Lifestyles - A Paradigm for the Description of Spatiotemporal Databases

by

Dipl.-Ing. Damir Medak

A THESIS

submitted in partial fulfillment of the requirements
of the degree of
Doctor of Technical Sciences

submitted at the
Technical University Vienna
Faculty of Science and Technology

Advisory Committee:

Univ.-Prof. Dr. Andrew Frank
Department of Geoinformation, E127
Technical University Vienna

Univ.-Prof. Dr. Thomas Eiter
Computer Science Department, E184
Knowledge Based Systems Group
Technical University Vienna

Vienna, May 1999 .....

ABSTRACT

This thesis investigates operations affecting identity of objects in a spatiotemporal database, ubiquitous for future temporal geographic information systems (GIS).

Two different techniques to record change in temporal databases are compared: database versioning and object versioning. We show formally that these techniques are equivalent and use conceptually simpler model of database versioning for the further development.

The conceptual model of our database is based on the entity-relationship model. The complete temporal database is an append-only series of snapshots, each of which represents the state of the universe of discourse at a particular moment on the time scale. Each snapshot consists of a set of objects connected with relations.

Objects are metaphorically perceived as having life: an object has its birth or creation, its life or existence, its death or destruction. The central concept in the life of an object is its identifier, which is unchanged from the birth to the death of an object. Identifiers are system constructs and they are maintained by the database independently of the user.

Category theory and many-sorted algebras provide the formal background for this thesis. Executable algebraic specifications are written in a categorical, point-free style of functional programming using Gofer environment. Gofer is a dialect of the functional language Haskell. It supports many-sorted algebras by multi-parameter classes.

The major result is the formal model for a universal spatiotemporal database, capable of representing different classes of objects in a uniform way with respect to change in identity of objects. We propose a theory of lifestyles: algebras of operations affecting identity of objects. Lifestyles are compositions of basic operations: create, destroy, suspend, and resume. The mereological relation (is part of) is the most important relationship among objects that affects the existence of composite objects and their parts. The concept of suspending parts when composed into a whole abstracts the detail in hierarchical cognitive reasoning.

We stress the differences between two major groups of compositions: fusions (composed parts are destroyed and cannot be resumed) and aggregates (composed parts are suspended and can be resumed).

The theory of lifestyles is compared with the work of other authors on the topic of identity change. It is formally shown that our system is capable to represent all operations enumerated in other models, being at the same time conceptually simpler and more flexible.

Finally, we apply the theoretical apparatus on several categories of real world objects, ranging from the examples in physical domain (natural objects, movable artifacts, liquids, containers, living beings) to non-tangible objects in the social realm (partnerships, ownership rights, and administrative units). We show that metaphorical mappings between the physical and social domain are possible. The major benefit is the reusability of functions and concepts that can be explored in building interoperable temporal information systems.

ACKNOWLEDGMENTS

This page is dedicated to all who helped me in getting this work done. Writing a thesis is like rowing a boat in a river. I thank Professor Krešimir Čolić and Adrijana Car for showing me where the river source was and where my journey should start. Colleagues on the Department of Geoinformation provided an excellent, friendly working atmosphere for my first oarstrokes. Since the department was in constant flux during these years, it is difficult to mention the complete crew - thanks to all those brave mermaids and sailors who were always willing to help when I was in trouble.

The admiral of the fleet, Professor Andrew Frank, was - of course - a constant in this changing environment. Although with the highest rank in our metaphorical navy, he was generously navigating my boat through streams and waves, escaping numerous reeves (one should keep in mind that rowers are sitting with their backs to the direction of movement of the boat). I thank Professor Frank for his advice, encouragement, help and patience during the whole journey, even when continents and oceans appeared to be between us.

Special thanks goes to Gwen Raubal for bringing my English into shape to such extent that I could understand all navigating instructions and write a reasonably readable diary of my journey. Last but not least, I am grateful to my second advisor, Professor Thomas Eiter, who carefully examined my diary when my boat came close to the mouth of the river, entering the open sea of wisdom.

to my wife Jasna, with love

TABLE OF CONTENTS

TABLE OF FIGURES

1. INTRODUCTION

The motivation for this thesis is best explained by an old philosophical puzzle about change in identity. Our leading hypothesis is that the set of operations affecting object identity in the changing world is finite. The method of algebraic specification, based on category theory is used to prove the hypothesis. Scientific contributions are enumerated as well as the targeted audience. Finally, the organization of the rest of this thesis is presented.

1.1 Motivation

> *"On those who enter the same rivers, ever different waters flow."*
>
> Heraclitus (fr. 12)

Change has attracted the attention of philosophers since antiquity: Heraclitus raised the question of identity and persistence: under what conditions does an object persist through time as the same object?

The idea was described by Plutarch in his writings about the Greek hero Theseus. A paraphrased version of Plutarch's story is:

> Theseus started his voyage in a simple wooden ship. During the journey, he replaced the wooden planks of the ship with new ones, throwing the old planks over board. At the same time, another ship sailed parallel to the ship of Theseus. The sailors of the second ship were collecting the planks thrown by Theseus and using them to replace their own planks. Until the end of the journey, Theseus replaced all parts of his ship, and the escort ship consisted of all parts of the ship Theseus started the journey. *(Vita Thesei*, 22-23)

The puzzling questions are: Which of the two ships is identical to the original? If it is the second ship, when it got the new identity? Is it possible that both ships are identical to the original ship? Alternatively, could neither of them be identical to the original?

Zeno of Elea was another Greek philosopher who analyzed change, motion, and plurality of things in the world. He argued that the motion in a continuum is impossible. In his arguments against the idea that the world contains more than one thing, Zeno derived his paradoxes from the assumption that if a magnitude can be divided then it

can be divided infinitely often. In his most famous paradox, *Achilles*, Zeno claims that the slower when running will never be overtaken by the quicker; for that which is pursuing must first reach the point from which that which is fleeing started, so that the slower must necessarily always be some distance ahead (Hofstadter 1979).

A contemporary parallel example to the ship of Theseus is an old car. The owner of the car changes its parts until the car is completely renewed. His neighbor picks the old parts and assembles a "new" old car. One may think that renewing an old car would have not removed its identity. The authorities have standardized methods for identification of movable goods to enforce legality of possessing them. For example, a car is uniquely described by its engraved chassis number. As long as this number is preserved, the owner is free to replace other parts of the car given that the parts have proper origin and functionality. Several other parts are numbered too (e.g., the engine), but the chassis number is in most countries legally recognized as the valid identifier for the car as a whole.

Liquids represent a more complicated situation where temporal behavior is concerned. Ships, cars, and their parts, like all solid objects, have crisp, observable boundaries allowing easy identification. On the other hand, liquid objects change their shape in different containers on the slightest action. Liquids slip out through the smallest hole in a container because of gravity.

The identity of living beings is another puzzling example. It is known that human cells are regenerating and it is certain that we have completely new cells every several years. Nevertheless, an average human will claim that his identity is not changing while the cells are regenerating. The problem arises if a new human being would have been made from the removed cells in the same way as a new car is assembled from old parts. This gives an idea of how difficult a strict decision about temporal continuity in the real world may be. Indeed, the answers depend on the circumstances one asks.

We do not attempt to solve all dilemmas concerning the old puzzle. We are concerned with representations of the real world for constructing information systems, which can track change in identity of objects. We perceive the real world as a collection of distinguishable entities that have properties and are mutually connected by relations. For example, pieces of furniture in a room are entities, having color and weight as properties, and there is at least a simple spatial relation between the furniture and the room: the furniture is *in* the room. We are able to individuate all entities in the observed

part of reality. Even if two chairs had equal properties, we can tell one from another and cognitively assign different identities to each chair.

An information system is a formal model of a part of reality. In an information system, we have to store the theory that represents the model. A data model is a formal construction that describes representation of the real world in a database. Entities are represented by objects, entity properties are represented by object attributes, and identity is represented by identifiers.

Early GIS dealt only with spatially referenced information. Research was concentrated the great majority of its effort upon the spatial components of the data. Such atemporal GIS describe only one state of the data. Thus, expressiveness of GIS was reduced to a snapshot view of the selected phenomena: an update replaced and destroyed the previously stored data. Historical states were lost and could not be recovered. Such limitation was due to the lack of readily available hardware. The problem of storage and fast retrieving of data was dominating the efforts to build GIS software. On the other hand, many scientific disciplines, as potential GIS users, requested maintenance of various data closely related to time as well as to space. A number of scientific contributions dealt with the application of GIS in different fields, especially in history, land management and ecology. It was found that not all these needs could be fulfilled with a single general model. As the technology became mature enough to support new requests, it was obvious that we have to revise and improve the models behind it. In case of temporal GIS, a well understood concept of change is an ultimate goal.

Recently, Frank argued that the major impediment to broader usage of modern GIS is the lack of tools dealing with temporal information and processes in general (Frank 1998b). Capabilities to manage temporal information are necessary for decision-making support in answering essential political questions. The conceptual models underlying modern GIS seem inherently incapable of dealing with dynamic information.

1.2 Hypothesis

Many GIS applications enforce an ‘object’ view: features in the world are represented as objects with well-defined boundaries (Frank 1996). The real world is simplified to the representation of selected objects being important for a given context - a universe of

discourse. The universe of discourse is modeled as a series of sets of inter-related objects. Features have properties; in representation, objects have attribute values.

Every feature has an identity that distinguishes it from all other features. In a representation, the identity is represented as an identifier. An identifier may not be arbitrarily changed by the user. Operations affecting object identifiers define how the objects get, change or lose identifiers through time. The objects are grouped in algebras with respect to the applicability of operations affecting identifiers. Since the period between the appearance and disappearance of an object is called life, such algebras are called lifestyles.

Our central hypothesis is that operations affecting object identity form a finite set and operate in algebras, which we call lifestyles.

Other types of change include motion (change in location), deformation (change in shape), and other changes of attribute values. We are interested in change of identifiers only, and other types of change will be neglected in this thesis.

1.3 Approach

In order to prepare the framework for reasoning about change in identities of objects, we build the formal model of a temporal database consisting of objects and relations. Such database is a representation of the real world, which is assumed to consist of features or things having properties. The inevitable philosophical issues of existence or non-existence of objects in the real world are not treated in detail - the ontological discussion is left to philosophers.

The subject of this thesis is the change in identity of objects. In the real world, identity is a product of human cognition: we need a concept to tell one object from other objects, since lower levels of abstraction (e.g., the atomistic view) are not appropriate in everyday life. In a representation (in a database), identity is represented by an identifier. An identifier is given to an object when the object is created and it remains unchanged as long as the object exists.

The world is continuously changing. In the real world, the change happens in the valid or world time. In the database, the change is registered in the transaction or database time, and the world time may be stored as an additional value for each change. It is important to compare these two orthogonal dimensions of time and to investigate which one is important for the task in hand: tracking the change in identity of objects.

On the implementation level, the world could be represented in a temporal database in many different ways. Two prominent possibilities are storing a new snapshot of a complete database for every change (database versioning) and storing the new version of the changed object only (object versioning). A comparison of these two methods is important for implementation purposes, but not for the theoretical consideration.

Objects can merge with other objects or they can split, building new objects in both cases. This process is described by the operations affecting object identity. These operations are the rules for change in the life of objects. We propose the theory of lifestyles that says that the number of operations affecting object identity is small and that objects can be classified depending on applicability of these operations.

The change of object identity is the main topic of this thesis. An object is observed as if it (metaphorically) has a life: a stretch of time or a set of non-connected stretches of time during which it exists. In a spatiotemporal database, change is captured as the change in the attributes of database objects. The question when an object loses its identity and becomes another object depends on the application domain. For example, changing the color of a car does not change the identity of the car. Changing its chassis, however, may be sufficient to assign the different identity to the car.

1.4 Methodology

We use algebraic specifications for formalization. Algebras capture the coordinated behavior of operations that are applied to the same objects. Algebras can be combined enabling the reuse of already gained knowledge without additional effort. Simple operations are then combined into complex ones by functional composition. Algebras can be parameterized, that is, applicable to the range of different types.

Algebras are special case of more general category theory - mathematical discipline that abstracts operations to arrows and individual values to sets of values. The formalization of the problem is provided in a categorical setting, having the category of functions as the central category. Parameterization of data types enabled a high level of abstraction. The properties of categorical product are exploited for developing the point-free model - functions that do not name their arguments, leading to generality of application.

The prototype of an object-oriented temporal database is developed as an executable specification in Gofer, a dialect of the functional programming language

Haskell. This prototype is used as the framework for testing the semantic correspondence of the theoretical apparatus.

The entity-relationship model of a temporal database is formalized in functional language, and the proposed theory of lifestyles is built on top of it. Further extensions of various applications are provided as executable prototypes as well.

1.5 Contributions of the thesis

This work proposes a unifying core of rules for change in identity of objects. These rules are applicable to a wide range of databases for administrative systems, socio-economical and natural sciences, wherever the historical databases are needed. The major contributions are:

- a formal model of a universal temporal database is provided as an executable specification written in a categorical, point-free style of functional language, independent of implementation details for objects and object types;
- conceptual clarification among ontological primitives (in the real world) and database elements (in a representation of the real world) is given: things are mapped to objects, identity to identifiers, properties of things to attributes of objects;
- properties of identifiers (uniqueness, immutability, and non-reusability) are justified and preserved in the model;
- the transaction time is necessary for a temporal database to properly treat temporal links between object identifiers, whereas control over the valid time could be left to the user (asymmetry between transaction and valid time);

The theory of lifestyles as algebras that group the operations affecting the object identity is proposed. The change in composite objects is based on the mereological relation "part of", which is essential in human abstraction of complex objects (the details about parts are neglected as long as the whole is observed). The theory of lifestyles consist of the following traits:

- a minimal set of four primitive operations affecting object identifiers is proposed, and a small number of possible compositions is derived and verified using precondition/postcondition verification procedure;

- lifestyles - as algebras of the composed simple operations affecting object identity - characterize the behavior of large categories of objects, and therefore are an efficient way for modeling change in various domains;
- lifestyles are compared with other prominent categorizations of operations on object identity. The lifestyles framework is simpler than other models, yet at least equally powerful;
- specifically, aggregation of objects has two faces: temporal and atemporal. An aggregation is temporal if it changes the identities of objects involved. If none of the identities are changed, the relation between part and whole is treated as any other relation;
- a fundamental difference between aggregation and fusion lies in the different nature of the underlying basic operations: suspend and destroy;
- the transfer of lifestyles from material to non-material objects is transparent and metaphorical mapping between the appropriate object classes exists. This can be explored for a rapid development of temporal GIS. Different systems, sharing the same core of lifestyles operations, are suitable for interoperability.

Finally, algorithms and functional programs are provided as the proof that the transformation functions between two different ways of storing temporal databases are lossless. Thus, the conceptually simpler model of database versioning (changing snapshots) can be used for theoretical considerations, and object versioning (which is "cheaper" in terms of storage space) can be used for implementation of temporal databases.

1.6 Audience

This work is related to several disciplines. We concentrated on giving a viable model for designers of temporal databases on the conceptual level. Thus, this work is targeted to the researchers studying the following areas:

- designers of temporal databases in general and temporal GIS in particular find a general approach to the modeling of change in the identity of objects;
- designers of cadastral and other historical databases have a concrete implementation-ready specification for their applications;

- artificial intelligence is enriched with a formal model of an important domain of common-sense knowledge - treatment of change in a universe of discourse consisting of individual objects.

1.7 Organization of the thesis

Related work of other researchers is presented in Chapter 2. Philosophers, from Aristotle to modern ontologists, contributed to better understanding of change and categorization of real world phenomena. Artificial intelligence was the first discipline that tackled the problem of change with a formal apparatus - the situation calculus. The research in temporal databases and temporal GIS was intensified during the last two decades, and several categorizations of change in identity of objects were proposed. Finally, we introduce formal background of this thesis: category theory and standard functional notation, which is helpful in reading the two subsequent chapters.

The basic concepts for a spatiotemporal database are informally explained in Chapter 3. We start from the ontological assumption that the world consists of individual objects with properties. The elements of the database based on the entity-relationship model are informally explained. Two different ways of updating the database with new states are compared, and algorithms for transformations are proposed.

In Chapter 4, we introduce primitive operations affecting object identity, and verify their definitions with pre- and post-conditions. The primitive operations are composed in two ways. The first set of composition affects a single object; the second set affects several objects. Resulting operations are categorized into algebras - lifestyles. Two major lifestyles are aggregations and fusions. An informal comparison with categorizations of other authors is given.

The method and the tool of formalization - algebraic specifications written in the Gofer dialect of the functional programming language Haskell - are described in Chapter 5. Connections between the functional notation and the algebraic approach are explained. Standard functions that are used in the rest of the thesis are shown.

The formal model of a complete spatiotemporal database is given in Chapter 6. The elements of the entity-relationship model (object, identifiers, attributes, values, value sets, and snapshots) are abstracted as Haskell classes. Collections are parameterized to achieve generality and extensibility of the model - operations are defined independently

of the concrete representation. A simple implementation is given together with examples that show functionality of the model. Finally, the algorithms for transformation between two versioning techniques are formalized in functional notation.

The formalization of the operations affecting object identity is done in Chapter 7. The result is a small set of lifestyle classes that are completely independent of implementation issues. All operations are written in a categorical, point-free style. The formal comparison to categorizations of other authors is provided: all operations are represented as compositions of lifestyle operations.

A categorization of physical objects, based on contemporary research in cognitive linguistic, is proposed in Chapter 8. Each category is informally discussed and then formally defined within the lifestyles framework. Representations and examples are provided as the extensions to the model developed in previous chapters.

Selected examples from the abstract, non-tangible, social domain are described and formalized in Chapter 9. Metaphorical mappings from the physical to the social domain resulted in simple models and the code re-usability. The connections are represented as dependencies between appropriate classes in both domains.

Conclusions and directions for the future work are given in Chapter 10. The complete printout of the executable program in the functional programming language is available in the Appendix.

2. CONTRIBUTING DISCIPLINES

In this chapter, the research on the topic of change in different scientific disciplines is presented. The chapter is divided into the following sections: philosophical background of ontology and epistemology; the efforts of artificial intelligence (situation calculus, naive physics), research in temporal databases (time structure, temporal data models, query languages) and GIS (federated GIS, qualitative representations of change in GIS). Finally, we present the formal background for this thesis - category theory.

2.1 Ontology and representation of the real world

Ontology is the science of what is. Ontology as traditionally conceived is not a description of how we conceptualize the world, but rather a description of the world itself. This, of course, assumes that there is only one true reality to be described. An ontology is either an abstraction of the formal features that characterize all scientific areas (a formal ontology), or it is a statement of the necessary and sufficient conditions for something to be a particular kind of entity within a given domain (a material ontology). Theories that are correct descriptions of a given domain of objects allow us to infer the material ontology for that domain. By investigating what is shared by all material ontologies we can infer the principles of formal ontology (Smith 1999).

The science of ontology was grounded by Aristotle in his two works: Categories and Metaphysics. Aristotle divided the world into substances (things, or bodies) and accidents (qualities, events, processes). In the Aristotelian view substances exist on their own, where accidents require substances to exist; substances may remain numerically one and the same, admitting different accidents at different times; a substance is self-identical from the beginning to the end of its existence. It is not substances that can have temporal parts. The existence of a substance is continuous through time (Smith to appear).

According to John McCarthy, a representation is called ontologically adequate if the world could have that form without contradicting the facts of the aspect of reality that interests us (McCarthy and Hayes 1969). Examples of ontologically adequate representations for different aspects of reality are:

1. The representation of the world as a collection of particles interacting through forces between each pair of particles.
2. Representation of the world as a giant quantum-mechanical wave function.
3. Representation as a system of interacting discrete automata.

Ontologically adequate representations are mainly useful for constructing general theories. Deriving observable consequences from the theory is a further step and it is the realm of ***epistemology*** - science of knowledge and its representation.

A representation is called epistemologically adequate for a person or machine if it can be used practically to express the facts that one actually has about the aspect of the world. Thus, none of the above-mentioned representations are adequate to express facts as "John is at home", or "dogs chase cats" or "John's telephone number is 321-7850".

2.2 Artificial intelligence

Artificial intelligence was the first scientific discipline that approached the problem of change with the formalization tools. Artificial intelligence concentrates its efforts on situation calculus and naive physics. Both directions have the common goal: automation of reasoning in the everyday dynamic world.

2.2.1 Situation calculus

*Situation calculus***,** developed by McCarthy, is a first order language designed to represent dynamically changing worlds in which all changes are the result of named actions, (McCarthy 1957; McCarthy and Hayes 1969). A situation or a state is a snapshot of the world at a given moment. The world is conceived as being in some state ***s***, and this state can change only in consequence of some agent (human, robot, or nature) performing an action. If ***a*** is such an action, then the successor state to ***s*** resulting from the performance of action ***a*** is denoted by ***do(a,s)***. The actions have preconditions - sufficient conditions, which the current world must satisfy, before the action can be performed in this state. For example: a possibility of my walking out of a room presupposes that I am in the room and I can walk:

$$in((I, room), s) \wedge walk(I,s) \supset Possible\ (walkout\ (I, room), s).$$

The result of the action walkout results in a new situation in which "I am not in the room" is represented by the following effect axiom:

$$Poss(walkout(I,room),s) \supset \neg in((I,room), do(walkout(I,room),s)).$$

Beside effect axioms, which change the situations, there are axioms invariant in the change: so called frame axioms. For example, my weight does not change if I walk out of the room:

$$Poss(walkout(I,room),s) \wedge weight(I,x,s) \supset weight(I,x,do(walkout(I,room),s)))$$

Only relatively few actions will affect the truth-value of a relation; all other actions leave the relation invariant, and need many frame axioms. The difficulty to specify all such axioms is the so-called *frame problem*.

Recently, Reiter has applied the situation calculus for solving database problems (Reiter 1994). Reiter proposed the solution for the database version of the frame problem using mathematical induction based on the analogy between database states and natural numbers. In his new manuscript, (Reiter in preparation), he revives the situation calculus as a formalization method for artificial intelligence. He argues that a situation is not the same as a state; a situation is a temporal history, while a state is a snapshot.

Stephen Bittner applied Reiter's ideas to model changes and inconsistencies in a legal cadastre system (Bittner 1998). Starting with an initial situation, he proposed a set of axioms for action preconditions and a set of axioms for succeeding states that are allowed in a legal cadastre. Possible discrepancies between the true states in the real world and the rights registered in a legal cadastre are marked as inconsistencies. The formal model was developed as an executable Prolog program.

2.2.2 Naive physics

Naive physics was proposed with the goal of studying the human common sense knowledge about the everyday physical world (Hayes 1978; Hayes 1985b).

The goal of the CYC project, (Lenat et al. 1990), was to identify the core of common sense knowledge which would enable an artificial agent (robot) to act intelligently in real world circumstances. It came out that an immense collection of facts stored in the computer still could not compensate for the common-sense knowledge about the physical world of an average human being (Hobbs and Moore 1985).

Hayes challenged the situation calculus with the argument that mutually unrelated facts are irrelevant for representing actions (Hayes 1985b). Interactions between

physical objects need to be taken into account only when their histories overlap, both spatially and temporally. Hayes proposed that a basic ontological primitive should be a piece of space-time with natural boundaries, both temporal and spatial. He called these primitives *histories*. History of a two-dimensional spatial object is represented in Figure 2.1. Unlike a situation, a history has a shape; it is restricted spatially and extended temporally. Indeed, situations are themselves histories of a very special kind, being spatially unbounded and having temporal boundaries defined by the events between which they are fitted.

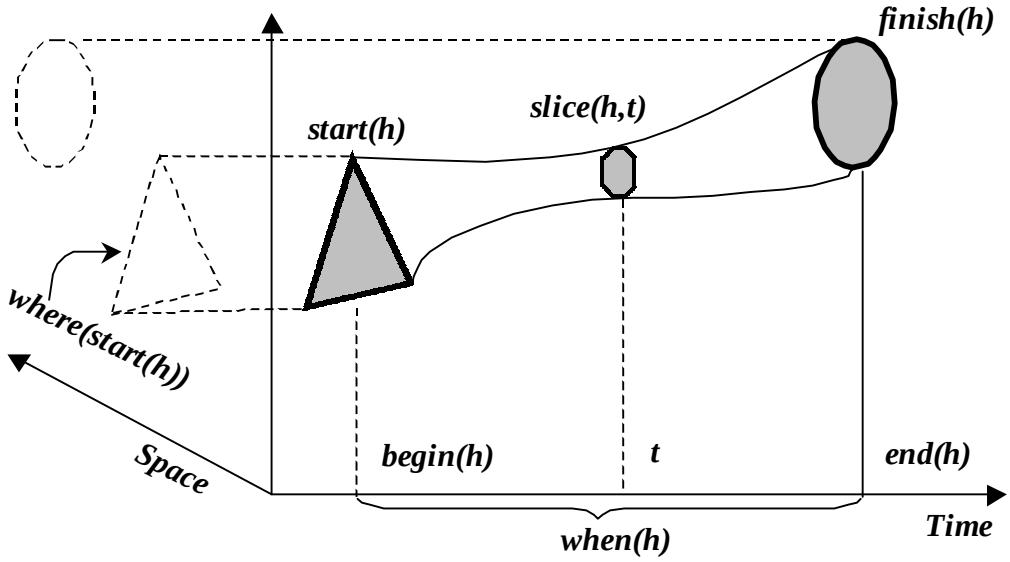

Figure 2.1: History of a moving and transforming 2D-object (Hayes 1985a).

Further, Hayes formalized the behavior of liquids (Hayes 1985a). In comparison with the formalization of solid objects, liquids posed a more difficult problem. Hayes tried to solve several problems coming from the strange properties of liquids: their merging, splitting, moving, disappearing. Each of these processes includes treatment of time and change. Hayes stressed that the individuation of liquids is much harder problem than the individuation of solid objects. The problem can be simplified if liquids are seen as liquid objects in contained space, but such representation does not capture the nature of liquids. Therefore, Hayes concluded that liquid objects exist as duals: enduring pieces of liquid and temporary liquid objects. An important conclusion was drawn about the proper approach: the world can be separated into smaller domains of interest, which are easier to formalize.

Recently, Kuipers formalized continuous change using differential equations, (Kuipers 1994). Kuipers approached the problem of change from the physicist's point of

view - using the language of differential equations for describing a system and drawing inferences about it. A differential equation represents the structure of the system by selecting certain continuous variables that characterize the state of the system, and certain mathematical constraints on the values those variables can take on. A set of continuous functions of time describes the way the variables of the system evolve over time starting from a given initial state.

2.3 Temporal databases

Database research concentrates on temporal dimensions, structure of time, and temporal query languages. A comprehensive survey of the development in relational temporal databases is given by Snodgrass (Snodgrass 1992). Snodgrass summarized the major concepts from application-independent DBMS support for time-varying information. He concludes that the semantics of the time domain, its structure and dimensionality is well understood. Many temporal query languages are currently proposed: Tquel (Snodgrass 1987), HQuel (Tansel 1986), Postgres (Stonebraker and Rowe 1986), HSQL (Sarda 1990), and TSQL2 (Snodgrass 1995b). While the query languages in relational databases are formal, object-oriented temporal query languages lack the formality (Snodgrass 1995a).

Time is multi-dimensional in a very particular sense. There is a consensus on terminological issues about two dimensions (Jensen and Dyreson 1998): *valid* time describes when an action happens in the modeled world, and *transaction* time describes when the information was entered in the database. The former is controlled by the changing agent, the latter by the database.

Temporal databases are classified according to the temporal domain they support. If neither transaction nor valid time is supported the database is *static*. A *rollback* database supports transaction time, but not valid time. It permits the entering of facts in a database, but only database time is stored with each fact. A *historical* database supports only valid time. It allows the entering of historical facts in a database without registration of the transaction time. Finally, a *bitemporal* database supports both valid and transaction time.

In all these variants, the user-defined time is not considered an attribute. In addition to these dimensions, several third temporal dimensions were proposed to capture

semantic details not covered by two-dimensional model: reference time (Clifford and Isakowitz 1994) and event time (Kim and Chakravarthy 1994).

Depending on which dimension of time is used, there are several choices in its representation. Time can be linear, branching or cyclic; discrete, dense or continuous; interval or point-based; absolute or relative; bounded or unbounded (Snodgrass 1992).

2.4 Time in GIS

Frank concluded that the discussion about different types of time resembles the discussion about different types of space in GIS (Frank 1998a). He proposed the taxonomy of types of time with lattice structure. Special attention was paid to different granularity of time in administrative systems. Temporal dimension is usually represented in metrical units: points or intervals on a time scale. In many application areas, however, the order of events is all that matters or that is known, giving rise to qualitative temporal reasoning. Frank gave the formalization of qualitative temporal reasoning in GIS (Frank 1994), based on Allen's interval work (Allen 1983). Worboys tried to amalgamate time and space in a simple bitemporal GIS, and concluded that there is a distinct asymmetry between the models of time and space (Worboys 1994).

2.4.1 GIS and administration

Human beings have invented and are still developing a complex set of abstract concepts such as land ownership, money, marriage, or government (Searle 1995). In particular, the concept of ownership is very important for modern GIS. The rights are tied to the land: if there is no land, there are no rights. This sort of dependence extends the set to the objects whose existence is dependent upon other objects. The main representatives in this group of entities are shadows and holes (Casati and Varzi 1994). The cadastre is an example of social construct attached to the concrete physical reality. The merging and splitting of cadastre parcels are crucial operations for proper functionality of land ownership information systems. Al-Taha presented a model of temporal reasoning in cadastre based on the extended relational databases (Al-Taha 1992).

2.4.2 Object identity in temporal GIS

Langran presented temporal database designs applicable to GIS applications (Langran 1989). In the relational database model, the change is captured by creating new versions. Langran compares three methods of database versioning in respect to GIS: table, tuple, and attribute versioning. She emphasized the recognition of different versions of a changing object as a fundamental problem for temporal databases. The semantic problem of what magnitude of change causes an entity to get a new identity and not another version of the old, depends on the application.

A similar problem emerges in an integrated GIS when several thematic databases, related to the same geographical domain, are used simultaneously, (Al-Taha and Barrera 1994). Namely, an old house can be represented as a residential area in the local cadastre database, and as a national monument in the state monuments database. Al-Taha and Barrera proposed the three criteria the identity must fulfill: uniqueness, immutability and non-reusability. Further, three capabilities for manipulating identities in a GIS were proposed: interrogation of a feature for its identity, using identity as a handle for the feature itself, and, finally, the comparison operation to decide if two identities correspond to the same feature. They also proposed the defining identity as an abstract data type that hides the actual structure of the identifying mechanism from the user and shows only relevant usage operations: for comparing identities, assigning one identity to a null element, constructing a null element, and for destroying the identity. The transaction time perspective was discussed only.

Finally, they enumerated a set of so called temporal constructs, that were already known from various sources, see Figure 2.2. The idea of temporarily suspending an object from a database (kill and reincarnate) was firstly presented by Clifford (Clifford and Croker 1988).

Claramunt and Thériault proposed a taxonomy of change for spatial entities that include deformations and movements (Claramunt and Thériault 1996). They divided the change according to the number of entities involved in three groups: evolution of a single entity, functional relationships between entities, and evolution of several entities, see Figure 2.3.

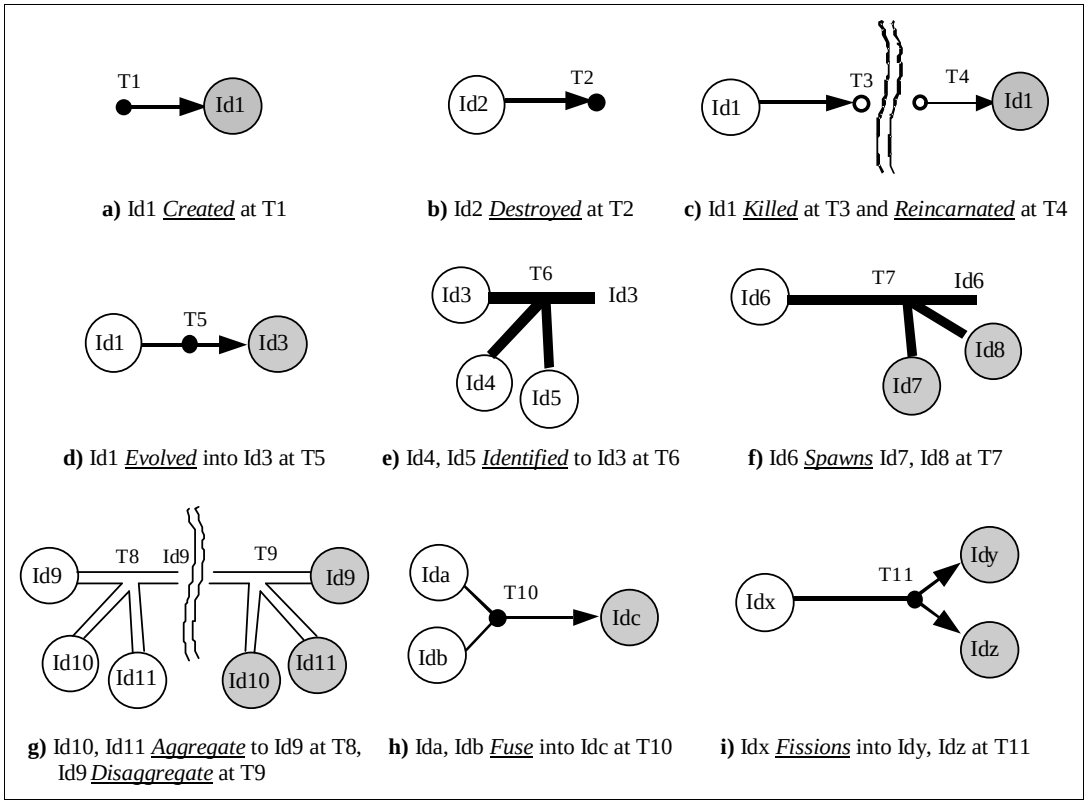

Figure 2.2: Temporal constructs of identities (Al-Taha 1994).

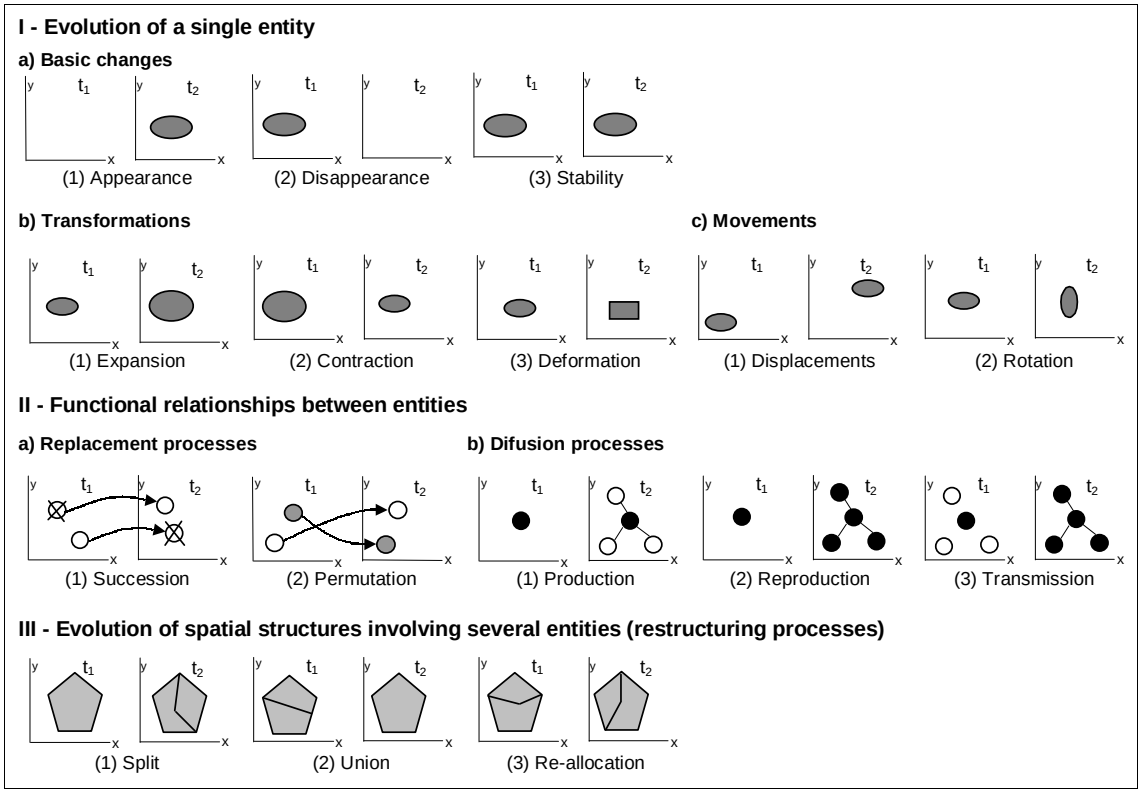

Figure 2.3: Typology of spatiotemporal processes (Claramunt and Thériault 1996).

In total, Claramunt and Thériault distinguished 16 different spatiotemporal processes. They stated that the range of phenomena that can be processed in a temporal GIS is probably inexhaustible.

Hornsby and Egenhofer discerned the following operations that either preserve or change object identity: create, destruct, reincarnate, issue, continue existence, continue non-existence, spawn, metamorphose, merge, generate, mix, aggregate, compound, unite, amalgamate, combine, separate, splinter, divide, secede, dissolve, select (Hornsby and Egenhofer 1997).

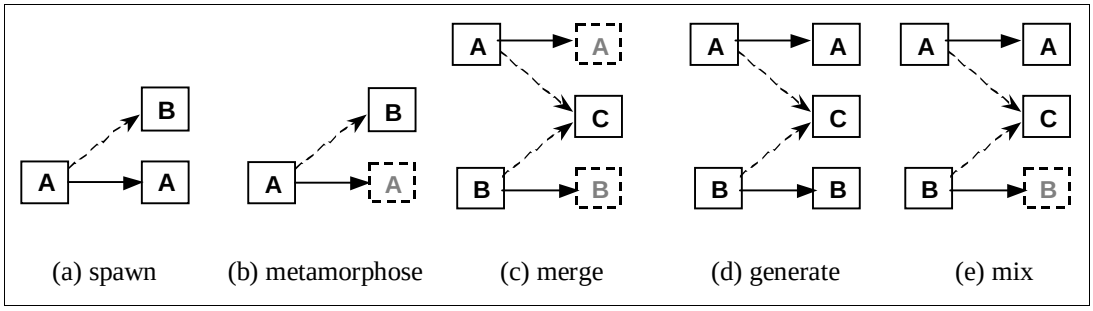

Figure 2.4: Object identity operations on simple objects (Hornsby and Egenhofer 1997).

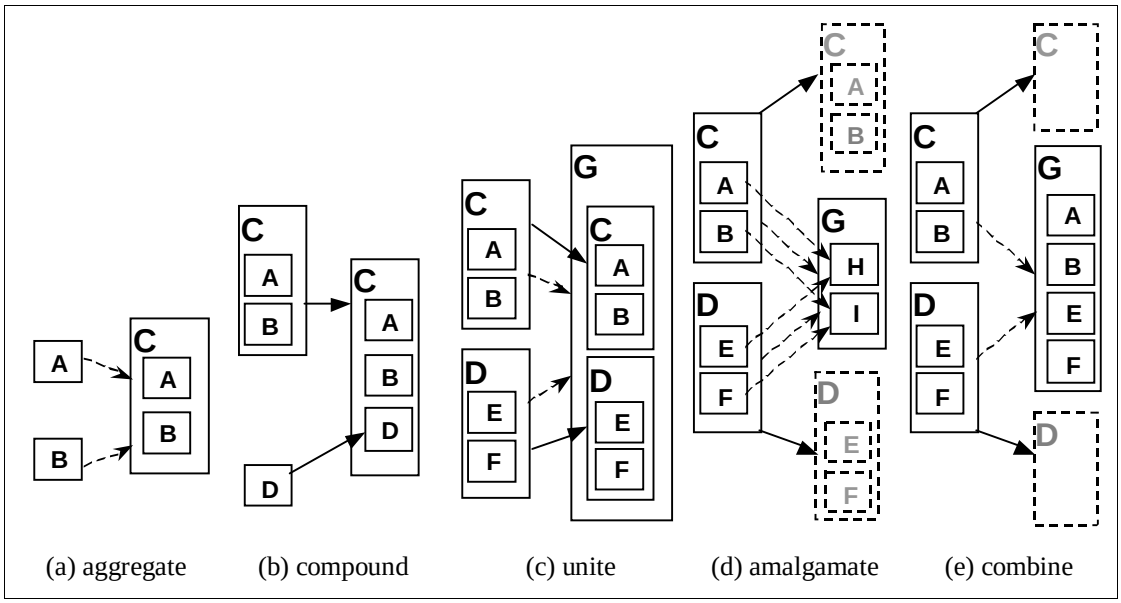

Figure 2.5: Object identity operations on composite objects (Hornsby and Egenhofer 1997).

The change description language (CDL) was proposed for qualitative graphical representation of operations on identity of objects. The effects of change on the object properties and the relationships between objects were analyzed with an emphasis on topological relations between spatial objects. Two real-world phenomena are used as examples: state borders and the spread of diseases. This work is extended to operations

for composite objects (Hornsby and Egenhofer 1998). They distinguished two types of composite objects: an aggregation (based on the relation *part-of*), and an association (based on the relation *member-of*). They introduced the notion of framework arguing that the parts form an aggregate only if they are sorted in a special configuration - framework.

All mentioned authors describe change operations informally, using intuitive notions for different situations from practical applications.

2.5 Formal background: Category theory

Category theory is a generalized mathematical theory of structures. One of its goals is to reveal the universal properties of structures of a given kind via their relationships with one another. Category theory was invented in 1945 by Eilenberg and Mac Lane, who borrowed the notion of category from Kant and Aristotle, (Eilenberg and Mac Lane 1945).

One of the interesting features of category theory is that it provides a uniform treatment of the notion of structure. This can be seen, first, by considering the variety of examples of categories. Almost every known example of a mathematical structure with the appropriate structure-preserving map yields a category. Sets with functions between them constitute a category. Metric spaces form a category whose primitive elements are points and whose primitive operation is distance. Category theory was used to model GIS applications (Herring et al. 1990). The basic framework for a category theory of spatial representations, relations and applications built upon them was defined.

Category theory unifies mathematical structures in a second, perhaps even more important, manner. Once a type of structure has been defined, it quickly becomes imperative to determine how new structures can be constructed out of the given one and how given structures can be decomposed into more elementary substructures. For instance, given two sets A and B, set theory allows us to construct their Cartesian product $A \times B$. For an example of the second sort, given a finite Abelian group, it can be decomposed into a product of some of its subgroups.

Category theory is the algebra of functions; the principal operation on functions is taken to be composition (Walters 1991). Category theory can be used in defining the basic building blocks of datatypes in programming, and it offers economy in definitions and proofs.

2.5.1 Definition of category

The following definition is taken from (Bird and de Moore 1997):

A category C is an algebraic structure consisting of a class of objects, denoted by A, B, C,..., and so on, and a class of arrows, denoted by f, g, h,..., and so on, together with three total operations and one partial operation.

The first two total operations are called target and source; both assign an object to an arrow. Formally, $f : A \rightarrow B$ indicates that the source of the arrow f is A and the target of f is B.

The third total operation takes an object A to an arrow $id : A \rightarrow A$, called the identity arrow on A.

The partial operation is called composition and takes two arrows to another one. The composition $g \,.\, f$ (pronounce "g after f") is defined if and only if $f : A \rightarrow B$ and $g : B \rightarrow C$ for some objects A, B, and C, in which case $g \,.\, f : A \rightarrow C$. In other words, if the source of g is the target of f, then $g \,.\, f$ is an arrow whose target is the target of g and whose source is the source of f.

Composition is required to be associative and to have identity arrows as units:

$$h \,.\, (g \,.\, f) = (h \,.\, g) \,.\, f$$

for all $f : A \rightarrow B$, $g : B \rightarrow C$ and $h : C \rightarrow D$, and

$$id_A \,.\, f = f = f \,.\, id_B$$

for all $f : A \rightarrow B$.

A simple example of a category is a preordered set. Given two elements p, q of the preordered set, there is a morphism $f : p \rightarrow q$ if and only if p is smaller or equal to q. Hence, a preordered set is a category in which there is at most one morphism between any two objects.

Beside functions, there are many more mathematical data that can be viewed as a category. Each directed graph determines a category: nodes of a graph are objects, and all paths are morphisms (arrows) typed with their start and end nodes. Composition is concatenation of paths. These data satisfy the axioms mentioned above, hence form a category.

2.5.2 Categorical product

A product of two objects A and B consists of an object and two arrows. The object is written as $A \times B$ and the arrows are written $outl : A \times B \rightarrow A$ and $outr : A \times B \rightarrow B$. These three things are required to satisfy the following property: for each pair of arrows $f : C \rightarrow A$ and $g : C \rightarrow B$ there exists an arrow (i.e. an operation) $\langle f,g \rangle : \mathrm{C} \rightarrow A \times B$ such that

$$h = \langle f,g \rangle \equiv outl \,.\, h = f \text{ and } outr \,.\, h = g$$

for all $h : \mathrm{C} \rightarrow A \times B$. The operator $\langle f,g \rangle$ is pronounced "pair f and g". The following diagram summarizes the type information:

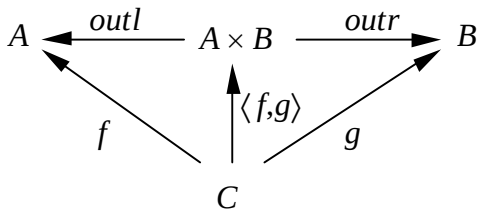

Figure 2.6: Commuting diagram for the categorical product.

2.5.3 Category of sets and total functions

The motivating example of a category is **Fun,** the category in which the objects are sets and the arrows are typed functions. An arrow (i.e. a function) is a triple (f, A, B), in which the set A contains the domain of f and set B is the range of f. By definition, A is the source and B the target of (f, A, B). The identity arrow $id_A : A \rightarrow A$ is the identity function on A, and the composition of two arrows (f, A, B) and (g, C, D) is defined if and only if $B = C$, in which case

$$(g, B, D) \,.\, (f, A, B) = (g \,.\, f, A, D)$$

where, on the right, $g \,.\, f$ denotes the usual composition of functions g and f.

In the category **Fun**, products are given by pairing. $A \times B$ is the Cartesian product of A and B, and $outl$ and $outr$ are the projection functions. The categorical product is useful in point-free programming.

2.5.4 Functions

Functions are basic building blocks in functional category theory. Mathematically speaking, a function *f* is a rule of correspondence which associates with each element of a given type *A* a unique member of a second type *B* (Bird and Wadler 1988). The type *A* is called the source or domain type, and *B* the target or range type. This fact is formally expressed by the following signature:

$$f :: A \rightarrow B$$

$$f\ a = b \text{ where } a \in B, b \in B$$

A function *f* is said to take arguments in *A* and return results in *B*. If *a* denotes an element of *A*, then we write *f* (*a*), or just *f a*, to denote the result of applying the function *f* to *a*. This value is the unique element of *B* associated with *a* by the rule of correspondence for *f*. The bracket notation, i.e. *f* (*a*), is usual in mathematics, but we will use the bracket-free notation, i.e. *f a*, usual in functional programming.

If a function is defined by the application to its argument, it is a point-wise definition. The examples are: $f\ x = x + 2$, $g\ x = 2 \times x$, etc.

Some functions have very general source and target types. The following definition defines the *identity* function:

$$id\ x = x$$

The identity function maps every member of the source type to itself. Its type is therefore $A \rightarrow A$ for any type *A*.

2.5.5 Functional composition

The composition of two functions *f* and *g* is the function *h* such that $h\ x = f\ (g\ x)$. Functional composition is denoted by the dot operator (.) for the symbol usual in mathematics (∘):

$$(f\ .\ g)\ x = f\ (g\ x)$$

A signature is the information about types of input and output for a particular function. Signatures start with the function name followed by the symbol :: meaning "have type of", input type(s) and an output type. The signature of functional composition (.) is given by:

$$(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$$

That is, functional composition takes a function of type $(b \to c)$, a function of type $(a \to b)$, and returns a function of type $(a \to c)$. The only restriction on functional composition is that the source type of its left-hand argument must agree with the target type of its right-hand argument (b in our example above).

Functional composition is an associative operation:

$$(f \,.\, g) \,.\, h = f \,.\, (g \,.\, h)$$

for all functions f, g and h. Therefore, there is no need to enclose the functions in brackets when writing sequences of compositions.

There are two basic styles for expressing functions: the *point-wise* style and the *point-free* style. In the point-wise style a function is described by its application to arguments. In the point-free style, a function is described exclusively in terms of functional composition and algorithmic strategies can be formulated without reference to specific datatypes (Bird and de Moore 1997). The advantage of functional composition is that some definitions can be written more concisely. For example, if the function h is composition of functions $f\,x = x + 2$ and $g\,x = 2 \times x$, we can write it in so-called point-wise notation as: $h\,x = f\,(g\,x)$, but the point-free definition is clearer: $h = f \,.\, g$. This leads to point-free style of programming, which is free of the complications involved in manipulating formula dealing with bound variables introduced by explicit quantifications.

2.5.6 Undefined values

Functions that are defined for all elements of their domain are called total functions. Addition (+) among natural numbers is an example for a total function. Functions that are not defined for all elements of their domain are called partial functions. The simplest example is numerical division by zero. If a computer encounters a task such as (1/0), it evaluates an error message "attempt to divide by zero", simply freezes, or crashes.

Partial functions play a major role in computer science because they are used to model algorithms that fail to halt for some input values. Unfortunately, the specification of abstract data types with partial functions poses serious problems (Loeckx et al. 1996). Therefore, a special element should be introduced to convert partial functions to total ones: it is $\perp$, pronounced "bottom", which stands for an undefined value. With the

undefined value, every partial function can be treated in the same way, as if it were a total function.

2.6 Summary

The philosophical discipline of ontology offers the basic assumptions about the real world. Artificial intelligence was the first discipline that attempted to formalize changing world using first order logic. Situation calculus with second order logic and mathematical induction promised better results. The research on temporal databases resulted in a plethora of temporal query languages, not yet fully standardized. Two different temporal dimensions are distinguished: valid and transaction time. Two principally different models for evolution of a database are database versioning and object versioning. Research on temporal GIS concentrates on qualitative models of change. Operations affecting object existence are drawn from specific applications and then systematized. A formal model of a spatiotemporal database with the support for change in object identity is missing. Algebra and category theory are sound mathematical foundations for declarative description of real world phenomena.

3. FRAMEWORK FOR A SPATIOTEMPORAL DATABASE

The goal of this thesis is to propose a new concept in modeling change of objects existing in the real world. To achieve the goal we must write down our assumptions about the real world - we must set up ***an ontology***. In this thesis, we assume that the real world consists of things or features that have properties. Things either are made of homogenous stuff or consist of other things. Each thing has its identity.

Once we have defined what is in the world, we are empowered to propose ***an epistemological model*** for a spatiotemporal database. The world is in continuous change: the objects are formed or born, they exist or live, and they disappear or die. Identities of things in the real world are represented by identifiers of objects in a database. Once given to an object, the same identifier may not be attached (re-used) to any other object, even if the original object was destroyed.

Objects and relations change over time, and hence the database changes, too. There are two temporal dimensions in which we capture this change: valid time and transaction time. Valid time registers the time when the change happens in the real world. Transaction time registers the time when the change is stored in the database.

The way objects, identities, and relations are represented is an ***implementation*** issue. In addition, there is a choice as to whether the database is represented as a series of database states or snapshots (so-called database-versioning) or a set of objects with a series of attribute states (object/attribute versioning). Implementation issues are not the topic of investigation of this thesis.

The rest of this chapter is divided as follows. Firstly, we discuss the ontology of the world making the important decisions for further modeling. Next, the epistemological model of the database is explained. Finally, implementation issues and the data model are discussed.

3.1 Ontology of the real world

Since we attempt to construct a model of the real world, we have to express our assumptions about the real world to resolve possible ambiguities of natural language. In other words, we have to describe the ontology of the problem we investigate. Ontology is an ancient philosophical discipline, developed originally by Aristotle and later

philosophers, and then rediscovered and redefined for the purpose of artificial intelligence.

In philosophy, ontology is the systematic account of existence. It has been concerned with the properties of objects, with their modes of existence and with questions such as how they can be divided in parts and how they fill space (Smith 1982).

In artificial intelligence, an ontology is an explicit specification of a conceptualization. A conceptualization is an abstract, simplified view of the world that we wish to represent for some purpose (Gruber 1993, p. 199). Since in AI "what exists is that which can be represented", ontology is often confused with epistemology - science of knowledge and knowledge representation.

We apply the ontological view of the world from Bunge (Bunge 1977; Bunge 1979). A formal model of objects in object-oriented programming based on this ontology can be found in (Wand 1989).

3.1.1 Things and their properties

According to Bunge, the world is composed of things and forms are properties of things. Things are grouped into systems or aggregates of interacting components. Every thing changes. Nothing comes out of nothing and no thing reduces to nothingness. Every thing abides by laws. Whether natural or social, laws are invariant relations among properties.

The world is viewed as composed of things of two kinds: concrete things that are called entities or substantial individuals, and conceptual things. An individual may be either simple or composite, namely, composed of other individuals.

Properties of substantial individuals are called substantial properties. A distinction is made between attributes and properties. An individual may have a property that is unknown to us. In contrast, an attribute is a feature assigned by us to an object. Indeed, we recognize properties only through attributes. A known property must have at least one attribute representing it.

Properties do not exist on their own but are "attached" to entities. On the other hand, entities are not bundles of properties. Thus, it might be said that the fundamental components of the world are entities. Entities are "known" to us through their properties. Properties are materialized in terms of attributes.

The properties of composite things may be related to the properties of the things in their composition. Hence, properties of composite things are of two kinds: hereditary, that is, properties that belong to the components of a (composite) entity, and nonhereditary. The latter are called emergent properties.

It is important that no two concrete, observable things can be the same. No two substantial individuals have exactly the same properties. If we perceive that two entities are identical, it is just because we do not assign attributes to all their substantial properties. For example, two glasses might have exactly the same superficial properties (color, weight, material, capacity), and we cannot see any difference between them. Nevertheless, two glasses are two different objects.

3.1.2 Changes

Full knowledge of a thing requires information about how the states of the thing can change. The necessary condition for this is that every (concrete) thing has at least two distinct states. When a thing undergoes a change, at least one property will have to change in value; hence, a change of a thing is manifested as a change of state. It follows that, for a change to be possible, the thing has to have more than one state.

Bunge introduced the principle of nominal invariance to clarify the persistence of things: “A thing, if named, shall keep its name throughout its history, as long as the latter does not include changes in natural kind - changes which call for changes in name.” (Bunge 1977, p. 221). Individuals with distinct names are distinct. This is similar to the unique name assumption incorporated in special theories of first order logic that are capable to represent relational databases (Reiter 1984).

3.2 Epistemology of the world

Ontologies are mainly useful for constructing general theories. Deriving observable consequences from the theory is a further step. This is the task of epistemology. Epistemology is the part of philosophy concerned with knowledge and knowledge representation. A representation is called epistemologically adequate for a person or machine if it can be used practically to express the facts that one actually has about the aspects of the world (McCarthy and Hayes 1969).

The elements of the ontology proposed in the previous section are mapped to an epistemology, which will be used in the rest of this thesis. Things are mapped to

objects, properties to attributes, nominal invariance to identity. As an epistemological fact, different temporal perspectives are introduced.

A model is a description of some phenomenon, created for some purpose. It embodies a closed-world assumption: that the set of objects and relations in the model include everything necessary for that purpose. The model of the changeable world depends on the closed-world assumption. It is impossible to predict the next value of an object attribute without assuming that all the influences of that attribute are known (Kuipers 1994). Therefore, in what follows, we assume that all the facts about the modeled selected part of the world are known - what is not known, does not exist.

3.2.1 Object categories

Object is a concept, abstraction, or thing with meaning for the problem at hand (Rumbaugh et al. 1991). This definition embraces different categories of objects in the real world. Here, categories mean classification and not categories in mathematical sense as in Chapter 2.

There are physical objects that simply exist - these are substances in Aristotelian ontology: stones, mountains, rivers, the earth. We can, at least in theory, touch and manipulate them. There are abstract objects that are immaterial - these are accidents in Aristotelian ontology: qualities, events, and processes. In addition, there is a special group of objects, which are called institutionalized accidents (Smith to appear). Such objects are result of human intellectual effort. Examples include the equator, the Northern Hemisphere, state boundaries.

Objects belong to categories or classes. The pigeon on my windowsill is a particular instance of the category of birds. A penguin is a less characteristic example, yet an instance of birds, too. Such categorization in which there are instances more and less characteristic for a particular category is known as radial categorization (Lakoff 1987, with more references to the pertinent cognitive literature).

Objects have attributes. Cars have color, year of production, length, weight, engine power; cadastre parcels have area, market value, usage. Each attribute may take a value from a pre-specified domain: a particular car can have the value *red* for the attribute *color* and the value *1979* for the attribute *year of production*.

3.2.2 Identity

In the real world, a thing simply exists, but within a representation, each object needs a special handle by which it can be uniquely referenced and distinguished from other objects. It is achieved by attaching an identifier to each object in the database. Object identity is considered one of the essential paradigms in object-oriented modeling (Cattell and Barry 1997; Rumbaugh et al. 1991). In a database, identities are represented by identifiers, which are constructed and maintained by the database management system.

The identity allows the distinguishing of one object from others, even if objects are of the same kind and have the same attribute values. For example, a car factory may produce two cars that have completely the same appearance (color, dimension, engine power, etc.). Yet, each car will have its own serial number that allows its unique identification.

Identity has a different meaning than in mathematical logic where the identity is interchangeably used with equality (Tarski 1946). Two different real objects may be equal under some concept of equality, but never identical.

The identity must fulfil three conditions to properly perform its role (Al-Taha and Barrera 1994):

- ***uniqueness***: Two distinct objects may not share the same identity.
- ***immutability***: Identity is assigned at the creation of an object and remains the same during the lifetime of the object. Neither the system nor a user can change the identity of the object.
- ***non-reusability***: A new object may not take the identity of any already destroyed object, since this could be interpreted as an invalidation of the deletion of the latter object.

The notion of object identity is different from the notion of a primary key in the relational model (Cattell and Barry 1997). Relational algebra, (Codd 1979), is based on relational calculus. It is a powerful tool for storing and processing tabular information, but lacks the capability of representing objects for dealing with complex applications like GIS. A tuple in a relational table is uniquely identified by the value of the columns comprising the primary key of the table. If the value in one of those columns is changed, the tuple changes its identity and becomes a different tuple. In addition, if two

tuples become equal in key values, they are merged. Traceability to the prior value of the primary key is lost.

3.2.3 Relations

Objects are connected through relations. Relations can be, for example, topological or mereological. The other groups of relations include comparatives (is longer than, is to the east of) and so called 'Cambridge relations' (is father of, is cousin to) (Mulligan and Smith 1986).

Topological relations describe spatial link among objects (my computer is *on* the desk; a car is parked *in* front *of* the house; I am *in* the room; the garden is *between* the wall and the fence).

Mereological relations describe how objects are composed of other objects (a keyboard is a *part of* a computer, a car *has* four wheels, a room is a *part of* a building). For solid physical objects, we accept the view from naive physics that "every solid physical object is either a piece of solid stuff, or else an assembly which is made up of a finite number of other solid physical objects" (Hayes 1985b, p. 73). In other words, there exist simple objects and objects composed of parts. The relation *part of* is a central relation in this thesis.

In our epistemological model, mereological relations will be used to cover ontological assumptions about the structure of things consisting either of homogeneous material or of other individual things.

3.2.4 The structure of time

The world is in continuous change: the objects are formed or born, they exist or live, and they disappear or die. Metaphorically speaking, the existence of an object is the life of an object. A house exists from the time it was built until it is destroyed.

The structure of time is complex - there are several choices: time can be linear or branching, discrete or continuous, absolute or relative, bounded or unbounded (Snodgrass 1995a).

Time can be linear, branching, or cyclic (mostly used for planning). Two basic structural models of time are linear and branching. In the linear model, time advances from the past to the future in a totally ordered fashion. In the branching model, time is

linear from the past to now, where it splits into several time lines, each representing a potential future sequence of events. Along any future path, additional branches may exist. The structure of branching time is a tree rooted at now. Finally, the model of cyclic time is applicable for recurrent processes such as year seasons, seasonal floods.

Time can be discrete, dense or continuous. The discrete time line (if linear model is assumed) is isomorphic to natural numbers: each point in time has a single successor. On the other hand, dense models of time are isomorphic to the rationals or the reals: between any two moments in time there exist another moment. Continuous models of time are isomorphic to the reals: they are dense, but there are no gaps. Although time itself is generally perceived to be continuous, most proposals for adding a temporal dimension are based on the discrete time model. In the discrete model, each natural number corresponds to a non-decomposable unit of time with an arbitrary duration called chronon. A chronon is the smallest duration of time other than a point that can be represented in the discrete model. It is not a point but a minimal line segment on the time line.

Time can be bounded or unbounded on both ends: in the past and in the future. Time began with the Big Bang. If the universe is closed, then the time will have an end in the Big Crunch; if it is open, time will go on forever (Hawking 1988). This is more a cosmological question and not relevant for our purposes.

Time can be absolute (anchored) or relative (unanchored). Absolute time is fixed with respect to a pre-defined time scale, usually Gregorian calendar time (e.g. January 1, 1998). Relative time, termed span, is a piece of time without a fixed position on the time scale, (e.g. 9 hours).

In this thesis, we use linear, discrete, bounded, absolute model of time. We prefer linear over branching time because of the simplicity. The discrete model is appropriate for the limited representation capabilities of database technology. Bounded time suits our needs because we are not interested in the history or future of the whole universe, but in the history and some short future of the resources concerning mankind.

3.2.5 Temporal dimensions

Two time dimensions are of general interest in the context of databases: *valid time* and *transaction time*. The following definitions are taken from the latest version of the consensus glossary in the temporal database community:

"The *valid time* of a fact is the time when the fact is true in the modeled reality. A fact may have associated any number of instants and time intervals, with single instants and intervals being important special cases. Valid times are usually supplied by the user." (Jensen and Dyreson 1998, p. 370).

"A database fact is stored in a database at some point in time, and after it is stored, it is current until logically deleted. The *transaction time* of a database fact is the time when the fact is current in the database and may be retrieved. Consequently, transaction times are generally not time instants, but have duration. Transaction times are consistent with the serialization of the transactions. They cannot extend into the future. In addition, as it is impossible to change the past, (past) transaction times cannot be changed. Transaction times may be implemented using transaction commit times, and are system-generated and -supplied." (Jensen and Dyreson 1998, p. 371).

With these definitions, four types of temporal databases are differentiated: static, rollback, historic, and bitemporal databases (Table 3.1). Static databases support neither transaction nor valid time. Historic databases support valid time, but not transaction time. Rollback databases support transaction time, but not valid time. Finally, bitemporal databases support both valid and transaction time.

	No Transaction Time	Transaction Time
No Valid Time	***Static*** *Database*	***Transaction*** *Database*
Valid Time	***Historical*** *Database*	***Bi-temporal*** *Database*

Table 3.1: Time perspectives and resulting temporal database models.

If a fact is stored in the database at the same time it is observed in the modeled reality, transaction time is equal to valid time. An example for such a database is the registration of cloud observations by a satellite camera, assuming that that the time necessary to save the data is infinitely short (Jensen and Snodgrass 1992).

Two essential criteria the identifiers must fulfil (uniqueness and non-reusability) are, by definition, expressible only in the transaction time. Therefore, we will consider transaction time as our primary temporal dimension and assume that the valid time is represented as an attribute.

3.3 Conceptual model of a temporal database

A database is a computer-based collection of data with the capability for controlled definition, access, retrieval, manipulation and presentation of data within the collection. A database represents a specific set of objects selected from the real world that can be represented. Such a set is called *a universe of discourse*. In this section, we analyze conceptual model of our database. The representation of the database elements is covered in Chapter 6.1.

3.3.1 Objects, attributes, and relations

In a typed system, each object is an instance of the class it belongs to. As we have already seen above, objects can be physical, such as a car, or abstract, such as ownership of a piece of land. Both physical and abstract objects have their static and dynamic side. To differentiate such objects, we use a complex type system.

The dynamic or behavioral side of an object is expressed as a set of operations that the object will perform. For example, a car can be started, stopped, or repaired. While under repair, it is not available for driving.

Finally, the objects in a database have life spans; their lives begin when they are entered in the database and end when they are removed from the database (see Figure 3.1). Between these two points the objects are updated: changes in their attributes are recorded. Both the static and the dynamic sides of an object are common for a particular *class* or *type* of objects.

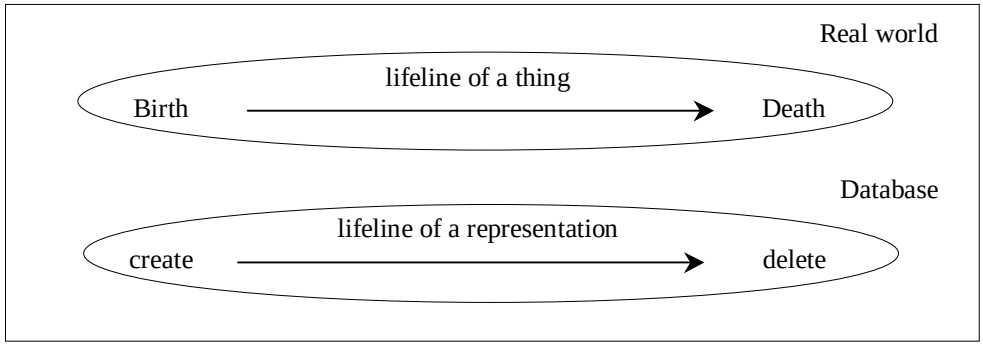

Figure 3.1: Life of a thing in the real world vs. life of its representation in a database.

The static aspect of an object is expressed by a collection of named *attributes*, each of which may take a value from a pre-specified domain (Worboys 1995). A car object might have *color*, *manufacturer name*, and *engine power* among its attributes. A

particular car might take the value *red* for the *color* attribute. All attribute values for a given object in a particular moment constitute its *state*.

Objects in the database are connected through *relations*. Relations are defined by object types; they are valid only if the related objects are of the proper type. Binary relationships involve two object types, ternary relations involve three object types, and so on. A binary relationship may be one-to-one, one-to-many, or many-to-many, depending on how many instances of each type participate in the relationship (Chen 1976). For example, marriage is a one-to-one relationship between two instances of type *Person*. A woman can have a one-to-many mother-of relationship with many children. Teachers and students typically participate in many-to-many relationship. Several books may be placed on a table. A question we might want to ask is "Which books lie on the table?" To answer this question, a connection between books and tables must exist in the model. This connection is called *relationship*. A *relationship type* connects one or more object types. In our example, the relationship type is *isOn*. This situation is shown in Figure 3.2, using a diagrammatic language called *entity-relationship diagram*, or *E-R diagram*.

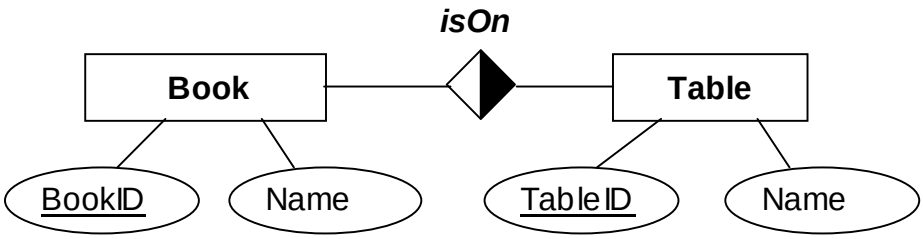

Figure 3.2: Entity-relationship diagram for a one-to-many relation *isOn*.

Objects (or entities) are enclosed in rectangular boxes; attributes are enclosed in ellipses; identifiers are underlined. The link between the boxes *book* and *table* is a relationship - *isOn.* The relationship *isOn* is many-to-one: the white part of the rhomboid denotes the *many* side, and the black part denotes the *one* side.

3.3.2 Database vs. object versioning

Much discussion in the literature centers around different strategies to record change, but these are logically equivalent, as it will be shown in this section. There are two distinct ways to represent change of objects in a database depending which elements of the static database are changed. If the complete new state of the database is stored for

every change, it is called database versioning. If the new states of modified objects only are stored, it is called object versioning.

Database versioning model produces a new ***snapshot*** of the whole database for each change. A snapshot is a set of objects and relationships among them valid at a particular point in time scale. An example is a photograph showing a visible field of a camera eye in the moment of exposure. In this view, the universe (world) of discourse or a complete temporal database is a sequence of snapshots representing discrete changes of objects and relations among them since the creation of a database. Typically, a movie is a temporal sequence of events recorded by a camera with a frequency of 24 snapshots per second. This model stores the complete history of a database - the snapshot for any moment in the history of a database is readily available. A drawback for implementation purposes is that a lot of unchanged information is stored many times.

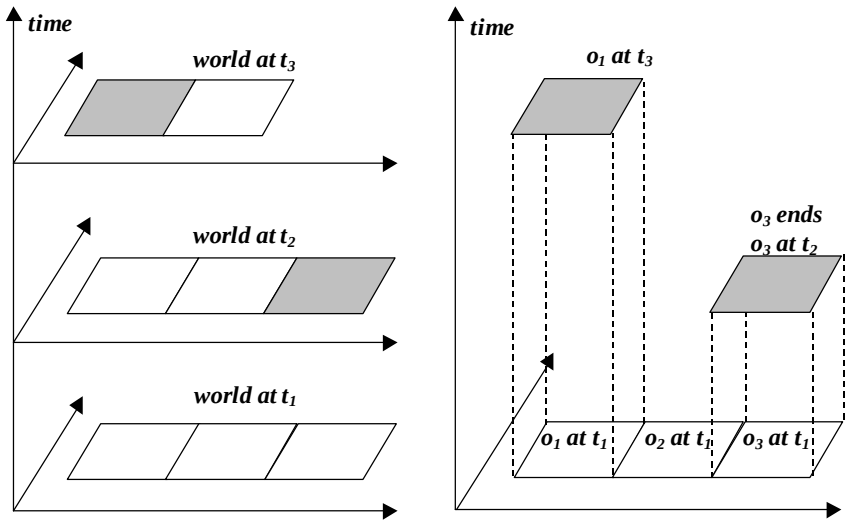

Figure 3.3: Database versioning (left) and object versioning (right).

Object versioning stores a new version for every changed object or attribute. A universe of discourse is a collection of objects each of which has a version for every change the object was involved in. This model stores the changed information only. Each version of changed objects must have a timestamp. A drawback of object versioning is that the state of the complete database at a particular point in time must be calculated. The deletion of an object must be explicitly stated (object o_3 in Figure 3.3). In the case of object versioning, it is not easy to follow the evolution of the database from one snapshot to the next.

The major difference between object versioning and database versioning is in the dimension on which the grouping is done. In object versioning, objects are fixed, and times are grouped. In database versioning, time is fixed, and objects are grouped. It should be noted that the actual change occurs among attributes, i.e. a set of attributes varies in an object, which has the permanent identity. The representation of two varying objects across the four time instants is shown in Figure 3.4.

```
TIME   OBJECTS                          OBJECTS        TIME
 1      Car(red)                        Car (red)      1
 2      Car(blue), House(white)         Car (blue)       2 3
 3      Car(blue), House(white)         House (white)    2 3 4
 4                 House(white)
```

Figure 3.4: Grouping of times (left) and grouping of objects (right).

The figure shows the universe of discourse consisting of two objects: a car and a house, having a color as the attribute. At the time point 1, only the car exists. At the time point 2, the car color is changed to blue and the house appears. At the time point 4, the car is deleted.

It is beneficial to have a mapping between database and object versioning. The transformation functions from the snapshot view to the objects view and vice versa are based upon this difference in grouping dimensions. An abstract data type of the snapshot view is an ordered list of temporal elements paired with appropriate lists of objects at a specific time: [(1, redCar), (2, blueCar, whiteHouse), (3, blueCar, whiteHouse), (4, whiteHouse)]. An abstract data type of object view is an exhausting list of objects paired with appropriate lists of temporal elements showing when those objects existed: [redCar (1), blueCar (2,3), whiteHouse (2,3,4)]. The algorithm for the transformation from the object view to the snapshots view can be informally described as follows:

a) distribute time (pair timestamp with each object in each snapshot),
b) find all objects,
c) select temporal points for each object: [(blueCar, 2), (blueCar, 3)],
d) normalize each object: blueCar [2, 3],
e) concatenate results.

The inverse transformation starts with a group of timed objects, and the algorithm is:

```
a) distribute objects (pair each object with all temporal points),
b) find all temporal points,
c) select objects for each temporal point: [(2,blueCar),
   (2,whiteHouse)],
d) normalize each timestamp: [(2, [blueCar, whiteHouse])],
e) concatenate results.
```

The complete code for both transformations is given in Chapter 6. Transformations are lossless, that is, no information is lost when converted from one view to another. Therefore, the choice of versioning technique is an irrelevant implementation question, and we will develop our database model on the conceptually simpler database-versioning approach. Each change in the database produces a new snapshot having a new succeeding timestamp.

3.4 Treatment of errors

In the ideal case, all observed and stored data match the true states in the universe of discourse. This is, however, never the case. Because of imperfection in observation apparatus or simply because of lack of appropriate knowledge, errors occur.

An error in a database is a piece of stored information that does not match the true state in the universe of discourse with the expected accuracy. If an error is discovered, it should be removed from the database. This section describes possible situations and proposes adequate solutions of problems related to error correction in databases.

We begin with a simple scenario of an error in a personal database. The personal database stores data about people: their names and dates of birth. Now, suppose the following transactions are performed on the database:

> *On 25 Oct 1998, the data for a person with the name John was entered and the date 20 Jan 1850 as John's birthday. One month later, 25 Nov 1998, it was found that John's birthday was not on 20 Jan 1850, but on 20 Jan 1950. An error occurred and needs to be corrected.*

The procedure of correction depends of the capability of the database management system to cope with various temporal dimensions: transaction and valid time. Four cases are distinguished: static, historical, rollback and bitemporal database.

Static database tracks neither the transaction nor valid time. Such database could record Johns birthdate as an optional attribute. In that case, the error data (the attribute 20 Jan 1850) is overwritten with the correct data (20 Jan 1950). After 25 Nov 1998, there is no information in the database if the error ever happened. An inspection of the database would have found that the database was always in the consistent (error-free) state.

Historical database tracks the valid time, but not the transaction time. A temporal reference (timestamp) about the time when an event happened in reality is required for every record in the database. The erroneous data (the timestamp for birth 20 Jan 1850) is overwritten with the correct timestamp for birth (20 Jan 1950). As in the previous example, there is no information about the existence of the error. The database is revised without evidence that the revision ever took place. The only difference between static and historical databases is that a static database may model the temporal dimension as an attribute, whereas a historical database must track the valid time.

The rollback database tracks the transaction time, but not the valid time. The required temporal reference for every record is the time the record is stored in the database. The valid time dimension is captured as an optional attribute. In this case, we have a transaction timestamp 25 Oct 1998 as the date the John's wrong birthdate was entered and an attribute, 20 Jan 1850, which should be corrected. The correction is timestamped as an event happening on 25 Nov 1998. After that date, we have correct state in our database, since the attribute - John's birthdate - has the correct value. What is more, an inspection about the state of the database between 25 Oct 1998 and 25 Nov 1998 would have found that the database had recorded different (factually wrong) information about John's birthdate.

Finally, bitemporal database records a bitemporal element for every record in the database - both valid and transaction times are recorded. The error is corrected by adding the new transaction with the correct timestamp in valid time for John's birthdate. The difference between bitemporal and rollback databases is that a bitemporal database supports temporal query in valid time about every record in the database, whereas a rollback database can inspect such data as the attribute only.

Katsuno and Mendelzon distinguished two concepts in correcting the error state in the database: *update* and *revision*. An *update* brings the knowledge base up to date when the described world changes. A *revision* is obtaining new information about a static world (Katsuno and Mendelzon 1991).

The scenario we analyzed here is clearly a revision, because the universe of discourse was static - John's birthday did not change. In the first two cases (static and historical databases), we lost the information about existence of an error in past times. In the last two cases (rollback and bitemporal databases) we just added the new information to the existing knowledge, preserving the fact that we had had an error before.

3.5 Summary

In this chapter, the setup for an object-oriented temporal database is described. The database is a collection of selected objects and relations from the real world. The essential concept of identity in a temporal database is explained: identity must be unique, immutable and non-reusable. An object has a type; it is an instance of the class. Objects are involved in relations, and altogether built a database.

Among different temporal models, a linear, discrete, bounded, absolute time is chosen. Transaction time is selected as primary temporal dimension, and the valid time is an optional attribute. Although simple, the model is powerful enough for modeling our universe of discourse. The change is modeled by database versioning - the mutation of the complete database for every change. It is conceptually simpler than object versioning. Mappings between both versionings are possible and lossless.

The conceptual model presented in this chapter is formalized in Chapter 6. In the next chapter, we introduce the rules for the identity change - operations affecting object identity.

4. OPERATIONS AFFECTING OBJECT IDENTITY

In the real world, changes happen over time. The change can be continuous (gradual) or catastrophic (abrupt). It may affect a particular attribute of an object (location, color, or dimension), its relation to other objects (topology, parthood), or its mere existence (identity). The topic of this thesis is the change affecting object identity. Operations affecting object identity can be grouped in categories, which we will call *lifestyles*.

The life of an object in a database begins with its *creation*. The creation connects the new identity with a set of attribute values. An object may be created only once, preserving its identity throughout its whole life. The creation is common to all objects. The end of an object's life is determined by its death. Since some objects might be modeled as eternal in the context (e.g., the sun from the perspective of the earth), this operation is not universal for all objects in the database. Deep philosophical questions about the real meaning of eternity are not considered in this thesis.

An object may be modeled as having multiple episodes of its existence. Such temporary loss of existence is modeled with two operations: *suspend* and *resume*. The operation *suspend* freezes the life of an object until it is *resumed*. A computer taken apart is *suspended* until it is assembled again and its identity *resumes*.

The four basic operations are defined with preconditions and postconditions: predicates that are valid before and after applying the operations. All other operations are compositions derived from the basic operations.

Basic operations can be composed: an object can be *destroyed*, triggering the *creation* of its successors. This is the crucial property for modeling the higher level operations like splitting and merging. The composition tables show that the number of compositions of basic operations is finite.

Two different lifestyles are recognized: *fusions* and *aggregates*. *Fused* objects lose their identity - they are *destroyed* (pouring two glasses of water into a jar destroys the liquid objects in both glasses). *Aggregated* objects do not lose their identity - they are *suspended* (assembling the parts of a car does not destroy the identity of parts). Both *fusion* and *aggregation* could be *constructive* or *non-constructive.* An example of constructive fusion is pouring the water from the glasses in a jar. If the water is poured back into the glasses, and then again into the jar, the second fusion may be considered as *resuming* the previously existed liquid object in the jar. Since no new objects are

created, such fusion is *non-constructive*. Disassembling and assembling a car is an example of *non-constructive aggregation*. The final destruction of the car is a *constructive segregation* (the inverse operation to *aggregation*).

We present the minimal set of conditions a temporal database must satisfy for dealing with the proposed lifestyle operations. The concept of lifestyles can be implemented in a temporal database only if the database is recording the transaction time (rollback and bitemporal databases).

4.1 Operations affecting the identity of a single object

We propose four basic operations affecting the identity of a single object: *create, destroy*, *suspend*, and *resume*. Basic operations are known under different names in literature: as create, destroy, kill, and reincarnate (Clifford and Croker 1988), as create, destruct, and reincarnate (Hornsby and Egenhofer 1997).

The first of them, *create*, is essential for all types of objects. An object can be created independently of other objects or as a child object of one or more parent objects. The temporal link with the predecessors of a newly created object is the essential part of the operation *create.*

The second basic operation, *destroy*, terminates the existence of an object. The previous existence of the destroyed object is preserved, but the object cannot be referenced in the future time.

A temporary loss of existence is modeled with the pair of operations: *suspend* and *resume*. A suspended object is not active in the database until it is resumed. The effects of all four simple operations on the existence of an object are shown in Figure 4.1. The value of the predicate *exist* changes after each operation.

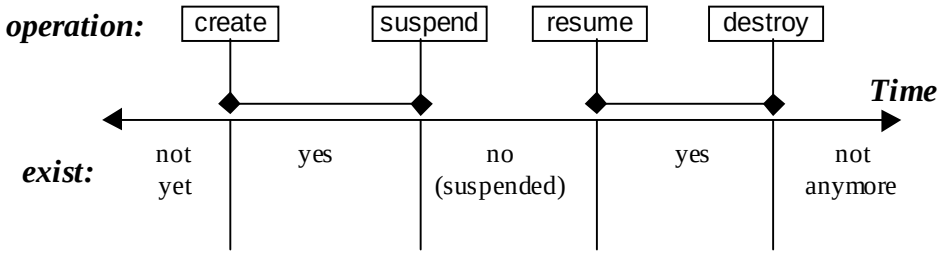

Figure 4.1: Possible episodes in the life of an object.

Before an object is created value of the predicate *exist* is *"not yet"*. When an object is created, the value of *exist* changes to *"yes"*. Then, if an object is suspendable, it can be

suspended and a special tag *"no(suspended)"* is used to describe this state. A subsequent resumption would have changed the state *"no(suspended)"* to *"yes"*. Finally, if an object is destroyed, the value of the predicate is *"not anymore"* meaning that the object cannot reappear in future states of the database. The last kind of transition could have happened without the intermediate suspension.

Further, the fifth operation, *evolve*, captures the semantics of changing the identity of an object that preserves a temporal link with its predecessor. This operation affects a single object, although it is a composition of two simpler operations: *destroy* and *create*.

4.1.1 Create

The existence of an object in the database begins with the creation of its identifier in the database. In Figure 4.2 an identity labeled "Id1" is created at the time point t1. The operation ***create*** is essential for both the static and the temporal databases, since the unique identity is needed for distinguishing objects in static databases.

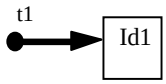

Figure 4.2: Identity operation *create*.

When an identifier is created in a database, it is chosen from an abstract set Ω. Therefore, the domain of the operation *create* is the set Ω, and the range is the set of identifiers. For a particular creation, the result type is an identifier (ID). Creating a specified identifier is not allowed, because such operation might violate the properties of uniqueness and non-reusability of identifiers. Note that the label *ID* for an identifier has different meaning from the label *id* for the identity function.

Using functional notation, the signature of the function *create* is written:

$$create :: \Omega \rightarrow ID$$

The effect of the creation on a database is explained using the standard technique in program verification of state-oriented specifications: precondition and postcondition, (Loeckx et al. 1996). Verification consists of a set of assertions of the form:

$$\{\varphi\}\ P\ \{\psi\},$$

where φ and ψ are formulas of predicate logic and where P contains the piece of program to be specified. If the precondition φ holds before execution of P, then this execution terminates and the postcondition ψ holds.

In our example, creation of a new object takes the database from one state to another. The precondition for the initial state is that the object with the identifier *i* does not yet exist. The postcondition for the final state after the execution of the operation *create* is that the object with the identifier *i* exists. The values for conditions are taken from Figure 4.1. Formally, this reads as:

```
Pre/Post - conditions:          Code:
{exist (i) = “not yet”}
                                create
{exist (i) = “exist”}
```

The concept of predecessors, inevitable in temporal databases, is inherently tied with the creation of an object. The creation is the only basic identity operation involving predecessors. The set of predecessors is empty if the object is created without predecessors. Therefore, there is no special notation in Figure 4.2. Predecessors will be shown in complex operations (see section 4.2 below).

4.1.2 Destroy

The existence of an object in the database is terminated by ***destroying*** its identifier from the database (Figure 4.3). For the database, it means that, from the moment of destruction, it is not possible to update the properties object. An object is required to exist if it is to be destroyed; *destroy* takes an identifier as an argument. The identifiers of destroyed objects are not forgotten, because of non-reusability. In addition, the database may be queried about the past existence of a destroyed object.

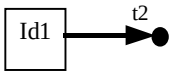

Figure 4.3: Identity operation *destroy*.

When an identifier is destroyed, i.e. removed from a database, it is disposed to an undefined space. Therefore, the domain of the operation *destroy* is the set of identifiers, and the range is infinity Ω. For a particular *destruction*, the argument type is an identifier ID. Using functional notation, the signature of the function *destroy* is written:

$$destroy :: ID \rightarrow \Omega$$

The precondition for the operation *destroy* is the existence of the identifier to be destroyed in the database. The postcondition is the non-existence of the destroyed identifier. In formal language:

```
Pre/Post - conditions:          Code:
{exist (i) = “yes”}
                                destroy (i)
{exist (i) = “not any more”}
```

The values of predicates for the precondition and the postcondition are given in Figure 4.1.

4.1.3 Suspend and resume

An object may have multiple episodes of its existence. A well-known example from history is the state of Austria, which disappeared and reappeared during this century.

A temporary loss of existence is modeled with two operations: ***suspend*** and ***resume***. The first operation requires an active (not-suspended) identifier, while the second requires a suspended identifier. In Figure 4.4, the shadowed box with the struck label represents the suspended identifier.

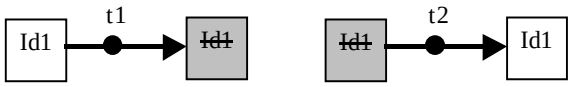

Figure 4.4: Identity operations *suspend* (at t1) and *resume* (at t2)**.**

The operation *suspend* freezes an object by preserving it from other operations until it is resumed. Appropriateness of this pair of operations is the matter of user's choice: one could assume that sleeping of living beings appears to be equivalent to a suspended state.

The type of arguments and results of the operations *suspend* and *resume* is the same: an identifier (ID). The signature of both operations is written:

$$suspend, resume :: ID \rightarrow ID$$

The precondition for the operation *suspend* is the existence of the object. The postcondition is that the object is suspended.

```
Pre/Post - conditions:          Code:
{exist (i) = “yes”}
                                suspend (i)
{exist (i) = “no (suspended”}
```

The precondition for the operation *resume* is that the object is suspended. The postcondition is that the object exists again.

```
Pre/Post - conditions:          Code:
{exist (i) = “no (suspended)”}
                                resume (i)
{exist (i) = “yes”}
```

This pair of operations, *suspend/resume*, reflects the pair kill/reincarnate (Clifford and Croker 1988). Hornsby and Egenhofer define reincarnation as the operation *destruct* (equivalent to *destroy* in our notation) followed by the operation *create* of an object with the same identity (Hornsby and Egenhofer 1997). In our model, destroyed identifiers cannot be recreated.

4.1.4 Evolve

Basic operations presented so far could be composed in various ways. In this section, we analyze compositions of two basic operations applied subsequently to the same object.

Since the order of composed operations matters, there are 16 different compositions. The composition symbol "." ("dot") is applied in the order as in the equation*: (g . f)(x)=g(f(x)).* We explore all possible compositions in Table 4.1. A composition yielding undefined results is marked with ⊥ ("bottom"). A composition yielding the state that is the same as the original is the identity function (*id*) in the mathematical sense.

f g	create	destroy	suspend	resume
create	⊥	evolve	⊥	⊥
destroy	id	⊥	destroy	destroy
suspend	suspend	⊥	⊥	id
resume	⊥	⊥	id	⊥

Table 4.1: Compositions *g.f* of two operations *f* and *g* on the same identity.

The results in Table 4.1 are calculated as *g . f*. That is, the operations from the row *f* are applied first. Then, the operations from the column *g* are applied to the results of the first operation (*f*).

A composition as *create.create* is undefined, because the postcondition of the first creation is the existence of the identifier *i*. Thus, it cannot be created again, since the precondition for the operation *create* is the non-existence of the object. The predicate *exist* has the signature $ID \rightarrow Bool$, and the string values (not yet, exist, not anymore) are introduced to improve understanding. We write the proof in functional notation:

```
Pre/Post - conditions:            Code:
{exist (i) = “not yet”}
                                  create
{exist (i) = “exist”}
                                  create
not fulfilled since
{exist (i) = “exist”} ≠ {exist (i) = “not yet”}
```

The same conclusion can be proved for the composition *destroy.destroy*. The precondition for the second application of *destroy* is not fulfilled, because the identifier is already destroyed:

```
Pre/Post - conditions:            Code:
{exist (i) = “yes”}
                                  destroy
{exist (i) = “not anymore”}
                                  destroy
not fulfilled since
{exist (i) = “not anymore”} ≠ {exist (i) = “yes”}
```

If an identity is destroyed immediately after its creation, the result is the identity function *id* (under assumption that the value “not yet” is identical to “not anymore” if the object is destroyed at the same time it was created) :

```
Pre/Post - conditions:            Code:
{exist (i) = “not yet”}
                                  create
{exist (i) = “yes”}
                                  destroy
{exist (i) = “not anymore”}
```

Both possible compositions of the operations *suspend* and *resume* yield the identity function. We show the case of the composition *resume.suspend*:

```
Pre/Post - conditions:            Code:
{exist (i) = “yes”}
                                  suspend
{exist (i) = “no (suspended)”}
                                  resume
{exist (i) = “yes”}
```

The composition of the operations *suspend* or *resume* with the operation *destroy* result in *destroy* if the former is performed last. If the opposite is the case, i.e. if *destroy* is performed first, the result is undefined: the destroyed identifier is not available for *suspend* or *resume*.

The compositions *create.suspend* and *create.resume* are undefined, because the object must not exist in order to be created (precondition for *create*). The composition *resume.create* fails because the precondition for *resume* is not fulfilled after the creation. Finally, *suspend.create* results in *suspend*.

A composition of *creation* and *destroying* where the *destroying* comes first is the most important result - a new operation ***evolve***, having the following signature:

$$evolve :: ID \rightarrow ID$$

Because of the identity properties, the newly created identifier is denoted *j* in the following verification. In addition, the old identifier (*i)* is the argument of the operation *create* in order to maintain a temporal link with its predecessor. In the original definition of *create*, the predecessor argument was empty and thus omitted for the sake of simplicity.

Pre/Post - conditions:	Code:
{exist (i) = “yes”}	destroy (i)
{exist (i) = “not anymore”}	create (i)
{exist (j) = “yes”}	

The concept of identity evolution allows an object to change its identity under conditions that a temporal link with the previous identity is established. An example for such an operation is a country suddenly changing its constitution from a monarchy to a republic (like Italy during the Second World War).

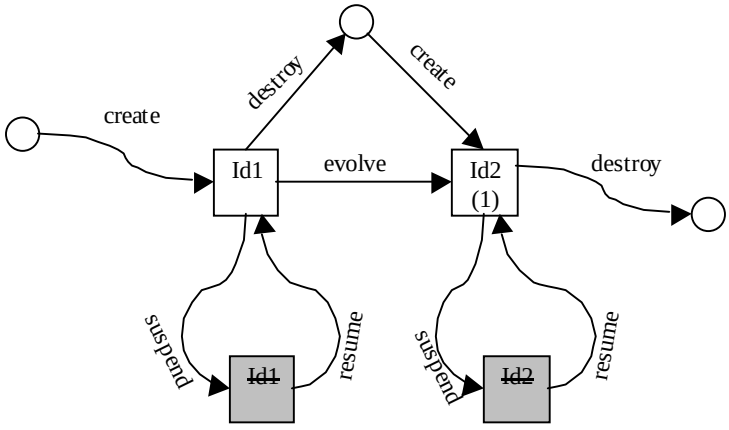

Figure 4.5: State diagram for operations affecting identity of a single object.

A state diagram represents possible operations on a single object identity (Figure 4.5). The operation *evolve* yields the same result as the combination of operations *destroy*

and *create*. The temporal chain is represented with the label "(1)" in the box of Id2: the identity Id1 is the predecessor of the identity Id2.

4.1.5 Removing histories

A short explanation about the difference between static and temporal databases with respect to the operation *destroy*, and of the difference between *destroying* and removing of the history of an object is necessary. If an object is *destroyed* from a static database, there is no way to recall it later: the object is gone forever. In a temporal database, the object is not present in the database from the time of *destroy* onwards, while its past states may be referred to.

If we want to remove an object from all snapshots of the database, we need another type of operation, which neglects the historical concept of temporal databases. Such an operation is dangerous because it can cause an irrecoverable loss of data. It gives the opportunity to forge the history in an arbitrary way.

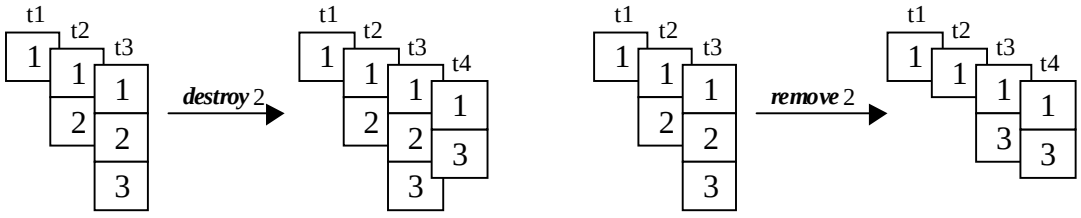

Figure 4.6: Destroying (left) vs. removing histories (right).

The left-hand side of Figure 4.6 shows the effect of the ***destroy*** operation: from the time-point t4 onwards, the object with the identifier 2 is not available, but its existence before t4 is preserved. The right-hand side of the same figure shows what happens if the objects with the identifier 2 are removed completely from the database: every track of its existence vanishes. Due to implementation constraints, we might at best conclude that the object with the identifier 2 existed, but the object with that identifier can not be retrieved.

4.2 Compositions of basic operations affecting identity of several objects

Basic identity operations and its simple composition *evolve* operate on a single object. Yet, there are many examples of change taking more than one object as arguments, or producing several objects as a result. Merging of two cadastral parcels produces a new,

third parcel. On the other hand, a single cadastral parcel, if divided, produces two or more parcels with new identifiers. Assembling all car parts produces a new object - a car. Disassembling the car produces, in general, the original parts with their old existence.

All possible combinations of basic operations are shown in Table 4.2. The operations affecting two or more identities come from the column *g*, and these are marked with the suffix "PL" (plural). The operations affecting single identity come from the row *f*. Operations are applied in such order that the creation is performed at the end, because it takes identifiers for predecessors.

f (one) g (many)	create	destroy	resume	suspend
createPL	⊥	fission	⊥	w-fission
destroyPL	fusion	⊥	w-fusion	⊥
resumePL	⊥	segregation	⊥	w-segregation
suspendPL	aggregation	⊥	w-aggregation	⊥

Table 4.2: Compositions of operations affecting multiple identities.

Signatures for "PL" operations are given as follows (brackets [] are the standard symbol for lists in functional languages):

$$createPL :: \Omega \rightarrow [ID]$$

$$destroyPL :: [ID] \rightarrow \Omega$$

$$suspendPL :: [ID] \rightarrow [ID]$$

$$resumePL :: [ID] \rightarrow [ID]$$

The compositions in the second and the third column of Table 4.2 are "constructive", because the object on the one-side is created or destroyed. The compositions in the fourth and the fifth column of Table 4.2 are called "non-constructive" or weak, because the one-side object is neither destroyed nor created. Non-constructive compositions are marked with the prefix "w-" (weak).

Depending on the operations applied on the PL-side, two groups of operations are distinguished: fusion/fission and aggregate/segregate group.

4.2.1 Fission and fusion

A (constructive) fission is the composition of destroying an object and creating a set of its successors at the same time. Emerging objects maintain a temporal link with the original object. A weak fission does not destroy the original object, but only suspend it.

A (constructive) fusion is the composition of destroying several objects and creating a new single object at the same time. The emerging object maintains a temporal link with the set of destroyed objects. A weak fusion does not create the new objects, but resumes an already existing one. The precondition for a weak fusion is a weak fission.

An example from a cadastral database involving *fission* and *fusion* of parcels is shown in Figure 4.7. The parcel 1 is *destroyed* and the new parcels (2 and 3) are *created*. Both new parcels maintain a temporal link to the parcel 1. At some later point, the two parcels are united again. The identifiers 2 and 3 are *destroyed*. The new parcel gets the new identifier (4), maintaining the temporal link with both of its predecessors.

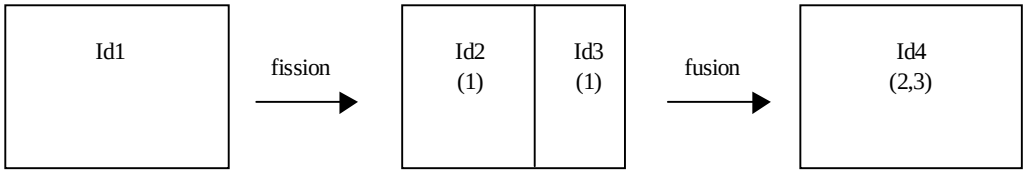

Figure 4.7: Fission and fusion of cadastral parcels with links to predecessors in parenthesis.

It is questionable whether the identity resulting from the fusion of a complete set of identities should be the same as the original identity which fissioned before. To give more flexibility to the designer, the concept of reversible fission and aggregation is introduced. The original identity must be *suspended* (instead of *destroyed*) to be *resumed*. This is modeled by *weak fission* and *fusion* (w-fission and w-fusion). As an example, consider a carafe full of water whose content is poured into two glasses. When the water is poured back to the carafe, the original liquid object with the identifier Id1 is resumed (Figure 4.8).

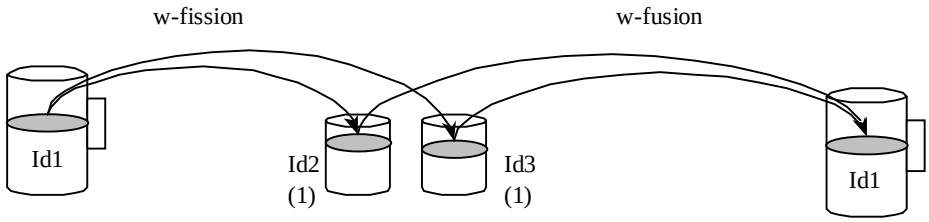

Figure 4.8: Weak fission and fusion of liquid objects.

Common to both types of fusion is the irreversibility of the *fusion* operation: fused identities are destroyed and cannot be re-used. Hence, identities of two cadastral parcels fused into one cannot be re-established; liquid objects in two glasses cannot be distinguished after having been poured into the carafe.

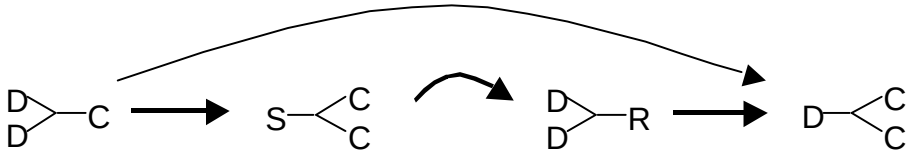

Figure 4.9: The lifestyle of fusions (D - destroy, C - create, S - suspend, R - resume).

The complete set of possible fusions and fissions is shown in Figure 4.9. The four distinctive operations are shown with the arrows indicating the order of their application to the set of objects:

First, in a constructive fusion (DD-C), the two (or more) objects are destroyed (DD) and a new object is created (C). Next, the resulting fused object can be suspended by a weak fission (S-CC), or destroyed by an immediate constructive fission (D-CC) - the longest arrow in Figure 4.9. In the former case, emerging objects may fuse again resuming the original object by a weak fusion (DD-R). At the end, the life of the fused object ends with a constructive fission. A practical example is an extension of the liquid example shown in Figure 4.8: if the carafe is empty in the beginning, a constructive fusion fills it up. Pouring the water into glasses is a weak fission. Pouring the water back into the carafe is a weak fusion. Finally, pouring the water on the floor is an irreversible, constructive fission. If we poured the water on the floor instead into the glasses, it would have the effect of the longest arrow in Figure 4.9.

4.2.2 Aggregation and segregation

Discussion about the identity and aggregation is connected with the part/whole relation. From the perspective of operations affecting object identity explained so far, only such

aggregations (and segregations) matter, which change identities of involved objects, i.e. perform one of the four basic operations. The relation member-of without influence on the identity is not considered as an aggregation in this thesis, see Figure 4.10. Such link is usually called association (Khoshafian and Abnous 1990).

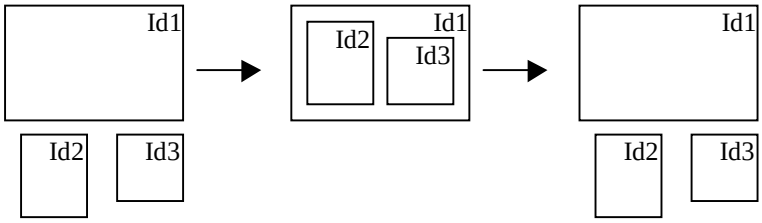

Figure 4.10: Association of objects and the reverse association.

An example of association is the membership of a person in a sport club: neither the identity of the person nor the identity of the club changes if the person leaves the club. The same goes for the associations of the states based on certain regional groupings (e.g., Scandinavian countries, Mediterranean states, etc.).

Hornsby and Egenhofer refer to both aggregations and associations as composite objects. Aggregations, based on the relation part-of, are formed from a framework - a predefined method of placing parts into “slots”. Collections, based on the relation member-of, are formed without framework, (Hornsby and Egenhofer 1998).

In this thesis, the aggregates are based on the relation part-of only. In addition, we assume an object can be a part of exactly one object, although this is true for physical things (the engine of one car cannot be in another car at the same time). The multiple levels of parthood can be modeled as hierarchies.

The example of constructive aggregation is a federation of several states (Hornsby and Egenhofer 1997). It is created by the political contract among the states in question. Federal government takes over certain representing functions (currency, foreign policy, defense) from the member-states. In that respect, member-states are suspended. Now, suppose that the federation breaks apart: its identity is destroyed, while the identities of member-states are resumed. A later re-union would have produced a new object.

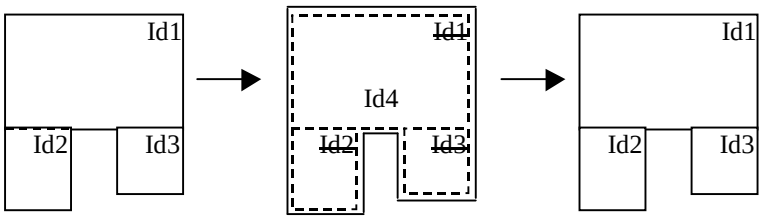

Figure 4.11: Constructive aggregation: the aggregate is a new object dependent on its parts.

An example of a weak aggregation (w-aggregate and w-segregate) is an episode in the life of a car. The identity of a particular car as a movable object emerges when all necessary parts are produced and properly connected. As long as the car functions, its parts do not have meaning outside the aggregate (car). If a part of the car is broken and needs to be repaired, the identity of the car is suspended (since the car does not function) and the identities of the parts are resumed. The broken part is repaired and all parts are aggregated again, resuming the original identity of the car. Even if one part is changed, the identity of the car is maintained.

The fundamental criteria for an aggregation to qualify for an identity affecting operation is the dependence of aggregated objects on the aggregate. If the aggregated objects are suspended and the aggregate is created for the first time, it is a constructive aggregation. If the aggregate already exists, it is a weak aggregation: the objects and the aggregate are mutually *suspended* and *resumed*.

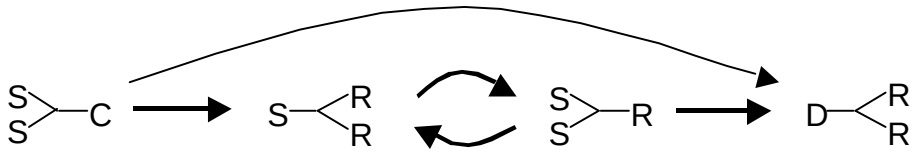

Figure 4.12: The lifestyle of aggregates (D - destroy, C - create, S - suspend, R - resume).

The complete set of possible aggregate and segregate operations is shown in Figure 4.13. It starts with a constructive aggregate when the parts are suspended giving birth to the new object. The new object could be destroyed in the next step (the long arrow) or suspended (weak segregation). The difference in respect to the previously explained fusion lifestyles is in the reversibility of weak aggregation/segregation. In the case of a weak fusion, objects on the many-side were destroyed and therefore not resumable. In the case of weak segregation, objects on the plural side are resumed, and can be suspended and resumed again.

A practical example of an aggregate is a car. When assembled for the first time, it is a constructive aggregation (SS-C). There are two alternatives for the next step: the

immediate destruction (D-RR) and a weak segregation (S-RR). An immediate destruction could be a car accident, after which remaining parts are individually used or sold. A weak segregation could be disassembling the car to repair malfunctioning parts. The identity of the car is resumed when the broken part is fixed or changed. Eventually, every car is destroyed, parts tend to live a little longer.

4.3 Object identity through time

This thesis merges together object oriented concepts with the temporal database framework. The concept of object identity is crucial for object orientation and for modeling processes in the dynamical world that surrounds us. The necessary condition for implementation of lifestyle operations (described in previous sections) in a temporal database is the proper dimension of time supported.

4.3.1 Transaction-time condition

The criteria for object identity (uniqueness, immutability, and non-reusability) can be satisfied in a database only if the system (database) is responsible for managing the object identifiers. This applies both for static and temporal databases, regardless of temporal dimension supported. If the user would have control of issuing new identifiers, the objects might have non-unique identifiers - the conditions would have been violated. The user is responsible for semantic decisions such as: when an object evolves (gets a new identity instead of an old one); is a particular object type destroyable or not; does a relationship apply between objects of specified classes; should an object be destroyed or suspended. The user must not decide, however, that an already destroyed object identity may be used again.

The criteria for the consistent behavior of identity in lifestyles framework are stronger. The crucial property for lifestyles is the ability of objects to maintain a temporal link with their predecessors. It is of the greatest importance in operations like *evolve*, *fusion*, *fission*, but a mere *creation* of an object needs a list of predecessors for the simplest model of parental relationship. In a static database, *destruction* of a parent object would leave the existing child object with a reference to a non-existing identifier (a dead pointer). The same happens in a database that tracks the valid time dimension only: histories of objects that are once removed from the database cannot be referred. Thus, removing a parent object would have the same effect in a valid time database as it

would have in a static database: the child object has a temporal link with an object, which cannot be queried for existence neither at the current time, nor at past times.

The transaction time is necessary for proper treatment of lifestyle operations, because the databases supporting the transaction time do not permanently remove destroyed objects. Such databases are append only - a new snapshot without the destroyed object is added. In both rollback and bitemporal databases (that track the transaction temporal dimension), the user can inspect the complete history of the database from its creation to the present moment. Even if an object is destroyed, the temporal links of its child objects are pointing to certain objects that can be referred and queried for their properties in the past times.

The concept of lifestyles can be implemented only in temporal databases that record transaction time: rollback databases and bitemporal databases. Static and historical (valid-time-only) databases cannot support lifestyle operations, because such databases are not capable of tracking temporal links among object identities.

4.3.2 Finiteness of the set of operations affecting object identity

A set of four basic operations affecting object identity was proposed (*create*, *destroy*, *suspend*, and *resume*). The only possible binary composition on a single object yielded the fifth operation - *evolve*. Operations can involve one or more objects in a single action. The basic operations are composed according to the cardinality of objects involved in change. The result was a group of eight compositions: constructive and weak fissions and fusions, aggregations and segregations. That makes the total of 13 possible operations affecting object identity.

What happens with many identifiers on both sides? Namely, some changes of multiple objects result in a multiplicity of objects as well. The process of land redistribution is well known in cadastre practice: a set of parcels is transformed into another set covering the same area. Oosterom grouped land redistribution, together with split and union, under restructuring processes involving several entities (van Oosterom 1997). Figure 4.13 shows a simplified view of such change.

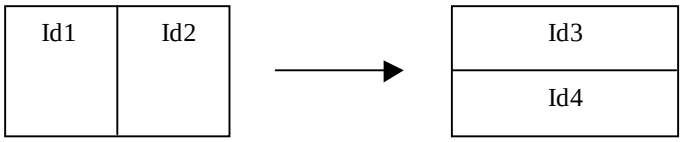

Figure 4.13: Redistribution of land parcels.

This anomaly is easily modeled as the composition of a *fusion* of identities 1 and 2 with a subsequent *fission* into two new identities.

restructure = fission2 . fusion (Id1,Id2)

The result of *fusion(Id1,Id2)* is a temporary identity *Id3*, which is immediately fissioned into 2 new identities. A query about the identity *Id3* would not have found any valid interval of its existence, but its predecessors are traceable as common predecessors of all emerged parcels.

Thus, we conclude that the set of operations affecting object identity is finite. There are 14 possible operations. Basic identity operations are: *create*, *destroy*, *suspend*, and *resume*. These operations build an algebra with the following axioms:

create . destroy = evolve

destroy . create = id

suspend . resume = id

resume . suspend = id

Further, compositions of basic operations define 9 new operations (the suffix “PL” means that the operation is applied to the group of two or more identities):

destroy . createPL = fission

create . destroyPL = fusion

suspend . createPL = w-fission

resume . destroyPL = w-fusion

create . suspendPL = aggregate

destroy . resumePL = segregate

resume . suspendPL = w-aggregate

suspend . resumePL = w-segregate

fission . fusion = restructure

These compositions extend the algebra with additional axioms:

w-fusion . w-fission = id

w-aggregate . w-segregate = id

w-segregate . w-aggregate = id

The axioms that are cited in this section are independent of the arguments (types of objects are irrelevant).

4.3.3 Comparison with the previous work

We compare our approach with the most detailed description of object identity change (Hornsby and Egenhofer 1997). Informal discussion given here is formalized in Section 7.3.

Hornsby and Egenhofer propose the following operations for manipulating single objects: `create` (without predecessors), `destruct`, `continue existence`, `continue non-existence`, `reincarnate` (as a `destruct` followed by a `create` of an object with the same identity, and two types of `issue` (creation of a new object from the existing one): `spawn` and `metamorphose`. In case of `spawn`, the original object continues to exist; in case of `metamorphose`, the original object is destroyed.

Our operation *create* covers both `create` and `spawn`, because an object can be created without or with predecessors. The operation *destroy* is conceptually similar to `destruct`. The logical model of our database ensures `existence` or `non-existence` of objects from one database version to the next, since all not affected objects are copied to the new state. The pair of operations *suspend* and *resume* explains the possibility of multiple episodes of existence of an object in a simpler way than a single operation `reincarnate`, which actually contradicts with the nature of permanent destruction. Finally, the operation *evolve* has the same meaning as the second type of issue - `metamorphose`.

For joining objects, Hornsby and Egenhofer proposed a number of operations: `merge` destroys the joined objects issuing the new object at the same time, `generate` does not destroy the original objects when a new object is issued (parenthood), `mix` issues a new object destroying one parent but not both, `aggregate` creates a new object from a set of individual objects that retain their identity, `unite` creates an aggregate of composite objects, `compound` adds a subpart to a composite object,

`amalgamate` merges the subparts of two composite objects yielding the new composite object with new parts, `combine` joins the two composite objects retaining the identity of their subparts.

In this thesis, the constructive *fusion* is identical to `merge`, *create* with predecessors is the same as `generate`. The composition of *creation* with parents as predecessors followed by *destroying* of one parent is equal to `mix`. The operation *aggregate* covers the semantics of both `aggregate` and `unite`. The composition of weak *segregation* and weak *aggregation* with one more object is identical to `compound`. The composition of *segregate* and *aggregate* is the adequate replacement for `combine`. Finally, the composition of *segregate, fusion* and *aggregate* is a model for the operation `amalgamate`.

Hornsby and Egenhofer gave five operations for splitting of objects: `splinter` separates a portion of the original object, which continues to exist; `divide` separates the original object, which ceases to exist, into *n* parts; `secede` separates a part from the composed object; `dissolve` completely splits a composite object into its constitutive parts; `select` allows for choice or selection of either the entire object or a portion of object.

In our model, the *create* with exactly one predecessor is equal to `splinter`. The constructive *fission* is exactly a `divide`. The composition of weak *segregation* and weak *aggregation* replaces `secede`, and constructive *segregation* is the same as `dissolve`.

To conclude, all types of change in identity discussed in (Hornsby and Egenhofer 1997), can be easily modeled with fewer operations. Besides the simpler compositional structure of our model, the concept of *suspend*/*resume* is better suited for modeling the multiple episodes in the existence of an object than other proposals. In addition, the concept of weak fission and weak fusion allows reincarnation of fissioned objects - an important property not covered in previous work.

4.4 Summary

In this chapter, we introduced operations that govern the change of identities in a spatiotemporal database. All operations are divided into two groups: basic operations and compositions. Basic identity operations are: *create*, *destroy*, *suspend*, and *resume*.

Their compositions operate either on a single object or on a group of objects. Composition of *destroy* followed by *create* is the only new operation among single object compositions and it is called *evolve*.

When several objects are considered, all possible operations are obtained as compositions of one basic operation on a single identity and another one operating on a group of two or more identities. If the single identity is destroyable, we have constructive compositions, if not we have weak compositions. If the many-side is destroyable, we have fusions, if not, we have aggregates. The fundamental difference between aggregation and fusion is the irreversibility of fission, while an aggregation is always reversible.

The criteria for object identity can be fulfilled in any database, regardless of temporal dimensions it support, assuming that the system controls the issuing of new identifiers. For the proper support to the lifestyle operations, the transaction time is necessary, because temporal links among an object and its predecessors can be preserved.

Since the compositions cover all possible cases of change in object identity, we conclude that the set of operations is finite. We compare our results with the work of other authors, concluding that our set of operations is more economical, conceptually simpler, and has more expressive power than other models.

All operations affecting object identity explained here are formalized in Chapter 7 in the context of an object-oriented temporal database, which is formalized in Chapter 6. The next chapter presents the formalization method: executable algebraic specifications written in the functional language, and the formalization tool: Gofer dialect of the functional programming language Haskell.

5. METHODOLOGY: ALGEBRAIC SPECIFICATIONS

In this chapter, I describe the method used for formalization: algebraic specifications written in a functional language. Algebraic specifications represent the necessary step between a conceptual model and its implementation, used to formally prove the correctness of the latter. Algebraic specifications are based on solid mathematical foundations (category theory) and mathematical methods can be applied to them.

Functional languages are formally defined: a compiler checks the syntax, completeness and other formal aspects of a program. Such programs are executable and can be used as a prototype. Gofer (Jones 1991), an experimental dialect of the non-strict and strongly typed functional programming language Haskell (Peterson et al. 1997), is used for formalization in this thesis. It unifies several advanced features from other similar languages: automatic type checking, user defined abstract data types, higher order functions, parameterized polymorphism and lazy evaluation. The Gofer code is compact, readable and portable. Abstract data types, polymorphism and inheritance, as implemented in Gofer, allow the specifications to be written in an object-oriented manner. After the machine has checked the syntax and the programmer checked the semantics by applying it to example cases, a specification can be easily translated into any other object-oriented environment.

This chapter is organized as follows: the first section is a short introduction in algebraic specifications stressing their importance in constructing programs; the second section is dedicated to general terms and concepts in functional programming; the third section explains Haskell syntax, to the extent necessary to understand specifications provided in this thesis.

5.1 Algebraic specifications

Algebraic specifications represent the necessary step between a conceptual model and its implementation, which is used to formally prove the correctness of the latter (Liskov and Guttag 1986). The purpose of a specification is to formally describe the behavior of objects. Algebraic specifications provide a clear and compact representation of theories for behavior of objects. They are based on solid mathematical foundations and mathematical methods can be applied to them.

Algebraic specifications were introduced to describe data abstractions (abstract data types) in software design (Guttag et al. 1978). The goal was to construct the axioms describing the behavior of data types independently of a particular implementation.

5.1.1 Definitions

An algebra is a description of a set of connected operations that are applied to a set of types. This is the generalized definition of algebra, introduced as "universal algebra" (Birkhoff 1945).

The algebraic specification consists of three parts (Ehrich et al. 1989):

- a set *S* of sorts (objects),
- a set Σ of operations applicable to this type, and
- a set E of axioms defining the behavior of these operations.

An algebraic axiom specification is defined by the triple (S, Σ, E), which represents an algebraic structure.

A *sort* is an element or object of a particular type. If the set *S* contains sorts of only one type, then we talk about *single-sorted* algebra. In a *multi-sorted* algebra, sorts of different types may occur. Multi-sorted algebras are used to build structured data types from more basic ones.

The set Σ contains operations applicable only to the sorts of *S*. Two kinds of operations exist (Liskov and Guttag 1986, Chapter 10): constructors and observers. Constructors are operations to create or modify a sort. Their result is an object of the defined sort. Observers are operations to observe properties of a sort. The result is an object of another sort (often Boolean). A minimal set of operations that are sufficient to generate all values of a sort is a set of *basic constructors*, whereas the minimal set of operations to retrieve these values is a set of *basic observers*.

A set *E* of *axioms* can be thought of as a set of rules which shows how each operation is applied to a sort. Axioms relate operations on sorts of the same type. An axiom states that an operation can be reduced or rewritten with some other operations while preserving its meaning.

5.1.2 Examples

We describe the familiar algebra of natural numbers, following the syntax from (Ehrich et al. 1989). Operations for addition, subtraction, negation and crating zero element are described by their signatures: types of arguments and the result. Axioms define the behavior of the operations are listed (two subsequent dashes "--" mean that the rest of the line is a comment).

```
Algebra AbelianGroup (number)
Operations:
         +, -   :: number -> number -> number
         negate :: number -> number
         0      :: number
Axioms:
        a + b = b + a               -- commutative law
        (a+b)+c= a+(b+c)= a+b+c     -- associative law
        0 + a = a + 0 = a           -- existence of identity
        a + (negate a) = 0          -- existence of inverse
        a - b = a + (negate b)      -- definition of subtraction
```

Algebras can be used to describe behaviors other than numbers, for example the properties of a stack. In such cases, more than one type is used and it is called multi-sorted or heterogeneous (Birkhoff and Lipson 1970).

A stack can accept elements pushed onto it. The operation *push* puts an element in a stack; the operation *top* returns the top element, the operation *pop* returns a stack with the top element removed. We show the parameterized algebra of stacks: the operations are independent of the type of *a*. Thus, the following specification is universal for all types.

```
Algebra Stack (stack of a, a)
Operations:
        empty :: stack of a                     -- constructor
        push :: a ->  stack of a -> stack of a  -- constructor
        pop :: stack of a -> stack of a         -- observer
        top :: stack of a -> a                  -- observer
Axioms:
        top (push a s) = a          -- a1
        pop (push a s) = s          -- a2
        top (empty) = error         -- a3
        pop (empty) = error         -- a4
```

Using the terminology from the previous subsection, we have defined an algebraic structure (S, Σ, E), where: S = {stack, a}, Σ = {empty, push, pop, top}, E = {*a1*, *a2*, *a3*, *a4*}.

The behavior of the operations *push*, *pop*, and *top* is fully explained by the axioms *a1*, *a2*, *a3*, and *a4*. The top element after pushing an element onto the stack is the element

that was pushed on. The stack that is returned after pushing something onto a stack and then applying a pop to the result is the same stack before the push operation.

Using the point-free notation, explained in Section 2.5.5, the axiom *a1* would be written as:

$$pop\ .\ push\ x = id$$

indicating that the combination of a *push* and a *pop* operation is the identity operation, which does not change the argument.

The axioms *a3* and *a4* yields an error as the result: the top of an empty stack is not defined. The operation *top* is then a partial function, undefined over an empty stack of any type. This operation can be made total by extending the set of carriers with a special element "error" that represents the "undefined value" (Loeckx et al. 1996). Then, the first line of the specification should be: `Algebra Stack (stack of a, a, error)`.

5.1.3 Advantages of algebraic specifications

In the process of developing reliable software, specifications are used for (Guttag et al. 1978):

- *design and implementation of abstract data types*:
  Algebraic specifications can capture the behavior of objects in a formal manner. It is possible to create complex types by using specifications of other, simpler data types. An important purpose of a specification is to organize types, values and operations that can be used for implementation.
- *proof that an implementation is correct*:
  It can be done by showing that the original axioms are satisfied by the implementation, which is probably the most important purpose of formal specifications.
- *early test*:
  If a specification is written in an executable programming language, it can be tested as a prototype (Frank and Kuhn 1995).

In this thesis, algebraic specifications are the essential meta-language for a formal description of lifestyles in order to communicate the information with potential implementors.

5.2 Functional programming

Functional programming languages are formally defined: a compiler checks the syntax, type completeness and other formal aspects of a program. Such programs are executable and can be used as a prototype. Furthermore, functional programming languages and algebraic specifications use a similar syntax and have similar mathematical foundations. Functional languages can express semantics and are easy to understand, which are the essential requirements for formal specification languages (Frank and Kuhn 1995). Since functional programming languages fulfill these requirements and allow for rapid prototyping in addition, they are used as specification and prototyping tools in this study.

Programming in a functional language consists of building definitions in the form of functions and using the computer to evaluate expressions (Bird and Wadler 1988). Definitions are constructed according to mathematical principles, and are expressed in notation that is similar to the traditional mathematical notation. If an expression possesses a well-defined value, then the order in which a computer evaluates the expression does not affect the result.

5.2.1 Functional vs. imperative languages

Most programming languages used today are imperative: the commands modify an implicit state. A typical example for implicit storing of a state is the assignment to a counter (e.g., $a := a + 1$). Examples of sequencing are *begin/end*, *while/loop*, and *goto* constructs.

In contrast to imperative programming languages, functional programming languages are declarative, i.e., there are no side effects and the programming is done with expressions rather than commands. If a functional language is completely free of side effects, it is called pure functional programming language. If some side effects exist, the language is impure.

An excellent comparison of imperative and functional programming languages is presented in the Turing Award lecture by John Backus (Backus 1978). Backus compared an imperative program for calculating the inner product with its functional counterpart. The imperative program was written in Pascal-like fashion:

```
c := 0
for i :=1 step 1 until n do
   c := c + a[i] * b[i]
```

The functional version of the program, translated to the standard Haskell notation by the author of this thesis, was:

```
innerproduct = foldr (+) 0 . map (foldr (*) 1) . transpose
```

Functions `(+)` and `(*)` are standard addition and multiplication; (.) denotes functional composition. Functions *foldr*, *map* are higher order functions defined in the standard prelude (library) of Haskell and explained in Section 5.2.6 below. The function *transpose* converts the rows of a matrix to its columns.

Backus concluded that the functional program has the following important advantages over its imperative counterpart:

- it operates only on its arguments,
- it is hierarchical, being built from simpler functions,
- it is static and nonrepetitive,
- it operates on whole conceptual units, not words,
- it incorporates no data; it is completely general, it works for any pairs of conformable vectors,
- it does not name its arguments,
- it employs forms and functions that are generally useful in many other programs.

The functional programs are computationally complete: any function can be expressed using those that are already defined. The most important elements of functional programming are referential transparency, strong typing and type inference, polymorphism, higher-order functions, pattern matching, and lazy evaluation. These elements are explained in the following subsections. First, we explain categorical combinators - the background for the point-free style of programming.

5.2.2 Categorical combinators

Categorical combinators are functions that reflect important concepts from category theory, thus enabling point-free style programming - description of a function exclusively in terms of functional composition. The combinators used in later chapters are described: categorical product, conditionals, and currying.

Categorical product, (see Figure 2.6) known as “cross-product” or “pairing” of two functions over a single argument is described with the following functions:

```
pair (f, g) a = (f a, g a)
outl (a, b) = a
outr (a, b) = b
```

These functions are related by the following properties

```
outl . pair (f, g) = f
outr . pair (f, g) = g
```

A pair of functions can be applied to a pair of arguments as well:

```
cross (f, g) (a, b) = (f a, g b)
```

The function *cross* can be expressed as composition of basic functions:

```
cross (f, g) = pair (f . outl, g . outr)
```

Some additional auxiliary combinators, related to the categorical product, are defined to simplify manipulation of functions: *swap*, *assocl*, *assocr*.

```
swap (a,b) = (b,a)
assocl (a,(b,c)) = ((a,b),c)
assocr ((a,b),c) = (a,(b,c))
```

The categorical product covers the cases when both functions are applied on an argument. The **conditional** operator covers the case when only one of the two functions is applied on an argument, depending on a predicate, as in the McCarthy conditional form *(p → f, g)* for writing conditionals. The conditional combinator is defined as:

```
cond p (f, g) a = if (p a) then (f a) else (g a)
```

Conditions can be combined with the functions ***meet*** and ***join***. Both functions originate from the lattice theory: *meet* is written as ∩, and *join* is written as ∪. The signature of both functions is:

```
meet, join :: (a -> Bool, a -> Bool) -> a -> Bool
```

The function *meet* returns True if and only if the second argument satisfies both conditions (relational *and*). The function *join* returns True if the second argument satisfies at least one condition (relational *or*):

```
meet ((>2),(<4)) 5 = False
join ((>2),(<4)) 5 = True
```

Functions of more than one argument can be defined in one of two basic styles: either by pairing the arguments, as in

```
plus (a, b) = a + b
```

or by **currying**, as in

```
cplus a b = a + b
```

The difference between *plus* and *cplus* is just one of type:

```
plus :: Num a => (a, a) -> a
cplus :: a -> a -> a
```

The function *curry* (after logician Haskell B. Curry) converts a non-curried function into a curried one, and the function *uncurry* does the inverse:

```
curry f a b = f (a, b)
uncurry f (a,b) = f a b
```

Curried functions are common in functional programming, because they usually lead to fewer brackets. We will, however, follow the advice from Bird (Bird and de Moore 1997) and use uncurried functions in cases when it leads to clearer definitions. The reason is that the product type `(a,b)` is a simpler object than the function type `a -> b` in an abstract setting.

5.2.3 Referential transparency

One of the properties of functional languages that are lost when side effects are introduced is referential transparency. The term *referentially transparent* refers to the style of programming where "*equals can be replaced by equals*" (Hudak, 1989, p.362). For example, in the expression (the valid Haskell syntax):

```
f x y = (a + 1) * (a + 2)
    where a = (x + y)/2
```

the application `(x+y)/2` created by the *where* expression may be substituted for any free occurrence of `a` such as in `(a+1)*(a+2)`. The substitution is possible because an expression (in our case the local definition of `a`) always denotes the same value. This is only guaranteed in the absence of side effects.

Referential transparency allows mathematical reasoning based on substitutions (equational reasoning). It permits mathematical proofs of program behavior, and is useful in writing and debugging programs.

5.2.4 Strong typing

Every object in a computer program has a type. The fundamental purpose of a type system is to prevent the occurrence of execution errors during the running of a program (Cardelli 1997). With the help of a *type inference mechanism* types of expressions can be inferred, when little or no type information is given explicitly (Cardelli and Wegner 1985; Milner 1978). For example, given some predetermined types (e.g., constants 1 and 2 are of the type *Int*), a type inference mechanism can logically deduce types of expressions (e.g., deducing from "*x*=1+2" that *x* is of the type *Int*).

The languages in which types are checked during program compilation are called strongly typed languages. The languages in which type checking is performed during program execution are called untyped languages. Miranda, ML, and Haskell are strongly typed languages, whereas LISP and Basic are untyped languages. Haskell support writing large program fragments where type information is omitted; the type system of Haskell automatically assigns types to such program fragments.

5.2.5 Polymorphism

A language is said to be polymorphic if its values and variables may have more than one type. A *polymorphic function* is a function that can be applied to arguments of different types. A *polymorphic type* is a type whose operations can be applied to other types. An example of a polymorphic function is addition of integers or floating numbers[1].

When the name of an operation is overloaded with two meanings it is *ad-hoc polymorphism* or *overloading*. In Haskell language, we have *parametric polymorphism*, since overloaded functions may be only applied to a predetermined range of types. How this is realized in Haskell-like languages is explained in Section 5.3.3.

5.2.6 Higher-order functions

The source and target types of functions in functional programming are not restricted in any way: functions can take any value as argument and yield any value as result. In

[1] In Haskell, the infix operator (+), with the signature `a->a->a`, is a member of the type class Num, which is, among other types, defined over the types Int and Float. Therefore, (+) 1 2 yields 3, (+) 1.0 2.0 yields 3.0.

particular, these values may themselves be functions. Hence, *a higher-order function* is every function, which has functions as its arguments or its result.

A mathematical example is the derivation function, which takes a function as the argument and yields its derivative (which is a function, too) as the result. In section 3.1, we have already seen the functional composition (.) - the most often cited example of higher-order functions in functional programming.

The standard example, the function ***map*** converts a function operating on elements to a function operating on lists of elements. Elements of lists are enclosed in square brackets and separated by commas. The type of *map* is given by:

```
map :: (a -> b) -> ([a] -> [b])
```

The source of *map* is a function of type `(a -> b)`, and the target is again a function having the type `([a] -> [b])`. For example:

```
map sqrt [1,4,9] = [1,2,3]
```

Another example is the function ***foldr*** (fold right), which recursively applies the given function on the result of the preceding application to the elements of a list. The informal description of *foldr*:

$$foldr\ f\ a\ [x_1, x_2, ..., x_n] = f\ x_1\ (f\ x_2\ ...\ (f\ a\ x_n)...))$$

The type of *foldr* is given by:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

An example is the summing of all elements in a list of integers. The function is (+), the start value is 0.

```
foldr (+) 0 [3,6,10] = 19
```

The third standard higher-order function is the function ***filter***, which returns the sublist of those elements of a list which satisfy the given predicate. The type of *filter* is given by:

```
filter :: (a -> Bool) -> [a] -> [a]
```

For example, if the predicate is `(<5)` having the type `(Int -> Bool)`, filtering the list of the first ten natural numbers yields:

```
filter (<5) [1,2,3,4,5,6,7,8,9,10] = [1,2,3,4]
```

The higher-order function ***flip*** swaps the arguments of a function:

```
flip f a b = flip f b a
flip (-) 3 2 = (-) 2 3 = -1
```

5.2.7 Pattern matching

In order to define new functions in functional programming, a programmer can choose between two possibilities: conditional equations or pattern matching. An example for a conditional equation is the following definition of standard factorial function:

```
fac n = if n = 0 then 1 else n * fac (n-1)
```

The same effect can be achieved by pattern matching:

```
fac 0 = 1
fac n = n * fac (n-1)
```

Pattern matching is one of the cornerstones of an equational style of definition. It leads to a cleaner and more readily understandable definition than a style based on conditional equations.

5.2.8 Lazy evaluation

Lazy (non-strict) evaluation is a technique of evaluating expressions that has two properties: no expression is evaluated until its value is needed, and no shared expression is evaluated more than once. The first of these ideas is illustrated by the following function:

```
ignoreArgument x = 3
```

Since the result of the function "ignoreArgument" does not depend on the value of its argument (x), that argument will not be evaluated. Shown here, the evaluation of the argument (1/0) gives 3.

5.3 Haskell and Gofer

A non-strict, lazy functional programming language Haskell, named after the logician Haskell Curry, is now widely regarded as the language of choice among lazy functional programming languages (Bird 1998). Its standardization is supported by the scientific community (Peterson et al. 1997), and the development is promising. The main impediment to its wider use in the past was the lack of simple portable interpreter of a huge Haskell compiler.

Gofer interpreter (Jones 1991) is an experimental dialect of Haskell. In addition to standard functionality of Haskell, Gofer supports multi-parameter type classes, a very important feature for complex modeling. Gofer is small, portable, stable, simple to learn

and use, and nevertheless powerful functional programming tool. These are the most important reasons for its popularity. Recently, Haskell Users' Gofer System (Hugs) became available, which unifies certain advantages of Gofer with conformity to Haskell standard. We decide to use Gofer for this thesis because of its stability - the version of Hugs matching the Gofer features is still in the testing phase.

Since Haskell is standardized, we describe its syntax and semantics. Each usage of Gofer features that differs from the standard Haskell will be marked. Further information about the basics and advanced topics of functional programming in Haskell can be found in the recent textbooks on that topic (Bird 1998), (Thompson 1999).

5.3.1 Layout rule

Readability of Haskell code is further improved by the *layout rule* - the level of indentation indicates the structure of a program. Non-indented lines represent top levels of a Haskell program. Every indentation shows that the indented line actually continues a previous, less-indented line. Equally indented lines share the same level in the structure. This rule allows the programmer to write long lines of code simply by breaking the line and indenting the rest and reduces the need for parentheses (like begin/end in a Pascal-like language.

5.3.2 Predefined and user-defined data type constructors

An identifier in Haskell begins with a letter of the alphabet optionally followed by a sequence of characters, each of which is either a letter, a digit, an apostrophe (') or an underbar (_). Identifiers representing functions or variables must begin with a lower case letter (identifiers beginning with an upper case letter are used to denote a special kind of function called a *constructor* function.

Several data types are predefined in Haskell standard prelude: integer (`Int`), floating point numbers (`Float`), Boolean values (`Bool`), characters (`Char`), lists (`[a]`), strings (`String`), and tuples (for example, `(a,b)` is a pair).

```
1 :: Int
1.0 :: Float
True, False :: Bool
'a' :: Char
```

If *a* is a type then [*a*] is a *list* whose elements are values of type *a*. Lists can be arbitrarily long, but all elements must be of the same type. There are several ways of writing list expressions:

- the simplest list of any type is the empty list, written `[]`;
- non-empty lists can be constructed either by explicitly listing the members of the list (for example: `[1,3,10]`) or by adding a single element onto the front of another list using the (:) operator (for example: `1 : 3 : 10 : []`).

A string is treated as a list of characters and the type *String* is simply an abbreviation for the type *[Char]*. *Strings* are written as sequences of characters enclosed between quotation marks (").

A product type (tuple) consists of a predefined number of elements of any kind. The examples are:

```
(1, 'a') :: (Int, Char),  -- pair
('a', 2, 1.0) :: (Char, Int, Float) -- triple
("Name",[1,2,21],(2,3),1) :: (String,[Int],(Int,Int),Int) -- quadruple
```

User-defined data types are declared by the keyword *data* together with type constructors. A *type constructor* is a function that constructs a new data type from other predefined data types. Constructors start with capitals. In the next example, the new data type *Person* is introduced by applying the constructor function *P* to an integer and a string.

```
data Person = P Int String
```

A sum type (enumeration) is represented as a series of values separated by a "`|`". The standard example is the definition of the days of the week:

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

A type synonym is an alias for an already existing data type. It is introduced by a keyword *type*. The new type *ID* behaves as the predefined type *Int* in our program.

```
type ID = Int
```

There are other predefined types and methods of introducing new types, but not relevant for the purpose of this thesis. Complete reference of Haskell data types and type synonyms can be found in (Peterson et al. 1997).

5.3.3 Classes and instances

A type class can be thought of as an algebra of types whose elements are called instances of the class (Jones 1991). It is used to model the behavior of a data type or a parameterized family of data types (Jones et al. 1997). In this section, we deal with simple classes that have a single parameter. Classes with multiple parameters are represented in Section 5.3.4 below.

To test the specifications, we need a representation and an implementation. In a class based functional programming language, these concepts are separated, leading to the following three notions: *class*, *data*, and *instance*. These three notions correspond to specification, representation and implementation of an abstract data type.

A ***class*** consists of a set of operations expressed by functions applied to a type (or types). In the class declaration, the first line (called the *class header*) states which class is defined, lists the parameters and may list conditions for the parameters. In the following lines the signatures of operations are given, describing the types of their arguments and of the result. In a Haskell signature, data types and type parameters before the last arrow ‘`->`‘ represent the arguments types, and the last one represents the type of the result.

The type class *Eq* is a simple and useful example, whose instances are precisely those types whose elements can be tested for equality. The declaration of this class given in the standard prelude is as follows:

```
class Eq a where
   (==), (/=) :: a -> a -> Bool
   x /= y = not (x == y)
```

The third line of the class declaration provides a default definition of the (`/=`) operator in terms of the (`==`) operator (similar to derived operations in terms of algebra). Thus, it is only necessary to give a definition for the (`==`) operator in order to define all of the member functions for the class *Eq*. It is possible to override default member definitions by giving an alternative definition as appropriate for specific instances of the class.

The ***data representation*** is constructed from predefined representations for basic types: integers (*Int*), floating point numbers (*Float*), and characters (*Char*). These can be combined as lists (a variable number of components of the same type) or records (a fixed number of components of different types). For most classes the representation is some sort of record, here for example consisting of a string and two integers:

```
data Point = Pt String Int Int
```

where a *String* represents the identifier of a point and the *Ints* are x and y coordinates.

Instances connect the data types with classes: they explain how the operations defined in a class are carried out using this particular representation. In our example, functions `(==)` and `(/=)` are *polymorphic*: they are applicable for each type that is an instance of the class *Eq*. We can freely choose the way in which points are compared for equality, for example just by testing for similarity of name:

```
instance Eq Point where
    (==) (Point n1 x1 y1) (Point n2 x2 y2) = (==) i1 i2
```

Finally, we need the physical realization of our model to test if the intended meaning is captured. These are created and initialized with the declarations like:

```
p1, p2, p3 :: Point
p1 = Pt 3 4
p2 = Pt 1 4
p3 = Pt 3 4
```

We can see that `p1 == p2` gives `False`, `p1 == p3` gives `True`, and so on. Frank and Kuhn used the similar approach to compare different approaches to point equality in the North-American Open GIS Consortium, where different systems cooperating in a heterogeneous environment can use different semantics in their equality operation (Frank and Kuhn 1995).

5.3.4 Classes with multiple parameters

Gofer is the first language to support the use of type classes with multiple parameters (Jones 1995), and thus the modeling of multi-sorted algebras. This feature allows further hiding of implementation issues from specification of functions. It is possible, for example, to define operations on two-dimensional points without specifying how coordinates are expressed (as integers or as floats):

```
class Points p a where
    getX :: p a -> a
    getY :: p a -> a
```

A representation (datatype *Point*) is also parameterized in the similar way.

```
data Point a = Pt a a
```

Finally, an implementation of the class *Points* on the datatype *Point* is defined as follows:

```
instance Points Point a where
    getX (Pt b c) = b
    getY (Pt b c) = c
```

The type of the result is not fixed and depends on the type of argument. If the parameter of *Point* is an integer, the result of the function *x* will be an integer.

The concept of inheritance is modeled within the context of a class. In addition, with multi-parameter classes, inherited behavior can be specified for each parameter. For example, we made no restrictions on the type of coordinates for the class *Points* above - the program would accept a string or character as well as any user-defined datatype. If we want only numbers as coordinates, we must add the context to the class declaration:

```
class Num a => Points p a where
    getX :: p a -> a
    getY :: p a -> a
```

The types for coordinates must be instances of the class *Num*. Any type that is not an instance of *Num* (e.g., a character) causes an error.

The testing examples are:

```
p4 :: Pt Int
p5 :: Pt Float
p4 = Pt 3 4
p5 = Pt 4.0 5.0
```

Then, `getX p4` gives 3 (Int), `getY p5` gives 5 (Float).

The multi-parameter classes are the key prerequisite for modeling many-sorted algebras. This advanced feature of Gofer is exploited in the rest of this thesis to hide implementation details (e.g., of representation of objects) in developing specification on the high level of abstraction (e.g., databases as collections of objects).

5.4 Summary

In this chapter, we presented the formalization method used in this thesis: algebraic specifications written in the functional programming language Gofer. The definitions of the main terms in algebraic specifications are given and clarified on simple examples. The advantages of executable specifications are readability, easy understanding and testing, and rapid prototyping.

The characteristics of the functional programming language Haskell are described with special attention paid to properties present in Gofer. Besides the most important syntax rules, which are necessary for reading the rest of this thesis, we stressed the more advanced concepts such as higher-order functions, type classes and classes with multiple parameters. These concepts are the building blocks for the formalization of an object-oriented temporal database in the next chapters.

6. SPATIOTEMPORAL DATABASE IN MODEL IMPLEMENTATION

The elements of a spatiotemporal database and the design decisions for a temporal model, described in Chapter 3, are formalized here, based on formalization concepts explained in Chapter 5. The result is a full-fledged executable model of a temporal database, which will serve as a starting point for modeling lifestyle operations in Chapter 7.

We begin with the specification of the database elements: objects with attributes and identifiers, value sets, values and relations, with a brief description of the entity-relationship data model (Chen 1976) introduced in Chapter 3, and continue with the formalization of snapshots and operations on full temporal databases. The second step is the choice of representation for abstract definitions given in the first step: we define appropriate datatypes for physical representation of database elements. In the third step, we connect the abstract specification with the representation in an implementation. The specification is applied on a simple example with queries that test if the model behaves as intended.

At the end of chapter, a formal description of transformation functions between the object versioning and the database versioning is given.

6.1 Data model for a temporal database

A data model is a model of the structure of the information system, independent of implementation details, and used as a basis for employing algorithms on the data. The goal of this section is to develop a formal model of a temporal database that is independent on implementation of objects and object types. We achieve such independence by attaching two additional parameters (datatypes for objects and object types) to all collections (snapshots and temporal database). Other elements of the model (attributes, value sets, values, identifiers, and relations) are not parameterized for the sake of clarity and simplicity. The data model is based on Chen's entity-relationship model (Chen 1976).

6.1.1 Object identifiers

Identifiers are modeled as an abstract class with the operations for producing the next (new) identifier (*nextID*), for observing the identifier (*getID*), and for the comparison of two identifiers (*sameID*). The operation for comparison of equality is inherited from the class *Eq*. For every implementation of the class *IDs* there must be an equality test for the datatype. Note that there is no constructor operation for setting an identifier to the abstract datatype. Identifiers are generated automatically by the function *nextID*, and cannot be arbitrarily changed for any implementation.

```
class Eq i => IDs i where
    nextID :: i -> i
    getID  :: i -> ID
    sameID :: i -> i -> Bool
    sameID i j = getID i == getID j
```

For simplicity, the identifiers are implemented as integers. Other structures would be possible. Natural numbers form an ordered set with equality defined. Peano's axioms guarantee that each new identifier is always different from all identifiers that are already issued, because $n + 1 > n$.

```
type ID = Int
instance IDs ID where
    nextID i = i + 1
    getID = id
```

The identifier of an object is unique for a whole database, because successors are always different. The identifier cannot be arbitrarily changed: it is not mutable. The source of new identifiers - the set of natural numbers - is theoretically infinite: old identifiers are not re-used. Therefore, all three conditions for identifiers, mentioned in Section 3.2.2, are fulfilled: uniqueness, immutability and non-reusability.

The class *IDs* can be instantiated not only on the object datatype, but on a collection of objects as well, in order to determine the latest identifier issued in a snapshot or a database.

6.1.2 Attributes, values sets and values

Attributes of objects are represented as value sets with assigned predefined types of values. According to Chen, an attribute can be formally defined as a function which maps from an entity set into a value set or a Cartesian product of value sets. (Chen

1976). Examples of attributes are a value set *Name* with a value of *String*, or a value set *Age* with a value of *Int*.

We model the attributes as the class *Attrib* with a constructor *attrib*, and observers *getValueSet* and *getValue*, that return the value set and value, respectively. The operation *setValue* updates the value of an attribute. Finally, the operation *selectAtt* extracts an attribute with the given value set.

```
class Attribs a where
    attrib      :: (ValueSet, Value) -> a
    getValueSet :: a -> ValueSet
    getValue    :: a -> Value
    setValue    :: Value -> a -> a
    selectAtt   :: ValueSet -> [a] -> a
    selectAtt s = head . filter ((s==).getValueSet)
```

The number of value sets is finite for a specific application domain. We define the datatype *ValueSet* that will cover the demonstrative purposes.

```
data ValueSet = Name | Age | Preds | Alive | Amount | Capacity | Weight
```

Chen assumed that there should exist direct representations of values (Chen 1976). Thus, values should be basic datatypes: characters, integers, and floats. We must wrap different types of values in a single datatype, because we need a list representation of attributes, and lists accept only elements having equal types.

```
data Value = Vs String | Vb Bool | Vi Int | Vf Float | Vp [Int]
```

There is a predicate associated with each value set to test whether a value belongs to it. This is modeled with the operation *checkV* in the class *ValueSets* with two parameters and an implementation over the datatypes *ValueSet* and *Value*. Usually, only a single type of value can be assigned to a particular type of value set, whereas each type value serves for several types of value sets.

```
class ValueSets vs v where
   checkV :: (vs, v) -> Bool
```

The operations for wrapping and unwrapping the basic types from and to the value datatype are defined in the class *Values* with the operations *wrapValue* and *unwrapValue*.

```
class Values v a where
   unwrapValue :: v -> a
   wrapValue   :: a -> v
```

Finally, we give the representation of attributes. It will be needed in definitions of the abstract specification of objects in the following subsection.

```
data Attrib = Att (ValueSet, Value)
```

After definition of identifiers, value sets and values, we proceed with the fully abstract definition of objects - the "first-class" citizens in our model.

6.1.3 Objects

The class representing the abstract data type of objects is parameterized in the object type and defines the operations for creation of a new object (*makeObj*), attaching the attributes to the object (*setAttribs*) and retrieving the object type and the list of attributes (*getObjtype* and *getAttribs*) respectively. These four operations are dependent of a particular representation of object datatype. The next four operations (for adding and updating a single attribute or several attributes at once) are defined by already known operations and do not depend on a particular implementation. Their default definitions are valid for every implementation of class *Objects*.

```
class IDs (o t) => Objects o t where
    makeObj    :: (t, ID) -> o t
    setAttribs :: [Attrib] -> o t -> o t
    getObjType :: o t -> t
    getAttribs :: o t -> [Attrib]

    addAtt     :: ValueSet -> Value -> o t -> o t
    addAtt s v = uncurry setAttribs . pair (f . getAttribs, id)
       where f = cons . pair (const (attrib (s, v)), id)

    addAtts    :: [(ValueSet, Value)] -> o t -> o t
    addAtts = (flip.foldr) (uncurry addAtt)

    updateAtt  :: Eq ValueSet => ValueSet -> Value -> o t -> o t
    updateAtt s v = uncurry setAttribs . pair (f . getAttribs, id)
       where f = updateBy ((s==).getValueSet) (attrib (s, v))

    updateAtts :: Eq ValueSet => [(ValueSet, Value)] -> o t -> o t
    updateAtts = (flip.foldr) (uncurry updateAtt)
```

The class *IDs* is mentioned in context, ensuring that for each implementation of the class *Objects* an implementation of the class *IDs* must exist. All functions are defined without the explicit naming of all their arguments, in a categorical point-free style. The functions *uncurry*, *pair*, *flip*, and *foldr* are explained in Chapter 5. The function *id* is the standard identity function; *cons* is the uncurried version of the list constructor (:) with the following meaning: *cons (1, [2,3]) = [1,2,3]; const* is the constant function - it takes two arguments and return the first one. The function *updateBy* replaces the elements of

a list that satisfy the given criteria with a given element: *updateBy even 6 [1,2,3,4] = [1,6,3,6]*.

6.1.4 Relations

Relations are represented as tuples consisting of a relation type and a pair of object identifiers. For the sake of the simplicity of notation, we introduce a type synonym *Rel* for representing relations:

```
type Rel = (RelType, (ID, ID))
```

The datatype for different relation types is *RelType*. In this chapter we will need only a spatial relation "On". The relation *PartOf* will be exploited in subsequent chapters.

```
data RelType = On | PartOf
```

A relation should be established only between appropriate types of objects, e.g., an engine cannot be a part of a table, but can be a part of a car. This is modeled with a class *Relatable*.

```
class Relatable t where
   relatable :: (RelType, (t, t)) -> Bool
```

The implementation of the operation *relatable* depends on the object type *t*.

6.1.5 Static database - a snapshot

Objects and relationships at a particular moment build a snapshot - a static database. A snapshot is modeled as an abstract datatype with the operations for manipulating objects and relations. Operations dependent on an implementation are: observers *getObjects* and *getRelations*, constructors *setObjects* and *setRelations*.

The operation *liftS* transforms a function that operates on a list of objects to a function that operates on the snapshot abstract type. For example, if a function *head* returns the first object from a list of objects, the function *headS = liftS head* will return the first object from a snapshot containing the list of objects. The similar operation to *liftS* is *liftR*, which "lifts" the function over a list of relations to a function over a snapshot containing the list of relations.

```
class (Objects o t, IDs (s o t), Relatable t) => Snapshots s o t where
    getObjects   :: s o t -> [o t]
    getRelations :: s o t -> [Rel]
    setObjects   :: [o t] -> s o t -> s o t
    setRelations :: [Rel] -> s o t -> s o t
```

```
liftS :: ([o t] -> [o t]) -> s o t -> s o t
liftS f = uncurry setObjects . pair (f . getObjects, id)

liftR :: ([Rel] -> [Rel]) -> s o t -> s o t
liftR f = uncurry setRelations . pair (f . getRelations, id)
```

A snapshot is a static database describing the universe of discourse in a particular moment. It is sufficient to define the representation of the snapshot datatype as a tuple consisting of a newest identifier, a list of objects and a list of relations:

```
data Snapshot o t = Snap ID [o t] [Rel]
```

The datatype for snapshots is parameterized for objects and object types. In order to represent change between the states in a database, a collection of snapshots is necessary.

6.1.6 Temporal database - a collection of snapshots

The class *TDBs* defines the abstract datatype of a collection of snapshots with only two operations: the observer *getSnapshots*, which retrieves the list of snapshots, and the constructor *setSnapshots*, which changes the list of snapshots.

```
class TDBs td o t where
    getSnapshots :: td o t -> [Snapshot o t]
    setSnapshots :: [Snapshot o t] -> td o t -> td o t
```

The crucial database operations are specified in the class *Databases*. All of these operations are polymorphic and can be applied either on a static database (a single snapshot) or on a collection of snapshots (abstractly defined in the class *TDBs*).

The operation *newObj* creates a new object in a database. Note that the only argument for this function is the object type (*t*). The identifier will be assigned automatically and cannot be changed by the user. The function *deleteObj* removes an object from the database; *updateObj* applies a function on an object that has the given identifier; *existObj* is a test if the object with the given identifier still exist; *selectObj* retrieves object from a database; *queryObj* returns a specific property of the object with the given identifier. The function *get* is a shortcut for retrieving the value of a given value set (the first argument of *get*) of an object (represented by its identifier). The default definition of *get* depends on the implementation of the function *queryObj*.

Functions over relations have the following meanings: *addRel* puts the given relation into a database; *deleteRel* removes a relation from a database; *addRels* adds several instances of the single relation type involving a single identifier (for example, several books are put on a table); *deleteRels* deletes all relations of a given type and a single identifier (for example, all objects that are *on* a particular table); *deleteRelsID*

removes all relations the given object participated in (if an object is removed from the database); *getRels* retrieves all identifiers that participate in a particular relation type with a given identifier; *getConvRels* retrieves converse relations.

```
class Snapshots d o t => Databases d o t where
    newObj    :: t -> d o t -> d o t
    deleteObj :: ID -> d o t -> d o t
    updateObj :: (o t -> o t) -> ID -> d o t -> d o t
    existObj  :: ID -> d o t -> Bool
    selectObj :: ID -> d o t -> o t
    queryObj  :: (o t -> x) -> ID -> d o t -> x

    queryObjs  :: (o t -> x) -> [ID] -> d o t -> [x]
    queryObjs q is = liftM (queryObj q) is

    get :: ValueSet -> ID -> d o t -> Value
    get a = queryObj (getValue . selectAtt a . getAttribs)

    addRel       :: ID -> RelType -> ID -> d o t -> d o t
    addRels      :: RelType -> [ID] -> ID -> d o t -> d o t
    deleteRel    :: RelType -> (ID,ID) -> d o t -> d o t
    deleteRels   :: RelType -> ID -> d o t -> d o t
    deleteRelsID :: ID -> d o t -> d o t
    getRels      :: RelType -> ID -> d o t -> [ID]
    getConvRels  :: RelType -> ID -> d o t -> [ID]

-- for queries (observers)
    liftQ :: TDBs d o t => (Snapshot o t -> x) -> d o t -> x
    liftQ f = f . head . getSnapshots

-- for updates (constructors)
    liftU :: TDBs d o t => (Snapshot o t -> Snapshot o t) ->d o t->d o t
    liftU f = h . cross (g, id) . pair (getSnapshots, id)
       where h = uncurry setSnapshots
             g = cons. pair (f . head, id)

-- for operations on a list of identifiers (map)
    liftM :: (ID -> d o t -> x) -> [ID] -> d o t -> [x]
    liftM f is = map (uncurry f) . cpl . pair (const is, id)
```

The "lift" operations have default definitions: *liftQ* transforms any query on a snapshot datatype to a query on an abstract type of temporal database; *liftU* transforms an update functions on a snapshot to an update function on an abstract type of temporal database; *liftM* applies a function of a single *ID* to a list of identifiers, returning a list of queried values. The great benefit of "lift" operations is that we have to define the implementation of operations only for snapshots.

The operation *liftU* is crucial for operations that create or change object identifiers. First, the collection of snapshots together with the complete database is retrieved with *pair (getSnapshot,id)*, and then the first element of the resulting list of snapshots (the

function *head* retrieves the first element of a list) is updated and added on the top of the ***unchanged*** original list (with the function *g*). The resulting list of snapshots is attached back to the original database. Thus, each update operation append a new snapshot to already existing list. There are no destructive updates. The temporal order of events is stored as the ordering of the snapshots.

So far, we specified all abstract classes necessary for a full-fledged temporal database. The model for representation of objects, object types, and temporal databases is presented in the next section.

6.2 Representation of objects, object types and temporal databases

The representations for attributes, values, value sets, relations, identifiers, and snapshots are already given. In this section, we give possible representations for parameters of classes in the previous section: objects and object types. Finally, a possible representation of temporal database is given.

An object will be represented as a tuple consisting of an identifier, an object type, and a list of attributes. The datatype *Object* is parameterized with respect to the object type. The same datatype for objects is retained throughout this thesis.

```
data Object t = Obj ID t [Attrib]
```

Object types are represented as enumerated datatype expressing a particular needs of the application domain. In this chapter, we present a simple example of a database consisting of two types of solid objects (a block world). In subsequent chapters, the parameterization of the database with respect to object types will be exploited for representation of various object classes.

```
data ObjType = Book | Table
```

Finally, our temporal database is the collection of snapshots: a simple datatype consisting of a list of snapshots.

```
data TDB o t = T [Snapshot o t]
```

The datatype *TDB* has two parameters: the first (*o*) for the object datatype and the second (*t*) for the datatype of object types. Thus, such representation is capable of representing various objects and object types without changing its implementation.

6.3 Implementation of the data model

In this section, we show an implementation of a simple, yet complete temporal database with two object types (books and tables) connected with a single relation (*On*). We start with the implementation by connecting the classes with datatypes in order the latter appeared in Section 6.1.

6.3.1 Implementation of values, value sets, and attributes

The implementation of the class *ValueSets* over datatypes *ValueSet* and *Value* is necessary to assure that values are always assigned to appropriate value sets.

```
instance ValueSets ValueSet Value where
   checkV (Name,     (Vs a)) = True
   checkV (Age,      (Vi a)) = True
   checkV (Amount,   (Vf a)) = True
   checkV (Capacity, (Vf a)) = True
   checkV (Preds,    (Vp a)) = True
   checkV (Alive,    (Vb a)) = True
   checkV (Weight,   (Vf a)) = True
   checkV _ = False
```

In this chapter, we will use only the value set *Name*. Therefore, the wrappers are defined only for string values.

```
instance Values Value String where
   unwrapValue (Vs s) = s
   wrapValue s = Vs s
```

The implementation of operations in the class *Attribs* over the datatype *Attrib*:

```
instance Attribs Attrib where
   attrib = cond checkV (Att, error "incompatible value types")
   getValueSet (Att (s,v)) = s
   getValue    (Att (s,v)) = v
   setValue v  (Att (s,u)) = attrib (s,v)
```

The implementation of the constructor function *attrib* checks the compatibility of its arguments (a value set and a value), and returns an error message if the types are not compatible.

6.3.2 Implementation of objects and relations

The class *Objects* can be instantiated over the parameterized datatype *Object t* without concrete implementation of object type *t*.

```
instance Objects Object t where
    makeObj (t,i) = Obj i t []
    getObjType (Obj i t as) = t
    getAttribs (Obj i t as) = as
    setAttribs as (Obj i t bs) = Obj i t as
```

Since the class *IDs* was in the context of the class *Objects*, it must be instantiated for *Object t*, too.

```
instance IDs (Object t) where
    sameID a b = sameID (getID a) (getID b)
    getID (Obj i t as) = i
```

The representation of relations is already defined in Section 6.1.4, but the class *Relatable* should be implemented over the datatype *ObjType*. The only valid relation in our simple database is between object types *Book* and *Table* (in that order), and its type is *On.*

```
instance Relatable ObjType where
   relatable (On, (Book, Table)) = True
   relatable _ = False
```

The class *Relatable* can have different instantiations for different representation of object types. We will take advantage of this in the following chapters.

6.3.3 Implementation of snapshots

The datatype *Snapshot* is an instance of classes *IDs*, *Snapshots*, and *Databases*. The first instance implements the most important operation of all: *nextID*. The new identifiers are issued by a snapshot and triggered each time a new object is created. The observer *getID* retrieves the latest identifier issued.

```
instance IDs (Snapshot o t) where
    getID (Snap i os rs) = i
    nextID (Snap i os rs) = Snap (nextID i) os rs
```

The instance of the class *Snapshots* is simple:

```
instance Snapshots Snapshot o t where
    getObjects (Snap i os rs) = os
    setObjects os (Snap i ps rs) = Snap i os rs
    getRelations (Snap i os rs) = rs
    setRelations ts (Snap i os rs) = Snap i os ts
```

All update operations in a database are actually defined over a snapshot as the instance of the class *Databases* over the parameterized type *Snapshot o t.* The operation *newObj* that creates a new object in a snapshot triggers the *nextID* for a snapshot and produces a new object with the new identifier. The operation *deleteObj* removes all relations the

object participated in. Other operations over objects have the behavior already explained in Section 6.1.6.

```
instance Databases Snapshot o t where
    newObj t = nextID . uncurry setObjects .
       cross (cons . pair (makeObj.outl, outr), id) .
       cross (assocl. pair (const t, id), id) .
       pair (cross (getID, getObjects), outr) . pair (nextID, id)

    existObj i = cond p (false, true) where
       p = null . filter ((i==).getID) . getObjects

    deleteObj i   = liftS f . liftR g where
       f = filter ((i/=).getID)
       g = filter (meet ((i/=).outl.outr, (i/=).outr.outr))

    updateObj f i = cond (existObj i) (g, h) where
       g = liftS (map (cond ((i==).getID) (f, id)))
       h = error ("the object " ++ show i ++ " does not exist.")

    selectObj i = (cond existObj i) (f, g) where
       f = head . filter ((i==).getID) . getObjects
       g = error ("the object " ++ show i ++ " does not exist.")

    queryObj q i  = q . selectObj i
    queryObjs q is = liftM (queryObj q) is

    addRel j t i = cond p (f, g) where
       p = relatable . pair (const t, pair (h i, h j))
       h a = queryObj getObjType a
       f = liftR(cons . pair (pair (const t, pair (const i,const j)),id))
       g = error "types are not relatable."

    addRels t is j = (flip . foldr) (addRel j t) is
    deleteRel t is = liftR (filter (join' ((t/=).outl, (is/=).outr)))
    deleteRels t i = liftR(filter (join' ((t/=).outl, (i/=).outr.outr)))
    deleteRelsID i = liftR(filter(meet((i/=).outl.outr,(i/=).outr.outr)))

    getRels t i   = map (outl.outr) . filter p . getRelations where
        p = meet ((t==).outl, (i==).outr.outr)

    getConvRels t i = map (outr.outr) . filter p . getRelations where
        p = meet ((t==).outl, (i==).outl.outr)
```

Among operations over relations, adding a new relation includes a check if the object types are relatable. The function *join'* is relational *or* already described in Chapter 5 (the apostrophe is added to avoid the name clash with the function *join* from the standard Gofer prelude).

6.3.4 Implementation of a temporal database

The datatype *TDB* implements following classes from our model: *IDs*, *TDBs*, and *Databases*. The instantiation of the class IDs enables the query about the latest identifier

in the whole database. Since the operation *liftQ* is necessary, the class *Database* must be mentioned in context and an instantiation of the class *Database* over the datatype *TDB* must be provided.

```
instance (Databases TDB o t) => IDs (TDB o t) where
    getID = liftQ getID
```

Further, the operations in the class *TDBs* are implement to enable retrieval and updating of a list of snapshots.

```
instance TDBs TDB o t where
   getSnapshots (T ss) = ss
   setSnapshots ss (T ts) = T ss
```

Finally, the implementation of the class *Databases* is surprisingly simple: all functions are transformed with appropriate lift operations to the functions operating on the latest snapshot.

```
instance (TDBs TDB o t, Databases Snapshot o t)
   => Databases TDB o t where
   newObj t       = liftU (newObj t)
   deleteObj i    = liftU (deleteObj i)
   updateObj f i  = liftU (updateObj f i)
   existObj  i    = liftQ (existObj i)
   selectObj i    = liftQ (selectObj i)
   queryObj q i   = liftQ (queryObj q i)

   addRel j t i   = liftU (addRel j t i)
   addRels t is j = liftU (addRels t is j)
   deleteRels t i = liftU (deleteRels t i)
   deleteRelsID i = liftU (deleteRelsID i)
   getRels t i    = liftQ (getRels t i)
   getConvRels t i= liftQ (getConvRels t i)
```

Thus, we finished with the implementation of our data model. All classes defined in Section 6.1 are connected with the representation types. Details about implementation of standard type classes (*Eq*, *Text*, *Num*) are omitted and can be found in the Appendix.

6.4 An example database

In this section, we show a full example of a simple temporal database. Behavior of all functions introduced in data model is tested. The universe of discourse that serves as the test-bed consists of two books named "bookA" and "bookB" and two tables named "tableA" and "tableB". Beside the population of the database with these four objects, we will test the capability of the model to prevent illegal operations like putting an already deleted book on the table.

First, we populate our database starting from an empty database *td0* with the following piece of code:

```
td0, td1, td2, td3, td4, td5, td6, td7 :: TDB Object ObjType
td0 = T [Snap 0 [] []]
td1 = foldr newObj td0 [Table, Table, Book, Book]
td2 = updateObj (addAtt Name (Vs "Book1")) 1 td1
td3 = updateObj (addAtt Name (Vs "Book2")) 2 td2
td4 = updateObj (addAtt Name (Vs "Table1")) 3 td3
td5 = updateObj (addAtt Name (Vs "Table2")) 4 td4
td6 = addRel (On, (1,3)) td5
td7 = addRel (On, (2,4)) td6
```

The result is the following state of the database (only the latest snapshot) represented using a simple implementation of Gofer type class *Text*:

```
? liftQ show td7

Snapshot
 Latest ID =4
 Objects: [
  4 Table  Attribs:[ name   = "Table2"],
  3 Table  Attribs:[ name   = "Table1"],
  2 Book   Attribs:[ name   = "Book2"],
  1 Book   Attribs:[ name   = "Book1"]]
 Relations: [2 is on 4,1 is on 3]
```

Thus, we conclude that the operations *newObj*, *updateObj*, and *addRel* show the intended behavior. Several tests can be performed on the final state *td7* and we will show possible actions and results as comments. For each test operation, the expected type is specified explicitly to avoid type errors. We use identifiers for referring the objects, because identifiers are guaranteed to be unique. If unique names are given to all objects, it would be possible to refer the objects in a more natural manner - by using their names.

```
tst1, tst2 :: Bool
tst3 :: Object ObjType
tst4 :: Value
tst5 :: String
tst6 :: Object ObjType
tst7 :: [Rel]
tst8 :: TDB Object ObjType

tst1 = existObj 4 td7
-- True

tst2 = existObj 4 (deleteObj 4 td7)
-- False

tst3 = selectObj 4 (deleteObj 4 td7)
-- error: the object 4 does not exist.
```

```
tst4 = get Name 1 td7
-- Vs "Book1"

tst5 = unwrapValue (get Name 1 td7)
-- Book1

tst6 = selectObj 3 td7
-- Obj 3 Table [Att (Name,Vs "Table1")]

tst7 = liftQ getRelations (deleteRel On (1,4) td7)
-- [(On, (2,4))]

tst8 = addRel (On, (4,1)) td7
-- error: types not relatable.
```

The first test shows that the function *existObj* gives the expected result for an existing object. The second and third test show that the deleted objects do not exist and cannot be selected from the latest snapshot. Two examples of querying existing objects are shown in *tst4* and *tst5*. A successful selection of an object is shown in *tst6*. The last two tests deal with relations: *tst7* shows which relations remain in the database after a successful deletion of an existing relation, and *tst8* shows what happens if we attempt to put a table on the book (a undefined relation).

6.5 Formal model of transformations between versioning techniques

In Section 3.3.2 we claimed that the transformations between two versioning techniques (object versioning and database versioning) are lossless. We give the formal model for transformations and show on a simple example that any transformation composed with the inverse transformation return the original database.

6.5.1 Specification

All functions are defined in the class *Groups*, which has two parameters: *t* for representation of time and *o* for representation of objects. The function *toOV* transforms a database versioning model to an object versioning model. The function *toDV* is the inverse operation to *toOV*. Both functions are composed in five steps explained in Section 3.3.2: distribute, find, select, normalize, and concatenate.

```
class (Eq t, Eq o) => Groups t o where
-- (database versioning -> object versioning)
  distrTime :: [(t,[o])] -> [(o,t)]
  distrTime = concat . map cpl . map swap

  findObjs :: [(o,t)] -> [o]
  findObjs = nub . map outl
```

```
-- select times for given object

  selTimes :: (o, [(o,t)]) -> [(o,t)]
  selTimes = uncurry filter . cross (flip ((==).outl), id)

  normObj :: [(o,t)] -> (o,[t])
  normObj = pair (head . map outl, map outr)

  toOV :: [(t,[o])] -> [(o,[t])]
  toOV = map (normObj.selTimes) . cpl
       . pair (findObjs, id) . distrTime

-- the opposite case (object versioning -> database versioning)
  distrObjs :: [(o,[t])] -> [(o,t)]
  distrObjs = concat . map cpr

  findTimes :: [(o,t)] -> [t]
  findTimes = nub . map outr

-- select objects at given time
  selObjs :: (t,[(o,t)]) -> [(o,t)]
  selObjs =  uncurry filter . cross (flip ((==).outr), id)

  normTime :: [(o,t)] -> ([o],t)
  normTime = pair (map outl, head . map outr)

  toDV :: [(o,[t])] -> [(t,[o])]
  toDV = map (swap.normTime.selObjs). cpl
       . pair (findTimes, id) . distrObjs
```

Beside several already seen functions, functions for Cartesian products (*cpl* and *cpr*) and the function *nub* deserve additional explanation[1].

6.5.2 Representation of time and objects

Time is represented with integers. Objects are simplified to a tuple consisting of an identifier, an object type and a single attribute (color).

```
type Time  = Int
data ObjX   = Ob ID ObjT Color
data ObjT  = House | Car
data Color = Red | Blue | White
```

6.5.3 Implementation

An instance of the class *Eq* is necessary for the datatypes *ObjX* and *Color* to compare objects for equality:

---

1 Cartesian product left (cpl) pairs a list of values with a single value: *cpl* ([1,2,3],4)=[(1,4), (2,4), (3,4)]. Cartesian product right (cpr) pairs a single value with a list of values: *cpr* (4,[1,2,3])=[(4,1), (4,2), (4,3)]. The function *nub* removes duplicates from a list: *nub* [1,2,3,3,2]=1,2,3.

```
instance Eq ObjX where
  (==) (Ob i t c) (Ob j u d) = i == j && c == d

instance Eq Color where
  Red == Red = True
  Blue == Blue = True
  White == White = True
  _ == _ = False
```

Since all functions of the class *Groups* have default definitions, we need just to connect the class with the representations for time and objects.

```
instance Groups Time Obj
```

In the next subsection, we test the model on the simple example mentioned in Section 3.3.2.

6.5.4 Examples

We construct the objects *House* and *Car*, and prepare a list of snapshots representing the universe of discourse.

```
o1, o2, o3 :: ObjX
o1 = Ob 1 Car Red
o2 = Ob 1 Car Blue
o3 = Ob 2 House White

dv1, dv2 :: [(Time, [ObjX])]
dv1 = [(1, [o1]), (2, [o2,o3]), (3, [o2,o3]), (4, [o3])]

ov1 :: [(ObjX,[Time])]
ov1 = toOV dv1

dv2 = toDV ov1
```

The results of executing tests *dv1*, *ov1*, and *dv2* are:

```
dv1 =
[(1,[redCar]),(2,[blueCar,whiteHouse]),
 (3,[blueCar,whiteHouse]),(4,[whiteHouse])]

ov1 =
[(redCar,[1]),(blueCar,[2,3]),(whiteHouse,[2,3,4])]

dv2 =
[(1,[redCar]),(2,[blueCar,whiteHouse]),
(3,[blueCar,whiteHouse]),(4,[whiteHouse])]
```

We can see that the application of the transformation function *toDV* after the transformation function *toOV* yields the original database. The composition of transformation functions is equal to the identity function. Transformations between two versioning techniques are lossless.

6.6 Summary

An entity-relationship model of a working temporal database is formally described and an executable specification is provided. The concepts of identifiers, attributes, value sets, values, objects, snapshots, and temporal databases are formalized as classes in Gofer. The object identifiers are issued by the system during the creation of new objects only, and cannot be arbitrarily changed after the creation of objects. Thus, the conditions of uniqueness, immutability and non-reusability of identifiers are satisfied. The relations are valid only if the objects represented in relations by their identifiers have appropriate types. A simple instantiation is made for a small database and the functionality of the operations is tested. Finally, we formalized the algorithms for transformations between versioning techniques presented in Section 3.3.2. We showed that these transformations are lossless.

In the next chapter, we build the model for operations affecting object identity on top of the spatiotemporal database presented here.

7. OPERATIONS AFFECTING OBJECT IDENTITY - A FORMAL MODEL

In previous chapter, we develop an executable functional specification of a full-fledged temporal database based on the entity-relationship data model. On top of that specification, we formalize lifestyles - classes of operations affecting object identity, described in Chapter 4. Lifestyle operations are completely independent of the representations for objects and object types. At the end, we give a comparison with other prominent proposals for categorizations of operations that change object identity. The implementations and applications of lifestyles are given in Chapters 8 and 9.

7.1 Operations affecting single identity

The basic operations that affect the identity of a single object (*create*, *destroy*, *suspend*, *resume*, and *evolve*) are formalized in separate Gofer classes with appropriate classes from the previous chapter in the context.

7.1.1 Create

In the previous chapter, the operation *newObj* for producing a new object in the database was introduced in the class *Databases*. The operation *create* maintains a set of temporal links with predecessors. Therefore, a list of predecessors is a mandatory argument of the operation *create*. Predecessors can be set only during the creation of the object. Objects, which are created from scratch, have an empty list of predecessors.

Two auxiliary operations are defined: *createWithID* and *createN*. The former operation returns a pair consisting of the newly created identifier and the updated database, while the latter creates several new objects with the same object type and with the same set of predecessors, yet with different identifiers. The number of created objects with the function *createN* is determined by its first argument (*n*).

```
class (Objects o t, IDs (d o t), Databases d o t)
   => Creatable d o t where
   create          :: ([ID], t) -> d o t -> d o t
   createWithID    :: ([ID], t) -> d o t -> (ID, d o t)
   createN         ::  Int -> ([ID], t) -> d o t -> d o t

   create (is, ot) = uncurry (updateObj f) . pair (getID, id) . newObj ot
      where f = addAtt Preds (Vp is)
   createWithID (is, ot) = pair (getID, id) . create (is, ot)
   createN n (is, ot) = flip (!!) n . iterate (create (is, ot))
```

The operations updateObj, *getID*, and *newObj* are inherited from the classes *Objects IDs*, and *Databases*, respectively. Two standard operations over lists are used for the definition of the operation *createN*: *iterate* and (!!)[1].

7.1.2 Destroy

The operation *destroy* is not applicable to all object types. In order to be destroyable, a specific object type must implement the method *destroyable* from the class *DestroyableT* ('T' stands for type):

```
class DestroyableT d where
   destroyable :: d -> Bool
```

This class is then added to the context of the class *Destroyable*, ensuring that for each implementation of class *Destroyable* (parameterized in object type *t*) an instance *DestroyableT t* exists.

```
class (DestroyableT t, Creatable d o t) => Destroyable d o t where
   destroy :: ID -> d o t -> d o t
   destroy i = cond p (f, g) where
      p = destroyable . getObjType . selectObj i
      f = deleteObj i
      g = error ("the object" ++ show i ++ "is not destroyable")
```

The prerequisite that the object type is destroyable is stated in the condition *p* for the operation *deleteObj* inherited from the class *Databases*. Although the class *Databases* is not explicitly mentioned in the context of the class *Destroyable*, it is implicitly inherited from the class *Creatable*, which is in the context. The operation *destroy* has the following effect: the object is removed from the latest snapshot; it cannot actively participate in further changes to the database, but its previous existence can be referenced. All relations in which the object had participated are removed as well.

7.1.3 Suspend and resume

The operations *suspend* and *resume* are mutually dependent. It is natural to model both of them in the single class *Suspendable*. Operations *suspendObj* and *resumeObj* operate on object level.

---

1 The function *iterate* generates an infinite list by iteratively applying a function on the last element: *iterate (+2) 1 = [1,3,5,7,...]*. The function (!!) n selects the n^{th} element of the list. (!!) ['a','b','c'] 1 = 'b' (indexing starts from 0).

A predicate *suspendable* is needed to check if an object type can be suspended or not. The predicate is defined in the class *SuspendableT* which must be instantiated for each object type.

```
class SuspendableT s where
   suspendable :: s -> Bool
```

On the object level (class *SuspendableO*), the predicate *suspended* checks whether an object is already suspended, whereas the functions *suspendObj* and *resumeObj* change the attribute *Alive* of the object.

```
class Objects o t => SuspendableO o t where
   suspended    :: o t -> Bool
   suspended = not .unwrapValue . getValue . selectAtt Alive . getAttribs

   suspendObj   :: o t -> o t
   suspendObj = updateAtt Alive (Vb False)

   resumeObj    :: o t -> o t
   resumeObj = updateAtt Alive (Vb True)
```

Finally, the operations *suspend* and *resume* push the operations *suspendObj* and *resumeObj*, respectively, to the database level. The class *Suspendable* needs both classes *SuspendableT* and *SuspendableO*, together with the class *Creatable* in its contexts.

```
class (SuspendableT t, SuspendableO o t, Creatable d o t)
   => Suspendable d o t where

   suspend, resume :: ID -> d o t -> d o t
   suspend i = cond p (f, g) where
      p = suspendable . getObjType . selectObj i
      f = updateObj suspendObj i
      g = error ("the object" ++ show i ++ "is not suspendable")

   resume i = cond p (f, g) where
      p = queryObj suspended i
      f = updateObj resumeObj i
      g = error ("the object" ++ show i ++ "is already suspended")
```

The operations *suspend* and *resume* are defined conditionally: an object can be suspended if it is suspendable and can be resumed only if it is already suspended. All operations on the database level are completely independent of the implementation at the cost of a complex set of conditions for the type parameters in the context.

7.1.4 Evolve

The evolvable objects must implement the class *Destroyable* as can be seen in the context. Any later instantiation is independent of the instantiation of the database.

```
class Destroyable d o t => Evolvable d o t where
   evolve :: ID -> d o t -> d o t
   evolve i = destroy i . uncurry (updateObj' setAttribs)
     . pair (pair (getID, getAttribs. selectObj i), id) . uncurry create
     . assocl. pair (const (wrap i), pair (getObjType . selectObj i, id))
        where  updateObj' f (i,x) = updateObj (f x) i
```

The new object is created with the original object as the predecessor: the code fragment *const (wrap i)* produces a singleton list *[i]* that is used as the argument for the operation *create*. All attributes of the original objects are transferred to the emerging object as well, leading to a complex definition.

7.2 Operations affecting multiple identities

Compositions of basic operations are modeled with the following four classes: *Aggregates*, *WAggregates*, *Fusions* and *WFusions*. Classes with the prefix 'W' cover the weak or non-constructive cases of fusions and aggregation.

7.2.1 Constructive aggregates

The constructive aggregates must implement the classes *Suspendable* and *Destroyable*. The operation *aggregate* suspends the objects having the identifiers from the given list, creates a new object with the given object type, and establishes the relation *PartOf* among the suspended objects and the newly created object.

```
class (Destroyable d o t, Suspendable d o t)
   => Aggregates d o t where
   aggregate :: [ID] -> t -> d o t -> d o t
   aggregate is t = uncurry (addRels PartOf is) . createWithID ([],t)
                    . (flip.foldr) suspend is
   segregate   :: ID -> d o t -> d o t
   segregate i = (uncurry.flip.foldr) resume . pair (getRels PartOf i, g)
      where g = deleteRels PartOf i . destroy i
```

The operation *segregate* first searches all identifiers that are parts of the given identifier, resumes matching objects, removes relations *PartOf*, and destroys the object with the given identifier.

7.2.2 Weak aggregates

Weak or non-constructive aggregations and segregations does not destroy objects. Therefore, such objects are only *creatable* and *suspendable*. It is sufficient to add the class *Suspendable* to the context, because the class *Creatable* is inherited implicitly.

```
class Suspendable d o t => WAggregates d o t where
   waggregate :: [ID] -> ID -> d o t -> d o t
   waggregate is i = (flip.foldr) suspend is.addRels PartOf is i.resume i
   wsegregate   :: ID -> d o t -> d o t
   wsegregate i = (uncurry.flip.foldr) resume . pair (getRels PartOf i,g)
      where g = deleteRels PartOf i . suspend i
```

The operation *waggregate* differs from *aggregate* only in that the aggregated object is resumed instead of created. Similarly, the operation *wsegregate* differs from *segregate* in that the segregated object is suspended instead of destroyed.

7.2.3 Constructive fusions

Constructive fusions are not reversible. Thus, the class *Destroyable* is the only class necessary in the context. The operation *fusion* creates a new object with the given object type and destroys the objects having identifiers from the given list. The new object has destroyed objects as predecessors.

```
class Destroyable d o t => Fusions d o t where
   fusion :: [ID] -> t -> d o t -> d o t
   fusion is t = (flip . foldr) destroy is . create (is, t)
   fissionN :: Int -> ID -> d o t -> d o t
   fissionN n i = uncurry (createN n) . pair (f, destroy i)
      where f = pair (wrap . const i, getObjType . selectObj i)
   restructure :: [ID] -> t -> Int -> d o t -> d o t
   restructure is t n = uncurry (fissionN n).pair (getID,id).fusion is t
```

The operation *fissionN* creates *n* new objects that all have the same type as the original object. Each of *n* new objects has exactly single predecessor - the identifier of the original objects. Finally, the operation *restructure* is modeled as a composition between a fusion and a subsequent fission of the fusioned object.

7.2.4 Weak fusions

Weak or non-constructive fusions implement the class *Suspendable*. The class *Destroyable* is necessary as well, because the operation *wfusion* destroys fused objects.

```
class (Destroyable d o t, Suspendable d o t) => WFusions d o t where
   wfusion :: [ID] -> ID -> d o t -> d o t
   wfusion is i = (flip . foldr) destroy is . resume i
   wfissionN :: Int -> ID -> d o t -> d o t
   wfissionN n i = uncurry (createN n) . pair (f, suspend i)
      where f = pair (wrap . const i, getObjType . selectObj i)
```

The only difference between constructive and weak fusion is formalized as the difference of the basic operation applied on the single object side of the operation:

constructive operations *create (destroy)* the object, while weak operations *resume (suspend)* objects.

7.3 Comparison of lifestyles with other categorizations of identity change

Theory of lifestyles is powerful enough to completely cover already existing proposals for categorization of change in identities, notably those of Al-Taha and Barrera (1994), and Hornsby and Egenhofer (1997). The mapping between lifestyles and the operations proposed by Al-Taha and Barrera (shown in Figure 2.2) is straightforward:

```
create = create
destroy = destroy
kill = suspend
reincarnate = resume
evolve = evolve
identify = foldr destroy
spawn i = uncurry create . pair (f, id)
   where f = pair (wrap . const i, getObjType . selectObj i)

aggregate = aggregate
disaggregate = segregate
fuse = fusion
fission = fission
```

Phenomena that can be modeled in the lifestyles framework and cannot be modeled in the proposal by Al-Taha and Barrera include weak fissions and aggregates, and the operation restructure.

Operations proposed by Hornsby and Egenhofer, informally discussed in Section 4.3.3 are formalized as compositions of the high-level lifestyles operations only:

```
metamorphose :: Evolvable d o t => ID -> d o t -> d o t
metamorphose = evolve

spawn :: Creatable d o t => ID -> d o t -> d o t
spawn i = uncurry create . pair (f, id)
   where f = pair (wrap . const i, getObjType . selectObj i)

mergeH :: Fusions d o t => [ID] -> t -> d o t -> d o t
mergeH = fusion

generate ::  Creatable d o t => [ID] -> t -> d o t -> d o t
generate = curry create

mix :: Destroyable d o t => [ID] -> t -> d o t -> d o t
mix (i:is) t = destroy i . curry create is t
```

The functions that operate on composite objects are *compound*, *unite*, *amalgamate*, *combine*, *secede*, and *dissolve*. The function *segregate'* returns a pair consisting of a list of aggregated objects and the database. It is used in definition of the function *combine.*

The operation amalgamate (see Figure 2.5-d) is especially interesting. It is a composition of fusions followed by an aggregation. Objects that amalgamate are aggregates with an arbitrary number of parts. It must be defined which parts of these objects are fusible. What happens if one composite has more parts than other composites that amalgamate? Is it possible to fuse several parts of one composite with a single part of other composite? It seems that amalgamate allows many different situations, and it is not clear if the proper behavior for each case can be standardized. Without loss of generality, we formalize only the case where the following conditions are fulfilled:

1. all composite objects have the same number of parts,
2. only one part of particular composite may participate in each fusion, and
3. parts are fused with respect to some ordering within the original composed objects - this is equal to the framework setting (Hornsby and Egenhofer 1998).

With these assumptions, the operation amalgamate is formally defined as a composition of multiple fusions followed by an aggregation of fused objects.

```
segregate'   :: Aggregates d o t => ID -> d o t -> ([ID], d o t)
segregate' i = pair (outl, (uncurry.flip.foldr) resume)
    . pair (getRels PartOf i, destroy i)

compound :: WAggregates d o t => ID -> ID -> d o t -> d o t
compound i j = uncurry (flip waggregate j)
      . cross (cons . pair (const i, id), id)
      . pair (getRels PartOf j, wsegregate j)

unite :: Aggregates d o t => [ID] -> t -> d o t -> d o t
unite = aggregate

combine :: Aggregates d o t =>  [ID] -> t -> d o t -> d o t
combine is t db = aggregate js t db where
   js = concat . map (outl . (flip segregate' db)) $ is

amalgamate :: (Fusions d o t, Aggregates d o t)
           => [ID] -> t -> t -> d o t -> d o t
amalgamate is t1 t2 db = uncurry (aggregate ns) (t1, db') where
   db' = outr (foldr fusion' (t2, db) jss)
   jss = transpose . map (outl . (flip segregate' db)) $ is
   fusion' is1 (t1,db1) = (t1, fusion is1 t1 db1)
   ns = [a+1 .. b]
   a = getID db
   b = a + length is

secede :: WAggregates d o t => ID -> ID -> d o t -> d o t
secede i j = uncurry (flip waggregate j)
       . cross (filter (i/=), id)
       . pair (getRels PartOf j, wsegregate j)
```

```
dissolve :: Aggregates d o t => ID -> d o t -> d o t
dissolve = segregate
```

The functions *compound* and *secede* shows an advantage of lifestyles: if there exist some minimal condition for an aggregation (e.g., a specific number of parts), *secession* will automatically signalize if the condition is not fulfilled any more.

Generality expressed with compositions is comparable with the iconic language used by Hornsby and Egenhofer. The operation missing in their proposal is the operation *suspend* and related concepts of weak operations on composite objects.

7.4 Summary

The operations affecting object identity are formalized on top of the formal model of an entity-relationship temporal database developed in Chapter 6. First, simple operations are modeled as primitive lifestyles, and their compositions are divided into four complex lifestyles: weak and constructive aggregations and fusions. The difference between the two variations of each (constructive and weak) is that the underlying operations on the single-side are *create* and *destroy* in the first case, and *suspend* and *resume* in the second case. Dependencies among classes for all lifestyles are shown in Figure 7.1.

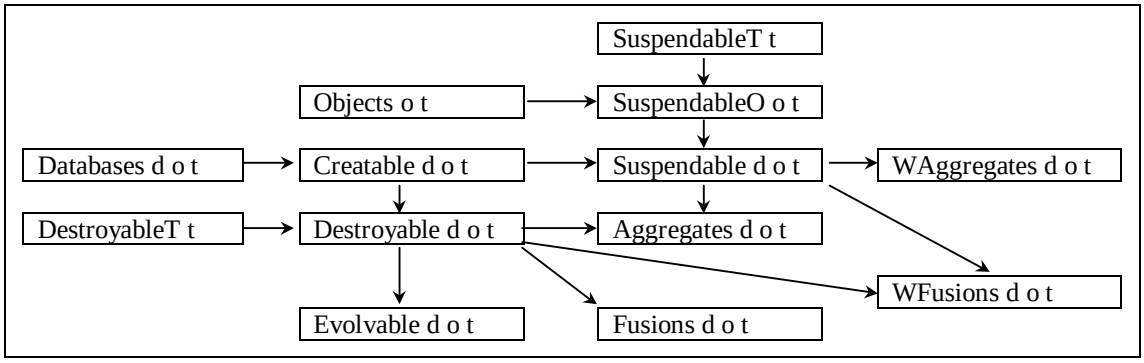

Figure 7.1: Classes hierarchy for lifestyles.

A comparison of our proposal with other categorizations of change showed that the theory of lifestyles offers greater flexibility and generality with less operations.

8. LIFESTYLES OF PHYSICAL OBJECTS

In previous chapters, we defined and formalized the general framework for the change affecting identity of objects. Objects were described as completely abstract, having only identifiers and operations on them in common. Whereas such abstract treatment is suitable for laying down the theoretical foundation for change operations, examples that are more concrete are necessary to show the practical importance of our approach.

We focus on the applicability of our theoretical considerations to the specific groups of real-world objects. Objects are divided into physical and abstract ones. Physical objects are concrete, graspable things that make up the physical reality of the world. Abstract objects are concepts that exist only as a matter of a social consensus (for example: marriages, partnerships, or unions of states). We analyze abstract objects in the subsequent chapter.

In this chapter, an extended account on physical objects is given. A categorization of physical objects, based on contemporary cognitive linguistics, is presented. Physical objects may be solid or liquid, movable or immovable, natural or human-made, living or non-living. As a special class, eternal objects are introduced for the representation of objects having the lifespan longer than the context within which such objects are considered.

Each subsection consists of an informal overview with the common sense background and a formal model. The formal model for each application is built on top of the general model of lifestyles, formalized in Chapter 7.

8.1 Solid objects

Solid objects are non-living objects with crisp boundaries that are physically observable by humans. They can be moved and their boundaries then become evident. These objects endure; they can be destroyed and created, but only by recognizable actions and events (through cutting, crushing, burning); they have a beginning and an end (Hayes 1985a).

Solid objects are divided into movable and immovable objects. Movable objects are typically manipulable by humans; they fit in the small-scale or tabletop space (Mark and

Frank 1996). Movable objects can be natural objects (objects as they existed without humans) and artifacts (objects produced by human activities).

Immovable objects are typically much larger than movables; they are not manipulable and form geographical space (Egenhofer and Mark 1995); they are places in which humans are placed, can move through or leave. Since there is no significant difference between natural and human-made immovable objects for the purpose of this thesis, these are treated together.

	movable	immovable
natural	stones, fruits	mountains, valleys
artifact	cars, computers	buildings, roads

Table 8.1: Categorization of solid objects.

The categorization of solid objects is proposed (see Table 8.1), and all categories are analyzed in the following subsections.

8.1.1 Movable natural objects

Natural movable solids are small-scale objects as found in the natural environment. Such objects are usually similar to other objects of the same kind, but can be easily individuated. They come to being by natural processes that separate smaller pieces from large masses: erosion, earthquakes, or volcanology. Typical examples are stones and fruits. Note that fruits are a border case to living objects. Once picked, however, fruits can be assumed “dead” for purposes of this section.

Natural objects are *creatable* and *destroyable*. The lifestyle of natural objects is characterized by the following formalization:

```
class Destroyable d o t => MovableNaturals d o t where
   createMovNat :: String -> Float -> ([ID], t) -> d o t -> d o t
   createMovNat name w a = uncurry (updateObj (addAtts as))
          . createWithID a
      where as = [(Name, Vs name), (Preds, Vp []), (Weight, Vf w)]
```

We assigned three default attributes for movable natural objects: a name, a list of predecessors, and weight. A simple representation of fruits and stones, followed by the implementation of the database model developed in Chapters 6 and 7 is:

```
data MovNat = Fruit | Stone
instance Relatable MovNat where
   relatable (On, (Fruit, Stone)) = True
instance Creatable TDB Object MovNat
instance DestroyableT MovNat where
   destroyable Fruit = True
   destroyable Stone = True
instance MovableNaturals TDB Object MovNat
```

Example queries on a small database are shown. Objects are created with two additional arguments: a *String* for names and a *Float* for weights of movable natural objects.

```
mn0, mn1, mn2 :: TDB Object MovNat
mn0 = T [Snap 0 [] [] ]
mn1 = createMovNat "appleA" 0.4 ([],Fruit) mn0
mn2 = createMovNat "stoneA" 1.2 ([],Stone) mn1
mn3 = createMovNat "stoneB" 2.3 ([],Stone) mn2

tstmn1 = existObj 3 (destroy 3 mn3)
-- False
tstmn2 = get Weight 2 mn3
-- Vf 1.2
```

The first test shows that the object named *stoneB* does not exist after the operation *destroy*. The second test extracts the value for the value set *Weight* of the object named *stoneA*.

8.1.2 Movable artifacts

Artifacts are solid objects produced by human activity. In this section, we consider movable artifacts only, while the immovable artifacts are described together with immovable natural objects in the next subsection.

A movable artifact can be a piece of homogenous solid stuff (e.g. a glass) or an assembly, which is made up of a finite number of other artifacts (e.g. a window). All manufactured goods we encounter and use in our everyday life conform to these criteria. Prototypical examples for simple artifacts are tires, wooden bricks, bolts, and screws. Prototypical examples for complex artifacts are cars, chairs, computers, and watches.

The property of being a part or having parts determines the lifestyle of movable artifacts. The individuation of assembled artifacts is simple as long as the original parts are holding together. The problem arises when the parts are changed or broken, for example. What makes a complex object the same through time, is a question from the story about the ship of Theseus, mentioned in the introduction of this thesis. Assuming spatiotemporal continuity, an assembly retains the same identity even if all of its parts

are replaced (Hayes 1985a). This view is plausible in the purely physical world we are dealing with in this chapter.

The second phenomenon related to complex artifacts is that of functionality. Such objects are constructed to fulfill certain human needs. If an artifact is not working properly, it must be repaired; often only a not-functioning part is replaced. We usually talk about “dead computers” or “dead cars”. When a computer or a car is repaired, it lives again - it is reincarnated. Therefore, it makes sense to allow such objects to be suspended (when broken) and resumed (when successfully repaired).

The formalization of movable artifacts resembles the aggregate lifestyle, allowing the objects to be suspended and resumed. Our example will be a car consisting of several changeable parts: a chassis, an engine, and four wheels. If a wheel is to be changed, the car is temporarily taken apart - suspended, and each of its parts is resumed. An aggregation of other original parts with the new wheel resumes the original identity of the car.

We formalize the behavior of movable artifacts with operations grouped in the class *MovableArtifacts*, which inherits operations from the classes *Aggregates* and *WAggregates*. The default attributes for movable artifacts are names and truth-values for the state (alive or suspended). The operations *destroy*, *segregate*, *waggregate*, *wsegregate* are inherited from the classes mentioned in the context. Three new operations are defined for aggregates: *addPart* brings additional part to an already existing aggregate; *removePart* takes a specific part away; *replacePart* exchanges an existing part of an aggregate with a part from the outside world. It is essential that all three mentioned operations be modeled as a composition of a weak segregation followed by a weak aggregation. If any criteria for the existence of the aggregate were not met after the parts are changed, the change would be rejected by the system.

```
class (Aggregates d o t, WAggregates d o t)
 => MovableArtifacts d o t where
   createMovArt :: String -> ([ID], t) -> d o t -> d o t
   createMovArt name t = uncurry (updateObj (addAtts as)).createWithID t
      where as = [(Name, Vs name), (Alive, Vb True)]

   aggregateMovArt :: String -> [ID] -> t -> d o t -> d o t
   aggregateMovArt name is t = uncurry (updateObj (addAtts as))
                             . pair (getID, id) . aggregate is t
      where as = [(Name, Vs name), (Alive, Vb True)]
```

```
addPart :: ID -> ID -> d o t -> d o t
addPart i j = uncurry (flip waggregate j)
   . cross (cons . pair (const i, id), id)
   . pair (getRels PartOf j, wsegregate j)

removePart ::  ID -> ID -> d o t -> d o t
removePart i j = uncurry (flip waggregate j)
    . cross (filter (i/=), id)
    . pair (getRels PartOf j, wsegregate j)

replacePart :: ID -> ID -> ID -> d o t -> d o t
replacePart i j k = uncurry (flip waggregate k)
   . cross (cons . pair (const i, filter (j/=)), id)
   . pair (getRels PartOf k, wsegregate k)
```

A simple representation of car parts with the necessary instances of the classes *Relatable*, *DestroyableT*, *Suspendable*, and *MovableArtifacts*:

```
data MovArt = Car | Chassis | Engine | Wheel

instance Relatable MovArt where
   relatable (PartOf, (Chassis, Car)) = True
   relatable (PartOf, (Engine, Car)) = True
   relatable (PartOf, (Wheel, Car)) = True

instance DestroyableT MovArt where
   destroyable Car = True
   destroyable Chassis = True
   destroyable Engine = True
   destroyable Wheel = True

instance SuspendableT MovArt where
   suspendable Car = True
   suspendable Chassis = True
   suspendable Engine = True
   suspendable Wheel = True

instance MovableArtifacts TDB Object MovArt
```

We populate an example database with several objects: a chassis, an engine, and four wheels shall build a car. Then, the test demonstrate the exchange of a wheel that is a part of the car with an "external" wheel (5).

```
ma0 = T [Snap 0 [] []]
ma1 = createMovArt "wheel-1 " ([], Engine) ma0
ma2 = createMovArt "wheel-2 " ([], Chassis) ma1
ma3 = createMovArt "wheel-3 " ([], Wheel) ma2
ma4 = createMovArt "wheel-4 " ([], Wheel) ma3
ma5 = createMovArt "wheel-5 " ([], Wheel) ma4
ma6 = createMovArt "chassisA" ([], Wheel) ma5
ma7 = createMovArt "engineA " ([], Wheel) ma6
ma8 = aggregateMovArt "carA    " [1,2,3,4,6,7] Car ma7

-- exchange wheel5 (5) and wheel2 (2) in the car (8)
tstma1 = replacePart 5 2 8 ma8

-- the new state of the database is then:
```

```
Snapshot
 Latest ID =8
  Objects: [
   #8 Car    [ "carA    ", resumed  , []],
   #7 Wheel  [ "engineA ", suspended, []],
   #6 Wheel  [ "chassisA", suspended, []],
   #5 Wheel  [ "wheel-5 ", suspended, []],
   #4 Wheel  [ "wheel-4 ", suspended, []],
   #3 Wheel  [ "wheel-3 ", suspended, []],
   #2 Chassis[ "wheel-2 ", resumed  , []],
   #1 Engine [ "wheel-1 ", suspended, []]]
  Relations: [
   5 is part of 8,
   1 is part of 8,
   3 is part of 8,
   4 is part of 8,
   6 is part of 8,
   7 is part of 8]
```

The wheel-2 is resumed, the relation *PartOf* between the wheel-2 and the *carA* is removed, the wheel-5 is suspended and it is the new part of the *carA*.

8.1.3 Immovable geographic objects

Immovable physical objects are human-made or natural objects that are not (easily) manipulable by humans. Such objects fill so-called large-scale geographical space: “space whose structure cannot be observed from a single viewpoint” as defined by Kuipers (Kuipers 1978, p.129), acknowledging all ambiguities coming from such a straightforward definition as explained by Mark and Frank (Mark and Frank 1996). Namely, “a single viewpoint” might be a plane or a satellite, when the viewer would be able to observe large-scale objects from a single viewpoint. If an average observer is a pedestrian, however, the definition given by Kuipers is valid.

Immovable objects made by humans are buildings: skyscrapers, roads, squares, bridges, dams. Natural immovables are earth topography phenomena: hills, mountains, valleys, islands, peninsulas. Human-made immovables have crisp boundaries, while natural immovables have fuzzy boundaries. This distinction is, however, imposed by human reasoning, and not by natural laws.

The operations that can be applied to these objects are creation, destroying and evolution.

```
class Destroyable d o t => Immovables d o t where
   createImmov :: String -> ([ID], t) -> d o t -> d o t
   createImmov name a = uncurry (updateObj (addAtts as)) . createWithID a
      where as = [(Name, Vs name), (Preds, Vp [])]
```

The necessary instances are:

```
data Immovable = Mountain | Building
instance Relatable Immovable
instance DestroyableT Immovable where
   destroyable Mountain = True
   destroyable Building = True
instance Immovables TDB Object Immovable
```

A brief example: a house evolves to a museum.

```
im0, im1, im2 :: TDB Object Immovable
im0 = T [Snap 0 [] [] ]
im1 = createImmov "Alps  " ([], Mountain) im0
im2 = createImmov "HouseA" ([], Building) im1
im3 = (uncurry (set Name (Vs "MuseumA")).pair (getID, id).(evolve 2)) im2
```

The result of *im2* (the state before evolution) is:

```
Latest ID =2
  Objects: [
   #2 Building[ "HouseA", []],
   #1 Mountain[ "Alps  ", []]]
```

Evolution (*im3*) is followed with setting the new name ("MuseumA") for a former house. The code fragment "*evolve* 2" produces the new object, which is referred by *getID* of the latest snapshot. Thus, the name is set to the newly created object (with identifier equal to 3) and not to the original object (*ID*=2), which does not exist after evolution. The resulting snapshot is:

```
Latest ID =3
  Objects: [
   #3 Building[ "MuseumA", [2]],
   #1 Mountain[ "Alps  ", []]]
```

The object 2 is destroyed, but its identifier is added to the list of predecessors of the newly created object.

8.2 Liquids

Liquids consist of many small loosely connected particles. Liquids differ from solid objects insofar they have no definite shape. They easily merge, split, move, and change shape because of gravity. Liquids are hard to grasp, but necessary for many fundamental physical and physiological processes.

Hayes gave the first formal account on liquids (Hayes 1985a). He proposed 15 different physical states of liquids ranging from wet surface to spray. In this thesis, we analyze only contained, bulk, lazy liquids in space, e.g. water in a glass, a river or a lake. Even such simplified view of liquids bears two different representations: liquids contained in solid objects - containers, and independent liquid objects. We start the discussion with the latter case: pure liquid objects, and then return to containers.

8.2.1 Liquid objects

The lifestyle of liquid objects is simple: it is a prototypical example of constructive fusion. The liquid objects fuse with other liquid objects into a new object that has the identifiers of fused objects as predecessors. Fused objects are destroyed, and cannot be resumed. If a liquid object is fissioned, new objects emerge and the original object is destroyed. We introduce the class *Liquids* that has a single operation - the function *createLiquid*, which adds three default attributes to liquid objects: a name, a list of predecessors, and an amount.

```
class (Fusions d o t) => Liquids d o t where
   createLiquid :: String -> Float -> ([ID], t) -> d o t -> d o t
   createLiquid name x a = uncurry (updateObj (addAtts as))
                           . createWithID a
      where as = [(Name, Vs name), (Preds, Vp (outl a)), (Amount, Vf x)]
```

A representation of water objects with instances necessary to inherit operations from the classes in context:

```
data Liquid = Water
instance Relatable Liquid
instance DestroyableT Liquid where
   destroyable Water = True
instance Liquids TDB Object Liquid
instance Fusions TDB Object Liquid
```

Finally, a simple example of two liquid objects that fuse into the third one. At the end the new object is fissioned into 3 new objects and the result is shown.

```
w0, w1, w2, w3 :: TDB Object Liquid
w0 = T [Snap 0 [] [] ]
w1 = createLiquid "waterA" 2.4 ([], Water) w0
w2 = createLiquid "waterB" 2.8 ([], Water) w1
w3 = fusion [1,2] Water w2
w4 = fissionN 3 3 w3
```

```
Latest ID =6
  Objects: [
   #6 water[ [3]],
   #5 water[ [3]],
   #4 water[ [3]]]
  Relations: []
```

In the next section, we analyze behavior of liquids in containers, a more interesting situation that is closer to the everyday perception of liquids by humans.

8.2.2 Liquids in containers

A container is a solid object or a part of solid object, which bounds a contained space – a connected volume of three-dimensional-space which has a contiguous rigid boundary below it and around it (Hayes 1985a). The surface of a container is impermeable and normally contains no leaks. The concept of quantity or amount is essential for reasoning about contained liquids. A container is limited by its capacity – the maximum amount of liquid it can contain.

This view of liquid objects allows their easy individuation through the individuation of containers, which are solid objects. The change of amount of liquid in a container does not change the identity of the contained liquid. Using this ontology, we can individuate and reason about dynamical liquid objects like rivers or baths. The lifestyle of containers is stable, similar to solid objects already discussed. They are created, destroyed, suspendable, and evolvable (a stream can grow to a river).

The remaining combination is the existence of liquid within a solid artifact, e.g., an amount of tea in the cup. We claim this case is an aggregate as well, but with some special properties which require careful analysis.

The aggregate is the filled cup consisting of the cup and the tea inside it. The liquid object inside the cup behaves as a fusible object, but it can be changed only after it has resumed after the segregation of the container.

If we are about to add some tea into the cup, the filled cup is suspended and the cup and the amount of tea are resumed (weak segregation); the amount of tea fuses with the added amount of tea into a new liquid object (fusion); the cup and the new amount of tea are aggregated as the old filled cup (weak aggregation).

The question is: what is, after all, an empty cup - is it a cup or a filled cup with no content? The answer is: if it is an “empty” cup, it is an aggregation of the cup and a zero amount of liquid.

The formalization has two levels: the object level and the database level. At the object level, the class *Containers* is necessary to capture operations for changing the amount of a single container (*pourIn* and *takeOut*). It has two observers (with the prefix "get") and two constructors (with the prefix "set") for retrieving and setting the mandatory attributes *Amount* and *Capacity* on the single objects.

```
class Objects o t => ContainersO o t where
   getAmount   :: o t -> Float
   getCapacity :: o t -> Float
   setAmount   :: (Float,o t) -> o t
   setCapacity :: (Float,o t) -> o t

   getAmount = unwrapValue . getValue . selectAtt Amount . getAttribs
   getCapacity = unwrapValue . getValue . selectAtt Capacity . getAttribs
   setAmount = uncurry (updateAtt Amount) . cross (wrapValue, id)
   setCapacity = uncurry (updateAtt Capacity) . cross (wrapValue, id)

   isEmpty :: o t -> Bool
   isEmpty = (==0.0) . getAmount

   pourIn :: (Float, o t) -> o t
   pourIn = cond p (f,g) where
      p = leq . pair (plus.cross(id,getAmount), getCapacity.outr)
      f = setAmount . pair (plus.cross (id,getAmount),outr)
      g = error "would overflow"

   takeOut :: (Float, o t) -> o t
   takeOut = cond p (f,g) where
      p = leq . cross (id, getAmount)
      f = setAmount . pair (minus.swap.cross (id,getAmount),outr)
      g = error "not enough in the container"
```

The predicate *isEmpty* and the functions *pourIn* and *takeOut* are defined in terms of basic operations and thus independent of the implementation. If the incoming amount is bigger than the free space in the container, an overflow error occurs. If the amount to be taken out from the container is bigger than the available amount, an underflow error occurs.

At the database level, the operation *createCont* produces a container with default attributes (name, predecessors, amount, and capacity). A new container is created only if the proposed capacity is greater than the proposed amount.

```
class (ContainersO o t, Aggregates d o t) => Containers d o t where
   createCont :: String -> Float -> Float -> ([ID], t) -> d o t -> d o t
   createCont name a c s = cond p (f, g) where
      p = const (a <= c)
      f = uncurry (updateObj h) . createWithID s
      g = error "amount cannot be greater than capacity"
      h = addAtts [(Name, Vs name), (Alive, Vb True),
                   (Amount, Vf a), (Capacity, Vf c)]
```

```
pourFromInto :: Float -> ID -> ID -> d o t -> d o t
pourFromInto a i j = updateObj (curry pourIn a) j
                   . updateObj (curry takeOut a) i
```

The operation *pourFromInto* is a composition of the operations *pourIn* and *takeOut*, that ensure that the total amount of liquid in the universe of discourse is preserved (conservation law).

The instantiation of the necessary classes for the representation types follows.

```
data Container = Cup | Tea | FilledCup
instance Relatable Container where
   relatable (In, (Tea, FilledCup)) = True
   relatable (PartOf, (Tea, FilledCup)) = True
   relatable (PartOf, (Cup, FilledCup)) = True

instance DestroyableT Container where
   destroyable Cup = True
   destroyable Tea = True
   destroyable FilledCup = True

instance SuspendableT Container where
   suspendable Cup = True
   suspendable Tea = True
   suspendable FilledCup = True
instance ContainersO Object Container
instance Containers TDB Object Container
```

A simple example with two cups with certain amounts of tea and pouring an amount to another cup is provided.

```
cs0, cs1, cs2, cs3, cs4 :: TDB Object Container
cs0 = T [Snap 0 [] []]
cs1 = createCont "firstCup " 4.0 10.0 ([],Cup) cs0
cs2 = createCont "secondCup" 4.0 10.0 ([],Cup) cs1
cs3 = createCont "teaA     " 5.0 5.0 ([],Tea) cs2
cs4 = aggregate [1,3] FilledCup cs3

tcs1, tcs2 :: Value
tcs1 = get Amount 1 cs4
-- Vf 4.0
tcs2 = get Amount 1 (pourFromInto 3.0 1 2 cs4)
-- Vf 1.0
```

The first cup has the amount 4.0 in cs4. After we pour the amount 3.0 to the second cup, the rest amount is 1.0.

8.3 Living beings

Living beings are able to reproduce their kind. They breath, eat, grow, and, finally, die. These are fundamental characteristics of biological life. Between the birth and death,

living beings retain their identity, although they are changeable in many ways: size, color, and appearance.

We begin with simple living beings: persons, animals, plants, and discuss a special case of tree with fruits in the subsequent section.

8.3.1 Persons, animals, and plants

Two fundamental properties of living beings are essential for the modeling of their lifestyle: birth and death. Death or the end of biological life is universal for all living beings, and it is naturally modeled with the operation *destroy*. Birth or the beginning of life is a more challenging task. It was modeled as an additional construct: reproduction (Hornsby and Egenhofer 1997), in accordance with common-sense representation of parental relations in the human society.

```
class Destroyable d o t => Livings d o t where
   createLiving :: String -> ([ID], t) -> d o t -> d o t
   createLiving name a = uncurry (updateObj (addAtts as)).createWithID a
      where as = [(Name,  Vs name), (Preds, Vp (outl a))]
```

The former case seems too complicate for the common sense based applications. Therefore, the simpler solutions are already incorporated into the semantics of the operation *createLiving*: a creation takes the identifiers of parents as the predecessors, preserving a temporal link among children and their parents. Thus, an additional construct for reproduction is superfluous. A simple implementation with examples follows.

```
data Living = Person | Animal | Plant
instance DestroyableT Living where
   destroyable Person = True
   destroyable Animal = True
   destroyable Plant  = True

instance Livings TDB Object Living
instance Relatable Living
liv0, liv1, liv2, liv3 :: TDB Object Living
liv0 = T [Snap 0 [] [] ]
liv1 = createLiving "John" ([], Person) liv0
liv2 = createLiving "Mary" ([], Person) liv1
liv3 = createLiving "Sue " ([1,2], Person) liv2
```

Thus, the person "Sue" has the identifiers of John and Mary as her predecessors - parents.

8.3.2 Trees with fruits

A tree with its seasonal fruits is another example of living objects. A tree with fruits is an aggregate. In an implementation, trees and fruits are modeled as suspendable objects (to enable aggregation). A tree is a persistent carrier, existence of which is independent of fruits. Fruits are seasonally created by the tree (neglecting some assistance from the nature). Fruits grow to a certain time when they start to rot if not collected and consumed by animals or people.

A tree with fruits is a composed object that consists of a tree (container) and fruits (containment). Although we can count items of fruit, it is possible to speak about fruits in terms of amount or mass. Thus, we can collect a certain amount of fruits from a tree, leaving the rest to be collected later. Eventually, the rest rots after a certain period. When the last fruit vanishes from a tree with fruits, it is a tree what is left - the composed object is destroyed. During the next season a new composed object will emerge: the old tree with new fruits.

Formally, a tree with fruits is a constructive aggregate between exactly one tree and exactly one amount of fruits. This is expressed in the condition for the operation *aggregateTree*.

```
class (ContainersO o t, Aggregates d o t) => TreeWithFruits d o t where
   createTree :: String -> Float -> ([ID], t) -> d o t -> d o t
   createTree name a s = uncurry (updateObj h) . createWithID s where
      h = addAtts [(Name, Vs name), (Alive, Vb True), (Amount, Vf a)]

   aggregateTree :: [ID] -> t -> d o t -> d o t
   aggregateTree is t = cond p (f, g) where
      p = eql . pair (const (length is), const 2)
      f = aggregate is t
      g = error "only a single fruits object allowed"
```

The fact that fruits cannot be poured back to the tree is expressed in the instantiation of the class *ContainersO*:

```
instance ContainersO Object Tree  where
   pourIn = error " not possible "
   takeOut = cond p (f,g) where
      p = leq . cross (id, getAmount)
      f = setAmount . pair (minus.swap.cross (id,getAmount),outr)
      g = error "not enough fruits on the tree"

data Tree = ATree | Fruits | TreeWithFruits
instance Relatable Tree where
   relatable (PartOf, (ATree, TreeWithFruits)) = True
   relatable (PartOf, (Fruits, TreeWithFruits)) = True
```

```
instance DestroyableT Tree where
   destroyable ATree = True
   destroyable Fruits = True
   destroyable TreeWithFruits = True

instance SuspendableT Tree where
   suspendable ATree = True
   suspendable Fruits = True
   suspendable TreeWithFruits = False

instance TreeWithFruits TDB Object Tree
```

The datatype of the aggregate (*TreeWithFruits*) is not suspendable - only constructive aggregation is possible. Fruits come seasonal and - once collected - cannot be aggregated with the tree.

```
tf0, tf1, tf2, tf3 :: TDB Object Tree
tf0 = T [Snap 0 [] []]
tf1 = createTree "TreeA  " 10.0 ([],ATree) tf0
tf2 = createTree "FruitsA" 5.0  ([],ATree) tf1
tf3 = aggregateTree [1,2] TreeWithFruits tf2

ttf1 :: Float
ttf1 = queryObj getAmount 1 tf3
-- 5.0
ttf2 = updateObj (curry pourIn 7.0) 2 tf3
-- not possible
```

8.4 Eternal objects

Eternal objects are never destroyed. In the model, it means that their life span is by orders of magnitude longer than the context they are considered in. An excellent example is the Sun from the human perspective.

The lifestyle of eternal objects is the simplest of all. They just exist, and are, in contrast to all other categories, not destroyable.

```
class (Creatable d o t) =>  Eternals d o t where
   createEternal :: String -> ([ID], t) -> d o t -> d o t
   createEternal name a = uncurry (updateObj (addAtts as)).createWithID a
      where as = [(Name, Vs name), (Preds, Vp (outl a))])

data Eternal = Star | Planet
instance Relatable Eternal
instance DestroyableT Eternal where
   destroyable Star = False
   destroyable Planet = False
instance Eternals TDB Object Eternal
```

Finally, we can test if a created object can be destroyed:

```
e0, e1, e2 :: TDB Object Eternal
e0 = T [Snap 0 [] [] ]
e1 = createEternal "Sun" ([],Star) e0
e2 = destroy 1 e1

-- error: the object #1 is not destroyable.
```

An attempt to destroy an eternal object results in an error message.

8.5 Summary

In this chapter, we gave a categorization of objects constructing the physical reality of our world. At the level of detail assumed here, the physical objects are divided into solids, liquids, living beings, and eternal objects. The operations on identity within each category are modeled with the apparatus developed in previous chapters. The theory of lifestyles is powerful enough to describe diversity of physical objects. Usually, a simple extension of sets of object types is all that is necessary. Sometimes, however, we must introduce new classes to capture the semantics of additional important operations. This is achieved easily by putting our general framework in the context of new instances. The hierarchy of classes is shown in Figure 8.1.

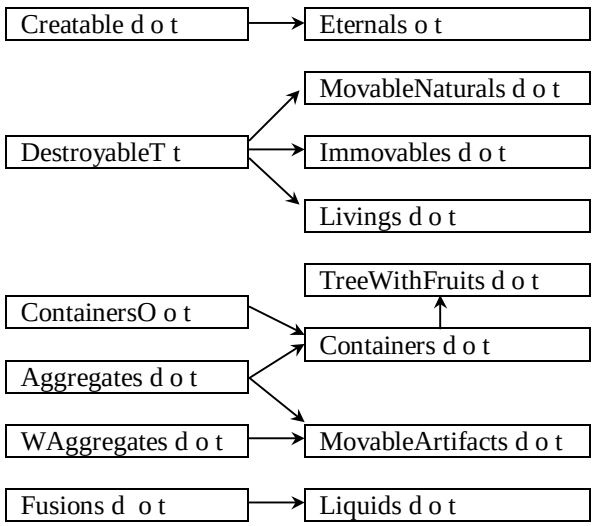

Figure 8.1: Classes hierarchy for lifestyles of physical objects.

The level of application presented in this chapter – physical reality - is the starting point for modeling a more complex environment of social reality – abstract objects.

9. LIFESTYLES OF ABSTRACT OBJECTS IN THE SOCIAL REALM

In this chapter, we continue to provide practical applications for the theoretical framework of operations affecting object identity. After the analysis of physical (tangible) objects in the previous chapter, we concentrate on abstract objects: non-graspable things that exist only in view of a particular social agreement. These objects construct the social reality of the world, the part completely dependent on human beings.

We give an informal description of typical social constructs: marriages and business partnerships. Such constructs emerge, develop and cease to exist through time. An analogy is drawn between social constructs and physical objects from the previous chapter. Humans have ability to use metaphorical transformation from one experience, to structure experience in another situation.

Several typical applications from the GIS domain are described: administrative units, ownership and usufruct rights on cadastre parcels. These important concepts are formalized using the tools developed in the previous chapters. Marriages, partnerships, and administrative units share behavior with movable artifacts, ownership rights with liquids, and usufruct rights with trees and fruits.

9.1 Constructs of social reality

Most human interaction is defined by the rules of social behavior. Some of these rules are not written (friendships, promises), but many of them exist in written form, called institutional facts (Searle 1995). The concepts that belong in this group are numerous: money, marriage, partnerships, ownership, governments, citizenship, etc.

Institutional facts need physical objects as status indicators. Pieces of paper or coins are physical objects, which under certain circumstances are considered as money. A passport is a valid indicator that its owner is empowered to travel to certain foreign countries and come back to his home country.

Institutional facts depend on collective intentionality; common will of the critical majority in a particular society to accept the rules. In contrast to physical objects, which wear out as we use them (cars, shirts), the institutional objects are renewed and

strengthened by their constant use (ownership, citizenship, money, marriage, government) (Searle 1995).

We have chosen two typical institutional facts for the analysis of lifestyles applicability: marriages and business partnerships. Both concepts are described by underlying metaphors that relate institutional facts with physical, graspable objects.

In cognitive science, metaphors are ways of understanding one domain of experience by using the terms from another, possibly simpler domain (Lakoff and Johnson 1980). A common prejudice is thinking about metaphors as strictly poetic and rhetorical figures of speech that express emotions, moods, and attitudes. In what follows, we consider and use of metaphors in the sense of cognitive science.

9.1.1 Marriage

Marriage begins with two people simply living together. The conversion of spatial proximity and cohabitation to an institutional fact is justified by the need for a system of collectively recognized rights, responsibilities, duties, and obligations (Searle 1995). The institutional fact of marriage is a result of the institutional fact of the speech act in a special context: mutual promises of spouses in front of a presiding official.

In this thesis, we consider the institutional fact of marriage as an abstract object on its own. Our goal is to show that such an object has its lifestyle similar to certain physical objects. The set of metaphors for marriage is proposed in (Johnson 1993). On the basis of individual interviews of married couples, Johnson found out that people perceive their marriage through one or more of the following metaphors: MARRIAGE IS A MANUFACTURED OBJECT, MARRIAGE IS AN ONGOING JOURNEY, MARRIAGE IS A DURABLE BOND BETWEEN TWO PEOPLE, MARRIAGE IS A RESOURCE/INVESTMENT, and MARRIAGE IS AN ORGANIC UNITY, (Johnson 1993). How to extract a unifying core from such wide palette of metaphors?

Johnson's metaphors are formed around the opinions of the people about their marriages. We will analyze the life cycle of a marriage and its effects on the spouses. The following scenario of a naive society is proposed: there are unmarried men and women, those who are married, and those who were married before. This is exactly how the real world is seen by the eyes of the appropriate legal office – let us call it "the marriage registry". Then, a marriage begins with the registration of a mutual agreement of both spouses in front of a registrar. Both spouses are removed from the list of

unmarried people and added to the list of married people. The end of marriage is registered either as a result of the death of a spouse or a legal procedure called divorce. If the spouses get divorced, they cannot be transferred back to the list of unmarried people – marriage is irreversible. Even in the case that they marry each other again, it would not be their first marriage.

The lifestyle described here gives rise to a new metaphor for marriage: MARRIAGE IS A CONSTRUCTIVE AGGREGATE. To support this claim we can draw the following parallels between two domains:

- the marriage begins as an aggregation of two people, and after the marriage they are not available for another marriage (spouses are shielded against other aggregation);
- the divorce ends the marriage, producing the two old objects – spouses;
- death of one of spouses ends the marriage, the other spouse is then free;
- the marriage is irreversible – the same spouses married again make the new marriage, (once destroyed, a marriage is not reincarnatable). It should be noted that this is the legal, but not a "naive" view.

Thus, the formalization of marriage resembles the formalization of movable artifacts from previous chapter. The only difference is stipulated by regulations: a new marriage is always a new marriage, even between the same persons. Marriages are constructive aggregates only - they cannot be suspended or resumed.

The formalization of marriages begins with the operation *createPerson* that incorporate the attribute *Age*. It is assumed that there is a minimal age condition for marriage being 18 years. Next, the operation *destroyPerson* has two cases: if a person was not married, only the person is destroyed; if the person was married, the marriage is destroyed as well.

```
class (Eq t, Aggregates d o t) => Marriages d o t where
   createPerson :: String -> Int ->  ([ID], t) -> d o t -> d o t
   createPerson name age s = uncurry (updateObj h) . createWithID s where
      h = addAtts [(Name, Vs name), (Alive, Vb True), (Age, Vi age)]

   destroyPerson :: ID -> d o t -> d o t
   destroyPerson i = cond (married i) (f . pair (head . h i, g), g) where
      married x = not . null . h x
      h x = getConvRels PartOf x
      f = uncurry destroy
      g = destroy i
```

```
createMarriage :: (ID,ID) -> t -> d o t -> d o t
createMarriage (i, j) t = cond (meet (p,q)) (f, g) where
   p = uncurry (/=) . pair (h i, h j)
   h x = queryObj getObjType x
   q = meet (age i, age j)
   age x = geq . pair ( y . getAttribs . selectObj x, const 18)
   y = unwrapValue . getValue . selectAtt Age
   f = aggregate [i,j] t
   g = error "not a legal marriage!"

divorceMarr :: ID -> d o t -> d o t
divorceMarr = segregate
```

Marriage is created with the operation *createMarriage*, which performs two checks: persons must be of different type (gender), and both must be older than 18 years. If one of these criteria is not fulfilled, the marriage is declared illegal and will not be registered. Finally, divorce is similar to the operation *segregate*. It ensures that a weak aggregation is not possible.

The representation and necessary instances:

```
data Marr = Marriage | Male | Female
instance Eq Marr where
   (==) Male Male = True
   (==) Female Female = True
   (==) Marriage  Marriage = True
   (==) _ _ = False

instance Relatable Marr where
   relatable (PartOf, (Male, Marriage)) = True
   relatable (PartOf, (Female, Marriage)) = True
instance SuspendableT Marr where
   suspendable Marriage = False
   suspendable Male = True
   suspendable Female = True
instance DestroyableT Marr where
   destroyable _ = True
instance Marriages TDB Object Marr
```

We are ready to demonstrate some examples of the presented theory:

```
mm0, mm1, mm2, mm3, mm4, mm5, mm6 :: TDB Object Marr
mm0 = T [Snap 0 [] []]
mm1 = createPerson "John" 20 ([], Male) mm0
mm2 = createPerson "Mary" 20 ([], Female) mm1
mm3 = createPerson "Sue " 17 ([], Female) mm2
mm4 = createMarriage (1,2) Marriage mm3 -- OK
mm5 = createMarriage (1,3) Marriage mm3 -- not legal
mm6 = destroyPerson 1 mm4
mm7 = divorceMarr 4 mm4
```

The operation *mm4* returns the following state (John and Marry are parts of the marriage, both are suspended):

```
Latest ID =4
  Objects: [
   #4 Marriage[ []],
   #3 Female  [ "Sue ", resumed  , 17, []],
   #2 Female  [ "Mary", suspended, 20, []],
   #1 Male    [ "John", suspended, 20, []]]
  Relations: [
   1 is part of 4,
   2 is part of 4]
```

The operation *mm5* (trying to construct a marriage between 17 years old Sue and John) results in an error message - marriage is not legal. The operation *mm6* (John dies) destroys the marriage and resumes Mary's identifier:

```
Latest ID =4
  Objects: [
   #3 Female  [ "Sue ", resumed  , 17, []],
   #2 Female  [ "Mary", suspended, 20, []]]
  Relations: []
```

We conclude that a powerful model for a marriage registry can be developed as an extension of the generic lifestyle of constructive aggregation.

9.1.2 Business partnerships

Our second example comes from the domain of economics. Legal organization of business activities and relationships is an important issue in modern society. To avoid disputes, promises and friendships are replaced by contracts and business partnerships. The goal reached by such institutionalization is the higher predictability of behavior of all parties involved.

The idea of partnership is based on sharing of common investments and added values through profits in equal parts: there are no special items exclusively owned by one partner. According to the Swiss civil law (*Schweizerisches Obligationenrecht - OR*) compiled in (Schönenberger 1976), a simple partnership "is a contractual agreement between two or more persons to attain a joint goal with joint forces and means", the translation by (Arpagaus 1997). The original text in German is:

> *"[Einfache] Gesellschaft ist die vertragsmäßige Verbindung von zwei oder mehreren Personen zur Erreichung eines gemeinsamen Zweckes mit gemeinsamen Kräften oder Mitteln." OR §530*

Each partner must share the profit with other partners. Each partner takes the equal part in a profit or a loss. A partnership is over if the goal it is grounded for does not exist any

more or is unreachable, or if one of partners dies, or if one of partners bankrupts, or on agreement of partners, or if the agreed period of time pasts, or if the court decide so.

The simple partnerships are very much like marriages, and we will not formalize the same thing twice. Instead, we shall look at the rules of grounding and dismissing stock corporations (*Aktiengesellschaften - AG*). A stock corporation "is a separate legal entity with a corporate name and legal capital that is fixed in advance and divided into shares of capital stock", (Arpagaus 1997). The original text in German is:

> *"Die Aktiengesellschaft ist eine Gesellschaft mit eigener Firma, deren zum voraus bestimmtes Kapital (Grundkapital) in Teilsummen (Aktien) zerlegt ist und für deren Verbindlichkeiten nur das Gesellschaftsvermögen haftet." OR §620*

A stock corporation is modeled as a movable artifact with two conditions for grounding and subsequent changes in the structure of shareholders: the number of shareholders and the amount of capital stock. Shareholders are modeled as containers of shares. The class *Partnerships* is characterized with the operation *sumStocks*, which establishes the condition for grounding the corporation in the operation *createCorporation*. The amount of ground stock remains always the same (although the par value of a single share may vary). In the following formalization, we assume that the fixed number of shares is 10000; each share has the value of 10 francs, hence reaching the minimal capital stock value of 10000 Swiss francs.

```
class (Containers d o t, MovableArtifacts d o t)
   => Partnerships d o t where
   createStockHolder :: String -> Float -> ([ID],t) -> d o t -> d o t
   createStockHolder name m s = uncurry (updateObj h) . createWithID s
      where
      h = addAtts [(Name, Vs name), (Alive, Vb True),
                   (Amount, Vf m), (Capacity, Vf 10000.0)]

   sumStocks :: [ID] -> ValueSet -> d o t -> Float
   sumStocks is a  = sum . map (unwrapValue . getValue . selectAtt a
      . getAttribs) . liftM selectObj is

   createCorporation :: String -> [ID] -> t -> d o t -> d o t
   createCorporation name ss t d =
        if s > 10000.0 then cond (meet (p,true)) (f,g) d
        else error " not enough capital" where
      p = geq . pair (const (length ss), const 3)
      s = sumStocks ss Amount d
      f = uncurry (updateObj (addAtts [(Alive, Vb True),
          (Amount, Vf s)])) . pair (getID, id) . aggregate ss t
      g = error "founding of the corporation not possible"
```

```
sellShares :: Float -> ID -> ID -> d o t -> d o t
sellShares = pourFromInto

sellAllShares :: ID -> ID -> ID -> d o t -> d o t
sellAllShares a b c d = removePart a c d'
   where d' = pourFromInto f a b d
         f = unwrapValue (get Amount a d)
```

The operation *sellShares* simulates the transfer of a certain number of shares from one shareholder to another one. If one shareholder decides to quit and sell all his shares, he is also removed from the aggregate - corporation. In both cases, it is essential that the number of shares remains the same. The representation and implementation of inherited classes on the datatype follows.

```
data Partnership = Corporation | StockHolder
instance Relatable Partnership where
   relatable (PartOf, (StockHolder, Corporation)) = True

instance SuspendableT Partnership where
   suspendable Corporation = True
   suspendable StockHolder = True

instance DestroyableT Partnership where
   destroyable _ = True
```

A prototype of a corporation and the possible changes are shown in the following example. First, a corporation (ID=5) is created with the total of 10000 shares. If the shareholder A (ID=1) buys 2500 shares from the shareholder B, the sum remains the same. If B sells all, he leaves the corporation, the sum of shares remains the same.

```
pa0, pa6 :: TDB Object Partnership
pa1, pa2, pa3, pa4, pa5 ::
   Snapshot Object Partnership -> Snapshot Object Partnership

pa0 = T [Snap 0 [] []]
pa1 = createStockHolder "holderA" 2000.0 ([],StockHolder)
pa2 = createStockHolder "holderB" 4000.0 ([],StockHolder)
pa3 = createStockHolder "holderC" 3000.0 ([],StockHolder)
pa4 = createStockHolder "holderD" 2000.0 ([],StockHolder)
pa5 = createCorporation "corporA" [1,2,3,4] Corporation
-- serialized transaction:
pa6 = liftU (pa5 . pa4 . pa3 . pa2 . pa1) pa0
pa7, pa8 :: TDB Object Partnership
-- shareholder A sells some shares (2500) to B:
pa7 = liftU (sellShares 2500.0 2 1) pa6
-- shareholder A sells all shares to B in the corporation 5:
pa8 = liftU (sellAllShares 2 1 5) pa6
```

The transaction *pa7* results in:

```
Latest ID =5
  Objects: [
   #5 Corporation[ "corporA", resumed  , 11000.0, []],
   #4 StockHolder[ "holderD", suspended, 2000.0, 11000.0, []],
   #3 StockHolder[ "holderC", suspended, 3000.0, 11000.0, []],
   #2 StockHolder[ "holderB", suspended, 1500.0, 11000.0, []],
   #1 StockHolder[ "holderA", suspended, 4500.0, 11000.0, []]]
  Relations: [
   1 is part of 5,
   2 is part of 5,
   3 is part of 5,
   4 is part of 5]
```

The transaction *pa8* results in:

```
Snapshot
 Latest ID =5
  Objects: [
   #5 Corporation[ "corporA", resumed  , 11000.0, []],
   #4 StockHolder[ "holderD", suspended, 2000.0, 11000.0, []],
   #3 StockHolder[ "holderC", suspended, 3000.0, 11000.0, []],
   #2 StockHolder[ "holderB", resumed  , 0.0, 11000.0, []],
   #1 StockHolder[ "holderA", suspended, 6000.0, 11000.0, []]]
  Relations: [
   1 is part of 5,
   3 is part of 5,
   4 is part of 5]
```

Stock corporations share the lifestyle with movable artifacts. The conditions for existence are different than in physical domain, because of legal regulations.

9.2 Lifestyles of land units

In this section, we focus on the application of lifestyles theory on land information systems. The land domain is selected because spatial administrative subdivisions of land area cover the whole ground of our planet, being the area of environmental concerns (in case of vital resources for the future), international disputes (in case of wars), and careful measuring and mapping (in case of national surveying).

9.2.1 Ownership rights on cadastre parcels

The right of ownership is the basic right in common law. In Austrian law, ownership in general is defined as follows (translation of the author):

“Viewed as a right, ownership is the competence to rule the substance and the use of a thing to one’s arbitrariness and to bar anybody else from substance and use.”

This definition is very broad and accounts for watches, cars, and pieces of land as well. The ownership of land is somewhat specific: land is non-perishable and it cannot be easily stolen, lost, destroyed, or counterfeited (Smith and Zaibert 1996).

The ownership rights behave like liquids: once melted, these are not splittable - new rights emerge. The reason is the legal nature of the ownership rights posed upon the parcel. A good example for indivisibility of rights is the mortgage.

```
class Liquids d o t => Parcels d o t where
   createParcel :: String -> Float -> ([ID], t) -> d o t -> d o t
   createParcel = createLiquid

data Parcel = Parcel
instance Relatable Parcel
instance DestroyableT Parcel where
   destroyable Parcel = True

instance Parcels TDB Object Parcel

instance Liquids TDB Object Parcel

p0, p1, p2, p3 :: TDB Object Parcel
p0 = T [Snap 0 [] [] ]
p1 = createParcel "parcelA" 2.4 ([], Parcel) p0
p2 = createParcel "parcelB" 2.8 ([], Parcel) p1
p3 = fusion [1,2] Parcel p2
p4 = fissionN 3 3 p3
p6 = restructure [1,2] Parcel 4 p2 --4,5,6,7
```

The metaphor behind the model is OWNERSHIP RIGHTS ON PARCELS ARE LIQUID OBJECTS.

9.2.2 Usufruct rights

There is a special right that can be imposed on a cadastre parcel: the owner of the parcel transfers the right of harvesting the parcel to another person, who then is said to have usufruct right on the parcel. Usufruct is the right to use another's property while not changing or harming it.

If the parcel is not harvested, the possible benefit is irreversibly gone, just like the fruits from the tree rot if not picked. Therefore, the right of usufruct can be formalized in the same way as the life of trees with fruits.

```
class TreeWithFruits d o t => Usufructs d o t where
   createAParcel :: String -> Float -> ([ID], t) -> d o t -> d o t
   createAParcel = createTree
   createUsufruct :: [ID] -> t -> d o t -> d o t
   createUsufruct = aggregateTree

data UsufructRight = AParcel | Usufruct | ParcelWithUsufruct

instance Relatable UsufructRight where
   relatable (PartOf, (AParcel, ParcelWithUsufruct)) = True
   relatable (PartOf, (Usufruct, ParcelWithUsufruct)) = True

instance DestroyableT UsufructRight where
   destroyable AParcel = True
   destroyable Usufruct = True
   destroyable ParcelWithUsufruct = True

instance SuspendableT UsufructRight where
   suspendable AParcel = True
   suspendable Usufruct = True
   suspendable ParcelWithUsufruct = False

instance ContainersO Object UsufructRight  where
   pourIn = error " not possible "
   takeOut = cond p (f,g) where
      p = leq . cross (id, getAmount)
      f = setAmount . pair (minus.swap.cross (id,getAmount),outr)
      g = error "not enough usufruct on the parcel"
instance TreeWithFruits TDB Object UsufructRight
instance Usufructs TDB Object UsufructRight

-- examples:

uf0, uf1, uf2, uf3, uf4 :: TDB Object UsufructRight
uf0 = T [Snap 0 [] []]
uf1 = createAParcel "parcelA    " 10.0 ([],AParcel) uf0
uf2 = createAParcel "usufructA  " 10.0 ([],Usufruct) uf1
uf3 = createAParcel "parcelB    " 5.0  ([],AParcel) uf2
uf4 = createUsufruct [1,2] ParcelWithUsufruct uf3

tuf1 :: Float
tuf1 = get Amount 3 uf4
-- Vf 5.0
tuf2 = updateObj (curry pourIn 7.0) 2 uf4
-- not possible
```

The lifestyle of usufruct rights can be fully matched by the model of trees with fruits.

9.2.3 Administrative units

Administrative units (states, counties, provinces) can be assembled and disassembled freely, depending only on the common will of the subjects involved. The formal model is completely similar to the model for movable artifacts, described in Section 8.1.2.

```
class MovableArtifacts d o t => Unions d o t where
   createUnit :: String -> ([ID], t) -> d o t -> d o t
   createUnit = createMovArt

   aggregateUnits :: String -> [ID] -> t -> d o t -> d o t
   aggregateUnits = aggregateMovArt

   addUnit :: ID -> ID -> d o t -> d o t
   addUnit = addPart

   secedeUnit :: ID  -> ID -> d o t -> d o t
   secedeUnit = removePart
```

The implementation for the appropriate object type is:

```
data AdminUnit = State | Union
instance Relatable AdminUnit where
   relatable (PartOf, (State, Union)) = True
instance DestroyableT AdminUnit where
   destroyable State = True
   destroyable Union = True
instance SuspendableT AdminUnit where
   suspendable State = True
   suspendable Union = True
instance MovableArtifacts TDB Object AdminUnit
instance Unions TDB Object AdminUnit
```

Our case study will be forming of Canada, followed by a hypothetical secession of Quebec (Hornsby and Egenhofer 1997). The union is formed by 10 states:

```
au0, au1, au2, au3 :: TDB Object AdminUnit
au0 = T [Snap 0 [] []]
au1  = createUnit "Quebec           " ([], State) au0
au2  = createUnit "Ontario          " ([], State) au1
au3  = createUnit "New Brunswick    " ([], State) au2
au4  = createUnit "Nova Scotia      " ([], State) au3
au5  = createUnit "British Columbia " ([], State) au4
au6  = createUnit "Prince Edward Isl" ([], State) au5
au7  = createUnit "Alberta          " ([], State) au6
au8  = createUnit "Manitoba         " ([], State) au7
au9  = createUnit "Newfoundland     " ([], State) au8
au10 = createUnit "Saskatchewan     " ([], State) au9
au11 = aggregateUnits "Canada       " [1,2,3,4,5,6,7,8,9,10] Union au10

-- all parts of canada as a list of IDs
tstau0 = getRels PartOf 11 au11

-- all parts as a list of objects:
tstau1 =  map (flip selectObj au11) (getRels PartOf 11 au11)

-- finally, secession
tstau2 = secedeUnit 1 11 au11

-- results in:
```

```
Latest ID =11
  Objects: [
   #11 Union[ "Canada           ", resumed  , []],
   #10 State[ "Saskatchewan     ", suspended, []],
   #9 State[ "Newfoundland     ", suspended, []],
   #8 State[ "Manitoba         ", suspended, []],
   #7 State[ "Alberta          ", suspended, []],
   #6 State[ "Prince Edward Isl", suspended, []],
   #5 State[ "Britisch Columbia", suspended, []],
   #4 State[ "Nova Scotia      ", suspended, []],
   #3 State[ "New Brunswick    ", suspended, []],
   #2 State[ "Ontario          ", suspended, []],
   #1 State[ "Quebec           ", resumed  , []]]
  Relations: [
   2 is part of 11,
   3 is part of 11,
   4 is part of 11,
   5 is part of 11,
   6 is part of 11,
   7 is part of 11,
   8 is part of 11,
   9 is part of 11,
   10 is part of 11]
```

The Quebec is segregated and resumed, while Canada survived. Other administrative divisions (e.g., the division of a state in provinces and counties) could be modeled in the same manner. The metaphor behind this model is: ADMINISTRATIVE UNITS ARE MOVABLE ARTIFACTS.

9.3 Summary

In this chapter, we focused on abstract objects – objects that evolved with human society and whose existence is dependent on human agreement. Such objects are called institutional facts, in contrast to *bona fide* facts that do not need to be enforced. We showed that institutional objects could be modeled in the same way as physical objects by the simple extension of the proposed theory of lifestyles.

Marriages and partnerships, seen from a registry perspective, exercise the lifestyle of aggregatable objects: marriages are constructive aggregates; partnerships are movable artifacts where the parts are containers shareholders.

Administrative subdivisions of land – regardless if they are unions, federations, states, provinces or counties – are similar to movable artifacts. Lower-level units can be removed, added, replaced or regrouped within higher-level units. Ownership rights on a cadastral parcel, on the other hand, resemble liquid objects: parcels fuse and fission

irreversibly creating the new parcels and destroying the existing parcels. Finally, the kind of partial right on cadastral parcel – usufruct right – is similar to a tree with fruits: if the parcel is not harvested, the usufruct right is gone for that year.

The hierarchy of classes for abstract objects and the dependencies upon the classes of physical objects and generic lifestyles are shown in Figure 9.1.

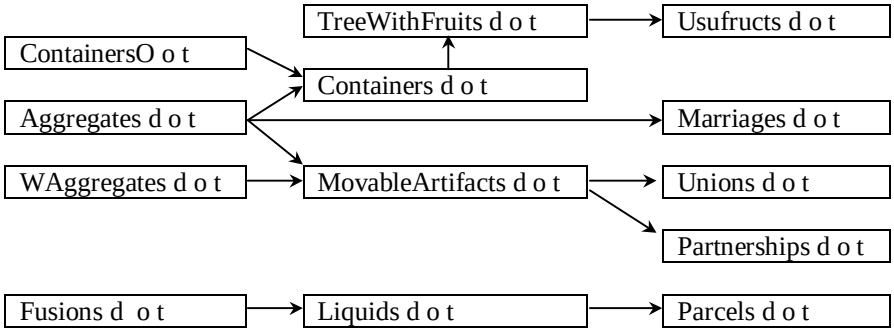

Figure 9.1: Classes hierarchy for non-tangible (abstract) objects from social realm.

The theory of lifestyles is applicable to a wide range of phenomena in the world of institutional facts – the part of real world human activities. Extensions to the class system proposed in the previous chapter are simple and straightforward. The most significant institutions of human society are modeled similar to simpler domains of physical objects.

10. CONCLUSIONS AND FUTURE WORK

The goal of this thesis was to explore the behavior of object identity in a spatiotemporal database, and to examine the applicability of the theory in various application domains both in physical and social realm.

We used the entity-relationship (E-R) model for the representation of the real world (Chen 1976). Identifiable features are represented as objects, which are distinguishable by unique identifiers. Properties of features are represented as attributes (functions from objects to values). Relationships among features are modeled as relations among objects.

All objects and relations at a single point of time build a snapshot. Objects and relations are changing over time. For each change, a new snapshot is appended to the database. The whole spatiotemporal database consists of a number of snapshots, and time is implicitly stored as the ordering of snapshots. We used linear, discrete, and totally ordered model of time.

Objects are metaphorically perceived as having life: an object has its birth or creation, its life or existence, its death or destruction. The central concept in the life of an object is its identifier, which is unchanged from the birth to the death of the object. Identifiers are system constructs and they are maintained by the database independently of the user. A long discussion about the model for a general temporal database was necessary to prepare the ground for change in identity, because we wanted to represent both the multi-purpose temporal database and the properties of identifiers in a single environment.

Four basic operations affecting object identity are proposed: *create*, *destroy*, *suspend*, and *resume*. Their compositions are either applicable on a single object (*evolve*), or on a group of objects (constructive and weak *fusion*, *fission*, *aggregate* and *segregate*, and *restructure*). Altogether, these operations build a finite set of identity affecting operations. Depending on the applicability of operations on identity, objects can be divided into two main lifestyles: fusions and aggregates. We showed that the examples of these lifestyles are found in both the physical and the abstract domain.

The theory of lifestyles reduces the efforts for constructing the applications that need temporal database models. In order to build an application (e.g., for temporal GIS),

the designer must only instantiate his objects to appropriate lifestyles classes, and all necessary properties will be automatically deduced using the inheritance mechanism.

10.1 Results and major findings

The major result of this thesis is the formal model for a universal spatiotemporal database, capable of representing different classes of objects in a uniform way with respect to change in identity of objects. The “representation mapping” between the real world and the database model is essential: features are mapped to objects; identities are mapped to identifiers; properties to attributes. The broad term of cognitively assigned identity is reduced to the prototypical cases: objects that have crisp boundaries either in virtue of their physical appearance (like cars and buildings) or in virtue of an institutional agreement (like cadastre parcels and administrative units).

The criteria an identifier must fulfill (uniqueness, immutability, and non-reusability) hold in any database, but lifestyle operations are possible only if the transaction time dimension is supported and there is no overwriting of the existing data. Only if new identifiers are assigned automatically and a new version of a database is appended for each change, a consistent treatment of temporal links among identifiers is possible.

10.1.1 Lifestyles

Lifestyles are algebras of operations affecting object identifiers. Not all operations are applicable for every object class (e.g., car tires cannot be fused).

The set of operations affecting object identifiers in a temporal database is finite. Assuming that there are four basic operations (*create*, *destroy*, *suspend*, *resume*), the number of their compositions is an exhaustible set, covering a wide range of situations encountered in the application domain. This set is developed in a logical system, based on category theory, using functional composition only.

```
create (is, ot) = uncurry (updateObj f) . pair (getID, id) . newObj ot
      where f = addAtt Preds (Vp is)
destroy i = cond p (f, g) where
      p = destroyable . getObjType . selectObj i
      f = deleteObj i
      g = error ("the object" ++ show i ++ "is not destroyable")
```

```
suspend i = cond p (f, g) where
      p = suspendable . getObjType . selectObj i
      f = updateObj suspendObj i
      g = error ("the object" ++ show i ++ "is not suspendable")

resume i = cond p (f, g) where
      p = queryObj suspended i
      f = updateObj resumeObj i
      g = error ("the object" ++ show i ++ "is already suspended")
```

All possible compositions of the four basic operations are divided into 5 classes: evolvable, constructive aggregates, weak aggregates, constructive fusions, and weak fusions. The point-free formulae follow (details are given in Chapter 7):

```
evolve i = uncurry (set Preds (Vp (wrap i)))
  . pair (getID, destroy i) . uncurry (updateObj' setAttribs)
  . pair (pair (getID, getAttribs. selectObj i), id) . uncurry create
  . assocl. pair (nil, pair (getObjType . selectObj i, id))
       where  updateObj' f (i,x) = updateObj (f x) i

aggregate is t = uncurry (addRels PartOf is) . createWithID ([],t)
                     . (flip.foldr) suspend is
segregate i = (uncurry.flip.foldr) resume . pair (getRels PartOf i, g)
      where g = deleteRels PartOf i . destroy i

waggregate is i = (flip.foldr) suspend is.addRels PartOf is i.resume i

wsegregate i = (uncurry.flip.foldr) resume . pair (getRels PartOf i,g)
      where g = deleteRels PartOf i . suspend i

fusion is t = (flip . foldr) destroy is . create (is, t)

fissionN n i = uncurry (createN n) . pair (f, destroy i)
      where f = pair (wrap . const i, getObjType . selectObj i)

restructure is t n = uncurry (fissionN n).pair (getID,id).fusion is t

wfusion is i = (flip . foldr) destroy is . resume i

wfissionN n i = uncurry (createN n) . pair (f, suspend i)
      where f = pair (wrap . const i, getObjType . selectObj i)
```

The difference between a constructive fusion (*fusion*) and a weak fusion (*wfusion*) is in the semantics of underlying operations: a constructive fusion *creates* a new object, whereas a weak fusion *resumes* an already existing one. In the same manner, a constructive fission *destroys* an object, whereas a weak fission *suspends* an object. Both constructive and weak fusions *destroy* fused objects. Thus, a fusion is always irreversible.

The rationale behind constructive and weak aggregations (segregations) is similar to constructive and weak fusions (fissions) with an essential difference: both

constructive and weak aggregations *suspend* the aggregated objects. Thus, a weak aggregation is a reversible operation.

The theory of lifestyles is compared to other prominent proposals for categorization of operations affecting object identity: the proposal by Al-Taha and Barrera (1994) and the proposal by Hornsby and Egenhofer (1997). In Section 7.3, we show that all well-defined operations in both proposals can be formally modeled with lifestyles. We provided translations of those operations in functional language. The theory of lifestyles is a step forward with respect to other proposals, because it is more general an powerful than the proposal of Al-Taha and Barrera, and it is conceptually simpler than, yet equally powerful as the proposal of Hornsby and Egenhofer.

10.1.2 Application of lifestyles

Objects are composed in aggregates, and details about single parts are hidden until we change the level of abstraction (a car is perceived as a unity as long as it functions properly; its tire becomes important if it is broken). The relation *part-of* (aggregation) matters when identifiers are concerned, because the identifiers of parts are suppressed in a whole. At the same time the relation *member-of* (association) does not change the identifiers: an object can be simultaneously member of many sets, none of which diminish its identity (a car can be a member of all red objects, of all movable objects, etc.).

A categorization of objects existing in the real world is proposed and each category is formalized. Lifestyles of physical (tangible) objects could be applied on abstract (non-tangible) objects using metaphorical transfer. Operations in one domain (e.g., trees with fruits) are applied to another domain (e.g., cadastre parcels with usufruct rights) without altering the definitions of operations. The dependencies among common classes in physical and abstract domains are shown in Figure 10.1. This leads to the simplification of the overall model, enabling the database designer to reuse a significant amount of code. Modeling of abstract objects (institutional facts) is of enormous importance for future GIS. The possibility to deal with such objects using simpler models of physical objects allows better insight into their nature as well.

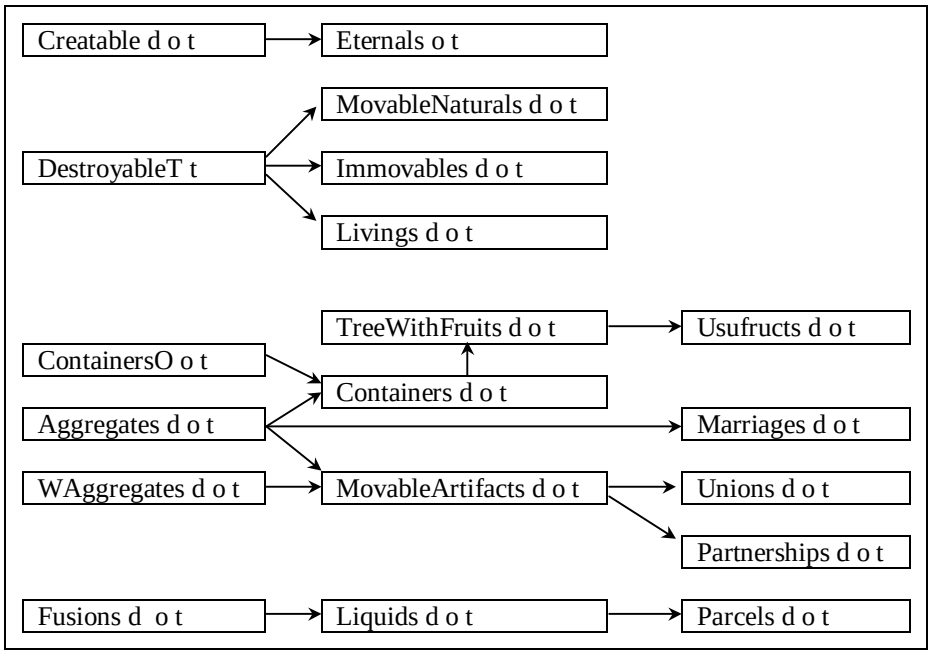

Figure 10.1: Classes hierarchy - from generic lifestyles along physical objects to abstract objects.

Transformations between the two different concepts of altering temporal database (database versioning and object versioning) are lossless - they transfer complete information from one view to another. The difference between the techniques lies in the different dimensions for grouping thematic and temporal elements. Database versioning groups thematic elements (objects) for each temporal element. Object versioning groups temporal elements for each thematic element (object), see Figure 3.4. We provided the algorithms for transformations between versioning techniques in Section 3.3.2, and formalized the algorithms in the functional language in Section 6.5. Since the transformations are lossless, we developed a conceptual model based on the simpler, database versioning view.

10.1.3 Discussion

In this thesis, we provided a model of an object-oriented spatiotemporal database with the emphasis on changes in identifiers of objects. The change in attributes was not the topic of discussion. An object may change its identity by changing its attributes. This is modeled as the evolution construct, but the semantic decision how much change in attributes is necessary for an object to evolve is left to the human expert for a particular domain. Such criteria could be formalized for different application areas and then added as compositions to the generic operation *evolve*.

A single relationship between objects is exclusively discussed: the mereological relation "part of". This relation is the basis for cognitive process of hierarchical abstraction: humans tend to omit details about parts of the whole as long as the whole is functioning properly (e.g., the engine in a car). Another relation for constructing composite objects - the association "member-of" - is omitted from our model, because this relation does not change identity of objects involved.

The last remaining issue is how lifestyles help in designing temporal GIS. A hint how this can be done is provided in Chapters 8 and 9: the generic lifestyles framework is extended to suite the needs each particular application. A designer of GIS needs only to assign the objects of his model to appropriate lifestyles and provide the implementation of basic classes. The unified treatment of the change in object identifiers is automatically inherited. Models based on lifestyles would share common behavior of operations, leading to increased interoperability among the applications in different domains. The theory of lifestyles is an important step in the direction of interoperable temporal information systems.

10.2 Directions for future work

In this thesis, we concentrated exclusively on change of object identity. Properties (attributes) of objects are also worth investigating: how attributes change under the proposed rules of change? This is especially important for such qualities of objects that can be added or averaged in case a new object results from the fusion of several existing objects. The examples of such attributes include area, volume, and weight. In order to be averaged or summed up, such attributes must be quantitatively measurable either on an interval or on a ratio scale (Stevens 1946), in contrast to qualities that are only nominal (color), or ordinal (hardness).

Operations affecting object identity imply significant changes to the relations among the changed objects. A detailed analysis of these consequences with respect to the nature of relations is an interesting research topic, especially for GIS applications. For example, how the topological relations of an object are distributed on its child objects after the original object is fissioned?

Finally, further work is necessary on producing more complete categorizations of real world objects for specific domains and on the investigation of the requirements the

general model of lifestyles should fulfill to capture the semantics of these categorizations.

The temporal model implemented in this thesis was simple. Branching time with multiple futures is possible. Which temporal dimension (transaction, valid) is appropriate for dealing with lifestyles in branching time? Does cyclic time negate the identity uniqueness, or just disable destroying (and lifestyles based on destroying)?

In a rollback database, modeling of future times is not possible. If we add the valid time dimension, we could model possible alternatives. In that case, some interesting questions about levels of existence arise: a creation of a particular car may be projected into the future even before concrete, physical parts are aggregated into a new car as a moving object.

Finally, in our model, the user has complete freedom in manipulating the valid time dimension. It is possible to impose some constraints on the valid time. For example, in the transaction time, a fusion implies that the deletion time of fused objects and the creation time of the emerging object coincide. If this requirement is applied to the valid time dimension, the freedom of updating the database by the user is restricted.

BIBLIOGRAPHY

Allen, J. F. (1983). “Maintaining Knowledge about Temporal Intervals.” *Communications of the ACM*, 26(11), 832-843.

Al-Taha, K. (1992). “Temporal Reasoning in Cadastral Systems.” Ph.D. thesis, University of Maine.

Al-Taha, K., and Barrera, R. (1994). "Identities through Time". In Proceedings of *International Workshop on Requirements for Integrated Geographic Information Systems*, (Ehlers, ed.), in New Orleans, Louisiana, pp: 1-12.

Arpagaus, R. (1997). “Business Associations under Swiss Law.” *http://www.swissemb.org/legal/html/corporation.html* (The Embassy of Switzerland in Washington, D.C.).

Backus, J. (1978). “Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs.” *Communications of the ACM*, 21, 613-641.

Bird, R. (1998). *Introduction to Functional Programming Using Haskell*, Prentice Hall Europe, Hemel Hempstead, UK.

Bird, R., and de Moore, O. (1997). *Algebra of Programming*, Prentice Hall, London.

Bird, R., and Wadler, P. (1988). *Introduction to Functional Programming*, Prentice Hall International, Hemel Hempstead (UK).

Birkhoff, G. (1945). "Universal Algebra". In Proceedings of *First Canadian Math. Congress*, Published by Toronto University Press, pp: 310-326.

Birkhoff, G., and Lipson, J. D. (1970). “Heterogeneous Algebras.” *Journal of Combinatorial Theory*, 8, 115-133.

Bittner, S. (1998). “Die Modellierung eines Grundbuchsystems im Situationskalkül.” Master thesis, Department of Computer Science, University of Leipzig, Leipzig.

Bunge, M. A. (1977). *Treatise on Basic Philosophy: Vol. 3: Ontology I: The Furniture of the World*, Reidel, Boston.

Bunge, M. A. (1979). *Treatise on Basic Philosophy: Vol. 4: Ontology II: A World of Systems*, Reidel, Boston.

Cardelli, L. (1997). “Type Systems.” *Handbook of Computer Science and Engineering*, CRC Press.

Cardelli, L., and Wegner, P. (1985). “On Understanding Types, Data Abstraction, and Polymorphism.” *ACM Computing Surveys*, 17(4), 471 - 522.

Casati, R., and Varzi, A. C. (1994). *Holes and Other Superficialities*, MIT Press, Cambridge, Mass.

Cattell, R. G. G., and Barry, D. K. (1997). *Object Database Standard: ODMG 2.0*, Morgan Kaufmann, San Francisco, CA.

Chen, P. P.-S. (1976). “The Entity-Relationship Model - Toward a Unified View of Data.” *ACM Transactions on Database Systems*, 1(1), 9 - 36.

Claramunt, C., and Thériault, M. (1996). "Toward Semantics for Modelling Spatio-Temporal Processes within GIS". In Proceedings of *7th International Symposium on Spatial Data Handling*, (M.-J. Kraak and M. Molenaar, eds.), August 12-16, in Delft,The Netherlands, Published by International Geographical Union, Vol. 2, pp: 2.27-2.43.

Clifford, J., and Croker, A. (1988). “Objects in Time.” *Database Engineering*, 7(4), 189-196.

Clifford, J., and Isakowitz, T. (1994). "Modeling Time: Adequacy of Three Distinct Time Concepts for Temporal Databases". In Proceedings of *Advances in Database Technology - EDBT*, (J. Matthias, J. Bubenko, and K. Jeffery, eds.), March 28-31, in Cambridge, UK, Published by Springer -Verlag, Lecture Notes in Computer Science, pp: 215-230.

Codd, E. (1979). “Extending the database relational model to capture more meaning.” *ACM TODS*, 4(4), 379-434.

Egenhofer, M. J., and Mark, D. M. (1995). “Naive Geography.” *Spatial Information Theory - A Theoretical Basis for GIS*, (A. U. Frank and W. Kuhn, eds.), Springer-Verlag, Berlin, 1-15.

Ehrich, H.-D., Gogolla, M., and Lipeck, U. W. (1989). *Algebraische Spezifikation abstrakter Datentypen*, B.G. Teubner, Stuttgart.

Eilenberg, S., and Mac Lane, S. (1945). “General Theory of Natural Equivalences.” *Transactions of the American Mathematical Society*, 58, 231-294.

Frank, A. U. (1994). "Qualitative temporal reasoning in GIS - ordered time scales". In Proceedings of *Sixth International Symposium on Spatial Data Handling, SDH'94*, (T. C. Waugh and R. G. Healey, eds.), Jan. 94, in Edinburgh, Scotland, Sept. 5-9, 1994, Published by IGU Commission on GIS, Vol. 1, pp: 410-430.

Frank, A. U. (1996). “The Prevalence of Objects with Sharp Boundaries in GIS.” *Geographic Objects with Indeterminate Boundaries*, (P. A. Burrough and A. U. Frank, eds.), Taylor & Francis, London, 29-40.

Frank, A. U. (1998a). “Different types of 'times' in GIS.” *Spatial and Temporal Reasoning in GIS*, (M. J. Egenhofer and R. G. Golledge, eds.), Oxford University Press, New York, 40-61.

Frank, A. U. (1998b). "GIS for Politics". In Proceedings of *GIS Planet'98*, in Dordrecht, the Netherlands (September 7-11, 1998).

Frank, A. U., and Kuhn, W. (1995). “Specifying Open GIS with Functional Languages.” *Advances in Spatial Databases (4th Int. Symposium on Large Spatial Databases, SSD'95, in Portland, USA)*, (M. J. Egenhofer and J. R. Herring, eds.), Springer-Verlag, 184-195.

Gruber, T. (1993). “A translation approach to portable ontologies.” *Knowledge Acquisition*, 5(2), 199-220.

Guttag, J. V., Horowitz, E., and Musser, D. R. (1978). “Abstract Data Types and Software Validation.” *Comm. ACM*, 21(12), 1048-1064.

Hawking, S. W. (1988). *A Brief History of Time*, Bantam, New York.

Hayes, P. J. (1978). “The Naive Physics Manifesto.” *Expert Systems in the Microelectronic Age*, (D. Mitchie, ed.), Edinburgh University Press, Edinburgh, 242-270.

Hayes, P. J. (1985a). “Naive Physics I: Ontology for Liquids.” *Formal Theories of the Commonsense World*, (J. R. Hobbs and R. C. Moore, eds.), Ablex Publishing, Norwood, NJ, 71-107.

Hayes, P. J. (1985b). “The Second Naive Physics Manifesto.” *Formal Theories of the Commonsense World*, (J. R. Hobbs and R. C. Moore, eds.), Ablex Publishing, Norwood, NJ, 1-36.

Herring, J., Egenhofer, M. J., and Frank, A. U. (1990). "Using Category Theory to Model GIS Applications". In Proceedings of *4th International Symposium on Spatial Data Handling*, (K. Brassel, ed.), in Zurich, Switzerland, Published by International Geographical Union IGU, Commission on Geographic Information Systems, Vol. 2, pp: 820-829.

Hobbs, J., and Moore, R. C. (1985). “Formal Theories of the Commonsense World.” Ablex Series in Artificial Intelligence, Ablex Publishing Corp., Norwood, NJ.

Hofstadter, D. R. (1979). *Gödel, Escher, Bach: An Eternal Golden Braid*, Vintage Books, New York.

Hornsby, K., and Egenhofer, M. J. (1997). “Qualitative Representation of Change.” *Spatial Information Theory - A Theoretical Basis for GIS (International Conference COSIT'97)*, (S. C. Hirtle and A. U. Frank, eds.), Springer-Verlag, Berlin-Heidelberg, 15-33.

Hornsby, K., and Egenhofer, M. J. (1998). "Identity-Based Change Operations for Composite Objects". In Proceedings of *8th Internal Symposium on Spatial Data Handling*, (T. K. Poiker and N. Chrisman, eds.), July 11-15, 1998, in Vancouver, Published by International Geographical Union, pp: 202-213.

Jensen, C. C., and Dyreson, C. E. (1998). "The Consensus Glossary of Temporal Database Concepts - February 1998 Version". In Proceedings of *Temporal Database - Research and Practice*, (O. Etzion, S. Jajodia, and S. Sripada, eds.), Published by Springer Verlag, Lecture Notes in Computer Science 1399, pp: 367-405.

Jensen, C. C., and Snodgrass, R. T. (1992). "Temporal Specialization and Generalization". In Proceedings of *IEEE Transactions on Knowledge and Data Engineering*.

Johnson, M. (1993). *Moral Imagination - Implications of Cognitive Science for Ethics*, The University of Chicago Press.

Jones, M. P. (1991). “An Introduction to Gofer. Technical Report.”, Yale University.

Jones, M. P. (1995). “A system of constructor classes: overloading and implicit higher-order polymorphism.” *Functional Programming*, 5(1), 1-35.

Jones, S. P., Jones, M., and Meijer, E. (1997). “Type classes: an exploration of the design space.”, Universities of Glasgow, Nottingham and Utrecht.

Katsuno, H., and Mendelzon, A. O. (1991). "On the Difference between Updating a Knowledge Base and Revising it". In Proceedings of *2nd International Conference on Principles of Knowledge Representation and Reasoning*, (J. Allen, R. Fikes, and E. Sandewall, eds.), April, in Cambridge, MA, USA, Published by Morgan Kaufmann Publishers, pp: 387-394.

Khoshafian, S., and Abnous, R. (1990). *Object Orientation - Concepts, Languages, Databases, User Interfaces*, John Wiley & Sons, New York, NY.

Kim, S.-K., and Chakravarthy, S. (1994). "Modeling Time: Adequacy of Three Distinct Time Concepts for Temporal Databases". In Proceedings of *Advances in Database Technology - EDBT*, (J. Matthias, J. Bubenko, and K. Jeffery, eds.), March 28-31, in Cambridge, UK, Published by Springer -Verlag, Lecture Notes in Computer Science, pp: 475-491.

Kuipers, B. (1978). "Modeling Spatial Knowledge." *Cognitive Science*, 2(2), 129-154.

Kuipers, B. (1994). *Qualitative Reasoning: Modeling and Simulation with Incomplete Knowledge*, MIT Press, Cambridge, MA.

Lakoff, G. (1987). *Women, Fire, and Dangerous Things: What Categories Reveal About the Mind*, University of Chicago Press, Chicago, IL.

Lakoff, G., and Johnson, M. (1980). *Metaphors We Live By*, University of Chicago Press, Chicago.

Langran, G. (1989). "A review of temporal database research and its use in GIS applications." *IJGIS*, 3(3), 215-232.

Lenat, D. G., Guha, R. V., Pittman, K., Pratt, D., and Shepherd, M. (1990). "CYC: Toward programs with common sense." *Communications of the ACM*, 33(8), 30 - 49.

Liskov, B., and Guttag, J. (1986). *Abstraction and Specification in Program Development*, MIT Press, Cambridge, MA.

Loeckx, J., Ehrich, H.-D., and Wolf, M. (1996). *Specification of Abstract Data Types*, Wiley, Teubner.

Mark, D. M., and Frank, A. U. (1996). "Experiential and Formal Models of Geographic Space." *Environment and Planning, Series B*, 23, 3-24.

McCarthy, J. (1957). "Situations, actions and causal laws." *AI-Memo 1*, Stanford University, Stanford, CA.

McCarthy, J., and Hayes, P. J. (1969). “Some Philosophical Problems from the Standpoint of Artificial Intelligence.” *Machine Intelligence 4*, (B. Meltzer and D. Michie, eds.), Edinburgh University Press, Edinburgh, 463-502.

Milner, R. (1978). “A Theory of Type Polymorphism in Programming.” *Journal of Computer and System Sciences*, 17, 348-375.

Mulligan, K., and Smith, B. (1986). “A Relational Theory of the Act.” *Topoi*, 5(2), 115-130.

Peterson, J., Hammond, K., Augustsson, L., Boutel, B., Burton, W., Fasel, J., Gordon, A. D., Hughes, J., Hudak, P., Johnsson, T., Jones, M., Meijer, E., Jones, S. P., Reid, A., and Wadler, P. (1997). “The Haskell 1.4 Report.” *http://haskell.org/report/index.html*.

Reiter, R. (1984). “Towards a logical reconstruction of relational database theory.” *On Conceptual Modelling, Perspectives from Artificial Intelligence, Databases, and Programming Languages*, (M. L. Brodie, M. Mylopolous, and L. Schmidt, eds.), Springer Verlag, New York, 191-233.

Reiter, R. (1994). “On Specifying Database Updates.” *The Journal of Logic Programming*, 19(20).

Reiter, R. (in preparation). *Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems*, University of Toronto.

Rumbaugh, J., Blacha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1991). *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ.

Sarda, N. (1990). “Extensions to SQL for Historical Databases.” *IEEE Transactions on Knowledge and Data Engineering*, 2(2), 220-230.

Schönenberger, W. (1976). *Schweizerisches Zivilgesetzbuch mit Obligationenrecht*, Shulthess Polygraphischer Verlag AG, Zürich.

Searle, J. R. (1995). *The Construction of Social Reality*, The Free Press, New York.

Smith, B. (1982). “Parts and Moments - Studies in Logic and Formal Ontology.” Analytica, Philosophia Verlag, München.

Smith, B. (1999). “An Introduction to Ontology.”, NCGIA, Bad Harbor, Maine.

Smith, B. (to appear). “Objects and their Environments: from Aristotle to Ecological Ontology.” *Life and Motion of Socio-Economic Units*, (A. U. Frank, J. Raper, and J.-P. Cheylan, eds.), Taylor & Francis, London.

Smith, B., and Zaibert, L. (1996). “Prolegomena to a Metaphysics of Real Estate.” *Shadows and Socio-Economic Units. Foundations of Formal Geography*, (R. Casati, ed.), Department of Geoinformation, Vienna, 151-155.

Snodgrass, R. T. (1987). “The temporal query language TQUEL.” *ACM Transactions on Database Systems*, 12, 247-298.

Snodgrass, R. T. (1992). “Temporal Databases.” *Theories and Methods of Spatio-Temporal Reasoning in Geographic Space*, (A. U. Frank, I. Campari, and U. Formentini, eds.), Springer-Verlag, Heidelberg-Berlin, 22-64.

Snodgrass, R. T. (1995a). “Temporal Object-Oriented Databases: A Critical Comparison.” *Modern Database Systems - The Object Model, Interoperability, and Beyond*, (W. Kim, ed.), Addison-Wesley, New York, 386-408.

Snodgrass, R. T. (1995b). *The TSQL2 Temporal Query Language*, Kluwer.

Stevens, S. S. (1946). “On the theory of scales of measurement.” *Science*, 103(2684), 677 - 680.

Stonebraker, M., and Rowe, L. A. (1986). “The Design of POSTGRES.” *ACM-SIGMOD International Conference on the Management of Data*.

Tansel, A. U. (1986). “Adding Time Dimension to Relational Model and Extending Relational Algebra.” *Information Systems*, 11(4), 343-355.

Tarski, A. (1946). *Introduction to logic and to the methodology of deductive sciences*, Oxford University Press, New York.

Thompson, S. (1999). *Haskell - The Craft of Functional Programming - Second Edition*, Addison-Wesley, Harlow, UK.

van Oosterom, P. (1997). "Maintaining consistent topology including historical data in a large spatial database". In Proceedings of *Autocarto 13*, in Seattle, WA, USA, Published by ACSM/ASPRS, Vol. 5, pp: 327-336.

Walters, R. F. C. (1991). *Categories and computer science*, Carslaw Publications, Cambridge, UK.

Wand, Y. (1989). “A Proposal for a Formal Model of Objects.” *Object-Oriented Concepts, Databases, and Applications*, (W. Kim and F. H. Lochovsky, eds.), Addison-Wesley, New York, 537-559.

Worboys, M. F. (1994). "Unifying the Spatial and Temporal Components of Geographical Information". In Proceedings of *Sixth International Symposium on Spatial Data Handling*, (T. C. Waugh and R. G. Healey, eds.), in Edinburgh, Published by AGI, Vol. 2, pp: 505 - 517.

Worboys, M. F. (1995). *GIS: A Computing Perspective*, Taylor & Francis, London.

GOFER PROJECT FILE

```
-- tdbProj.gp calls all other scripts
-- collection of Gofer scripts for the temporal database (chapter 6) and
--                             for all applications from chapters 8 and 9

adds.gs         -- from categorical prelude (Bird&deMoor 1997)
attrib.gs       -- attributes, values, value sets
object.gs       -- identifiers and objects
database.gs     -- database operations
snapshot.gs     -- implementations on snapshots
tempdb.gs       -- implementations on temporal databases
lifestyles.gs   -- lifestyle operations
comparison.gs   -- comparison with previous work
text.gs         -- text instances for all types so far
simplelife.gs   -- books and tables example
transf.gs       -- versionings transformation
chapter08.gs    -- chapter 8: (movable naturals, immovables,
                -- living, eternal, liquids)
movarts.gs      -- movable artifacts
containers.gs   -- containers
treeFruits.gs   -- trees with fruits
marriages.gs    -- chapter 9
partnerships.gs
usufruct.gs
unions.gs
parcels.gs
```

PRELUDE ADDITIONS

```
-- adds.gs
-- categorical additions to the standard Gofer prelude
-- based on Algebra of programming (Bird & de Moor 1997)

-- standard combinators:
outl :: (a,b) -> a
outr :: (a,b) -> b
swap :: (a,b) -> (b,a)
outl (a, b) = a
outr (a, b) = b
swap (a, b) = (b, a)

assocl :: (a,(b,c)) -> ((a,b),c)
assocr :: ((a,b),c) -> (a,(b,c))
assocl (a,(b,c)) = ((a,b),c)
assocr ((a,b),c) = (a,(b,c))

pair  :: (a -> b,a -> c) -> a -> (b,c)
cross :: (a -> b,c -> d) -> (a,c) -> (b,d)
cond  :: (a -> Bool) -> (a -> b,a -> b) -> a -> b
pair (f,g) a = (f a, g a)
cross (f,g) (a,b) = (f a, g b)
cond p (f,g) a = if (p a) then (f a) else (g a)

-- relations:
leq, eql, geq :: Ord a => (a, a) -> Bool
leq = uncurry (<=)
eql = uncurry (==)
geq = uncurry (>=)

false = const False
true  = const True
```

```
meet, join' :: (a -> Bool,a -> Bool) -> a -> Bool
meet (r,s) = cond r (s, false)  -- logical AND
join' (r,s) = cond r (true, s)  -- logical OR

--numerical functions:

plus, minus :: Num a => (a, a) -> a
plus = uncurry (+)
minus = uncurry (-)

-- list processing functions:
nil :: a -> [b]
nil = const []

wrap :: a -> [a]
wrap = cons . pair (id, nil)

cons :: (a,[a]) -> [a]
cons = uncurry (:)

-- cartesian product left
cpl :: ([a],b) -> [(a,b)]
cpl (x,b) = [(a,b) | a <- x]
-- cartesian product right
cpr :: (a,[b]) -> [(a,b)]
cpr (a,y) = [(a,b) | b <- y]

-- end of categorical additions

-- list update (for updateAtts in object.gs)
updateBy :: Eq b => (a -> b) -> a -> [a] -> [a]
updateBy f a = map (cond ((f a ==).f) (const a, id))

-- end of adds.gs
```

ATTRIBUTES

```
-- attrib.gs
-- attributes, values, value sets

class Attribs a where
    attrib      :: (ValueSet, Value) -> a
    getValueSet :: a -> ValueSet
    getValue    :: a -> Value
    setValue    :: Value -> a -> a
    selectAtt   :: ValueSet -> [a] -> a
    selectAtt s = head . filter ((s==).getValueSet)
class ValueSets vs v where
   checkV :: (vs, v) -> Bool
class Values v a where
   unwrapValue :: v -> a
   wrapValue   :: a -> v

data Attrib = Att (ValueSet, Value)
instance (Eq ValueSet, Eq Value) => Eq Attrib where
   (==) a b = getValueSet a == getValueSet b
              && getValue a == getValue b
instance Attribs Attrib where
   attrib = cond checkV (Att, error "incompatible value types")
   getValueSet (Att (s,v)) = s
   getValue    (Att (s,v)) = v
   setValue v  (Att (s,u)) = attrib (s,v)
```

```
data ValueSet = Name | Age | Preds | Alive | Amount | Capacity | Weight
instance Eq ValueSet where
   (==) Name Name = True
   (==) Age Age = True
   (==) Amount Amount = True
   (==) Capacity Capacity = True
   (==) Preds Preds = True
   (==) Alive Alive = True
   (==) Weight Weight = True
   (==) _ _ = False
instance ValueSets ValueSet Value where
   checkV (Name,   (Vs a)) = True
   checkV (Age,    (Vi a)) = True
   checkV (Amount, (Vf a)) = True
   checkV (Capacity, (Vf a)) = True
   checkV (Preds,  (Vp a)) = True
   checkV (Alive,  (Vb a)) = True
   checkV (Weight, (Vf a)) = True
   checkV (_, _) = False

data Value = Vs String | Vb Bool | Vi Int | Vf Float | Vp [Int]
instance Eq Value where
   (==) (Vs a) (Vs b) = a == b
   (==) (Vb a) (Vb b) = a == b
   (==) (Vi a) (Vi b) = a == b
   (==) (Vf a) (Vf b) = a == b
   (==) (Vp a) (Vp b) = a == b
   (==) _ _  = False
instance Values Value String where
   unwrapValue (Vs s) = s
   wrapValue s = Vs s
instance Values Value Bool where
   unwrapValue (Vb b) = b
   wrapValue b = Vb b
instance Values Value Int where
   unwrapValue (Vi i) = i
   wrapValue i = Vi i
instance Values Value Float where
   unwrapValue (Vf f) = f
   wrapValue f = Vf f
instance Values Value [Int] where
   unwrapValue (Vp is) = is
   wrapValue is = Vp is

instance  Num Value where
   (+) (Vf a) (Vf b) = Vf (a + b)
   (-) (Vf a) (Vf b) = Vf (a - b)

-- end of attrib.gs
```

IDENTITIFIERS AND OBJECTS

```
-- object.gs
-- identifiers and objects

class Eq i => IDs i where
    sameID, notSameID :: i -> i -> Bool
    nextID :: i -> i
    getID :: i -> ID
    sameID i j = getID i == getID j
    notSameID i = not . sameID i
class IDs (o t) => Objects o t where
    makeObj    :: (t, ID) -> o t
    getObjType :: o t -> t
    getAttribs :: o t -> [Attrib]
    setAttribs :: [Attrib] -> o t -> o t

    addAtt     :: ValueSet -> Value -> o t -> o t
    addAtt s v = uncurry setAttribs . pair (f . getAttribs, id)
       where f = cons . pair (const (attrib (s, v)), id)
    addAtts    :: [(ValueSet, Value)] -> o t -> o t
    addAtts = (flip.foldr) (uncurry addAtt)
    updateAtt  :: Eq ValueSet => ValueSet -> Value -> o t -> o t
    updateAtt s v = uncurry setAttribs . pair (f . getAttribs, id)
       where f = updateBy ((s==).getValueSet) (attrib (s, v))
    updateAtts :: Eq ValueSet => [(ValueSet, Value)] -> o t -> o t
    updateAtts = (flip.foldr) (uncurry updateAtt)

type ID = Int
instance IDs Int where
    nextID = (+1)
data Object t = Obj ID t [Attrib]
instance IDs (Object t) => Eq (Object t) where
    (==) = sameID
instance IDs (Object t) where
    sameID a b = sameID (getID a) (getID b)
    getID (Obj i t as) = i
instance Objects Object t where
    makeObj (t,i) = Obj i t []
    getObjType (Obj i t as) = t
    getAttribs (Obj i t as) = as
    setAttribs as (Obj i t bs) = Obj i t as
-- end of object.gs
```

DATABASE OPERATIONS

```
-- database.gs
-- database operations

class (Objects o t, IDs (s o t), Relatable t) => Snapshots s o t where
    getObjects :: s o t -> [o t]
    getRelations :: s o t -> [Rel]
    setObjects :: [o t] -> s o t -> s o t
    setRelations :: [Rel] -> s o t -> s o t

    liftS :: ([o t] -> [o t]) -> s o t -> s o t
    liftS f = uncurry setObjects . pair (f . getObjects, id)
    liftR :: ([Rel] -> [Rel]) -> s o t -> s o t
    liftR f = uncurry setRelations . pair (f . getRelations, id)
```

```
class TDBs td o t where
    getSnapshots :: td o t -> [Snapshot o t]
    setSnapshots :: [Snapshot o t] -> td o t -> td o t

class Snapshots d o t =>  Databases d o t where
    newObj    :: t -> d o t -> d o t
    existObj  :: ID -> d o t -> Bool
    existObjs :: [ID] -> d o t -> Bool
    deleteObj :: ID -> d o t -> d o t
    updateObj :: (o t -> o t) -> ID -> d o t -> d o t
    selectObj :: ID -> d o t -> o t
    queryObj  :: (o t -> x) -> ID -> d o t -> x

    queryObjs  :: (o t -> x) -> [ID] -> d o t -> [x]
    queryObjs q is = liftM (queryObj q) is
-- a shortcut for getting the Value from an object
    get :: ValueSet -> ID -> d o t -> Value
    get a = queryObj (getValue . selectAtt a . getAttribs)
-- a shortcut for updating the Attribute in an object
    set :: ValueSet -> Value -> ID -> d o t -> d o t
    set s v i = updateObj (updateAtt s v) i

    addRel    :: ID -> RelType -> ID -> d o t -> d o t
    addRels   :: RelType -> [ID] -> ID -> d o t -> d o t

    deleteRel  :: RelType -> (ID,ID) -> d o t -> d o t
-- involving a reltype and an ID on the right
    deleteRels :: RelType -> ID -> d o t -> d o t
-- involving an ID either on left or right
    deleteRelsID :: ID -> d o t -> d o t
    getRels    :: RelType -> ID -> d o t -> [ID]
    getConvRels :: RelType -> ID -> d o t -> [ID]

-- for queries and select
    liftQ :: TDBs d o t =>  (Snapshot o t -> x) -> d o t -> x
    liftQ f = f . head . getSnapshots

-- for updates, deletions and creations
    liftU :: TDBs d o t => (Snapshot o t -> Snapshot o t) -> d o t -> d o t
    liftU f = h . cross (cons . g, id) . pair (getSnapshots, id)
       where h = uncurry setSnapshots
             g = pair (f . head, id)
-- for operations on a list of identifiers (map)
    liftM :: (ID -> d o t -> x) -> [ID] -> d o t -> [x]
    liftM f is = map (uncurry f) . cpl . pair (const is, id)

class Relatable t where
   relatable :: (RelType, (t, t)) -> Bool

type Rel = (RelType, (ID, ID))
data RelType = PartOf | In | On | NoneRel
instance Eq RelType where
   (==) PartOf PartOf = True
   (==) In In = True
   (==) On On = True
   (==) _ _ = False
data Snapshot o t = Snap ID [o t] [Rel]

-- end of database.gs
```

SNAPSHOTS

```
-- snapshot.gs
-- implementation for a static database

instance Eq [o t] => Eq (Snapshot o t) where
   (==) (Snap i o r) (Snap j p s) = i == j && o == p && r == s
instance IDs (Snapshot o t) where
    getID (Snap i os rs) = i
    nextID (Snap i os rs) = Snap (nextID i) os rs
instance Snapshots Snapshot o t where
    getObjects (Snap i os rs) = os
    setObjects os (Snap i ps rs ) = Snap i os rs
    getRelations (Snap i os rs) = rs
    setRelations ts (Snap i os rs ) = Snap i os ts

instance Databases Snapshot o t where
    newObj t = nextID . uncurry setObjects .
       cross (cons . pair (makeObj.outl, outr), id) .
       cross (assocl. pair (const t, id), id) .
       pair (cross (getID, getObjects), outr) . pair (nextID, id)

    existObj i = cond p (false, true) where
       p = null . filter ((i==).getID) . getObjects

    existObjs is = and . liftM existObj is

    deleteObj i   = liftS f . liftR g where
       f = filter ((i/=).getID)
       g = filter (meet ((i/=).fst.snd, (i/=).snd.snd))

    updateObj f i = cond (existObj i) (g, h) where
       g = liftS (map (cond ((i==).getID) (f, id)))
       h = error ("the object " ++ show i ++ " does not exist.")

    selectObj i   = cond (existObj i) (f, g) where
       f = head . filter ((i==).getID) . getObjects
       g = error ("the object " ++ show i ++ " does not exist.")

    queryObj q i  = q . selectObj i

    addRel j t i = cond p (f, g) where
       p = relatable . pair (const t, pair (h i, h j))
       h a = queryObj getObjType a
       f = liftR (cons . pair (pair (const t, pair (const i, const j)), id))
       g = error "types are not relatable."
    addRels t is j = (flip . foldr) (addRel j t) is

    deleteRel t is = liftR (filter (join' ((t/=).outl, (is/=).outr)))
    deleteRels t i = liftR (filter (join' ((t/=).outl, (i/=).outr.outr)))
    deleteRelsID i = liftR (filter (meet ((i/=).outl.outr, (i/=).outr.outr)))
    getRels t i = map (outl.outr) . filter p . getRelations where
        p = meet ((t==).outl, (i==).outr.outr)
    getConvRels t i = map (outr.outr) . filter p . getRelations where
        p = meet ((t==).outl, (i==).outl.outr)

-- end of snapshot.gs
```

TEMPORAL DATABASE

```
-- tempdb.gs
-- implementation for temporal databases

data TDB o t = T [Snapshot o t]
instance Eq [Snapshot o t] => Eq (TDB o t) where
   (==) (T s) (T t) = s == t
instance (Databases TDB o t) => IDs (TDB o t) where
    getID = liftQ getID
instance Snapshots TDB o t
instance TDBs TDB o t where
   getSnapshots (T ss) = ss
   setSnapshots ss (T ts) = T ss
instance (TDBs TDB o t, Databases Snapshot o t)
   => Databases TDB o t where
   newObj t       = liftU (newObj t)
   deleteObj i    = liftU (deleteObj i)
   updateObj f i  = liftU (updateObj f i)
   existObj  i    = liftQ (existObj i)
   existObjs is   = liftQ (existObjs is)
   selectObj i    = liftQ (selectObj i)
   queryObj q i   = liftQ (queryObj q i)

   addRel j t i   = liftU (addRel j t i)
   addRels t is j = liftU (addRels t is j)

   deleteRel t is = liftU (deleteRel t is)
   deleteRels t i = liftU (deleteRels t i)
   deleteRelsID i = liftU (deleteRelsID i)
   getRels t i    = liftQ (getRels t i)
   getConvRels t i    = liftQ (getConvRels t i)

-- end of tempdb.gs
```

LIFESTYLE OPERATIONS

```
-- lifestyles.gs

class  Databases d o t => Creatable d o t where
   create       :: ([ID], t) -> d o t -> d o t
   createWithID :: ([ID], t) -> d o t -> (ID, d o t)
   createN      ::  Int -> ([ID], t) -> d o t -> d o t
   create (is, ot) = uncurry (updateObj f) . pair (getID, id) . newObj ot
      where f = addAtt Preds (Vp is)
   createWithID (is, ot) = pair (getID, id) . create (is, ot)
   createN n (is, ot) = flip (!!) n . iterate (create (is, ot))

class DestroyableT d where
   destroyable :: d -> Bool

class (DestroyableT t, Creatable d o t)
   => Destroyable d o t where
   destroy :: ID -> d o t -> d o t
   destroy i = cond p (f, g) where
      p = destroyable . getObjType . selectObj i
      f = deleteObj i
      g = error ("the object #" ++ show i ++ " is not destroyable")
```

```
class SuspendableT s where
   suspendable :: s -> Bool
class (Objects o t, SuspendableT t) => SuspendableO o t where
   suspended    :: o t -> Bool
   suspended = not . unwrapValue . getValue . selectAtt Alive . getAttribs
   suspendObj   :: o t -> o t
   suspendObj = updateAtt Alive (Vb False)
   resumeObj    :: o t -> o t
   resumeObj = updateAtt Alive (Vb True)
class (SuspendableO o t, Creatable d o t)
   => Suspendable d o t where
   suspend, resume :: ID -> d o t -> d o t
   suspend i = cond p (f, g) where
      p = suspendable . getObjType . selectObj i
      f = updateObj suspendObj i
      g = error ("the object #" ++ show i ++ " is not suspendable")
   resume i = cond p (f, g) where
      p = queryObj suspended i
      f = updateObj resumeObj i
      g = error ("the object #" ++ show i ++ " is already suspended")

class Destroyable d o t => Evolvable d o t where
   evolve :: ID -> d o t -> d o t
   evolve i = uncurry (set Preds (Vp (wrap i)))
        . pair (getID, destroy i) . uncurry (updateObj' setAttribs)
        . pair (pair (getID, getAttribs. selectObj i), id) . uncurry create
        . assocl. pair (nil , pair (getObjType . selectObj i, id))
       where  updateObj' f (i,x) = updateObj (f x) i

class Destroyable d o t => Fusions d o t where
   fusion :: [ID] -> t -> d o t -> d o t
   fusion is t = (flip . foldr) destroy is . create (is, t)
   fissionN :: Int -> ID -> d o t -> d o t
   fissionN n i = uncurry (createN n) . pair (f, destroy i)
       where f = pair (wrap . const i, getObjType . selectObj i)
   restructure :: [ID] -> t -> Int -> d o t -> d o t
   restructure is t n = uncurry (fissionN n) . pair (getID, id) . fusion is t

class (Destroyable d o t, Suspendable d o t)
   => WFusions d o t where
   wfusion :: [ID] -> ID -> d o t -> d o t
   wfusion is i = (flip . foldr) destroy is . resume i

   wfissionN :: Int -> ID -> d o t -> d o t
   wfissionN n i = uncurry (createN n) . pair (f, suspend i)
      where f = pair (wrap . const i, getObjType . selectObj i)

class (Destroyable d o t, Suspendable d o t)
   => Aggregates d o t where

   aggregate :: [ID] -> t -> d o t -> d o t
   aggregate is t = uncurry (addRels PartOf is) . createWithID ([],t)
                    . (flip.foldr) suspend is
   segregate   :: ID -> d o t -> d o t
   segregate i = (uncurry.flip.foldr) resume . pair (getRels PartOf i, g)
      where g = deleteRels PartOf i . destroy i

class Suspendable d o t => WAggregates d o t where
   waggregate :: [ID] -> ID -> d o t -> d o t
   waggregate is i = (flip . foldr) suspend is . addRels PartOf is i .resume i
   wsegregate   :: ID -> d o t -> d o t
   wsegregate i = (uncurry.flip.foldr) resume . pair (getRels PartOf i, g)
      where g = deleteRels PartOf i . suspend i
```

```
--instances:
instance Creatable Snapshot o t
instance Creatable Snapshot o t => Creatable TDB o t where
   create = liftU . create
   createN n = liftU . createN n
instance Destroyable Snapshot o t
instance Destroyable TDB o t
instance Suspendable Snapshot o t
instance Suspendable TDB o t
instance Evolvable Snapshot o t
instance Evolvable Snapshot o t => Evolvable TDB o t where
   evolve = liftU . evolve
instance Fusions Snapshot o t
instance Fusions TDB o t
instance WFusions Snapshot o t
instance WFusions TDB o t
instance Aggregates Snapshot o t
instance Aggregates TDB o t
instance WAggregates Snapshot o t
instance WAggregates TDB o t
instance SuspendableO Object t
-- end of lifestyles.gs
```

COMPARISON WITH PREVIOUS WORK

```
--comparison.gs

-- comparison with hornsby&egenhofer:
metamorphose :: Evolvable d o t => ID -> d o t -> d o t
metamorphose = evolve
spawn :: Creatable d o t => ID -> d o t -> d o t
spawn i = uncurry create . pair (f, id)
   where f = pair (wrap . const i, getObjType . selectObj i)

mergeH :: Fusions d o t => [ID] -> t -> d o t -> d o t
mergeH = fusion

generate ::  Creatable d o t => [ID] -> t -> d o t -> d o t
generate = curry create

mix :: Destroyable d o t => [ID] -> t -> d o t -> d o t
mix (i:is) t = destroy i . curry create is t

-- auxiliary functions
segregate'   :: Aggregates d o t => ID -> d o t -> ([ID], d o t)
segregate' i = pair (outl, (uncurry.flip.foldr) resume)
    . pair (getRels PartOf i, destroy i)
wsegregate'  :: WAggregates d o t => ID -> d o t -> ([ID], d o t)
wsegregate' i = pair (outl, (uncurry.flip.foldr) resume)
    . pair (getRels PartOf i, suspend i)

-- composite objects:
compound :: WAggregates d o t =>  ID -> ID -> d o t -> d o t
compound i j = uncurry (flip waggregate j)
      . cross (cons . pair (const i, id), id)
      . pair (getRels PartOf j, wsegregate j)

unite :: Aggregates d o t => [ID] -> t -> d o t -> d o t
unite = aggregate

combine :: Aggregates d o t =>  [ID] -> t -> d o t -> d o t
combine is t db = aggregate js t db where
   js = concat . map (outl . (flip segregate' db)) $ is
```

```
amalgamate :: (Fusions d o t, Aggregates d o t)
            => [ID] -> t -> t -> d o t -> d o t
amalgamate is t1 t2 db = uncurry (aggregate ns) (t1, db') where
   db' = outr (foldr fusion' (t2, db) jss)
   jss = transpose . map (outl . (flip segregate' db)) $ is
   fusion' is1 (t1,db1) = (t1, fusion is1 t1 db1)
   ns = [a+1 .. b]
   a = getID db
   b = a + length is
secede :: WAggregates d o t => ID -> ID -> d o t -> d o t
secede i j = uncurry (flip waggregate j)
       . cross (filter (i/=), id)
       . pair (getRels PartOf j, wsegregate j)
dissolve :: Aggregates d o t => ID -> d o t -> d o t
dissolve = segregate
-- end of comparison.gs
```

REPRESENTATIONS OF DATATYPES

```
-- text.gs
-- text instances for datatypes given in comments
--data Attrib = Att (ValueSet, Value)
instance Text Attrib where
   showsPrec 0 a = showString "" --. shows (getValueSet a)
       . showChar ' ' . shows (getValue a)
--data ValueSet = Name | Age | Preds | Alive | Amount | Capacity | Weight
instance Text ValueSet where
   showsPrec d Name  = showString "name   ="
   showsPrec d Age   = showString "age    ="
   showsPrec d Preds = showString "preds  ="
   showsPrec d Alive = showString "state  ="
   showsPrec d Amount = showString "amount ="
   showsPrec d Capacity = showString "capacity  ="
--data Value = Vs String | Vb Bool | Vi Int | Vf Float | Vp [Int]
instance Text Value where
   showsPrec d (Vs a) = shows a
   showsPrec d (Vb a) = if a then showString "resumed  "
                             else showString "suspended"
   showsPrec d (Vi a) = shows a
   showsPrec d (Vf a) = shows a
   showsPrec d (Vp a) = shows a
--data Object t = Obj ID t [Attrib]
instance (Objects Object t, Text t) => Text (Object t) where
   showsPrec d a = showString "\n   #" . shows (getID a) . showString " "
                 . shows (getObjType a) --. showString "  Attribs:"
                 . shows (getAttribs a)
--data RelType = PartOf | In | On | NoneRel
instance Text RelType where
   showsPrec d PartOf = showString " is part of "
   showsPrec d In = showString " is in "
   showsPrec d On = showString " is on "
--data Snapshot o t = Snap ID [o t] [Rel]
instance (IDs (Snapshot o t), Snapshots Snapshot o t, Text [o t])
   => Text (Snapshot o t) where
   showsPrec d a = showString "\nSnapshot\n Latest ID =" . shows (getID a)
         . showString "\n  Objects: " . shows (getObjects a)
         . showString "\n  Relations: " . shows (getRelations a)
--data TDB o t = T [Snapshot o t]
instance (TDBs TDB o t, Text [Snapshot o t]) => Text (TDB o t) where
   showsPrec d a = showString "\nTDB " . shows (getSnapshots a)
instance Text Rel where
   showsPrec d (a, (b,c)) = showString "\n   " . shows b . shows a . shows c
x :: (Text (Snapshot o t), Databases d o t, TDBs d o t) => d o t -> String
x = liftQ show -- shortcut for showing the latest snapshot of the database
-- end of text.gs
```

AN EXAMPLE OF DATABASE (SECTION 6.4)

```
-- simplelife.gs

-- implementation of lifestyles for ObjType
-- with examples for Section 6.4

data ObjType = Book | Table | Room
instance Text ObjType where
   showsPrec d Book  = showString "Book "
   showsPrec d Table = showString "Table"
   showsPrec d Room  = showString "Room "
instance Relatable ObjType where
   relatable (On, (Book, Table)) = True
   relatable (In, (Table, Room)) = True
   relatable _ = False
class Destroyable d o t => Simples d o t where
   createSimple :: String -> ([ID], t) -> d o t -> d o t
   createSimple s t = uncurry (updateObj f) . createWithID t
      where f = addAtts [(Name, Vs s)]

instance Simples Snapshot Object ObjType
instance Simples TDB Object ObjType

-- optional instances for t = ObjType (could be SimpleType)
instance DestroyableT ObjType where
   destroyable Book  = True
   destroyable Table = True
   destroyable Room  = True
   destroyable _ = False
instance SuspendableT ObjType where
   suspendable Book  = True
   suspendable Table = True
   suspendable Room  = True
   suspendable _ = False

-----------------------------------------------------
-- examples for section 6.4 two books and two tables
-----------------------------------------------------
td0, td7 :: TDB Object ObjType
td1, td2, td3, td4, td5, td6
   :: Snapshot Object ObjType -> Snapshot Object ObjType
td0 = T [Snap 0 [] []]
td1 = createSimple "BookA" ([], Book)
td2 = createSimple "BookB" ([], Book)
td3 = createSimple "TableA" ([], Table)
td4 = createSimple "TableB" ([], Table)
td5 = addRel 3 On 1
td6 = addRel 4 On 2
td7 = liftU (td6.td5.td4.td3.td2.td1) td0

tst1, tst2 :: Bool
tst3 :: Object ObjType
tst4 :: Value
tst5 :: String
tst6 :: Object ObjType
tst7 :: [Rel]
tst8 :: TDB Object ObjType

tst1 = existObj 4 td7
-- True
tst2 = existObj 4 (deleteObj 4 td7)
-- False
tst3 = selectObj 4 (deleteObj 4 td7)
-- error: the object 4 does not exist.
```

```
tst4 = get Name 1 td7
-- Vs "Book1"
tst5 = unwrapValue (get Name 1 td7)
-- Book1
tst6 = selectObj 3 td7
-- Obj 3 Table [Att (Name,Vs "Table1")]
tst7 = liftQ getRelations (deleteRel On (1,3) td7)
-- [(On, (2,4))]
tst8 = addRel 1 On 4 td7
-- error: types are not relatable.

-- end of simplelife.gs
```

TRANSFORMATIONS BETWEEN VERSIONINGS

```
-- transfs.gs

-- transformation between the fixed time (database versioning)
-- and the fixed theme (object versioning)

type Time = Int

class (Eq t, Eq o) => Groups t o where
  distrTime :: [(t,[o])] -> [(o,t)]
  distrTime = concat . map cpl . map swap

  findObjs :: [(o,t)] -> [o]
  findObjs = nub . map outl

-- select times for given object (DV -> OV)
  selTimes :: (o, [(o,t)]) -> [(o,t)]
  selTimes = uncurry filter . cross (flip ((==).outl), id)

  normObj :: [(o,t)] -> (o,[t])
  normObj = pair (head . map outl, map outr)

  toOV :: [(t,[o])] -> [(o,[t])]
  toOV = map (normObj.selTimes) . cpl . pair (findObjs, id) . distrTime

-- the opposite case (OV -> DV)
  distrObjs :: [(o,[t])] -> [(o,t)]
  distrObjs = concat . map cpr

  findTimes :: [(o,t)] -> [t]
  findTimes = nub . map outr

-- select objects at given time
  selObjs :: (t,[(o,t)]) -> [(o,t)]
  selObjs =  uncurry filter . cross (flip ((==).outr), id)
  normTime :: [(o,t)] -> ([o],t)
  normTime = pair (map outl, head . map outr)

  toDV :: [(o,[t])] -> [(t,[o])]
  toDV = map (swap.normTime.selObjs). cpl . pair (findTimes, id) . distrObjs
```

```
instance Groups Time ObjX
data ObjX = Ob ID ObjT Color
instance Text ObjX where
   showsPrec d (Ob i t c) = shows c . shows t
instance Eq ObjX where
  (==) (Ob i t c) (Ob j u d) = i == j && c == d
data ObjT = HouseT | CarT
instance Text ObjT where
   showsPrec d HouseT = showString "House"
   showsPrec d CarT = showString "Car"
data Color = Red | Blue | White
instance Text Color where
   showsPrec d Red = showString "red"
   showsPrec d Blue = showString "blue"
   showsPrec d White = showString "white"
instance Eq Color where
  Red == Red = True
  Blue == Blue = True
  White == White = True
  _ == _ = False

-- thesis example, sections 3.2.2 and 6.5
o1, o2, o3 :: ObjX
o1 = Ob 1 CarT Red
o2 = Ob 1 CarT Blue
o3 = Ob 2 HouseT White
dv1, dv2 :: [(Time, [ObjX])]
dv1 = [(1, [o1]), (2, [o2,o3]), (3, [o2,o3]), (4, [o3])]
ov1 = toOV dv1
dv2 = toDV ov1

-- end of transf.gs
```

PHYSICAL OBJECTS

```
-- chapter08.gs
-- applications from chapter 8

--the applications of lifestyles: each object is a class with attributes added
--during its creation
--8.1.1 movable naturals (stones,fruits)
--8.1.3 immovable geographic objects (mountains, buildings)
--8.2 living beings (persons)
--8.4 eternals


class Destroyable d o t => MovableNaturals d o t where
   createMovNat :: String -> Float -> ([ID], t) -> d o t -> d o t
   createMovNat name w a = uncurry (updateObj (addAtts as)) . createWithID a
      where as = [(Name, Vs name), (Preds, Vp []), (Weight, Vf w)]


-- immovable objects
class Destroyable d o t => Immovables d o t where
   createImmov :: String -> ([ID], t) -> d o t -> d o t
   createImmov name a = uncurry (updateObj (addAtts as))
       . pair (getID, id) . create a
      where as = [(Name, Vs name)]

-- living objects:
class Destroyable d o t => Livings d o t where
   createLiving :: String -> ([ID], t) -> d o t -> d o t
   createLiving name a = uncurry (updateObj (addAtts as)) . createWithID a
      where as = [(Name,  Vs name), (Preds, Vp (outl a))]
```

```
-- simple liquids
class (Fusions d o t) => Liquids d o t where
   createLiquid :: String -> Float -> ([ID], t) -> d o t -> d o t
   createLiquid name x a = uncurry (updateObj (addAtts as)) . createWithID a
      where as = [(Name, Vs name), (Preds, Vp (outl a)), (Amount, Vf x)]
   fusionLiquid :: [ID] -> t -> d o t -> d o t
   fusionLiquid is t d = f (fusion is t d) where
      f x =  updateObj (updateAtt Amount (Vf z)) (getID x) x
      z = sum (map (flip (queryObj g) d) is)
      g = unwrapValue . getValue . selectAtt Amount . getAttribs

-- eternals:
class (Creatable d o t) =>  Eternals d o t where
   createEternal :: String -> ([ID], t) -> d o t -> d o t
   createEternal name a = uncurry (updateObj (addAtts as)) . createWithID a
      where as = [(Name, Vs name), (Preds, Vp (outl a))]

-- EXAMPLES:
-- movable naturals
data MovNat = Fruit | Stone
instance Text MovNat where
   showsPrec d Fruit = showString "Fruit"
   showsPrec d Stone = showString "Stone"
instance Relatable MovNat where
   relatable (On, (Fruit, Stone)) = True
   relatable _ = False
instance Creatable TDB Object MovNat
instance DestroyableT MovNat where
   destroyable Fruit = True
   destroyable Stone = True
instance MovableNaturals Snapshot Object MovNat
mn0, mn2 :: TDB Object MovNat
mn0 = T [Snap 0 [] [] ]
mn2 = liftU (createMovNat "appleA" 0.4 ([],Fruit)
            .createMovNat "stoneA" 1.2 ([],Stone)
            .createMovNat "stoneB" 2.3 ([],Stone)) mn0
tstmn1 = existObj 3 (destroy 3 mn2)
-- False
tstmn2 = get Weight 3 mn2
-- Vf 2.3

-- immovables
data Immovable = Mountain | Building
instance Text Immovable where
   showsPrec d Mountain = showString "Mountain"
   showsPrec d Building = showString "Building"
instance Relatable Immovable
instance DestroyableT Immovable where
   destroyable Mountain = True
   destroyable Building = True
instance Immovables Snapshot Object Immovable
im0, im2 :: TDB Object Immovable
im0 = T [Snap 0 [] [] ]
im2 = liftU (createImmov "Alps  " ([], Mountain)
             .createImmov "HouseA" ([], Building)) im0
im3 = (uncurry (set Name (Vs "MuseumA")).pair (getID, id).(evolve 1)) im2
```

```
--simple liquids
data Liquid = Water | Wine
instance Text Liquid where
   showsPrec d Water = showString "water"
   showsPrec d Wine  = showString "wine "
instance Relatable Liquid
instance DestroyableT Liquid where
   destroyable Water = True
   destroyable Wine = True
instance Liquids Snapshot Object Liquid
instance Fusions Snapshot Object Liquid
w0, w2, w3 :: TDB Object Liquid
w0 = T [Snap 0 [] [] ]
w2 = liftU ( fissionN 3 3
           . fusion [1,2] Water
           . createLiquid "waterA" 2.4 ([], Water)
           . createLiquid "waterB" 2.8 ([], Water)
           )    w0
w3 = liftU (fusionLiquid [4,5,6] Water) w2

-- livings:
data Living = Person | Animal | Plant
instance Text Living where
   showsPrec d Person = showString "person"
   showsPrec d Animal = showString "animal"
   showsPrec d Plant  = showString "plant "
instance DestroyableT Living where
   destroyable Person = True
   destroyable Animal = True
   destroyable Plant  = True
instance Livings TDB Object Living
instance Livings Snapshot Object Living
instance Relatable Living
liv0, liv1 :: TDB Object Living
liv0 = T [Snap 0 [] [] ]
liv1 = liftU (createLiving "John" ([], Person)
              . createLiving "Mary" ([], Person)
              . createLiving "Sue " ([1,2], Person)) liv0

-- eternals
data Eternal = Star | Planet
instance Text Eternal where
   showsPrec d Star   = showString "star  "
   showsPrec d Planet = showString "planet"
instance Relatable Eternal
instance DestroyableT Eternal where
   destroyable Star = False
   destroyable Planet = False
instance Eternals TDB Object Eternal
instance Eternals Snapshot Object Eternal
e0, e1, e2 :: TDB Object Eternal
e0 = T [Snap 0 [] [] ]
e1 = liftU (createEternal "Sun" ([],Star)) e0
e2 = destroy 1 e1
-- error: the object #1 is not destroyable

-- end of chapter08.gs
```

MOVABLE ARTIFACTS

```
-- movarts.gs

-- movable artifacts (car, chassis, wheel, engine)
-- creation, aggregation, replacePart example

class (Aggregates d o t, WAggregates d o t) => MovableArtifacts d o t where
   createMovArt :: String -> ([ID], t) -> d o t -> d o t
   createMovArt name ti = uncurry (updateObj (addAtts as)) . createWithID ti
      where as = [(Name, Vs name), (Alive, Vb True)]
   aggregateMovArt :: String -> [ID] -> t -> d o t -> d o t
   aggregateMovArt name is t = uncurry (updateObj (addAtts as))
                             . pair (getID, id) . aggregate is t
      where as = [(Name, Vs name), (Alive, Vb True)]

   addPart :: ID -> ID -> d o t -> d o t
   addPart i j = uncurry (flip waggregate j)
      . cross (cons . pair (const i, id), id)
      . pair (getRels PartOf j, wsegregate j)

   removePart ::  ID -> ID -> d o t -> d o t
   removePart i j = uncurry (flip waggregate j)
       . cross (filter (i/=), id)
       . pair (getRels PartOf j, wsegregate j)

   replacePart :: ID -> ID -> ID -> d o t -> d o t
   replacePart i j k = uncurry (flip waggregate k)
      . cross (cons . pair (const i, filter (j/=)), id)
      . pair (getRels PartOf k, wsegregate k)

-- movable aritifacts
data MovArt = Car | Chassis | Engine | Wheel
instance Text MovArt where
   showsPrec d Car     = showString "Car    "
   showsPrec d Chassis = showString "Chassis"
   showsPrec d Engine  = showString "Engine "
   showsPrec d Wheel   = showString "Wheel  "

instance Relatable MovArt where
   relatable (PartOf, (Chassis, Car)) = True
   relatable (PartOf, (Engine, Car)) = True
   relatable (PartOf, (Wheel, Car)) = True
   relatable (PartOf, (_, Car)) = False
instance DestroyableT MovArt where
   destroyable Car = True
   destroyable Chassis = True
   destroyable Engine = True
   destroyable Wheel = True
instance SuspendableT MovArt where
   suspendable Car = True
   suspendable Chassis = True
   suspendable Engine = True
   suspendable Wheel = True
instance MovableArtifacts Snapshot Object MovArt
instance MovableArtifacts TDB Object MovArt

-- case study: the carA has chassisA, engineA, and wheels 1,2,3, and 4
--    change the wheel w3 with wheel w5
ma0, ma9 :: TDB Object MovArt
ma0 = T [Snap 0 [] []]
ma1, ma2, ma3, ma4, ma5, ma6, ma7, ma8 ::
            Snapshot Object MovArt -> Snapshot Object MovArt
ma1 = createMovArt "wheel-1 " ([], Engine)
```

```
ma2 = createMovArt "wheel-2 " ([], Chassis)
ma3 = createMovArt "wheel-3 " ([], Wheel)
ma4 = createMovArt "wheel-4 " ([], Wheel)
ma5 = createMovArt "wheel-5 " ([], Wheel)
ma6 = createMovArt "chassisA" ([], Wheel)
ma7 = createMovArt "engineA " ([], Wheel)
ma8 = aggregateMovArt "carA    " [1,2,3,4,6,7] Car
ma9 = liftU (ma8.ma7.ma6.ma5.ma4.ma3.ma2.ma1) ma0
-- all parts of a car as a list of IDs
tstma0 = getRels PartOf 8 ma9
-- all parts as a list of objects:
tstma1 =  map (flip selectObj ma9) (getRels PartOf 8 ma9)
-- use: liftQ show (outr tstma2)
tstma2 = pair (getRels PartOf 8, wsegregate 8) ma9
-- exchange wheel5 (5) and wheel2 (2) in the car (8)
tstma3 = liftU (replacePart 5 2 8) ma9
tstma4 = liftU (removePart 1 8) ma9
tstma5 = liftU (addPart 1 8) tstma4

-- end of movarts.gs
```

CONTAINERS

```
-- containers.gs
-- chapter 8.3
-- pourFromInto example, conservation law

class Objects o t => ContainersO o t where
   getAmount   :: o t -> Float
   getCapacity :: o t -> Float
   setAmount   :: (Float,o t) -> o t
   setCapacity :: (Float,o t) -> o t
   getAmount = unwrapValue . getValue . selectAtt Amount . getAttribs
   getCapacity = unwrapValue . getValue . selectAtt Capacity . getAttribs
   setAmount = uncurry (updateAtt Amount) . cross (wrapValue, id)
   setCapacity = uncurry (updateAtt Capacity) . cross (wrapValue, id)

   isEmpty :: o t -> Bool
   isEmpty = (==0.0) . getAmount

   pourIn :: (Float, o t) -> o t
   pourIn = cond p (f,g) where
      p = leq . pair (plus.cross(id,getAmount), getCapacity.outr)
      f = setAmount . pair (plus.cross (id,getAmount),outr)
      g = error "would overflow"

   takeOut :: (Float, o t) -> o t
   takeOut = cond p (f,g) where
      p = leq . cross (id, getAmount)
      f = setAmount . pair (minus.swap.cross (id,getAmount),outr)
      g = error "not enough in the container"

class (ContainersO o t, Aggregates d o t) => Containers d o t where
   createCont :: String -> Float -> Float -> ([ID], t) -> d o t -> d o t
   createCont name a c s = cond p (f, g) where
      p = const (a <= c)
      f = uncurry (updateObj h) . createWithID s
      g = error "amount cannot be greater than capacity"
      h = addAtts [(Name, Vs name), (Alive, Vb True),
          (Amount, Vf a), (Capacity, Vf c)]

   pourFromInto :: Float -> ID -> ID -> d o t -> d o t
   pourFromInto a i j = updateObj (curry pourIn a) j
                      . updateObj (curry takeOut a) i
```

```
data Container = Cup | Tea | FilledCup
instance Text Container where
   showsPrec d Cup = showString "Cup   "
   showsPrec d Tea = showString "Tea   "
   showsPrec d FilledCup = showString "Filled cup"
instance Relatable Container where
   relatable (In, (Tea, FilledCup)) = True
   relatable (PartOf, (Tea, FilledCup)) = True
   relatable (PartOf, (Cup, FilledCup)) = True
   relatable _ = False
instance DestroyableT Container where
   destroyable Cup = True
   destroyable Tea = True
   destroyable FilledCup = True
instance SuspendableT Container where
   suspendable Cup = True
   suspendable Tea = True
   suspendable FilledCup = True
instance ContainersO Object Container
instance Containers TDB Object Container
instance Containers Snapshot Object Container

-- examples:
cs0, cs5 :: TDB Object Container
cs1, cs2, cs3, cs4 :: Snapshot Object Container -> Snapshot Object Container
cs0 = T [Snap 0 [] []]
cs1 = createCont "firstCup " 4.0 10.0 ([],Cup)
cs2 = createCont "secondCup" 4.0 10.0 ([],Cup)
cs3 = createCont "teaA     " 5.0 5.0 ([],Tea)
cs4 = aggregate [1,3] FilledCup
cs5 = liftU (cs4 . cs3 . cs2 . cs1) cs0
--tcs1, tcs2 :: Value
tcs1 = get Amount 1 cs5
-- Vf 4.0
tcs2 = get Amount 1 (pourFromInto 3.0 1 2 cs5)
-- Vf 1.0
tcs3 = liftU (updateObj (curry pourIn 7.0) 2) cs5
-- owould overflow

-- end of containers.gs
```

TREES WITH FRUITS

```
-- treeFruits.gs
-- chapter 8.3
-- fruits can be just takenOut (collected) but not pouredInto"

class (ContainersO o t, Aggregates d o t) => TreeWithFruits d o t where
   createTree :: String -> Float -> ([ID], t) -> d o t -> d o t
   createTree name a s = uncurry (updateObj h) . createWithID s where
      h = addAtts [(Name, Vs name), (Alive, Vb True), (Amount, Vf a)]
   aggregateTree :: [ID] -> t -> d o t -> d o t
   aggregateTree is t = cond p (f, g) where
      p = eql . pair (const (length is), const 2)
      f = aggregate is t
      g = error "only a single fruits object allowed"


data Tree = ATree | Fruits | TreeWithFruits
instance ContainersO Object Tree  where
   pourIn = error " not possible "
   takeOut = cond p (f,g) where
      p = leq . cross (id, getAmount)
      f = setAmount . pair (minus.swap.cross (id,getAmount),outr)
      g = error "not enough fruits on the tree"
```

```
instance Text Tree where
   showsPrec d ATree          = showString "Tree     "
   showsPrec d Fruits         = showString "Fruits   "
   showsPrec d TreeWithFruits = showString "FruitTree"
instance Relatable Tree where
   relatable (PartOf, (ATree, TreeWithFruits)) = True
   relatable (PartOf, (Fruits, TreeWithFruits)) = True
   relatable _ = False
instance DestroyableT Tree where
   destroyable ATree = True
   destroyable Fruits = True
   destroyable TreeWithFruits = True
instance SuspendableT Tree where
   suspendable ATree = True
   suspendable Fruits = True
   suspendable TreeWithFruits = False
instance TreeWithFruits TDB Object Tree
instance TreeWithFruits Snapshot Object Tree

-- examples:
tf0, tf1 :: TDB Object Tree
tf0 = T [Snap 0 [] []]
tf1 = liftU   (createTree "TreeA  " 10.0 ([],ATree)
              . createTree "FruitsA"  5.0 ([],ATree)
              . aggregateTree [1,2] TreeWithFruits  ) tf0
ttf1 :: Float
ttf1 = queryObj getAmount 1 tf1
-- 5.0
ttf2 = updateObj (curry pourIn 7.0) 2 tf1
-- not possible
-- end of treeFruits.gs
```

MARRIAGES

```
-- marriages.gs, chapter 9.1.1
class (Eq t, Aggregates d o t) => Marriages d o t where
   createPerson :: String -> Int ->  ([ID], t) -> d o t -> d o t
   createPerson name age s = uncurry (updateObj h) . createWithID s where
      h = addAtts [(Name, Vs name), (Alive, Vb True), (Age, Vi age)]
   destroyPerson :: ID -> d o t -> d o t
   destroyPerson i = cond (married i) (f . pair (head . h i, g), g) where
      married x = not . null . h x
      h x = getConvRels PartOf x
      f = uncurry destroy
      g = destroy i
   createMarriage :: (ID,ID) -> t -> d o t -> d o t
   createMarriage (i, j) t = cond (meet (p,q)) (f, g) where
      p = uncurry (/=) . pair (h i, h j)
      h x = queryObj getObjType x
      q = meet (age i, age j)
      age x = geq . pair ( y . getAttribs . selectObj x, const ( 18))
      y = unwrapValue . getValue . selectAtt Age
      f = aggregate [i,j] t
      g = error "not a legal marriage!"
   divorceMarr :: ID -> d o t -> d o t
   divorceMarr = segregate
```

```
data Marr = Marriage | Male | Female
instance Eq Marr where
   (==) Male Male = True
   (==) Female Female = True
   (==) Marriage  Marriage = True
   (==) _ _ = False
instance Text Marr where
   showsPrec d Marriage = showString "Marriage"
   showsPrec d Male     = showString "Male    "
   showsPrec d Female   = showString "Female  "
instance Relatable Marr where
   relatable (PartOf, (Male, Marriage)) = True
   relatable (PartOf, (Female, Marriage)) = True
   relatable _ = False
instance SuspendableT Marr where
   suspendable Marriage = False
   suspendable Male = True
   suspendable Female = True
instance DestroyableT Marr where
   destroyable _ = True
instance Marriages TDB Object Marr
instance Marriages Snapshot Object Marr

-- examples:
mm0, mm4, mm5, mm6, mm7, mm8 :: TDB Object Marr
mm1, mm2, mm3 :: Snapshot Object Marr ->  Snapshot Object Marr
mm0 = T [Snap 0 [] []]
mm1 = createPerson "John" 20 ([], Male)
mm2 = createPerson "Mary" 20 ([], Female)
mm3 = createPerson "Sue " 17 ([], Female)
mm4 = liftU (mm3 . mm2 . mm1) mm0
mm5 = liftU (createMarriage (1,2) Marriage) mm4 -- OK
mm6 = liftU (createMarriage (1,3) Marriage) mm4 -- not legal
mm7 = liftU (destroyPerson 1) mm5
mm8 = liftU (divorceMarr 4) mm5

-- end of marriages.gs
```

PARTNERSHIPS

```
-- parnerships.gs

class (Containers d o t, MovableArtifacts d o t)
   => Partnerships d o t where
   createStockHolder :: String -> Float -> ([ID],t) -> d o t -> d o t
   createStockHolder name m s = uncurry (updateObj h) . createWithID s
      where
      h = addAtts [(Name, Vs name), (Alive, Vb True),
                   (Amount, Vf m), (Capacity, Vf 11000.0)]

   sumStocks :: [ID] -> ValueSet -> d o t -> Float
   sumStocks is a  = sum . map (unwrapValue . getValue . selectAtt a
      . getAttribs) . liftM selectObj is

   createCorporation :: String -> [ID] -> t -> d o t -> d o t
   createCorporation name ss t d =
        if s > 10000.0 then cond (meet (p,true)) (f,g) d
        else error " not enough capital" where
      p = geq . pair (const (length ss), const 3)
      s = sumStocks ss Amount d
      f = uncurry (updateObj (addAtts [(Name, Vs name), (Alive, Vb True),
          (Amount, Vf s)])) . pair (getID, id) . aggregate ss t
      g = error "founding of the corporation not possible"
```

```
   sellShares :: Float -> ID -> ID -> d o t -> d o t
   sellShares = pourFromInto

   sellAllShares :: ID -> ID -> ID -> d o t -> d o t
   sellAllShares a b c d = removePart a c d'
      where d' = pourFromInto f a b d
            f = unwrapValue (get Amount a d)

data Partnership = Corporation | StockHolder
instance Text Partnership where
   showsPrec d Corporation = showString "Corporation"
   showsPrec d StockHolder = showString "StockHolder"
instance Relatable Partnership where
   relatable (PartOf, (StockHolder, Corporation)) = True
   relatable _ = False
instance SuspendableT Partnership where
   suspendable Corporation = True
   suspendable StockHolder = True
instance DestroyableT Partnership where
   destroyable _ = True
instance Partnerships TDB Object Partnership
instance Partnerships Snapshot Object Partnership
instance Containers Snapshot Object Partnership
instance Containers TDB Object Partnership
instance ContainersO Object Partnership
instance MovableArtifacts TDB Object Partnership
instance MovableArtifacts Snapshot Object Partnership
instance Suspendable TDB Object Partnership
--instance WAggregates Snapshot Object Partnership

-- examples:
pa0, pa6 :: TDB Object Partnership
pa1, pa2, pa3, pa4, pa5 ::
   Snapshot Object Partnership -> Snapshot Object Partnership
pa0 = T [Snap 0 [] []]
pa1 = createStockHolder "holderA" 2000.0 ([],StockHolder)
pa2 = createStockHolder "holderB" 4000.0 ([],StockHolder)
pa3 = createStockHolder "holderC" 3000.0 ([],StockHolder)
pa4 = createStockHolder "holderD" 2000.0 ([],StockHolder)
pa5 = createCorporation "corporA" [1,2,3,4] Corporation
-- serialized transaction:
pa6 = liftU (pa5 . pa4 . pa3 . pa2 . pa1) pa0

pa7, pa8 :: TDB Object Partnership
-- shareholder A sells some shares (2500) to B
pa7 = liftU (sellShares 2500.0 2 1) pa6
-- shareholder A sells all shares to B in the corporation 5
pa8 = liftU (sellAllShares 2 1 5) pa6

{-
the result of: x pa6  -- starting situation
Snapshot
 Latest ID =5
  Objects: [
   #5 Corporation[ "corporA", resumed  , 11000.0, []],
   #4 StockHolder[ "holderD", suspended, 2000.0, 11000.0, []],
   #3 StockHolder[ "holderC", suspended, 3000.0, 11000.0, []],
   #2 StockHolder[ "holderB", suspended, 4000.0, 11000.0, []],
   #1 StockHolder[ "holderA", suspended, 2000.0, 11000.0, []]]
  Relations: [
   1 is part of 5,
   2 is part of 5,
   3 is part of 5,
   4 is part of 5]

the result of: x pa7 -- the sum of stock is the same,
                           B sold some shares to A
```

```
Snapshot
 Latest ID =5
  Objects: [
   #5 Corporation[ "corporA", resumed  , 11000.0, []],
   #4 StockHolder[ "holderD", suspended, 2000.0, 11000.0, []],
   #3 StockHolder[ "holderC", suspended, 3000.0, 11000.0, []],
   #2 StockHolder[ "holderB", suspended, 1500.0, 11000.0, []],
   #1 StockHolder[ "holderA", suspended, 4500.0, 11000.0, []]]
  Relations: [
   1 is part of 5,
   2 is part of 5,
   3 is part of 5,
   4 is part of 5]

the result of: show pa8 -- the sum of stocks is the same, B is free

Snapshot
 Latest ID =5
  Objects: [
   #5 Corporation[ "corporA", resumed  , 11000.0, []],
   #4 StockHolder[ "holderD", suspended, 2000.0, 11000.0, []],
   #3 StockHolder[ "holderC", suspended, 3000.0, 11000.0, []],
   #2 StockHolder[ "holderB", resumed  , 0.0, 11000.0, []],
   #1 StockHolder[ "holderA", suspended, 6000.0, 11000.0, []]]
  Relations: [
   1 is part of 5,
   3 is part of 5,
   4 is part of 5]

-}
-- end of partnerships.gs
```

USUFRUCT RIGHTS

```
-- usufruct.gs
-- chapter 9.2.3
-- usufruct <= tree (fruits can be just takenOut (collected) but not
-- pouredInto")
-- "usufruct rights are fruit trees"

class TreeWithFruits d o t => Usufructs d o t where
   createAParcel :: String -> Float -> ([ID], t) -> d o t -> d o t
   createAParcel = createTree
   createUsufruct :: [ID] -> t -> d o t -> d o t
   createUsufruct = aggregateTree

data UsufructRight = AParcel | Usufruct | ParcelWithUsufruct

instance Text UsufructRight where
   showsPrec d AParcel         = showString "Parcel     "
   showsPrec d Usufruct         = showString "Usufruct   "
   showsPrec d ParcelWithUsufruct = showString "FruitTree"
instance Relatable UsufructRight where
   relatable (PartOf, (AParcel, ParcelWithUsufruct)) = True
   relatable (PartOf, (Usufruct, ParcelWithUsufruct)) = True
   relatable _ = False
instance DestroyableT UsufructRight where
   destroyable AParcel = True
   destroyable Usufruct = True
   destroyable ParcelWithUsufruct = True
instance SuspendableT UsufructRight where
   suspendable AParcel = True
   suspendable Usufruct = True
   suspendable ParcelWithUsufruct = False
instance ContainersO Object UsufructRight  where
   pourIn = error " not possible "
   takeOut = cond p (f,g) where
      p = leq . cross (id, getAmount)
      f = setAmount . pair (minus.swap.cross (id,getAmount),outr)
      g = error "not enough usufruct on the parcel"
instance TreeWithFruits TDB Object UsufructRight
instance TreeWithFruits Snapshot Object UsufructRight
instance Usufructs TDB Object UsufructRight
instance Usufructs Snapshot Object UsufructRight

-- examples:
uf0, uf1, uf2, uf3, uf4 :: TDB Object UsufructRight
uf0 = T [Snap 0 [] []]
uf1 = liftU (createAParcel "parcelA    " 10.0 ([],AParcel)) uf0
uf2 = liftU (createAParcel "usufructA  " 10.0 ([],Usufruct)) uf1
uf3 = liftU (createAParcel "parcelB    " 5.0  ([],AParcel)) uf2
uf4 = liftU (createUsufruct [1,2] ParcelWithUsufruct) uf3

tuf1 :: Value
tuf1 = get Amount 3 uf4
-- Vf 5.0
tuf2 = updateObj (curry pourIn 7.0) 2 uf4
-- not possible

-- end of usufruct.gs
```

UNIONS

```
-- unions.gs

-- administrative units (country unions)
-- creation, aggregation, secede example

class MovableArtifacts d o t => Unions d o t where
   createUnit :: String -> ([ID], t) -> d o t -> d o t
   createUnit = createMovArt

   aggregateUnits :: String -> [ID] -> t -> d o t -> d o t
   aggregateUnits = aggregateMovArt

   addUnit :: ID -> ID -> d o t -> d o t
   addUnit = addPart

   secedeUnit :: ID  -> ID -> d o t -> d o t
   secedeUnit = removePart

-- movable aritifacts
data AdminUnit = State | Union
instance Text AdminUnit where
   showsPrec d State = showString "State"
   showsPrec d Union = showString "Union"

instance Relatable AdminUnit where
   relatable (PartOf, (State, Union)) = True
   relatable _ = False
instance DestroyableT AdminUnit where
   destroyable State = True
   destroyable Union = True
instance SuspendableT AdminUnit where
   suspendable State = True
   suspendable Union = True
--instance AdminUnitsO Object AdminUnit
instance MovableArtifacts TDB Object AdminUnit
instance MovableArtifacts Snapshot Object AdminUnit
instance Unions TDB Object AdminUnit
instance Unions Snapshot Object AdminUnit
-- case study: Canada and Quebec
--    secede Quebec, and put it back later
au0, au12 :: TDB Object AdminUnit
au11, au10, au9, au8, au7, au6, au5, au4, au3, au2, au1 ::
  Snapshot Object AdminUnit -> Snapshot Object AdminUnit
au0 = T [Snap 0 [] []]
au1  = createUnit "Quebec           " ([], State)
au2  = createUnit "Ontario          " ([], State)
au3  = createUnit "New Brunswick    " ([], State)
au4  = createUnit "Nova Scotia      " ([], State)
au5  = createUnit "Britisch Columbia" ([], State)
au6  = createUnit "Prince Edward Isl" ([], State)
au7  = createUnit "Alberta          " ([], State)
au8  = createUnit "Manitoba         " ([], State)
au9  = createUnit "Newfoundland     " ([], State)
au10 = createUnit "Saskatchewan     " ([], State)
au11 = aggregateUnits "Canada           " [1,2,3,4,5,6,7,8,9,10] Union
au12 = liftU (au11 . au10 . au9 . au8 . au7 . au6 . au5 . au4 . au3 . au2 .
au1) au0

-- all parts of canada as a list of IDs
tstau0 = getRels PartOf 11 au12
-- all parts as a list of objects:
tstau1 =  map (flip selectObj au12) (getRels PartOf 11 au12)
```

```
-- use: x (outr tstau2)
tstau2 = liftU (secedeUnit 1 11) au12
tstau3 = pair (getRels PartOf 11, wsegregate 11) au12
tstau4 = liftU (evolve 1) tstau2
tstau5 = liftU (set Name (Vs "Quebec Noveau    ") 12) tstau4
tstau6 = liftU (addUnit 12 11) tstau5

-- end of unions.gs
```

PARCELS

```
-- parcels.gs
-- parcels are liquids

class (Liquids d o t) => Parcels d o t where
   createParcel :: String -> Float -> ([ID], t) -> d o t -> d o t
   createParcel = createLiquid
   mergeParcel :: [ID] -> t -> d o t -> d o t
   mergeParcel = fusionLiquid

-- instances:
data Parcel = Parcel
instance Text Parcel where
   showsPrec d Parcel = showString "parcel"
instance Relatable Parcel
instance DestroyableT Parcel where
   destroyable Parcel = True
instance Parcels TDB Object Parcel
instance Parcels Snapshot Object Parcel
instance Liquids TDB Object Parcel
instance Liquids Snapshot Object Parcel

p0, p3, p4, p5, p6, p7 :: TDB Object Parcel
p1, p2 :: Snapshot Object Parcel -> Snapshot Object Parcel
p0 = T [Snap 0 [] [] ]
p1 = createParcel "parcelA" 2.4 ([], Parcel)
p2 = createParcel "parcelB" 2.8 ([], Parcel)
p3 = liftU (p2 . p1) p0
p4 = liftU (fusion [1,2] Parcel) p3
p5 = liftU (fissionN 3 3) p3
p6 = liftU (mergeParcel [1,2] Parcel) p3 -- 3 [1,2]
p7 = liftU (restructure [1,2] Parcel 4)  p3 --4,5,6,7

-- end of parcels.gs
```

Dipl.-Ing. Damir Medak

CURRICULUM VITAE

19th Aug 1968	born in Dubrovnik, Croatia,
1986	finishing high school, Metković, Croatia,
1986-1987	army service,
1987	beginning of studies at Faculty of Geodesy, University of Zagreb,
1988-1992	teaching assistant in areas of mathematics and informatics,
1989-1992	six different rector awards for best students of the university,
1993 May	defense of Master Thesis (advisor: Prof. Krešimir Čolić), the average mark: 4.7 (excellent = 5),
1993 July	employed as young scientist at the University of Zagreb,
1993 Oct	guest researcher at Graz University of Technology (three months grant given by Austrian government),
1994 June	participated in GPS-campaigns EUREF'94 and CRODYN'94,
1995 Sep	organization committee member of the GPS-campaign CROREF'95,
1995 Oct	Ph.-D. studies at Vienna University of Technology, (a grant given by Austrian government), advisor: Prof. Andre Frank,
1996 May	"Formalizing and Representing Change of Spatial Socio-Economic Units in GIS", ESF GISDATA Specialist Meeting, Nafplion (Greece),
1996 Oct	"Ontology of Cadastre" Workshop, Vienna,
1997 July	"Advances in Spatial Databases", 5th SSD, Berlin, Germany,
1997 Oct	"Chorochronos Intensive Workshop '97", Petronel Carnuntum, Austria,
1998 Sep	"3rd International Summer School on Advanced Functional Programming", Braga, Portugal.