOF

First of all note that:

$$\begin{aligned}(f_1 \times f_2) \circ (h_1 \times h_2) &= (f_1 \circ h_1) \times (f_2 \circ h_2) \\ &= (g_1 \circ h_1) \times (g_2 \circ h_2) \\ &= (g_1 \times g_2) \circ (h_1 \times h_2)\end{aligned}$$

Moreover, let X be a set with an arrow $k \colon X \to B_1 \times B_2$ such that

$$f_1 \times f_2 \circ k = g_1 \times g_2 \circ k$$

Then we have that:

$$\begin{aligned}f_1 \circ \pi_{B_1} \circ k &= \pi_{C_1} \circ (f_1 \times f_2) \circ k \\ &= \pi_{C_1} \circ (g_1 \times g_2) \circ k \\ &= g_1 \circ \pi_{A_1} \circ k\end{aligned}$$

Hence, by the universal property of the equaliser, there is a unique function $l_1 \colon X \colon \to A_1$ such that $h_1 \circ l_1 = \pi_{B_1} \circ k$. Similarly, there is a unique function $l_2 \colon X \colon \to A_2$ such that $h_2 \circ l_2 = \pi_{B_2} \circ k$. These two functions induce the function:

$$(l_1, l_2) \colon X \to A_1 \times A_2$$

Moreover:

$$\begin{aligned}(h_1 \times h_2) \circ (l_1, l_2) &= (h_1 \circ l_1, h_2 \circ l_2) \\ &= (\pi_{B_1} \circ k, \pi_{B_2} \circ k) \\ &= k\end{aligned}$$

Hence, by the universal property of the equaliser, $eq(f_1, f_2) \times eq(g_1, g_2) \equiv eq(f_1 \times f_2, g_1 \times g_2)$.

2) By the duality principle and 1)

3) For the function $\Delta = (id_A, id_A)\colon A \to A \times A$ we have $\pi_{1,A} \circ \Delta = \pi_{2,A} \circ \Delta$. Moreover, let X be a set with a function $f\colon X \to A \times A$ such that $\pi_{1,A} \circ f = \pi_{2,A} \circ f$. Then

$$\begin{aligned} \Delta \circ \pi_{1,A} \circ f &= \Delta \circ \pi_{2,A} \circ f \\ &= (\pi_{2,A} \circ f, \pi_{2,A} \circ f) \\ &= (\pi_{1,A} \circ f, \pi_{2,A} \circ f) \\ &= f \end{aligned}$$

Hence, by the universal property of the equaliser, $eq(\pi_1,\pi_2)(A \times A) \equiv A$

4) By the duality principle and 3). □

The last examples given in this subsection, illustrate the arithmetical behaviour of our constructions. The proof of these examples requires rather sophisticated category theory, therefore we refrain from giving these proofs. As an aside, note that these examples are often used as exercises in the literature in that area:

THEOREM 5.23

Let T be a topos, and let A,B and C be objects in T, then:

1) $(A + B) \times C \equiv (A \times C) + (B \times C)$
2) $A^{B+C} \equiv A^B \times A^C$
3) $(A^B)^C \equiv (A^C)^B$

In this subsection, we have seen quite a few examples of canonically equivalent type constructions. To implement the canonically equivalent types as a feature in the equivalence types of Flock, however, we need a procedure to decide whether two constructions are canonically equivalent or not. This is the subject of the next subsection.

5.4.2 An axiom system

If we want $[t]$ to represent not only the types s that are equivalent to t given by the type graph, but also those that are canonical equivalent to such an s, we need a procedure that decides for two types s and t whether they are canonically equivalent. In the previous subsection, we have provided a set of examples of canonically equivalent types. What one would like to have is a general description of such canonically equivalent types. Clearly, such an undertaking is

infeasible since there is always an infinite number of these equivalences. The most we can hope for is to find an algorithm proving equivalence within a setting of a sound and complete axiom system.

Given such an axiom system, it is straightforward to define a rewrite system such that s and t are canonically equivalent if s can be rewritten to t. In this subsection, we will try to find such an axiom system.

The first result toward such an axiom system is given by Bruce and Longo in (Bruce and Longo (1985)).

THEOREM 5.24

Let C be a category with exponentiation, then:

$$(\mu^\tau)^\sigma \equiv (\mu^\sigma)^\tau$$

is a sound and complete axiom system to decide whether two types are canonically equivalent.

PROOF

In the previous subsection we have seen that this rule is sound. To see that the axiom system is complete, the reader is referred to (Bruce and Longo (1985)). □

It is only a partial result, as it is assumed that all we know about the category C is that it has exponentiation. In other words for all categories with exponentiation, this sole axiom is sound

A slightly more general, but still partial result is given by the following theorem, which is a result by L.G.L.T Meertens and the author (unpublished):

THEOREM 5.25

Let C be a category with exponentiation and products, then:

$$(\mu^\tau)^\sigma \equiv (\mu^\sigma)^\tau$$
$$\mu^{\sigma \times \tau} \equiv (\mu^\sigma)^\tau$$
$$(\sigma \times \tau)^\mu \equiv (\sigma^\mu) \times (\tau^\mu)$$
$$\sigma \equiv \mu \Leftrightarrow (\forall z : z^\sigma \equiv z^\mu)$$

is a sound and complete axiom system to decide whether two types are canonically equivalent.

PROOF (SKETCH)

The completeness of this axiom system is a direct consequence of the previous theorem, as the last three axioms allow us to rewrite each expression to an expression containing only exponentiation, using a *fresh* variable z whenever the fourth rule is used.

The soundness of the first three axioms has been proven in the previous

subsection. So, we only have to proof the soundness of the fourth axiom. This proof is rather tedious, therefore we only give the intuition:
It is trivial that:

$$\sigma \equiv \mu \Rightarrow (\forall z : z^{\sigma} \equiv z^{\mu})$$

holds. The hard part is the implication the other way around. The truth of this implication can be seen intuitively, because if we have a procedure that transforms functions $\sigma \to z$ into functions $\mu \to z$ for an arbitrary z then this procedure has to use a function $f : \sigma \to \mu$. Basically, the proof formalises this intuition. □

We did not give the formal proof of this theorem, as there is a related result by Rittri (Rittri (1989)) which has an easier proof. Before we give this theorem and its proof, we make some observations that are well-known, but in this context due to Rittri.

The set of natural numbers N can be considered as a category, say $\boldsymbol{N}$. The objects of $\boldsymbol{N}$ are the natural numbers and the are the set-theoretic functions between two numbers viewed as sets in the standard way, i.e. $0 = \varnothing$, $1 = \{\varnothing\}$ et cetera. Note that this implies that two numbers are isomorphic iff they are the same number. As a convention, we will write n for a natural number in N, and $\boldsymbol{n}$ or $Cat(n)$ for the same number considered as an element of $\boldsymbol{N}$.

The following lemma is easy to verify:

LEMMA 5.26

$\boldsymbol{N}$ has a terminal object, products and exponentiation. Moreover:

1) The terminal object $\boldsymbol{1} = Cat(1)$
2) $\boldsymbol{n} \times \boldsymbol{m} = Cat(n \times m)$
3) $\boldsymbol{n}^{\boldsymbol{m}} = Cat(n^m)$

So, when we consider N as a category, products and exponentiation behave as would be expected.

Furthermore, let expressions be generated by the grammar:

$$e ::= e \mid e \times e \mid e^{e} \mid \textit{variable}$$

An assignment ψ is a mapping from the variables to the natural numbers, ψ is extended to expressions as usual. If $\psi(e_1) = \psi(e_2)$ for all assignments ψ, we write:

$$(N,\ 1,\ \times,\ \exp) \models e_1 = e_2$$

There exists a finite axiom system for $(N,\ 1,\ \times,\ \exp)$, due to Martin (Martin (1973)):

THEOREM 5.27
The axiom system:

1) $A \times B = B \times A$
2) $A \times (B \times C) = (A \times B) \times C$
3) $(C^B)^A = C^{B \times A}$
4) $(B \times C)^A = B^A \times C^A$
5) $1 \times A = A$
6) $A^1 = A$
7) $1^A = 1$

is sound and complete for $(N, 1, \times, \exp)$.

Rittri (Rittri (1989)) extended this result to categories usin g the N view on N as follows:

THEOREM 5.28
Let C be a category with an initial element, exponentiation and products, then the following axiom system:

1) $\sigma \times \tau = \tau \times \sigma$
2) $\sigma \times (\tau \times \mu) = (\sigma \times \tau) \times \mu$
3) $(\mu^\tau)^\sigma = \mu^{\tau \times \sigma}$
4) $(\tau \times \mu)^\sigma = \tau^\sigma \times \mu^\sigma$
5) $1 \times \sigma = \sigma$
6) $\sigma^1 = \sigma$
7) $1^\sigma = 1$

is sound and complete to decide whether two types are canonically equivalent.

PROOF
We have already proven that all these axioms are sound. So, we only have to prove that they are complete. So, suppose that there exists a canonical isomorphism $t_1 \equiv t_2$ that cannot be proven using the axiom system given above. As N is a category that fulfils the requirements with its similarity to N, we can translate this to $(N, 1, \times, \exp)$ as saying that there are expressions e_1 and e_2 such that $(N, 1, \times, \exp) \models e_1 = e_2$ but cannot be proven from the axioms. This is in contradiction with Martin's result. □

Independent from Rittri, this theorem is also announced in (Longo, Asperti, and Di Cosmo (1989)). Moreover, it turned out that there exists an older proof of this theorem, due to Solov'ev (Solov'ev (1983)).

This theorem and its proof seem easily adapted to a category that has an initial object, a terminal object, products, co-products and exponentiation. However, Martin proved in (Martin (1973)) that the theory $(N, +, \times, \exp)$ (whose definition should be obvious from the discussion above) has no finite axiomatisation. Moreover, Gurevič proved in (Gurevič (1990)) that the theory $(N, 1, +, \times, \exp)$ also has no finite axiomatisation. A positive result that can be mentioned is that Macintyre proved in (Macintyre (1981)) that both the theory $(N, +, \times, \exp)$ and the theory $(N, 1, +, \times, \exp)$ are decidable. It is doubtful whether these results carry over to a general category; moreover, we are interested in the theory $(N, 0, 1, +, \times, \exp)$.

Given the negative results cited above, the reader should not be surprised that we do not have a sound and complete axiom system to decide whether two entity types are canonically equivalent. In fact, we do not even have a clue whether or not this problem is decidable! So, rather than a sound and complete axiomatisation, we present a sound axiom system which we think will cover all practical cases. Clearly, this axiom system will include the one given above for products and exponentiation. We will now look at the other constructions and decide what axioms should be in at least:

The first construction we consider is the co-product. For this construction, alone we have the following laws:

1) $A + B \equiv B + A$

2) $A + (B + C) \equiv (A + B) + C$

3) $A + 0 \equiv A$

In combination with products and exponentiation, we have the following laws:

4) $A \times (B + C) \equiv A \times B + A \times C$

5) $A^{B+C} \equiv A^B \times A^C$

6) $A^0 \equiv 1$ provided $A \not\equiv 0$

7) $0^A \equiv 0$ provided $A \not\equiv 0$

It seems unlikely that a database designer, or a user, would use the type $A + 0$ as it expresses that an entity is either of type A or of type 0, i.e., the entity is of type A. Similarly, A^0 and 0^A seem rather far-fetched. Therefore, we are satisfied when the rules 1), 2), 4) and 5) are present in our axiom system. The reader might object that to the type 1 similar objections may be made, however, this type has proven its worth in functional programming languages.

The next construction is the power set construction. Set-theoretically, we now

that $P(A) \equiv 2^A$, in which 2 is the canonical two-element set. In the next chapter, we will prove that in a topos the equation $P(A) \equiv \Omega^A$ in which Ω is the set of truth values for this topos. For the moment, the importance of this equation is that it allows us to remove the power type construction from a type. So, besides this rule, we do not need new rules for the power set construction.

For the remaining two constructions, the equaliser and the co-equaliser, we do not add any rules. For the co-equaliser, this is obvious as we indicated that its main use lies in the construction of equivalence types. The equaliser construction is a selection on a previously constructed domain. So, the only hope for proving two equalisers to be isomorphic is that the underlying domains are isomorphic and that in both cases the same functions are equalised. Equality of functions can be decided for the typed lambda calculus. However, this is beyond the scope of our thesis.

So, the axiom system we are proposing can be defined as follows:

DEFINITION 5.29

The axiom system Γ consists of the following axioms:

1) $A + B \equiv B + A$
2) $A + (B + C) \equiv (A + B) + C$
3) $A \times (B + C) \equiv A \times B + A \times C$
4) $A^{B+C} \equiv A^B \times A^C$
5) $A \times B = B \times A$
6) $A \times (B \times C) = (A \times B) \times C$
7) $(C^B)^A = C^{B \times A}$
8) $(B \times C)^A = B^A \times C^A$
9) $1 \times A = A$
10) $A^1 = A$

The axiom system Γ^+ is the axiom system Γ with the extra axiom:

12) $P(A) = \Omega^A$

The axiom system Γ^+ is sound in any topos, but not necessarily comple te. The problem we now have to solve is to show that this axiom system can be used to proof equivalence between two entity types. Moreover, to be useful, this proof should also yield the isomorphism between the two entity types.

In computer science, the usual approach to such a problem is the use of normal forms. If both entity types have the same normal form, they are isomorphic. So, we have to prove that unique normal forms exist for Γ^+. The first observation is that if the first step of the rewrite procedure axiom 12) is applied whenever possible, the problem reduces to the problem of unique normal forms for Γ.

The axiom system Γ is, not completely by coincidence, related to Tarski's famous High School Algebra Conjecture, see (Martin (1973)). This conjecture can be paraphrased as stating that Γ is a sound and complete axiom system for $(N, 1, +, \times, \exp)$; in this context, Γ is often referred to as *the High School Axioms*. We have already seen that Gurevič proved that no finite sound and complete axiom system exists for this theory (Gurevič (1990)). However, Henson and Rubel have proved (Henson and Rubel (1984)) that, with a small restriction on the terms, a unique normal form exists. Translated to Flock their result reads:

THEOREM 5.30

Let RT be the set of all possible Flock types that do not contain nodes of the form $(A + B)^C$, $eg(f, g)$ and $co\text{-}eq(f, g)$, then each of the elements of S has a unique normal form with regard to Γ. □

So, exponentiation is allowed under the restriction that the mantissa does not contain a $+$.

Summarising this subsection, we can say that for an important subclass of the Flock types canonical equivalence can be proven. However, a sound and complete axiom system could not be found for the complete set.

Chapter 6

Methods

In chapter 5, we have defined structured objects and structured object types. These definitions are not particularly fit for the description of methods, due to the inheritance rules on exponential types. In the first section of this chapter we sketch this problem and discuss an approach for its solution. In the second section, we define an alternative semantics for entities of an exponential type as well as for the related inheritance rule. These alternative semantics are used to discuss methods in the third section. Methods should have types, the definition of such method types is discussed informally in Section 4. Using the intuitions offered in the first four sections, we finally provide formal definitions of methods and method types in Section 5. Finally, in Section 6, we examine the semantics of the application of a method to a structured object.

6.1 The problem

One of the type constructions of Flock is exponentiation. Given two types A and B, B^A can be seen as the collection of all functions $f : A \rightarrow B$. At the introduction of this construction, we mentioned that it is used to model the dynamic aspects of the UoD. Moreover, we gave a simple example of how this can be done for entity types. However, this simple solution for dynamic aspects does not carry over straightforwardly from entities to structured objects.

To explain this problem, we briefly look at object-oriented formalisms. Often, the concept of a *class* is important in such a formalism. For our purposes in this section, a class can be seen as a type plus a set of *methods*; for the formal definition of a class in Flock, see chapter 10. The methods of a class are the only means through which objects belonging to that class can be manipulated. So, the methods are a way to model dynamical aspects of the UoD. Hence, one would

expect that methods are encoded using exponential types.

Inheritance of classes implies both type inheritance and method inheritance. Using an example from Danforth and Tomlinson (1988):

1) Suppose we have the types *thing* and *car* where $car = thing \times speed$, with the usual projection functions $is\text{-}thing : car \rightarrow thing$ and $max\text{-}speed ; car \rightarrow speed$. So, *car* is a subtype of *thing*. Furthermore, suppose we have only one method for *things*, viz. *make-older* of type $thing^{thing}$. An easy way to encode the *class* of things would use the type $thing\ class = thing \times thing^{thing}$. One could look at this definition as stating that *thing* represents the static aspects of *thingclass*, while $thing^{thing}$ represents the dynamic aspects.

2) As *car* is a subtype of *thing*, it seems reasonable to create the class *carclass* as a subclass of the class *thingclass*. Clearly, we would like that inheritance on methods to give us a method *make-older-car* of type car^{car}. So, we would expect that we could define the class *carclass* using the type $carclass = car \times car^{car}$.

3) However, now we do not have that *carclass* is a subtype of *thingclass*, as car^{car} is not a subtype of $thing^{thing}$ using the rules we gave in chapter 5.

So, the straightforward encoding of a class does not give us the inheritance relationship on classes we would expect. Moreover, the reason for this problem lies in the the subtyping relationship on exponential types.

In (Danforth and Tomlinson (1988)), the authors suggest that the solution to this problem may be either a looser coupling between the polymorphic methods and the polymorphic (static) types or to give up polymorphic methods all together. In this chapter, we follow the first approach, in fact, each method will reflect the structure of the structure types it can act upon. This is achieved by giving an alternative semantics to dynamic entities.

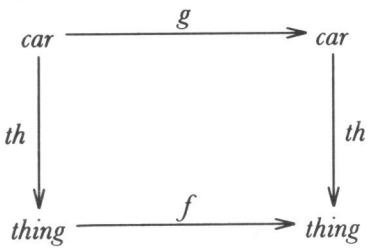
Before we turn to these alternative semantics in the next section, we give one more example of the problem. This second example illustrates that the problem does not depend on the fact that *car* is a 'record-type' with *thing* as a 'sup-record type'. Moreover, it shows that it is even a problem to decide whether a method can be inherited at all. The example is taken from Balsters and de By (1989).
For the types *nat* and *real*, the naturals can be taken as a subtype of the reals, with the canonical injection $i : nat \rightarrow real$ establishing this subtype relationship. Moreover, suppose that the *age* component of *thing* from the example above is the reals as domin. Then, we can make a class of *naturalthing*, which is the class of things with a natural number as age. If we have a method *make-younger* for *thing*, which divides the age component by 2, it is not clear if this method can be inherited by *naturalthing*. For, $x \rightarrow x/2$ does not define a function $nat \rightarrow nat$.

From the description of the problem above, it should be clear that it is a semantic problem. So, the solution to this problem has to be found in the semantics of Flock.

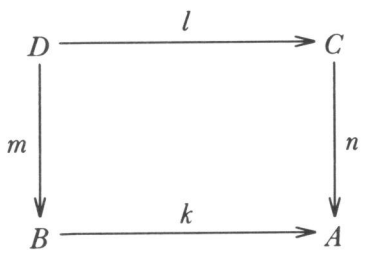
6.2 Alternative semantics

In the previous section, we have seen that inheritance of methods, or perhaps better functions, does not agree with inheritance of static types. Moreover, we have seen that it is not straightforward that functions can be inherited. So, our first task is to define what it means that a function is inherited.

To motivate our definition, we start with an example.
Let *car* be a subtype of *thing*, with the function $th: car \to thing$ establishing this relationship. Moreover, let $f: thing \to thing$. Let g be a function $g: car \to car$. It seems reasonable to say that f and g are essentially the same function, if for $x \in D_{car}$, $f \circ th(x) = th \circ g(x)$ holds. This requirement ensures that if we look at x with *thing* glasses, i.e. we see x as a thing, the application of g yields the same result as the application of f. This means that for someone with *thing* glasses, f and g are the same function. The condition given above, can be depicted graphically as follows:



The functions f and g in the type graph, are mapped by the semantics to arrows f and g in some topos T. Let C be an arbitrary category. From C, we can construct a new category $C^{\to}$. The objects in $C^{\to}$ are the arrows from C and an arrows in $C^{\to}$ from $k: D \to C$ to $l: B \to A$ is given by a *pair* (m, n) of C-arrows such that the following diagram commutes:



It is easy to see that $C^{\to}$ is a category, with pointwise composition. For example, if $(m, n): k \to l$ and $(o, p): l \to h$, then

$$(o, p) \circ (m, n) = (o \circ m, p \circ n): k \to h.$$

But even a stronger result holds:

THEOREM 6.1

If C is a topos, then so is $C^{\rightarrow}$.

For the proof of this result, the reader is referred to e.g. (Goldblatt (1979)).

If we compare the two pictures above, we see that our condition for inheritance of functions means that (th, th) is a function from g to f in $C^{\rightarrow}$! In the terminology of Chapter 3, this means that g should be a specialisation of f. This suggests that we should use $C^{\rightarrow}$ to give semantics to methods. However, this is not straightforward, as entities of an exponential type, say $f: B^A$, are mapped to $f: 1 \rightarrow B^A$ by the semantics as defined in Chapter 3. Moreover, the correspondence between elements of B^A and actual functions $A \rightarrow B$ is not obvious. In fact, the author would not know how to encode this correspondence in a category in general. However, one part of the correspondence is easy:

LEMMA 6.2

Let C be a category with exponentiation, with each function $f: A \rightarrow B$ we can associate uniquely an element $\hat{f}: 1 \rightarrow B^A$, such that for each element $x: 1 \rightarrow A$ of A, $eval(\hat{f}, x) = f \circ x$.

PROOF

This is an easy consequence of the universal property of exponents. Combining the functions $f: A \rightarrow B$ and $\pi_A: A \times 1 \rightarrow A$, we get the function:

$$f \circ \pi_A: A \times 1 \rightarrow B$$

Now, the universal property of exponents gives us the required function $\hat{f}$. □

The element $\hat{f}: 1 \rightarrow B^A$ is sometimes called the *name* of f.

In chapter 3, we already restricted the user defined edges in a type graph to functions that can be constructed explicitly in the topos. Now we will go one step further:

> *In the rest of this thesis, we only allow elements of exponential types that are the name of a function in the category that is explicitly constructed.*

This assumption means that whenever we have an element $x: A^B$, we know the arrow $y: A \rightarrow B$ such that $x = \hat{y}$. In the rest of this thesis, we will use the same symbol both for $f: A \rightarrow B$ and its name $f: 1 \rightarrow B^A$. It will be clear from the context which of the two C-arrows is meant.

By this assumption, we can now exploit our intuition for an alternative semantics for exponential type for a formal definition of methods and their types. However, before we come to the formal definitions, we informally define both methods and method types in the following section, respectively the section after that. The formal definitions are given after these informal discussions.

6.3 Methods, informally

In the previous section, we have seen that an alternative semantics for dynamic entities can be found in $C^{\rightarrow}$. Moreover, we have seen that under these semantics, our intuition on inheritance of methods coincides with specialisation. In this section, we briefly explore some of the consequences of these alternative semantics. The section is divided into two subsections. In the first subsection, we investigate the structure of methods. In the second, we look at the question of automatic inheritance of methods.

6.3.1 Method structure

In object-oriented formalisms, methods are the only means to manipulate objects. In other words, whenever an object has to be changed, a method has to be applied. So, methods can be seen informally as a kind of functions, or better, morphisms that act on objects.

As explained before, classes can be seen as a type together with a set of methods that can be applied to objects of that class. A class B is a subclass of a class A, if the type of B is a subtype of the type of A and if the set of methods associated with B is a superset of the set of methods associated with A. Perhaps this second requirement needs some motivation.

As with subtyping the subclass hierarchy encodes ISA relationships. In other words, if B is a subclass of A, then each B-object is also an A-object. In particular this means that each B-object can be manipulated as an A-object. Suppose we have a user of the database who is only aware of A-objects. As B is a subclass of A, this means that she can *see* both A- and B-objects, but she cannot see that these are different kinds of objects. So, this user will treat both kinds of objects completely the same and thus apply A-methods to B-objects.

This means that methods should be inherited downwards in the subtyping hierarchy. As an aside, note that this explains the idea from the first subsection, to model methods by 'exponentially typed attributes' as shown in the first section of this chapter. Because attributes inherit downwards also, i.e. if a is an attribute of A and B is a subtype of A, then a is an attribute of B.

In principle, there are two ways to ensure that a method works on two different structure types. The first defines the method for each type separately, and gives them the same name. This *overloading* of the name gives the appearance that the different methods are the same method. The other approach is defining one method which is applicable to both types. The former approach is chosen e.g. in O_2 (Lecluse and Richard (1989)), it is rather a pragmatic approach than a satisfying solution to the problem of method inheritance. We choose the second approach using the $C^{\rightarrow}$ semantics.

As already indicated in the previous sections, at the entity type level a method can simply be seen as a dynamic entity, i.e. an entity of an exponential type. Now consider a structure type s, each structured object o of type s is completely

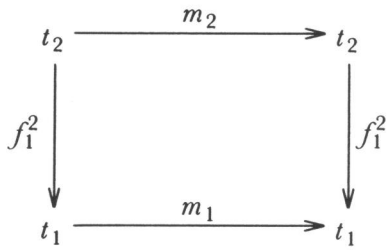
determined by its 'root value'. So, a method that maps objects of type s to objects of type s is completely specified by the way it maps root values to root values. More general, if s_{t_1} and s_{t_2} are structure types, a method from s_{t_1} to s_{t_2} is completely specified by a dynamic entity of type $t_2^{t_1}$.

Combining these two ideas, we get:

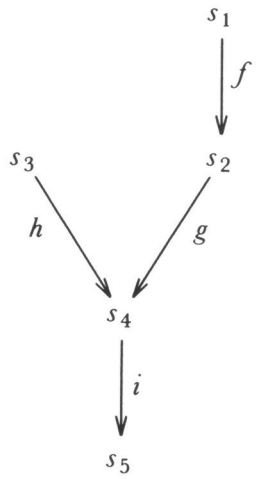
1) Let m be a method that is applicable to structured objects of type s_{t_1}. Moreover, let m transform s_{t_1} objects into s_{t_1} objects. So, from the s_{t_1} point of view m is a dynamic entity of type $t_1^{t_1}$. Denote this entity by m_1.

2) Let s_{t_2} be a subtype of s_{t_1} with the function f_1^2 establishing the subtype relationship. Then m is also applicable to s_{t_2}. Assume we want the inheritance of m to be such that m transforms s_{t_2} objects in s_{t_2} objects. Again, we have that m is a dynamic entity of type $t_2^{t_2}$ from the point of view of s_{t_2}. Denote this entity by m_2.

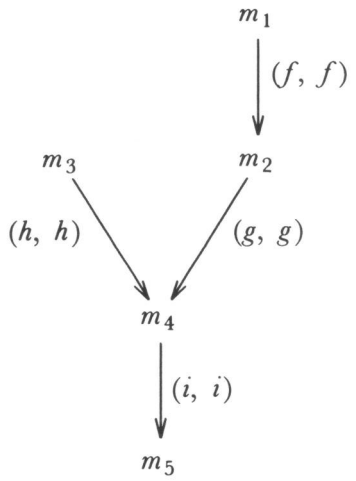
3) From our discussion in the previous section, we know that the relationship between m_1 and m_2 is given by the following commuting diagram:



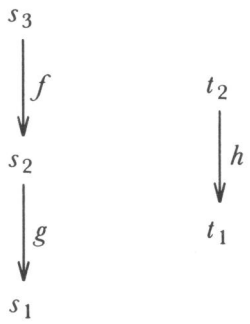
Generalising somewhat, if the structured types $s_1, \ldots, s_5$ are in the following subtype relationship (with the functions establishing the subtyping):



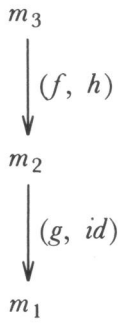
then a method m, which should map s_i to s_i has the following structure $(m_i{:}s_i \to s_i)$:



And finally, to give an example, in which source and target types of the method do not coincide, suppose we have the structure types s_1, s_2, s_3, and t_1 and t_2 with the following subtyping relationships:



Moreover, let m be a method mapping s_1 objects to t_1 objects, and s_2 and s_3 objects to t_2 objects. Then m consists of the functions m_1, m_2 and m_3 and has the following structure:



So, methods can be defined as subgraphs of $C^{\rightarrow}$. In analogy with structured objects, methods are given a unique *sink*, i.e. a unique lowest structure type it is applicable to. This informal definition will be the basis for our formal definition in the next section. However, before we turn to that section, we first look at the question of automatic inheritance of methods. But before we turn our attention to this topic, we show a curious effect of the fact that methods are inherited anti-monotonically rather than monotonically:

Let s_{r_1} and s_{r_2} be structure types, such that s_{r_1} is a subtype of s_{r_2} established by the function $f: r_1 \rightarrow r_2$. Moreover, let m be a method such that m is defined for s_{r_1} and acts as $m: s_{r_1} \rightarrow s_{r_1}$, but not for s_{r_2}. Note that this is consistent with the method structure as described in this section. Now, each s_{r_1}-object is also an s_{r_2}-object.

Now suppose that we have two observers, A_1, who can distinguish between s_{r_1}-objects and s_{r_2}-objects and A_2 for whom the world consists only of s_{r_2}-objects. Now suppose that A_1 applies m to some s_{r_1}-object o. So, for A_1, o changes from o to $m(o)$, which she would expect. For A_2, curious things may happen if $f \neq f \circ m$, because in this case $m(o)$ is a different s_{r_1}-object from o itself. So, A_2 observes an s_{r_2}-object change, but she has no method to explain this change.

The first reaction of the reader might be that these kind of methods should be forbidden, as they perform miracles for certain observers. This would mean that we would have to adjust our picture of method structures considerably. However, there is no ground for this reaction, as it simply illustrates that A_2 does not have enough knowledge of the world to explain all events. In this particular case, A_2 misses the information provided by m. If A_2 knew about m, no miracle would have been reported.

6.3.2 Automatic inheritance

The types in Flock are defined inductively and, as we have seen in the previous subsection, the structure of a method reflects the type structure. So, a natural question is whether this structure automatically defines the method; all the more as we indicated that methods have a unique sink. In general, if a canonical method inheritance mechanism exists, it conists of trivial inheritance. However, there are several cases for which a more interesting default mechanism can be given.

First of all, we show that in general only the trivial inheritance, if any, can be generated automatically.

As methods have a unique sink, we may assume that we start with a function $m_1: A \rightarrow B$ which defines a method m from type s_A to type s_B. Let s_C be a subtype of s_A, with $f: C \rightarrow A$, so the method m has to be applicable to objects of type s_C.

1) Now, s_B may have many subtypes and hence, it is a priori unknown to which of the subtypes of s_B the method m applied to s_C-objects is supposed

to map to. If this is all that is known, the only reasonable suggestion is the definition of the function m_2 with $eval(m_2, x) = eval(m_1, f(x))$. But this means we choose for the trivial inheritance.

2) Let s_D be a subtype of s_B with the function $g: D \to B$. Moreover, assume that m should map s_C objects to s_D objects. If this is all there is known, there is no reasonable suggestion for the function $m_2: C \to D$, as in general the set $\{h \mid m_1 \circ f = h \circ g\}$ contains more than one element and there is no reason to prefer one above the other.

As an aside, note that above, we more or less dismissed the trivial inheritance as uninteresting. This does not imply that there are no cases in which the trivial inheritance is the best choice. An example of this situation is in fact already given in section 1 of this chapter. The only possible way that the method *make-older* from *thing* to *thing* can be inherited by *natural thing* is by trivial inheritance.

From the two cases examined above, we see that quite some information is needed to generate 'inheritance'. The first example for which it is possible can be given as follows:
Let the method m be already specified by the functions $m_1: a_1 \to b_1$ and $m_2: a_2 \to b_2$. Moreover, let both $s_{a_1 \times b_1}$ and $s_{a_2 \times b_2}$ be structure types. Then:

$$m_1 \times m_2: a_1 \times b_1 \to a_2 \times b_2$$

is a specialisation of both m_1 and m_2. Hence, m_3 is a candidate to specify the inheritance of m. Of course, as above, there may be many more candidates, however, m_3 seems the natural candidate.

We can generalise the above example slightly, if we assume that m does not change the type of a structured object, i.e. m is specified by functions $m_i: a_i \to a_i$. Let m be a method that does not change types. Moreover, let $s_{a_1 \times a_2}$, s_{a_1} and s_{a_2} be structure types. There are two cases for which we can define a default inheritance:

a) As above, m is specified for both s_{a_1} and s_{a_2}, i.e. m is already specified by functions $m_1: a_1 \to a_1$ and $m_2: a_2 \to a_2$. Then we have the default inheritance by:

$$m_1 \times m_2: a_1 \times a_2 \to a_1 \times a_2.$$

b) Let m be specified for s_{a_1} by the function $m_1: a_1 \to a_1$ and assume that s_{a_2} is not a sub-type of the sink-type of m. In other words, we assume that m can not be applied to structured objects of type s_{a_2}. Then the inheritance of m should not 'act' on the s_{a_2}-part of an $s_{a_1 \times a_2}$-object. Hence, the default inheritance is given by:

$$m_1 \times id: a_1 \times a_2 \to a_1 \times a_2$$

Clearly, it is also possible to descripe 'inheritance' of a method m on s_{a_1} by $s_{a_1 + a_2}$ and $s_{P(a_1)}$. Both cases describe inheritance in the sense that $a_1 + a_2$ and a_1 are necessarily constructed after a_1. Note that both cases are generaly not considered, as this type of inheritance does not coincide with the inheritance hierarchy. But anyway, the default in these cases could be defined as:

$$m_1 + id\colon a_1 + a_2 \rightarrow a_1 + a_2$$

and

$$m_1^*\colon P(a_1) \rightarrow P(a_2)$$

respectively.

From the discussion in this section, we see that in general automatic inheritance is only possible in special cases. In general, the inheritance of methods should be given explicitly. However, this observation is technically correct, but not necessarily practically. Because, from a theoretical point of view, objects may be mapped from one type to another, but in reality such migrations are limited. To give an example, an object of type *person* will always remain an object of type *person*.

This observation implies that the special cases for which a default mechanism is given above are more important than one is led to think from a theoretical point of view. In other words, we hypothesize that the rules formulated above cover a large part of 'every day life method inheritance'

6.4 Method types informally

In the previous sections, we offered some intuition concerning methods. Clearly methods need to be typed and thus we need method types. In this section, we informally discuss such types in order to improve our intuition on how such types should be defined. This section is, again, divided into two subsections. In the first subsection, we develop the notion of a second-order entity type from the intuition of methods as developed before. In the second subsection, we discuss how method types or second-order structure types can be defined from these second-order entity types. Moreover, we briefly discuss the expressiveness of an inductively defined hierarchy of type orders.

6.4.1 Second-order entity types

Informally, a method type can, of course, be seen as a set of methods. Now, in the previous section, we have seen that methods can be seen as subgraphs of $C^{\rightarrow}$. So, it is to be expected that the formal semantics of method types are also given in terms of $C^{\rightarrow}$. However, this seems to be impossible. To illustrate why, consider a UoD with two structure types, s_{a_1} and s_{a_2}. Moreover, let s_{a_1} be a subtype of s_{a_2}, with the function $f\colon a_1 \rightarrow a_2$ establishing this subtype relationship. From

the previous section, we know that a method m on s_{a_2}, and thus inherited on s_{a_1} is given by two functions $m_1: a_1 \to a_1$ and $m_2: a_2 \to a_2$, such that $f \circ m_1 = m_2 \circ f$. So, the type of m is informally the set of all pairs (m_1, m_2) that satisfy this requirement. Now, m_1 and m_2 are objects in $C^{\to}$, but collections of functions from a_1 to a_1 are *not* elements of $C^{\to}$.

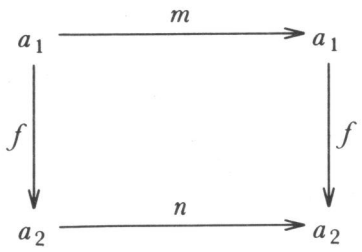
So, we cannot use $C^{\to}$ to give semantics to the types. However, the above analysis already contains a hint on how we should give semantics to types: 'the type of m is informally the set of all pairs (m_1, m_2) that satisfy this requirement'. The pair (m_1, m_2) is an element of the type $X = a_1^{a_1} \times a_2^{a_2}$. Those pairs that satisfy the requirement are a subobject Y of X, given by:

$$Y = \{x \in X \mid f \circ \pi_1(x) = \pi_2(x) \circ f\}$$

using the comprehension principle of Chapter 6.

Clearly, a similar construction can be used for any method type. Hence, we can give method types a semantics in C. The disadvantage of this solution seems to be that, in contrast to structure types, such method types do not reflect the structure of methods clearly. But we should not identify this kind of types with structure types, but with entity types; since, a method that is defined for a structure type s and all its subtypes, is also defined for a subtype s_1 of s and all of *its* subtypes, i.e. again we have subtyping. This implies that method types should be defined as 'second-order' structure types over second-order entity-types!

We note that, it is still possible to see a method, or better an 'element' of a second-order type, as a subgraph of $C^{\to}$. In section 2 of this chapter, we made the assumption that elements of exponential types that are used should be the name of arrows. In particular this means that for $(k, l) \in Y$ (Y as defined above) we have functions $m: a_1 \to a_1$ and $n: a_2 \to a_2$ such that $\hat{m} = k$ and $\hat{n} = l$. So, we can map (k, l) to the structure:



The fact that we can still see methods conceptually as substructures in $C^{\to}$ will prove to be important in later chapters of this thesis.

Secondly, we note that it is possible to devise a semantics which is closer to the $C^{\to}$ intuition we have of methods. For example, we could define a category, say $\{C^{\to}\}$, whose objects are sets of C-arrows with the same source and the same target. Let O_1 and O_2 be two such objects, such that the source of the O_1 elements is s_1 and the target is t_1. Similarly, the source of O_2 objects is s_2 and the target is t_2. Then a $\{C^{\to}\}$-arrow from O_1 to O_2 could be defined as a pair (k, l) of C arrows, such that for each $f \in O_1$ there exists a unique $g \in O_2$ such that

$g \circ k = l \circ f$. However, this category is not a topos, which can, e.g., be seen by the fact that it does not have an initial element. Below we will discuss that the fact that second-order entity types have their semantics in a topos opens a world of possibilities. Therefore, we choose the semantics as indicated above.

In the next subsection, we sketch a partial order on the second-order entity types, based on projections. This means that we have a graph of second-order entity types, called, say, the second-order type graph. Moreover, the semantics of the second-order entity types are again given in C, as sketched above; and C is a topos. So, we could consider this second type graph as a basic type graph and construct new second-order types by the usual constructions. This may seem a byproduct of our definitions without much relevance. However, that is not true, which we known explain.

To give an illustration of the use of such new constructed types we look at the product construction. The methods we have taken into consideration up till know have structure types as their input type and as there output type. This means that we can only describe methods whose input type is an existing structure type. In other words, we cannot describe a method that takes an object of type s_1 and an object of type s_2 as input and convert it to say an s_3 type unless there is a structure type $s_1 \times s_2$. One can take two points of view concerning this problem:

1) If the database designer wants to define such a method, he simply has to add the structure type $s_1 \times s_2$.

2) Support products of second-order entity types.

Clearly, the second option is the most elegant. Not only from a formal point of view, but also of a database design point of view. Because the first option requires the definition of structure types, and thus entity types, for which no natural semantics in the UoD exists. These artificial entity types are only constructed so that all perceived dynamical aspects of the UoD can actually be described. As an illustration consider a UoD in which *persons*, *papers*, and *locations* exist. Moreover, suppose that we perceive that the persons transport papers from one location to another (so, our UoD resembles an office). To model this transportation with a method under the first option, we have to define the type *person* $\times$ *paper*. Moreover, objects of this type only exist for the time that a person actually carries a certain paper.

From this discussion, it will be clear that we prefer the second option. And, as indicated above it is a straightforward extension of Flock as we have defined it up to this point. However, we will not carry out this generalisation for the simple reason that it would take up too much space. Therefore, we stand by our observation of the usefulness of products et cetera of second-order entity types, but we defer the actual description to the future.

6.4.2 Second-order structure types

We already indicated that the second-order entity types can easily be embedded in a graph. In this subsection, we actually sketch this graph and show how it can be used to define second-order structure types. Moreover, we show how this process gives rise to N-th order entity types.

The functions between the second-order structure types are essentially projections. To see how these functions are derived, consider a UoD with the structure types s_{a_1}, s_{a_2}, and s_{a_3}. Moreover, suppose that s_{a_1} is a subtype of s_{a_2}, established by $f: a_1 \rightarrow a_2$ and that s_{a_2} is a subtype of s_{a_3}, established by $g: a_2 \rightarrow a_3$. Then, in the notation of the second subsection, we get that a method defined on s_3 (and higher) is of type Y_1, defined by:

$$X_1 = a_1^{a_1} \times a_2^{a_2} \times a_3^{a_3}$$

$$Y_1 = \{x \in X_1 \mid f \circ \pi_1(x) = \pi_2(x) \circ f \wedge g \circ \pi_2(x) = \pi_3(x) \circ g\}$$

Similarly, methods defined on s_2 (and higher) are of type Y_2, defined by:

$$X_2 = a_1^{a_1} \times a_2^{a_2}$$

$$Y_2 = \{x \in X_2 \mid f \circ \pi_1(x) = \pi_2(x) \circ f\}$$

Obviously, (π_1, π_2): $X_1 \rightarrow X_2$ induces a map π: $Y_1 \rightarrow Y_2$. This map is the edge between the second-order entity types Y_1 and Y_2 in the second-order type graph.

The second-order type graph is similar to the normal type graph. Hence, we can define second-order structure types analogical to structure types on the usual type graph. So, second-order structure types are essentially subgraphs of the second-order type graph with a unique root type. Similar to structure types, we can now define a subtyping relationship on second-order structure types. The second-order structure types are what could be called the method types. The subtyping relationships of the method types, reflect the structure of a method seen as a subgraph of $C^{\rightarrow}$. Moreover, it ensures that methods are polymorphic.

Now that we have informally defined second-order structure types, it is possible to define third-order structure types et cetera. The third-order structure types would be the type of second-order methods that act upon first order methods. So, these meta-methods could be used to model changes in methods! More general, objects of an n+1-order type are the methods for the n-order type objects. Clearly it is useful for a formalism to be able to express such higher order changes in a UoD. However, for the same reason as in the previous section, we have to defer a complete description of this feature of Flock to the future.

6.5 Formal definitions

Until now, we discussed the semantics of methods and their types, or better second-order entity types and second-order structure types without giving the

syntactic definition of such animals. Clearly, we can not give a formal semantics prior to the syntactic definition. Therefore, we will first define second-order entity type syntactically. There are a few things we have to bear in mind in this definition:

1) If a method is defined on a structure type s, it is inherited by all subtypes of s. This means that all these subtypes have to be taken into account in the definition.

2) We have seen that methods have unique sinks, this sink should be explicitly given in the definition; which means that method types should have unique sink-types.

3) In the previous section, we have seen that it is not a priori clear what the goal of a method m on a structure type s_1 should be; even if the goal of m is known on a supertype s_2 of s_1. So, for each of the subtypes of s, the goal-type should be specified in the definition.

4) Finally, the goal types are not unrelated. Indeed, to ensure that the elements of the type are methods as we discussed in the previous sections, we should have the following requirement:
If the goal-type of s_1 is t_1, and the goal type of s_2 is t_2, and s_1 is a subtype of s_2, then t_1 should be a subtype of t_2.

Taking these considerations into account, we define:

DEFINITION 6.3
A second-order entity type, abbreviated by *se-type*, is defined by the following expression:

se-type name = •

sink-type: s_1 (*a structure type*)

sources and *targets*:

$s_1 \rightarrow t_1$

.

.

.

$s_n \rightarrow t_n$

such that:

1) the s_i and the t_i are structure types;

2) each subtype of s_1 is a source exactly once.

2) If t_i and t_j are the target types of s_i and s_j, respectively, and s_i is a subtype of s_j, then t_i is a subtype of t_j.

In the previous section, we already sketched the semantics of se-types. Therefore, we immediately define the formal semantics of such a type:

DEFINITION 6.4

Let *soet* be the se-type defined by:

se-type soet =

sink-type: s_1 (*a structure type*)

sources and *targets*:

$s_1 \rightarrow t_1$

.

.

.

$s_n \rightarrow t_n$

The semantics of *soet* are given by the object *Sem*(*soet*) in C that is constructed as follows:

1) First construct the type:

$$X_{soet} = s_1^{t_1} \times \cdots \times s_1^{t_1}$$

2) In the following step, we use the same symbol for a structure type and its root entity type, to avoid a cluttering of sub- and super-scripts. Define the set V as follows:

$$V = \{(f, g) \mid f: s_i \rightarrow s_j, g: t_i \rightarrow t_j \text{ in the def of soet}\}$$

Furthermore, define the predicate ϕ_{soet} by:

$$\phi_{soet}(x, f, g) \Leftrightarrow g \circ \pi_i(x) = \pi_j(x) \circ f$$

where $(f, g) \in V$ and $f: s_i \rightarrow s_j$ and $g: t_i \rightarrow t_j$. The predicate $\Phi_{soet(x)}$ is now defined by:

$$\Phi_{soet}(x) \Leftrightarrow (\wedge_{(f, g) \in V} \phi_{soet}(x, f, g))$$

3) Finally, we get to the semantics:

$$Sem(soet) = \{x \in X_{soet} \mid \Phi_{soet}(x)\}$$

Note that the conjunction $\Phi(x)$ contains equations for all functions between two structure types, this ensures that there can be no 'clashing' in the inheritance of

methods. In other words, if there are two paths through which a method m can be inherited, both these paths are taken into account.

The following step is the construction of the second-order type graph, TG_2. Syntactically, this is straightforward:

DEFINITION 6.5

The second-order type graph, denoted by TG_2 is defined as follows:

1) The nodes are the second-order entity types.
2) If the sink-type of node n_i is a subtype of the sink-type of node n_j, then there is an edge e_i^j: $n_j \rightarrow n_i$

The semantics are a bit trickier but, as indicated in the previous section, the semantics of e_i^j can for all practical purposes be seen as the projection from X_j to X_i:

DEFINITION 6.6

Let TG_2 be a second-order type graph, semantically, TG_2 is a subgraph of C. The mapping *Sem* is given by:

1) A node n_i is mapped to $Sem(n_i)$ as defined above.
2) The edge e_i^j is mapped to the unique function f: $Sem(n_j) \rightarrow Sem(n_i)$ induced by the universal property of $Sem(n_i)$ and the projection function $X_j \rightarrow X_i$.

Now that we have defined the second-order type graph, we can define second order structure types analogical to the definition of structure types, with the exception that we do not have to define an equivalence type graph, as there are no cycles in a second-order type graph. So, we get:

DEFINITION 6.7

Let TG_2 be a second-order type graph, and let n_i be a node in TG_2. The second-order structure type (also called method type) m_{n_i} induced by n_i is defined syntactically by the expression:

$$sos - type\ m_{n_i} = (n_i, TG)$$

n_i is called the root type, or simply the root, of m_{n_i}.

The semantics of second-order structure types are defined analogical to the semantics of structure types, i.e. we use again the category $Fsub(C)$ of finite subcategories of C:

DEFINITION 6.8

Let $m_{n_i} = (n_i, TG_2)$ be a second-order structure type and let $G(n_i, TG_2)$ be the maximal connected directed subgraph of TG_2 with n_i as root. Then the semantics of m_{n_i} are given by the element $Sem(m_{n_i})$ of $Fsub(C)$ which has:

$$Nodes(Sem(m_{n_i})) = \{Sem(n_j) \mid n_j \in Nodes(G(n_i, TG_2))\},$$

$$Edges(Sem(m_{n_i})) = \{id_n \mid n \in Nodes(Sem(m_{n_i}))\} \cup \{Sem(e) \mid e \in Edges(G(n_i, TG_2))\}$$

Finally, we can define methods:

DEFINITION 6.9

A method, or second-order structured object, is an element of a second-order structure type.

Similar to the subtyping relationship on structure types, we can define a subtyping relationship on second-order structure types. This subtyping is less important, as expressions over such types are, again, beyond the scope of this paper. However, this definition shows that we can (sub)-type methods by the usual subtyping schema, something which seemed impossible in the first section of this chapter. In other words, this definition shows the solution to the problem which we posed, in the first section, as the theme of this chapter:

DEFINITION 6.10

Let mt_1 and mt_2 be method types. We define subtyping of method types as follows:

$mt_1 \leqslant mt_2$ iff $Sem(mt_2)$ is a subcategory of $Sem(mt_1)$.

This definition concludes our definition of methods, their types and their semantics. Our last task is to give semantics to the application of a method to a structured object. This is the subject of the next and final section of this chapter.

6.6 Methods and structured objects

In the previous sections, we have defined methods and their typing. This typing schema is completely different from the typing of structured objects. Yet, methods are supposed to act on structured objects. Hence, we have to reconcile these two typing schemas to give semantics to the application of a method to a structured object. Now, let m be a method, and let o be a structured object. Considering the expression $eval(m, o)$ we have two questions to answer:

1) When is this expression well-typed?

2) If the expression is well-typed, what is its semantics?

The latter question can be rephrased as the question: what is the result of this application?

Obviously, for the definition of a formal semantics for this expression, we need to know whether the expression is well-typed. So, the two subquestions should be answered in the order they are posed. However, to gain some intuition on how the definitions should look like, it is worthwhile to start with the second question.

If x is an entity of type t and f is an entity of type s^t, it is clear what the semantics of the application of f to x is, as $eval(f, x)$ is defined in the topos. Now, for a structured object o_1 of structure type s_{r_1}, we know that o_1 is completely determined by an entity o_{r_1} of type r_1. Similarly, a structured object o_2 of structure type s_{r_2} is completely determined by an entity o_{r_2} of type r_2. So, a 'method' from s_{r_1} to s_{r_2} is determined by a function $m: r_1 \to r_2$.

In the definition of methods (and their types), we already took this into account. Similar to structure types, method types are completely determined by their root *se-type*. Such *se-types* can be seen as tuples of functions of the form $s_1^{t_1} \times \cdots \times s_n^{t_n}$. Now suppose that there is an i such that $t_i = r_1$; note that there is at most one such i, as all the t_i are different. Then, we know how to apply an $s_1^{t_1} \times \cdots \times s_n^{t_n}$-entity m to an r_1 entity o: simply project m on its i-th coordinate and apply the result, i.e.:

$$eval(m, o) = eval(\pi_i(m), o)$$

Moreover, if there is no such i, there is no way we can give semantics to this expression. So, by reversing our argument, we get that the expression $eval(m, o)$ is well-typed iff the root type of m contains a 'factor' s^t, where t is the root type of o. Moreover, if the expression is well-typed, the semantics are as indicated above. So, the intuition is as follows:
Let *mt* be a method type, and s a structure type, moreover,

1) let $rmt = s_1^{t_1} \times \cdots \times s_n^{t_n}$ be the root se-type of *mt* and

2) let t be the root entity type of s.

then a method m of type *mt* can be applied to a structured object o of type t iff *rmt* contains a 'factor' s^t.

To formalise this intuition, we have to take care of the fact, that in general, the root type of a method type is a subobject of a $s_1^{t_1} \times \cdots \times s_n^{t_n}$-type. Moreover, we have to formalise the notion of a factor. Both these problems can be solved by subtyping. For the first problem, this is directly given by the fact that the actual type is a subobject of *rmt*. For the latter problem, we are going to associate a structure type to each method type as follows:
Let *rmt* be the root es-type of the method type *mt*. $Sem(rmt)$ is defined as a

subobject of X_{rmt}. This object X_{rmt} is of the form:

$$X_{rmt} = s_1^{t_1} \times \cdots \times s_n^{t_n}$$

The structure type *smt* is the structure type which has *Sem*(*rmt*) as its root type, the inclusion arrow i: *rmt* $\rightarrow X_{rmt}$, and for the rest it consists of the structure type induced by X_{rmt}

DEFINITION 6.11

Let *mt* be a method type, with *rmt* as its root es-type, its associated structure type *smt* is as defined above.

Now, each method of type *mt* can be be seen as a structured object of type *smt*. To make this more precise, we need to make the distinction between the two typing schemes clear. We will write m ; *mt* to denote that m is a method of method type *mt* and m : *smt* to denote that a structured object of type *smt*. Then we have:

LEMMA 6.12

Let m ; *mt*, then m induces as structured object *sm* : *smt* and vice versa

PROOF

Each method m is completely determined by its value for the root es-type *rmt*, i.e. by a value of *Sem*(*rmt*), but *Sem*(*rmt*) is the root entity type of *smt*. □

In the remainder of this thesis, we will often write m : *smt* rather than *sm* : *smt* for the induced structured object. The context will allow the reader to disambiguate this statement. In particular, this means that we have: m ; *mt* iff m : *smt*.

Now, we can formalise our intuition about well-typed expressions, and the semantics of an expression of the form *eval*(m, o) as follows:

DEFINITION 6.13

Let m be a method of type *mt* with root es-type *rmt*, and o an object of type t, then the expression:

$$eval(m, o)$$

is well-typed if there is a structure type s_r, such that *smr* is a subtype of r^t. Moreover, if f : *Sem*(*rmt*) $\rightarrow r^t$ establishes this subtype relationship, then the semantics of the expression are given by:

$$Sem(eval(m, o)) = eval(f(m), o)$$

Note that $f(m)$ should be read as $f(sm)$ with *sm* the structured object induced by m.

6.7 Conclusions

Structured objects and structure types as defined in the previous chapter are not particularly adequate to represent methods and method types. In fact, it is well-known that anti-monotony in Cardelli's typing rule for exponential types 'blocks' the straightforward solution for method inheritance. As our subtyping rules entail Cardelli's rules, we cannot use this straightforward solution either. In this chapter, we show a solution to this problem, using alternative semantics in the category $C^{\rightarrow}$. Moreover, we show that method types can be seen as the usual graph-structure in these alternative semantics.

The object-oriented ideal is that methods are inherited automatically. In other words, one would like that methods defined for some type s are automatically extended to all subtypes t of s. It is shown that in general this is in general impossible. However, we show that in restricted cases automatic inheritance is possible. The restrictions are:

1) Only products are used in the type constructions.

2) The methods preserve the type of the objects.

In this case, a method can be inherited by assuming it acts as the identity on other components.

Implicit in the discussion above, is the assumption that the structure types are known at the time that the methods and their types are defined. Therefore, we use the term 'second-order' types when we formalise the description above in the last part of this chapter.

Chapter 7

Logic and Set Theory in a topos

In this chapter, we introduce basic logic and set theory within the context of a topos. There are several reasons to do this. For set theory, the most important reasons are:

1) A database can be seen as a set of extensions, one for each type, each extension is a set of objects. Querying the database can then be seen is manipulating extensions set-theoretically.

2) Flock has the power type construction, that yields $P(t)$ for a given type t. An element of $P(t)$ is a set of elements from t. So, we should be able to manipulate elements of $P(t)$ as sets.

For the logic, the most important reasons are:

1) One of the objectives for Flock is that it supports the representation of incomplete information such as: the colour is either red or brown.

2) As in ordinary set theory, the logic can be used in the definition of the set theory.

Hence we need both logic and set theory. Now the semantics of Flock are defined in a topos. So, the logic and the set theory have to be defined within this categorical framework. In fact, we will show that each topos has its own notion of logic and set theory. Moreover, this private version of logic does not have to be classical.

All the definitions and results given in this chapter can be found in any book on topos theory. In fact, the connection between logic and set theory and category theory is far deeper than we can cover in this chapter, c.f. the 551

pages of (Goldblatt (1979)). Therefore, we can only hope to give an introduction. Moreover, this introduction will only mention the facts that we need and will sometimes be more descriptive than rigorous.

The reader who is less interested in the formal semantics of Flock or in category theory for that matter may skip this chapter. The important result of this chapter he should remember is that the logical and set-theoretic expressions we will introduce in Flock have a formal semantics within the topos C. Moreover, this logic does not have to be classical, e.g. $\phi \vee \neg\phi$ does not have to be a tautology.

For the reader who remains with us, we give an outline of this chapter:
In chapter 4, we have used two characterisations of subsets, viz.:

1) $A \subseteq B$ iff $A \in P(B)$,

2) $A \subseteq B$ iff there is a monic function $i: A \to B$.

Moreover, we showed how intersections can be defined in both characterisations. In the first section, we repeat these constructions for completeness and show how the union can be defined similarly. Moreover, we make the first steps towards the reconciliation of the two views on subsets, or better subobjects. In the second section, we introduce the *subobject classifier* Ω together with its universal property, often called the $\Omega-$*axiom*. Moreover, we characterise subobjects using Ω, and thus reconciliate the two views on subobjects. In the next section, we define the *logic* of the topos as induced by Ω, and show that in general, this logic is intuitionistic rather than classic. As a further illustration of this logic, we show how it can be used to define the two most important predicates in query languages, viz. $\in$ and $=$.

In the fourth and fifth section we show how this logic can be used to define the set-theoretic constructions abstractly and that the naive definitions of the first section are equivalent to their abstract counterparts. In the sixth section, we define the *quantifiers* categorically. In the final section, we finally look at *natural number objects* in a topos. Not because we will use these objects in Flock, but to illustrate that usual type theory can also be given categorical semantics.

7.1 Intersections and unions naively

In chapter 4, we have used two characterisations of subsets, viz.:

1) $A \subseteq B$ iff $A \in P(B)$,

2) $A \subseteq B$ iff there is a monic function $i: A \to B$.

Moreover, we showed how intersections can be defined in both characterisations. To refresh the readers memory and to keep this chapter complete, we give these constructions again.

For the first construction, we characterise subobjects by $A \subseteq D$ iff $A \in P(D)$. The intersection under this characterisation is defined as a function $\cap_{P(D)}: P(D) \times P(D) \to P(D)$. As an element of an object X is defined categorically as an arrow $x: 1 \to X$, this function allows us to compute the intersection of two elements of $P(D)$.
The function $\cap_{P(D)}$ can be constructed using the universal property of the power type construction. For $\Theta, \Upsilon \in P(D)$:

$$\cap_{P(D)}(\Theta, \Upsilon) = \{\theta \mid \theta \in \Theta \wedge \theta \in \Upsilon\}.$$

Now, consider $\epsilon_D \times \epsilon_D$, this type has the injective function:

$$\in \times \in : \epsilon_D \times \epsilon_D \to P(D) \times D \times P(D) \times D.$$

Moreover, $(\Theta, \theta, \Upsilon, \upsilon) \in \in \times \in (\epsilon_D \times \epsilon_D)$ iff $\theta \in \Theta$ and $\upsilon \in \Upsilon$. Let π_1 and π_2 denote the two projections from $P(D) \times D \times P(D) \times D$ to D. Now, let $i: E \to \epsilon_D \times \epsilon_D$ be the equaliser of $\pi_1 \circ (\in \times \in)$ and $\pi_2 \circ (\in \times \in)$. Then E 'is' the set of $(\Theta, \theta, \Upsilon, \upsilon)$ tuples for which $\theta = \upsilon$. Moreover, $(\in \times \in) \circ i$ is an injective function. In fact, we have an injective function $j: E \to P(D) \times P(D) \times D$ as both D-components in an E-element are the same. Hence, the universal property of power types gives us the intersection function $\cap_{P(D)}: P(D) \times P(D) \to P(D)$ with:

$$\begin{aligned} \cap_{P(D)}(\Theta, \Upsilon) &= \{\theta \mid (\Theta, \Upsilon, \theta) \in E\} \\ &= \{\theta \mid \theta \in \Theta \wedge \theta \in \Upsilon\} \end{aligned}$$

For the second construction, the characterisation of subobjects is given by $A \subseteq B$ iff there is a monic function $i: A \to B$. So, let A, B, and C be sets with injective functions $i: A \to C$ and $j: B \to C$. We now claim that the pullback of i and j is essentially the intersection of A and B as subsets of C. To justify this claim, we spell out the construction of the pullback:

1) Let $D = A \times B$, consisting of pairs (a, b) with $a \in A$ and $b \in B$ or, as we see A and B as subsets of C, it consists of pairs (c_1, c_2), with $c_1 \in A \subseteq C$ and $c_2 \in B \subseteq C$. By definition, we have two arrows:

$$i \circ \pi_A: D \to C, j \circ \pi_B: D \to C$$

2) Let E be the equaliser of these two functions, i.e. E is the set:

$$\{(c_1, c_2) \in D \mid i \circ \pi_A((c_1, c_2)) = j \circ \pi_B((c_1, c_2))\}$$

Hence, it is straightforward to see that E consists of the pairs (c, c) such that $c \in A$ and $c \in B$. Moreover, as E comes by definition with an injective function $k: E \to D$, we have the injection

$$i \circ \pi_A \circ k = j \circ \pi_B \circ k: E \to C$$

Hence, E is essentially the intersection of A and B

Note, we say essentially, because set-theoretically E consists of pairs of C elements rather than C elements. However, categorically everything is defined up to isomorphism. Since E and *the* intersection are isomorphic, E is the intersection.

As a further illustration of the second characterisation of subobjects, we will now show that it can also be used to construct the union of two subobjects. This construction uses the epic/monic factorisation of a function.

Let A, B, and C be three objects such that A and B are subobjects of C in the same sense as above, i.e. there are injective (monic) functions $i: A \to C$ and $j: B \to C$. Then by the universal property of the co-product, we have the function:

$$[i, j]: A + B \to C.$$

Let $g: A + B \to D$ and $h: D \to C$ be the epic/monic factorisation of $[i, j]$, we claim that $D = A \cup B$. By definition, $D = [i, j](A + B)$. Hence, D consists of those elements c of C for which either there is an $a \in A$ such that $i(a) = c$ or there is an $b \in B$ such that $j(b) = c$.

Summarising the discussion above, we have two characterisations of subobjects and both characterisations can be used to define e.g. intersections. A main task is of course to reconciliate these two view points and to show that both definitions of intersections coincide. The first steps towards the reconciliation of the two viewpoints on subobjects are made in the remainder of this section.

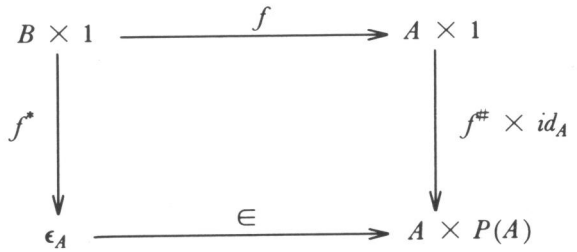
The first step is to show that $B: 1 \to P(A)$ induces a monic arrow $i: R \to A$ and vice versa that a monic arrow $i: B \to A$ induces an arrow $1 \to P(A)$:

1) Let B be an element of $P(A)$, i.e. $B: 1 \to P(A)$. Then $B \times id_A: 1 \times A \to P(A) \times A$. By the definition of power types, we have the monic function $\in: \epsilon_A \to P(A) \times A$. By taking the pullback of these two functions, we get an object R together with a monic function $B^{\#}: R \to 1 \times A$. Since $1 \times A \equiv A$, we might as well say, we have the monic function $B^{\#}: R \to A$. So, for every element of $P(A)$ we have a monic function into A. Moreover, set-theoretically

$$R = \{(x, y) \in (A \times 1) \times \epsilon_A \mid B \times id_A(x) = \in(y)\}$$

and thus R denotes the 'right' subset of A.

2) Conversely, let $i: B \to A$ be a monic arrow, since $A \equiv A \times 1$, we might as well say that we have a monic arrow $f: B \times 1 \to A \times 1$. Thus, by the universal property of the power type construction, we have unique functions $f^*: B \times 1 \to \epsilon_A$ and $f^{\#}: 1 \to P(A)$ such that the following diagram is a pullback:



So, with each monic into A corresponds an element of $P(A)$. Moreover, if we construct the pullback of $id_A \times f^*$ and $\in$ set-theoretically, we see that B has to be isomorphic to

$$R = \{(x, y) \in (A \times 1) \times \epsilon_A \mid (id_A \times f^*)(x) = \in(y)\}$$

and, hence, that f^* is the right element of $P(A)$.

The correspondence found above is not bijective, therefore, we investigate this relationship a bit deeper.
Let $i: B \to A$ and $j: C \to A$ be monic functions, define $i \equiv j$ if there are functions $h: B \to C$ and $k: C \to B$ such that $i = j \circ h$ and $j = i \circ k$. Obviously, this defines an equivalence relation on the set of all monic functions with codomain A. Denote by $[i]$ the set of all monic functions into A equivalent to i in this sense. Now we can define the set:

$$Sub(A) = \{[f] \mid f: X \to A, f \textit{ monic }\}.$$

Set-theoretically, $Sub(A) \equiv P(A)$, the isomorphism is established by the two constructions given above. In a general topos $Sub(A)$ will not be an object of that topos, $Sub(A)$ is the *external* representation of the sub-objects of A, while $P(A)$ is the internal representation ($P(A)$ is always an object in the topos).
As an aside, note that if $i \in [j]$ with i and j as above, then $B \equiv C$ as we have by the equations that:

$$i = i \circ k \circ h, j = j \circ h \circ k$$

Both i and j are monic, thus by definition we have that $id_B = k \circ h$ and $id_C = h \circ k$. This justifies our use of $\equiv$ for i and j.

$Sub(A)$ is a partial order, if we define (as we always do) $[i] \subseteq [j]$ iff there is a function $h: B \to C$ such that $i = j \circ h$:

LEMMA 7.1

$Sub(a)$ with $\subseteq$ as defined above is a partially ordered set.

PROOF

Note that $[i] \subseteq [j]$ does not depend on the particular representatives we choose from these equivalence classes, as all their origins are isomorphic as we have seen above.

1) Let $i: B \to A$, then $[i] \subseteq [i]$ by $id_B: B \to B$.

2) Let $[i] \subseteq [j]$ and $[j] \subseteq [i]$ then $[i] = [j]$ by definition.

3) Let $[i] \subseteq [j]$ by f and $[j] \subseteq [k]$ by g. Then $i = j \circ f$ and $j = k \circ g$, hence $i = k \circ g \circ f$. So $[i] \subseteq [k]$ by $g \circ f$. □

To justify the use of $\subseteq$ rather than $\leqslant$, note that if $h: B \to C$ such that $i = j \circ h$, then h is monic. For let $h \circ p = h \circ q$, then $j \circ h \circ p = j \circ h \circ q$ and thus $i \circ p = i \circ q$. So, as i is monic, $p = q$. In other words, if $i \subseteq j$, then $B \subseteq C$.

In the next subsection, we use the partial ordered set properties of $Sub(A)$ to reconciliate the two viewpoints on subobjects.

7.2 The sub-oject classifier

In the previous section, we have seen that $Sub(A)$ is a partially ordered set, moreover, set-theoretically, $Sub(a) \equiv P(A)$. In set theory, we know more: $P(A)$ is a Boolean algebra with the intersection as the meet and the union as the join. It is well-known that Boolean algebra is intimately connected with classical logic. We may define the truth-values as the elements of the Boolean algebra $P(1)$ and then define the semantics of the usual logical connectors as functions on this algebra. In fact, this story may be reversed, given the *logic of* $P(1)$, constructions such as *intersections* may be recovered. This is the path we will take in this chapter. More specifically, this section is devoted to $P(1)$ which may have properties which one would not expect from its set-theoretic counterpart.

$P(1)$ plays such an important role in this chapter, that it is entitled to a new name:

DEFINITION 7.2

The *sub-object classifier* Ω of a topos is defined by:

$$\Omega = P(1)$$

In category theory, ϵ_A is part of the definition of the power type construction. Set-theoretically, it is easy to see that $\epsilon_1 \equiv 1$, as ϵ_1 'registers' which elements of 1 belong to which subset of 1 and 1 has only one element. This observation is true in every topos; before we prove this, we make the following observation:

LEMMA 7.3

Each object A has an arrow $id_A^*: A \to \epsilon_1$

PROOF

$id_A: A \to A$ is a monic arrow and as $A \equiv A \times 1$, we have by the universal property of power types the arrow $id_A^*: A \to \epsilon_1$. □

This lemma is almost the universal property of 1, with the exception that we know for 1 that there is exactly *one* arrow $1 \to A$. Rather than trying to proof that the function $id_A^*: A \to \epsilon_1$ is unique directly, we proof that $\epsilon_1 \equiv 1$ in another way:

THEOREM 7.4

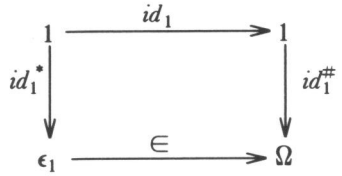
In every topos, $\epsilon_1 \equiv 1$

PROOF

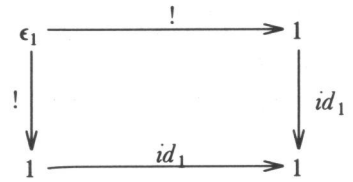
By the definition of power types, we have the monic arrow:

$$\in: \epsilon_1 \to \Omega$$

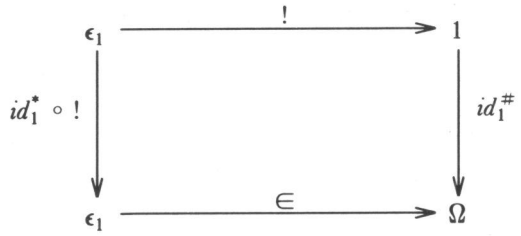
Moreover, for each monic function $f: A \to B$, there are functions $f^*: A \to \epsilon_1$ and $f^{\#}: B \to \Omega$ such that the pair f, f^* is the pullback of the pair $\in$, $f^{\#}$. So, for the monic function $id_1: 1 \to 1$ we get that id_1^*, id_1 is the pullback of $\in$, $id_{\#}$. Graphically, we get that the following diagram is a pullback:



Moreover, the following diagram, which is obviously commutative, is also a pullback diagram, because each object A has a function $A \to \epsilon_1$, by the lemma above and by the universal property of 1, A has exactly one arrow $!: A \to 1$.



So, by combining these two diagrams, we get a pullback diagram:



Now, $!: \epsilon_1 \to 1$ is monic, because $\in: \epsilon_1 \to \Omega$ is monic. Hence, by the

universal property of the power type construction, there is a unique function $\epsilon_1 \to \epsilon_1$ making the above diagram a pullback. Obviously, id_{ϵ_1} satisfies this condition, and hence $id_1^* \circ ! = id_{\epsilon_1}$.

Further, $! \circ id_1^* = id_1$, as each object (including 1) has only *one* function to 1. □

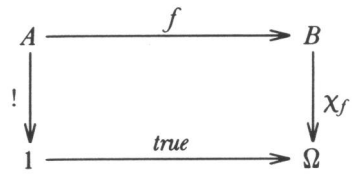
The arrow $1 \to \Omega$, induced by $\in: \epsilon_1 \to P(1) \times 1$ is often called *true*. By the previous theorem we know:

$\Omega-AXIOM$ (7.5)

For each monic $f: A \to B$, there is exactly one arrow

$$\chi_f: D \to \Omega$$

such that the following diagram is pullback:



PROOF

Obvious from the universal property of the power type construction, the theorem above and the definition of Ω. □

Set-theoretically, $P(1)$ is a two element set, say $\{0,1\}$, obviously we may choose $true: 1 \to \{0,1\}$ as $true(1) = 1$. A monic function $f: A \to B$ indicates that A is a subset of B. The fact that the diagram above is a pullback, implies that $\chi_f(b) = 1$ iff $b \in A$. In other words, χ_f is 1 on A, and 0 on $B \setminus A$. This is why χ_f is often called the character of f.

As is often the case in category theory, the universal property of Ω is turned into a definition:

DEFINITION 7.6

In a category C with terminal object 1, an object Ω together with an arrow $true: 1 \to \Omega$ is called a subobject classifier iff it satisfies the Ω-axiom.

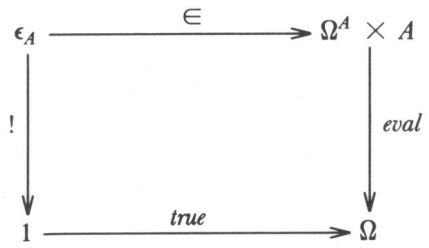
There is a connection between subobject classifiers and the power type construction; this connection is the categorical reconciliation of the viewpoints on subobjects. In the course of this section we have seen that a topos has a subobject classifier. To see the 'reverse' connection, let a *toposs* be a category that has all finite limits, all finite co-limits, exponentiation and a subobject classifier. Then, the following theorem holds:

THEOREM 7.7

A toposs has the power type construction.

PROOF

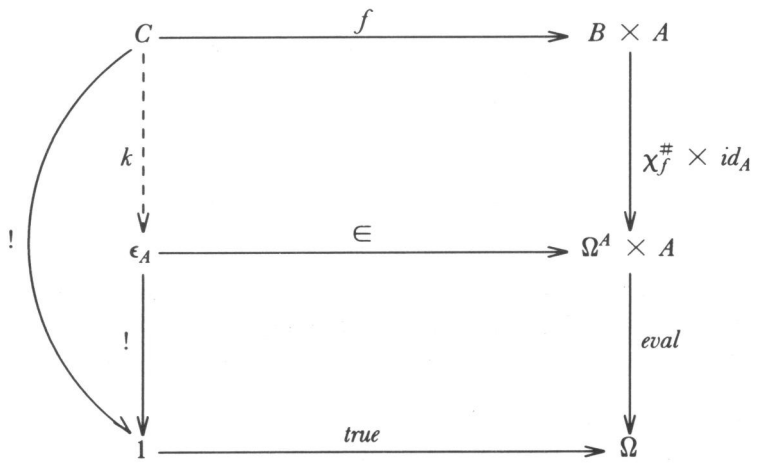
Set-theoretically, $P(A) \equiv 2^A$, where $2 = 1 + 1$ is a two element set. As Ω is set-theoretically also a two element set, we have $P(A) \equiv \Omega^A$. This is what we are going to exploit in the construction of power types in a toposs:
For a given object A, define $P(A) = \Omega^A$. Next, we have to define ϵ_A and the monic arrow $\in: \epsilon_A \to \Omega^A \times A$. Because $true: 1 \to \Omega$ and $eval: \Omega^A \times A \to \Omega$, we may take the pullback of *true* and *eval*. We define ϵ_A and $\in: \epsilon_A \to \Omega^A \times A$ (monic) such that the following diagram is pullback:



Now we have to prove the universal property of power types:
Let $f: C \to B \times A$ be a monic, then by the omega axiom, we have the character $\chi_f: B \times A \to \Omega$ of f. But then, by the universal property of exponentiation, we have a unique function $\chi_f^{\#}: B \to \Omega^A$ such that:

$$eval \circ (\chi_f^{\#} \times id_A) = \chi_f$$

Now, consider the diagram:



The outer 'rectangle' is a pullback by the Ω-axiom, but as the bottom rectangle is also a pullback, there is a unique function $k: C \to \epsilon_A$ such that the whole diagram commutes. As the outer rectangle and the bottom

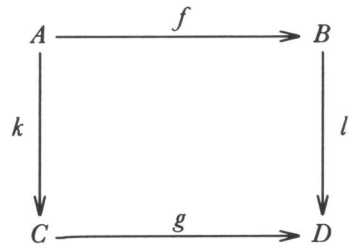
rectangle are pullbacks, the top rectangle is necessarily a pullback.
To see that $\chi_f^{\#}$ is the only functions that can make the top rectangle a pullback, let m be another such function. As the top and the bottom rectangles are pullbacks, the outer rectangle is also a pullback. Hence, by the Ω-axiom we now that $\chi_f = eval \circ (m \times id_A)$. But above we have seen that $\chi_f^{\#}$ is the unique function with this property. □

This theorem gives us immediately the following corollary:

COROLLARY 7.8
Each topos is a toposs and vice versa.

More down to earth, we may paraphrase this corollary as stating that that the two viewpoints on subobjects are interchangeable.

In the next subsection, we will use Ω to develop the logic of a topos. We already know that in set theory, thus in the topos *SET*, Ω is the familiar two element set $\{true, false\}$. At this point it seems illustrative to give an example of a topos in which $P(1)$ is not a two element set! In the next section, we will see what the effect of this remarkable fact on the logic of the topos is. As an aside, note that this is also the first example of a topos other than *SET*. We will not go into the details of all constructions in terms of this new topos, i.e. we will not prove that it is a topos, we only exhibit Ω and try to convince the reader that it is indeed the right definition.
Let $Finset^{\rightarrow}$ be the category in which the objects are arrows in *Finset*, i.e. arrows between finite sets. Let $f: A \rightarrow B$ and $g: C \rightarrow D$ be objects in $Finset^{\rightarrow}$, then the *pair* of *Finset* arrows $(k: A \rightarrow C, l: B \rightarrow D)$ is a $Finset^{\rightarrow}$-arrow iff $g \circ k = l \circ f$; hence, the following diagram should commute:



Before we define the subobject classifier of $Finset^{\rightarrow}$, note that its terminal object, which we denote by $1_{Finset^{\rightarrow}}$, is the *Set*-arrow $id_1: 1 \rightarrow 1$. So, we are looking for an object $\Omega_{Finset^{\rightarrow}}$, i.e. a *Set*-arrow $X \rightarrow Y$ and an arrow $true_{Finset^{\rightarrow}}: 1_{Finset^{\rightarrow}} \rightarrow \Omega_{Finset^{\rightarrow}}$, i.e. a pair of *Set*-arrows $x: 1 \rightarrow X$ and $y: 1 \rightarrow Y$, such that the Ω-axiom is satisfied.

If, in the diagram above, A is a subset of B, C a subset of D and k the restriction of l to B, then an element $x \in B$ can be classified in three ways:

1) $x \in A$ or

2) $x \notin A$ and $l(x) \in C$ or

3) $x \notin A$ and $l(x) \notin C$.

With f and g injective functions, this translates to

1) $\exists y \in A : f(y) = x$

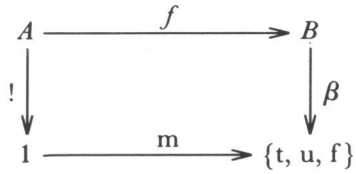
2) $\forall y \in A : f(y) \neq x$ and $\exists z \in C : l(x) = g(z)$

3) $\forall z \in C : l(x) \neq g(z)$

This defines a function β from B to a three element set, say $\{t, u, f\}$ by

$$\beta(x) = \begin{cases} t & \text{in case 1)} \\ u & \text{in case 2)} \\ f & \text{in case 3)} \end{cases}$$

There is only one function $m : 1 \to \{t, u, f\}$ such that the following diagram commutes:



viz. $m(*) = t$. Moreover, with this function the diagram is even a pullback:

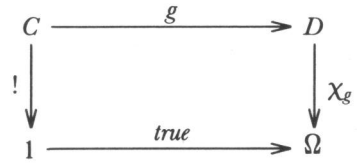
LEMMA 7.9

The diagram above is a pullback diagram.

PROOF

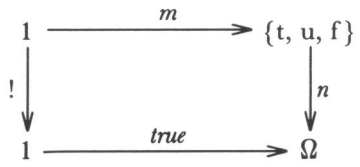
Let E be a set, with two functions $x : E \to B$ and $! : E \to 1$ such that $\beta \circ x = m \circ !$. As $m(*) = t$, this implies that $\beta \circ x(e) = t$ for all $e \in E$. Hence, by the definition of β, $x(e) \in A$. □

Similarly, we have the pullback:

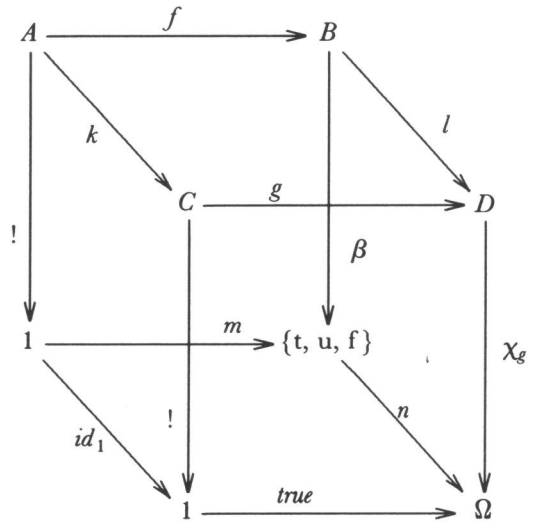


So, if we could find a function $n : \{t, u, f\} \to \Omega$ such that the following diagram

commutes:



we have a likely candidate for the subobject classifier. There are a number of choices. An obvious choice seems to be χ_m; which would even make this diagram a pullback diagram. However, to let $n: \{t, u, f\} \to \Omega$ qualify for $\Omega_{Finset^{\to}}$, the following diagram should be a pullback diagram:



So, as a first step, the right-hand face of this cube should commute. Of course, this decreases the number of choices for n, in fact it determines n completely: Let $\Omega = \{t, f\}$ with $true(*) = t$, and let $x \in B$, then:

1) Let $x \in B$ with $x = f(y)$ for an $y \in A$, then:

$$\begin{aligned} n(t) &= n \circ g(x) = n \circ g \circ f(y) \\ &= \chi_g \circ l \circ f(y) = \chi_g \circ g \circ k(y) \\ &= t \text{ (by definition of } \chi_g \text{).} \end{aligned}$$

2) Let $x \in B$ such that there is an $y \in C$ with $l(x) = g(y)$ and $x \neq f(z)$ for all $z \in A$, then:

$$\begin{aligned} n(u) &= n \circ \beta(x) = \chi_g \circ l(x) \\ &= \chi_g \circ g(y) = t \text{ (by definition of } \chi_g \text{).} \end{aligned}$$

3) Let $x \in B$ such that $l(x) \neq g(y)$ for all $y \in C$, then:

$$n(f) = n \circ \beta(x) = \chi_g \circ l(x)$$
$$= f \text{ (by definition of } \chi_g \text{).}$$

So, $n(t) = n(u) = t$ and $n(f) = f$. Obviously $n \circ m = true \circ !$, moreover:

LEMMA 7.10

In $Finset^{\rightarrow}$, the diagram above is pullback:

PROOF

Let $z: E \rightarrow F$ be an $Finset^{\rightarrow}$ object, with two $Finset^{\rightarrow}$-arrows $(!, !): z: \rightarrow id_1$ and $(x_1, x_2): z \rightarrow l$ such that:

$$(true, m) \circ (!, !) = (\chi_g, \beta) \circ (x_1, x_2)$$

As $true \circ ! = \chi_g \circ x_1$ and the front face is a pullback, there is a unique function $y_1; F \rightarrow C$ such that $g \circ y_1 = x_1$. Similarly, there is a unique function $y_2: E \rightarrow A$ such that $f \circ y_2$.
Next, $x_1 \circ z = l \circ x_2$, hence $g \circ y_1 \circ z = l \circ f \circ y_2$. As the upper face commutes, this implies that $g \circ y_1 \circ z = g \circ k \circ y_2$. But g is monic, thus $y_1 \circ z = k \circ y_2$. So $(y_1, y_2): z \rightarrow l$ is the required $Finset^{\rightarrow}$ arrow. □

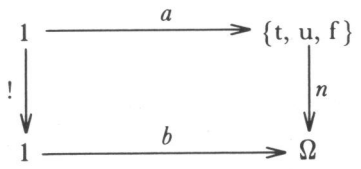
From the construction above it is obvious that $(\beta, \chi_g): l \rightarrow \Omega_{Finset^{\rightarrow}}$ is the only arrow that can make this diagram a pullback diagram. Hence, we may define:

DEFINITION 7.11

In $Finset^{\rightarrow}$, the subobject classifier is given by

1) $\Omega_{Finset^{\rightarrow}} = n: \{t, u, f\} \rightarrow \{t, f\}$, with $n(t) = n(u) = t$ and $n(f) = f$
2) $true_{Finset^{\rightarrow}}: 1_{Finset^{\rightarrow}} \rightarrow \Omega_{Finset^{\rightarrow}} = (true', true'')$ with $true'(*) = t$ and $true''(*) = t$

Now that we know $\Omega_{Finset^{\rightarrow}}$, we can determine its elements. Recall, that such an element is an arrow $1_{Finset^{\rightarrow}} \rightarrow \Omega_{Finset^{\rightarrow}}$. Hence, we want all pairs of *SET*-arrows, such that the following diagram commutes:



Hence, $\Omega_{Finset^{\rightarrow}}$ has three elements, viz.:

1) $true_{Finset^{\rightarrow}} = (true', true'')$ as defined above
2) $undefined_{Finset^{\rightarrow}} = (a, b)$ with $a(*) = u$ and $b(*) = t$

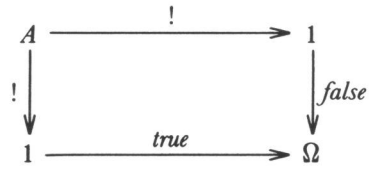
3) $false_{Finset^{\to}} = (c, d)$ with $c(*) = f$ and $d(*) = f$

The reason why we used these names for the elements the subobject classifier will become clear in the next section, in which we define the logic of a topos. Moreover, for convenience, we abbreviate these names to T, U and F respectively.

7.3 The logic of a topos

The subobject classifier has an arrow $true: 1 \to \Omega$. In set theory, we know that Ω has one more element, viz. *false*. Moreover, the logical connectors can be defined as functions into the two element set $\{t, f\}$. In this section, we analyse these constructions in *SET* and use these categorical characterisations as definitions in a topos. We have already seen that in general Ω may have more than two elements, so it should not come as a surprise that the 'logic' thus constructed does not have to be classical. We give an example of a non-classical topos by extending the $Finset^{\to}$ example of the previous section.

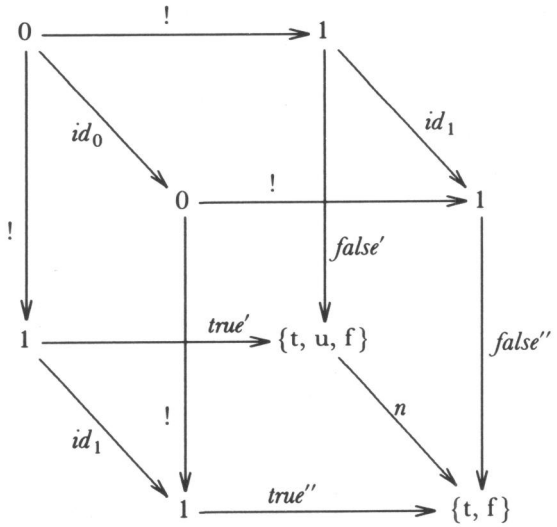
If *true* in *SET* is the element $true: 1 \to \Omega$ with $true(*) = t$ then, obviously, *false* is the element with $false(*) = f$. We can characterise *false* more abstractly by the following observation:
In the pullback diagram:



A is the set of all elements x of 1 for which $false(x) = true(x)$, hence $A = \varnothing$. Moreover, as the diagram above is a pullback diagram, we know by the Ω-axiom that *false* is the character of the unique function $!: \varnothing: \to \{*\}$. In chapter 3 we have seen that the initial object 0 is the categorical generalisation of $\varnothing$. This gives us the following definition of *false*.

DEFINITION 7.12
$false = \chi_!$, where $!: 0 \to 1$.

To illustrate this definition, we construct $false_{Finset^{\to}}$ in $Finset^{\to}$. This means that we have to define the $Finset^{\to}$-arrow $(false', false'')$ such that the following diagram is a pullback diagram:



From the construction of the subobject classifier in the previous section, we know that $false'' = \chi_!$, i.e. $false'' = false_{SET}$; thus $false''(*) = f$. Similarly we know that:

1) $false'(x) = t \Leftrightarrow \exists\, y \in 0\colon !(y) = x$;

2) else $false'(x) = u$ if $\exists\, y \in 0\colon !(y) = x$;

3) $false'(x) = f$ otherwise.

So, $false'(*) = f$ and $false_{Finset^{\rightarrow}} = (false', false'')$; note that this exactly the element of $\Omega_{Finset^{\rightarrow}}$ we called *false* in the previous section.

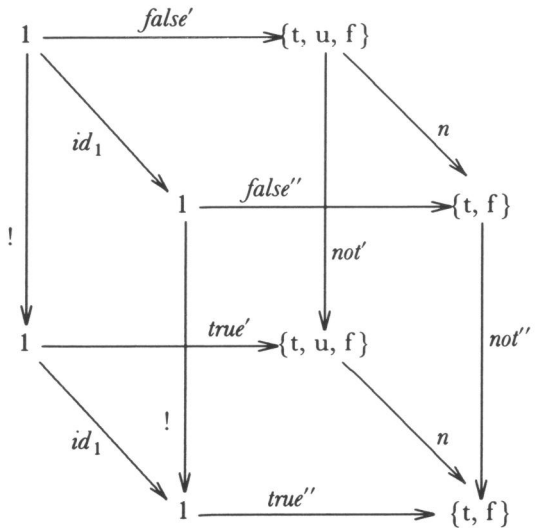
As an aside, the definition of false shows that in each topos Ω has at least two elements if 0 and 1 are not isomorphic. It may have more elements, but these cannot be characterised so easily, as this depends on the actual topos. However, we do not need these elements to define the logical connectors categorically. Before we do this, note that if 1 and 0 *are* isomorphic then all objects in the category are isomorphic. As this is rather uninteresting from the database theorists point of view, we may conclude that for all practical purposes Ω has at least two elements.

Set-theoretically, *not* is a function from $\{t, f\} \rightarrow \{t, f\}$, with $not(t) = f$ and $not(f) = t$. Hence, *not* is the character of an injective function $g\colon A \rightarrow \{t, f\}$ with $g(a) = f$ for all $a \in A$. Coincidentally, *false* as defined above has this property, with $A = 1$. So, in *SET*, *not* is the character of *false*. This generalises to the following definition:

DEFINITION 7.13

$not = \chi_{false}\colon \Omega \rightarrow \Omega$

To construct the not in $Finset^{\rightarrow}$, we have to define the $Finset^{\rightarrow}$-arrow (not', not'') such that the following diagram is pullback:



As for false, we know from the definition of $\Omega_{Finset^{\rightarrow}}$ that $not'' = \chi_{false''}$, hence $not''(t) = f$ and $not''(f) = t$. Similarly, we have that:

1) $not'(false'(x)) = t$ for all $x \in 1$,

2) else $not'(x) = u$ if $n(x) = f$

3) otherwise $not'(x) = f$

Hence, $not'(f) = t$ and $not'(t) = not'(u) = f$. So, by applying this function to the truth elements, we get the following truth-table for $Finset^{\rightarrow}$:

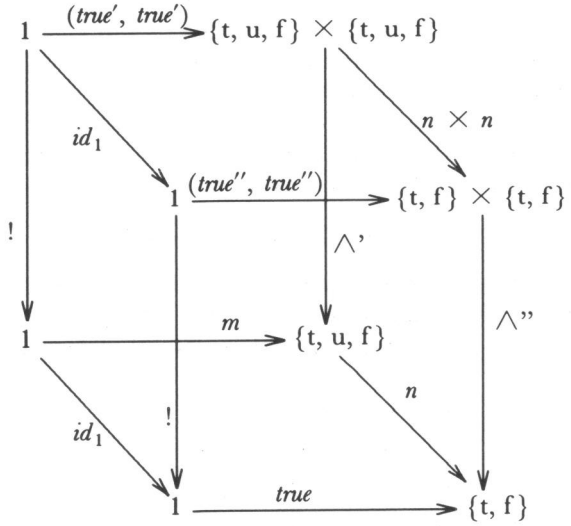
$\neg$	
T	F
U	F
F	T

The following logical connector we consider is $\wedge$. Set-theoretically, this is a function $\{t, f\} \times \{t, f\} \rightarrow \{t, f\}$, with $\wedge(t,t) = t$ and $\wedge(t,f) = \wedge(f,t) = \wedge(f,f) = f$. Hence, $\wedge$ is the character of an injective function $f: A \rightarrow \{t, f\} \times \{t, f\}$ with $f(a) = (t, t)$ for all $a \in A$. The function $true: 1 \rightarrow \{t, f\}$ has $true(*) = t$. So, $(true, true): 1 \rightarrow \{t, f\} \times \{t, f\}$ has $(true, true)(*) = (t, t)$. Therefore, we get the following definition:

DEFINITION 7.14

$\wedge = \chi_{(true,\ true)}: \Omega \times \Omega \rightarrow \Omega$

To construct $\wedge_{Finset^{\rightarrow}}$ we need to know how products look like in this topos. In fact, we already defined this in chapter 3 in which we defined $f \times g$ for two functions f and g. So we need to find an arrow $(\wedge', \wedge'')$ such that the following diagram is pullback



So, as before we know that $\wedge'' = \chi_{(true'', true'')}$, hence $\wedge''(t, t) = t$ and $\wedge''(t, f) = \wedge''(f, t) = \wedge''(f, f) = f$. Moreover:

1) $\wedge'(x) = t \Leftrightarrow x = (true', true')(*)$;

2) else $\wedge'(x) = u$ if $n \times n(x) = (true', true')(*)$;

3) otherwise $\wedge'(x) = f$.

Hence, $\wedge'(t, t) = t$, $\wedge'(t, u) = \wedge'((u, t) = \wedge'(u, u) = u$ and for all other combinations (x, y), $\wedge'(x, y) = f$. This yields $\wedge_{Finset^{\rightarrow}} = (\wedge', \wedge'')$, with the following truth-table:

$\wedge$	T	U	F
T	T	U	F
U	U	U	F
F	F	F	F

Note that in both definitions above, we used objects and arrows that are present in each topos. The same should of course also hold for the other two connectors, viz. $\vee$ and $\Rightarrow$. These constructions are however a bit more complicated.

The next connector we consider is the $\vee$. Set-theoretically, this is a function $\{t, f\} \times \{t, f\} \rightarrow \{t, f\}$, with $\vee(t,t) = \vee(t,f) = \vee(f,t) = t$ and $\vee(f,f) = f$. Hence, $\vee$ is the character of an injective function $f: A \rightarrow \{t, f\} \times \{t, f\}$ with $f(a) \in \{(t, t), (t, f), (f, t)\}$ for all $a \in A$.

Moreover, for each element x of $\{(t, t), (t, f), (f, t)\}$ there should be an $a \in A$ with $f(a) = x$. So, we have to construct a three element set! To construct this set, we actually use the union as defined in the first section of this chapter. This may seem circular, as we define the logic of the topos to define e.g. the union in the next section. However, we use one special case of the union construction of section 1 and this will be enough to define the union in general. Moreover, the construction used is perfectly legal in any topos, so the fact that set-theoretically it establishes a union is merely a coincident.
Note that $\{(t, t), (t, f), (f, t)\}$ is the union of the sets $\{(t, t), (t, f)\}$ and $\{(t, t), (f, t)\}$. Next, we simply construct the union of these two sets as the required set A. Let $true_A: A \to \Omega$ be the function:

$$true_A = true \circ !: A \to 1 \to OMEGA;$$

i.e. $true_A$ is the constant function mapping on the element true of Ω. Then these two sets are easily characterised using $true_{\{t, f\}}: \{t, f\} \to \{t, f\}$ and $id_{\{t, f\}}: \{t, f\} \to \{t, f\}$. As these two induce the injective functions:

1) $(true_{\{t, f\}}, id_{\{t, f\}}): \{t, f\} \to \{t, f\} \times \{t, f\}$
of which $\{(t, t), (t, f)\}$ is the image; and

2) $(id_{\{t, f\}}, true_{\{t, f\}}): \{t, f\} \to \{t, f\} \times \{t, f\}$
of which $\{(t, t), (f, t)\}$ is the image.

From section 1, we now know what to do. Consider, the (induced) function:

$$[(true_{\{t, f\}}, id_{\{t, f\}}), (id_{\{t, f\}}, true_{\{t, f\}})]: \{t, f\} + \{t, f\} \to \{t, f\} \times \{t, f\}$$

It factorises in a surjective function $g: \{t, f\} + \{t, f\} \to A$ and an injective function $f: A \to \{t, f\} \times \{t, f\}$. Hence, $\vee$ is the character of f. This discussion yields the following definition:

DEFINITION 7.15
Let $g: \Omega + \Omega \to A$ and $f: A \to \Omega \times \Omega$ be the epic-monic factorisation of:

$$[(true_\Omega, id_\Omega), (id_\Omega, true_\Omega)]: \Omega + \Omega \to \Omega \times \Omega$$

Then, $\vee = \chi_f$.

To construct $\vee_{Finset^{\to}}$, we ought to factorise the function in the definition above. However, as $true_\Omega(x) = T$, it is straightforward that A is the following set:

$$A = \{(T, T), (T, U), (U, T), (T, F), (F, T)\}$$

Hence, $\vee(x, y) = T$ for $(x, y) \in A$. Moreover, as $n(u) = t$, we have $\vee(U, U) = \vee(U, F) = \vee(F, U) = U$. And finally, $\vee(F, F) = F$. So, we have the following truth-table:

$\vee$	T	U	F
T	T	T	T
U	T	U	U
F	T	U	F

The last connector is $\Rightarrow$. Set-theoretically, this is a function $\{t, f\} \times \{t, f\} \to \{t, f\}$, with $\Rightarrow(t,t) = \Rightarrow(t,t) = \Rightarrow(f,f) = t$ and $\Rightarrow(t,f) = f$. Hence, $\Rightarrow$ is the character of an injective function $f: A \to \{t, f\} \times \{t, f\}$ with $f(a) \in \{(t, t), (f, t), (f, f)\}$ for all $a \in A$. Moreover, for each element x of $\{(t, t), (f, t), (f, f)\}$ there should be an $a \in A$ with $f(a) = x$. Note that these are exactly the elements of $\{t, f\} \times \{t, f\}$ for which the projection on the second coordinate is the same as the $\wedge$. Hence, $f: A \to \{t, f\} \times \{t, f\}$ is the equaliser of π_2 and $\wedge$. This yields the following definition:

DEFINITION 7.16

Let $f: A \to \Omega \times \Omega$ be the equaliser of the functions

$$\wedge, \pi_2: \Omega \times \Omega \to \Omega$$

Then $\Rightarrow = \chi_f$

Inspection the truth table of $\wedge$, shows that the equaliser of $\wedge$ and π_2 is the set:

$$A = \{(T, T), (T, U), (T, F), (U, U), (U, F), (F, F)\}$$

Moreover, $\Rightarrow(U, T) = U$ and $\Rightarrow(x, y) = F$ in all other cases. So:

$\Rightarrow$	T	U	F
T	T	T	T
U	U	T	T
F	F	F	T

From the truth-tables, we can see that the logic in $Finset^{\rightarrow}$ is non-classical. Not only do we have a three valued logic, but also the law of the excluded middle no longer holds:

$$(\neg U) \vee U = F \vee U = U$$

This non-classical behaviour has its influence on the set-constructions we define in the next section. As the types of Flock are supposed to be objects in *SET*, this strange behaviour does not influence its 'Set Theory'.

Before we define the abstract set theory, we illustrate the logic by defining the comparison operators needed in a query language:

The first comparison operator we define is the element-operator. This operator should output true on the input (X, x) iff $x \in X$. If X is a subset of A, this information is exactly encoded in ϵ_A. Hence the element operator we are interested in is exactly the character $\chi_{\in}: P(A) \times A \to \Omega$ of the function

$\in: \epsilon_A \to P(A) \times A$. To adjust to common practice and because the meaning will be clear from the context, we denote it as an infix operator using the symbol: $\in_A$.

DEFINITION 7.17

Let $x: 1 \to A$ and $X: 1 \to P(A)$, then

$$x \in_A X \equiv \chi_{\in} \circ (X, x)$$

Note that as the topos logic may be non-classical, $\in_A$ may behave unexpectedly.

The following comparison operator we consider is (typed) equality denoted by $=_A$, with $=_A: A \times A \to \Omega$. In Set Theory we have $a_1 =_A a_2$ iff a_1 and a_2 are equal. This is of course a not very useful observation. However, we may characterise *equality* differently using the equaliser †:
Given $\pi_{1,A}: A \times A \to A$ and $\pi_{2,A}: A \times A \to A$ we have two functions $A \times A \to A \times A$, viz.:

1) $id_{A \times A} = (\pi_{1,A}, \pi_{2,A}): A \times A \to A \times A$;

2) $twist_{A \times A} = (\pi_{2,A}, \pi_{1,A}): A \times A \to A \times A$;

The equaliser $i: E \to A \times A$ of $id_{A \times A}$ and $twist_{A \times A}$ consists by definition of all those elements of $A \times A$ on which the two functions coincide, i.e. of all pairs (a_1, a_2) for which $a_1 = a_2$. So, as $\in: \in_A \to P(A) \times A$, the equaliser exactly encodes the needed information, i.e. χ_i is the function we were looking for. As above, we will use an infix notation for the equality.

DEFINITION 7.18

Let $id_{A \times A}$ and $twist_{A \times A}$ be as defined above, let $i: E \to A \times A$ be their equaliser and let $a_1: 1 \to A$ and $a_2: 1 \to A$, then:

$$a_1 =_A a_2 \equiv \chi_i \circ (a_1, a_2)$$

Clearly, we need a more constructive characterisation:

LEMMA 7.19

$$=_{A \times B} \equiv \wedge \circ =_A \times =_B$$

PROOF

First of all note that we may rewrite $(A \times B) \times (A \times B)$ to $(A \times A) \times (B \times B)$, then we have:

† Nomen est Omen

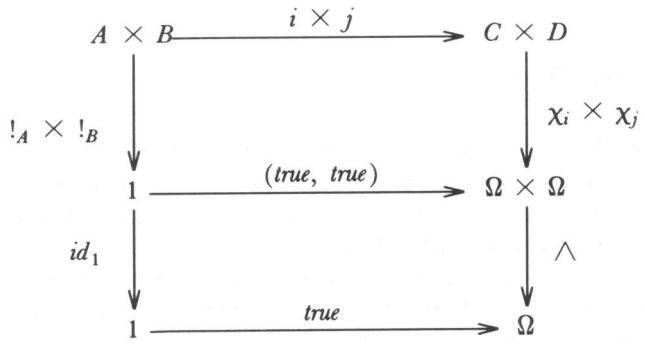
1) $id_{(A \times A) \times (B \times B)} = id_{(A \times A)} \times id_{(B \times B)}$

2) $twist_{(A \times A) \times (B \times B)} = twist_{(A \times A)} \times twist_{(B \times B)}$

So, the equaliser $i_{A \times B}: E_{A \times B} \rightarrow (A \times B) \times (A \times B)$ of $id_{(A \times B) \times (A \times B)}$ and $twist_{(A \times B) \times (A \times B)}$ is the product of the equalisers $i_A: E_A \rightarrow A \times A$ and $i_B: E_B \rightarrow B \times B$. Hence, if we can proof the slightly more general:

$$\chi_{i \times j} = \wedge \circ \chi_i \times \chi_j$$

the proof is complete. So, we have to proof that the outer diagram of the picture below is pullback:



The lower rectangle is pullback by definition, and the upper is pullback as 'products' and 'pullbacks' commute and moreover $true \times true = (true, true)$. □

Note, that in Set Theory, we may rephrase this lemma to the ordinary definition of equality on products: let $(a_1, b_1), (a_2, b_2) \in A \times B$, then:

$$(a_1, b_1) =_{A \times B} (a_2, b_2) \Leftrightarrow a_1 =_A a_2 \wedge b_1 =_B b_2$$

However, such a more constructive characterisation is not available for all constructions. A pragmatics solution to the equality predicate in an implementation would be:

1) $(a_1, b_1) =_{A \times B} (a_2, b_2) \Leftrightarrow a_1 =_A a_2 \wedge b_1 =_B b_2$

2) $a =_{A + B} b$ if there are $x, y \in A$ with $i_A(x) = a$ and $i_A(y) = b$ and $x = y$ or if there are $x, y \in B$ with $i_B(x) = a$ and $i_B(y) = b$ and $x = y$

3) $a =_{A^B} b$ only syntactical. So, we do not use the typed lambda calculus normal form for functions, as the integration of this formalism in topos theory is beyond the scope of this paper.

4) Let $i: E \rightarrow A$ be the equaliser of $f, g: A \rightarrow B$, then $a =_E b$ iff $i(a) = i(b)$.

5) $a =_{P(A)} b$ iff $x \in a \Leftrightarrow x \in b$.

6) For the co-equaliser, i.e. for equivalence types [t], define equality by transforming both elements to the same representative t.

The last comparison operator we consider is $\subseteq$. This operator is easily defined in the set-theoretic operations we define in the next section:

DEFINITION 7.20
Let $A, B: 1 \to P(C)$, then:

$$A \subseteq_C B \equiv A \cap B =_C A$$

7.4 Simple Set Theory

The logic defined in the previous section can be used to define the simple set-theoretic constructions categorically. Similar to the previous section, we do this by analysing the construction in *SET* and generalising (categorising) the result in categorical terms. Moreover, we will illustrate the definitions in *Finset*$^{\to}$. Furthermore, we show that the result of a union and an intersection of two sets using the ad-hoc definition of the first section of this chapter and the abstract definition developed in this section yield the same result.

In Set Theory, the complement of a subset X of a set A is the set $\bar{X}$, defined by $y \in \bar{X} \Leftrightarrow y \notin X$. The character χ_i of the injective function $i: X \to A$ has $\chi_i(a) = t \Leftrightarrow (\exists x \in X: a = i(x))$ and $\chi_i(a) = f \Leftrightarrow (\forall x \in X: a \neq i(x))$. So if $j: \bar{X} \to A$ is the injection of $\bar{X}$ in A, then we have:

$$\chi_i(a) = t \Leftrightarrow \chi_j(a) = f;$$

$$\chi_i(a) = f \Leftrightarrow \chi_j(a) = t.$$

So, $\chi_j = \neg \circ \chi_i$. Spelled out this expression says that $\bar{X}$ is the pullback of $\neg \circ \chi_i$ along $true: 1 \to \{t, f\}$. This generalises to the following definition:

DEFINITION 7.21
Let $i: X \to A$ be a monic, then $\bar{i}: \bar{X} \to A$ is the pullback of $\neg \circ \chi_i$ along $true: 1 \to \Omega$.

To illustrate this definition in *Finset*$^{\to}$, consider the object $X = g: C \to D$ with $C = \{1, 2, 3, 4\}$, $D = \{1, 2, 4\}$ and $g(1) = g(2) = 2$, $g(3) = 4$ and $g(4) = 1$. This is an object with four elements, which we denote by $1 \to 2$, $2 \to 2$, $3 \to 4$ and $4 \to 1$. The object $Y = f: A \to B$ with $A = \{1, 2\}$, $B = \{2, 4\}$ and f the restriction of g is clearly a subobject of X, with the pair $I = (i, j)$, $i: A \to C$ and $j: B \to D$ as the injection; the elements of Y are $1 \to 2$ and $2 \to 2$. Hence, one would expect that the complement of Y has the elements $3 \to 4$ and $4 \to 1$; however, this turns out to be wrong:

To compute the complement of Y, we need the character of I. From the definition of the subobject classifier in *Finset*$^{\rightarrow}$, we know that $I = (\beta, \chi_j)$, with:

1) $\beta(1) = \beta(2) = t$, $\beta(3) = u$ and $\beta(4) = 4$;

2) $\chi_j(2) = \chi_j(4) = t$ and $\chi_j(1) = f$.

Hence, $\chi_I(1 \to 2) = \chi_I(2 \to 2) = T$, $\chi_I(3 \to 4) = U$ and $\chi_I(4 \to 1) = F$. Moreover, $\neg(T) = \neg(U) = F$, so $\neg \circ \chi_i(x) = T \Leftrightarrow x = 4 \to 1$. So, $\overline{Y} = k: E \to F$ with $E = \{4\}$, $F = \{1\}$ and k is the restriction of g.

The union of two sets A and B is the set of all elements in A or in B. So if both A and B are subsets of D, $A \cup B$ consists of all $d \in D: d \in A \vee d \in A$. If $i: A \to D$ and $j: B \to D$ are the injections of A and B into D, then we may translate this to:

$$d \in A \cup B \Leftrightarrow (\chi_i(d) = t \vee \chi_j(d) = t).$$

As $\chi_i(d) = t \vee \chi_j(d) = t$ is the same as $\vee \circ (\chi_i, \chi_j)(d) = t$, we have that $A \cup B$ is the pullback of $\vee \circ (\chi_i, \chi_j)$ along *true*: $1 \to \{t, f\}$. This yields the following definition:

DEFINITION 7.22

Let $i: A \to D$ and $j: B \to D$ be monics, then $i \circ j: A \cup B \to D$ is the pullback of $\vee \circ (\chi_i, \chi_j)$ along *true*: $1 \to \Omega$.

As an illustration of this definition, we continue our example from above. More specifically, we want to compute $Y \cup \{\overline{Y}\}$. As we have already the character χ_I of $I: Y \to X$ and $\chi_{\overline{I}}$ of $\overline{I}: \overline{Y} \to X$, it is straightforward to see that $Y \cup \{\overline{Y}\} = l: G \to H$ with $G = \{1, 2, 4\}$, $H = \{1, 2\}$ and l the restriction of g. Hence, the union of a subobject with its complement does not necessarily yield the entire object. This is a straight forward consequence of the fact that the law of the excluded middle no longer holds as we have seen in the previous section.

The intersection of two sets A and B is the set of all elements both in A and B. So if both A and B are subsets of D, $A \cap B$ consists of all $d \in D: d \in A \wedge d \in A$. If $i: A \to D$ and $j: B \to D$ are the injections of A and B into D, then we may translate this to:

$$d \in A \cap B \Leftrightarrow (\chi_i(d) = t \wedge \chi_j(d) = t)$$

As $\chi_i(d) = t \wedge \chi_j(d) = t$ is the same as $\wedge \circ (\chi_i, \chi_j)(d) = t$, we have that $A \cap B$ is the pullback of $\wedge \circ (\chi_i, \chi_j)$ along *true*: $1 \to \{t, f\}$. This yields the following definition:

DEFINITION 7.23

Let $i: A \to D$ and $j: B \to D$ be monics, then $i \circ j: A \cap B \to D$ is the pullback of $\wedge \circ (\chi_i, \chi_j)$ along *true*: $1 \to \Omega$.

To illustrate this definition, we compute the intersection of Y with $Y \cup \{\overline{Y}\}$. As we have already computed the character of both these subobjects, it is easy to see that this is in fact the subobject Y itself. So, the intersection behaves as one would expect.

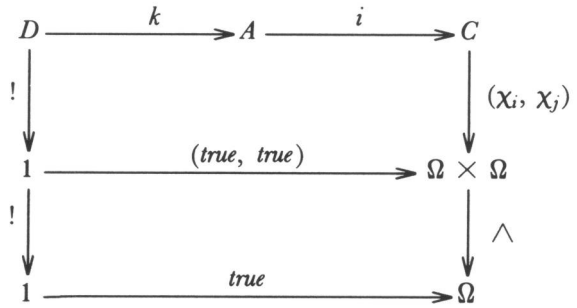
We now have two definitions of intersections and unions, so we have to show that these definitions coincide. Set-theoretically, this is easy, as we know that in both cases the intersection yields the set-theoretic intersection, moreover, the same holds for the union. The definitions are also equivalent in any topos, i.e. the result of the two constructions are isomorphic. However, the proof is rather technical. Instead of giving a formal proof of both equivalences, we give an proof for the intersection; the interested reader is referred to Goldblatt (1979) where an outline of a formal proof for unions can be found.

THEOREM 7.24

The two definitions of intersection yield the same result

PROOF

Let $i: A \to C$ and $j: B \to C$ be the injection of A in C respectively the injection of B in C. Let $k: D \to A \times B$ be the equaliser of $i \circ \pi_A, j \circ \pi_B: A \times B \to C$, i.e. D is the pullback of i and j. We have to proof that the character of $i \circ k = j \circ k$ is $\chi_i \circ \chi_j$. So, we have to proof that the following diagram is a pullback diagram:



The lower rectangle is a pullback diagram by the definition of $\cap$. So, we only have to proof that the upper rectangle is a pullback.
By definition we know that $\chi_i \circ i = true \circ !_A$ and $\chi_j \circ j = true \circ !_B$. Hence, we have:

$$(\chi_i \circ i, \chi_j \circ j) = (true, true) \circ !_A \times B \Leftrightarrow$$

$$(\chi_i, \chi_j) \circ (i, j) = (true, true) \circ !_A \times B \Leftrightarrow$$

$$(\chi_i, \chi_j) \circ (i, j) \circ k = (true, true) \circ !_A \times B \circ k \Leftrightarrow$$

$$(\chi_i, \chi_j) \circ (i, j) \circ k = (true, true) \circ !_D$$

So, the upper rectangle commutes, to see that it is also pullback, let $l: E \to C$ be such that:

$$(\chi_i, \chi_j) \circ l = (true, true) \circ !_E$$

As

$$\chi_i = \pi_{1,\Omega} \circ (\chi_i, \chi_j)$$

we have:

$$\chi_i \circ l = true \circ !_E$$

Hence, there is a unique function $m: E \to A$ such that $i \circ m = l$; similarly, we have a unique function $n: E \to B$ with $j \circ n = l$. Thus, $(m, n): E \to A \times B$, moreover:

$$i \circ \pi_A \circ (m, n) = i \circ m = j \circ n$$
$$= j \circ \pi_B \circ (m, n)$$

Consequently there is a unique function $p: E \to D$ such that $k \circ p = (m, n)$.
So, both the upper and the lower rectangles are pullback and hence the outer rectangle is a pullback. By the Ω-axiom, we have that

$$\cap \circ (\chi_i, \chi_j) = \chi_{i \circ k} \quad \square$$

The definition of intersection and union of two sets A and B depends on a set C of which both A and B are subsets, so the reader might justly wonder if the result depends on the choice of the set C. In other words: if A and B are both subsets of C and of D, is the intersection of A and B in C, denoted by $A \cap_C B$ different from $A \cap_D B$? The answer to this question is negative as long as C and D are comparable, i.e. as long as there exists a set F of which both C and D are a subset:

LEMMA 7.25
Let $i: A \to C$, $j: B \to C$ and $k: C \to D$ be monics, then:

$$A \cap_C B \equiv A \cap_D B$$

PROOF
Let $f_1: A \cap_C B \to A$ and $f_2: A \cap_C B \to B$ be the functions that are the pullback of i and j. Similarly, let $g_1: A \cap_D B \to A$ and $g_2: A \cap_D B \to B$ be the functions that are the pullback of $k \circ i$ and $k \circ j$.
By definition of pullback, we have $k \circ i \circ g_1 = k \circ j \circ g_2$, but k is monic so this implies that $i \circ g_1 = j \circ g_2$. Hence, there is a unique function $l: A \cap_D B \to A \cap_C B$ such that $g_i = f_i \circ l$.
As $i \circ f_1 = j \circ f_2$ by definition, we have that $k \circ i \circ f_1 = k \circ j \circ f_2$. Hence, there is a unique function $m: A \cap_C B \to A \cap_D B$ such that $f_i = g_i \circ m$.
It is straightforward that l and m are each others inverses. $\square$

A similar result holds for the union of two sets.

Above we have used the logical operators $\wedge$, $\vee$ and $\neg$ to define the set operators $\cap$, $\cup$ and *complement*. So, one logical operator, viz. $\Rightarrow$, still remains. As usual, $\Rightarrow$ can also be used to define a set operator. However, although this new operator yields more examples of the possible non-classical behaviour of sets we refrain from introducing it, as we do not need it in this thesis. Again the interested reader is referred to Goldblatt (1979).

7.5 Building sets

Set Theory has two powerful constructions to create new sets, viz. the comprehension principle and enumeration. In this section we define their categorical counterparts. Moreover, we discuss the use of the comprehension principle to define subtypes in Flock.

The naive version of the comprehension principle states:

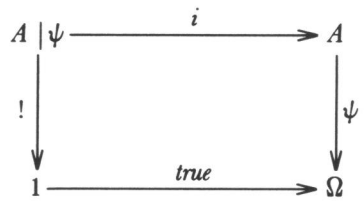
Let ψ be a statement pertaining objects, then there is a set $\{x \mid \psi(x)\}$.

The extensionality axiom then gives us that there is exactly one such set. It is well-known that this naive form leads to paradoxes, such as the Russel Paradox, when used unrestrictedly. A standard solution to this problem is to require that the x's in the definition are elements of an already existing larger set, i.e. to use $\{x \in A \mid \psi(x)\}$. The Ω axiom allows us to translate this principle in categorical terms.

The statement ψ can be seen as a function $\psi: A \to \{t, f\}$, the set we are constructing is the set $B = \{x \in A \mid \psi(x) = t\}$. This is exactly the pullback of ψ along *true*: $1 \to \{t, f\}$; in other words, B is the set whose character is ψ. This observation leads to the following definition:

DEFINITION 7.26

Let $\psi: A \to \Omega$ be a function, then $i: A \mid \psi \to A$ is the pullback of ψ along *true*: $1 \to \Omega$. Graphically:



Set-theoretically, we know that the elements of$A \mid \psi$ are exactly those elements of A for which ψ holds. We have defined elements of A as arrows $1 \to A$, so we have to define what we mean by $x \in A$ and $x \in A \mid \psi$ to see if the observation generalises:

DEFINITION 7.27

Let $f: A \rightarrow B$ be a monic, for $x \in B$ we say $x \in A$ if there exists a $k: 1 \rightarrow A$ such that $x = f \circ k$.

If $x \in A$ for which ψ holds, we have that $\psi \circ x = \mathit{true}$. Hence, as $i: A \mid \psi \rightarrow A$ is a pullback, there is a unique function $k: 1 \rightarrow A \mid \psi$ such that $i \circ k = x$. Conversely, each $y \in A \mid \psi$ defines an element $i \circ y$ of A for which ψ holds. Hence, our definition is a correct generalisation of the set-theoretic comprehension principle.

The comprehension principle is important for query languages. As an illustration, let $\mathit{Car} = \mathit{colour} \times \mathit{make} \times \mathit{registration} - \mathit{number}$ denote the type of cars. Then we can define the subset *RedCar* by:

$$\mathit{RedCar} = \{x \in \mathit{Car} \mid \mathit{colour} \circ x = \mathit{red}\}$$

As a further example, let $P(ABC)$ denote the set of all relations over the schema ABC. The set of all extensions for which the mvd $A \rightarrow\!\!\rightarrow B \mid C$ holds is determined by the proposition $\psi(x) \equiv \pi_{AB}(x) \bowtie \pi_{BC}(x) = x$. Hence, we simply define:

$$\mathit{Mvd} - \mathit{holds} = \{x \in P(ABC) \mid \pi_{AB}(x) \bowtie \pi_{BC}(x) = x\}$$

7.6 The quantifiers

With logic developed so far in this chapter, we can use a topos e.g. as a model for propositional logic. As we have seen, this propositional logic is enough to develop some Set Theory in Flock. In this short section, we investigate some of the properties of the, first order, quantifiers and use the results as a motivation for their categorical generalisation. The main purpose for the introduction of quantifiers is that they are indispensable for query languages; see chapter 7.

Before we introduce the quantifiers, we make the observation that in set theory, a function $f: A \times B \rightarrow C$ corresponds with a function $f^*: B \rightarrow C^A$ with $f^*(b)(a) = f(a, b)$. Clearly, this correspondence is injective, for if $g^*(b) = f^*(b)$ we have $f(a, b) = f^*(b)(a) = g^*(b)(a) = g(a, b)$, i.e. $f = g$. To see that it is also surjective, let $h: B \rightarrow C^A$ and define $h: A \times B \rightarrow C$ by $h(a, b) = h(b)(a)$. So, if we denote the set of all functions $X \rightarrow Y$ by $\mathit{Hom}(X, Y)$, then we have proven that $\mathit{Hom}(A \times B, C) \equiv \mathit{Hom}(A, C^B)$. This observation holds in any topos. Note that we have already seen an instance of this rule as we have seen that there is a bijective correspondence between the functions $f: A \rightarrow B$ and the elements of B^B, i.e. functions $f: 1 \rightarrow B^A$

Quantifiers are used in first order logic for expressions like:

$$\forall a \in A: \phi(a)$$

This should be read as: for all a in A, $\phi(a)$ is true. So, ϕ can be seen as a mapping $\phi: A \rightarrow \{t, f\}$ and the expression then says that for all a $\phi(a) = t$. As we

have seen above ϕ corresponds with an element $\phi^*: 1 \to \{t, f\}^A$. Hence, the arrow denoting $\forall a \in A$, abbreviated to $\forall_A$, should be an arrow $\forall_A: \{t, f\}^A \to \{t, f\}$. Generalising this to toposes, we see that $\forall_A: \Omega^A \to \Omega$. Similarly, we get $\exists_A: \Omega^A \to \Omega$.

To see what function $\Omega^A \to \Omega$ the universal quantifier $\forall_A$ is, consider a function $\phi: A \to \{t, f\}$, or better $\phi^*: 1 \to \{t, f\}^A$. Recall that $\Omega^A \equiv P(A)$, and thus $\{t, f\}^A \equiv P(A)$. Hence, ϕ^* can be seen as a function $\phi^*: 1 \to P(A)$. So, we can identify ϕ^* with a $B \subseteq A$; B is the subset of A of all elements of A on which ϕ is t. Therefore, we have that $\forall_A\phi$ iff $B = A$. In other words, ϕ_* should be the element A of $P(A)$. The correspondence between $P(A)$ and $\{t, f\}^A$ is that $B \subseteq A$ is identified with the function $g: A \to \{t, f\}$ with $g(a) = t$ iff $a \in B$ and vice versa. Hence, the subset A of A corresponds with the function that is always true, which we denoted before by $true_A$. The function $true_A$ corresponds with the element $true_A^*: 1 \to \{t, f\}^A$. Summarising, $\forall_A\phi$ holds iff $\phi^* = true_A^*$. As $true_A^*: 1 \to \{t, f\}^A$ is monic, we may use the Ω axiom: define $\forall_A = \chi_{true_A^*}$, then we have $\forall_A\phi$ iff $\forall_A \circ \phi_* = t$. Generalising this observation, we get the following definition:

DEFINITION 7.28

Let $true_A^*: 1 \to \Omega$ be the arrow corresponding with $true_A: A \to \Omega$, the $\forall_A$ is defined as its character $\chi_{true_A^*}$.

To spell it out Set-theoretically, $\forall_A$ is the function that outputs t on the function $true_A$ and false on all other inputs

The justification for choosing $\forall_A$ as a function $\Omega^A \to \Omega$ may seem a little frail, as we only considered one place predicates, i.e. $\phi: A \to \Omega$. However, a two-place predicate such as $\psi: A \times B \to \Omega$ is a *one-place* predicate over $A \times B$.

Next, we want to define $\exists_A$. By analogy with the analysis of $\forall_A$ above, we have that ϕ_* should be a nonempty element B of $P(A)$, as this implies that $\phi(a)$ is t for some $a \in E$. Recall that ϵ_A consists of all pairs (C,c) such that $C \subseteq A$ and $c \in C$. Hence the only subset of A not represented in ϵ_A is the empty set. In other words, all non-empty subsets of A can be found using ϵ_A. Consider the function:

$$\pi_{P(A)} \circ \in : \epsilon_A \to P(A) \times A \to P(A).$$

A subset B of A is non-empty iff B is in the image of $\pi_{P(A)} \circ \in$. So, if $f: \in_A \to D$, $g: D \to P(A)$ is the epic-monic factorisation of $\pi_{P(A)} \circ \in$, B should be an element of D. As above, this can be translated to $\forall_A(phi)$ holds iff $\chi_g \circ \phi_* = t$. Generalising, we get:

Definition

Let $f: \in_A \to D$, $g: D \to P(A)$ be the epic-monic factorisation of $\pi_{P(A)} \circ \in$, then $\exists_A$ is the character of g.

Spelled out Set-theoretically, this definition says that $\exists_A$ outputs t on all but non-empty inputs.

7.7 Natural numbers

The reader who is well-versed in type theory may wonder on the relation between ordinary type theory and the type structure of Flock. For, in normal type theories, both the booleans, often denoted by Ω, and the natural numbers, often denoted by N, should be present. In this chapter he has seen that Ω could be an object in the type graph of Flock. So a natural question is: is N also present in a topos. The answer is no, a topos does not necessarily has a *natural-numbers object*. As natural numbers occur rather often in practice, we briefly introduce the categorical variant of natural numbers in this section.

As often before, the categorical definition is a generalisation of the set theoretical definition. Considering the Peano axioms, the two most important properties of the natural numbers seem to be that N contains a special element denoted by 0 and that it has a successor function $s: N \to N$. Moreover, let A be another set, with an element $a \in A$ and a function $f: A \to A.$, then we may define a function $g: N \to A$ inductively as follows:

1) $g(0) = x$
2) $g(s(n)) = f(g(n))$

In other words, $g \circ s = f \circ g$. Obviously, h is the only function satisfying this requirement. This observation leads to the following definition:

DEFINITION 7.29

A category C has a natural number object N if there are arrows:

1) $0: 1 \to N$ and
2) $s: N \to N$,

such that for each object A with arrows:

1) $a: 1 \to A$ and
2) $f: A \to A$

there exists a unique function $g: N \to A$ such that:

1) $g \circ 0 = a$ and
2) $g \circ s = f \circ g$.

This definition may seem rather weak†; however, it can be used to construct all primitive recursive functions. Moreover, it can be shown that the Peano postulates hold for such a natural number object. As always, more details may be found in (Goldblatt (1979)).

Those readers not satisfied with this admittedly frail connection between type theory and the type graph of Flock are referred to (Lambek and Scott (1986)), in which the authors show the deep connection between topos theory and higher-order intuitionistic type theories.

7.8 Conclusions

The motivation for this chapter was, as mentioned at the beginning, to show that we can use both Set Theory and Logic in the syntax and semantics of Flock. The reader who has waded through the pages of this chapter should have learned that this is indeed the case. Moreover, she should have learned that the logic of a topos does not have to be classical. However, from a database poin of view, *SET* is the interesting topos, and se have seen that the logic of *SET* is indeed classical.

† at least it first did to the author.

Chapter 8

SOQL, A Query Language

In Chapter 5, we defined structure types and structured objects. These structured objects resemble greatly the complex objets as defined in other formalisms. We already observed in chapter 5 that in Flock structured objects are not complex objects, as they cannot represent incomplete information, nor do they have an identity. However, the structured objects are the foundations for the complex objects. Similarly, the query language for complex objects will be built on a query language for structured objects. So, if we define the query language for structured objects before we define the query language for complex objects, the introduction of the latter language is greatly simplified. Another advantage of a separate query language for structured objects is that this simplifies the comparison with other formalisms. Therefore, the topic of this chapter is the definition of a query language for structured objects. This query language is named SOQL for Structured Object Query Language. As an aside note that SOQL should be pronounced as *socle*; in this way the pronunciation discloses the function of this query language for Flock.

Although the name SOQL may suggest an SQL-like syntax, it more resembles the relational calculus. Moreover, the use of subqueries in SOQL resembles Prolog. These Prolog-like features suggest that queries may be recursive, and indeed a limited form of recursion is supported in SOQL. After some preliminary definitions, we provide a number of examples in the first section to illustrate the kind of queries supported by SOQL. Moreover, we give an informal description of SOQL syntax and semantics. The semantics of SOQL queries reflect the subtyping relationship we defined on structure types. This may cause infinite answers, therefore, a safety condition on *TG* is defined. In the second section we provide the formal syntax of SOQL. The formal semantics of SOQL are given in the third section. Finally, in the fourth section, we discuss a way to incorporate rules in a Flock database.

8.1 Example Queries

In principle, the syntax of SOQL could be defined without the notion of a database. However, to give the semantics of a query, the definition of a database is necessary. The usual definition of a database in relational database theory is that a database is a set of relations, one for each relation schema in the relational schema. Such a relation, often called the extension, is a set of tuples, i.e. a subset of the domain of a relation schema. This can be translated to Flock, by defining an extension of a structure type $s_{[r]}$ semantically as a subobject of $Sem_s(s_{[r]})$: However, $[r]$ is completely determined by r and a subobject of $Sem_s(s_{[r]})$ is completely determined by a subobject of $Sem([r])$. Hence, we may as well identify an extension of $s_{[r]}$ with a subobject of $Sem(r)$:

DEFINITION 8.1

The *s-extension* (structure type extension) of a structure type $s_{[r]}$is denoted by $Ext(s_{[r]})$. Semantically, $Ext(s_{[r]})$ denotes a subobject of $Sem_s(s_{[r]})$.

Note that we simply write *the* extension, denoted by $Ext(s_{[r]})$, as we only consider queries and no updates in this chapter. Hence, the extension of a structure type is *fixed.*

For every day use, the semantics of $Ext(s_{[r]})$ can be simplified. A subobject of $Sem_s(s_{[r]})$ is completely determined by a subobject of $Sem([r])$ and $[r]$ is completely determined by r. Hence, we may as well identify an extension of $s_{[r]}$ with a subobject of $Sem(r)$. This latter can intuitively be seen as either an injective function $x : X \rightarrow Sem(r)$ or as an element of $P(Sem(r))$.

As already mentioned, a database is usually defined as a set of extensions over some schema. Let DBS_{TG} denote the set of structure types defined over some type graph *TG*. Now a database can be defined as a function from the database schema to the extensions:

DEFINITION 8.2

An *s-database* (structured object database) is a function *Ext* that maps each structure type s to its extension $Ext(s)$.

Note, *s-extensions* are objects in the semantics of Flock, while an *s-database* is not. This poses no problem, as the *s-query* language uses the extensions, or the domains themselves, rather than the s-database.

As a final remark before we give examples of SOQL queries, note that for the relational data model, two kinds of query languages are defined, viz. the relational algebra and the relational calculus. Moreover, the equivalence of these two languages has been proven. This approach is followed in many more recent data models. The calculus is then declarative, i.e. one specifies what to retrieve rather than how to retrieve it, while the algebra expresses how the data should be retrieved. SOQL resembles the relational calculus with some Prolog like

features. The effect of this is that SOQL is a declarative language. However, the semantics are given in a category and this 'translation' could be used to define an algebraic query language based on the six constructions. Such an algebraic language definition is, however, outside the scope of this thesis.

In chapter 6, we have defined the set-theoretic comprehension principle in a topos. As usual, this comprehension principle will form the basis for the semantics of SOQL. Hence, we may think of the following expression as the prototypical SOQL query:

$$\{\alpha(x : s_{[r_1]}) \mid \phi(x)\}$$

with $x \in Ext(r_1)$, $\phi: Sem(r_1) \to \Omega$ and $\alpha(x)$ denotes either $f(x)$ for some $f: r_1 \to r_2$ in TG or $f_1 \times f_2(x, y)$ or $eval(y, x)$ if y has an exponential type, i.e. $y: r_2^{r_1}$, or α is built inductively from these forms.

Defining a query language is in fact defining which ϕ's are admissible in this expression. The first step is obviously that we may use the logical connectors in the definition of ϕ, for we have (by definition):

LEMMA 8.3

Let $\phi, \psi: r \to \Omega$, then:

1) $\neg\phi = \neg \circ \phi: r \to \Omega$
2) $\phi \wedge \psi = \wedge \circ (\phi, \psi): r \to \Omega$
3) $\phi \vee \psi = \vee \circ (\phi, \psi): r \to \Omega$
4) $\phi \Rightarrow \psi = \Rightarrow \circ (\phi, \psi): r \to \Omega$ □

So, a formula ϕ can be built recursively from sub-formula and the definition of a query language reduces to the definition of basic formulae.

Obviously, if $\psi: r \to \Omega$ is in TG, then ψ is such a basic formula. More interesting examples arise if we use the comparison operators. The comparison operators supported by SOQL are $=$, $\in$ and $\subseteq$, and as $\neg$ is a logical connector we may also use the negation of these comparisons. Note that we do not add comparison operators like $\leqslant$, as we do not assume that the domains of types have a (partial) order. The reason is that we wish to use the category *Set* for our intuitive semantics. However, it is straightforward to extend SOQL to partial orders, see Chapter 11 on future research.

We start our examples as if we were dealing with extensions of entity types rather than structure types.

To illustrate the use of the comparison operators, let the entity type *car* be defined by $car = colour \times speed \times P(tire)$, with functions $col: car \to colour$, $spd: car \to speed$ and $tires: car \to P(tires)$. An example query is then be given by:

$$\{x: car \mid col(x) = red\}$$

The result of this query would be the subset of the extension consisting of all red cars. Let *flat-tire* be a value of type *tire*, another example is then given by:

$$\{x: car \mid col(x) = red \vee flat-tire \in tires(x)\}$$

The result of this query is the subset of $Ext(car)$ consisting of all cars that are either red or have a flat tire.

Often, one would like to relate extensions of different types in one query, c.f. the use of the join in relational algebra. To illustrate this, let *ship* be another entity type, defined by $ship = colour \times engine$ with functions $col: ship \rightarrow colour$ and $eng: ship \rightarrow engine$. The query that selects the cars for which there is a ship that has the same colour is given by the expression:

$$\{x: car \mid \exists y : ship(\, col(x) = col(y))\}$$

Clearly, the more extensions that are related in a query, the more complicated the relationships gets, and thus the more complicated the expression gets. Therefore, SOQL supports *sub-queries*. If such intermediate results are to be used in other sub-queries, they have to be named. To keep the definition of the *SOQL* syntax relatively simple, we require that all results are named. Hence, first two examples should have been presented as: have to be named.

$$X \supseteq \{x: car \mid col(x) = red\}$$

$$Y \supseteq \{x: car \mid (col(x) = red \vee flat-tire \in tires(x))\}$$

Note that we use $\supseteq$ rather than $=$ as SOQL allows, similarly to Prolog, that the definition of Y is spread over several clauses. We do not use a Prolog-style $:-$, as we believe that $\supseteq$ is closer to the intuitive set-semantics of Flock. To give an example of a query in which the definition of Y is spread over several clauses, note that the example above could also be specified by:

$$Y \supseteq \{x: car \mid (col(x) = red\}$$

$$Y \supseteq \{x: car \mid flat-tire \in tires(x))\}$$

So, if several rules have the same head, the result is obtained by taking the union of the partial results. The third example above could be specified by two rules as:

$$X \supseteq \{x: car \mid \exists y \in Y: col(x) = col(y)\}$$

$$Y \supseteq \{y: ship\}$$

Clearly, there is no real need to use two lines to specify this query, if only because all ships in the extension are considered. But this situation already changes when a pre-selection is performed on the ships. As an illustration, let *bungler_inc* be an instance of *engine*, then the following query is clearly preferable to its equivalent 'one-liner':

$$X \supseteq \{x : car \mid \exists y \in Y : col(x) = col(y)\}$$

$$Y \supseteq \{y : ship \mid eng(y) = bungler\_inc\}$$

Besides using the semantical transformations in the type graph to check if certain conditions are fulfilled, we can also use these transformations to modify the result, e.g.

$$X \supseteq \{spd(x : car) \mid \exists y \in Y : col(x) = col(y)\}$$

$$Y \supseteq \{y : ship \mid eng(y) = bungler\_inc\}$$

The result of this query is that X is bound to the set of speeds of those cars which have the same colour as a ship with an engine from bungler_inc.

Not only semantical transformations can be used to check conditions or modify the result, but also *dynamic* objects may be used. To illustrate this use, suppose that there is a law that prescribes what colour a car should get when it is repainted. Let this law be given by $repaint : car^{car}$, then the following query results in those cars that will be *red* if they are repainted:

$$X \supseteq \{x : car \mid col(eval(repaint, x)) = red\}$$

And the following query results in the cars as if they were repainted:

$$X \supseteq \{ eval(repaint, x : car)\}$$

Note that $eval(repaint, x : car)$ is not in the current state of the database, it is a *hypothetical*-value. Hypothetical in the sense that it is possible that it will become part of the database. Queries using such hypothetical database states are sometimes called *hypothetical queries*, c.f. (Bonner (1988); Bonner (1989)).

The fact that we may use intermediate results in a query suggests that SOQL queries may be recursive, and indeed they may. Our following example is such a recursive query. Let *parent* be the type defined by $parent = human \times human$ with functions $par : parent \rightarrow human$ and $child : parent \rightarrow human$. The interpretation is of course that x is a parent of y if their is a z of type *parent* with $par(z) = x$ and $child(z) = y$. Let $ancestor = human \times human$ with functions $anc : ancestor \rightarrow human$ and $desc : ancestor \rightarrow human$ be defined similarly, then the following query computes the ancestor relation from the parent relation:

$$X \supseteq \{x : parent\}$$

$$X \supseteq \{anc \times desc((x, y)) \mid x \in X \wedge y \in X \wedge desc(x) = anc(y)\}$$

In the examples above, we tacitly assumed that we were querying entity type extensions rather than structure type extensions. Clearly, this will have no effect on the syntax of SOQL, but it does have an impact on the semantics:

1) The result should consist of structured objects rather than entities.

2) The subtyping relation defined on structure types implies that all structured objects of type s with $s \leqslant t$ are candidate solutions.

The first effect is not very interesting for this introduction, but the second is. As an illustration, let *truck* be defined as $truck = car \times load$ with functions *is-car* : $truck \rightarrow car$ and max-*load*: $truck \rightarrow load$. Then the result of the query:

$$X \supseteq \{x : car \mid col(x) = red\}$$

is the set of all cars that are red, united with the set of all trucks that are red when they are considered as *car*, i.e. for trucks the transformation $col(is-car(x))$ is evaluated.

For another illustration of this effect and the last example in this section, let $person = name \times address$ with functions $name : person \rightarrow name$ and $addr : person \rightarrow address$ and let $couple = person \times person$ with (old fashioned, but illustrative) functions $husband : couple \rightarrow person$ and $wife : couple \rightarrow person$. Then the query:

$$X \supseteq \{x : person \mid name(x) = pamela\}$$

results in the set of all persons with name *pamela* united with the set of couples such that either $name(husband(x)) = pamela$ or $name(wife(x)) = pamela$.

The phrase 'united with' used in the last two examples above is rather vague, but suffices for the intuitive semantics. It should be read as stating that the result of the query is the union of two subresults: the subresult consisting of all persons (that are *not* couples) that satisfy the query and the subresult consisting of all couples that, after coercion, satisfy the query.

Note that the use of subtyping as sketched above, may have the effect that the computation of the result takes infinite time. As an illustration, let $s_{[t]}$ be a structure type such that we have the semantical transformation $f : t \rightarrow t$ is in the type graph with $f \neq id_t$. Let the query be given as:

$$X \supseteq \{x : s_{[t]} \mid g(x) = c\}$$

Now suppose that for an $x \in Ext(s_{[t]})$, $g(x) \neq c$ then as $s_{[t]}$ is a subtype of $s_{[t]}$, we have to check whether $g(f(x)) = c$, et cetera.

Clearly, if there are no cycles in the type graphs, such as formed by f and g, such infinite computations can not occur. Therefore, we pose this as *safety condition* on *TG*:

SAFETY CONDITION (8.4)

If t is a root type of a structure type, then the only semantical transformation from t to t in *TG* is id_t.

A type graph that satisfies the safety condition is called *safe*. In the rest of this thesis, we assume that the type graphs are safe!

A typed calculus definition is rather tedious. Therefore, we end this section with an informal sketch; following the general outline of such definitions.
A SOQL query program P is a set of rules of the form:

$$X_i \supseteq \{\alpha_i(x) \mid \phi_i(x)\}$$

The ϕ_i are formulae in a logical language that has a set of constants C (the set of all entity type values) and to sets of variables, viz. *UVar* (for the intermediate results) and *LVar* (the usual variables). The atoms of the logical language are defined inductively by:

1) C, *UVar*, and *LVar* are atoms.
2) $x \in LVar$, then $x : t$ is an atom.
3) If a is an atom and $f \in TG$, then $f(a)$ is an atom.
4) If $a_1, \ldots, a_n$ are atoms, and $f_1, \ldots, f_n$ are in TG, then $f_1 \times \cdots \times f_n(a_1, \ldots, a_n)$ is an atom.
5) If a_1 and a_2 are atoms, then $eval(a_1, a_2)$ is an atom.

The logical language is then built from the atoms and:

the logical symbols $\wedge$, $\vee$, $\Rightarrow$, $\neg$, $\exists$ and $\forall$,

the comparison operators $\in$, $=$, $\subseteq$ and

the set operators $\cap$, $\cup$ and complement.

The $\alpha_i(x)$ are simply atoms and X_i is an UVar variable.

Besides confirming to this syntax, the program P should be well-typed and stratified. The well typedness ensures that functions, evaluations and comparisons are only used when applicable. The stratification, which is similar to the usual logic programming stratification (Apt, Blair, and Walker (1986)), forbids most recursive programs. In particular, recursion is not allowed over $\neg$ and $\forall$. The formal definition of the syntax, the well-typedness relation, and the stratification is given in the next section.

The stratification of SOQL query programs allow for a bottom up computation of the result. The intermediate results from the lower strata are used as as indicated in the examples above. The formal semantics are given in the third section.

8.2 SOQL query programs, the syntax

In the previous section, we have given some example SOQL queries and an informal sketch of SOQL query programs. In this section, we give the formal definition. To enhance readablity, this section is divided in three subsections. In the first subsection, we give the formal syntax for rules. The well-typedness of a set of rules is defined in the second subsection. While stratification and the resulting definition of SOQL query programs is given in the third subsection.

8.2.1 The rule syntax

In this subsection, we define the rule syntax along the lines of the examples and the informal sketch of SOQL programs given in the first section of this chapter. As mentioned before, a rule is of the form:

$$X_i \supseteq \{\alpha_i(x) \mid \phi_i(x)\}.$$

In such a rule, X_i is called the goal, $\alpha_i(x)$ is called the selection and $\phi_i(x)$ is called the formula. To keep the syntactical definition of a rule relatively simple, we first define the *formulae* then the *selections* and finally the *goals*.

The first definition for the SQL formulae is the definition of the constants. In principle, there are two kinds of constants, constants of entity types and constants of structure types. But as each structured object o is determined by its structure type s_r and a value v for the root entity type r of s_r, it is enough to add the constants of the entity types:

DEFINITION 8.5

The set of constants of a given entity type $t = <name, D>$, denoted by $Cons_t$, is defined by:

$$Cons_t = \{c \in D\}$$

For a given type graph TG, we assume that the sets $Cons_t$ are pairwise disjoint. The set of constants of SOQL, relative to some type graph TG, denoted by $Cons_{TG}$, is then defined as the union of the $Cons_t$, i.e.:

$$Cons_{TG} = \bigcup_{t \in TG} Cons_t$$

In the rest of this subsection, we will simply write $Cons$ instead of $Cons_{TG}$, as the general syntax of SOQL does not depend on the actual constants present.

The second 'building block' for SOQL formulae is formed by the variables. In the examples of the previous section, we have used two kinds of variables, viz. upper case and lower case symbols. The reason for this distinction is that it highlights which variables denote intermediate results, and which variables vary over some domain. Consequently, we have two sets of variables:

DEFINITION 8.6

SOQL has two sets of variables, denoted by $LVar$ and $Uvar$, defined by:

1) $LVar$ is a set of strings starting with a lower case symbol.
2) $UVar$ is a set of strings starting with a upper case symbol.

Note that we do not define different sets of variables for each type, as this would hide the type of the variables from the SOQL query. This hiding would make type checking for humans more complicated.

Now that we have variables and constants, we can define expressions. This definition is more or less the usual inductive one. To limit the cases we have to make in this definition and following definitions, for each t in TG, id_t is treated as if it were in TG:

DEFINITION 8.7

The set of expressions Exp of SOQL is defined inductively by:

1) Constants are expressions, i.e. $Cons \subseteq Exp$.
2) Variables are expressions, i.e. $LVar \subseteq Exp$ and $UVar \subseteq Exp$.
3) $LVar$-variables may be assigned a type, i.e. $\{x : t \mid x \in LVar \wedge [t] \in DBS_{TG}\} \subseteq Exp$.
4) If f is in TG and $e \in Ext$, then $f(e) \in Exp$.
5) If f and g in TG and $e_1, e_2 \in Ext$, then $f \times g(e_1, e_2) \in Ext$.
6) If $e_1, \ldots, e_n \in Exp$, then
   a) $(eval(e_1, e_2)) \in Exp$
   b) $(e_1 \cap e_2) \in Exp$
   c) $(e_1 \cup e_2) \in Exp$
   d) $(\overline{e_1}) \in Exp$
   e) $\{e_1, \ldots, e_n\} \in Exp$

Note that many of the expressions in Exp are nonsensical, e.g. 4) allows any semantical transformation on any value regardless if the particular transformation is applicable to that value. Clearly, the set of expressions should be reduced to those that make sense. This reduction can of course be made using type information. However, we want to keep the burden of typing the variables as low as possible. In particular, this means that we do not want that each occurrence of a variable should be accompanied by typing information. Hence, in general a single expression will not contain enough information to deduce whether it is well typed. Therefore, we have to postpone the type checking till we have defined SOQL programs; only then can we define well typed SOQL programs; this is the subject of the next subsection.

Now that we have defined expressions, we can built *formulae* by comparing expressions. The set of comparison operators consists of $=$, $\in$, and $\subseteq$. So, as already mentioned before, this set does not contain order-comparisons. The reason for this ommission is that in general sets do not have to have a comparison operator. Note that if we would have chosen the category of partial ordered sets as the prefered semantics of Flock, order comparisons would have been possible.

However, it would have made the semantics of Flock more complicated.

DEFINITION 8.8

The set *BForm* of basic SOQL formulae is defined by:
Let $e_1, e_2 \in Exp$, then:

1) $e_1 \in BForm$
2) $(e_1 = e_2) \in BForm$;
3) $(e_1 \in e_2) \in BForm$;
4) $(e_1 \subseteq e_2) \in BForm$.

Again, these basic formulae need to be type checked, the most notable example is 1). Expressions can of course only be considered as (basic) formulae if their type is Ω. However, as above the type checking will be done in the next subsection.

The basic formulae are composed to formulae using the logical symbols of *SOQL*. In the definition of these formulae, we use *freevar*(ϕ) to denote the free variables of a formula ϕ. We assume that the reader knows the notion of a free variable in a formula:

DEFINITION 8.9

The set *Form* of SOQL formulae is defined inductively by:

1) $BForm \subseteq Form$
2) Let $\phi \in FORM$, $Y \in UVar, y \in LVar$ and $y \in$ *freevar*(ϕ), then:
   a) $(\forall y \in Y: \phi(y)) \in Form$,
   b) $(\exists y \in Y: \phi(y)) \in Form$.
3) Let $\phi_1, \phi_2 \in Form$, then:

a) $\neg\phi_1 \in Form$,
b) $\phi_1 \wedge \phi_2 \in Form$,
c) $\phi_1 \vee \phi_2 \in Form$,
d) $\phi_1 \Rightarrow \phi_2 \in Form$.

The elements of *Form* are the formulae that may occur in a rule.

The next step toward the definition of a rule is the definition of a *selection*. In the previous section, the examples had selections of the form: $x: t$, $f(x: t)$ and *eval*$(c, x: t)$ with c a constant and f in *TG*. All these selections are expressions.

This is always true, i.e. the set of selections is a subset of the set of expressions. The difference between the two sets is that selections may not use UVar variables nor set constructions, i.e.:

DEFINITION 8.10
The set *Sel* of SOQL selections is defined inductively by:

1) $LVar \subseteq Sel$.
2) *LVar*-variables may be assigned a type, i.e. $\{x : t \mid x \in LVar \wedge [t] \in DBS_{TG}\} \subseteq Sel$.
3) If f is in TG and $e \in Sel$, then $f(e) \in Sel$.
4) If f and g in TG and $e_1, e_2 \in Sel$, then $f \times g(e_1, e_2) \in Sel$.
5) If $e_1, e_2 \in Sel$ and $c \in Cons$, then $(eval(e_1, e_2)) \in Sel$ and $(eval(c, e_1)) \in Sel$

The final step before we can define rules is the definition of *goals*, but it should be clear that goals are *UVar*-variables. Hence, we get the following definition of a SOQL rule:

DEFINITION 8.11
A SOQL rule is an expression of the form:

$$X \supseteq \{s \mid \phi\}$$

in which $X \in UVar$, $s \in Sel$ and $\phi \in Form$.

This definition of SOQL rules induces the following definition of SOQL programs:

DEFINITION 8.12
A SOQL program P is a set of SOQL rules.

Now that we have the formal definition of SOQL programs, we can define a well-typedness relation to check whether or not a program makes sense. This is the subject of the next subsection.

8.2.2 Well-typed SOQL programs

Let TG contain the entity types $car = colour \times speed$ with functions $col : car \to colour$ and $spd : car \to speed$ and $person = name \times age$ with functions $nm : person \to name$ and $ag : person \to age$. Moreover, suppose that *person* is also the root type of a structure type. Then the syntax of the previous subsection allows the following query;

$$X \supseteq \{x : person \mid spd(x) = 37\}$$

Clearly, this query is nonsensical, as *spd* is not applicable to the type *person*. To filter this kind of nonsensical queries from our language, we have to use the type information from *TG*. In this subsection, we show how to do this.

There are two type systems we have to deal with, entity types and structure types. Both structure types and entity types are needed in the query language. The structure types are needed as the database is associated with these types and the entity types are needed as otherwise many comparisons such as $spd(x) = 37$ could not be used.

To make the distinction between the two type systems visible, we will write $\alpha : t$ to denote that α is of structure type t and $\alpha :: t$ to denote that α is of entity type t. Moreover, we will often use s for structure types and t for entity types.

Recall, that structure types are in a subtyping relation. This means that:

if $s_1 \leqslant s_2$ and $\alpha : s_1$, then $\alpha : s_2$.

Moreover, structure types can be cast into entity types as the *roottype* $[r]$ of the structure type $s_{[r]}$ completely determines the structure type, and $[r]$ is determined by any of its representatives. Therefore, the rule for this type casting is:

if $t_1 \in [t_2]$ and $\alpha : s_{[t_2]}$, then $\alpha :: t_1$

The definition of the well-typing relation on SOQL programs P follows the definition of their syntax closely. In fact, we could have defined the well-typed programs directly. However, the well-typing of a program cannot be decided per rule as it depends on the complete set of rules. Moreover, it is also not possible to decide whether an expression such as $spd(x) = 37$ is well typed as this depends on other parts of the rule. Therefore, if we would have defined well-typed programs directly, the typing would have rather opaque.

If we follow the definition of SOQL programs closely, the first typing has to occur for the constants. This is easy, as they are already typed:

DEFINITION 8.13

Let $c \in Cons_{TG}$, then $c :: t$ iff $c \in Cons_t$.

The following step is then the typing of the variables. This is already a bit harder as sometimes, the type has to be inferred from other type information. Variables from *LVar* can be typed in two ways, viz. $x : t$ and $x \in X$; in both cases, their type is obvious. Variables from *UVar* can be typed in only one way, as their type is determined by the rules defining these 'result'-variables. So, the definition becomes:

DEFINITION 8.14
Let $x \in LVar$ and $X \in UVar$, then:

1) If the expression $x : s$ occurs in a rule, this implies that $x : s$ for every occurrence of x in the rule.
2) If $x \in X$ occurs in a rule and $X : P(s_t)$, then $x : s_t$ for every occurrence of x in the rule.
3) If $x \subseteq X$ occurs in a rule, and $X : s$, then $x : s$ for every occurrence of x in the rule
4) Let $X \subseteq \{\alpha | \phi\}$ be a rule in P, let $\alpha : s_1$, and let $s_1, \ldots, s_n$ be the complete set of subtypes of s_1, then $X : P(s_1 + \cdots + s_n)$ for every occurrence of X in the program.

Note that if a variable can be assigned more then one type, these types should coincide, otherwise the rule is not well-typed.

This definition will need some explanation. In particular, the rules 1), 2) and 3) may seem more restrictive than needed. For example, the following query makes perfectly good sense although the variable x has two different typings in the same rule:

$$X \supseteq \{x : s_1 \mid \exists y : s_2(\forall x : s_3(f(x) = g(y)) \wedge h(x) = k(y))\}$$

The trick in this rule is of course that it we may substitute any variable from $LVar$ for x in $\forall x : s_3(f(x) = g(y))$ without changing the result of this query. It should be clear that supporting this kind of rules would make the definition above rather complicated.

We have two remarks to make on rule 3). The first one is that the type uniqueness of the variable X throughout the program is an example of the fact that X is a *global* variable in P, while x is local to a rule. The second remark is that we choose the type $P(s_1 + \cdots + s_n)$ for X, as we also want subtypes to contribute to the result of a query. As an aside, note that as all the s_i are subtypes of s_1, $P(s_1 + \cdots + s_n)$ is a subtype of $P(s_1)$; the latter being the type the reader may had expected.

The final remark we have to make on this definition is that these rules are the only rules that can be used to infer the type of a variable. All the definitions that follow can only be used to check whether the typing in the program is done consistently. In particular, this means that each rule in a program has to contain a 'clause' of type 1), 2) or 3) for each of the LVar variables that occur in that rule. Otherwise, at least one of the LVar variables cannot be assigned a type and thus the program can not be well-typed.

Now that variables and constants are typed, we can type the expressions; given that we have to deal with two typing systems, the definition is rather large:

DEFINITION 8.15

The set *Texp* of typed SOQL expressions is defined as follows:

1) Constants and variables are typed expressions, with their type as defined above.

2) If $x : s \in Exp$, then $x : s \in TExp$. Moreover, the type of this expression is s, i.e. by abuse of notation $(x : s) : s$.

3) Let $f : t_1 \rightarrow t_2$ in TG be and let t_1 and t_2 be root types of respectively structure types $s_{[t_1]}$ and $s_{[t_2]}$ respectively. If $e : s_{[t_1]} \in TExp$, then $f(e) : s_{[t_2]} \in TExp$

4) Let $f : t_1 \rightarrow t_2$ in TG. If $e :: t_1 \in TExp$, then $f(e) :: t_2 \in TExp$.

5) Let $f_1 : t_1 \rightarrow t_2, \ldots, f_n : t_{2n-1} \rightarrow t_{2n}$ in TG and let $t_1, t_3, \ldots, t_{2n-1}$, and $t_2 \times t_4 \times \cdots \times t_{2n}$, be the root entity types of the structure types $s_1, s_3, \ldots, s_{2n-1}$ and s_{even} respectively.
If $e_1 : s_1, e_3 : s_3, \ldots, e_{2n-1} : s_{2n-1} \in TExp$, then
$f_1 \times \cdots \times f_n(e_1, .., e_{2n-1}) : s_{even} \in TExp$.

6) Let $f_1 : t_1 \rightarrow t_2, \ldots, f_n : t_{2n-1} \rightarrow t_{2n}$ in TG.
If $e_1 :: t_1, e_3 :: t_3, \ldots, e_{2n-1} :: t_{2n-1} \in TExp$, then
$f_1 \times \cdots \times f_n(e_1, .., e_{2n-1}) :: t_2 \times t_4 \times \cdots \times t_{2n}$.

7) Let $e_1 :: t_1^{t_2} \in TExp$, $e_2 : s_{[t_1]} \in TExp$, and $e_3 : s_{[t_2]} \in TExp$ then: $eval(e_1, e_3) : s_2 \in TExp$

8) Let $e_1 : s_{P(t)}$, $e_2 : s_{P(t)} \in TExp$ and

   a) $(e_1 \cap e_2) : s_{P(t)} \in TExp$

   b) $(e_1 \cup e_2) : s_{P(t)} \in TExp$

   c) $(\overline{e_1}) : s_{P(t)} \in TExp$

9) Let $e_1 : s_t, \ldots, e_n : s_t \in TExp$, then $\{e_1, \ldots, e_n\} : s_{P(t)} \in TExp$.

10) Let $e_1 :: t_1^{t_2} \in TExp$, $e_2 :: t_1 \in TExp$, and $e_3 :: t_2 \in TExp$ then: $eval(e_1, e_3) :: t_2 \in TExp$

11) Let $e_1 :: P(t)$, $e_2 :: P(t) \in TExp$ and

   a) $(e_1 \cap e_2) :: P(t) \in TExp$

   b) $(e_1 \cup e_2) :: P(t) \in TExp$

   c) $(\overline{e_1}) :: P(t) \in TExp$

12) Let $e_1 :: t, \ldots, e_n :: t \in TExp$, then $\{e_1, \ldots, e_n\} :: P(t) \in TExp$.

Most rules are obvious, but rule 3) and 5) probably needs some explanation.

The idea behind these rules is that structure types are much more interesting than entity types. Thus, we should try to type expressions as structure types whenever this is reasonable.

As *selections* are a special kind of expressions, the typing of selections is the same as that for expressions. Hence, these three typing definitions together allow us to type each constant, variable and expression; note that all these types should be in TG or DBS_{TG} respectively. If this typing is consistent throughout the program P, i.e. *LVar* variables have a unique type in a rule and *UVar* variables have a unique type in P, then the program is called *typeable*. Of course, only typeable programs can be well typed. A notion we can finally define. The well typedness of a program depends on the well-typedness of its basic formulae:

DEFINITION 8.16

The set *WTBForm* of well-typed basic SOQL formulae is defined by:
Let $e_1 : s_t$, $e_2 : s_t$, $e_3 :: t$, $e_4 :: t$, $e_5 : s_{P(t)}$, $e_6 : s_{P(t)}$, $e_7 :: P(t)$ and $e_8 :: P(t)$ all be elements of *TExp*.

1) $e_1 \in WTBForm$, iff $e_1 :: \Omega$
2) $(e_1 = e_2) \in WTBForm$;
3) $(e_1 \in e_5) \in WTBForm$;
4) $(e_5 \subseteq e_6) \in WTBForm$;
5) $(e_3 = e_4) \in WTBForm$;
6) $(e_3 \in e_7) \in WTBForm$;
7) $(e_7 \subseteq e_8) \in WTBForm$;
8) $(e_1 = e_3) \in WTBForm$;
9) $(e_1 \in e_7) \in WTBForm$;
10) $(e_1 \subseteq e_7) \in WTBForm$;
11) $(e_7 \in e_1) \in WTBForm$;
12) $(e_7 \subseteq e_1) \in WTBForm$.

Note that the last five rules are the cases in which a structure type is cast into an entity type.

Now we can define well typed programs simply as follows:

DEFINITION 8.17

Let P be a typeable SOQL program, it is well-typed iff all its basic formula are well-typed.

Demanding that a program is well-typed is still not enough to ensure that it also makes sense. Due to the possibility of recursion in the program, it is possible that the query admits no answer. In the next subsection, we filter out this type of programs by requiring that a program is stratified.

8.2.3 Stratification

In the area of logic programming, it is well-known that there are many programs that do not have a well-defined semantics, even though they are syntactically correct. The most important examples arise with the use of negation, e.g.

$$a(X, Y) :- f(X, Y).$$

$$a(X, Y) :- a(X, Z), a(Z, Y), \neg a(Z, Z).$$

$$f(1, 2).$$

$$f(2, 1).$$

Has two possible solution sets, viz. $\{a(1, 2), a(2, 1), a(1, 1)\}$ and $\{a(1, 2), a(2, 1), a(2, 2)\}$. This example is easily rewritten to a SOQL query:

$$X \supseteq \{x : s\}$$

$$X \supseteq \{(\pi_1,\pi_2)((x_1 : s, x_2 : s)) \mid x_1 \in X \wedge x_2 \in X \wedge \pi_2(x_1) = \pi_1(x_2) \wedge \neg((\pi_2,\pi_1)((x_1, x_2)) \in X)$$

So, we can still define well typed SOQL programs that we cannot give semantics.

In logic programming languages, such programs are filtered out by requiring that programs are *stratified*, see e.g. (Apt, Blair, and Walker (1986)). Stratification is easiest explained using the notions of *partial* and *total* dependence (Abiteboul and Grumbach (1987)). A predicate p is said to depend on the predicate q if q is used in the definition of p. Such a dependency is called total if the extension of q has to be known completely to compute the extension of p and partial otherwise. The dependencies induce a dependency graph, a program is called stratified if no edge in a cycle depicts a total dependency. In the example program above we need to know if $\neg a(Z, Z)$ holds to be able to conclude $a(X, Y)$. But $\neg a(Z, Z)$ can only be known if the complete extension of a is known.

In this subsection, we define stratified SOQL programs along the same lines. We define a dependency relation on the UVAR variables in a program, and if no UVAR variable is totally dependent on itself, the program is called stratified. The first step is the definition of dependency:

DEFINITION 8.18

Let X and Y be two UVAR-variables in an *SOQL* program P, then X is called dependent on Y if there is a rule defining X in which Y occurs.

The following step is the definition of *total* and partial dependencies. To define these notions, we need to inspect the various ways in which an UVAR variable can occur in a rule. It is much easier to define when a dependency is partial than to define when it is total. Therefore, we first define partial and then total as the complement of partial. However, we first give some examples of total dependencies and the motivation for calling them total; beware that this list is not exhaustive.
The first example is:

$$X \supseteq \{z : s \mid \forall y \in Y\ (f(z) \in g(y))\}$$

In this example, X is totally dependent on Y, because we have to check all elements of Y before we can decide whether a z belongs to X. In other words, this rule cannot be used to compute new elements of X if Y is not yet totally computed.
The second example is given by the following rules:

$$X \supseteq \{eval(c_1, z) \mid z \in Z\}$$

In such a rule, we also say that X is totally dependent on Z. The difference with the example above is that we could start computing new elements of X if Z is only partially known. However, suppose that the following rule is also in the program:

$$Z \supseteq \{eval(c_2, x) \mid x \in X\}$$

Combining these two rules, we see that $\{eval(c_1, eval(c_2, x)) \mid x \in X\} \subseteq X$. So, the computation of the result may very well never finish! Therefore, X is said to be totally dependent on Z. For similar reasons, X is totally dependent on both Y and Z in the following rule:

$$X \supseteq \{eval(c_1, z) \mid z \in Z \wedge f(z) \in Y\}$$

Problems as above, do not occur if we have a rule of the form:

$$X \supseteq \{f(z) \mid z \in Z\}$$

for we know that if their is another rule:

$$Z \supseteq \{g(x) \mid x \in X\}$$

then $f \circ g = id$ as we do have that $h : t \to t$ for a root type t implies that $h = id_t$ in a safe type graph. So, in such rules, the dependency of X on Z is only partial.

However, in a rule of the form:

$$X \supseteq \{f \times g(z_1, z_2) \mid z_1 \in Z \wedge z_2 \in Z\}$$

the dependency from X on Z is again total, as we might have another rule of the form:

$$Z \supseteq \{k \times l(x_1, x_2) \mid x_1 \in X \wedge x_2 \in X\}$$

Similar to the situation above, we cannot guarantee that the computation halts. Different from the situation above, we can make exceptions in special cases. For example, if f and g are projections or identity functions, as in the rule:

$$X \supseteq \{\pi_i \times id(z_1, z_2) \mid z_1 \in Z \wedge z_2 \in Z\},$$

the dependency of X on Z is partial. This simple exception already allows us to compute simple recursive queries such as the ancestor problem.

From these examples, we see that the selections that may yield a partial dependence are a subset of all selections. We can define this set as follows:

DEFINITION 8.19

The set *PSel* of those well-typed selections that may result in a partial dependence is defined inductively by:

1) $LVar \subseteq PSel$.
2) $LVar$-variables may be assigned a type explicitly, i.e. $\{x : s \mid x \in LVar \wedge [t] \in DBS_{TG}\} \subseteq PSel$.
3) If f is in TG, $f : t_1 \rightarrow t_2$ with t_1 and t_2 root types of entity types s_1 and s_2 and $e : s_1 \in PSel$, then $f(e) \in PSel$.
4) If f and g are either projections or identity functions and $e_1, e_2 \in PSel$, then $f \times g(e_1, e_2) \in PSel$, provide that $f \times g(e_1, e_2)$ is well-typed.

Using *PSel*, we get the following definition of partial dependence:

DEFINITION 8.20

Let Y be dependent on X in the well typed rule:

$$X \supseteq \{\alpha \mid \phi\}$$

Then Y is partially dependent on X in this rule iff:

1) $\alpha \in PSel$ and
2) Y does not occur in a negated formula in ϕ, nor is there universal quantification over T in ϕ.

In all other cases, X is totally dependent on Y in this rule.

We can now extend these dependencies to programs:

DEFINITION 8.21

Let P be a well-typed SOQL program and let X and Y be two UVAR variables in P. Then X is partially dependent on Y in P, denoted by $Y \leqslant X$, if X is partially dependent on Y in all rules in P in which X is dependent on Y. In all other cases, X is totally dependent on Y, denoted by $Y < X$.

This definition induces a graph G_P as follows:
The nodes of G_P are the UVar variables in P, whenever $X \leqslant Y$, there is an edge from X to Y and whenever $X < Y$, there is a marked edge from X to Y.
Now we can define stratified programs similar to (Abiteboul and Grumbach (1987)):

DEFINITION 8.22
A well-typed SOQL program P is called stratified if there is no cycle in G_P in which at least one of the edges is marked.

Now we are almost ready to define SOQL queries, we need only one more restriction. To formulate this last restriction, we need a well-known consequence from stratification:

THEOREM 8.23
Let Q be a SOQL query and V_Q its set of *UVAR* variables, then there is a partion $V_Q = V_1 \cup \cdots \cup V_n$ such that for X, $Y \in V_Q$:

$$(X \leqslant Y \wedge X \in V_i) \Rightarrow \exists j(i \leqslant j \wedge Y \in V_j)$$

$$(X < Y \wedge X \in V_i) \Rightarrow \exists j(i < j \wedge Y \in V_j)$$

PROOF
Suppose that such a partition is not possible, then there have to be a pair X and Y such that $X < Y$ and $Y < X$. But this implies that Q is not stratified. □

Of course, each program can have a number of different stratifications. It is straightforward to put a partial order on the different stratifications by:

DEFINITION 8.24
$s_1 \leqslant s_2$ if for any two UVar variables X and Y in P we have that X and Y in the same stratus under s_1 if they are in the same stratus under s_2.

To give semantics to a query, it is necessary that it is unambiguous which of the UVar variables in the program is deemed as *the* result of the query. This means that in each of the finest stratifications, P should have the same UVar variable (the result variable) in the highest stratus.
This leads to the following definition:

DEFINITION 8.25
A SOQL query Q is a stratified well-typed SOQL program, with a well-defined result variable.

We will often use the type of the result variable as the type of the query.

As a final remark in this subject, note that a major difference with a recursive query language such as LDL (Naqvi and Tsur (1989)) and SOQL is that LDL supports grouping, while SOQL does not seem to do so. However, this is only appearance, as we may have both $P(ABC)$ and $P(AP(B)C)$ in the type graph with the grouping function from the first to the latter. A disadvantage of our safety requirement is however that only one of the two can be the root entity type of a structure type. Clearly, this is a blow to our 'equivalence types campaign' as grouping is often used to get a different equivalent representation of the same data. However, in chapter 10, we will see how to model functional dependencies in Flock and consequently how $P(ABC)$ and $P(A\{B\}_A C)$ can be considered as to representing the same equivalence type. At that point, our query language will support the same grouping facilities as LDL.

8.3 SOQL query programs, the semantics

In the previous section, we have defined SOQL queries as stratified well-typed SOQL programs. In this section we define the semantics of such queries. Usually, the formal semantics of recursive queries are given as fixpoint semantics, see e.g. (Naqvi and Tsur (1989)). It is possible to define fixpoint semantics in a categorical framework. However, this is beyond the scope of this thesis. So, rather than fixpoints, we use a different strategy to give semantics to recursive queries. In particular, we generalise a method from Abiteboul and Beeri (1988) to SOQL.

This method allows us to rewrite a recursive SOQL query in an equivalent non-recursive semi-SOQL query. Only semi, as some of the types occurring in the transformed query do not have to be in the type graph. The justification for using this transformation rather than a fixed point semantics can be found in (Abiteboul and Beeri (1988)) in which the the authors show that for a simple recursive language the semantics of the rewritten query and the fixed point semantics of the original query coincide.

The strategy in this section is based on the fact that we have the comprehension principle in a general topos. So, the semantics will be based on:

$$\{x \mid \phi(x)\}$$

In which ϕ: $D \rightarrow \Omega$. To use this principle, we have to 'pack' the query into one such expression. If the selections are of the form $x : t$, and their is no recursion in a query, this is an easy task. As an illustration, consider the query:

$$X \supseteq \{x : car \mid \exists y \in Y : col(x) = col(y)\}$$

$$Y \supseteq \{y : ship \mid eng(y) = bungler\_inc\}$$

This can be packed into one expression:

$$\{x : car \mid \exists y : ship(col(x) = col(y) \wedge eng(y) = bungler\_inc)\}$$

For the general strategy, note that as their is no recursion, we can stratify the program such that no two *UVar* variables are defined in the same stratus. Now, from stratus two upward, we simply substitute the definition of an UVAR variable (given in a lower stratus) for every occurrence of that variable in a rule. This will result finally in a query definition in which no intermediate results are used.

If the selection of an intermediate result is of the form $f(x: t)$, it is still relatively easy to pack the query into one expression. Consider e.g. the following modification of the query above:

$$X \supseteq \{x: car \mid \exists y \in Y: col(x) = col(y)\}$$

$$Y \supseteq \{f(y: ship) \mid eng(y) = bungler\_inc\}$$

This can be packed into the expression:

$$\{x: car \mid \exists y: ship(col(x) = col(f(y)) \wedge eng(f(y)) = bungler\_inc)\}$$

So, the general strategy is the same as above, with the difference that each LVar variable y ranging over an UVar variable is replaced by $f(y)$.

Similarly, if the selection of an intermediate result is of the form $eval(c, x)$ we simply have to ensure that the appropriate LVar variables y are replaced by $eval(c, x)$.

The above translation schema always works except when the result variable is defined using a selection of the form $f(x: t)$ or $eval(c, x)$. But the remedy for this problem is the introduction of a new UVar variable Q'. As an illustration, let the result variable be defined by:

$$Q \supseteq \{f(x: t) \mid \phi(x)\}$$

This is rewritten to :

$$Q' \supseteq \{z \mid z = f(x) \wedge x \in Q\}$$

$$Q \supseteq \{x: t \mid \phi(x)\}$$

Note that this is not a valid SOQL query, as z ranges over types rather than over extensions. However, this intermediate step is not meant as a query, but as a step toward the semantics of a query.

The modifications that recursion requires of our general strategy are much more complicated. Therefore, this section is divided into two subsections. In the first subsection, we examine the modifications required by recursion. In the second subsection, we define the formal semantics of a SOQL query.

8.3.1 Recursion

If some of the UVar variables in a program are defined recursively, we obviously cannot replace each reference of the variable by its definition. In fact, if we would replace infinitely often (which we would have to do to eliminate the variable from our query) we would be carrying out the fixpoint computation

manually. Therefore, we need to do a more intelligent replacing schema. In this section, we develop such a schema based on (Abiteboul and Beeri (1988)).

We begin with an example, viz. the most cited recursive query: the ancestor problem. Let the type graph consist of two entity types, *person* and *parent* = *person* × *person* with the two projection functions π_1: *parent* → *person* and π_2: *parent* → *person*. The ancestor problem can be stated in SOQL as:

$$X \supseteq \{x : \textit{parent}\}$$

$$X \supseteq \{\pi_1 \times \pi_2((x, y)) \mid x \in X \wedge y \in X \wedge \pi_2(x) = \pi_1(y)\}$$

Perhaps the most important observation on this query is that for each $x \in X$, $\pi_1(x) \in \pi_1^*(\textit{Ext}(\textit{parent}))$ and $\pi_2(x) \in \pi_2^*(\textit{Ext}(\textit{parent}))$. So, if we define Y to be the set:

$$Y_1 = \{id \times id(a, b) \mid \exists z_1 : \textit{parent}, z_2 : \textit{parent} \; (a = \pi_1(z_1) \wedge b = \pi_2(z_2))\}$$

we know that the solution X is a subset of Y_1. In other words, if we define the set Y_2 as the set:

$$Y_2 = P(Y_1) = \{y \mid y \subseteq Y_1\}$$

then the solution X is an element of Y_2. We can characterise X as an element of Y. Furthermore, we know that $\textit{Ext}(\textit{parent})$ is a subset of X. Hence, if we define the subset Y_3 of Y_2 by:

$$Y_3 = \{y \in Y_2 \mid x : \textit{parent} \rightarrow x \in y\}$$

we know that the solution X is an element of Y_3. The following characterisation of X we can use is that X satisfies that second rule in the query. So, if we define the subset Y_4 of Y_3 by:

$$Y_4 = \{y \in Y_3 \mid (a \in y \wedge b \in y \wedge \pi_2(a) = \pi_1(b)) \Rightarrow (\pi_1(a), \pi_2(b)) \in y\}$$

Now it is easy to see that the solution X is a subset of all elements of Y_4, hence, we compute the solution by:

$$X = \bigcap_{y \in Y_4} y = \{x \in Y_1 \mid \forall y \in Y_4 (x \in y)\}$$

Note that we actually computed the result of the query without fixpoint computations. In fact, we have rewritten the recursive query into a non-recursive variant.

If *parent* has supertypes $s_1, \ldots, s_n$ with functions f_i: s_i → *parent*, only the construction of Y_1 and Y_3 have to be modified. For clarity, we give the complete construction:

$$Y_1 = \{id \times id(a, b) \mid \exists z_1 : parent, z_2 : parent \\ (a = \pi_1(z_1) \wedge b = \pi_2(z_2))\}$$

$$Y_1 = \{id \times id(a, b) \mid \exists z_1 : s_1, z_2 : s_1 \\ (a = \pi_1 \circ f_1(z_1) \wedge b = \pi_2 \circ f_1(z_2))\}$$

$$\ldots$$

$$Y_1 = \{id \times id(a, b) \mid \exists z_1 : s_n, z_2 : s_n \\ (a = \pi_1 \circ f_n(z_1) \wedge b = \pi_2 \circ f_n(z_2))\}$$

Again, Y_1 is the set of all *person-pairs* that could be the representation of an ancestor. Hence, Y_2 is constructed exactly as above:

$$Y_2 = P(Y_1) = \{y \mid y \subseteq Y_1\}$$

In the construction of Y_3, we have to take again the s_i into account, this gives:

$$Y_3 = \{y \in Y_2 \mid x : parent \rightarrow x \in y \wedge x_1 : s_1 \rightarrow f_1(x_1) \in y \cdots \wedge \\ x_n : s_n \rightarrow f_n(x_n) \in y\}$$

And the final constructions remain the same, i.e.:

$$Y_4 = \{y \in Y_3 \mid (a \in y \wedge b \in y \wedge \pi_2(a) = \pi_1(b)) \Rightarrow \\ (\pi_1(a), \pi_2(b)) \in y\}$$

$$X = \bigcap_{y \in Y_4} y = \{x \in Y_1 \mid \forall y \in Y_4 (x \in y)\}$$

In (Abiteboul and Beeri (1988)) a proof is sketched that a similar construction can be used for all, stratified, recursive queries in a simple language. Our position is the opposite, we define the semantics of a recursive query using this method.

A formal description of the rewriting algorithm is extremely tedious and unreadable. Therefore, we refrain from giving this. Instead, we give one more examples, and after that an informal version of the algorithm.

The first example, is one with mutual recursion. Let the type graph consist of the entity types a, b, c, $s = a \times a \times c$ and $t = b \times c$ together with the projection functions $\pi_1^s : s \rightarrow a$, $\pi_2^s : s \rightarrow a$, $\pi_3^s : s \rightarrow b$, $\pi_1^t : t \rightarrow b$, $\pi_2^t : t \rightarrow c$. Consider the following query:

$$X \supseteq \{x : s\}$$

$$X \supseteq \{\pi_1^s \times \pi_2^s \times \pi_1^t((x_1, x_2, y)) \mid x_1 \in X \wedge x_2 \in X \wedge y \in Y \\ \wedge \pi_3^s(x_1) = \pi_3^s(x_2)\}$$

$$Y \supseteq \{y : t\}$$

$$Y \supseteq \{\pi_3^t \times \pi_2^s((x, y)) \mid x \in X \wedge y \in X \wedge \pi_1^s(x) = \pi_2^s(x)\}$$

The solution to this program is given by $\mathbf{Z} = (\mathbf{X}, \mathbf{Y})$, with $\pi_1 : \mathbf{Z} \rightarrow \mathbf{X}$ and $\pi_2 : \mathbf{Z} \rightarrow \mathbf{Y}$ in which $\mathbf{X}$ is the solution for X and $\mathbf{Y}$ the solution for Y. As these two solutions depend on each other, they have to be computed simultaneously. As above, we first determine the set in which the final solution will be contained: Inspecting the rules, we get:

$$\begin{aligned} x \in \mathbf{X} \Rightarrow & \exists q \in Ext(s)(\pi_1^s(x) = \pi_1(q)) \wedge \\ & \exists q \in Ext(s)(\pi_2^s(x) = \pi_2(q)) \wedge \\ & (\exists q \in Ext(s)(\pi_3^s(x) = \pi_3(q)) \vee \\ & \exists y \in \mathbf{Y}(\pi_3^s(x) = \pi_1(y))) \\ y \in \mathbf{Y} \Rightarrow & (\exists q \in Ext(t)(\pi_1^s(y) = \pi_1(q)) \vee \\ & \exists x \in \mathbf{X}(\pi_1^s(y) = \pi_3(x))) \wedge \\ & \exists q \in Ext(t)(\pi_2^t(y) = \pi_2^t(q) \end{aligned}$$

Simplifying these conditions, we get:

$$\begin{aligned} x \in \mathbf{X} \Rightarrow & \exists q \in Ext(s)(\pi_1^s(x) = \pi_1(q)) \wedge \\ & \exists q \in Ext(s)(\pi_2^s(x) = \pi_2(q)) \wedge \\ & (\exists q \in Ext(s)(\pi_3^s(x) = \pi_3(q)) \vee \\ & \exists q \in Ext(t)(\pi_3^s(x) = \pi_1(q))) \\ y \in \mathbf{Y} \Rightarrow & (\exists q \in Ext(t)(\pi_1^s(y) = \pi_1(q)) \vee \\ & \exists q \in Ext(s)(\pi_1^s(y) = \pi_3(q))) \wedge \\ & \exists q \in Ext(t)(\pi_2^t(y) = \pi_2^t(q) \end{aligned}$$

This yields the approximations:

$$\begin{aligned} X_1 &\supseteq \{x : s\} \\ X_1 &\supseteq \{\pi_1^s \times \pi_2^s \times \pi_1^t(x, x, y) \mid x : s \wedge y : t\} \\ Y_1 &\supseteq \{y : t\} \\ Y_1 &\supseteq \{\pi_3^s \times \pi_2^s(x, y) \mid x : s \wedge y : t\} \end{aligned}$$

And following, we get the approximation Z_1 of $\mathbf{Z}$ as:

$$Z_1 = \{z : s_1 \times s_2 \mid \pi_1(z) \in X_1 \wedge \pi_2(z) \in Y_1\}$$

As in the example above, we take the power set of Z_1, i.e. Similar as above, the next step is:

$$Z_2 = \{z : P(s_1 \times s_2) \mid z \subseteq Z_1\}$$

Now we have, $\mathbf{Z} \in Z_2$. As above, we now proceed to filter out those elements of Z_2 that cannot be the solution using the rules of the program as a filter. The

first rule translates to the filter:

$$Z_3 = \{z \mid z \in Z_2 \wedge (x : s \Rightarrow \exists q \in z(\pi_1(q) = x)\}$$

The second rule translates to:

$$Z_4 = \{z \mid z \in Z_2 \wedge (z_1, z_2 \in z(\pi_3^s \circ \pi_1(z_1) = \pi_3^s \circ \pi_1(z_2) \Rightarrow$$
$$\exists z_3 \in z((\pi_1^s \circ \pi_1) \times (\pi_2^s \circ \pi_1) \times (\pi_1^t \circ \pi_2)(z_1, z_1, z_2) = \pi_1(z_3)))\}$$

The third rule translates similar to the first:

$$Z_5 = \{z \mid z \in Z_4 \wedge (y : t \Rightarrow \exists q \in z(\pi_2(q) = y)\}$$

And finally, the fourth rule translates similar as the second:

$$Z_6 = \{z \mid z \in Z_5 \wedge (z_1, z_2 \in z(\pi_1^s \circ \pi_1(z_1) = \pi_2^s \circ \pi_1(z_1) \Rightarrow$$
$$\exists z_3 \in z((\pi_1^t \circ \pi_2) \times (\pi_2^t \circ \pi_2)(z_1, z_1, z_2) = \pi_2(z_3)))\}$$

Similar as above, we have that **Z** is the intersection of all the remaining possible solutions, i.e.

$$\mathbf{Z} = \{z \in Z_1 \mid \forall a \in Z_6(z \in a)\}$$

Which finally yields:

$$\mathbf{X} = \{\pi_1(z) \mid z \in Z_4\}$$
$$\mathbf{Y} = \{\pi_2(z) \mid z \in Z_4\}$$

The general approach will be clear by now, it is given by the following informal algorithm:

1) Generate from the rules a set X_1 such that the, joint, solution **X** is a subset of X_1
2) Let X_2 be the power set of X_1, such that the, joint, solution **X** is an element of X_2
3) Filter out all elements of X_2 that do not satisfy the rules, the remaining set is called X_3.
4) The joint solution **X** is the intersection of all elements from X_3.

As in the examples, this algorithm allows us to rewrite recursive SOQL queries into non-recursive semi-SOQL queries. The result queries are semi-SOQL queries rather than SOQL queries as some of the intermediate types used in the program do not have to be in DBS_{TG}. However, for the semantics we define in the next subsection, it does not make any difference.

8.3.2 The semantics

In the previous subsection, we have seen that recursive SOQL queries can be linearised. This means that we after 'linearising' each stratus in a program P, the resulting program P' can be stratified such that each stratus contains the definition of exactly one UVar variable X. Moreover, each stratus in P' is non-recursive. So, such a stratus is of the form:

$$X \supseteq \{\alpha_1(z) \mid \phi_i(z)\}$$
$$\ldots$$
$$X \supseteq \{\alpha_n(z) \mid \phi_n(z)\}$$

With all the UVar variables in the ϕ_i belonging to the lower strata. Now, the different rules in a stratus simply give the different ways an element of X can be formed. Hence, it can be rewritten to one rule as follows, using the rewriting sketched in the beginning of this section:

$$X = \{x \mid (x = \alpha_1(z) \wedge \phi_1(z)) \vee \cdots \vee (x = \alpha_n(z) \wedge \phi_n(z))\}$$

Now, from the lowest stratus upwards, we simply substitute the right hand side of this equation for each occurrence of the left-hand side UVar variable in higher strata. This will result in the following expression:

$$Q = \{x \mid \phi(x)\}$$

in which ϕ does not contain any UVar variables. The semantics of this rule are simply given by the comprehension principle as follows:
Let ϕ: $s \rightarrow \Omega$, for some s in the type DBS_{TG}. Moreover, let $s_1, \ldots, s_n$ be all the subtypes of s, with functions f_i: $s_i \rightarrow s$. Define Q_0 by applying the comprehension principle to

$$Q_0 = \{x : s \mid \phi(x)\}$$

and

$$Q_1 = \{x : s_i \mid \phi(f_i(x))\}$$

Then, the result of the query is defined by:

$$Q = Q_0 + Q_1 + \cdots + Q_n$$

8.4 Rules in the database

The SOQL query language defined in this chapter supports a set of recursive queries, usually, this is a feature of deductive databases. Another important aspect of these deductive databases is that *rules* can be stored as well. In this section, we show how rules can be stored in Flock.

As already mentioned in chapter 3, exponential types can be used to store the usual rules. As the recursive rules from deductive databases can be written as recursive functions in some functional language, e.g. the following recursive function computes the transitive closure of a relation:

$$\begin{aligned} &\textit{transclose } X \times Y: \\ &\qquad \text{if } (x, y_1), (y_1, y_2) \in X \times Y \wedge (x_1, y_2) \notin X \times Y \rightarrow \\ &\qquad\quad \textit{transclose } X \times Y \cup \{(x_1, y_2)\} \\ &\qquad \textit{else } X \times Y \end{aligned}$$

However, this can be considered as cheating. For, if we allow general recursive functions to be entered as elements of an exponential type, we cannot use a general topos for the semantics of Flock. Because there is no way to guarantee that this functions exists in a topos T. Therefore, we made the assumption in chapter 3 that all elements in an exponential type are actually constructed. So, we have to show that recursive rules can be used to construct the related recursive function.

In the previous section, we have seen that each stratified well-typed SOQL program has an expression of the form:

$$\{x : t \mid \phi(x)\}$$

for its semantics. Moreover, such a ϕ is a function:

$$\phi: D_1 \times P(D_2) \times \cdots \times P(D_n) \rightarrow \Omega$$

for appropriate objects D_i; note that some of these D_i may denote the same object. To rewrite this expression to a function, means that we want to construct a function ψ:

$$\psi: P(D_2) \times \cdots \times P(D_n) \rightarrow P(D_1)$$

such that:

$$\psi(\alpha_2, \ldots, \alpha_n) = \{x \in D_1 \mid \phi(x, \alpha_2, \ldots, \alpha_n) = \textit{true}\}$$

But this function can easily be constructed using the universal property of ϵ_{D_1}:

1) Let X with the functions

$$!: X \rightarrow 1, f: X \rightarrow D_1 \times P(D_2) \times \cdots \times P(D_n)$$

be the pullback of

$$\phi: D_1 \times P(D_2) \times \cdots \times P(D_n) \rightarrow \Omega, \textit{true}: 1 \rightarrow \Omega.$$

Then:

$$(x, \alpha_1, \ldots, \alpha_n) \in f(X) \Leftrightarrow \phi(x, \alpha_1, \ldots, \alpha_n) = \textit{true}$$

2) Clearly, f is a monic function, hence, by the universal property of ϵ_{D_1}, we have the function ψ as defined above, i.e.

$$\psi\colon P(D_2) \times \cdots \times P(D_n) \to P(D_1)$$

with:

$$\psi(\alpha_2, \ldots, \alpha_n) = \{x \in D_1 \mid \phi(x, \alpha_2, \ldots, \alpha_n) = \mathit{true}\}$$

So, each recursive SOQL query yields a recursive function of the appropriate exponential type.

The drawback of this solution to store rules in the database is that the user has to invoke the rules explicitly in a query; something that is done automatically in a deductive database. In the Chapter 10, we show how this problem can be overcome in Flock.

8.5 Conclusions

In this chapter, we defined a query language, called SOQL, for databases consisting of structured objects. SOQL is a calculus defined along the usual lines. The highlights of this calculus are:

1) It is a typed declarative language.
2) Set values can be manipulated using the set-theoretic operations.
3) Queries may be recursive, provided they are stratified.

As an aside, note that we do not support a *grouping*-operator in the query language as some other formalisms do. The reason for this ommission is simply that the equivalence relation on classes (the types of complex objects), as defined in Chapter 10, automatically entails grouping.

The semantics of non-recursive queries are defined along the usual lines. For recursive queries, however, we resort to a trick. Rather than developing the formal semantics of fixed points or unification within our categorical framework, we resort to an operational semantics given in (Abiteboul and Beeri (1988)). The motivation for these operational semantics is that a formal definition would require yet another extensive subset of Category theory.

This chapter is finished with a short note on how the semantics of recursive queries can be exploited to store rules in the database. The drawback of this solution is that the rules have to be invoked explicitly in a query. The solution to this problem can be given as soon as we have an object identity.

Chapter 9

Incomplete Information

Often, not all information on some real world entity is known when it is decided that this entity has to be represented in the database. Examples can be found e.g. in police databases. Solving a crime can be likened to solving a jigsaw puzzle of which the pieces are partial descriptions of the crime UoD. An additional problem is that these pieces are not readily available, but have to be found first. If the officer investigating the crime wishes to store her information on the crime UoD in a database, she may run into problems. The complete description of an entity in this UoD may be scattered over various pieces of the puzzle. So, if the database can only represent completely specified real world entities, they can only be represented when if all pieces concerning them are found. This is of course rather awkward if this real world entity is e.g. the criminal.

Less prosaic examples of the importance of incomplete information can be found in the usual-business like applications of databases. In this chapter, we show how incomplete information can be represented in Flock using so-called *incomplete structured objects*, or *is-objects*. In fact, we have already seen a way to represent incomplete information, viz. queries in SOQL. A query can be seen as a partial, or incomplete, description of a structured object. Evaluation of the query yields those structured objects that satisfy this partial description. One important difference between SOQL queries and *is*-objects is that *is*-objects do not necessarily refer to the extension of the database. However, the query analogy is good enough to built some intuition on how *is*-objects can be defined.

In the first section of this chapter, we informally describe entity types with incomplete information and show how this affects the definition of structure types. Moreover, we describe the effect of incomplete information on the query language. The formal definition of *is*-objects together with their semantics is given in the second section. As we have now a new kind of objects, we have to define how methods, or second order objects, can be applied to these objects.

This is the topic of the third section. In the fourth and final section, we extend SOQL to allow querying of databases of *is*-objects.

9.1 *is*-objects, informally

Both in Chapter 1 and in the introduction above, we have stressed the importance of incomplete information. And we are not among the first to recognise the importance. Indeed, several attempts have been made to solve the problem of the representation of incomplete information in a database. Perhaps the best known approach is the use of Null-values in the extended relational model (Codd (1979)). However, this solution is not general enough because a Null-value means that no information is known save that a value for an attribute does exist in the UoD. A much more general solution is outlined by Reiter (1984) for deductive databases as outlined in Chapter 2. In principle, it is possible to adapt this proof-theoretic approach to the categorical semantics of Flock. However, we feel that this would seriously compromise the functional simplicity of Flock. Therefore, we choose another approach, which resembles the approach taken by Lipski, jr. (1979) and even more the approach of Jaegermann (1978).

In this section, we give an informal introduction to our philosophy towards incomplete information. To ease this introduction, it is split into two subsections. In the first subsection, we restrict ourselves to finite disjunctive information and we describe what effect such information could have on a query language. In the second subsection, we generalise the first subsection to full-fledged incomplete information.

9.1.1 Finite disjunctive information

In this subsection, we restrict ourselves to limited uncertainty in that we assume that the incomplete information can be specified by a finite disjunction. As an illustration, consider the entity type $car = speed \times colour$, with the projection functions $spd: car \to speed$ and $col: car \to colour$. The type of incompleteness we are considering in this section is: there is a car X, with $col(X)$ is either red or blue and its speed is either 56 or 65. This specification could be written more formally as a query-like expression:

$$X \in \{x: car \mid (col(x) = red \vee col(x) = blue) \wedge (spd(X) = 56) \vee spd(x) = 65)\}$$

For the left-hand side of this specification, the important difference with a query is that x does not range over the extension of *car*, but over the type itself.

One option would be to represent this information by storing all different possibilities separately, i.e. all elements in the 'set' at the left-hand side and encode that these entries refer to the same real world object by giving them all the same identity. However, we do not have an identity yet, and, therefore, we use

another option, viz. identify X with the set consisting of all possibilities. This means that under this option, we see the value of an incompletely specified entity of type e as a finite subset of D_e.

This suggests that for each entity type e, we can construct an incomplete entity type ie such that $D_{ie} = FP(D_e)$, where $FP(x)$ denotes the set of all finite subsets of x. All the more as it is straightforward to map a function $f: e_1 \rightarrow e_2$ to an incomplete function $if: ie_1 \rightarrow ie_2$; simply map f to f^*, i.e. $if(X) = \{f(x) \mid x \in X\}$. This is more or less the approach we take, although the approach has to be more subtle for the semantics. We start with a syntactic translation.

If *TG* is a type graph then *ITG*, the incompleteness type graph, is constructed from *TG* by replacing all entity types by their incomplete counterparts, and similarly, the semantical transformations are replaced by their incompleteness counterparts.

Of course, such a syntactic construction does not tell us very much, it only states that there is a node ie in *ITG* for each node e in *TG* and that there is a semantical transformation if in *ITG* for each semantical transformation f in *TG*. In other words, the above construction only gives us the new names not their semantics. These semantics are the following step.

To make the definition of the semantics of *ITG* easier, we assume that there are no user defined transformations between entity types in *TG*, save those between the basic entity types. We now proceed as follows:

1) If e is a basic entity type, then $Sem(ie) = P(Sem(e))$.

2) If f is a transformation between two basic entity types, then $Sem(if) = (Sem(f))^*$ as explained above.

3) For all the other entity types and transformations, we simply built the new semantics analogically to the old semantics. For example,

$$Sem(e_1 \times e_2) = Sem(e_1) \times Sem(e_2)$$

To see the difference with this 'official definition' and the one suggested above, note that under these semantics, the car X is mapped to $\{red, blue\} \times \{56, 65\}$ rather than to $\{(red, 56), (red, 65), (blue, 56), (blue, 65)\}$; which is a subtle, but important difference. Why this choice for the semantics is the right one, and why the subtle difference is important, will be explained in the next subsection.

Now that we have 'defined' the semantics of the new type graph *ITG*, it is straightforward to generalise the definition of structure types to the definition of incomplete structure types, or *is*-types. We will not carry out this exercise at this place, but trust on the reader's intuition. Instead, we look at the effect of incompleteness on queries. Again, rather than any formalising, we look at an example. Let the car X, we defined above, i.e.

$$X = (icar, \{red, blue\} \times \{56, 65\})$$

be in the extension of *icar*. Consider the query:

$$Q_1 \supseteq \{q : icar \mid col(q) = red \vee col(q) = blue \vee col(q) = green\}$$

Clearly, whatever colour X turns out to have, it will always be *red*, *blue* or *green*. Hence, X ought to be in the result of Q_1.

For the next example, consider the query:

$$Q_2 \supseteq \{q : icar \mid col(q) = purple \vee col(q) = green\}$$

Now, it is obvious that whatever colour X turns out to have, it will be neither *green* nor *purple*. Hence, X does not belong to the answer set of Q_2.

For 'normal' objects, these two cases exhaust the possibilities, an object belongs to the answer set or it does not; there is no room for doubt. However, with *is*-objects there is. To illustrate the doubt, consider the query:

$$Q_3 \supseteq \{q : icar \mid col(q) = red \vee col(q) = green\}$$

If X turns out to be *red*, it clearly is an answer to Q_3, but if X turns out to be *blue* it clearly is not. So X may be answer to X; note that this is reminiscent of the 'maybe' in e.g. (Codd (1979)).

This 'doubt' introduced by the incompleteness means that we can and have to make our query language more expressive. The user should be able to express whether she wants certainty or would allow some doubt. For the query language, this means that we might introduce a 'maybe' operation, that would allow the user to express her willingness to accept doubt in the answer. However, we adopt a different strategy that is sketched at the end of the next subsection.

9.1.2 General incompleteness

In practice, it will be a reasonable assumption that if a real world entity is represented in the database through disjunction, then the disjunction will be finite. If only as some user has to specify the possibilities. So, the finite set approach to representations will be adequate for such incompleteness. However, as always, the picture drastically changes when we allow *negation*, i.e. if we allow specification by 'negative facts'. Because, in general, a negative fact does not specify a finite subset of a domain. For example, if we specify a car by:

$$X \in \{x : car \mid col(x) \neq red\}$$

the right-hand side 'set' is infinite if *colour* has an infinite domain.

So, if we allow *is*-object specification by negative facts, we can no longer restrict ourselves to $FP(domain)$. One solution seems to be to specify objects my two finite subsets, say $T(X)$ and $F(X)$ such that $T(X)$ contains the positive facts known of X, while $F(X)$ contains the negative facts. However, this is not as straightforward as it may seem:

1) One has to ensure that $F(X)$ and $T(X)$ are consistent, i.e. it should not occur that $F(X)$ states that $col(X)$ is *red*, while $T(X)$ states that $col(X)$ is definitely not red.

2) The second problem is more subtle: what is the status of domain elements that are neither in $F(X)$ nor in $T(X)$?

Rather than trying to solve these problems, we choose another approach, inspired by SOQL.

In defining the semantics of a query, we constructed a function $\phi\colon D \to \Omega$ for each query. The result of the query is completely determined by this function ϕ. In fact, in terms of Chapter 6, the result of the query:

$$Q = \{x\colon D \mid \phi(x)\}$$

is the subobject $i\colon Q \to D$ such that ϕ is the character of i. Hence, we may as well identify Q and ϕ. This observation is the basis for our approach to the representation of general incompleteness. Although our approach will be more subtle than simply replacing D with Ω^D for each entity type. This straightforward implementation of our observation would cause severe difficulties and would yield unintuitive semantics of *is*-objects as can be seen form the following sketch.

The attentive reader will note the similarity between this straightforward approach towards incomplete information and our approach to methods. Indeed, in both cases, entities are mapped to objects in $C^{\rightarrow}$. So, it will not come as a surprise that *is*-objects and methods contain similar problems towards inheritance. The main problem for *is*-objects is, as to be expected, caused by the possibility of negative facts. To illustrate the problem, consider a car specified by:

$$X \in \{x\colon car \mid (col(x) = blue \wedge spd(x) = 56) \vee (col(x) = red \wedge spd(x) = 65)\}$$

Whatever car X turns out to be, it will always be either *red* or *blue*. Now, consider the following specification:

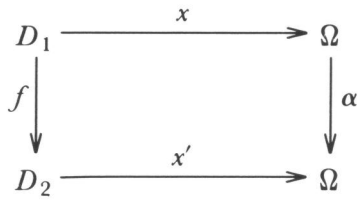
$$X \in \{x\colon car \mid (col(x) = red \wedge spd(x) = 56) \vee (col(x) \neq red \wedge spd(x) = 65)\}$$

Now X may turn out to have any colour! In other words, negative facts are not automatically inherited downward. This problem is perhaps even better illustrated by looking at the semantics of *is*-objects as sketched above.

Let s_{e_1} be a structure type. Moreover, let $f\colon e_1 \to e_2$ be an edge in the type graph. For structure types, a structured object is completely specified by a value for the root type. In this example this means that the e_2-valued can be inferred from the e_1-value by simply applying f. For *is*-objects, it is much more complicated, because a function $x\colon D_1 \to \Omega$ and a function $f\colon D_1 \to D_2$ do not imply a function $fx\colon D_2 \to \Omega$. In fact, the situation is reversed in that a function $x\colon D_2 \to \Omega$ and a function $f\colon D_1 \to D_2$ imply a function $x \circ f\colon D_1 \to \Omega$.

One might expect that this problem is caused by the fact that x is a $C^{\rightarrow}$ object and that f is a C-arrow, not a $C^{\rightarrow}$-arrow. In other words, one might expect that rather than f, we should consider the $C^{\rightarrow}$ arrow (f, α) with some

α: $\Omega \to \Omega$. This should then yield an $x' = (f, \alpha)(x)$ which is a $C^{\to}$ arrow such that the following diagram commutes:



This raises some question:

1) Does such an x' always exist?

2) If it exists, is it unique?

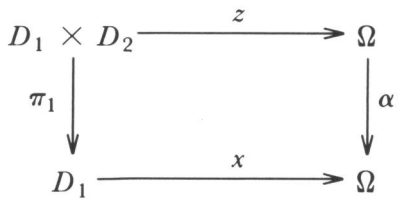
In other words, we are asking if (f, α) is a function from the set of all $D_1 \to \Omega$ to all $D_2 \to \Omega$ arrows?

1) The answer to the first question is in general of course no. For example, if f is a constant function, α is the identity, and x maps d_1 to true and all other elements of D_1 to false, then there is no x' that would make the diagram commute.

2) The answer to the second question is also no if f is not surjective, for in that case x' may map elements of $D_2 - f^*(D_1)$ to whatever element of Ω it pleases. Only if f is surjective, such an x' is unique if it exists; because surjective functions cancel on the right.

The answers to these questions should not come as a surprise, because subtyping of exponential types is anti-monotonic in the *exponents*.

The similarity with methods suggests that *is*-objects have to be specified not only by their *root*-value, but that the value at all levels should be specified. Moreover, it should be ensured that the specifications at the various levels are consistent. To sketch the consistency requirement and its consequences, consider the *is*-objects $x: D_1 \to \Omega$ and $y: D_2 \to \Omega$. Suppose we want to construct the *is*-object $z = (x, y)$: $D_1 \times D_2 \to \Omega$. Similar to the consistency of methods, this means that there should be an α: $\Omega \to \Omega$ such that, among others, the following diagram commutes:



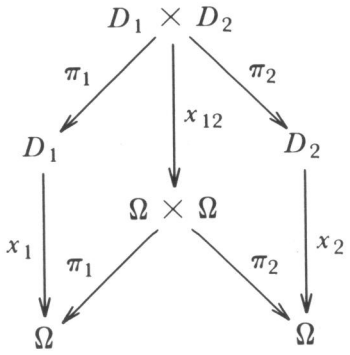
Now, this implies that $z \neq (x \wedge y)$: $D_1 \times D_2 \rightarrow \Omega$, as for this z, there would be no α: $\Omega \rightarrow \Omega$ such that the above diagram commutes. In fact, for all the elementary logical formula containing x and y, there does not exist such an α. This means:

1) Given two objects x and y, the database designer has to specify which object has to be considered as the object (x, y).

2) Moreover, she can not use the intuitively obvious choices for this object.

Given these observations, the reader will not be surprised that we have a more subtle implementation of our Ω^D observation. In fact, the implementation we use in this thesis is the natural one in a categorical setting. The above sketch can be seen as an example of the pitfalls that awaits the traveler that leaves the categorical path.

In fact, we have already sketched our approach in the previous subsection, i.e. we first map the basic types to $B \rightarrow \Omega$ and, following, the semantics of the constructed types are reconstructed, e.g. $Sem(ie_1 \times ie_2) = Sem(ie_1) \times Sem(ie_2)$. The consistency rules that are suggested by the analogy with methods are now automatically true, as can be seen by the following diagram in which both 'rectangles' should and do commute:



Moreover, note that this approach again ensures that *is*-objects are completely specified by their value for the *root*-type. These two observations should convince the reader that this approach to incomplete information is the right one, as we stated in the previous subsection.

Now that we have sketched how *is*-objects look like, we can re-examine the effect of *is*-objects on queries. In the last subsection, we have seen that in limited incompleteness case, there may be doubt whether an object belongs to the answer of a query. To generalise this to *is*-objects, note that semantically, *is*-objects can be seen as arrows $D \rightarrow \Omega$. The logical connectors are defined on Ω, so we can compose *is*-objects with the logical connectors resulting in new *is*-objects. For example, if o_1, o_2: $D \rightarrow \Omega$, then:

$$o_1 \wedge o_2 = \wedge \circ (o_1, o_2): D \rightarrow \Omega$$

To see that this use of the logic of the topos is enough to express the doubt of the previous subsection, note that for two subsets A and B of D we have:

$$(A \cap B \neq \varnothing) \Leftrightarrow (\chi_A \wedge \chi_B \neq \mathit{false}_D)$$

More on the use of the logic of the topos in the query language can be found in the last section of this chapter, in which the query language is officially defined.

9.2 Formalising incompleteness

In the previous subsection, we have sketched our approach to incompleteness as similar to our approach to methods. In this section, we formalise the sketch, along the lines of our formalisation of method (types). Recall, that we do not consider methods in this chapter, i.e. we are only defining what incompleteness means for 'static' objects and their type. In other words, we are not concerned with exponential types.

Syntactically, *is*-types are much easier to define than method types, as virtually all information that is needed is available in the type graph. The only information that might be missing is related to user defined functions. Now, in Chapter 3, we already made the assumption that the user defined functions are constructed categorically; the information we miss is available through these construction. But before we 'dive' into this problem, we first define the *incompleteness* semantics of the type graph assuming there are no user defined functions.

Note that we said we are going to define the incompleteness semantics of the yupe graph. This implies that we are not foing to define syntactically a new type graph as we suggested in the first section. We only defined this new type graph for the readers convenience. We think that by now the reader has developed enough intuition and thus that we do not have to make this syntactic distinction any longer.

As already indicated in the previous section, the definition of the incompleteness semantics follows the line of the definition of the semantics. So, we start by giving the semantics of the type graph:

DEFINITION 9.1

Let TG be a type graph and T a topos, such that the domains of the basic type graphs are objects in T. The *incompleteness semantics* of TG are given by the function $ISem$ defined inductively as follows:

1) If $<e, D> \in B$, then $ISem(<e, D>) = \Omega^D$, for $e \neq 0$ and $e \neq 1$. If $e = 0$ or $e = 1$ then $ISem(e) = Sem(e)$.

2) Let δ be a diagram in TG, then:

$$ISem(lim\delta) = \lim(ISem(\delta)),$$

$$ISem(co-lim\delta) = co-lim(ISem(\delta)).$$

3) For $e \in TG$, $ISem(P(e)) = P(ISem(e))$.

4) For an edge e, $ISem(e)$ is the corresponding arrow in T, e.g.

$$ISem(\pi_A\colon A \times B \to A) = \pi_{ISem(A)}\colon$$
$$ISem(A) \times ISem(B) \to ISem(A).$$

Now, for the user defined functions in the type graph, we know that they are constructed explicitly in T. All the types that are used in the construction are constructed from the basic types. Although some of the intermediate types used in the construction do not have to be in the type graph, we can assume for a moment that they do. This means that we can give *ISem*-semantics to all of the types; with ϵ_A mapped to $\epsilon_{ISem(A)}$. But then it is clear how to give semantics to the user defined function. Simply perform the construction on the *ISam*-semantics rather than on on the *Sem*-semantics. If *consrt* denotes the normal construction, then *ISem*(*constr*) denotes the construction as outlined above. This leads to the following definition:

DEFINITION 9.2

If $f\colon e_1 \to e_2$ in TG and $Sem(f) = constr(f)$, then

$$ISem(f) = ISem(constr(f)).$$

Note that if two entity types are isomorphic through the 'old' semantics, they will also be isomorphic through the 'new' semantics. Moreover, it does not matter whether the isomorphism is user-defined, or can be inferred automatically. Together these two definitions do not only define the semantics of the type graph, but also of the equivalence type graph.

The following step is the definition of *incomplete structure types* and their semantics. Syntactically, the definition of these *is*-types is identical to that of structure types, with the distinction that we specify *is*-type rather than structure type. So, we get:

DEFINITION 9.3

Let TG be a type graph and t an entity type in TG, the incomplete structure type, or *is*-type induced by t is defined by the expression:

$$incomplete\ structure\ type\ name = ([t], TG)$$

$[t]$ is called the root entity type, or simply root, of the *is*-type *name*.

As before, we will use the name $is_{[t]}$ for the name of the *is*-type with root $[t]$ in the rest of this thesis. Moreover, we will again assume that the type graph is known, i.e. we will not specify the type graph.

The semantics of the *is*-types is again the straightforward translation of the semantics of structure types to the new setting. Recall that $Fsub(C)$ is the

category consisting of all finite subcategories of C. Moreover, $SG([t])$ is the subgraph of ETG induced by $[t]$. Then we we get:

DEFINITION 9.4

Let $[t]$ be a node in ETG, the semantics of the *is*-type $is_{[t]}$, denoted by $ISem_{IS}(is_{[t]})$ is given by the following object in $Fsub(C)$:

$$Nodes(ISem_{IS}(is_{[t]})) = \{ISem(n) \mid n \in Nodes(SG([t]))\},$$
$$Edges(ISem_{IS}(is_{[t]})) = \{id_n \mid n \in Nodes(ISem_{IS}(is_{[t]}))\} \cup \{ISem(f) \mid f \in Edges(SG([t]))\}$$

Finally, the subtyping of *is*-types is again the direct translation of the subtyping of structure types, so we get:

DEFINITION 9.5

Let $is_{[r_1]}$ and $is_{[r_2]}$ be *is*-types, $is_{[r_1]}$ is called a subtype of $is_{[r_2]}$, denoted by $is_{[r_1]} \leqslant is_{[r_2]}$, if $ISem_{IS}(is_{[r_2]})$ is a subcategory of $ISem_{IS}(is_{[r_1]})$.

Before we can generalise SOQL to handle *is*-types and *is*-objects, we first have to define how methods interact with *is*-objects. This is the subject of the next section.

9.3 Methods and *is*-objects

Methods and method types were given semantics assuming that methods act on structured objects. Now, the methods have to act on *is*-objects, this means that we have to change the semantics of methods and method types. We will call these new semantics *ISem*-semantics to show the link with the *ISem*-semantics of *is*-objects. Note that this does *not* mean that the method is incompletely specified, but that it is defined to act on *is*-objects rather than on structured objects. Incompletely specified methods and their semantics are outside the scope of this thesis; if only because incomplete specified methods will occur much less frequent in practice.

The first step towards giving new semantics to methods is to give new semantics to exponential types. Because we did not include exponential type constructions for *is*-objects so far. Similar to the user defined functions in a type graph, the new semantics of such exponential types are straightforward, as we made the assumption that inhabitants of exponential types are the name of arrows in the category that are explicitly constructed. So, we can proceed similar to the process of giving semantics to these user defined functions. In other words, we repeat the construction of the function over the *ISem* semantics rather than over the *Sem*-semantics. So, we get for the type:

DEFINITION 9.6

Let A_B be an exponential type in the type graph, then the *ISem* semantics are defined by:

$$ISem(A^B) = ISem(A)^{ISem(B)}$$

For completeness sake, we will also give the new semantics of an exponential entity in accordance with the definition of the previous section:

DEFINITION 9.7

Let $f = \hat{g}: A^B$, i.e. f is the name of $g: B \to A$, then

$$ISem(f) = (\hat{ISem}(g))$$

Now that we have given new semantics to exponential entity types, we can faithfully copy the definition of method types from Chapter 7.

So, the first step is giving *ISem* semantics to se-types, analogically to their *Sem* semantics, i.e:

DEFINITION 9.8

Let *soet* be the se-type defined by:

se-type soet =

sink-type: s_1 (*a structure type*)

sources and *targets*:

$s_1 \to t_1$

. . .

$s_n \to t_n$

The *ISem* semantics of *soet* are given by the object $ISem(soet)$ in C that is constructed as follows:

1) First construct the type:

$$X_{soet} = ISem(s_1^{t_1}) \times \cdots \times ISem(s_1^{t_1})$$

2) In the following step, we use the same symbol for a structure type and its root entity type, to avoid a cluttering of sub- and superscripts. Define the set V as follows:

$V = \{(f, g) \mid f: s_i \to s_j, g: t_i \to t_j$ *in the def of soet*$\}$

Furthermore, define the predicate ϕ_{soet} by:

$\phi_{soet}(x, f, g) \Leftrightarrow ISEM(g) \circ \pi_i(x) = \pi_j(x) \circ ISEM(f)$

where $(f, g) \in V$ and $f: s_i \to s_j$ and $g: t_i \to t_j$. The predicate $\Phi_{soet(x)}$ is now defined by:

$$\Phi_{soet}(x) \Leftrightarrow (\wedge_{(f, g) \in V} \Phi_{soet}(x, f, g))$$

3) Finally, we get to the semantics:

$$ISem(soet) = \{x \in X_{soet} \mid \Phi_{soet}(x)\}$$

Note that we do not need extra equations to guarantee the consistency with regard to the 'truth'-mappings: this consistency hold automatically, as we have seen before.

The following step is to give new semantics to TG_2. Again, this is a direct translation of the 'old' definition, i.e.:

DEFINITION 9.9

Let TG_2 be a second order type graph, semantically, TG_2 is a subgraph of C. The mapping *ISem* is given by:

1) A node n_i is mapped to $ISem(n_i)$ as defined above.
2) The edge e_i^j is mapped to the unique function $f: ISem(n_j) \to ISem(n_i)$ induced by the universal property of $ISem(n_i)$ and the projection function $X_j \to X_j$.

And finally, we can give the new semantics for method types:

DEFINITION 9.10

Let $m_{n_i} = (n_i, TG_2)$ be a second order structure type and let $G(n_i, TG_2)$ be the maximal connected directed subgraph of TG_2 with n_i as root. Then the *ISem*-semantics of m_{n_i} are given by the element $ISem(m_{n_i})$ of $Fsub(C)$ which has:

$$Nodes(ISem(m_{n_i})) = \{ISem(n_j) \mid n_j \in Nodes(G(n_i, TG_2))\}$$

$$Edges(ISem(m_{n_i})) = \{id_n \mid n \in Nodes(ISem(m_{n_i}))\} \cup \{ISem(e) \mid e \in Edges(G(n_i, TG_2))\}$$

As an aside, note that these new semantics have no influence on the subtyping relationship on method types, as we have the following trivial lemma:

LEMMA 9.11

Let mt_1 and mt_2 be two method types, $Sem(mt_2)$ is a subcategory of $Sem(mt_1)$ iff $ISem(mt_2)$ is a subcategory of $ISem(mt_1)$

PROOF

Without loss of generality, we may assume that mt_1 and mt_2, are not isomorphic. But then, the structure of TG_2 induces the validity of both statements. □

Now we are in the position that we can define the semantics of the application of a method on an *is*-object. Again, this is analogous to the semantics of the application of a method to a structured object. So, we get:
Let *rmt* be the root-type of the method type *mt*. *ISem*(*rmt*) is defined as a subobject of X_{rmt}. This object X_{rmt} is of the form:

$$X_{rmt} = ISem(s_1^{t_1}) \times \cdots \times ISem(s_n^{t_n}).$$

The *is*-type *smt* is the *is*-type which has *ISem*(*rmt*) as its root type, the inclusion arrow i: $rmt \rightarrow X_{rmt}$ and for the rest, it consists of the structure type induced by X_{rmt}

DEFINITION 9.12

Let *mt* be a method type, with *rmt* as its root-type, its associated *is*-type *smt* is as defined above.

Now, each method of type *mt* can be be seen as a structured object of type *smt*. To make this more precise, we have to make the distinction between the two typing schemes clear. We will write m; *mt* to denote that m is a method of method type *mt* and m: *smt* to denote that a structured object of type *smt*. Then we have:

LEMMA 9.13

Let m; *mt*, then m induces as structured object *sm*: *smt* and vice versa □

And finally:

DEFINITION 9.14

Let m be a method of type *mt* with root es-type *rmt*, and o an object of type t, then the expression:

$$eval(m, o)$$

is well typed if there is a *is*-type s_r, such that *smr* is a subtype of r^t. Moreover, if f: $ISem(rmt) \rightarrow r^t$ establishes this subtype relationship, then the semantics of the expression are given by:

$$ISem(eval(m, o)) = eval(f(m), o)$$

Now that we have defined the application of methods on *is*-objects, we can redefine SOQL. This is the subject of the next section

9.4 ISOQL

In the previous section, we have defined *is*-objects as well as the application of methods on such objects. Obviously, the next step is the adaptation of the query language SOQL to a language ISOQL that supports queries on extensions consisting of *is*-objects.

For the topic of the second subsection, recall that we made the assumption that elements of an exponential type are constructed explicitly. Now *is*-objects are specified using elements of exponential types. Hence, to confirm with our assumption, we have to device a method to define *is*-objects. In the second subsection, we show how a small adaptation to ISOQL may serve this purpose.

9.4.1 "The query language"

The query language SOQL we defined earlier for structured objects can, in principle, also be used for *is*-objects. Because *is*-objects are semantically arrows $1 \to X$ in some topos, which is also true for structured objects. In fact, this was the key to provide semantics for SOQL. However, SOQL does not use the extra information on *is*-objects that we have, viz. that *is*-objects can be seen as arrows $D \to \Omega$ for some D. Therefore, we extend SOQL to the new query language ISOQL, which allows the user to use the extra information.

In SOQL, the basic formulae can be used to built formulae using the logical connectors, because each basic formula is an arrow $D \to \Omega$ for some D, and the logical connectors are defined on Ω. Now, *is*-objects have $D \to \Omega$ semantics themselves, this means that we can apply the connectors directly to the objects themselves! In principle, there are two ways to incorporate this observation into the query language:

1) In the first approach, a calculus of *is*-objects is defined using the logical operators, and following the basic formulae are built using the calculus operators.

2) In the second approach, the set of typed expressions is greatly extended, by logical connections of typed expressions again as typed expressions

Both approaches are in principle equivalent, they only differ syntactically. The advantage of the first approach is that it yields a friendlier language, in that it makes a syntactic difference between applying the logical connectors to *is*-objects in the calculus and applying them to connect formulae. The advantage of the second approach is that it shows that there is no semantic difference between the two applications. Moreover, it forms a smaller deviation from the original SOQL definition. We feel that the advantages of the second approach are more important than the advantage of the first. Therefore, we choose the second option.

The difference between the new query language ISOQL and SOQL are sufficiently small, that we can define the well-typed version directly. In fact, the

only difference between ISOQL and SOQL lies in the collection of typed expressions. Hence, we only redefine this concept

DEFINITION 9.15
The set *ITexp* of typed ISOQL expressions is defined as follows:

1) Constants and variables are typed expressions, with their type as defined above.

2) If $x: s \in Exp$, then $x: s \in ITExp$, moreover, the type of this expression is s, i.e. by abuse of notation $(x: s): s$.

3) Let $f: t_1 \rightarrow t_2$ in TG be and let t_1 and t_2 be root types of respectively structure types $s_{[t_1]}$ and $s_{[t_2]}$ respectively. If $e: s_{[t_1]} \in ITExp$, then $f(e): s_{[t_2]} \in ITExp$

4) Let $f: t_1 \rightarrow t_2$ in TG. If $e:: t_1 \in ITExp$, then $f(e):: t_2 \in ITExp$.

5) Let $f_1: t_1 \rightarrow t_2, \ldots, f_n: t_{2n-1} \rightarrow t_{2n}$ in TG and let $t_1, t_3, \ldots, t_{2n-1}$ and $t_2 \times t_4 \times \cdots \times t_{2n}$, be the root entity types of the structure types $s_1, s_3, \ldots, s_{2n-1}$ and s_{even} respectively.
If $e_1: s_1, e_3: s_3, \ldots, e_{2n-1}: s_{2n-1} \in ITExp$, then
$f_1 \times \cdots \times f_n(e_1,..,e_{2n-1}): s_{even} \in ITExp$.

6) Let $f_1: t_1 \rightarrow t_2, \ldots, f_n: t_{2n-1} \rightarrow t_{2n}$ in TG. If $e_1:: t_1, e_3:: t_3, \ldots, e_{2n-1}:: t_{2n-1} \in ITExp$, then
$f_1 \times \cdots \times f_n(e_1,..,e_{2n-1}):: t_2 \times t_4 \times \cdots \times t_{2n}$.

7) Let $e_1:: t_1^{t_2} \in ITExp$, $e_2: s_{[t_1]} \in ITExp$, and $e_3: s_{[t_2]} \in ITExp$ then: $eval(e_1, e_3): s_2 \in ITExp$

8) Let $e_1: s_{P(t)}$, $e_2: s_{P(t)} \in ITExp$ and

   a) $(e_1 \cap e_2): s_{P(t)} \in ITExp$

   b) $(e_1 \cup e_2): s_{P(t)} \in ITExp$

   c) $(\overline{e_1}): s_{P(t)} \in ITExp$

9) Let $e_1: s_t, \ldots, e_n: s_t \in ITExp$, then $\{e_1, \ldots, e_n\}: s_{P(t)} \in ITExp$.

10) Let $e_1:: t_1^{t_2} \in ITExp$, $e_2:: t_1 \in ITExp$, and $e_3:: t_2 \in ITExp$ then: $eval(e_1, e_3):: t_2 \in ITExp$

11) Let $e_1:: P(t)$, $e_2:: P(t) \in ITExp$ and

   a) $(e_1 \cap e_2):: P(t) \in ITExp$

   b) $(e_1 \cup e_2):: P(t) \in ITExp$

   c) $(\overline{e_1}):: P(t) \in ITExp$

12) Let $e_1 :: t, \ldots, e_n :: t \in ITExp$, then $\{e_1, \ldots, e_n\} :: P(t) \in ITExp$.

13) Let $e_1 :: t$ and $e_2 :: t \in ITExp$, moreover, let $t \in BT$ (i.e. t is a basic type) or $t = t_1 + t_2$ with both $t_1 \in BT$ and $t_2 \in BT$, then:

a) $\neg e_1 :: t \in ITExp$

b) $e_1 \wedge e_2 \in ITExp$

c) $e_1 \vee e_2 \in ITExp$

d) $e_1 \Rightarrow e_2 \in ITExp$.

14) Let $e :: t \in ITExp$ and $t = t_1 \times t_2$ with both $t_1 \in BT$ and $t_2 \in BT$, then:

a) $\wedge e \in ITExp$

b) $\vee e \in ITExp$

c) $\Rightarrow e\ e \in ITExp$

15) Let $e : \in ITExp$ and $t = P(t_1)$, with $t_1 \in BT$, then:

a) $\forall e \in ITExp$

b) $\exists e \in ITExp$

Note that the last three items of this definition should be read as an inductive definition.

The rest of the definitions of well typed ISOQL programs is the same as the definition of well-typed SOQL programs, with the exception that in all the definitions, each occurrence of *TExp* should be replaced by *ITExp*.

The following step in the definition of SOQL queries, was the definition of stratification. Now, one might expect that the possibility of negating a typed expression might lead to further complications with regard to stratification. However, this is not the case for the simple reason that the following two formulae are not equivalent:

$$x \in Y \wedge \phi(\neg x)$$

$$x \notin Y \wedge \phi(x)$$

In the first formula, the x's and thus the $\neg x$'s are bound in their variance by Y, while in the second formula, x ranges over the complement of Y. So, while the second formula would unable us to give semantics to a recursive query, the first will not. Hence, the definition of stratified is identical to the definition of stratified SOQL programs. This means that the definition of ISOQL queries is identical to the definition of SOQL queries.

Given that the syntax of ISOQL queries is almost identical to the syntax of SOQL queries, the reader will not be surprised that the semantics of this new query language is also almost identical to the semantics of the old language. In fact, the only difference is that we have to give semantics to our new expressions. Hence, we only have to consider expressions that are built using rule 13) above. However, these semantics are straightforward, e.g. if $e_1 :: t$ and $e_2 :: t$, this means that the semantics of e_1 and e_2 are the name of arrows $f, g: D_t \to \Omega$. Then, the semantics of $e_1 \wedge e_2$ is given by the name of the arrow: $\wedge \circ (f, g): D_t \to \Omega$.

Note that only in providing semantics to *ITExp*-expressions, we use the extra information that *is*-objects can be seen as arrows from some domain to Ω. For the rest of the semantics, this extra information is irrelevant.

9.4.2 Defining *is*-objects

In this thesis, we made the assumption that elements of an exponential type are constructed explicitly in the topos. We did not define a separate definition language, as defining a realistic definition language for such 'functions' would surely involve the natural number object and its associated recursive functions, as sketched in Chapter 6. Hence, we would have had to give an introduction to this topic as well. For reasons of both time and space, this was considered outside the scope of this thesis (see also the 'topics for future research' in the last chapter).

Now, *is*-objects are defined using exponential types. So, for the *is*-objects the same assumption holds. The easy way out would be to assume that the constants of the basic types are given as elements of $C^{\to}$ rather than as elements of C-objects. However, we feel that this would be cheating. Moreover,it is not necessary to add the full power of recursive functions to define *is*-objects. In fact, the definition language for *is*-objects can be relatively simple. One may want to 'insert' some disjunctive or negative information directly, possibly connected with data already in the database. This means that the ISOQL query language as defined in the previous subsection is in principle strong enough to act as a *is*-object definition language. The two major differences between ISOQL as a query language and ISOQL as an *is*-definition language are:

1) LVar-variables should be allowed to range over domains as well as extensions.

2) The semantics are slightly different, for queries we are interested in the C object specified by: $\{x \mid \phi_x\}$, while for *is*-objects we are interested in the $C^{\to}$ object specified by ϕ. Moreover, the result should be interpreted as one object rather than a collection of objects.

The reason for the first item is clearly that it may happen that some of the values needed in the specification of an *is*-object are not yet present in the

database. Clearly, this extra freedom for LVar variables has some impact on the definition of stratification. The adaptation is, however, obvious, recursion over rules containing LVar variables should not be allowed.

A second remark we have to make, is on a 'boot-strapping' problem: to start the *is*-object creation, as described above, we need a mechanism to construct some basic constants first. The straightforward solution is of course to extend ISOQL with some simple SOQL queries like:

$$x :: t \mid f(x) = 37 \vee g(x) = 9283746$$

So, SOQL queries in which no extensions are used. Note that we would only have to support such queries for the basic types as the constants of the other types can be built inductively from these domains.

We assume that from this sketch, the *is*-object definition language is obvious to the reader. In other words, rather than giving a formal definition, which would again be very similar to the definition of ISOQL, we supply the informal discussion above as an informal definition.

9.5 Conclusions

In this chapter, we have introduced a functional representation of incomplete information. This functional representation is close to the representation of methods. So, one might expect that incompleteness causes similar problems as methods did. However, we show that a careful definition of *is*-types ensures that such problems do not appear.

After the definition of incomplete information for structure types, we describe the effects incomplete information has on methods and there application as well as the effect on the query language.

Chapter 10

Complex Objects

In this chapter, we will merge the various threads of this thesis, such as dependencies, type equivalences, methods, objects, and queries. The result of this merging is the definition of *classes* and, finally, *complex objects*. Before we can begin this merging process, we first need to introduce the concept of an *object identity*; one of the main characteristics of object oriented database systems (Atkinson *et al.* (1989)).

Usually, object identity is seen as a surrogate, i.e. a machine defined number, that is immutable to changes. In the first section, we give another definition of object identity, based on the history of an object. The result is that object identity is defined as a set of expressions over the methods defined with the database. So, for the definition of methods, we assumed that the set of types was fixed, now we assume that the set of methods is fixed. Special attention is given to the question of how methods should be applied to such identities. The result of this analysis is yet again, that the method definition should be extended such that it also encodes this information. Moreover, we show how this definition of identity supports in a natural way the modelling of object sharing and *HAS-A*-relationships in general. We end this section, by showing how identities can be used to incorporate rules in the database.

In the second subsection, we define constraints on object identities, by defining grammars the expression in an identity should satisfy. Moreover, we show how such constraints can be used to model both structural and temporal data dependencies in the database.

In the fourth section, we extend the definitions of the third section to classes and complex objects. The construction of classes from identity types is similar to the construction of structure types from entity types. This implies that again type equivalence plays a major role in this definition. The grammars used to restrict the possible identities of a type, play a crucial role in these equivalences.

Different from entity types, it is easy to show that the question whether two (restricted) entity types are equivalent is undecidable. This means that we can only offer some examples.

The examples are based on a grammatical formulation of the functional dependency. This is achieved by observing that a functional dependency can be seen as a dynamical dependency. Note that this result together with the results of Chapter 4 mean that we can finally give a natural translation for relation schemas with sets of functional dependencies and conflict free sets of multivalued dependencies. The fact that we can finally model multivalued dependencies induce the examples: we simply use the inference axioms for multivalued dependencies.

The lack of a formal semantics for classes means that we cannot give formal semantics to their query language. So rather than defining a new query language in the last section of this chapter, we simply illustrate four aspects of such a query language, viz. value oriented queries, HAS-A relations, grammars and historical data.

10.1 Object Identity

According to (Atkinson *et al.* (1989)) object identity belongs to the mandatory features a DBMS should satisfy to be called an OODBMS. In Chapter 1, we have seen that the concept of an object identity captures the fact that although the representation of a real world entity in the database may change, the entity itself (Das Ding an sich) does not change. Moreover, the identity of an object distinguishes it from all other objects that were, are, or might become, part of the database. Rather than depicting a set of attributes as the unique identifier of an object, such as the key concept in the relational model, the identity is seen as a property that is distinct from all the other properties an object might have. Moreover, this identity is immutable for any action on the database. Note that this implies that two objects may have the same value but a different identity.

In (Atkinson *et al.* (1989)), object sharing and object updates are given as two important consequences of the notion of an object identity. The former means that two objects can share a common component. For example, two parents can share a child, the identity of the child ensures that the two parents have the *same* child. The latter is a consequence of the former, suppose parents A and B share child C, then an update in the value of C through the link with A should also be reflected if C is visited through B. Note that these advantages are the advantages of a *pointer*. Another advantage of object identities is given in (Abiteboul and Kanellakis (1989)), in which a powerful query language is defined in which the object identity is a primitive notion.

Often, see e.g. (Khoshafian and Copeland (1986); Lecluse and Richard (1989); Zaniolo (1989)), object identity is seen as some unique machine generated number, i.e. a surrogate, which is in general inaccessible for and immutable by the user. This implementation of identity is accordance with the pointer

analogy made above, the surrogate can be seen as an abstraction of the physical address of the object. An elegant extension of this schema has been proposed in (Wieringa and van de Riet (1988); Wieringa (1990)); it can be sketched as follows.
Suppose we want to consider a set of objects as a new object, rather than giving this a new number as identity, give it the set of identities as an identity.
This idea not only applies to the set constructor, but also to the other type constructions. The effect is that the structure of the identity reflects the structure an object has.

A drawback of the surrogate approach to object identity is that the identity itself is incomprehensible to a user, if it is accessible at all. This means that its representation of 'das Ding an sich' may be true from an implementations point of view, but has no significance for the user. Now, one might argue that this drawback is insignificant, because the 'true' identity is inaccessible. However, in this section, we show a different approach to identities that is accessible and comprehendable. The rest of this chapter shows advantages of this type of identity that are not shared by the surrogate approach.

To introduce our definition of object identity, consider a real world entity e. Presumably, there is a point in time, at which it is decided that e should be represented in the database. Probably, there is some method *new* that yields an object o of some type t, such that o is the representation of e. Graphically, we have:

$$1 \xrightarrow{o} t_1$$

If, after some time, e undergoes a change, o has to be changed so that we get again a faithful representation of e. In an OODBMS, there is only one way to change an object, viz. by applying a method. Hence, some method m_1 is applied to o. From our discussion on methods, we know that (blurring the distinction between the name of an arrow and the arrow itself) $eval(m_1, o) = m_1 \circ o$, which can graphically be seen as:

$$1 \xrightarrow{o} t_1 \xrightarrow{m_1} t_2$$

This happens every time a change in e is recorded, so after some time, e is represented by an object that can be pictured as follows:

$$1 \xrightarrow{o} t_1 \xrightarrow{m_1} t_2 \longrightarrow \cdots \longrightarrow t_n \xrightarrow{m_n} t_{n+1}$$

Note that for the moment we assume that no other objects are necessary to apply a method.

The arrow $m_n \circ \cdots \circ m_1 \circ o$ in the picture above has the curious property that it encodes all the information we have ever recorded about e. Because, o, $m_1 \circ o, \ldots, m_n \circ \cdots \circ m_1 \circ o$ are all the different representations we have ever used.

So, the expression $m_n(\cdots(m_1(new(param)))\cdots)$, in which *param* are the parameters given to *new* at its invocation, is a concise description of all the different representations of e in the database. The idea is now that for a human, at least for me, the identity of some person is a mix of current and previous facts of that person. Moreover, such an identity can be transmitted to other persons:
Q: Who is that guy? (pointing to X)
A: That is such and so (passing some current information on X)
Q ?
A : Who is/was ... (passing some more current or historic information on X)
After some time, this discussion will end in one of two possible states:

1) Either Q has been able to match the information she got with the information she already has on some person X. In this case, she recognises X as being Y.

2) Or, no such match occurs, which means that she does not recognise X; however at this point she knows a new person X and has some information on X, which can be used in future situations.

Of course, I am aware that a third state is possible, if one of the two persons has incomplete information on X. In such a case, a match may seem impossible due to the incompleteness of the description. However, we are interested in the ideal situation meaning that the database 'knows' when its information is incomplete and that it can convey the incompleteness.

The upshot of this little discourse is that the identity can be seen as encoded in the expression given above. This observation inspires the idea of using such expressions as the identity of an object. In other words, to use the life-cycle of an object as its identity. There are two major objections to such an approach:

1) Such expressions do not have to be unique.

2) The expressions change all the time.

We discuss both objections below.

Clearly, such expressions do not have to be unique. It is very well possible that two objects have the same life-cycle up to a point. In fact, it is even possible that two objects have completely the same life-cycle. However, if we invoke time, i.e. not only record the methods applied but also the time at which they are implied, this becomes extremely unlikely. Moreover, it means that either some information is held from the representation or even someone in the real world cannot distinguish the two objects. So, besides e.g. for nails that are manufactured at the same moment our 'expression approach' towards identity can be made reasonable unique. Moreover, in the case of the nails, a 'set-identity' will probably be good enough.

For the second objection, note that when a method m is applied to an object o, it is also applied (symbolically) to its identity, i.e. the identity i changes to $m(i)$.

But then the old identity i is still a subexpression of the new identity $m(i)$. Now, both i and $m(i)$ are both unique, by the above schema, moreover, the correspondence between i and $m(i)$ is also unique. In other words, this objection merely implies that we should consider the expression i with all its subexpressions as the identity. But, as the correspondence between subexpressions and expressions is unique, we might as well record the most recent expression.

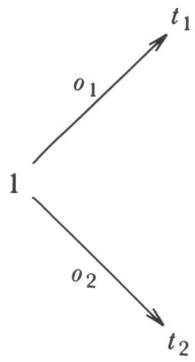
So, both objections can be countered and do not really attack our intuition. We will not try to formalise time in this thesis, in fact, we will simply use expressions without time stamps for identities. We are satisfied with the observation that such time stamps can be used to disambiguate the expressions. However, we want to make a small description how time could be modelled. The obvious way is of course to assume a clock, and use the clock time for the time stamps. However, there is also a way that is more in line with our line of thought, which can be sketched as follows.

Although time is probably the most difficult concept to define, it is safe to say that time is the vehicle used to order the events we perceive. For instance, the counting of days can be seen as the recording of subsequent sunrises. So, suppose we have a special object g in the database that records all events it perceives in the database; moreover, suppose it perceives every events. For simplicity, assume that an event, i.e. the application of a method, takes zero time. Then, g perceives two events as either simultaneous or in an order. If it uses a special symbol, say $|$, to record that two events happen simultaneous, g can linearise the events to a string. Whenever an event happens, the current time string is used as the time stamp.

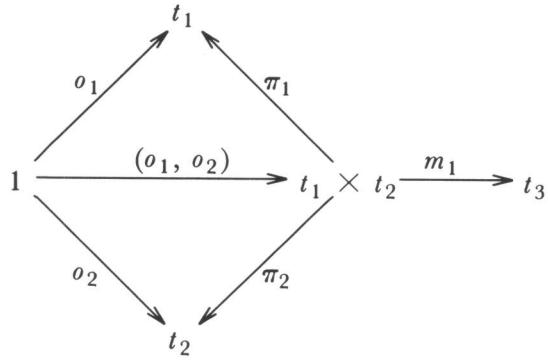
As said above, we will not use the time stamps, however, we hope that the above sketch convinces the reader that time could be modelled using our schema.

Now that we have dismissed the two objections listed above, we have one difficult step left to be made: many methods, will not be applied to single objects, but to e.g. pairs of objects, or sets of objects. For example, promoting an employee to manager of a department, probably uses both the employee object and the department object. So, we have to solve the problem of methods applied to more than one object at the same time. The solution, or better the generalisation of the sketch above, is in some ways similar to the identity-giving schema of (Wieringa and van de Riet (1988); Wieringa (1990)) as sketched above. Moreover, at the same time it illustrates our view on object sharing or more general *HAS-A*-relationships.

To illustrate our approach, suppose we have an object o_1 of type t_1 and an object o_2 of type t_2. So, graphically we have:



Suppose we want to apply a method m_1 to the pair (o_1, o_2), then we get the following picture:



The question after the application of m_1 is: what are the objects and what is their identity. In principle, there are three choices:

1) After the application, there is one object, with identity $m_1(o_1, o_2)$.

2) After the application, there are three objects, viz. the 'old' o_1 and o_2 as well as the new object $m_1(o_1, o_2)$.

3) After the application, there are still two objects, viz. the 'old' o_1 and o_3.

Each of these options has its own merits. In fact in the following discussion of the three options we will see that for each there are situations in which it is the natural choice, but that there are also situations in which it is an awkward choice.

The first option, replaces the two objects, with identity Id_1 and Id_2, by one new object, with identity say $Id_3 = m_1(Id_1, Id_2)$. Note that this 'merging' of identities does not cause loss of any information, because Id_1 and Id_2 can be reconstructed from Id_3. So, this option is consistent with our 'invariant' that identities encode the complete history. The effect of this option is that the 'old' objects cease to exist as distinguished objects, they are amalgamated into one

new object. Examples in which this option is natural (or preferable over the others) can be found e.g. in manufacturing processes. If a *car* is assembled from its components, none of its components can be manipulated independently, except if the car is first, partially, disassembled. Hence, in such a case, the 'old' identities cease to exist from a practical point of view. An even tighter coupling can be found, if o_1 is coffee, o_2 is milk and m_1 represents the process of stirring the milk through the coffee. After stirring, the coffee and the milk can no longer be distinguished; in fact, from a practical point of view the merge cannot be undone.

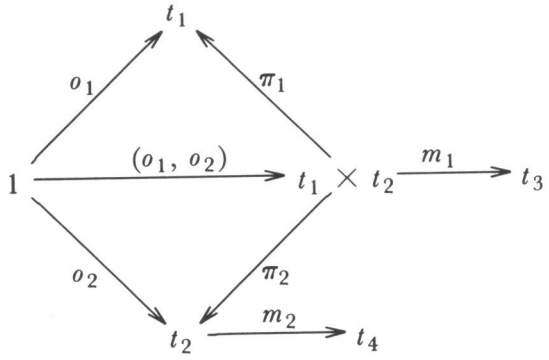
As the general solution, the first option has some drawbacks. To indicate these drawbacks, let method m_1 assign person o_2 telephone o_1, the under the first option, we have that

1) The person, telephone pair gets one identity, i.e. the objects o_1 and o_2 would not be distinguishable.

2) In the line of 1) above, probably many objects are connected with each other throughout there lifetime. The effect of the solution would be that there is only one identity for each of the partitions of the transitive closure of the connection set.

In this particular example, the objects *person* and *telephone* are naturally perceived as distinguished objects in the UoD, both before and after the method has been applied. This means that the object identity can no longer be used to distinguish between various entities in the UoD! So, in the example given above, we should turn our attention to the other two options.

The second option above, is stated rather vague, it only states that there are three objects, viz. the 'old' o_1 and o_2 as well as the new object $m_1(o_1, o_2)$. It does not mention what the identities of these objects should be. It is tempting to say that o_1 and o_2 keep their old identity, while the new object gets the identity $m_1(o_1, o_2)$. However, this means that no longer all information on the object is encoded in its identity. To illustrate the problems this might cause, suppose that we want to make person o_2 one year older using method m_2, then we get graphically something like:



Now, the information that o_2 is aged (changed into $m_2(o_2)$) is not recorded in $m_1(o_1, o_2)$. Hence, the information in $m_1(o_1, o_2)$ is no longer up to date. In other words, only the combination of $m_2(o_2)$ and $m_1(o_1, o_2)$ represents all the information we have on the real world person.

One might argue that this is still no real problem, therefore, suppose that m_2 represents the death of a person rather than his aging. In this case, one clearly wants a change in the information encoded by $m_1(o_1, o_2)$; if only because a deceased can have no property.

The problems sketched above, disappear if we don't retain that an identity should represent all the information on a real world entity. In other words, the identity of an object is defined as a *set* of expressions rather than a single expression. So, in our example, we have to decide what set of expressions will form the new identity of o_2. After the discussion above, we have two choices left:

$$\{o_2, m_1(o_1, o_2)\}$$

$$\{o_1, o_2, m_1(o_1, o_2)\}$$

The first 'identity', encodes all information on person o_2, but does not encode explicitly which telephone o_2 has. The second 'identity' however, encodes this information also. It might seem that this makes the second identity preferable. However, note that it would imply that the person and the telephone have the same identity! Therefore, we have to chose the first set as the new identity of o_2.

So, we get the following identities for the *person* P and the *telephone* T:

$$ident_P = \{o_2, m_1(o_1, o_2)\}$$

$$ident_T = \{o_1, m_1(o_1, o_2)\}$$

The following step is to decide what identity should be given to the third object, which we denoted above by $m_1(o_1, o_2)$, which encodes the relation between the *person* and the *telephone*. Clearly, its identity should at least contain the expression $m_1(o_1, o_2)$. But, should it also contain other expressions? Clearly, we have to choose between the identities:

$$\{m_1(o_1, o_2)\}$$

$$\{o_1, o_2, m_1(o_1, o_2)\}$$

The former 'identity' only encodes the relationship, while the latter encodes the relationship as well as its participants. To make this choice, we should consider the 'dynamical' effects of both options.

Given that the identity of an object is a set of expressions, it is obvious that the application of a method to an object implies that the method should be applied to all expressions in the identity. So, if we choose the second identity, an update on the relationship between the P and T, might change P and T beyond a registration of the mutation of the relationship. But this means that P and T are subobjects of PT. In other words, we might as well have chosen option 1) and use subtyping. Therefore, we choose the first identity. This has the effect that

changes in P or T can only be gotten through P or T themselves. The new object, say PT can only be used to change the relationship itself. On the other hand, changes in either P or T are directly reflected in PT, which is in accordance with our observation that if P 'dies' all its 'relations' should be modified. Summing up the identities under this option, we get:

$$ident_P = \{o_2, m_1(o_1, o_2)\}$$
$$ident_T = \{o_1, m_1(o_1, o_2)\}$$
$$ident_{PT} = \{m_1(o_1, o_2)\}$$

The fact that we can only update PT if this update is independent from both P and T, i.e. there is no explicit effect on P and T (but there is of course, an implicit effect), is argued above. This possibility is not very useful if there is only one *person* with a *telephone*. However, if we have a collection of such 'relations', it can e.g. be used to switch telephones. This collection of *person-telephone*-objects brings us to the third option listed above; i.e. a method applied to two (or more) objects does not create a new object.

Given such a collection of *person-telephone*-objects, it is reasonable to assume a *telephone-book*-object, i.e. an object that represents a set of *person-telephone*-objects. In the line of the example above, let o_1 be such a *person-telephone*-object, o_2 a *telephone-book*-object and let m_1 be the method that enters o_1 into o_2. The the expression $m_1(o_1, o_2)$ represents the membership relation of o_1 and o_2. Note that the application of this method has an explicit effect on o_2: o_2 has to be changed such that o_1 really is a member of o_2. In other words, with m_1 has an explicit effect on o_2 we mean that the object o_2 'an sich' is changed.

The explicit effect of m_1 on o_2 implies that we cannot update $m_1(o_1, o_2)$ independently from o_2; because a change in the relationship between o_1 and o_2 established by m_1 changes o_2 also. Hence, if we want $m_1(o_1, o_2)$ as a distinguished object, o_2 has to be a subobject of this object! Similarly, o_1 has to be a subobject, because a change in o_1 has to be reflected in o_2, and consequently in $m_1(o_1, o_2)$. So, $m_1(o_1, o_2)$ can only be seen as a distinguished object if the objects o_1 and o_2 cease to exist (note that this is the first option). Hence, if we want to preserve the old objects, we cannot create a new object. In other words, after the application of m_1, we have the two identities:

$$\{o_1, m_1(o_1, o_2)\}$$
$$\{o_2, m_1(o_1, o_2)\}$$

Note that the relationship between o_1 and o_2 is still recorded, although not with a separate identity, as $m_1(o_1, o_2)$ can be found in both identities.

The above discussion of the three options with their relative merits are summarised in the following table, for which we assume objects o_1 and o_2 and the method m:

new identities	effect
$m(o_1, o_2)$	old objects can only be manipulated through their relationship
$m(o_1, o_2)$ o_1 o_2	old objects can be manipulated independently. Direct manipulation of their relationship may not affect the old objects.
o_1 o_2	the relationship can only be manipulated through its constituents

Clearly, which of the three options the effect of a method should follow has to be made explicit. However, before we discuss the application of a method to objects, we first give two 'invariants' method application should respect and illustrate the ratio behind these invariants:

1) As we have seen above, the relationship between o_1 and o_2 in the third entry of the table can be detected by the non-empty intersection of the identities. This will turn out to hold in general, i.e. two objects have a relationship iff their identities have a nonempty intersection.

2) Another observation that can be made after the discussion above is that all expressions in an identity have a common subexpression. This will also hold in general.

These invariants prove to be the key to a question that is probably nagging the reader for the last few paragraphs: identities are sets of expressions, moreover, these identities can have a non-empty intersection, doesn't that imply that we have to have replicas of expressions scattered around in the database? Moreover, doesn't that imply that whenever, we 'update' such a expression we have to hunt the database for all the copies to ensure consistency? Clearly, if these implications are true, our approach to object identities would be a dead-end, as this would mean that we would need some way to indicate that some expressions have to be equal.

The implications above, have the underlying assumption that we keep track of identities as actual sets of expressions. However, we do not have to do this. We only have to keep one set in which all expressions can be found. Because, two expressions contain information about the same real world entity if they have a common subexpression. This means that we can always reconstruct the identity of an object as a 'maximal' (the precise meaning of maximal will be given later in this chapter) set of objects such that each pair of objects in the set has a common subexpression. In other words, from a formal point of view, we do not need to have replicas of expressions.

Note that this implies that we no longer assume that a real world entity has one unique 'highest' representation, from which all other representations can be derived as we assumed in the previous chapters. The reason for this extension is obviously that in a system without identity, there is no way to encode that

unconnected objects are representations of the same real world entity. While in a formalism with identity, the identity may serve as the encoding that 'unconnected' objects are representations of the same real world entity.

The relationship encoded (either explicitly or implicitly) by the fact that two identities have a non-empty intersection is sometimes referred to as an *HAS-A*-relationship, in contrast with the more familiar ISA-relationships. In the example above, the fact that $ident_P$ and $ident_T$ have a non-empty intersection, encodes that person $ident_P$ *HAS* telephone $ident_T$, and symmetrically, telephone $ident_T$ *HAS* person $ident_P$ (as owner). As a generalisation of this example, we say that two objects with a non-empty intersection are in a *HAS-A*-relationship.

As an aside, note that it is only a small step from HAS-A relationships to object sharing. In fact, one could argue that HAS-A-relationships already imply object sharing. However, we would like to restrict the notion of object sharing to a smaller class of relationships. Suppose that two objects both of whose identity has a non-empty intersection with the identity of a third object, i.e. $ident_1 \cap ident_3 \neq \emptyset$, and $ident_2 \cap ident_3 \neq \emptyset$. Then we say that the objects $ident_1$ and $ident_2$ *share* the object $ident_3$.
Note that this notion is not symmetrical, i.e. it is very well possible that $ident_1$ and $ident_2$ share $ident_3$ but that the intersection of $ident_1$ and $ident_3$ is empty!

Now that we have illustrated the merits of the invariants that method application should respect, we can illustrate method application. From the discussion above, we have that:

1) it should be encoded which of the three options listed in the table above the evaluation should follow; and
2) regardless of the chosen option, the evaluation should respect the two invariants.

For both problems, we should keep in mind that the methods are applied symbolically to sets of expressions. So, if we analyse our running example, we have the identity sets Id_1, Id_2 and the method m. Under the first option, we have:

$$m(Id_1, Id_2) = \{m(Id_1, Id_2)\}$$

The second and third option give:

$$m(Id_1, Id_2) = \{Id_1, Id_2, m(Id_1, Id_2)\}$$

The result set of expressions is interpreted as union of the following identity sets under the second option:

$$\{Id_1, m(Id_1, Id_2)\}, \{Id_2, m(Id_1, Id_2)\}, \{m(Id_1, Id_2)\}$$

and as the union of the following identity sets under the third option:

$$\{Id_1, m(Id_1, Id_2)\}, \{Id_2, m(Id_1, Id_2)\}$$

Now, the choice whether or not $m(Id_1, Id_2)$ represents a new object, is simply a

typing choice: if objects of the type of $m(Id_1, Id_2)$ are defined, then it represents a new object, otherwise it doesn't.

Hence, the option the application of a method should follow is easily encoded in the method itself, by specifying the result of the method on identities as sets of expressions. Finally, note that in this example the invariants are respected.

As an aside, note that under the first option, objects disappear because their identity is removed from the set of expressions. In general, one probably does not want to remove expressions from the database, but rather move the object to some terminal type. Under such an option, we get something like:

$$m(Id_1, Id_2) = \{m(Id_1, Id_2), m_1(Id_1), m_2(Id_2)\}$$

in which m_1 and m_2 are 'submethods' of m

The example of the application of a method on two identities is generalised straightforwardly to more complicated examples. The lesson from the example above is that the method should be defined by specifying its result on identities (this corresponds with the three options in our example in this section). Moreover, as long as method application can at most 'enlarge' an expression (i.e. $m(\{x\}) = \{m(x)\}$) the invariants are obviously kept.

We do not attempt to formalise our notion of object identity in this section, that is postponed to a later section. Rather, we conclude this informal discussion on object identities by illustrating three of the effects of our definition. For the first effect, we turn our attention towards queries. Already in (Khoshafian and Copeland (1986)), it is suggested that object identity is a useful property for historical databases, or more general a database that maintains several versions of the same object. In such a database, the identity links together the historical objects or the different versions of the same object. For us, the object identity itself encodes both the history and the different versions of the same object. In fact, it even combines the seemingly different problems in that it even encodes when the versions were originated. For queries the upshot is that the identity can be used to ask for objects with certain properties in past 'incarnations' or indeed in concurrent incarnations.

The last remark may seem puzzling: how can one object have two concurrent incarnations. For, this seems to mean that two different objects are seen as different from one point of view, but as the same from another. This really seems to violate the very notion of an identity. However, we shall see later that identity is in the eye of the beholder. In this context, this proverb means that indeed two objects may seem different from one point of view, while they seem identical from another. To illustrate our intuition on this subject, suppose that we have two observers A and B. Moreover, suppose that A makes a distinction between objects O_1 and O_2, while B does not make this distinction. This means that A assigns different identities two O_1 and O_2, while B has only one identity. In other words, A knows identities $ident_{A_1}$ and $ident_{A_2}$, while B knows $ident_B$. This can be consistent if $ident_B$ consist of subexpressions common to $ident_{A_1}$ and $ident_{A_2}$. This means that B looks at O_1 and O_2 from a historical perspective and

identifies O_1 and O_2 as distinguished concurrent versions of the same object.

Identities as sketched in this section, can not be used to represent the abstract fact that O_1 and O_2 are versions of the same object other than by the historical perspective sketched above. One could argue that an object O_3 concurrent with O_1 and O_2 that identifies the two as essentially equal is a preferable solution for versioning. To achieve this, we need a type for which an equivalence relation is defined that identifies the different identities. Such a construction is, however, outside the scope of this thesis.

The second effect is in a sense the opposite of our discussion above. In (Khoshafian and Copeland (1986)) the following problem with respect to object identity is given: suppose, we have two different objects in our database and at a certain point it is recognised that these two objects are really the same object. What should happen with the object identity? So, whereas versions have the same identity from an historical object, we now have two objects that should have had the same identity in retrospect. In (Codd (1979)), Codd argues for a 'coalescing' operator, which can be used to merge the identities. In (Khoshafian and Copeland (1986)), Khoshafian and Copeland introduce a merge operation to merge the identity of two similarly structured objects. They note that the semantics of this operation could be both tricky and expensive, and propose a simple approach which requires the objects to have the same type and are deep-equal.

We have already seen that methods can be used to merge two identities. Now, if two objects turn out to be the same object, this implies that we have two objects of the same type that are incompletely specified. So, we simply need a method, say *merge*, that merges two objects of the same type into a new object of that type. Clearly, this may result in an inconsistent object if the information encoded in the two objects is not compatible. For example, one object insists that a person is male while the other encodes that the person is female. However, in a sense it is up to the user to resolve these inconsistencies, as the user wants to merge the two objects, so probably knows the correct information. This is easiest enforced by defining *merge* such that it can only be applied if the two objects are relative consistent, i.e. when the resulting object is consistent. This approach simply forces the user to make both representations of the real world entity consistent with the real world *before* the merge is done. Note that this approach forces the relative consistency in the same way as it is enforced that the two objects have the same type.

This sketch is all we have to say on the subject of object merging, as besides the definition of a method *merge*, the problem seems to be an implementation problem rather than a theoretical problem.

The third, and last, effect of our notion of object identity we want to sketch is with regard to storing rules in the database. In Chapter 8 we had have seen that we could store rules in the database, but that they had to be invoked explicitly. Using identities, we can automate this invokation as follows:

First note that the situation that the extension of type t can be derived from the extension of type s implies a relationship between the two extensions. In turn,

this implies that both extensions are objects, because only objects can have relationships. But is both extensions are objects, both objects have an identity. Moreover, if there is a relationship between the extensions, their identities have a non-empty intersection. In other words, the rule is used to 'link' the two identities together.

The way we sketched the application of a method to an identity in this section suggests that if two extensions are linked by a rule, an update of the first immediately causes an update on the second. This would mean that our approach to storing rules cannot be used save storage requirements. However, there is no reason to prohibit lazy application, i.e., the application of the rule on the derived extension is onbly calculated if this extension is actually used. Again, this more an implementation problem than a theoretical problem.

Before we formalise object identities, we first sketch how identities can be used to define constraints on a database. This is the subject of the next section.

10.2 Restrictions on identities

In the previous section, we have introduced object identities as sets of expressions. The expressions were merely a convenient notation for paths through the topos C. The fact that they were represented as expressions in some formal language did not have much significance, outside the observation that through a 'subexpression-relationship' identities could be reconstructed. In this section, we are going to exploit the expression characterisation of identities much further. By restricting the possible identities an object can have, we will be able to model both static, dynamic and temporal constraints. Moreover, the constraints on identities have a considerable influence on the equivalence of types.

The identities are sets of expressions, whenever a method is applied on the object, it is also applied symbolically to the identity. So, if we want to forbid the application of a method to an object, we might as well forbid the symbolic application. Now, this symbolic manipulation is simply a way to create new expressions from existing expressions. Hence, we can prohibit application, if we declare certain expressions illegal. In other words, we can define a grammar on the set of expressions and methods can only be applied if the resulting expression(s) are again grammatical.

Note that this use of a grammar in a database context is completely different from the applications that are given in (Kersten and Siebes (1990)) and in (Gyssens, Paredaens, and van Gucht (1989)). In both references, the grammar is used to describe the structure of the database. In other words, the grammar can be likened to a typing schema. As far as the author is aware, this is the first time that grammars are used to define constraints on a database.

We deliberately used the word constraint in the sentence above, as the grammar constraints the user in the methods she can apply to an object. Moreover, we shall illustrate later in this section that the grammar can be used to model some of the more traditional constraints. But before we look at such applications

of the grammar, we first sketch how such a grammar might look like.

The first observation is that an object identity consists of a set of expression rather than one expression. This suggests that it in contrast with the usual grammars, it is possible that such an expression is not ungrammatical on its own account, but that a method application is declared to be illegal because the resulting set of expressions in its entirety is considered ungrammatical. However, such an approach is beyond the scope of this thesis. In other words, we assume that the application of a method is prohibited because at least one of the resulting expressions is ungrammatical.

This assumption is still not strong enough to bring us into the 'save' realm of the usual formal language theory, because the expressions may be similar to objects in a formal language, but it is not identical. The important difference is that sentences in a formal language are linear, while the expressions do not have to be linear; viz. $m(o_1, o_2)$. Clearly, the solution to this problem is to linearise the expression.

One might be tempted to think that, similar to the different options available for the application of a method to a identity, there should be different options to linearise such an expression. However, this is not necessary as the different ways an expression may be linearised are induced by the different ways in which a method can be applied by an identity. Therefore, the expression $m(o_1, o_2)$ is simply linearised to mo_1o_2.

This means that we stay within the realms of the usual theory of formal languages. The logical question is what kind of restrictions we put on the choice of a grammar. The grammars are introduced as a means to declare the applications of a method to an object legal or illegal for reasons other than typing. In other words, the grammar describes a language, and we want to know whether an expression is in the language or not. It is well-known that the question whether a string belongs to a language is decidable for all classes of formal languages save the recursive enumerable sets, c.f. (Hopcroft and Ullman (1979)). From a database designers point of view, one would also like to be able to ask questions like is $L = \varnothing$ or is $L = \Sigma^*$. These questions are only decidable for deterministic context free languages, c.f. (Hopcroft and Ullman (1979)). However, such questions are outside the scope of this thesis, therefore we do not restrict the class of the grammar further than the language should be a recursive set.

Obviously, the grammar is declared for some type; i.e. the grammar is locally defined. This means that it is very well possible that for one object a sequence of method applications is considered illegal, while it is perfectly legal for another object. For example, if *employee* is a subtype of *person*, all methods applicable to *persons* are also applicable to *employees*. However, certain sequences of methods may be grammatical for *persons* while they are ungrammatical for *employees*. Note that this example illustrates that there is an intimate connection between types and grammars.

The grammars may be defined locally, but the different grammars are of course related, as already indicated above. Continuing our example, the

grammars for *person* and *employee* have to be consistent with each other. With consistent, we mean that the grammar belonging to employees should not allow sequences of methods that are forbidden for persons (assuming that all the methods in the sequence are defined for persons). If the grammars would be inconsistent, this would mean that certain *employees* could not be considered as *persons*, as their identity could never be the identity of a *person*. This means that *employee* is not a specialisation of *person*, but that both types are specialisations of a common supertype.

So, 'consistency' means that constraints are inherited. However, because grammar rules specify the *legal* transformations rather than the *forbidden* transformations, the inheritance is 'upside down'. In other words, if *employee* is a subtype of *person*, the the grammar rules for an *employee* seen as a *person* have to be a subset of the grammar rules defined for *persons*. In general, we cannot say that the grammar rules defined for *employees* have to be a subset of the grammar rules defined for *persons* as not all methods that are defined for *employees* have to be defined for *persons* as well.

This observation implies that the inheritance of grammar rules is rather complicated. Therefore, we make a simplifying assumption:

> If s is a subtype of t, then for each expression of type s, there is an associated expression of type t.

In plain terms this assumption states that if a certain sequence of methods is not allowed for *persons*, than it is also not allowed for *employees*. The precise definition of associated expression is given later in this chapter.

The consequence of this assumption will be that we have to define each grammar rule only once. The inheritance of this rule is 'automatic' using the associated expression.

Now that we have built some intuition on both identities and the function of grammar rules, we can indicate some of the effects of these concepts. The first effect we illustrate is on dependencies, or more precisely on functional dependencies:

If in a relational database we say that A functionally defines B, this means that an A-value will always be accompanied by the same B-value. Now, we assume in our context that a dependency denotes a certain type of relationships between two or more objects, i.e in the case of functional dependencies, relationships between pairs of objects. The functional dependency then simply translates to the constraint that an object A of a certain type s can be related to at most one object B of type s. Now, this is clearly an example of a grammar rule. This means that the grammar rules can be used to express functional dependencies. If we combine this with the results of Chapter 4, we see that this implies that we can model acyclic join dependencies as well.

As an aside note that the justification for considering dependencies between objects rather than values is that a dependency should hold globally in the database. So, the different occurrences of a 'value' are identified, but this exactly

what the notion of a complex object expresses.

Other types of dependencies that can be modelled with the grammar are simple temporal constraints. For example, if the grammar only allows expressions in which a method m_1 can only be applied if the method m_2 is already applied, the grammar models the temporal constraint that an m_1 event should always be preceded by an m_2 event. A detailed treatment of this type of constraints is outside the scope of this thesis, but see the section on future research.

The second effect of grammar rules we briefly discuss in this introduction is the effect on type equivalences. In fact, we only give an example:
It is straightforward that the type $P(A \times B)$ is isomorphic with the type $P(A \times \{B\}_A)$. Now that we can define functional dependencies on types, we can actually use this isomorphism as another example of canonical equivalent type.
As an aside, note that this example shows why we did not introduce a separate grouping operator in our query language. Using this kind of type equivalences, we get the grouping for free.

10.3 Identity types

The first two sections of this chapter have been an extended expose meant to introduce informally our philosophy towards identities and related issues. As in the previous chapters, the rest of this chapter is used to formalise this intuition. However, the formalisation in this chapter is rather sketchy. The reason for this lack of rigour is that we would need lists in our category to give a formal semantics. The machinery needed for the list construction is well beyond the limited introduction to Category Theory given in this thesis. So, a rigorous formalisation would increase the size of this thesis again considerably or it would be incomprehensible for the non-categorist. Therefore, the definitions in this chapter are rather operational and proofs of theorems are only sketched.

Recall, that to define the second order or method types, we needed the structure types. In turn, we now need the set of methods (note the methods themselves, not their types) to define the concepts of this chapter. So, we assume that the set of (typed) methods M is fixed. The following observation is the similarity between structured objects and *is*-objects; both syntactically and semantically (only the topos is different). This similarity is exploited in this chapter, in that its machinery can be used both for structured objects and *is*-objects. In other words, the question of object identity is orthogonal to the question of incomplete information. In this chapter, we will use the terms *structure type* or *is-type* and *structured object* or *is-object*, the reader should parse these terms as *structure type / is type* and *structured object / is-object* respectively.

Clearly, the first thing we need to formalise in order to formalise identities is the expressions we allow to occur in the identity. The expressions are formed by, symbolic, method application, starting with some special method called new. In the first step, we only considered well-typed applications, later we introduce

grammars on these sequences; as discussed in the previous section. The well-typing check can be likened to a static type check, while the grammatical check can be likened to a dynamic type check.

The well typed sequences are built inductively from basic sequences. These basic sequences are the creation of new objects using *new*. This special method is truly polymorphic in that it is defined for all types. As a parameter it gets a constant c of a certain type s, the result is an object of type s with value c. Hence, we get the following definition:

DEFINITION 10.1

The set of basic expressions is given by the set:

$$BE = \{new(c)\colon s \mid s \text{ an } is-type,\ c \in D_s\}$$

Note that the typing of the *new*-method is consistent with the usual typing definition of methods as *new* has no input type (just parameters), i.e. the input type is 1.

From this set of basic expressions and the set M of typed methods, we can define the set of typed identity expressions *TIexp* inductively as follows:

DEFINITION 10.2

Let M be the set of, typed, methods, and let BE be the set of basic expressions, then the set *TIexp* of types identity expressions is given inductively by:

1) $x\colon s \in BE$, then $x\colon s \in TIexp$
2) $x\colon s \in TIexp$, $m\colon t^s \in M$, then $m(x)\colon t \in TIexp$.
3) $x_1\colon s_1, \ldots, x_n\colon s_n \in TIexp$, and $m\colon t^{s_1 \times \cdots \times s_n} \in M$, then $m(x_1, \ldots, x_n) \in TIexp$.
4) $x_1\colon s, \ldots, x_n\colon s \in TIexp$, and $m\colon t^{P(s)} \in M$, then $m(\{x_1, \ldots, x_n\}) \in TIexp$.

For each of the *is*-types s, we have a subset of *TIexp* consisting of those expressions that have type s. This subset is called expression-type of s:

DEFINITION 10.3

Let s be an *is*-type, then *expression-type(s)* is the subset of *TIexp* with type x is s.

In the first section of this chapter, we have sketched identities as sets of expressions. Clearly, identities are not arbitrary sets of expressions, but depend on:

1) All the expressions should encode information on the same real world entity. As sketched in the previous sections, this is achieved by a sub-expression relation.

2) The *is*-types in the schema, as some types are solely used to define *HAS-A*-relations, which are/can not be considered as objects in their own right.

3) The application of methods, or better *i-methods*, as the methods determine what sets of expressions can be seen as a database.

4) The database itself; in the first section of this chapter, we sketched that the identity is defined relative to a database.

So, the next step toward the definition of identities, is the, natural, definition of subexpression; which is defined inductively as follows:

DEFINITION 10.4

The subexpression relationship on TIExp is defined recursively by: e_1 is a *subexpression* of e_2 if:

1) $e_2 = e_1$

2) $e_2 = m(e_3, \ldots, e_n)$ and e_1 is a subexpression of at one of the expressions in $\{e_3, \ldots, e_n\}$.

The following step is less straightforward, we coined the term *i-methods* above, because methods as we defined them before do not yet encode how they should be applied to identities. So, we seem to be in a vicious circle, to define identities, we need *i*-methods, and to define *i*-methods, we need identities. However, the way out is that we do not need identities, but only *pre-identities* to define *i*-methods. A pre-identity, is a set of expressions of which the set of types have a lowest element. Hence, we have the following definition:

DEFINITION 10.5

A set of expressions S is a *pre-identity*, if there is an $x: s \in S$, such that for all $y: t_y \in S$, $t_y \leqslant s$. The type s is called the type of s

Now we can define *i-methods* as follows:

DEFINITION 10.6

Let $m \in M$ be a method of type $t^{\sigma_1 \times \cdots \times \sigma_n}$, the *i-method i-m(m)* is the method m, extended with rules of the form:

$$\alpha_1: s_1 \in i_1, \ldots, \alpha_n: s_n \in i_n m\alpha \Rightarrow m(\alpha_1, \ldots, \alpha_n) \in m(i_1, \ldots, i_n)$$

$$\alpha: s \in i_j \Rightarrow m\alpha \in m(i_1, \ldots, i_n)$$

$$\alpha: s \in i_j \Rightarrow \alpha \in m(i_1, \ldots, i_n)$$

in which the $i_j: \sigma_j$ are pre-identities.

The rules in the definition define how m should be applied symbolically to the expressions in the identities. Note that the result consists only of those expressions that are explicitly mentioned in the rules definition.

Now that we know how methods can be applied to pre-identities, we can define the operational semantics of methods, i.e. the 'update' semantics of a database consisting of a set of expressions:

DEFINITION 10.7

Let db be a set of expressions, moreover, let $m \in M$ be a method of type $t^{\sigma_1 \times \cdots \times \sigma_n}$ and let $i_j : \sigma_j$ be pre-identities, such that $\sigma_j \subseteq db$. Then the application of m to $(i_1, \ldots, i_n)$ in db id defined by:

$$m(i_1, \ldots, i_n; db) = db \setminus (i_1 \cup \cdots \cup i_n) \cup i - m(m)(i_1, \ldots, i_n)$$

Clearly, a database is not a random set of expressions. The set of methods m determine the possible states of a database. In fact, now that we have defined the operational semantics of a method, we can define the set DB of extensions of a database schema inductively. The starting point of this inductive definition lies of course in the basic expressions. Hence, we get the following definition:

DEFINITION 10.8

Let BE be the set of basic expressions, and let M be the set of methods, then the set DB is defined inductively by:

1) If $x \subseteq BE$, then $x \in DB$.
2) If $x \in DB$, $m : t^{\sigma_1 \times \cdots \times \sigma_n} \in M$ and the pre-identities $i_1 : \sigma_1 \subseteq x, \ldots, i_n : \sigma_n \subseteq x$, then $m(i_1, \ldots, i_n; x) \in DB$.

Identities are pre-identities, defined relative to some $db \in DB$. Moreover, we want that the different expressions in such a pre-identity convey information on the same real world object. As we have sketched in the previous sections, this is achieved using subexpressions. Hence, we get the following definition of identities:

DEFINITION 10.9

Let $db \in DB$, let $x : s \in db$, and let s be an *is*-type. The identity of type s generated by x in db, denoted by $i_{x,db} : s$ is the pre-identity that consists of all expressions $y : t \in db$, such that:

1) $t \leqslant s$.
2) x and y have a common sub-expression.

From now on, method application is only allowed if all the pre-identities involved are actually identities. Note that due to our operational definition of

DB, this assumption changes the definition of *DB* in retrospect; and thus the definition of identities themselves. However, these changes are obvious.

Now that we have defined identities, we can define the identity types:

DEFINITION 10.10

Let s be an *is*-type, the identity type $i(s)$ associated with s is the the set of all pre-identities i of type s, such that there exists a $db \in DB$ such that i is an identity of type s in *db*.

The *is*-types are in a subtyping relationship. This relationship is established by means of functions in the type graph. So, to transfer the subtyping relationship to identity types, we first have to define functions between identity types; or better how to apply the functions from the type graph to identities. The identities represent objects, hence, function application has two different semantics, depending on the target type:

1) If the target type is an *is*-type, the result of the function application should again be an object, i.e. an identity.
2) If the target type is not an *is*-type, the result can never be an identity, hence it should be a value; the value the object represented by the identity has for this particular attribute.

The function application under the second semantics is the easiest. It simply means that the identity should be evaluated to yield a value; the function should then be applied to that value. Evaluating an identity simply means the evaluation of each of expressions in the identity. So the value of an identity is a set of *is*-objects. The function is than simply applied to the *is*-object with the highest type; note that this object is necessarily unique by our definition of an identity. Now and again, we will refer to the final result as $f(eval(i))$, to highlight the intended semantics of the function application

The first kind of semantics for function application are much more difficult, as the result should again be an identity. Clearly, this is a difficult problem for which different options are available. However, to keep this chapter within reasonable bounds we simply make an assumption that renders the problem trivial:

ASSUMPTION 10.11

Let $s \leqslant t$, with $f: s \to t$ and let i_1 be an identity of type s in a database $db \in DB$. Then there exists exactly one identity i_2 of type t in *db* such that:

1) $i_1 \subseteq i_2$.
2) $f(eval(i_1)) = eval(i_2)$.

The first requirement of the assumption ensures that i_2 and i_1 encode information on the same real world entity. The second requirement is needed as there

might be more than one function from s to t. More than one function implies that there might be more than one object i_2 that satisfies the first requirements. For example, if s is the type *couple* and t the type *person*, there will always be two *person*-identities that satisfy the first requirement. The second requirement makes the choice between these two persons unique.

Clearly, the intention of the assumption is that we can define $f(i_1) = i_2$, with i_1 and i_2 as in the assumption above. So, the assumption is most certainly effective, but is it also a reasonable assumption? The assumption can be enforced if we restrict method application by the following two requirements:

1) If a new object is created, either using *new* or by the *merge* of two existing objects, it is created with the possible highest type(s). So, if we want a new object *employee*, we first create a new object *person* and than we *hire* this *person* as an *employee*.

2) If an identity i has an expression e_1 of type s and an expression e_2 of type t with $s \leqslant t$ and the application of a method preserves an expression of type s, it should also preserve an expression of type t. Moreover, if a method application changes an 'attribute value' that is shared with an expression of a higher type, the method should also be applied to this expression as well. In other words, methods should be applied at the highest type level possible. For example, if an *employee's* age should be increased by one year, the associated *person's* age is increased by one year.

Both requirements seem to be simply requirements of good database design rather than severe restrictions. Hence, we feel that our assumption is reasonable.

The definition of the application of functions under the first kind of semantics, as defined above, is in fact a function $id \times db \rightarrow id$. This means that we cannot simply generalise our function application to *identity types*; as the result depends on the current database. This may seem a drawback, as it implies that if we built structures from identities, in analogy of the definition of structure types from entity types, the 'root-identity' can not be used to determine the other identities. However, we do not have to built structures, as the assumption above guarantees that we can always find the 'other' values by simply inspecting the database. Besides, the 'value' of an expression is still an *is*-object, so the type of the expression encodes which functions can be applied to the expression.

The subtyping relationship between identity types, is now simply the subtyping relationship between their *is*-types. This may seem contradictory as an identity i_1 of type t can not be coerced into an identity of type s if $t \leqslant s$. However, by the assumption above we can see i_1 as the specification of a unique identity i_2 of type s. Moreover, as $i_1 \subseteq i_2$, they encode information about the same real world entity. In this sense, i_1 and i_2 can be seen as the same identity.

10.4 Classes and complex objects

In this section, we use the identity types defined in the previous section to define classes and, finally, complex objects. Classes are defined from identity types through the intermediate definition of *grid types*. These grid types are identity types extended with a grammar. Applications of such a grammar in the database design area are sketched by showing how the grammar can be used to model dependencies. Classes are then defined as equivalence classes of grid types. This means that contrary to the usual class definition, a class describes which methods have been applied to an object, rather than which methods can be applied to an object. It is shown that the usual definition of a class is a special case of our definition, viz. the case in which the grammars are restricted to *regular grammars*.

As indicated above, the first step toward the definition of classes is the definition of *grid types*. A grid type is an identity type in which the expressions are restricted by grammars, as sketched in section 2 of this chapter. This restriction is accomplished by defining grexpression-types as $<$*expression type*, *grammar*$>$-pairs. Let $s_G = <s, G_s>$ be such a pair, an expression $exp : s$ is an element of s_G if s belongs to $L(G_s)$. Hence, $L(G_s)$ should be a recursive set as in more general cases the membership problem is undecidable, c.f. (Hopcroft and Ullman (1979)).

To define a grammar, we need an alphabet. The obvious choice for the alphabet in this case is the fixed set M of methods, i.e. $\Sigma = M$. So, we get the following definition of an *grexpression type*:

DEFINITION 10.12

Let s be an expression type, and let G_s be a grammar such that $L(G_s)$ is a recursive subset of Σ^*, then the *grexpression type* s_G is defined as the pair:

$$s_G = <s, G_s>$$

Above, we sketched the semantics of such a pair by stating that it contains the elements of s that satisfy G_s. But clearly, the expressions of type s are not even elements of Σ^*. We first have to define a function from s to Σ^* before we can formally define the semantics of the grexpression type s_G. As already sketched in the second section of this chapter, this function should simply forget the brackets in an expression as well as the parameters of *new*-methods. Hence, we get the following definition:

DEFINITION 10.13

Let s be an expression type, the function $\phi_s : s \rightarrow \Sigma^*$ is defined by:

$$\phi_s(new(c)) = new$$

$$\phi_s(m(m_1, \ldots, m_n)) = m(\phi_s(m_1) \cdots \phi_s(m_n))$$

In the remainder of this chapter, we will use the 'generic' function ϕ, the context will make clear which ϕ_s is intended.

Given the function ϕ, the semantics of an grexpression type is now obvious:

DEFINITION 10.14

Let $s_G = <s, G_s>$ be an grexpression type, an expression $exp : s$ is an element of s_G iff $\phi(expr) \in L(G_s)$.

Note that we typing issues do not have to be taken care of in the definition of G_s, as the s component of the grexpression type ensures the well-typedness of the expression. In fact, that is the reson why we use the abbreviation *grid types* for grammatical identity types, as the grammar can be seen along one axis, while the type check occurs along the other edge.

Grid types are defined from grexpression types analogically to the construction of identity types from expression types. The main difference is that the application of a method to a set of grexpressions is only allowed if all the resulting expressions are again grammatically correct. Rather than repeating the complete construction, we use the following, informal, definition:

DEFINITION 10.15

A *grid type* is an identity type, in which all the expression types are replaced by their grexpression types. Identities of a grid type are called *gridentities*

The application of a method on an identity of a grid type is only allowed if the resulting expressions are grammatical. Moreover, the assumption for identities is extended to gridentities by replacing the word identity by gridentity, i.e.:

ASSUMPTION 10.16

Let $s \leqslant t$, with $f : s \rightarrow t$ and let i_1 be an gridentity of type s in a database $db \in DB$. Then there exists exactly one gridentity i_2 of type t in db such that:

1) $i_1 \subseteq i_2$.

2) $f(eval(i_1)) = eval(i_2)$.

Note, the relationship $s \leqslant t$ holds for grid types s and t if it holds for the associated identity types. This assumption can be 'implemented' similarly to the implementation of the assumption for identity types as discussed in the previous section. The effect of the assumption is that an *employee* gridentity is also 'grammatical' as a *person* gridentity

The advantage of defining the grammars with expression types rather than with identity types is that the grammar is now automatically shared by all identity types that have this grexpression type. To illustrate the expression power gained by adding the grammar, we turn our attention, again, to dependencies.

10.4.1 Grammars and dependencies

The first dependency we investigate is the functional dependency. In Chapter 4, we have seen that in principle all sets of algebraic dependencies on a relation schema can be translated to a subtype construction of that relation schema. Moreover, we have seen that for a smaller class of dependencies a more natural translation exists provided we find a natural translation for functional dependencies. The aim of this example is to show that grammars can be used to give such a natural translation of functional dependencies.

A functional dependency (fd) $A \rightarrow B$ in a relation schema $r = ABC$ means that an A-value can be associated with at most one B-value throughout the table. The name of the dependency as well as its semantics suggest that we could use functions to model fd's. However, consider the functional dependency that states that a person can have at most one dentist. Clearly, there is no function expression that given a person returns a dentist. This means that we need a *person* $\times$ *dentist*-table that encodes the currently known part of the function. But that means that we did not solve the problem of modelling a functional dependency, as we still have to define a fd on the *person* $\times$ *dentist*-table.

Another way to look at our example fd is that a *person* can enlist with only one *dentist* at the time. In other words, before a person can enlist with a new dentist, she has to revoke her previous enlisting. But this means that the functional dependency can be seen as a *dynamic constraint*: enlisting with a dentist is only allowed if there was no previous enlisting, or if there was a previous enlisting, this enlisting should first be revoked. In a pseudo grammatical notation, this dynamical constraint can be written as:

$$(X_{rev,enl}\ rev\ Y_{rev,enl}\ enl\ Z_{rev,enl})^*$$

in which the $X_{rev,enl}$ stand for a string of methods not containing *rev* or *enl*.

The upshot of this example is that we can model functional dependencies as dynamical dependencies using the grammars of grexpression type. Moreover, note that in our example, *person* and *dentist* will denote objects, while the *person* $\times$ *dentist*-relation denotes a HAS-A relationship between *persons* and *dentists*. We believe that this is in general the 'correct' way to model dependencies, i.e. we see dependencies as a proposition on a relationship between objects rather than merely values. The ration behind this point of view is that the different occurrences of the same value in a table are identified as being the same. The motivation for the identity of objects is just this identification of different appearances.

The Armstrong Axioms for functional dependencies are a sound and complete axiom system for the inference of new fd's from a given set of fd's, c.f. (Armstrong (1974)). This theorem can be stated as follows:

THEOREM 10.17

Let r be a relation schema with attribute set U, then the following axioms form a sound and complete axiom system for the inference of functional dependencies:

1) If $Y \subseteq X \subseteq U$, then $X \rightarrow Y$.

2) If $X \rightarrow Y$ and $Z \subseteq U$, then $XZ \rightarrow YZ$.

3) If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$. □

Clearly, this axiom system is defined, and the theorem proved, within relational database theory. It is an interesting exercise to interpret these axioms in our approach to fd's:

1) The first axiom simply states that where ever an object appears in a database, it appears with the same subobjects. The validity of this statement is immediate.

2) The second axiom states that if an X can be associated with one Y at the time, then an association of an X and a Z can be associated with an YZ association at most one at the time. This is again immediately true, for $XZYZ$ is a subtype of XY, hence, an $XZYZ$-expression should be grammatical with regard to the XY-grammar.

3) The validity of the third axiom, can be seen by the observation that the following two pseudo-grammar rules:

$$(X_{m_1,m_2}\, m_1\, Y_{m_1,m_2}\, m_2\, Z_{m_1,m_2})^*$$

$$(X_{m_3,m_1}\, m_3\, Y_{m_3,m_1}\, m_1\, Z_{m_3,m_1})^*$$

can be combined to the rule:

$$(X_{m_3,m_2,m_1}\, m_3\, Y_{m_3,m_2,m_1}\, m_2\, Z_{m_3,m_2,m_1}\, m_1\, A_{m_3,m_2,m_1})^*$$

Due, to our operational semantics, a formal proof of the soundness of these rules would be rather complicated. Therefore we leave the proof at this informal level.

The fact that functional dependencies are translated to grammatical (dynamical) constraints implies that (non-anomalous) sets of multi-valued dependencies and acyclic join dependencies are modeled partially structural and partly dynamical. For example, the mvd $A \rightarrow\!\!\rightarrow B$ over the schema leads to the type $P(A \times P(B)_A \times C)$, as we have seen in the fourth chapter. The $P(B)$ represents the structural component of the translation of the mvd. While the fd $A \rightarrow P(B)$ represents the dynamical translation of the mvd. In plain words, we can restate this translation as saying that an A object can be associated with an 'set of B's'-object only once. Note that this does not means that we can not change the contents of the 'set of B's'- object but that it is associated with at most one such object.

This translation of static integrity constraints to dynamic constraints is rather different from the approach taken by Abiteboul and Vianu in e.g. (Abiteboul and Vianu (1985)). The purpose of these authors is a complete replacement of static constraints by dynamic constraints; whereas our translation is only partially dynamical, the other part is structural. The reason why we prefer such a mixed approach is that a purely dynamical translation of mvd's would again be counter intuitive; in the sense that the modification of one tuple would still have to be reflected in other tuples, whereas the static structure does not indicate that these tuples are connected. Note, that our approach shares a drawback with the approach from Abiteboul and Vianu (1985); viz. the undecidability of proving transaction schemas (in their terminology) equivalent. We return to this problem later in this section.

It would be interesting to investigate whether a partially dynamic translation of more complicated (sets of) algebraic dependencies would yield them more amenable to a structural translation. However, such an investigation is beyond the scope of this thesis. if only as such dependencies are rather artificial as argued in both Chapter 1 and Chapter 4. Instead, we turn our attention to dynamical and temporal constraints.

Dynamical and temporal constraints are far less studied in the literature than static constraints; see however (Meyer, Weigand, and Wieringa (1989)). Consequently, a detailed study of the class of dynamical and temporal constraints that can be modelled with grammars is rather complicated. So, we only note that the functional dependency is a very limited example of the class of dynamical constraints that can be modelled with a grammar. In general, almost each dynamical constraint that makes a statement on the (partial) order of events in the UoD can be modelled using a grammar. In fact, the class of temporal constraints, we can model without referring to time (which we did not model explicitly) is a subclass of the dynamical constraints. For the grammar can be used to express temporal constraints of the form: 'such actions will never happen in the future' or 'such an action will certainly occur in the future'. As an aside, note that the possibility to express such a requirement does not imply that it is easy to check that such a requirement actually holds.

10.4.2 Grammars and type equivalences

In the previous subsection, we have shown that the addition of grammars to our formalism extended its expressiveness with regard to dependencies. In this subsection, we briefly investigate the effect of the addition of grammars on type equivalences. So, the problem we try to solve is, given two grexpression types $<s, G_s>$ and $<t, G_t>$, decide whether or not these types are equivalent. There are two levels at which this problem can be tackled; i.e. two notions of equivalence:

1) The 'value-oriented' notion of equivalence. For this approach, note that G_s defines a subtype of s, say s^{G_s}, consisting of those values of s that can

actually be reached. The value oriented notion of equivalence is then based on the equivalence of s^{G_s} and t^{G_t}.

2) The 'object oriented' notion of equivalence. Under this notion of equivalence, not only s^{G_s} and t^{G_t} should be equivalent, but also G_s and G_t. This approach is nick-named the object oriented approach, as it does not only 'save' the value of an object, but also its identity.

The only reasonable definition of equivalence of grammars is that their languages are isomorphic. However, it is well-known that the question whether two grammars produce the same language is undecidable for context free languages, moreover, the question is not settled for deterministic context free languages, c.f (Hopcroft and Ullman (1979)). This implies that a general solution to our problem requires a rather drastic reduction on the expressiveness of the grammar. It seems a better solution to leave the equivalence in principle up to the database designer to proof and furnish her with a set of 'solved problems'.

In this subsection, we are going to solve such a problem. In fact, due to our rather informal semantics in this chapter, we only pursue the value oriented equivalence; the equivalence of the grammars is left as an exercise to the reader.

The problem we are going to investigate is motivated by the 'grouping'-operator found in many data models. This operator can be used to group values according to some (variable) requirement; i.e. all values that satisfy the same requirement instance are grouped together. The simplest case of grouping is given by a table of tuples over an AB-relation, in which one wants to group the B-values on their associated A-values. In other words, in this simple case, grouping can be seen as a function:

$$group : P(A \times B) \rightarrow P(A \times P(B))$$

In fact, the *group* function in this example should be defined such that each A-value in the table is associated with exactly one set of B-values. Hence, the function group can be seen as a function:

$$group : P(A \times B) \rightarrow P(A \times P(B)_A)$$

In this form, the grouping function has an obvious inverse, viz.

$$ungroup : P(A \times P(B)_A) \rightarrow P(A \times B)$$

which maps the tuple $(a, \{b_1,..,b_n\})$ to the set of tuples $\{(a, b_1), \ldots, (a, b_n)\}$.

So, $P(A \times B)$ and $P(A \times P(B)_A)$ are isomorphic. Moreover, as we have seen in the previous subsection, both are the 'value-set' of grexpression types. Hence, this shows that the grexpression types:

$$<P(A \times B), \varnothing>$$

and

$$<P(A \times P(B)), A \rightarrow P(B)>$$

are value oriented equivalent. We hope that the reader sees that they are also object oriented equivalent.

This simple example, is easily extended using the result of the previous subsection. For, we have seen that a multivalued dependency can be modelled by a grouping and a functional dependency. It is easy to see that the following result holds:

$$P(A \times P(B)_A \times C) \equiv$$
$$P(A \times P(B)_A \times P(C)_A) \equiv$$
$$P(A \times B \times P(C)_A)$$

And similarly as above, this implies that:

$$<P(A \times P(B) \times C),\ A \rightarrow P(B)> \equiv$$
$$<P(A \times P(B) \times P(C)),\ A \rightarrow P(B) \wedge A \rightarrow P(C)> \equiv$$
$$<P(A \times P(B) \times C),\ A \rightarrow P(C)>$$

Note that this result is a restatement of one of the inference axioms for multivalued dependencies.

The fact that grouping is a special case of equivalence of grexpression types, implies that if we support equivalence types over grexpression types, grouping comes for free. In fact, then the user does not have to group explicitly, but may assume that the data is grouped whenever that is convenient. This is the reason why we did not include a 'grouping' operation into SOQL.

10.4.3 Classes and complex objects

Finally, we have developed enough concepts to be able to define *classes* and *complex objects*. Classes are constructed from grid types, by replacing the grexpression types by equivalence types of grexpression types. Complex objects are defined as elements of a class. We start this subsection by a more formal definition of these concepts, and we end it by comparing our notion of a class with its usual definition.

Clearly, the first concept we have to define is the notion of an equivalence class of grexpression types. In principle, this is of course similar to the definition of equivalence classes of entity types. However, we have the choice between two notions of equivalence as shown in the previous subsection. So, should we use the object oriented equivalence or restrict ourselves to value oriented equivalence, or should both be supported? To illustrate the relative merits of both equivalences, let $s = <P(A \times B,\ \varnothing>$:

1) If a method m is defined on $P(A \times P(B)_A)$ rather than $P(A \times B)$, the object oriented equivalence should of course be used. Similarly, if a query explicitly asks for identities in $P(A \times P(B)_A)$-format.

2) If however, in a query a value comparison occurs, we only have to convert values, and hence a value oriented equivalence seems sufficient.

In the example for value oriented equivalence we say 'seems to be sufficient' as we have to be careful: if a value equivalence is used for which no matching object oriented equivalence exists the result value cannot be seen as a legal value of the object. Hence, besides value oriented equivalence should not be supported. Note that this does not rule the equivalence class of the associated *is*-type; as for those equivalences, the associated object oriented equivalence acts as the identity on the grammars!

So, equivalence classes of grexpression types consists of grexpression types that are equivalent under the object oriented equivalence relation. We have already seen in the previous section that it is undecidable whether two grexpressions are equivalent under this relation. Hence, the equivalence class of a grexpression has to be defined explicitly by the database designer. So, we get the following definition:

DEFINITION 10.18

Let $<s, G_s>$ be a grexpression type, the equivalence class of $<s, G_s>$, denoted by $[<s, G_s>]$, consists of those grexpression types of whom the equivalence with $<s, G_s>$ is explicitly provided by the database designer.

Equivalence types of grexpression types are often referred to as eqgrexpression types in the remainder of this thesis. As an aside, note that we assume that all *is*-type equivalences are automatically taken care of.

The semantics of equivalence classes of entity types were constructed using the co-equaliser construction. For grexpressions, we only have an operational semantics. Hence, we cannot use co-equalisers. In other words, the semantics of eqgrexpression types can only be given operationally. These operational semantics are simply that the system chooses one of the offered representations and converts the representation automatically upon, implicit, request using the functions provided by the database designer.

Classes are the equivalent of identity types and grid types, i.e. classes are grid types in which the grexpression types are replaced by eqgrexpression types. Similar to the definition of grid types, the lack of a formal semantics for identity types prohibit a more formal definition of classes. Hence, we have the following definition:

DEFINITION 10.19

A class is an identity type in which all expression types are replaced by eqgrexpression types.

Complex objects finally, are simply defined elements of a class, i.e. as identities all of whose expressions are eqgrexpressions:

DEFINITION 10.20

A complex object is an element of a class. The identity of a complex object is given by its expressions, its value can be computed by evaluating these expressions.

The assumption we made for identities and extended for gridentities is again extended for complex objects, i.e. we make the assumption:

ASSUMPTION 10.21

Let $s \leqslant t$, with $f: s \to t$ and let i_1 be a complex object of the class c_1 associated with s in a database $db \in DB$. Then there exists exactly one complex object i_2 of the class c_2 associated with the type t in db such that:

1) $i_1 \subseteq i_2$.

2) $f(eval(i_1)) = eval(i_2)$.

It is interesting to compare our definition of a class with the usual class definition. Usually, a class can be seen as a type definition together with a list of methods that can be applied to an object in the class. In our definition, the respective grammars of the eqgrexpressions in the class definition are in a sense the opposite of the list of applicable methods. The grammars do not list which methods can be applied in the future but what methods may have been applied in the past.

However, it only seems to be the opposite, because if all the grammars in a schema are regular, we could easily replace our class definition by a more conventional definition. The translation step is than simply based on the facts that the set of regular grammars is closed under *union* and that regular grammars are equivalent to finite state machines; c.f. (Hopcroft and Ullman (1979)). This results in a partition of the types such that if a method can be applied to an object of type a, the result is guaranteed to be grammatical. As an aside, note that this will lead to a combinatorial explosion of the number of types in the schema.

Translations in the other directions are also simple, as it amounts again to the equivalence of regular grammars and finite state machines. So, the usual definition of a class is simply a special case of our definition. It may seem that the conventional definition is preferable to ours as it clearly specifies how objects may be manipulated. However, there are several reasons why we prefer our solution:

1) The methods that are potentially applicable to an object of a class c can be found using the associated *is*-type s of c. For, given s, it is easy to write an ISOQL query that results in the set of all methods that are applicable to objects of type s. In fact, as we assume that the set of methods is fixed, such a query could be done 'at compile time' such that the user automatically

sees a list of all methods that are potential applicable. As an aside, note that such queries were the motivation for Rittri to study type equivalences, c.f. (Rittri (1989)).

2) One might argue that the list supplied above contains only methods that are potentially applicable, i.e. it may very well be that a method m from the list cannot be applied to a particular object i in the class, because the result would be ungrammatical. However, we do not feel that this is not a drawback but a feature as it can be used to distinguish between two different kinds of errors:

   a) You cannot give a *raise* to *cazzo*, as *cazzo* is a *cat* and *pets* can never be *employees*.

   b) You cannot give a *raise* to *Jones*, as *Jones* already got a raise this week and company policy forbids more than one raise a week; even for presidents.

3) Clearly, if all the grammars are regular, such errors can also be found by conventional classes, as sketched above. However, this means a far larger number of classes plus no distinction between the two types of errors.

4) The final reason is simply that conventional class definition is a special case of ours. So, our definition yields a greater expressibility.

We end this section, by the formal definition of HAS-A relations between complex objects and the related notion of object sharing:

DEFINITION 10.22

Let i_1 be a complex object of class c_1 and let c_2 be some other class. We say that i_1 *HAS A* c_2 iff there is an object i_2 of class c_2 in the database such that i_1 and i_2 have a common eqgrexpression.

Object sharing is defined as the extension of HAS-A relations already sketched in the first section of this chapter:

DEFINITION 10.23

Let i_1, i_2 and i_3 be objects of class c_1, c_2 and c_3 respectively, then we say that objects i_1 and i_2 share object i_3 if both i_1 and i_2 have an eqgrexpression in common with i_3.

Note that for object sharing that i_1 and i_2 do *not* have to share the *same* eqgrexpression with i_3.

These last two definitions end our discussion of classes and complex objects. In the next section, we investigate query languages for complex objects.

10.5 Queries

In the previous sections, we developed an operational definition of classes and complexes objects, as an extension of *is*-types and *is*-objects. This means that yet again, we have to redefine our query language, to adapt it to the new situation. Now, the fact that we did not give categorical semantics for the concepts in this chapter is going to work against us. Because, the lack of categorical semantics mean that we cannot use the semantics of ISOQL. In fact, any attempt to give formal semantics to the new query language are doomed by the lack of formality in the definition of the concepts.

So, we can only sketch the semantics of the new query language and trust that the reader understands the intended semantics. The lack of a formal semantics has in turn its influence on the definition of a syntax of a query language. Many of the finer details of the syntax of a language are directly influenced by the semantics of that language. Therefore, we do not attempt to define a new query language, rather we describe four aspects of such a query language in four short subsections. The four aspects are: value oriented queries, HAS-A relations, grammars and historical data; hence, the last three subsections can be seen as object oriented queries.

10.5.1 Value oriented queries

Queries on a database of complex objects will in general result in complex objects. The only exception to this rule could be that one would allow a user to compute functions over the database. This is however not the subject of this subsection. Rather, we are interested in queries that only use the value(s) of an object, rather than its identity. In this sense, the title of this subsection may seem to be misleading.

The values of complex objects are *is*-objects. So, these values are amenable to ISOQL queries. The main difference with *ISOQL*-queries in *is*-databases is that the result of a query will be a collection of objects rather than a set of values. This holds of course both for intermediate results and for the final result of a query. Although this difference is conceptual rather than syntactical, it has syntactical consequences. For it means that we in every comparison we have to explicate whether we mean that the two sides have to be seen as *is*-objects or as complex objects. In other words, we have to make a distinction between e.g. *value-equality* or *object-equality*.

Such distinctions are well-known in object oriented languages and object oriented databases. In fact, in (Masunaga (1989)) the author lists a potentially infinite number of equalities. In this subsection, however, we restrict ourselves to the two equalities mentioned above. As an aside, note that the famous *deep-equal* and *shallow-equal* coincide in Flock, as the root-value completely determines the other values of an *is*-object.

The difference between a true value comparison and a object comparison is easily made by specifying whether or not a complex object should be evaluated or not. This can of course be done by a special key-word, say *ev*. Note that we did not use the key-word *eval* as this would cause confusion whether method application or object evaluation is meant. Moreover, it would complicate the stratification rules of a program severely, as recursion over *ev* is of course completely legal, while recursion over *eval* is unstratified.

We end this subsection, by pointing out a semantical difference between queries over *is*-databases and queries over complex object databases. Complex objects are described as a set of expressions. Moreover, we have seen in the first subsection that such a set can be constructed from the collection of expressions called the database; i.e. the complex objects are not 'stored' as sets of expressions, but constructed when they are needed. Finally, we made the assumption that whenever we have an object of type s and t is a type with $s \leqslant t$, then we can construct the related t object. In the semantics of *ISOQL* queries, we had to deal with the different types of the *is*-objects that satisfy a query. Now we no longer have to do this, because the identity of a complex object that satisfies a query has the identity of all of its specialisation as a subset. In other words, we no longer have to use the co-product of types to describe the result type of a query.

10.5.2 HAS-A relations

A major difference between *is*-types and classes is that we will have far less classes than *is*-types because not every abstract 'relation' between classes will lead to a new class. We have seen in the first section of this chapter that it is very well possible that the relationship is only encoded within the identities themselves. The advantage is clearly that the user is not bothered with all kinds of exotic types, caused perhaps by the fact that one object has many relations. The disadvantage is of course that the user cannot refer directly to the relations as she could with *is*-types

However, this disadvantage is only seemingly. For example, suppose that we do not have a distinct class that registers the relationship between *persons* and *telephones*; in other words we do not associate a class with the type *person* $\times$ *telephone*. If we want information on the telephone a certain *is*-object *person* has, we have to refer to the *person* $\times$ *telephone*-type. For classes and complex objects however, we can simply use a HAS-A relation! So, the query which *telephone person i* could be phrased something like:

$$T \supseteq \{t\text{: } \mathit{telephone} \mid i \ \mathit{HAS}\ t\}$$

or symmetrically:

$$T \supseteq \{t\text{: } \mathit{telephone} \mid t \ \mathit{HAS}\ i\}$$

Moreover, the keyword *HAS* can be generalised to *SHARE*, which can be used to find out which objects are shared by objects $i_1, \ldots, i_n$. An example of such a

query would look something like:

$$T \supseteq \{t\text{: } \textit{telephone} \mid \{i_1, \ldots, i_n\}\ \textit{SHARE}\ t\}$$

The result of this query is obviously that T consists of those telephones that are shared by the persons in $\{i_1, \ldots, i_n\}$.

Note that both *HAS* and *SHARE* are set-theoretic manipulations of the identities involved. So, an obvious generalisation of these constructions is to allow set-theoretic manipulation of identities. This has two major advantages:

1) No special keywords are needed, which makes the syntax simpler. Moreover, it will bring the syntax much closer to the semantics.

2) The only other way to create a set of keywords which has the same expressibility as set theory, boils down to reinventing set theory.

So, obviously set-theoretic manipulations of identities are preferable over new keywords. The only potential hazard is that we can both manipulate identities and values set-theoretically. However, with the simple keyword *ev* given in the previous subsection, the distinction between the two interpretations of the following expression becomes obvious:

$$i_1 \cap i_2 \neq \varnothing.$$

Using the set-theoretic operations, the two queries given above can be phrased like:

$$T \supseteq \{t\text{: } \textit{telephone} \mid i \cap t \neq \varnothing\}$$

and

$$T \supseteq \{t\text{: } \textit{telephone} \mid i_1 \cap t = \varnothing \wedge \cdots \wedge i_n \cap t \neq \varnothing\}$$

10.5.3 Grammars

In the previous subsection we introduced set-theoretic manipulations of identities in a query. Given that classes are defined by placing grammatical restrictions on expressions, a logical next step is allowing grammars in a query. The grammars can be used to single out objects that have a particular history. To give an example, recall that we have translated functional dependencies to a dynamical constraint. The example in which a *person* was allowed to be enlisted with at most one *dentist* at the time, was encoded by the pseudo grammatical rule:

$$(X_{rev,enl}\ rev\ Y_{rev,enl}\ enl\ Z_{rev,enl})^*$$

This translation can also be used to query the database for *persons* that have only one *dentist* whether or not this is an actual constraint, by simply adding the rule as an extra requirement the objects should satisfy. This satisfaction relation is

computed the same as the question whether an expression is grammatical, i.e. $\phi(expr)$ is checked.

The functional dependency is an example in which the history of an object is used to determine current information. Clearly, grammars can also be used for purely historical information. For example, we can write a query that returns those persons that have lived the American dream, i.e. who became a millionaire after being a paper-boy, provided there are methods from which we can determine these states. For simplicity, we assume that these methods are known as *paper-boy* and *millionaire* respectively. The rule would then look something like:

$$X \textit{ millionaire } Y \textit{ paperboy } Z$$

Note that such a rule would also return people that failed the american dream in the sense that they started out as a millionaire to become a paper-boy, but who became a millionaire again later. It should be clear how such 'failures' can be ruled out.

Using the grammars, we can also use objects that do not exist as separate objects any more. For example, we can ask which *cars* have an *engine* that was rejected after its assembly, but admitted after renovation, although the engine is not seen as a separate object. The key to such a queries is that the identity of a *car* has at least one expression which has this history of the engine as a subexpression. So, the query could use a rule like:

$$X \textit{ assemble } Y \textit{ renovate } Z \textit{ reject } A$$

From the examples above, we see that grammars in a query can be used to phrase conditions on the order in which certain methods have been applied to an object. In general, they cannot be used to relate the history of unrelated objects. For example, we cannot ask for cars that were assembled after Bush became president. For such comparisons we need time in our formalism. This is the subject of the next and final subsection.

10.5.4 Historical data

The identity of an object encodes the complete history of that object. So, if we allow a partial evaluation of an expression, we can retrieve historical states of an object. Such an historical state can be used in the query similar as the current state can be used. For example, suppose we want to know those employees that used to earn 60.000 when they were promoted to a managers position. The first step is then to retrieve those managers that have been a manager:

$$X \textit{ promote manager } Y$$

The following step is the partial evaluation, which could be denoted by something like:

$$y = \textit{part-ev}(x, \textit{ promote manager})$$

The operation *part-ev*$(x,$ *promote manager*$)$ should be interpreted as return the value of x right after the *promote manager* method was applied. Finally, the salary of y should be compared with 60.000.

Note that *part-ev*$(x,$ *promote manager*$)$ is not necessarily uniquely defined, it may very well happen that an employee has been promoted to manager more than once in his career. Such employees should be in the result of the query if on one of these occasions there salary was set to 60.000.

Partial evaluation is a limited means to retrieve historical data from the database. The most severe limitation is that it cannot be used to retrieve a historical database state. This means that query that asks for the cars that were assembled after Bush became president cannot be answered.

To answer such queries, we need a way to relate the history of different objects. Moreover, this objects do not have to be related in the current state. In other words, such queries need a time stamp mechanism. For, time as we discussed it in the second section can be used to order the different object histories. Obviously, the power of the query language can be increased greatly if time stamp comparisons can be made. However, we will not pursue such opportunities any further, as we did not incorporate time stamps in our model.

10.6 Conclusions

In this chapter, we will merge the various threads of this thesis, such as dependencies, type equivalences, methods, objects, and queries. The chapter starts with the definition of object identity by expressions that represent the history of an object. We show how this definition of identity supports in a natural way the modelling of object sharing and *HAS-A*-relationships in general.Moreover, we show how our notion of identity can be used to store rules in the database.

In the second subsection, we define constraints on object identities, by defining grammars the expression in an identity should satisfy. Moreover, we show how such constraints can be used to model both structural and temporal data dependencies in the database in a natural way.

In the fourth section, we extend the definitions of the third section to classes and complex objects. The construction of classes from identity types is similar to the construction of structure types from entity types. This implies that again type equivalence plays a major role in this definition. The grammars used to restrict the possible identities of a type, play a crucial role in these equivalences. Different from entity types, it is easy to show that the question whether two (restricted) entity types are equivalent is undecidable. This means that we can only offer some examples.

The examples are based on a grammatical formulation of the functional dependency. This is achieved by observing that a functional dependency can be seen as a dynamical dependency. Note that this result together with the results of

Chapter 4 mean that we can finally give a natural translation for relation schemas with sets of functional dependencies and conflict free sets of multivalued dependencies. The fact that we can finally model multivalued dependencies induce the examples: we simply use the inference axioms for multivalued dependencies.

The lack of a formal semantics for classes means that we cannot give formal semantics to their query language. So rather than defining a new query language in the last section of this chapter, we simply illustrate four aspects of such a query language, viz. value oriented queries, HAS-A relations, grammars and historical data.

Chapter 11

Conclusions and directions for future research

This brief chapter concludes this thesis. It consists of two sections. In the first section, we summarise the main results and compare these results with our initial goal as formulated in Chapter 1. In the second section, we discuss some future research directions, including remaining open questions.

11.1 Conclusions

To facilitate the reader, we give a summary of the main results of this thesis before we draw some conclusions. The main results are given by summarising each chapter, highlighting these results.

11.1.1 Chapter 1

The first chapter serves as an introduction to and a motivation for this thesis. A detailed introduction to the area of database theory is clearly beyond the scope of this thesis, c.f. the two volume introduction of Ullman (Ullman (1988); Ullman (1989)). Therefore, this introduction is directed towards our main field of interest, viz. modelling 'real world semantics'. In particular, we discuss what kind of information a database should contain.

After this general introduction, we re-introduce the well-known relational model. Many deficiencies of the relational model are known. In our survey we restrict ourselves, by focusing on representation semantics. In other words, we show that the relational model is not rich enough to give an adequate representation of the Universe of Discourse (UoD). This deficiency of the relational model is illustrated from two viewpoints, viz. database theory and information systems design.

In database theory, the relational model is extended with dependencies as an extra means to describe real world semantics. Moreover, the definition of dependencies on a schema influence the description of a schema, in that dependencies force normalisation of the tables. Normalisation is necessary to avert anomalous behaviour of the database with regard to updates. The term normalisation suggests that a schema with a set of dependencies can be brought into a unique normal form. Clearly, one of the requirements to formulate such a normal form is that given a set of dependencies D, all dependencies that are logically entailed by D should be derivable. It is well-known in database theory, that for an axiomatisation of derivability the dependencies should either be restricted to a small subset of all possible dependencies or all domain independent dependencies should be considered.

The large class of dependencies suffer from the problem that many of these dependencies do not seem to have a natural real world interpretation. The small class for which an axiomatisation for derivability exists, suffers from the problem that it has to be restricted even further to allow a unique normal form.

Information systems design is the engineering counterpart of database theory. It is mainly concerned with problems encountered during actual database design. Some of the concepts this field has discovered as necessary for modelling are described and it is shown that these concepts do not really fit in the relational model.

The lack of concepts for modelling real world semantics can be seen as the main motivation for the research on which this thesis reports. Obviously, we are not the first to address this topic. Therefore, the introduction continues with a short description of the two main research directions in the area of data models, viz. object-oriented databases and deductive databases. Moreover, we give the relative merits of both directions.

The proponents of either stream believe that their direction is the 'right' direction. Rather than joining one of the two camps, we observe that both directions are to a large extend complementary. Therefore, we propose a unified approach. This search for a unified approach can be seen as the second motivation for this thesis.

A complete development of an unified approach is clearly to large a task to be completed within four man-years or one thesis. Therefore, we restrict ourselves to one aspect, viz. complex objects. Complex objects arose originally in the object-oriented database approach. Broadly speaking, one could say that a formalism has complex objects if it supports a set of orthogonal type constructions.

Transplanting complex objects from object-oriented databases to a unified formalism implies that there are a few requirements the formalism for complex objects should satisfy. At the end of the first chapter, this list is formulated as follows:

1) The type system should support the use of *equivalent* types.

2) The type system supports (multiple) inheritance.

3) The set of type constructors should at least contain the Cartesian product, the disjoint union, and a power type construction. Moreover, it should contain a construction that allows for the modelling of the dynamic aspects of the UoD.

4) Power types imply that 'values' may be sets of less structured 'values', thus the formalism should allow for the set-theoretic manipulations of such set-values.

5) The formalism supports object identity.

6) The formalism supports incomplete information.

7) The formalism supports the use of rules, both in the database and in queries.

8) The query language should be declarative.

The rest of this thesis is devoted to the development of such a formalism, nicknamed *Flock*. Clearly, a purely syntactic definition of such a formalism is not sufficient, as many of the requirements are semantical rather than syntactical. Therefore, a formal semantics of the formalism is needed. We have chosen to use Category Theory as a vehicle. Besides the authors fascination with this subject, there are several reasons why we feel that Category Theory is the adequate vehicle for the semantics:

1) The theory is constructive, this implies that the semantics of Flock are adequate as a specification of an implementation.

2) The theory is very fundamental; this means that we need only a few concepts, but these concepts are strong enough to develop, e.g., the Set Theory and the Logic Theory we need in this thesis. As a result we do not have to use an amalgamation of different mathematical theories to provide Flock with semantics.

3) The theory is graph-oriented, which gives a natural translation of our concept of a *type graph*. Moreover, the type constructors of Flock have a natural counterpart in Category Theory.

The drawback of the choice for Category Theory is reflected in the size of this thesis. As it is a relatively unknown field, we have to introduce many of its concepts and theorems along the way.

11.1.2 Chapter 2

Developing a new formalism implies that one is not satisfied with the already existing formalisms. To justify this opinion, the second chapter of this thesis is devoted to related work. The chapter starts with the object oriented direction. A

difficulty encountered in this field is that there is no real consensus on what constitutes an object oriented database. Only recently, an influential group of researchers published the so-called Object-Oriented Manifesto (Atkinson *et al.* (1989)) in which they supply a list of requirements such a system should meet.

The direct consequence of this lack of consensus is that there exists a plethora of formalisms for complex objects. So, a complete overview is outside the scope of this thesis. Therefore, we have chosen five formalisms, which in our view represent the major trends in this area. The five formalisms are:

1) The 'calculus for complex objects', defined by Bancilhon and Khoshafian (Bancilhon and Khoshafian (1986));
2) The IFO semantical database model, defined by Abiteboul and Hull (Abiteboul and Hull (1987); Abiteboul and Hull (1987));
3) The Logical Data model defined by Kuper (Kuper (1985));
4) The ψ-calculus defined by Ait-Kaci (Ait-Kaci (1984));
5) The O_2 data model from GIP Altair (Lecluse and Richard (1989))

The second section is a short introduction to the area of deductive databases. This area is much better understood and therefore, a shorter introduction is possible. In this section, two subjects are highlighted, viz. the logical reformulation of relational database theory and the, so-called, knowledge bases.

The first subject, which is introduced following (Reiter (1984)), reformulates relational database theory either as a model or a logical theory. This reformulation is done because the logical semantics provide a strong grip on problems such as incomplete information.

The second subject, knowledge base systems, is introduced by means of an example, viz. LDL (Naqvi and Tsur (1989)). There are two main motivations for this area:

1) The relational query languages are not Turing complete. Although this is on purpose, as this has proved instrumental in query optimisation, it is felt that the choice is too restrictive. Many reasonable queries, such as the transitive closure of a relation, are not expressible as a relational query. The desire for more expressive query languages which are still suitable for query optimisation is one of the motivations for knowledge base systems.
2) The other motivation is the incorporation of rules in a database. If data can be derived from other data in the database, only the rule to derive this data should be represented. The obvious reasons are that redundancy poses update problems and that redundancy increases the necessary memory. A nice side effect of storing the rules is that it enables reasoning over the data in the database.

Both motivations are highlighted in our description.

The third and final section is devoted to unifying proposals described in the literature. Just as we were not the first to discover the semantical inadequacy of the relational model, we were not the first to note the complementarity of object oriented and deductive databases. The actual unifying formalisms depend, of course, on the starting point of the researchers. One may start with a deductive database language and embed a formalism for complex objects. An example of this approach is LDL with ψ-terms (Beeri, Nasr, and Tsur (1988)). Or one could unify an existing logic programming language and an existing formalism for complex objects on a basis of equality. An example of this approach is Log In (Ait-Kaci and Nasr (1986a)). Or, one could define a logical data language which has constructs for complex objects. An example of this approach is COL (Abiteboul and Grumbach (1987)). Finally, one could also start with a formalism for complex objects and extend this into a logical programming language. This is the approach used in O-logic (Kifer and Wu (1989)). As before, this selection of formalisms reflects the knowledge and the bias of the author. We give a brief overview of these models in chronological order.

Whereas the first two sections of this chapter can be seen as an introduction to the areas of object-oriented and deductive databases, the last section obviously describes formalisms that are closely related with Flock. Therefore, we evaluate these formalisms against our list of requirements. Not surprisingly, we find none of these formalisms adequate (the lack of surprise is of course due to the fact that this thesis is actually written). Moreover, we argue that none of these formalisms is particularly suitable to be extended to meet our requirements.

The first motivation to dismiss some of these formalisms is that we believe that the complex object should be the central concept; as we are interested in complex objects. The remaining formalisms already have a rather complicated semantics, this makes it awkward to add concepts as subtyping or type equivalences. If one succeeds, the resulting semantics will most probably be rather opaque.

11.1.3 Chapter 3

The first two chapters were introductory in nature, the third chapter finally, is the starting point of the development of Flock. This chapter starts with the introduction of the concept of a *type graph*. Usually, the types of a language are defined by a set of basic types and a set of type constructions. However, we make two observations on the relational model that inspire the notion of a type graph. These observations are:

1) The relational operators may be used ad libertum on the tables of a schema. However, it is not guaranteed that the relation schema of the result of such a manipulation describes anything remotely familiar in the UoD.

2) Even if the resulting schema is meaningful, there may be several different ways to construct a table over this schema. In general, the different constructions will lead to different result tables. This means that there is semantics hidden in the way the table is constructed. Some constructions may be meaningful, while others are senseless.

Now, a type graph can be likened to a description of the meaningful relation schemas and the meaningful conversions between these relation schemas. The main difference is that rather than relation schemas, we allow arbitrary types and that instead of relational constructions, we allow arbitrary functions for the conversion.

After this motivation for a type graph, we show how its directed graph nature can be used to define well known concepts such as specialisation and generalisation. Moreover, exploiting the functional nature of the directed arcs of the type graph, we define the notion of *equivalent* or *synonymous* types. These concepts are central to Flock, as can be seen by requirements Flock has to meet.

After the introduction of the concept of a type-graph, the introduction to Flock begins with the definition of its type structure. This is the subject of the rest of this chapter. As is obvious from the beginning of this chapter, the type structure of Flock will be a type graph.

Usually, in the definition of a language a set of basic types, such as *Bool*, are pre-defined. In the definition of Flock, however, we make an exception to this rule. This means that we do not define a basic type graph, but rather let the database designer define this basic type graph herself. The reason is that in Flock the basic types represent the 'smallest' observable types in a UoD. Clearly, these 'atoms' cannot be defined uniquely over all UoD's.

Flock is provided with six type constructors, viz.:

1) Products, this construction can be likened to the array construction in conventional languages.

2) Co-products, this construction can be likened to the disjoint union provided in some languages.

3) Exponents, this constructs a function type. It will proof to be instrumental in the definition of methods in Flock.

4) Equalisers, this construction does not have a counterpart in conventional languages. Its interpretation can best be sketched as a selection.

5) Power types, this is the usual construction that allows attributes to be tables in NF^2 data models.

6) Co-equalisers, this is perhaps the most difficult operator. Its main use lies in the fact that it allows us to construct *equivalence types*. Hence, it is instrumental to meet our requirements.

Note that these constructions do not only add nodes to the type-graph, but may also add new arcs.

All the constructions are introduced accompanied by their set-theoretic interpretation. These set-theoretic interpretations can be used as the informal semantics of the constructions. To facilitate the step towards the formal semantics of the constructions, a universal property is proven for each of the constructors. Moreover, to introduce the reader to categorical reasoning, these universal properties are used to proof simple theorems. Most of these theorems pertain to type equivalence.

This chapter ends with a short introduction to category theory. Which is used to define the formal semantics of the type constructions of Flock.

11.1.4 Chapter 4

In the fourth chapter, we construct types whose elements are exactly the valid extensions of a relation schema over which a set of algebraic dependencies are defined. This exercise serves two purposes:

1) It illustrates the use of the constructions introduced in the third chapter.

2) It allows a comparison between 'relational semantics' and Flock-semantics

In the first part of this chapter, we show how the relational constructions *join* and *projection* can be defined within Flock. Using these constructions, the construction of the required types becomes straightforward. This means that Flock is at least as expressive as the relational data model with regard to the representation of real world semantics.

However, in Chapter 1, we have seen that dependencies may cause update anomalies. The equivalent Flock-types constructed in the first part are just as intuitive as their relational counterparts, i.e. although the elements of a type are sets of tuples, we cannot simply insert or delete a tuple and remain in the same type. Therefore, the remainder of this chapter is devoted to more intuitive translations. In other words, given a relation schema r with a set of dependencies D, we want to find a type t, such that:

1) The elements of t correspond to the extensions of r that satisfy D and vice versa.

2) The update semantics of t are natural.

The latter requirement is of course, rather vague. In this chapter, we have reformulated it as stating that the elements of t should consist of independent components. Independence roughly means that a modification of one component does not affect any of the other components.

This notion of independence suggests that the horizontal decompositions of relations over r satisfying D can be used to find such a natural translation. In

the second part of this chapter, we show that indeed, horizontal decomposition leads to such a natural translation under the assumption that we have a natural translation for functional dependencies and provided that D is restricted to conflict-free sets of multivalued dependencies.

The latter requirement shows that our requirement for natural translations coincides with the requirements for normalisation; provided we find a natural translation for functional dependencies. The functional dependencies reappear in Chapter 10.

Clearly, different criteria for natural update semantics will lead to different classes of permitted dependencies. A natural candidate would seem to be to allow both horizontal and vertical decompositions. We end this chapter by a short introduction in the theory of horizontal and vertical decompositions into independent components.

11.1.5 Chapter 5

In Chapter 3, we have shown how a type graph can be defined. In this chapter, we take the next step towards the definition of complex objects, viz. we define structure types and structured objects. The structured objects are complex objects for which we have subtyping and type-equivalence. Both concepts are defined using the terminology for type graphs developed in the first part of Chapter 3. More specifically, subtyping is defined using the specialisation/generalisation hierarchy, while type equivalence is defined using synonymity. Roughly spoken, structure types are defined as subgraphs of the type graph, while structured objects are elements of such a type.

The remainder of this chapter is devoted to the definition and study of subtyping and type equivalences. Again, roughly spoken, subtyping is defined using a subgraph relationship, i.e. $s < t$ if the graph of t is a subgraph of the graph of s. The ratio behind this choice is that each object of type s can be transformed into an object of type t by 'forgetting' certain nodes and edges.

This definition of subtyping implies that many structure types are in a subtyping relation because of the way in which the type graph is constructed. In a theorem, we exhibit some general subtyping rules, induced by the type constructions of Flock. To compare our notion of subtyping, we reformulate the famous subtyping rules of Cardelli (Cardelli (1988)), in Flock. It is then an easy corollary that each of these rules is induced by a type construction. In other words, Cardelli's rules are implied by our definition of subtyping.

Type equivalence is simply defined using isomorphisms or synonymous types. There are two cases of type equivalence: either the database designer explicitly gives the isomorphism or two types are equivalent by construction. The latter case is interesting from a theoretical point of view. Moreover it greatly enhances the 'flexibility' of the system. Some examples of types that are equivalent construction are already given in Chapter 3. In this chapter, we provide some more examples.

In the ideal situation, the system can decide whether two types are equivalent by construction or not. In other words, one would like to have a sound and complete axiom system that can be used to proof equivalence. The last part of this cahpetr is devoted to this, well-known, problem. It turned out to be an extremely diifcult problem. Therefore, we cannot give a complete axiom system. Instead we give axiom systems for special cases in which the number of type constructions is limited.

11.1.6 Chapter 6

Structured objects and structure types as defined in the previous chapter are not particularly adequate to represent methods and method types. In fact, it is well-known that anti-monotony in Cardelli's typing rule for exponential types 'blocks' the straightforward solution for method inheritance. As our subtyping rules entail Cardelli's rules, we cannot use this straightforward solution either.

So, the inheritance of methods requires that we find another way of looking at subtyping. As this is a semantical problem, the reader will not be surprised that the solution towards this problem is found using Category Theory. More in particular, we show that our definition of specialisation can still be used, but we have to define our semantics in a different category; for categorists, the alternative semantics are placed in $C^{\rightarrow}$ rather than in C. Moreover, we show that method types can be seen as the usual graph-structure in these alternative semantics.

The object-oriented ideal is that methods are inherited automatically. In other words, one would like that methods defined for some type s are automatically extended to all subtypes t of s. It is shown that in general this is in general impossible. Consider e.g. the types *employee* and *manager*, such that *manager* is a subtype of *employee*. Let *promote* be a method that promotes an *employee* to a *manager*. As managers are *employees*, this method should also be applicable to *managers*, however, it is not a priori clear what the result-type of this application should be; let alone what the result itself should be.

However, we show that in restricted cases automatic inheritance is possible. The restrictions are:

1) Only products are used in the type constructions.

2) The methods preserve the type of the objects.

In this case, a method can be inherited by assuming it acts as the identity on other components. To continue our example, suppose that a manager is defined as an employee with a budget, i.e. the type is *employee* $\times$ *budget*. The method *raise-salary* is inherited as the function *raise-salary* $\times$ id_{budget}.

Implicit in the discussion above, is the assumption that the structure types are known at the time that the methods and their types are defined. Therefore, we use the term 'second-order' types when we formalise the description above in the

last part of this chapter. This formalisation is rather complicated as we have to project our alternative semantics back to the usual semantics. The reason for this projection is that methods should act on structured objects; hence, these concepts should be defined in the same category. The last part of the formalisation is devoted to the semantics of the application of methods on structured objects

11.1.7 Chapter 7

Now that we have defined structured objects, methods and the application of structured objects, it is worthwhile to define a query language for structured objects. To give semantics to the query language, however, we need set-theoretic manipulations. Therefore, we need the intermediate Chapter 7. This chapter develops logic and Set Theory in a categorical framework. All the material in this chapter is well-known in the realm of Category Theory. In other words, we do not present any new material in this chapter. Rather we introduce those concepts that we need in the remainder of this thesis.

The main points the reader should have learned after reading this chapter or, indeed, may take for granted, are:

1) The logical operations can be embedded naturely in Category Theory. Although the logic does not have to be classical. In particular, we show that the law of the excluded middle does not have to hold.

2) Using the logic, Set Theory can be developed in a categorical framework. Clearly, if the logic is non-classical, neither is the Set Theory.

These main points imply that we can incorporate set-theoretic and logical manipulations in the syntax of Flock.

11.1.8 Chapter 8

Using the results of the last chapter, we define a query language, called SOQL, for databases consisting of structured objects. We start developing a query language before the definition of complex objects for two reasons:

1) As we have less concepts to deal with, the semantics of this query language are easier to define than those of a full-fledged query language.

2) Complex objects are built modularly from structured objects, similarly the query language for complex objects can be built modularly from SOQL.

SOQL is a calculus defined along the usual lines. The highlights of this calculus are:

1) It is a typed declarative language.

2) Set values can be manipulated using the set-theoretic operations.

3) Queries may be recursive, provided they are stratified.

As an aside, note that we do not support a *grouping*-operator in the query language as some other formalisms do. The reason for this ommission is simply that the equivalence relation on classes (the types of complex objects), as defined in Chapter 10, automatically entails grouping.

The semantics of non-recursive queries are defined along the usual lines. For recursive queries, however, we resort to a trick. Rather than developing the formal semantics of fixed points or unification within our categorical framework, we resort to an operational semantics given in (Abiteboul and Beeri (1988)). The motivation for these operational semantics is that a formal definition would require yet another extensive subset of Category theory.

This chapter is finished with a short note on how the semantics of recursive queries can be exploited to store rules in the database. The drawback of this solution is that the rules have to be invoked explicitly in a query. The solution to this problem can be given as soon as we have an object identity.

11.1.9 Chapter 9

In the real world, it often happens that one wants to store some information, although it is not yet complete. In database terminology, such information is known as incomplete information. The formalism we developed so far in this thesis cannot be used to represent incomplete information. Therefore, we develop the concept of *is*-objects and *is*-types from structured objects and structure types in this chapter.

Naively spoken, one could use a Set-theoretic encoding of incomplete information: simply group all the possible values in a set, and use this set as the value. Although this model is conceptually sound, it has some drawbacks if domains are large, let alone if they are infinite. Therefore, we use functions rather than sets, however, these functions can be seen as abstract representations of sets.

This functional representation is close to the representation of methods. So, one might expect that incompleteness causes similar problems as methods did. However, we show that a careful definition of *is*-types ensures that such problems do not appear.

Incomplete information puts an extra burden on the definition of methods. For example suppose that we have a method that assigns a new colour to cars and suppose that all colours are mapped to red, except red, which is mapped to blue. Now suppose that we have a car of which we know that it is not red. What colour does it have after the method has been applied? Clearly, we want the answer to be red rather than not blue. This means that the method should 'know' how to deal with incompleteness.

Finally, we extend the query language SOQL such that it can query databases consisting of *is*-objects. For structured objects, there were two possibilities, either an object belongs to the result of a query or it does not. For *is*-objects, there is the possibility of doubt. To illustrate this doubt, suppose that we know that a car is either red or green. Moreover, suppose that we query for cars that are either red or blue. Does our car belong to the result or not? To enable the user to specify exactly what she considers an answer, the query language is extended such that logical manipulations of the incompleteness is possible.

11.1.10 Chapter 10

In this chapter, we will merge the various threads of this thesis, such as dependencies, type equivalences, methods, objects, and queries. The result of this merging is the definition of *classes* and, finally, *complex objects*. Before we can begin this merging process, we first need to introduce the concept of an *object identity*; one of the main characteristics of object oriented database systems (Atkinson *et al.* (1989)), and the last requirement left on our list.

Usually, object identity is seen as a surrogate, i.e. a machine defined number, that is immutable to changes. In the first section, we give another definition of object identity based on the history of an object. The result is that object identity is defined as a set of expressions over the methods defined with the database. So, for the definition of methods, we assumed that the set of types was fixed, now we assume that the set of methods is fixed. Special attention is given to the question of how methods should be applied to such identities. The result of this analysis is yet again, that the method definition should be extended such that it also encodes this information. Moreover, we show how this definition of identity supports in a natural way the modelling of object sharing and *HAS-A*-relationships in general. Finally, we show how our notion of identity can be used to store rules in the database.

In the second subsection, we define constraints on object identities, by defining grammars the expression in an identity should satisfy. Moreover, we show how such constraints can be used to model both structural and temporal data dependencies in the database in a natural way.

In the fourth section, we extend the definitions of the third section to classes and complex objects. The construction of classes from identity types is similar to the construction of structure types from entity types. This implies that again type equivalence plays a major role in this definition. The grammars used to restrict the possible identities of a type, play a crucial role in these equivalences. Different from entity types, it is easy to show that the question whether two (restricted) entity types are equivalent is undecidable. This means that we can only offer some examples.

The examples are based on a grammatical formulation of the functional dependency. This is achieved by observing that a functional dependency can be seen as a dynamical dependency. Note that this result together with the results of Chapter 4 mean that we can finally give a natural translation for relation

schemas with sets of functional dependencies and conflict free sets of multivalued dependencies. The fact that we can finally model multivalued dependencies induce the examples: we simply use the inference axioms for multivalued dependencies.

The lack of a formal semantics for classes means that we cannot give formal semantics to their query language. So rather than defining a new query language in the last section of this chapter, we simply illustrate four aspects of such a query language, viz. value oriented queries, HAS-A relations, grammars and historical data.

11.1.11 Evaluating Flock

From the extensive summary above, we see that Flock more or less meets all the requirements we listed in the first chapter. Some requirements, notably the recursive queries and object identities are met by rather ad-hoc solutions. So, these subjects deserve further research in the future. Similarly, the results on type equivalence, are far from conclusive. For entity types, we could not give a sound and complete axiomatisation. The partial results reported in Chapter 5, deserve future attention. Although equivalence of classes is undecidable, incomplete axiomatisations are an interesting and useful field of study.

Other interesting problems are outlined in the next section.

11.2 Directions for future research

In this section, we list a set of important research questions not answered in this thesis. For the sake of clarity, each of the topics is given its own subsection. Each of these subsections contains a formulation of the question and some general remarks on how this problem could be attacked within the formalism developed in this thesis. Note that the order in which the topics appear bears no relationship to their importance. The order is more or less random, we have only tried to keep related topics near each other.

11.2.1 New type constructions

Basically, Flock has only six ways to construct a new type. This set is rather limited in that we cannot even define lists (which caused problems for the formalisation of identities). The solution to this problem can be found by going to a more abstract level of category theory. In particular, the type constructions of Hagino (Hagino (1987a); Hagino (1987b)) seem a promising starting point; see also the subsection on implementation.

As an aside, note that this more abstract view on type constructions will make the proof of type equivalences again more difficult; although pragmatic solutions are readily available. For example, we could simply state that $List(a) \equiv List(b)$ iff $a \equiv b$.

11.2.2 Function construction

In the course of this thesis, we have limited the functions that can be used to those functions that we could actually construct. Rather than trying to characterise these functions from a categorical point of view, we feel that it is better to exploit the relationship between higher order lambda calculus and topos theory as described in (Lambek and Scott (1986)).

11.2.3 Query Optimisation

Traditionally, query optimisation is an important subject within database theory. In this thesis, we have defined several query languages, but paid no attention to optimisation. Clearly, this deserves further study. Due to the functional nature of Flock, the Bird-Meertens formalism for program transformation (Meertens (1986)), seems a good starting point.

The aim of this formalism is to derive algorithms from a specification by algebraic transformations. This means that we can start with an obviously correct but inefficient algorithm and transform it by correctnes-preserving transformations to an efficient algorithm. But this is exactly the goal of query optimisation.

11.2.4 Implementations

This thesis only describes a formalism, it does not deal with its implementation. This does not mean that this formalism cannot be implemented, but it is simply caused by a lack of time. In fact one of the reasons to use category is its constructive nature. In (Rydeheard and Burstall (1988)), Category Theory is introduced by implementing the theory itself in a functional language. Hence, we could extend this code to implement Flock.

Another, perhaps more interesting approach, would be to use Hagino's Categorical Programming Language (Hagino (1987a)). This language can be seen as a categorical construction of a functional programming language. This language is even more economical then Flock, in that it has only one way to define new types. However, it is shown that this suffices to define all constructions used in Flock. If we compare this language with the work reported in (Rydeheard and Burstall (1988)) or Flock, it is less powerfull in that it does not work for an arbitrary category. However, *SET* is of course the most interesting category as far as databases are concerned.

Yet another possibility is given by (Dyckhoff (1985)) in which the author gives an implementation of a part of category theory in the Gotenborg Type Theory System. Moreover, this work could also be duplicated in any other implemented constructive type theory, such as Nuprl (al. (1985)). Such an approach exploits the intimate relationship between higher order lambda calculus and topos theory (Lambek and Scott (1986)) to which we already refered above.

11.2.5 Concurrency

The grammars introduced in Chapter 10, can be used as a weak means to control the events in the database. Clearly, it is not powerful enough to deal with full fledged concurrency. For this problem, one could extend Flock with e.g. process algebra similar to (Wieringa (1990)).

Another solution, which is closer in spirit to the work reported in this thesis, is given in (Monteiro and Pereira (1986)). In this report, the authors present a model of concurrency based on Sheaf Theory. Formalisms such as process algebra are observational in nature, i.e. all that is known of a system is that what can be seen by on outside observer. In contrast, the theory of Monteiro and Pereira is structural, i.e. it is concerned with the internal organisation of the system. Hence, this approach is close to our crude grammatical approach. The other reason why this approach seems to be the natural choice is that categories of sheaves are typical examples of toposes.

Chapter 12

References

In the following list of references we use, besides the usual abbreviations such as ACM, the abbreviations LNCS and OOPSLA. The former is an abbreviation of the Springer Verlag series Lecture Notes in Computer Science. The latter is the abbreviation of the proceedings of the ACM conference on Object-Oriented Programming Systems, Languages and Applications.

12.1 References

S. Abiteboul and V. Vianu (1985). 'Transactions and integrity constraints'. *Proc. 4th Symp. on Principles of Database Systems*: 193-204, 1985.

S. Abiteboul and S. Grumbach (1987). *COL*. INRIA RR 714, 1987.

S. Abiteboul and R. Hull (1987). 'IFO, a formal semantic data model'. *ACM Transactions on Database Systems* **12**(4): 525-565, December 1987.

S. Abiteboul and C. Beeri (1988). *On the power of languages for the manipulation of complex objects*. INRIA RR 846, 1988.

S. Abiteboul and P.C. Kanellakis (1989). *Object identity as a query language primitive*. INRIA RR 1022, 1989.

A.V. Aho, C. Beeri, and J.D. Ullman (1979). 'The theory of joins in relational databases'. *ACM Transactions on Database Systems* **4**(3): 297-314, 1979.

A.V. Aho and J.D. Ullman (1979). 'Universality of data retrieval languages'. *Proc. 6th ACM Symp. on Principles of Programming Languages*: 110-117, 1979.

H. Ait-Kaci (1984). *A Lattice-Theoretic Approach to Computation Based on a Calculus of Partially ordered Type Structures*. Ph.D. Thesis, Computer and Information Science, University of Pennsylvania, 1984.

Hassan Ait-Kaci and Roger Nasr (1986a). 'LOGIN: a logic programming language with built-in inheritance'. *Journal of Logic Programming* **3**: 185-215, 1986.

Hassan Ait-Kaci and Roger Nasr (1986b). *Residuation: A Paradigm for Integrating Logic and Functional Programming*. MCC Technical Report Number AI-359-86, October 1986.

Hassan Ait-Kaci (1986). 'An algebraic semantics approach to the effective resolution of type equations'. *Theoretical Computer Science* **45**: 293-351, 1986.

Hassan Ait-Kaci and Patrick Lincoln (1988). *LIFE, A Natural Language for Natural Language*. MCC Technical Report Number ACA-ST-074-88, MCC, ACA Program, Austin, Texas, March 24, 1988.

R.L. Constable. et. al. (1985). *Implementing Mathematics with the NuPRL Proof Development System*. Prentice Hall, 1985.

K. Apt, H. Blair, and A. Walker (1986). 'Towards a theory of declarative knowledge'. *Proc. Workshop on Foundations of Deductive Databases and Logic Programming*: 546-629, 1986.

W.W. Armstrong (1974). 'Dependency structures of database relationships'. In *Information Processing 74*, pages 580-583. North-Holland, 1974.

M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik (1989). 'The Object-Oriented Database Systems Manifesto'. *Proc. 1st International Conf. on Deductive and Object-Oriented Databases*: 40-57, 1989.

H. Balsters and R. de By (1989). Private communication. 1989.

Herman Balsters and Maarten M. Fokkinga (1990). 'Subtyping can have a simple semantics'. *Theoretical Computer Science (to appear)*, 1990.

Herman Balsters and C. de Vreeze (1990). *Power Types*. Research Report Twente University (to appear)., 1990.

F. Bancilhon and N. Spyratos (1981a). 'Update semantics of relational views'. *ACM Transactions on Database Systems* **6**(4): 557-575, 1981.

F. Bancilhon and N. Spyratos (1981b). 'Independent components of databases'. *Proc. 7th Conf. on Very Large Data Bases*: 398-408, 1981.

F. Bancilhon and S. Khoshafian (1986). 'A calculus for complex objects'. *Proc. 5th ACM Symp. on Principles of Database Systems*: 53-59, 1986.

F. Bancilhon (1988). 'Object-oriented database systems'. *Proc. 7th ACM Symp. on Principles of Database Systems*: 152-162, 1988.

C. Beeri, R. Fagin, and J.H. Howard (1977). 'A complete axiomatisation for functional and multivalued dependencies'. *Proc. of the ACM SIGMOD International Symp. on Management of Data*: 47-61, 1977.

C. Beeri, R. Fagin, D. Maier, A. Mendelzon, J. Ullman, and M. Yannakakis (1981). 'Properties of Acyclic Database Schemes'. *Proc. 3rd Symp. on Theory of Computing*: 355-362, 1981.

C. Beeri and M. Kifer (1986). 'Elimination of intersection anomalies from database schemes'. *Journal of the ACM* **30**, July 1986.

Catriel Beeri, Roger Nasr, and Shalom Tsur (1988). 'Embedding ψ-terms in a Horn-clause logic language'. *Proc. 3rd International Conf. on data and knowledge bases*: 347-358, 1988.

J. Biskup (1987). 'Open problems in database theory'. In J. Biskup, J. Demetrovics, J. Paredaens, and B. Thalheim (Eds.), *Proc. 1st Symp. on Mathematical Foundations of Database Systems.* LNCS 305, pages 246. Springer Verlag, 1987.

A.J. Bonner (1988). 'Hypothetical Datalog: complexity and expressibility'. In M. Gyssens, J. Paredaens, and D. van Gucht (Eds.), *Proc. 2nd International Conf. on Database Theory.* LNCS 326, Volume 326, pages 144-160. Springer Verlag, 1988.

A.J. Bonner (1989). 'Hypothetical Datalog: negation and linear recursion'. *Proc. 8th Symp. on Principles of Database Systems*: 286-300, 1989.

K.B. Bruce and G. Longo (1985). 'Provable isomorphisms and domain equations in models of typed languages'. *Proc. 7th Symp. on Theory of Computing*: 263-272, 1985.

Kim. B. Bruce and Guiseppe Longo (1988). 'A modest model of records, inheritance and bounded quantification'. *Proc. 3rd Annual Symp. on Logic in Computer Science* : 38-50, 1988.

Peter Bunemann, Susan Davidson, and Aaron Waters (1988). 'A semantics for complex objects and approximate queries'. *Proc. 7th ACM Symp. on Principles of Database Systems*: 305-314, 1988.

L. Cardelli (1988). 'A semantics of multiple inheritance'. *Information and Computation* **76**: 138-164, 1988.

A.K. Chandra (1988). 'Theory of database queries'. *Proc. 7th ACM Symp. on Principles of Database Systems*: 1-9, 1988.

P.P.S. Chen (1976). 'The entity-relationship model toward a unified view of data'. *ACM Transactions on Database Systems* **1** (1), March 1976.

W. Chen and D.S. Warren (1989). 'C-logic of complex objects'. *Proc. 8th ACM Symp. on Principles of Database Systems*: 369-378, 1989.

P.P.S. Chen (Ed.) (1980). *Entity-Relationship Approach to Systems Analysis and Design*. North Holland, 1980.

E.F. Codd (1970). 'A relational model of data for large shared data banks'. *Communications of the ACM*: 377-397, June 1970.

E.F. Codd (1972). 'Further normalization of the data base relational model'. In R. Rustin (Ed.), *Database Systems*, pages 33-64. Prentice Hall, 1972.

E.F. Codd (1979). 'RM/T: extending the relational model to capture more meaning'. *ACM Transactions on Database Systems* **4** (4): 397-434, 1979.

Cosmadakis and Papadimitriou (1983). 'Updates of Relational Views'. *Proc. 2nd Symp. on Principles of Database Systems*: 317-331, 1983.

Scott Danforth and Chris Tomlinson (1988). 'Type theories and object-oriented programming'. *ACM Computing Surveys* **20** (1): 29-72, 1988.

C.J. Date (1981a). 'Referential integrity'. *Proc. 7th International Conf. on Very Large Data Bases*: 2-12, September 1981.

C.J. Date (1981b). *An introduction to database systems, third edition*. Addison Wesley, 1981.

P. De Bra and J. Paredaens (1983). 'An algorithm for horizontal decompositions'. *Information Processing Letters* **17** (2): 91-95, 1983.

P. De Bra and J. Paredaens (1984). 'Horizontal decompositions for exceptions to functional dependencies'. In H. Gallaire, J. Minker, and J.M. Nicolas (Eds.), *Advances in Database Theory 2*, pages 123-144. Plenum Press, 1984.

P. De Bra (1986). 'Horizontal decomposition based on functional dependency set implications'. In *Proc. of the first ICDT*. LNCS 243, pages 157-170. Springer Verlag, 1986.

R. Dyckhoff (1985). *Category Theory as an Extension of Martin-Löf Type Theory*. CS 85/3, University of St. Andrews, 1985.

R. Fagin (1977). 'Multivalued dependencies and a new normal form for relational databases'. *Transactions on Database Systems* **2** (3): 262-278, 1977.

R. Fagin (1982). 'Horn clauses and database dependencies'. *Journal of the ACM* **29** (4): 952-985, 1982.

R. Fagin (1983). 'Degrees of acyclicity for hypergraphs and relational database schemas'. *Journal of the ACM* **30** (3): 514-550, 1983.

Robert W. Floyd (1987). 'The Paradigms of Programming 1978 Turing Award Lecture'. In R.L. Ashenhurst and S. Graham (Eds.), *1966-1985 ACM Turing Award Lectures the first twenty years*, pages 131-142. ACM Press, 1987.

H. Gallaire and J. Minker (Eds.) (1978). *Logic and Databases.* Plenum Press, 1978.

H. Gallaire, J. Minker, and J. M. Nicolas (Eds.) (1981). *Advances in Database Theory 1.* Plenum Press, 1981.

H. Gallaire, J. Minker, and J. M. Nicolas (Eds.) (1984). *Advances in Database Theory 2.* Plenum Press, 1984.

R. Goldblatt (1979). *Topoi The Categorical Analysis of Logic.* Nort-Holland, Amsterdam, 1979.

Georg Gottlob, Paolo Paolini, and Roberto Zicari (1988). 'Properties and update semantics of consistent views'. *ACM Transactions on Database Systems* **13**(4): 486-524, 1988.

R. Gurevič (1990). 'Equational theory of positive numbers with exponentiation is not finitely axiomatizable.'. *Annals of Pure and Applied Logic (to appear)*, 1990.

M. Gyssens, J. Paredaens, and D. van Gucht (1989). 'A grammar based approach towards unifying hierarchical data models'. *Proc. 1989 ACM SIGMOD International Conf. on Management of Data*: 263-272, 1989.

T. Hagino (1987a). *Categorical Data Types.* Ph.D. Thesis, University of Edinburgh, 1987.

T. Hagino (1987b). 'A typed lambda calculus with type constructors'. In *Proc. Conf. on Category Theory and Computer Science.* LNCS 283. Springer Verlag, 1987.

S.J. Hegner (1983). 'Algebraic aspects of relational database decomposition'. *Proc. 2nd Symp. on Principles of Database Systems*: 400-413, 1983.

S.J. Hegner (1984). 'Canonical view update support through boolean algebras of components'. *Proc. 3rd Symp. on Principles of Database Systems*: 163-172, 1984.

S.J. Hegner (1990). Relational Database Decomposition: Logical and Algebraic Foundations (to appear). 1990.

C.W. Henson and L.A. Rubel (1984). 'Applications of Nevanlinna Theory'. *Transactions of the American Mathematical Society* **282**(1): 1-32, March 1984.

Wilfrid Hodges (1987). 'What is a structure theory'. *The Bulletin of the London Mathematical Society* **19**: 209-237, 1987.

John E. Hopcroft and Jeffrey D. Ullman (1979). *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

Barry E. Jacobs (1985). *Applied Database Logic, vol. 1: Fundamental Database Issues.* Prentice Hall, 1985.

Michal Jaegermann (1978). 'Information storage and retrieval systems with incomplete information I'. *Fundamenta Informaticae* **2**: 17-41, 1978.

A. Kemper and M. Wallrath (1987). 'An analysis of geometric modelling in database systems'. *ACM Computing Surveys* **19**(1), March 1987.

Martin L. Kersten, Peter M.G. Apers, Maurice M.G. Houtsma, Eric J.A. van Kuijk, and Rob L.W. van de Weg (1988). 'A distributed, main-memory database machine: research issues and a preliminary architecture'. In Masaru Kitsuregawa and Hidehiko Tanaka (Eds.), *Database Machines and Knowledge Base Machines*, pages 353-369. Kluwer Academic Publishers, 1988.

Martin L. Kersten and Arno Siebes (1990). The grammatical datamodel; its algebra (to appear). 1990.

S.N. Khoshafian and G.P. Copeland (1986). 'Object Identity'. *OOPLSA '86*: 406-416, 1986.

M. Kifer and J. Wu (1989). 'A Logic for object-oriented logic programming Maiers O-logic revisited'. *Proc. 8th ACM Symp. on Principles of Database Systems*: 379-393, Philadelphia, 1989.

G.M. Kuper (1985). *The Logical Data Model: A New Approach to Database Logic.* STAN-CS-85-1069 (Ph.D. Thesis), Stanford University, Department of Computer Science, September 1985.

J. Lambek and P.J. Scott (1986). *Introduction to Higher Order Categorical Logic.* Cambridge University Press, 1986.

S. Mac Lane (1971). *Categories for the Working Mathematician.* Springer Verlag, New York etc., 1971.

C. Lecluse and P. Richard (1989). 'The O_2 database programming language'. *Proc. 15th Conf. on Very Large Data Bases*: 411-422, 1989.

Witold Lipski, jr. (1979). 'On semantic issues connected with incomplete information databases'. *ACM Transactions on Database Systems* **4**(3): 262-296, 1979.

G. Longo, A. Asperti, and R. Di Cosmo (1989). 'Coherence and valid isomorphism in closed categories; applications of proof theory to category theory in a computer scientists perspective'. In D.H. Pitt, D.E. Rydeheard, P. Dybjer, A.M. Pitts, and A. Poigné (Eds.), *Category Theory and Computer Science* LNCS 389, pages 1-4, 1989.

A. Macintyre (1981). 'The laws of exponentiation'. In C. Berline, K. McAloon, and J.P. Ressayre (Eds.), *Model Theory and Arithmetic.* Lecture Notes in Mathematics 890, pages 185-197. Springer Verlag, 1981.

T.S. Maibaum (1977). 'Mathematical semantics and a model for databases'. In B. Gilchrist (Ed.), *Information Processing 77*, pages 133-138. North-Holland, 1977.

David Maier (1983). *The Theory of Relational Databases.* Computer Science Press, 1983.

David Maier, Jacob Stein, Allen Otis, and Alan Purdy (1986). 'Development of an object-oriented DBMS'. *OOPSLA '86*: 472-482, 1986.

D. Maier, D. Rozenshtein, and D.S. Warren (1986). 'Window functions'. In P. Kanellakis (Ed.), *Advances in Computing Research 3*, pages 213-246. JAI Press, London, 1986.

E.G. Manes (Ed.) (1974). *Category Theory applied to Computation and Control.* LNCS 25. Springer Verlag, Berlin etc., 1974.

Leo Mark (1985). *Self-Describing Database Systems: Formalisation and Realisation.* Technical Report #1484, Department of Computer Science, University of Maryland, April 1985.

C.F. Martin (1973). *Equational Theories of Natural Numbers and Transfinite Ordinals.* Ph.D. Thesis, University of California at Berkely, 1973.

Yoshifumi Masunaga (1989). 'Object identity, equality and relational concept'. *Proc. 1st International Conf. on Deductive and Object-Oriented Databases*: 170-187, 1989.

L.G.L.T. Meertens (1986). 'Algorithmics towards programming as a mathematical activity'. In J.W. de Bakker, M. Hazewinkel, and J.K. Lenstra (Eds.), *Proc. CWI Symp. on Mathematics and Computer Science*, pages 289-344. North Holland, 1986.

A.O. Mendelzon (1984). 'Database states and their tableaux'. *ACM Transactions on Database Systems* **9**(2): 264-282, 1984.

J.J. Meyer, H. Weigand, and R. Wieringa (1989). 'A specification language for static, dynamic and deontic integrity constraints'. In J. Demetrovic and B. Thalheim (Eds.), *Proc. 2nd symposium on mathematical foundations of database systems.* LNCS 364, Volume 364, pages 347-366. Springer Verlag, 1989.

J. Minker (Ed.) (1988). *Foundations of Deductive Databases.* Kaufmann, 1988.

L.F. Monteiro and F.C.N. Pereira (1986). *A sheaf-theoretic model of concurrency.* CSLI-86-62, Center for the Study of Language and Information., 1986.

Shanin Naqvi and Shalom Tsur (1989). *A Logical Language for Data and Knowledge Bases.* Computer Science Press, 1989.

O.M. Nierstrasz (1985). 'Hybrid: a unified object oriented system'. *Database Engineering* **8**(4), 1985.

J. Paredaens, P. De Bra, M. Gyssens, and D. van Gucht (1989). *The structure of the relational database model.* EATCS Monographs on Theoretical Computer Science 17. Springer Verlag, 1989.

D.H. Pitt, S. Abramsky, A. Poigné, and D.E. Rydeheard (Eds.) (1985). *Category Theory and Computer Programming.* LNCS 240. Springer Verlag, Berlin etc., 1985.

D.H. Pitt, A. Poigné, and D.E. Rydeheard (Eds.) (1987). *Category Theory and Computer Science.* LNCS 283. Springer Verlag, Berlin etc., 1987.

Walter D. Potter and Robert P. Trueblood (1988). 'Traditional, semantic, and hyper-semantic approaches to data modelling'. *IEEE Computer*: 53-63, June 1988.

R. Reiter (1984). 'Towards a logical reconstruction of relational database theory'. In M.L. Brodie, J. Mylopoulos, and J.W. Schmidt (Eds.), *On Conceptual Modelling*, pages 191-233. Springer Verlag, 1984.

J.C. Reynolds (1984). 'Three approaches to type structure'. In *Mathematical Foundations of Software Development.* LNCS 173, pages 145-156. Springer Verlag, 1984.

Jorma Rissanen (1977). 'Independent components of relations'. *ACM Transactions on Database Systems* **2** (4): 317-325, 1977.

Mikael Rittri (1989). 'Using types as search keys in function libraries'. *Proc. of the 4th International Conf. on Functional Programming Languages and Computer Architecture*: 174-183, 1989.

D.E. Rydeheard and R.M. Burstall (1988). *Computational Category Theory.* Prentice Hall, New York etc., 1988.

H.A. Schmid and J.R. Swenson (1975). 'On the semantics of the relational data model'. *Proc. 1975 ACM SIGMOD International Conf. on Management of Data*, 1975.

E. Sciore (1981). 'Real-world MVD's'. *Proc. 1981 SIGMOD International Conf. on Management of Data*: 121-132, 1981.

E. Sciore (1982). 'A complete axiomatisation of full join dependencies'. *Journal of the ACM* **29** (2): 363-372, 1982.

A. Sernadas, C. Sernadas, and H.D. Ehrich (1987). 'Object-oriented specification of databases: an algebraic approach'. *Proc. 13th Conf. on Very Large Data Bases*: 107-116, 1987.

John C. Sheperdson (1984). 'Negation as failure'. *Journal of Logic Programming* **1**: 51-79, 1984.

E.H. Sibley (1976). 'The development of database technology'. *ACM Computing Surveys*: 1-5, March 1976.

A. Siebes and M.L. Kersten (1988). *A Functional Approach to Database Semantics.* CWI Report CS-R8804, Amsterdam, 1988.

J.M. Smith and D.C.P. Smith (1977). 'Database abstractions: aggregation and generalisation'. *ACM Transactions on Database Systems* **2** (2), June 1977.

S.V. Solov'ev (1983). 'The category of finite sets and cartesian closed categories'. *Journal of Soviet Mathematics* **22** (3): 1387-1400, 1983.

D. Tsichritzis and A. Klug (1977). *The American National Standards Institute/X3/SPARC DBMS Framework. Report of the Study Group on Database Management Systems.* AFIPS Press, Montvale NJ, 1977.

J.D. Ullman (1987). 'Database theory past and future'. *Proc. 6th ACM Symp. on Principles of Database Systems*: 1-10, 1987.

Jeffrey D. Ullman (1988). *Principles of Database and Knowledge-Base Systems 1.* Computer Science Press, 1988.

Jeffrey D. Ullman (1989). *Principles of Database and Knowledge-Base Systems 2.* Computer Science Press, 1989.

J.J van Griethuysen (Ed.) (1984). *Concepts and Terminology for the Conceptual Schema and the Information Base.* ISO TC97/SC5/WG3, 1984.

R.J. Wieringa and R.P. van de Riet (1988). 'Algebraic specification of object dynamics in knowledge base domains'. In *Proc. Working Conf. on the Role of Artificial Intelligence in Databases and Information Systems*, pages 346-371. North-Holland, 1988.

R.J. Wieringa (1990). *Algebraic Foundations for Dynamic Conceptual Models.* Ph.D. Thesis, (to appear). Department of Mathematics and Computer Science, Vrije Universiteit, 1990.

G.C. Wraith (1975). 'Lectures on elementary topoi'. *Model Theory and Topoi.* Lecture Notes in Mathematics 445, 1975.

Mihalis Yannakakis and Christos H. Papadimitriou (1981). *Algebraic Dependencies.* MIT/LCS/TM-193, February 1981.

C. Zaniolo (1989). 'Object identity and inheritance in deductive databases an evolutionary approach'. *Proc. 1st International Conf. on Deductive and Object-Oriented Databases*: 20-39, 1989.

Samenvatting

Het is de gewoonte dat bij een engelstalig proefschrift een nederlandstalige samenvatting wordt gevoegd. Gezien de grote hoeveelheid engelstalige vakliteratuur die mijn vakgenoten moeten verwerken, zullen zij geen moeite hebben de engelstalige samenvatting in het hoofdstuk Conclusies te lezen. Dat betekent dat ik de nederlandstalige samenvatting kan gebruiken om de niet-vakgenoten uit te leggen waar dit proefschrift over gaat. Om dit doel te bereiken, is deze samenvatting in twee secties verdeeld. In de eerste sectie leg ik uit waar dit proefschrift over gaat en geef ik de motivatie voor het onderzoek waar over wordt gerapporteert. In de tweede sectie behandel ik kort de verschillende hoofdstukken van dit proefschrift.

1 Motivatie

Om uit te begrijpen waar dit proefschrift over gaat, moeten we eerst weten wat gegevensbanken zijn. Want, hoewel het mischien niet uit de titel valt af te leiden, dit proefschrift gaat over gegevensbanken.

Een computer zou je een gegevensverwerkende machine kunnen noemen. Die gegevens kunnen uiteen lopen van de patienten-gegevens van een huisarts tot de meetresultaten van een experiment. Het grote voordeel van een computer boven een gewone rekenmachine is dat je niet alleen met die gegevens kunt rekenen, maar dat je die gegevens ook kunt opslaan. Dat betekent dat je die gegevens niet elke keer opnieuw hoeft in te voeren als je er mee wil werken, maar slechts hoeft aan te geven met welke gegevens je wilt werken. Zo'n verzameling gegevens wordt vaak een bestand of een gegevensbestand genoemd.

Iedereen die met een computer werkt, kan zijn eigen bestanden maken en met die bestanden werken. Als verschillende mensen met dezelfde gegevens werken, is het echter niet handig als zij hiervoor hun eigen bestanden gebruiken. Als bijvoorbeeld de verkoopafdeling en de crediteurenadministratie van een bedrijf twee dezelfde maar gescheiden bestanden hebben, zou het kunnen voorkomen dat deze twee afdelingen een verschillend idee hebben over hoeveel goederen een klant besteld heeft. Om dit soort problemen te vermijden, zouden deze bestanden geïntegreerd moeten worden. Zo'n geïntegreerd gegevensbestand heet een gegevensbank.

Het opzetten en bijhouden van zo'n gegevensbank is geen eenvoudig probleem, en dus ontstond er binnen de informatica het vakgebied 'gegevensbanken'. Het centrale probleem van dit vakgebied laat zich als volgt omschrijven:

> *Het efficiënt beheren van grote hoeveelheden persistente, betrouwbare en gedeelde gegevens.*

In deze volzin betekent persistent, dat de gegevens altijd behouden moeten blijven, ook als per ongeluk de stroom uitvalt of de programmatuur het opgeeft. Betrouwbaar houdt in dat als er zich een noodgeval heeft voorgedaan, de gegevens zoals de computer zich die herinnert precies diegene zijn zoals ze vlak voor de ramp waren. Gedeelde gegevens wil zeggen dat verschillende mensen er, eventueel tegelijkertijd, mee kunnen werken. Grote hoeveelheden betekent dat er zoveel gegevens kunnen zijn dat ze niet in het 'eigen geheugen' van de computer passen, en dat er van secundaire geheugen faciliteiten zoals banden of schijven gebruikt gemaakt moet worden. Efficiënt tenslotte, betekent dat er zo zuinig mogelijk met computergeheugen en computertijd moet worden omgesprongen.

Een groot gedeelte van het centrale probleem, zoals we het hierboven geschetst hebben is onafhankelijk van de specifieke toepassing waarvoor een gegevensbank gebruikt gaat worden. Persistentie en betrouwbaarheid bijvoorbeeld, zijn eigenschappen die elke gegevensbank zal moeten hebben. Deze onafhankelijkheid van de toepassing maakt het mogelijk zogenaamde *gegevensbankbeheerssystemen* of DBMS (deze afkorting komt van de engelse term DataBase Management System) te ontwikkelen. Het idee van zo'n DBMS is dat een specifieke gegevensbank uit een nauwkeurige omschrijving van zijn toepassing wordt gegenereerd. Met andere woorden, het DBMS zorgt er automatisch voor dat bijvoorbeeld persistentie en betrouwbaarheid gewaarborgd zijn. Het grote voordeel van zulke beheerssystemen is dus dat de ontwerper van een gegevensbank geen specifieke kennis hoeft te hebben op het gebied van persistentie en betrouwbaarheid maar zich alleen hoeft toe te leggen op het nauwkeurig omschrijven van het toepassingsgebied. Immers, de andere problemen worden door het DBMS geregeld.

Om de ontwerper de mogelijkheid te geven zijn toepassingsgebied nauwkeurig te omschrijven, moet het DBMS een taal begrijpen waar de ontwerper zich in kan uitdrukken. Deze taal legt ondermeer vast hoe de gegevens eruit kunnen zien. Het onderdeel van de taal waarin de algemene vorm van de gegevens is vastgelegd noemt men vaak het *gegevensmodel* van het DBMS. Op het moment is

het populairste gegevensmodel ongetwijfeld het *relationele model*, waarin een gegevensbank als een verzameling tabellen wordt beschouwd. Een voorbeeld van zo'n tabel is:

Crediteurenboek		
Klantnaam	Ordernummer	Orderbedrag
Jansen	415152	25.67
Johnson	986986	765.09
Jansen	415653	285.47

Aan dit voorbeeld valt te zien dat het relationele model bijzonder geschikt is voor administratieve gegevens. Dit is niet verwonderlijk als men weet dat de administratieve automatisering van oudsher de belangrijkste afnemer van DBMS'en is.

Behalve de mogelijkheid om tabellen te specificeren, biedt het relationele model mogelijkheden (operatoren) om met de inhoud van tabellen te manipuleren. Als de gegevensbank bijvoorbeeld ook nog een tabel met de adressen van de klanten bevat:

Klantenboek	
Klantnaam	Adres
Jansen	Amsterdam
Johnson	London

dan kan men deze tabellen aan elkaar plakken met behulp van de *join operator*. Het resultaat is dan:

Rekeningenboek			
Klantnaam	Adres	Ordernummer	Orderbedrag
Jansen	Amsterdam	415152	25.67
Jansen	Amsterdam	415653	285.47
Johnson	London	986986	765.09

De join en de andere relationele operatoren worden bijvoorbeeld gebruikt in zogenaamde vraagtalen waarvan SQL (deze afkorting komt van het engelse Structured Query Language) de bekendste is. Een vraagtaal is een taal waarmee een gebruikster de gegevensbank kan doorzoeken naar gegevens die voor haar van belang zijn. Zo kun je in SQL bijvoorbeeld vragen naar alle klanten die in Amsterdam wonen en een order boven de duizend gulden hebben.

Het laatste aspect van het relationele model wat voor ons van belang is zijn de *constraints*. Constraints kunnen gebruikt worden om extra invloed op de inhoud van een tabel uit te oefenen. Als we bijvoorbeeld in het Klantenboek *Klantnaam* tot *sleutel* verklaren, dan betekent dat dat er voor elke klantnaam maar één adres en woonplaats in de tabel mag zijn opgenomen. Vervolgens zouden we een constraint op het Crediteurenboek kunnen leggen die er voor zorgt dat er alleen maar gegevens over klanten in kunnen worden opgenomen waarvan ook het adres bekend is.

Zoals we hierboven al gezegd hebben, is het relationele model op het moment veruit het populairste gegevensmodel. Dat wil zeggen dat de meeste DBMS'en die nu verkocht worden op dit model gebaseerd zijn. Er komen echter barsten in dit bolwerk doordat steeds duidelijker wordt dat het relationele model zo z'n gebreken heeft. Deze gebreken komen niet alleen in administratieve toepassingen naar voren, maar vooral ook in nieuwe toepassingsgebieden. Een voorbeeld van zo'n nieuw toepassingsgebied is de opkomst van CAD/CAM systemen. Dit zijn systemen waarmee nieuwe produkten ontworpen en later geproduceerd kunnen worden. Het ontwerpen gebeurt dan helemaal via de computer: in plaats van een tekentafel gebruikt de ontwerper het beeldscherm. In de gegevensbank kunnen dan bijvoorbeeld de verschillende stadia van het ontwerp worden opgeslagen. Ook kunnen de verschillende deelontwerpen apart worden opgeslagen. Het moge duidelijk zijn dat tabellen niet de handigste manier zijn om de ontwerptekeningen van bijvoorbeeld een auto in op te slaan.

De tekortkomingen van het relationele model hebben een hernieuwd enthousiasme veroorzaakt in het onderzoek naar gegevensmodellen. De belangrijkste stromingen op dit moment zijn ongetwijfeld de Object Georiënteerde Datamodellen en de Deductieve Datamodellen. We zullen deze twee richtingen illustreren aan de hand van problemen die zij trachten op te lossen. We beginnen met de object georiënteerde richting.

Een van de problemen van het relationele model is dat het alleen maar *platte* tabellen kent. Om een voorbeeld te geven, de Rekeningboektabel die we boven hebben gegeven kan niet worden samengevat in de volgende tabel:

Rekeningenboek			
Klantnaam	Adres	Ordernummer	Orderbedrag
Jansen	Amsterdam	415152	25.67
		415653	285.47
Johnson	London	986986	765.09

De reden hiervoor is dat er alleen maar zogenaamde *atomaire* waarden in een kolom mogen worden opgenomen. Met atomaire waarde wordt bijvoorbeeld een plaatsnaam of een ordernummer bedoeld. En in de kolom ordernummer in de tabel hierboven staan *samengestelde* waarden; d.w.z. achter Jansen staan twee

ordernummers.

Dit betekent dat gegevens onnodig vaak herhaald opgenomen moeten worden. Deze redundantie van gegevens kan fouten in de hand werken omdat bijvoorbeeld deze gegevens altijd precies hetzelfde moeten blijven. De oplossing die de object georiënteerde gemeenschap voorstelt is dan ook (simpel gezegd) dat er ingewikkeldere tabellen moeten worden toegestaan. De 'regels' in zo'n (ingewikkelde) tabel worden vaak *complexe objecten* genoemd.

Een daaraan gerelateerd probleem van het relationele model is dat het geen *overerving* kent. Stel je voor dat we al een tabel hebben gemaakt waarin we de gegevens over *personen* opslaan; de kolommen van zo'n tabel zouden bijvoorbeeld *Naam*, *Adres* en *Woonplaats* kunnen heten. Als we nu een tabel voor *personeelsleden* gaan maken, dan moeten we zelf zorgen dat alle personeelsleden ook in de personentabel voorkomen. Het zou veel makkelijker zijn als we de tabel personeelsleden konden opbouwen vanaf de personentabel; immers, dan zijn personeelsleden automatisch personen. We zouden dit overerving van eigenschappen of overerving van structuur kunnen noemen.

Om het andere belangrijke aspect van overerving uit te leggen, nemen we aan dat we een programma'tje *verhuizing* hebben geschreven. Dit programma zorgt ervoor dat als een persoon verhuisd, we haar adres kunnen veranderen in de personentabel. Overerving, betekent nu ook dat het programma *verhuizing* automatisch ook werkt voor personeelsleden. Dit betekent dat we geen nieuw programma *verhuizing-personeelsleden* hoeven te schrijven. Dit soort overerving zouden we overerving van programma's kunnen noemen. Beide aspeceten van overerving komen niet voor in het relationele data model maar wel in object georiënteerde modellen.

Een van de sterke punten van deductieve gegevensmodellen is dat zij onvolledige informatie kunnen verwerken, iets wat het relationele model niet kan. Als een bedrijf een nieuw personeelslid aanneemt, kan het gebeuren dat nog niet alle gegevens over haar bekend zijn. Het kan bijvoorbeeld gebeuren dat er nog over haar salaris onderhandeld moet worden. In zo'n geval weten we bijvoorbeeld wel al dat het salaris tussen de vier en de vierenhalfduizend zal liggen, maar nog niet het exacte bedrag. Met een duur woord noemen we dit een voorbeeld van onvolledige informatie. Zoals al gezegd, in het relationele model, kunnen we onvolledige informatie niet opslaan, in deductieve gegevensmodellen wel.

Het tweede sterke punt van deductieve gegevensbanken kunnen we het beste illustreren aan de hand van de volgende tabel:

Ouder	Kind
Jan	Marie
Jan	Piet
Marie	Klaas
Kees	Klaas
Klaas	Henk

Als we willen weten wie de voorouders van Klaas zijn, dan is dat niet eenvoudig met een relationele gegevensbank uitvinden. De enige manier is om eerst naar de ouders van Klaas te zoeken, en dan naar de ouders van de ouders van Klaas enzovoort enzovoort. Voor een deductieve gegevensbank ligt dat heel anders. In zo'n geval hoeven we alleen maar het regeltje te geven waarmee je beslist of iemand een voorouder is van iemand anders; bijvoorbeeld:

> *X is een voorouder van Y, als X een ouder is van Y of als X een ouder is van een voorouder van Y.*

Het is zelfs nog mooier, we kunnen dit regeltje ook in de gegevensbank opnemen, en dan 'weet' de gegevensbank wat voorouder zijn betekent. Deze mogelijkheid om regeltjes op te nemen heeft er toe geleid dat deductieve gegevensbanken ook wel kennisbanken genoemd worden.

Uit de bovenstaande voorbeelden blijkt al een beetje dat de object georiënteerde en de deductieve gegevensbanken verschillende problemen van het relationele model proberen op te lossen. Dit betekent dat het de moeite waard is om te proberen deze stromingen samen te voegen. Dat is het probleem waar dit proefschrift over handelt. Nog precieser gezegd: wij beschrijven in dit proefschrift hoe complexe objecten passen in zo'n algemene gegevensbank.

2 Samenvatting

In de vorige sectie hebben we aangegeven waarom we aan dit onderzoek zijn begonnen, en wat het doel ervan is. In deze sectie beschrijven we per hoofdstuk wat het onderwerp ervan is.

2.1 Hoofdstuk 1

Het eerste hoofdstuk van dit proefschrift is een uitgebreide motivatie van het onderzoek. Dit gebeurt op dezelfde wijze als in de vorige sectie; alleen wat grondiger. Nadat we de tekortkomingen van het relationele model hebben laten zien, geven we een korte inleiding op object georiënteerde en deductieve gegevensmodellen (een uitgebreidere samenvatting is in hoofdstuk 2 te vinden). Na deze inleiding vergelijken we de twee manieren van aanpak en bespreken we de mogelijkheid om deze twee manieren samen te voegen. In het bijzonder geven we een lijst met eigenschappen die een formalisme voor complexe objecten moet bezitten om in zo'n algemeen (geunificeerd) gegevensmodel te passen. In dit proefschrift ontwikkelen we een formalisme, genaamd Flock (van het franse Formalisme pour Les Objects Complex), dat aan deze eigenschappen voldoet.

2.2 Hoofdstuk 2

In de eerste twee secties van het tweede hoofdstuk, geven we een uitgebreider overzicht van object georiënteerde en deductieve gegevensmodellen. Voor object georiënteerde gegevensmodellen is dat niet eenvoudig, omdat er nogal wat verschillende formalismen zijn voorgesteld. Daarom geven we een kort overzicht van een aantal van deze voorstellen. Om dit hoofdstuk niet al te groot te laten worden, beperken we ons bovendien tot wat deze formalismen over complexe objecten te zeggen hebben. Over deductieve gegevensmodellen is veel meer overeenstemming, daarom kunnen we hiervoor volstaan met een overzicht van een van de voorgestelde modellen.

Wij zijn natuurlijk niet de eerste om op te merken dat object georiënteerde en deductieve gegevensmodellen in principe samen te voegen zijn. Daarom geven we in de derde sectie van het tweede hoofdstuk een overzicht van een aantal formalismen die deze integratie trachten te realiseren. Vervolgens toetsen we deze formalismen aan de criteria die we in het eerste hoofdstuk hebben opgesteld. Aangezien we in dit proefschrift een formalisme willen ontwikkelen dat aan deze criteria voldoet, zal het niemand verbazen dat geen van de formalismen die we in dit hoofdstuk de revue laten passeren ongeschonden door deze test heenkomt.

2.3 Hoofdstuk 3

De eerste twee hoofdstukken zijn van algemene aard, in het derde hoofdstuk beginnen we aan de ontwikkeling van Flock. In het bijzonder definiëren we wat voor soort typen in Flock geconstrueerd kunnen worden; ruwweg betekent dit dat we zeggen wat voor soort tabellen in Flock gemaakt kunnen worden. Een formalisme in de informatica heeft veel weg van een taal. Het bestaat namelijk uit twee delen, een syntaxis en een semantiek. De syntaxis legt vast welke uitspraken tot het formalisme (de taal) horen. De semantiek legt vast wat deze uitspraken betekenen. Het verschil met een gewone (natuurlijke) taal als het Nederlands is dat de betekenis niet omschreven wordt in de taal zelf (denk aan het woordenboek), maar wiskundig wordt geformuleerd. In dit hoofstuk definiëren we dan ook niet alleen de syntaxis van de typen (tabellen), maar ook de semantiek.

2.4 Hoofdstuk 4

In de eerste sectie van deze samenvatting hebben we gezien dat het relationele model twee manieren kent om het toepassingsgebied te beschrijven. De eerste wordt gevormd door de tabellen; immers als een bedrijf geen klanten heeft zal het geen klantentabel hebben. En de tweede wordt gevormd door de constraints. De platte tabellen van het relationele model kunnen ook beschreven worden in

Flock. Als we dus de constraints van het relationele model ook in Flock kunnen beschrijven, dan weten we dat Flock minstens zo krachtig is als het relationele model. In feite weten we dan meteen dat Flock zelfs krachtiger is, omdat het niet moeilijk is voorbeelden te geven van zaken die wel in Flock, maar niet in het relationele model uitgedrukt kunnen worden.

De vertaling van de relationele constraints naar Flock is het onderwerp van het vierde hoofdstuk. In het eerste deel van dit hoofdstuk laten we zien dat elke relationele gegevensbank met een aantal constraints daarop gedefinieerd naar Flock vertaald kan worden. Een probleem met constraints in het relationele model is dat ze zo kunnen zijn dat je niet zomaar een regel uit een tabel kunt weghalen. De vertaling in Flock lijdt aan hetzelfde euvel. In het tweede deel van dit hoofdstuk, laten we zien dat als we de verzameling constraints beperken, we een veel natuurlijkere vertaling kunnen geven die niet aan dit euvel lijdt. Er is echter een voorwaarde waaraan voldaan moet zijn om deze vertaling te kunnen gebruiken. In hoofdstuk 10 laten we zien dat aan deze voorwaarde eenvoudig valt te voldoen.

2.5 Hoofdstuk 5

In de vorige sectie hebben we twee vormen van overerving beschreven, nl. overerving van structuur en overerving van programma's. In dit hoofdstuk definiëren we de overerving van structuur. Dit betekent dan dat een werknemer automatisch ook een persoon is. Dus als wij dan bijvoorbeeld vragen naar alle personen die ouder zijn dan 25, dan krijgen we ook alle werknemers die ouder zijn dan 25 als antwoord.

In de tweede helft van dit hoofdstuk houden we ons met bezig met een ander probleem, namelijk met *synoniemen*. In het dagelijks taalgebruik zijn twee woorden synoniem als ze (ongeveer) hetzelfde betekenen. De betekenis die synoniem in Flock heeft is hier nauw mee verwant. We zullen dit proberen te illustreren aan de hand van een voorbeeld.

Stel je voor dat een club van vogel-liefhebbers al haar waarnemingen in een gegevensbank wil opslaan. Dan kan het gebeuren dat de ene waarnemer graag per gebied opgeeft welke vogels zij daar gezien heeft, terwijl een ander juist per vogel wil opgeven in welke gebieden die gezien is. Het moge duidelijk zijn dat een tabel van vogels per gebied makkelijk om te zetten is in een tabel van gebieden per vogel en omgekeerd. Zulke in elkaar overvoerbare tabellen noemen we synoniem.

In veel gegevensmodellen moet je kiezen of je de 'vogels per gebied-tabel' of juist de 'gebieden per vogel-tabel' wilt hebben. Als je deze keuze eenmaal hebt gemaakt, ligt deze vast. Als er voor de ene tabel is gekozen, en iemand wil juist de andere tabel zien, dan zal zij die tabel met de hand moeten maken. Het zou veel vriendelijker zijn als de computer dit soort omzettingen automatisch zou doen. Flock is een gegevensmodel waarin dit soort omzettingen automatisch gebeuren. In het tweede deel van dit hoofdstuk beschrijven we welke

automatische omzettingen mogelijk zijn.

De overerving van structuur samen met de 'automatische' synoniemen worden gebruikt om de eerste soort objecten te definiëren. Deze objecten heten *gestructureerde objecten.*

2.6 Hoofdstuk 6

In het vorige hoofdstuk behandelden we het overerven van structuur, in dit hoofdstuk behandelen we het automatisch erven van programma's. We beginnen het hoofdstuk met uit te leggen wat de moeilijkheden van dit probleem zijn. Om een tip van de sluier op te lichten, zullen we ook hier een voorbeeld geven.

Stel je voor dat we in onze gegevensbank werknemers en managers hebben, zodat elke manager ook een werknemer is. Precieser, stel je voor dat een manager een werknemer is die een afdeling toegewezen heeft gekregen. Vervolgens maken we een *promotie*-programma, dat gewone werknemers tot manager promoveert door ze een afdeling toe te wijzen. Nu is elke manager ook een werknemer, dus in principe zou *promotie* ook voor managers moeten werken. Alleen is er dan een probleempje: de manager heeft immers al een afdeling toegewezen gekregen. Betekent promotie dat zij een nieuwe (belangrijkere) afdeling krijgt, of betekent het dat ze er een afdeling bijkrijgt?

Uit dit voorbeeld blijkt dat het lang niet altijd mogelijk is om automatisch een programma te erven. In dit hoofdstuk beschrijven we eerst wiskundig wat het betekent dat een programma geërfd wordt, en daarna geven we een aantal gevallen waarin automatische erving mogelijk is. Tenslotte gebruiken we de erving om *methoden* te definiëren. Ruwweg gezegd, is een methode een programma samen met alle manieren waarop dit programma geërfd is. Dus in ons voorbeeldje boven, zou een methode *promotie* bestaan uit het programma promotie voor werknemers samen met het programma promotie voor managers.

Tenslotte definiëren we wat het toepassen van een methode op een gestructureerd object betekent.

2.7 Hoofdstuk 7

Hoofdstuk 7 is een technisch hoofdstuk. Het bestaat uit een inleiding tot een soort wiskunde die we verderop in het proefschrift nodig hebben. In het bijzonder hebben we deze wiskunde nodig om onvolledige informatie te beschrijven en om een semantiek van de vraagtaal van Flock te kunnen definiëren.

2.8 Hoofdstuk 8

Gestructureerde objecten zoals gedefinieerd in Hoofdstuk 5 zijn nog wel geen complexe objecten, maar toch zou je je kunnen voorstellen dat je gegevensbanken met gestructureerde objecten hebt. Als je een gegevensbank hebt wil je er ook een vraagtaal bij hebben. Immers, een gegevensbank heeft pas zin als de gebruikers er ook informatie uit kunnen halen. Het ontwerpen van deze vraagtaal is het onderwerp van dit hoofdstuk.

Je zou natuurlijk kunnen zeggen, definieer eerst maar eens complexe objecten, en definieer dan pas een vraagtaal. Echter, gestructureerde objecten zijn een bouwsteen waar complexe objecten mee gebouwd worden. Evenzo kan een vraagtaal voor gestructureerde objecten als bouwsteen gebruikt worden voor een vraagtaal voor complexe objecten. Dus het heeft duidelijk didactische voordelen om de vraagtaal voor gestructureerde objecten apart te definiëren.

Buiten dit didactische voordeel is er ook een technisch voordeel: gestructureerde objecten hebben een eenvoudigere structuur dan complexe objecten. Dit betekent dat een vraagtaal voor gestructureerde objecten makkelijker te definiëren is dan een vraagtaal voor complexe objecten. Het voordeel later in het proefschrift is bovendien dat we deze simpele taal alleen maar hoeven uit te breiden. Met andere woorden, in latere hoofdstukken hoeven we ons alleen maar te concentreren op de nieuwe aspecten van de vraagtaal.

Uit deze discussie moge duidelijk zijn dat het ontwerp van zo'n vraagtaal het onderwerp is van het achtste hoofdstuk. Een korte beschrijving van deze taal geven, is vrijwel onmogelijk. Daarom volstaan wij voor deze samenvatting met de opmerking dat *regels* zoals die van voorouders in de vorige sectie in deze taal gebruikt kunnen worden.

2.9 Hoofdstuk 9

In de eerste sectie van deze samenvatting hebben we gezien dat een van de gebreken van het relationele model is dat je er geen onvolledige informatie in kwijt kan. In dit hoofdstuk laten we zien hoe dit soort informatie in Flock past.

De technische oplossing van dit probleem is te ingewikkeld om hier uit te leggen, maar we kunnen wel een indicatie geven. Stel je voor dat je van een auto niet precies de kleur weet, maar wel dat het of rood of groen moet zijn. Dan kun je natuurlijk gewoon in het kolommetje kleur 'rood/groen' invullen, en daar de betekenis 'of rood of groen' aan toe kennen. Als het aantal mogelijkheden groot wordt, zijn al die schuine streepjes wat onoverzichtelijk. Het is daarom handiger in plaats van streepjes een verzameling te gebruiken. Dus als we weten dat de auto rood of groen is, dan noteren we: { rood, groen }.

Deze verzamelingsnotatie is bruikbaar totdat je ook negatieve informatie wilt opslaan. Stel je voor dat je alleen maar weet dat een auto *niet* rood is. Dan moet je alle andere kleuren als mogelijkheid opsommen. Dat is vooral bezwaarlijk als

er in principe oneindig veel kleuren mogelijk zijn. Daarom gebruiken we in dit hoofdstuk een variant op de verzamelingsnotatie die dit probleem niet heeft.

Nadat we netjes hebben gedefinieerd hoe we onvolledige informatie moeten opslaan, bekijken we hoe methoden met onvolledige informatie moeten omgaan. Ook dit is een probleem waar je erg bij uit moet kijken. Wat moet er bijvoorbeeld gebeuren als we besluiten alle rode auto's paars te spuiten en we weten alleen maar dat een auto rood of groen is?

Nadat we dit probleem hebben opgelost, gaan we onze vraagtaal aanpassen aan onvolledige gegevens. Deze onvolledige informatie betekent dat we soms niet zeker weten of iets een antwoord op een vraag is of niet. Om een voorbeeld te geven nemen we maar weer even aan dat we een auto hebben waarvan we niet zeker weten of hij rood of groen is. Stel nu dat we vragen naar alle rode en groene auto's. Dan hoort onze auto erbij, want hoe het ook uit zal pakken, hij is altijd of rood of groen.

Als we echter vragen naar alle rode en alle blauwe auto's, dan weten we het niet zo zeker meer. Immers als de auto rood blijkt te zijn, dan hoort hij bij het antwoord. Maar als hij groen blijkt te zijn, dan hoort hij er niet bij. We sluiten het hoofdstuk af met te laten zien hoe je deze onzekerheid in de vraagtaal kunt inbouwen.

2.10 Hoofdstuk 10

In dit laatste echte hoofdstuk kijken we naar de laatste eigenschap die we nog missen, namelijk *object identiteit*. Object identiteit betekent dat een object meer is dan een verzameling waarden. Stel je bijvoorbeeld voor dat personen in de gegevensbank worden opgenomen door hun naam, adres en woonplaats op te slaan. Stel je nu voor dat iemand zowel van naam verandert als verhuist (door een huwelijk bijvoorbeeld). Dan veranderen alle gegevens die we over die persoon hebben opgenomen, toch is de persoon zelf niet veranderd.

De notie van object identiteit heeft zo zijn voordelen in een object georiënteerde omgeving. Om een voorbeeld te geven, als een echtpaar een kind krijgt, dan kunnen we de gegevens van het kind bij beide ouders noteren. Maar dan moeten we er wel opletten dat als we die gegevens bij de een wijzigen we dat dan ook bij de ander doen. Het zou veel handiger zijn als we het kind als zelfstandig persoon zouden opvoeren, en bij de ouders alleen maar de identiteit van het kind opnemen. Immers, de identiteit van een kind verandert nooit meer.

In dit hoofdstuk stellen we een mechanisme voor die identiteit voor wat gebaseerd is op de geschiedenis van dat object (immers iemands verleden verandert niet meer). Bovendien laten we zien dat het bijhouden van de identiteit op deze manier allerlei extra voordelen heeft. Zo kunnen we er bijvoorbeeld makkelijk voor zorgen dat iemand met maar één persoon tegelijk getrouwd kan zijn. Tevens laten we zien dat de voorwaarde die we in Hoofdstuk 4 nodig hadden eenvoudig te vervullen is met deze historische identiteit.

We sluiten dit hoofdstuk af met te laten zien hoe methoden met de identiteit moeten omgaan en tenslotte hoe je de identiteit in de vraagtaal kunt gebruiken.

2.11 Hoofdstuk 11

In dit hoofdstuk geven we een (engelstalige) samenvatting van dit proefschrift, en concluderen dat Flock redelijk voldoet aan de eisen die we er in hoofdstuk 1 aan gesteld hadden. Na deze positieve conclusie bespreken we nog kort een aantal open problemen die de moeite van een nadere bestudering waard zijn.

Curriculum Vitae

Arno Siebes werd op 27 juni 1958 te Gouda geboren. De eerste vier jaar van zijn middelbare schooltijd bracht hij door op de Petrus Canisius Mavo te Gouda. Nadat hij deze opleiding met succes voltooid had, vervolgde hij zijn middelbare schooltijd aan het Antonius College, eveneens te Gouda, waar hij in 1977 het Atheneum diploma behaalde.

Daarna is hij begonnen aan de wiskunde studie aan de Rijks Universiteit te Utrecht. In 1981 behaalde hij zijn kandidaats diploma met als bijvak Natuurkunde. In 1983 voltooide hij deze opleiding met het doktoraal diploma, hoofdvak Wiskunde en bijvak Wiskunde.

Per 1 januari 1984 is hij als systeem ingenieur in dienst getreden bij het rekencentrum van de Directie Automatisering der Rijksbelastingen. Hij werkte daar bij het project Databases van de afdeling Systemen.

Per 1 september 1985, is hij in dienst getreden van het CWI, als wetenschappelijk medewerker van het DAISY project van de afdeling AA. Sinds 1 september 1989 werkt hij aan het door NFI ondersteunde inter-universitaire ISDF projekt.