Universidad Nacional de San Luis
Facultad de Ciencias Físico-Matemáticas y Naturales
Departamento de Informática

Procesamiento de Datos Politípicos

Claudia Mónica Necco

Asesor Científico: Prof.Ph.D. José Nuno Fonseca de Oliveira

Tesis para optar al Grado de Magister en Ciencias de la Computación

San Luis - Argentina

Diciembre 2004

Resumen

Muchos aspectos del *procesamiento de datos* son de naturaleza funcional y pueden beneficiarse con los recientes desarrollos en las áreas de cálculo y *programación funcional*.

El trabajo descripto en esta tesis es un intento de contribuir en estas líneas temáticas, en particular explotando el lenguaje funcional Haskell como herramienta de soporte.

El lenguaje Haskell se utiliza basicamente para animar un modelo abstracto del cálculo de una base de datos relacional, escrito en el estilo de una especificación formal orientada al modelo.

El modelo incorpora un monad $Error$ para capturar errores. Este monad (también llamado el monad de *Excepción*) expresa la estrategia para combinar computaciones que pueden disparar excepciones, que consiste en conducir las funciones desde el punto donde se produjo una excepción hasta el punto en que la misma es tratada.

Se presenta una colección de funciones que capturan algunas de las funcionalidades actualmente provistas por los productos que implementan bases de datos multidimensionales. En particular, funciones que permiten clasificar y reducir relaciones (tablas), las que convenientemente combinadas harán posible el análisis multidimensional de una base de datos.

El estudio de temas como parametricidad y genericidad (politipismo) conduce a otras extensiones del modelo. Se definen versiones genéricas de las operaciones relacionales estándares (paramétricas sobre el constructor de tipo) y de la operación de análisis multidimensional en lenguaje Generic Haskell.

Se sugiere una teoría de normalización de datos más general que la teoría de bases de datos relacionales estándar (de la cual ésta parece ser un caso particular), usando polimorfismo de orden superior y clases de constructores en Haskell 98.

Además de su animación, el modelo funcional es sometido a razonamiento y cálculo formales, preparando el camino para la eventual formulación politípica (genérica) del cálculo relacional estándar.

Agradecimientos

Deseo manifestar mi agradecimiento a la gente que me ha alentado y ayudado en esta etapa de mi vida.

A mi padre, por acompañarme en todos los momentos de mi desarrollo intelectual, dándome fuerzas con su fe en mí y sus oraciones.

A mi director de tesis, el profesor Nuno Fonseca de Oliveira, sin cuya ayuda y conocimientos no hubiera sido posible este trabajo. A él le debo el haberme reencontrado con la sensación tan particular de emoción-placer-motivación-felicidad que había experimentado hace muchos años (en la etapa escolar) al estudiar matemáticas. Sensación que a mi juicio se produce no sólo por la afinidad personal con el tema de estudio, sino por la calidad de los conductores del conocimiento. Me siento orgullosa de ser su alumna.

Por su intermedio, quiero agradecer además a la Universidad de Minho, en particular al Grupo de Lógica y Métodos Formales por la asistencia material y afectiva brindada durante mis estadías en Braga. Aprovecho este contexto para agradecer también al proyecto de investigación PURE:*Program Understanding and Re-engineering: Calculi and Applications* (http://www.di.uminho.pt/ lsb/pure/) creado por la Fundación de Ciencia y Tecnología de Portugal (Grant POSI/CHS/44304/2002), en el marco del cuál este trabajo se ha desarrollado (WP-4:*Polytypic Data Procesing*).

A Luis Quintas, mi director en el proyecto de investigación *P-319002: Decisión Artificial, Uso de Autómatas en Problemas de Decisión*, por su apoyo permanente y por permitirme desarrollar este trabajo en el proyecto investigación mencionado.

A los seres que amo, por el apoyo afectivo y emocional....

Polytypic Data Processing

Claudia Mónica Necco

Supervisor: Prof.Ph.D. José Nuno Fonseca de Oliveira

Submitted to the
Department of Computer Sciences
Faculty of Physics-Mathematics and Natural Sciences
of the National University of San Luis
in fulfillment of the requirements
for the degree of Master in Computer Sciences

San Luis - Argentina

December 2004

Abstract

Many aspects of *data processing* are functional in nature and can take advantage of recent developments in the area of *functional programming* and calculi.

The work described in this thesis is an attempt to contribute to this line of thought, in particular exploiting the Haskell functional language as support tool.

Haskell is used mainly to animate an abstract model of the relational database calculus as defined by Maier, written in the style of model-oriented formal specification.

The model uses an $Error$ monad for capturing errors. This monad (also called the $Exception$ monad) embodies the strategy for combining computations that can present exceptions by passing bound functions from the point an exception is found to the point where it is handled.

A collection of functions that capture some of the functionality currently provided by multidimensional database products are presented. In particular, functions that permit to classify and to reduce relations (tables) which, suitably combined, will permit to carry out the multidimensional analysis of a relational database.

Parametricity and genericity (polytypism) make room for further extensions of the model. Generic versions of relational standard (type-constructor parametric) and multidimensional analysis operations are expressed in Generic Haskell.

A theory of data normalization which is more general than the standard relational database theory (of which this appears to be a particular case) is suggested using higher-order polymorphism and constructor classes in Haskell 98.

Besides animation, the functional model is further subjected to formal reasoning and calculation, paving the way to the eventual polytypic (generic) formulation of the standard relational calculus.

Acknowledgments

First I want to thank all the people that has encouraged and helped me in this important part of my life.

My father, who back me up in all moments of my intellectual development with his faith and prayers.

To my Thesis Director, Professor Nuno Fonseca de Oliveira, for his continuous help and for sharing with me his deep knowledge of the subject. I owe him to have find again that particular sensation of emotion-pleasure-intellectual motivation-happiness that I have experienced when studying mathematics in elementary and high school. These sensations come not only from affinity with the subject under study, but also from the quality of the teacher. I am very proud of being one of his students.

Also through Prof. Oliveira I want to thank the University of Minho, in particular the Group of Logic and Formal Methods for their goodwill in assisting me during my stays at that University. Within this context, I also wish to thank the Research Project PURE: *Program Understanding and Re-engineering: Calculi and Applications* (http://www.di.uminho.pt/ lsb/pure/) funded by the Portuguese Science and Technology Foundation (Grant POSI/CHS/44304/2002) within which this work was carried out (work package WP-4: *Generic Data Processing*).

My deep appreciation also goes to Prof. Luis Quintas, Director of the Research Project *P-319002: Decisión Artificial, Uso de Autómatas en Problemas de Decisión* of the Universidad Nacional de San Luis within which I have developed this research work, for his continuous support and encouragement.

Finally, to all the persons that I love, for their continuous support.

Contents

Chapter 1

Introduction

1.1 Background

Productivity and scientific progress in the software development technology is often hindered by artificial, "application domain" border-lines which prevent cross-fertilization of results and the even spread of novelty.

Such frontiers or border-lines sometimes have an academic, social or cultural bias. For instance, the average database programmer will regard *functional programming* as too academic and perhaps useless. Conversely, many functional programmers will regard *database programming* as a too specific and not sufficiently exciting topic.

However, these two research areas have more in common than it appears at first sight. Both put an emphasis on the rôle of *data structuring* in software development and both have developed their own calculus. Can these two seemingly disparate notations and calculi be merged together? This question was the motivation of the present M.Sc. thesis.

The relational data model and associated calculus [Mai83] (originating from *e.g.* [Cod71b, Cod72a]), are today standard matters in computer science curriculæ. The functional programming addressed toward data structuring and calculation is less well–known. The widely accepted fact that *data precede algorithms* in software construction — known since the days of structured programming in the 1970s — has been the subject of recent research in the field of mathematics of program calculation. This has provided further insight into the role — either *real* or *virtual* — of data structuring in programming. A large database file stored in a disk is surely a visible, *real* data structure. However, the binary search tree which controls *quick-sort*'s double recursion is only evident to the software formalist who knows that real data structures may exist at specification level which disappear (*i.e.* become *virtual*) throughout the process of software refinement by calculation.

One of the most significant advances of the last decade has been the so-called *functorial* approach to data types which originated mainly from [MA86], was popularized by [Mal90] and reached the textbook format in [BdM97]. A comfortable basis for explaining *polymorphism* [Wad89], the "data types as functors" motto has proved ben-

eficial at a higher level of abstraction, giving birth to *polytypism* [JJ96], *i.e*. higher-order polymorphism.

Polymorphism and polytypism are steps of the same ladder, that of *generic programming* [Bo98]. The main target of this fast evolving discipline is to raise the level of abstraction of the programming discourse in a way such that seemingly disparate programming techniques, algorithms *etc*. are unified into idealized, kernel programming notions.

1.2 Context of Application

In (relational) database programming one models real-life facts as tuples which are recorded in large, mutually dependent persistent data-sets which are subjected to intensive search and processing in order to gather knowledge about a particular application domain. The need for larger and larger data-sets calls for the integration of disparate data-models (*data warehousing*); routine data inspection gives place to *data-mining*, and sophisticated on-line analytical processing (OLAP) replaces manual consolidation of data.

In the past, *textual data* has, by-and-large, been out of this data processing trend, due to its lack of structure and a too strong technology bias [1]. This has changed recently when, with the advent of the INTERNET, open mark-up textual standards eventually gained wide acceptance and world-wide prominence. Text processing has always been a privilege of the grammar theorist, the language analyst or compiler writer, as well as fertile ground for PERL and AWK scripting. However, how does one combine this with "flat" data mining and OLAP technologies, which we want to scale-up to arbitrarily structured textual documents? Is there room for a single, generic theory of data calculation able to cope with such *heterogeneous* data sources (*e.g*.text, tabular data, *etc*.)?

These questions call for the unification of (from the relational side) normalization, browsing, analytical processing with the universal-morphism approach which underlies the calculation theory of generic programming. For instance, the relational *join*/*unjoin* operators can be seen as relational instances of the polytypic functions zip and $unzip$, respectively ([JJ98] and [Oli98, NSO99]).

However heterogeneous a data source may happen to be, if it is "structured" this means that it has the shape of an inductive finite data structure (*e.g*. a finite set or list, a finitely branching tree or a combination of these two). Inductive data types are expressible in generic programming as fix-points of appropriate (regular) functors [BdM97]. So the main task appears to be that of generalizing the functorial constructs which describe relational database types to arbitrary regular functors and *see what happens*. Of course we have to broaden our view of functional programming to that of generic (polytypic) programming [Bo98].

Altogether, the following benefits are targets in the "going functional" approach to data processing:

[1] Think of the variety of text editors in existence today.

- Animation: development of new formal specification animation standards through the use of advanced functional languages such as Haskell
- Calculation: to exploit the functional programming calculus, in particular with respect to parametric polymorphism, "theorems for free" [Wad89], etc
- Genericity: higher-order polymorphism which makes for truly generic software models and solutions.
- Foundations: search for new foundations for the relational database calculus.

1.3 Related Work

Formal models of relational databases. The interest in formal models of the relational database systems is not new and dates back to (at least) the work of [BJ82], where a formalization of relational data model is given using the "Vienna Development Method" (VDM). The authors consider two possible representations of a *row (tuple)*: as a list of values and as a mapping from attribute names to values. They illustrate the differences of choosing one over the other by exemplifying both alternatives in the definition of the relational operations. The preconditions of the relational operations, required for syntactic well-formedness, do not include all conditions that should be satisfied by the relational operations. The specification of the relational data model does not include concepts like *key*, *integrity rule* and manipulative operations.

The formal specification and design of a program implementing simple update operations on a binary relational database called NDB is described in [Wal90]. This single level description of NDB is the starting point of [FJ90], where a case study in the modular structuring of this "flat" specification is presented. The authors present a second specification which makes use of an n-ary relation module, and a third one which uses an n-ary relation module with type and normalization constraints. They demonstrate the reusability of their modules, and also outline specifications of an n-ary relational database with normalization constraints, and an n-ary relational database with a two-level type hierarchy and no normalization constraints. However, their emphasis is on the modularization techniques adopted to organize VDM specifications into modules.

Samson and Wakelin [SW92] present a comprehensive survey about the use of algebraic methods to specify databases. They compare a number of approaches according to the features covered and enumerate some features not normally covered by such methods.

The purpose of Baluta [Bal95] is to define, in a rigorous and precise manner, the basic features of the relational data model version 2 (RM/V2) as defined by Codd [Cod90], using the Z specification language.

There have also been efforts to model *multidimensional databases*. In [AGS97], a model for multidimensional databases is introduced. The model provides for symmetric treatment of dimensions and measures. A set of minimal (but rather elaborate) operators is also introduced dealing with the construction and destruction of cubes, join and restriction of cubes and merging of cubes through direct dimensions.

In [LW96] a multidimensional data model is introduced based on relational elements. Dimensions are modeled as "dimension relations", in practice annotating attributes with dimension names. Cubes are modeled as functions from the Cartesian product of the dimensions to the measure and are mapped to "grouping relations" through an applicability definition. A grouping algebra is presented, extending existing relational operators and introducing new ones, such as ordering and grouping, to prepare cubes for aggregations. Furthermore, a multidimensional algebra is presented, dealing with the construction and modification of cubes as well as with aggregations and joins.

In [GL97] n-dimensional tables are defined and a relational mapping is provided through the notion of completion. An algebra and calculus for restructuring, classification and summarization is defined with classical relational operators as well as new operators. The expressive power of the algebra is demonstrated through the definition of operators such as the data cube operator and monotone roll-up.

Data processing by calculation The work described in this thesis finds its roots in [Oli01b], where abstract data-modeling is based on a *calculus* of data combinators which exhibit universal properties. These properties help in reasoning, transforming and calculating about them. Data-model calculation is shown to encompass not only data *refinement* but also data transformation, *data-mining* and *data-migration*.

Based on set-theory (and a modest use of category theory), reference [RO97] presents a constructive approach to relational database normalization theory. A set of laws which prevent from the violation of normal forms caused by partial dependencies, transitive dependencies and multi-valued dependencies are presented.

Boiten and Hoogendijk [BH95] sketch an approach to calculi for databases which provide multiple data types. They present a first layer of calculus with operations on basic values that correspond to the traditional tuples of relational databases. These operations are provided by a category with special products. The labels of the tuples are provided by the categorical typing. Then, in a second layer, these operations are lifted to operations on sets, resulting in the traditional relational algebra operators.

Haskell as a formal model animation tool Despite its expressive power and genericity, HASKELL has seldom been used mainly to animate abstract models in a explicit way. Mukherjee's translation of VDM specifications into GOFER is among the first experiments in the field [Muk97].

Edison [Oka00] is a library of functional data structures implemented Haskell. It supports three main families of abstractions: sequences, collections (e.g., sets and priority queues), and associative collections (e.g., finite maps).

HaskellDB [LM99] is a combinator library for expressing queries and other operations on relational databases in a type safe and declarative way. All the queries and operations are completely expressed within Haskell, that is to say, no embedded (SQL) commands are needed. At the time of writing, HaskellDB is not yet finished: documentation should be written and the implementation is not yet complete.

1.4 Structure of the Dissertation

This dissertation is organized as follows. Chapter 2 summarizes some basic concepts which are central to the thesis. Chapter 3 presents a relational data model expressed in Haskell notation. Chapter 4 extends the relational algebra with the OLAP operator mda and illustrates its expressive power via examples.

Chapters 5 and 6 are the core of this work. They show two different approaches to generic data processing. Chapter 7 discusses these two approaches to genericity and presents the main conclusions. It also points out directions for future research.

Chapter 2

Background

2.1 Formal Methods

A *formal method* for software development is a method that provides a *formal language* for describing a software artifact (for example, specifications, designs, or source code) such that *reasoning* and formal *proofs* are possible, in principle, about *properties* of the artifact so expressed. At the heart of a formal method there always is a formal language, that is, a set of strings over some well-defined alphabet. Rules are given for distinguishing strings in the language from those outside.

Formal methods aim at driving software production into good engineering standards by splitting software production into a *specification phase*, in which a mathematical model is built from contractor requirements, followed by an *implementation phase* in which such a model is somehow converted into a runnable software artifact. Formal methods research shows that implementations can be effectively *calculated* from specifications [Mor90, Oli90, Oli92]. So, in a sense, software technology is becoming a mature discipline in its adoption of the "universal problem solving" strategy which one is taught at school:

- understand the problem
- build a mathematical model of it
- reason in such a model
- upgrade it, wherever necessary
- calculate a final solution and implement it.

The sophistication of this strategy is only dependent on the underlying mathematics. In the context of software calculi, data manipulation is based on solving systems of (recursive) equations on domain spaces, up to isomorphism. This entails the definition of data transformations which can be expressed functionally and animated using a functional programming language such as HASKELL [Tho96].

2.1.1 Reasoning, Proofs and Verification

We consider validation as the process of increasing the confidence that a model is an accurate representation of the system under consideration.

One aspect of validation is checking the internal consistency of a model. If we use a formal modeling language, we can check the syntax and at least some semantics aspects with the aid of tools (c.f. *syntax checker* and *type checker* in a programming language compiler).

The other aspect of validation is checking the accuracy with which the model records the desired system behavior. One wants an implementation to behave exactly in the way prescribed by its specification. The relationship between specifications and implementations is one-to-many, that is, specifications are more *abstracts* than implementations. The "epistemological gap" between specifications and implementations is far from being a "smooth" one and is the major concern of the so-called *reification* (or refinement) technology, a recent branch of software engineering using formal methods.

In software engineering, we distinguish between *constructive* and *analytical means* of validation.

Constructive means to prevent the occurrence of errors in the process of development, *e.g.*, by the use of less error-prone programming languages (like Java instead of C++) or through automated generation of implementations from high-level models. In the well-known *constructive style* for software development [Jon80, Jon86] design is factored into as many "mind-sized" design steps as required. Every intermediate design is first proposed and then proved to follow from its antecedent. Despite improving the primitive approach to correctness (full implementation prior to the overall correctness argument), such an "invent-and-verify" style is often impractical due to the complexity of the mathematical reasoning involved in real-life software problems.

Recent research seems to point at alternatives reification styles. The idea is to develop a *calculus* allowing *programs* to be actually calculated from their specifications. In this approach, an intermediate design is drawn from a previous design according to some *law* available in the calculus, which must be *structural* in order for the components of an expression to be refined in isolation (*i.e.*pre-existing refinement result can be "re-used"). Proof discharge is achieved by performing structural calculation instead of proof from first principles. This is the point of a calculus, as witnessed elsewhere in the past (cf. the differential and integral calculi, linear algebra, *etc.*).

Analytical means are used to detect errors in models or implementations. There are a number of analyses that can be performed in order to increase our confidence that the model is complete and captures the key properties needed by the proposed system. Three known approaches to do that are:

- **Animating** the model- *works well with clients unfamiliar with the modeling notation but requires a good interface*.
- **Testing** the model- *can asses coverage but limited to the quality of the tests and the model must be executable*.
- **Proving** properties of the model - *provides excellent coverage and does not require executability, but is not well supported by tools*.

2.1.2 Model Animation and Support Tools

Declarative languages, *e.g.*the functional language HASKELL [Jon03], state *what* is to be computed in a form that is largely independent of *how* the computation is performed. Declarative languages are based on sound mathematical foundations, have well-defined semantics, permit descriptions at a very high level of abstraction, and are referentially transparent. Thus declarative languages are specially suitable as specification languages, they add the following advantage: an executable specification represents not only a conceptual, but also a behavioral model of the software system to be implemented. The behavior of the system interacting with its environment can be demonstrated and observed before it actually exists in its final form.

Furthermore, executable specifications can serve as prototypes which allow to experiment with different requirements, or to use an evolutionary approach for software development.

These considerations bring us to the section which follows.

2.2 Functional Programming

Almost two decades ago John Backus read, in his Turing Award Lecture, a revolutionary paper [Bac78] proclaiming conventional command-oriented programming languages obsolete because of their inefficiency arising from retaining, at a high-level, the so-called "memory access bottleneck" of the underlying computation model — the well-known *Von Neumann* architecture. Alternatively, the (at the time already mature) *functional programming* style was put forward for two main reasons. First, because of its potential for concurrent and parallel computation. Second, and Backus emphasis was really put on this, because of its strong algebraic basis.

Backus *algebra of (functional) programs* was providential in alerting computer programmers that computer languages alone are insufficient, and that only languages which exhibit an *algebra* for reasoning about the objects they purport to describe will be useful in the long run.

The impact of Backus first argument in the computing science and computer architecture communities was considerable, in particular if assessed in quality rather than quantity and in addition to the almost contemporary *structured programming* trend [1].

By contrast, his second argument for changing computer programming was by and large ignored, and only the so-called *algebra of programming* research minorities pursued in this direction. However, the advances in this area throughout the last two decades are impressive and can be fully appreciated by reading a textbook written relatively recently by Bird and de Moor [BdM97].

Functional programming literally means "programming with functions". Programming languages such as LISP [Gra96] or HASKELL [Jon03] allow us to program with functions. However, the functional intuition is far more reaching than producing code which runs on a computer. The idea of producing programs by *calculation*, that is to

[1] Even the C programming language and the UNIX operating system, with their implicit functional flavor, may be regarded as subtle outcomes of the "going functional" trend.

say, that of calculating efficient programs out of abstract, inefficient ones has a long tradition in functional programming [BD77].

Functional programming can moreover support the use of formal methods in developing executable specifications and prototype implementations. In this thesis the HASKELL functional programming language will be used for both purposes, animation and calculation. This calls for an introduction to the theory of functional programming in general, and to the syntax of HASKELL in particular.

The following sections provide such as an introduction in a light-weight style. The main emphasis is on explaining how to construct new functions out of other functions using a minimal set of predefined functional *combinators*. This leads to a programming style which is *point free* in the sense that functions descriptions dispense with variables ("definition points"). Examples will be provided in concrete HASKELL syntax.

2.2.1 Introducing Functions and Types

The definition of a function

$$f : A \longrightarrow B$$

can be regarded as a kind of "process" abstraction: it is a "black box" which produces an output once it is supplied with an input:

$$x(\in A) \rightarrow f \rightarrow fx(\in B)$$

From another viewpoint, f can be regarded as a kind of "contract": it commits itself to producing a B-value provided it is supplied with an A-value. How is such a value produced? In many situations one wishes to ignore it because one is just *using* function f. In others, however, one may want to inspect the internals of the "black box" in order to know the function's *computation rule*. For instance,

$$\begin{array}{rcl} succ & : & \mathbb{N} \rightarrow \mathbb{N} \\ succ\, n & \stackrel{\text{def}}{=} & n+1 \end{array}$$

expresses the computation rule of the *successor* function — the function $succ$ which finds "the next natural number" — in terms of natural number addition and of natural number 1. What was above meant by a "contract" corresponds to the *signature* of the function, which is expressed by arrow $\mathbb{N} \rightarrow \mathbb{N}$ in the case of $succ$ and which, by the way, can be shared by other functions, *e.g.* $sq\, n \stackrel{\text{def}}{=} n^2$.

In programming terminology one says that $succ$ and sq have the same "type". Types play a prominent rôle in functional programming (as they do in other programming paradigms). Informally, they provide the "glue", or interfacing material, for putting functions together to obtain more complex functions. Formally, a "type checking" discipline can be expressed in terms of compositional rules which check for functional expression well-formedness.

It has become standard to use arrows to denote functions signatures or function types. In this work the following variants will be used interchangeably to denote the fact that the function f accepts arguments of type A and produces results of type B:

$f : A \longrightarrow B$, $A \xrightarrow{f} B$. This corresponds to writing `f :: a -> b` in the HASKELL functional programming language, where type variables are denoted by lowercase letters. A will be referred to as the domain of f and B will be referred to as the co-domain of f. Both A and B are symbols which denote sets of values, very often called types. In others words, A and B are the source and target types associated with a function f.

The ability to introduce new data types and to define functions that manipulate their values is the essence of functional programming. Data types can be introduced by simple enumeration of their elements; for example:

```
data Bool = False | True
data Char = Ascii0 | Ascii1 |  ... |  Ascii127
```

The type $Bool$ consist of two values and $Char$ consist of 128. The various identifiers, $Ascii0, True$, and so on, are called *constructors* and the vertical bar "|" is interpreted as the operation of disjoint union. Thus, distinct constructors are associated with distinct values.

Data types can be defined in terms of other data types; for example:

```
data BoolOrChar = Bool Bool | Char Char
data  Both   = Tuple (Bool, Char)
```

The type $BoolOrChar$ consists of 130 values: $Bool\ False, Bool\ True, Char\ Ascii0$, and so on. The type $Both$ consists of 256 values, one for each combination of a value in $Bool$ with a value in $Char$. In these data types the constructors $Bool$, $Char$ and $Tuple$ denote functions; for example, $Char$ produces a value of type $BoolOrChar$ given a value of type $Char$.

Given the assurance about different constructors producing different values, we can define functions on data types by pattern matching; for example,

$$\begin{aligned} not\ False &= True \\ not\ True &= False \end{aligned}$$

defines the negation operator $not : Bool \rightarrow Bool$, and

$$switch(Tuple(b, c)) = Tuple(not\ b, c)$$

defines a function $switch : Both \rightarrow Both$.

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

$$\begin{aligned} and(False, b) &= False \\ and(True, b) &= b \end{aligned}$$

or by "currying", as in

$$\begin{aligned} cand\ False\ b &= False \\ cand\ True\ b &= b \end{aligned}$$

The difference between and and $cand$ is just one of type:

$$\begin{aligned} and &: (Bool \times Bool) \to Bool \\ cand &: Bool \to (Bool \to Bool) \end{aligned}$$

More generally, a function f of two arguments can be defined by choosing any of the types

$$\begin{aligned} f &: (B \times C) \to A \\ f &: C \to (B \to A) \\ f &: B \to (C \to A) \end{aligned}$$

With the first type we would write $f(b, c)$; with the second, $f\ c\ b$; and with the third, $f\ b\ c$. For obvious reasons, the first and third seem more natural companions. The function $curry$, with type:

$$curry \quad : \quad ((B \times C) \to A) \to (B \to (C \to A))$$

converts a non-curried function into a curried one:

$$curry\ \ f\ \ b\ \ c \quad = \quad f(b, c)$$

One can also define a function $uncurry$ that goes the other way round. Functional programmers prefer to curry their functions as a matter of course, one reason being that it usually leads to fewer brackets.

Returning to data types, they can be parameterised by other data types; for example, the definition

$$Maybe\ a = Nothing \mid Just\ a$$

introduces a type $Maybe\ a$ in terms of a type parameter a. For example, $Just\ True$ has type $Maybe\ Bool$, while $Just\ Ascii0$ has type $Maybe\ Char$. Another example of a type constructor in HASKELL is the finite-list type -constructor [ ].

Data types can also be defined recursively; for example,

$$Nat = Zero \mid Succ\ Nat$$

introduces the type of natural numbers. Nat is the union of an infinite number of distinct values: *Zero, Succ Zero, Succ (Succ Zero)*, and so on.

In the sequel, it will be useful to know the HASKELL syntax for algebraic data type declaration. In general, this has the form [Jon03]:

$$data\ cx \quad \Rightarrow \quad T\ u_1 \ldots u_k \quad = \quad K_1\ t_{11} \ldots t_{1k_1} \mid \ldots \mid K_n t_{n1} \ldots t_{k_n}$$

where cx is the so-called *context*. A *context* consists of zero or more class assertions, and has the general form $(C_1 u_1, ...C_n u_n)$, where $C - 1, ..., C_n$ are class identifiers, and each of the $u_1, ..., u_n$ is either a type variable or the application of type variable to one or more types. We write $cx => t$ to indicate the type t restricted by the context cx. The context cx must only contain type variables referenced in t.

This declaration introduces a new *type constructor* T with one or more constituent *data constructors* $K_1 \ldots K_n$. The types of the data constructors are given by

$$K_1 :: \forall u_1 \ldots u_k.cx_i \quad \Rightarrow \quad t_{i1} \rightarrow \ldots \rightarrow t_{ik_i} \rightarrow (T\ u_1 \ldots u_k)$$

where cx_i is the largest subset of cx constraining only those type variables free in the types $t_{i1} \ldots tik_i$. The type variables u_1 through u_k must be distinct and may appear in both cx and the t_{ij}. It is a static error for any other type variable to appear in cx or on the right-hand-side. The new type constant T has a *kind* [2] of the form $\kappa_1 \rightarrow \ldots \rightarrow \kappa_k \rightarrow *$, where the kinds κ_i of the argument variables u_i are determined by kind inference. This means that T may be used in type expressions with anywhere between 0 and k arguments.

2.3 Haskell 98

Haskell is a non-strict, pure functional language, named after the mathematician Haskell Brooks Curry [3]. Haskell is a member of the *ML family of languages*. Other, more-or-less similar languages in this family are Standard ML [Mil84], Hope and Miranda [4] [Tur85], which are some of the most immediate predecessors of Haskell. Haskell has been developed with the purpose to serve as a vehicle for teaching, research and application of the functional programming paradigm. All languages in the ML-family incorporate features such as *pattern matching*, a *strong polymorphic type system* and *higher order functions*. Haskell has evolved continuously since it was introduced in 1987. At the 1997 Haskell workshop in Amsterdam, it was decided that a stable variant of Haskell was needed. As a result of this, the latest version of the language was released in February 1999 and it is called "Haskell 98" [Jon03].

Hugs is a portable Haskell interpreter written in C which runs on almost any machine. Hugs is best used as a Haskell program development system: it boasts extremely fast compilation, supports incremental compilation, and has the convenience of an interactive interpreter (within which one can move from module to module to test different portions of a program). However, being an interpreter, it does not nearly match the run-time performance of, for example, GHC, nhc98, or HBC (see `http://www.haskell.org/libraries/`). It is certainly the best system for newcomers to learn Haskell. Hugs 98 is in conformity with Haskell 98 and it is available for all Unix platforms including Linux, DOS, Windows 3.x, and Win 32 (Windows 95, Win32s, NT) and Macintosh. It has many libraries including Win32 libraries, a foreign interface mechanism to facilitate inter-operability with C and the Windows version has a simple graphical user interface.

[2] Type expressions are classified into different *Kinds*, which take one of two possible forms:

- The symbol $*$ represents the kind of a nullary type constructors.
- If κ_1 and κ_2 are kinds, then $\kappa_1 \rightarrow \kappa_2$ is the kind of types that take a type of kind κ_1 and return a type of kind κ_2.

[3] Haskell B. Curry (1900-1982) was a pioneer of modern mathematical logic. His research led to, among others results, the development of combinatory logic.

[4] Miranda is a trademark of Research Software Ltd.

2.3.1 Monads

An advantage of the pure functional programming style is the absence of side-effects. But in many cases it could be very laborious to have all such effects (inc. I/O) defined in an explicit way. One possible way is to let a global state represent the world outside of the program, which is passed and updated as the computations proceed. But – again — passing this state around can make the "main" computation indistinct and difficult to follow. HASKELL provides a way of dealing with side-effects by use of the mathematical concept of a *monad* [Wad92].

A monad is a concept which arises from category theory. In a computational context, a monad represents a computation: if m is a monad identifier, then a object of type ma represents a computation which produces a result of type a. Monads are incorporated in HASKELL by means of a so-called `Monad` class.

Haskell uses a traditional Hindley-Milner polymorphic type system to provide a static type semantics, but the type system has been extended with *type classes* that provide a structured way to introduce *overloaded* functions.

We can use adjectives *ad-hoc* an *parametric* to distinguish two varieties of *polymorphism*.

Ad-hoc polymorphism occurs when a function is defined over several different types, acting in a different way for each type. A typical example is overloaded multiplication: the same symbol may be used to denote multiplication of integers and multiplication of floating point values.

Parametric polymorphism occurs when a function is defined over a range of types, acting in the same way for each type. A typical example is the length function, which acts in the same way on a list of integers and a list of floating point numbers.

A `class` declaration introduces a new *type class* and the overloaded operations that must be supported by any type that is an instance of that class [5].

An `instance` declaration declares that a type is an instance of a class and includes the definitions of the overloaded operations —called *class methods*— instanced on the named type.

Using the Haskell type class system, we can define functions that be:

- **polymorphic**: its use is not restricted to values of any single type.
- **overloaded**: its interpretation is determined by the types of its arguments.
- **extendible**: the definition of well-formation can be extended to include new data-types.

The class monad in Haskell is defined:

```
class Monad m where
   (>>=)  :: m a -> (a -> m b) -> m b
   (>>)   :: m a -> m b -> m b
   return :: a -> m a
```

[5] Haskell type class is not a type, types are separate, they "join" classes.

This class is an example of a *constructor class*, which is a class whose members are *type constructors*. As we have seen above, type constructors are functions which build types from types. The $(>>=)$ ($bind$) operation combines computations when a value is passed from one computation to another. $(>>)$ is defined in terms of $(>>=)$ to combine two computations when the second computation does not need the result of the first one. The $return$ operation takes a value a and returns the corresponding "monadic" value of type $m\ a$. A simple example of these operations is the following I/O program which reads a line of input, parses the string and returns the value as a " monadic" integer:

```
getInt :: IO Int
getInt =
   getLine >>= \line ->
   return (read line::Int)
```

Function `getLine` reads a line from `stdin`, the $(>>=)$ operator sequences the interaction by passing the result of `getLine` to a function $\lambda x.e$. In the example, the parameter of this function is `line`, which is expecting an argument of type `String` and whose body is the application of the `return` function. The `read` function parses `line` into an integer, which is then returned as a monadic integer.

Monadic expressions can adopt the so-called `do`-syntax, which provides a more conventional syntax for monadic programming. It allows an expression such as in the program above to be written as:

```
getInt :: IO Int
getInt = do
   line <- getLine
   return (read line::Int)
```

The differences between the two programs are just *syntactic sugar*.

2.3.2 Point free Notation

There are two basic styles for expressing functions: the *point wise* style and the *point free* style. In the point wise style one describes a function by describing its application to arguments. By contrast, in the point free style one describes a function exclusively in terms of functional *combinators*, among which functional *composition* is a basic one.

Some people will argue that such compact "point free" notation (that is, the notation which hides variables, or function "definition points") is too cryptic to be useful as a practical programming medium. In fact, point free programming languages such as Iverson's APL or Backus' FP have been more respected than loved by the programmers community [Oli99]. Virtually all commercial programming languages require variables and so they implement the more traditional "point wise" notation.

The main purpose of the point free notation is to make reasoning easier to perform [6]. Chapter 6 will serve to illustrate how a point-free style leads to a very simple method

[6] The move from the *point wise* level (involving operators as well as variable symbols, logical connectives, quantifiers, *etc*.) to the *point free* one is compared elsewhere [Oli01a] to the *Laplace transformation*. The former is more intuitive but harder to reason about, the latter is less descriptive but more algebraic and compact. As in traditional mathematics, there is room for both in formal specification.

for reasoning about functions.

2.4 Generic Programming

The ability to name and reuse common patterns of computation as *higher-order* functions is at the heart of the power of functional languages. Higher-order functions like *maps* and *catamorphisms* (see [BdM97]) capture very general programming idioms that are useful in many contexts. This kind of *polymorphic* functions enable us to abstract away from the unimportant details of an algorithm and concentrate on its essential structure.

The type of a polymorphic function has *type parameters*, but all monomorphic instances of the function can use identical code. A generalization is to parameterize also the functional definition on types. Functions that are parameterized in this way are called *polytypic functions* [Jeu95]. Equality functions, pretty printers and parsers, traversals functions and other recursion combinators are all examples of polytypic functions.

While a normal polymorphic function is an algorithm that is independent of the type parameters, the class of instances of a polytypic function contains functions that are different, but which share a common structure. Any algorithm in the class can be obtained by instancing a template algorithm with (the structure of) a data type. Other terms used denoting polytypism in the literature are *structural polymorphism* [Rue92], *generic programming* [RBH96], *type parametric programming* [SN95], *shape polymorphism* [Jay95b], *polynomial polymorphism* [Jay95a] and *type-indexed functions* [Hin99].

Polytypic or generic programming offers a number of benefits:

Reusability: Polytypism extends the power of polymorphic functions to allow classes of related algorithms to be described in one single definition. For example, the class of printing functions for different data types can be expressed as one polytypic function — cf. eg. the *show* function in HASKELL. Thus polytypic functions are very well suited for building program libraries.

Adaptability: Polytypic programs automatically adapt to changing data types. This adaptability reduces the need for time consuming and boring rewrites of trivial functions and reduces the associated risk of making mistakes.

Closure and orthogonality: Currently some polytypic functions can be *used* but not *defined* in some functional languages (for instance, in ML [7] [Mil84] , the equality function(s)).

This asymmetry can be removed by extending these languages with polytypic definitions.

[7] ML has introduced the notion of parametric polymorphism in languages. ML types contain type variables which are instanced to different types in different contexts. Hence it is possible to partially specify type information and to write programs based on partially specified types which can be used on all the instances of those types.

Applications: Some problems are polytypic by nature: maps and traversals, pretty printing and parsing, data compression, matching, unification, term rewriting, etc.

Probability: The more general a function, the more general proofs are about it. In a polytypic context, each of the earlier benefits acquires an additional interpretation: one gets reusable proofs, adaptable proofs, less *ad hoc* semantics of programming languages and new proofs of properties of printing and parsing, packing, term rewriting, and so on.

There are various ways to implement polytypic programs in a typed language, in particular:

- using a universal data type;
- using higher-order polymorphism and constructor classes;
- using a special syntactic construct.

Polytypic functions can be implemented by defining a universal data type, on which we define the functions we want to have available for large classes of data types. These polytypic functions can be used on a specific data type if we provide translation functions to and from the universal data type. An advantage of using a universal data type for implementing polytypic functions is that we do not need a language extension for writing polyytpic programs. However, using universal data types has several disadvantages: type information is lost in the translation phase to the universal data type, and type errors can occur when programs are run. Furthermore, different people will use different universal data types, which will make program reuse more difficult.

Should higher-order polymorphism and constructor classes be used for defining polytypic functions, then type information is preserved, and we can use a functional language such as HASKELL for implementing polytypic functions. In this style, all regular data types [Hin00a] are represented by type

```
data Mu f a = In ( f a (Mu f a))
```

and the class system is used to overload functions like map and $cata$. However, writing such programs is rather cumbersome: programs become cluttered with instance declarations, and type declarations become cluttered with contexts. And the user still has to write all translation functions.

The third solution, to extend the language with a syntactic construct for defining polytypic functions is based on the initial algebra semantics of data types (see details in eg. [Bac95].) Examples of this approach are the POLYP system [Jan00], Generic H∀SKELL [HJ03], derivable type classes [HJ00], etc.

In this thesis, genericity will be approached in two different ways. We will experiment with Generic H∀SKELL in chapter 5 , and chapter 6 will exploit the use of higher-order polymorphism and constructor classes.

2.5 The Relational Data Model

The first theoretical contributions to the so-called *relational data model*, were published in the early seventies, among which the contributions of Codd [Cod70, Cod71a, Cod71b, Cod72a] were outstanding.

The information system community is indebted to Codd for his pioneering work on the very foundation of the *relational data model* theory [Cod70]. Relational algebra was defined in [Cod72b], which includes operators from set theory and specific operators from the relational model. (Some of the relational operators were already introduced in [Cod70].) the concept of a *functional dependency* is already present in [Cod70], in the form of keys. Functional dependencies that do not follow from keys were introduced in [Cod72a] , for the purpose of *normalization*, that is, of information redundancy elimination. In [Cod71b, Cod72a] the so-called second and third *normal forms* of relational databases were introduced. Finally, the so-called *Boyce-Codd normal form* was introduced in [Cod74].

In 1983 a book by Maier [Mai83] was published which became a standard in the literature on the mathematics of relational database design.

In the following sections we will review the main concepts of the relational data model. Our main reference will be the textbook of David Maier [Mai83]

2.5.1 Standard Relational Algebra

A *relation scheme R* is a finite set of *attribute names* $A_1, A_2, ..., A_n$. Corresponding to each attribute name A_i is a set $D_i, 1 \leq i \leq n$, called the *domain* of A_i by *dom*(A_i). Attribute names are sometimes called *attribute symbols* or simply *attributes*, particularly in the abstract. The domains are arbitrary, non-empty sets, finite or countably infinite. A relation r on scheme R is written $r(R)$ or $r(A_1, A_2, ..., A_n)$.

Definition 1 (Relation) *Given $D = D_1 \cup D_2 \cup ... \cup D_n$, a* **relation** *$r$ on relation sche-me R is a finite set of mappings $\{t_1, t_2, ..., t_p\}$ from R to D with the restriction that for each mapping $t \in r, t(A_i)$ must be in D_i, $1 \leq i \leq n$. These mappings are called* tuples.

Definition 2 (key, superkey) *A* **key** *of a relation $r(R)$ is a subset K=$\{B_1, B_2, ...B_m\}$ of R such that for any distinct tuples t_1 and t_2 in r, there is a $B \in K$ such that $t_1(B) \neq t_2(B)$. That is, no two tuples have the same value on all attributes in K. We could write this condition as $t_1(k) \neq t_2(k)$ and no proper subset k' of K shares this property. K is a* **super key** *of r if K contains a key of r.*

Two relations on the same scheme can be considered sets over the same universe, the set of all possible tuples on the relation scheme. Thus, Boolean operations can be applied to two such relations.

Definition 3 (Boolean operators) *If r and s are relations on the scheme R, then $r \cap s$ is the relation $q(R)$ containing all tuples that are in both r and s, $r \cup s$ is the relation $q(R)$ containing all tuples that are in either r or s and $r - s$ is the relation $q(R)$ containing those tuples that are in r but not in s.*

Definition 4 (Complement) *Let $dom(R)$ be the set of all tuples over attributes of R and their domains. The* **complement** *of a relation $r(R)$ is defined as $\overline{r} = dom(R) - r$ (it can be infinite).*

Definition 5 (Active domain) *If $r(A_1A_2...A_n)$ is a relation and $D_i = dom(A_i), 1 \leq i \leq n$, the* **active domain** *of A_i relative to r is the set*

$$adom(A_i, r) \stackrel{\text{def}}{=} \{d \in D_i \mid \exists t \in r : t(A_i) = d\}$$

Definition 6 (Active complement) *Let $adom(R, r)$ be the set of all tuples over the attributes of R and their active domains relative to r. The* **active complement** *of r is $\tilde{r} = adom(R, r) - r$.*

Definition 7 (Select operator) *Let r be a relation on scheme R, A an attribute in R, and a an element of $dom(A)$. Using mapping notation, $\sigma_{A=a}(r)$ ("***select** *A equal to a on r") is the relation*

$$r'(R) \stackrel{\text{def}}{=} \{t \in r \mid t(A) = a\}$$

Definition 8 (Project operator) *Let r be a relation on scheme R, and let X be a subset of R. The* **projection** *of r onto X, written $\pi_X(r)$, is the relation $r'(X)$ obtained by striking out columns corresponding to attributes in $R - X$ and removing duplicate tuples in what remains. In mapping notation, $\pi_X(r)$ is the relation*

$$r'(X) \stackrel{\text{def}}{=} \{t(X) \mid t \in r\}$$

Definition 9 (Natural join operator) *Let r be a relation on scheme R, and let s be a relation on scheme S, with $RS = T$. The* **join** *of r and s written $r \bowtie s$, is the relation $q(T)$ of all tuples t over T such that there are tuples $t_r \in r$ and $t_s \in s$ with $t_r = t(R)$ and $t_s = t(S)$. Since $R \cap S$ is a subset of both R and S, as a consequence of the definition $t_r(R \cap S) = t_s(R \cap S)$. Thus, every tuple in q is a combination of a tuple from r and a tuple from s with equal $(R \cap S)$-values.*

Definition 10 (Divide operator) *Let $r(R)$ and $s(S)$ be relations, with $S \subseteq R$. Let $R' = R - S$. Then r* **divided** *by s, written $r \div s$, is the relation*

$$r'(R') \stackrel{\text{def}}{=} \{t \mid \forall t_s \in s : \exists t_r \in r : t_r(R') = t \ \wedge \ t_r(S) = t_s\}$$

Definition 11 (Renaming operator) *Let r be a relation on scheme R, where A is an attribute in R and B is an attribute not in $R - A$. Let $R' = (R - A)B$. Then r with A* **renamed** *to B, denoted $\delta_{A \leftarrow B}(r)$, is the relation*

$$r'(R') \stackrel{\text{def}}{=} \{t' \mid \exists t \in r : t'(R - A) \ = \ t(R - A) \wedge t'(B) = t(A)\}$$

2.5.2 Functional Dependences and Normalization

Relational database design starts from the so-called *database relation scheme* definition. Since relational databases are collections of n-ary relations, such a scheme, consisting in a set of *attribute names*, records the common format of all the elements (tuples) of each individual relation.

Concerning each relation scheme, there may be explicitly listed subsets of its attribute name set (the so-called "designated keys") which uniquely identify the tuples of relations with that scheme, that is, which exhibit the "property of key".

Designated keys with a minimum number of attributes are simply called *keys*. By contrast, sets of attributes containing more than one key are called *super keys* [8]. Designated keys are a way of prescribing *semantics* to relational data in the sense that they impose restrictions to the admissible relations with a certain scheme. So-called *functional dependencies* are generalizations of (designated) keys. The formal definition of a *functional dependency* follows.

Definition 12 (Functional dependency) *Let r be a relation on scheme R, with X and Y subsets of R. Relation r satisfies the* **functional dependency (FD)** $X \rightarrow Y$ *if for every X-value x, $\pi_Y(\sigma_{X=x}(r))$ as at most one tuple.*

One way to interpret this expression is to look at two tuples, t_1 and t_2, in r. If $t_1(X) = t_2(X)$, then $t_1(Y) = t_2(Y)$.

In relational database design practice, starting relation schemes tend often to be *too large, unstructured* and affected by functional dependencies. The goal of *normalization* is to find a set of relations schemes –also called a *database scheme* which is free of information redundancy (*i.e.*"is normalized") and free of explicit functional dependencies, and yet able to represent the same information as the initial relation scheme.

The first step of standard normalization consist of eliminating the structure of the relation scheme, *i.e.*of decomposing attributes with structured domain. Each time this first step is applied, the relevant relation scheme is decomposed into two or more relation schemes. The result is a relation scheme which is said to be in the *first normal form* (1NF). Normalization proceeds via one of two alternative procedures: normalization through decomposition or normalization through *synthesis*. The first option is chosen more often. Proceeding by relation schema decomposition the aim is to eliminate "undesirable" properties, namely those which contradict the aphorism: *every attribute which does not occur in the relation scheme key should depend on the key, the whole key and nothing but the key*. The second step consists of eliminating partial dependencies on the key in order to attain the so-called *second normal form* (2NF).

Finally, the third step eliminates transitive dependencies on the key. Transitive dependencies often correspond to functional dependencies involving attributes not occurring in the key. By eliminating them one obtains a database scheme which is said to be in the so-called *third normal form* (3NF), usually referred to simply as *normalized form*.

The notion of decomposition was explained so far admitting that the relation scheme has a single key. In the general case, one has to admit the possibility of many

[8] Super keys verify the property of key but their number of attributes can be reduced without affecting that property.

different keys. The two later normalization steps above are easily generalized to this case, by introducing the concept of a *non prime* attribute, that is, an attribute that does not occur in any key.

Normalization via decomposition has some well-known inconveniences, namely: it does enforce the absence of explicit functional dependencies, and it does not enforce a database scheme with a minimum number of relation schemes.

The so-called *synthesis algorithm* [Mai83] is free of these problems. However, known objections to its practical usefulness are: it is less intuitive, and it is not gradual. There are further normal forms which, in practice, are not so often considered in database design. Such is the case of the so-called *Boyce-Codd normal form* (4FN). The latter involves another type of dependencies: *multi-valued dependencies*.

2.6 On Line Analytical Processing

On-Line Analytical Processing (OLAP) is a category of software technology that enables live *ad hoc* data access and analysis. While the more familiar On-Line transaction Processing (OLTP) generally relies solely on relational databases, OLAP has become synonymous with *multidimensional* views of business data. These multidimensional views provide the technical basis for the calculation and analysis required by Business Intelligence applications.

OLTP applications are characterized by many users creating, updating, or retrieving individual records. Therefore, OLTP databases are optimized for transaction updating. OLAP applications are used by analysts and managers who frequently want a higher-level aggregated view of the data, such as total sales by product line, by region, and so forth. The OLAP database is usually updated in batch, often from multiple sources, and provides a powerful analytical back-end to multiple user applications. Hence, OLAP databases are optimized for analysis.

OLAP functionality is characterized by dynamic multi-dimensional analysis of consolidated enterprise data supporting end user analytical and navigational activities.

Two are the basic architectures for storing data in an OLAP database: ROLAP and MOLAP. ROLAP (Relational OLAP) [CD97] is based on a relational database server, extended with capabilities such as extended aggregation and partitioning of data [GCB+97]. The scheme of the database can be a star, snowflake, or fact constellation scheme [CD97]. On the other hand, MOLAP (Multidimensional OLAP) is based on ”pure” Multidimensional Databases (MDDs), which logically store data in multidimensional arrays, which are heavily compressed and indexed, in the physical level, for space and performance reasons.

OLAP is implemented in a multi-user client/server mode. An OLAP server is a high-capacity, multi-user data manipulation engine specifically designed to support and operate on multi-dimensional data structures. A multi-dimensional structure is arranged so that every data item is located and accessed based on the intersection of the dimension members which define that item. The design of the server and the structure of the data are optimized for rapid ad-hoc information retrieval in any orientation, as well as for fast, flexible calculation and transformation of raw data based on formulaic relationships.The OLAP server may either physically stage the processed multi-

dimensional information to deliver consistent and rapid response times to end users, or it may populate its data structures in real-time from relational or other databases, or offer a choice of both.

2.6.1 Basic Relational OLAP Operations

The multidimensional view of data considers that information is stored in a multi-dimensional array (sometimes called a Hyper cube, or Cube). A Cube is a group of data cells arranged by the dimensions of the data [Cou97]. A dimension is defined in [Cou97] as "a structural attribute of a cube that is a list of members, all of which are of a similar type in the user's perception of the data". Each dimension has an associated hierarchy of levels of aggregated data i.e. it can be viewed from different levels of detail (for example, Time can be detailed as Year, Month, Week, or Day). Measures (which are also known as variables, metrics, or facts) represent the real measured values [ABNO97].

Navigation is a term used to describe the processes employed by users to explore a cube interactively, by manipulating the multidimensionally viewed data [ABNO97], [Cou97]. Possible operations which can be applied are: Aggregation (or Consolidation, or Roll-up) which corresponds to summarization of data for the higher level of a hierarchy, Roll Down (or Drill down, or Drill through) which allows for navigation among levels of data ranging from higher level summary (up) to lower level summary or detailed data (down), Selection (or Screening, or Filtering or Dicing) whereby a criterion is evaluated against the data or members of a dimension in order to restrict the set of retrieved data, Slicing which allows for the selection of all data satisfying a condition along a particular dimension and Pivoting (or Rotation) throughout which one can change of the dimensional orientation of the cube, e.g. swapping the rows and columns, or moving one of the row dimensions into the column dimension, etc. ([ABNO97], [Cou97]).

Relational and SQL data extraction How do traditional relational databases fit into this multi–dimensional data analysis picture?. How can 2D flat files (SQL tables) model an N–dimensional problem?. Furthermore, how do the relational systems support operations over N–dimensional representations that are central to visualization and data analysis programs?

The answer to the first question is that relational systems model N–dimensional data as a relation with N–attribute domains. For example, 4–dimensional (4D) earth temperature data is typically represented by a `Weather` table (table 2.1). The first four columns represent the four dimensions: latitude,longitude, altitude and time. Additional columns represent measurements at the 4D points such as temperature, pressure, humidity, and wind velocity. Each individual weather measurement is recorded as a new row in this table. Often these measured values are aggregates over time (the hour) or space (a measurement area centered on the point).

Visualization and data analysis tools extensively use dimensional reduction (aggregation) for better comprehensibility. Often data along the other dimensions that are not included in a "2–D" representation are summarized via aggregation in the form of his-

Weather					
Time(UCT)	Latitude	Longitude	Altitude(m)	Temp. (c)	Pres. (mb)
96/6/1:1500	37:58:33N	122:45:28W	102	21	1009
. . . . .					
96/6/7:1500	34:16:18N	27:05:55W	10	23	1024

Table 2.1: `Weather Table.` Example of 4–dimensional (4D) earth temperature data

togram, cross–tabulation, subtotals, etc. In the SQL Standard, we depend on aggregate functions and the `GROUP BY` operator to support aggregation.

The SQL Standard (*IS 9075 International Standard for Database Language SQL, 1992*) provides five functions to aggregate the values in a table: `COUNT(), SUM(), MIN(), MAX(),` and `AVG()`. For example, the average of all measured temperatures is expressed as:

```
SELECT   AVG(Temp)
FROM     Weather;
```

In addition, SQL allows aggregation over distinct values. The following query counts the distinct number of reporting times in the `Weather` table:

```
SELECT   COUNT(DISTINCT Time)
FROM     Weather;
```

Aggregate functions return a single value. Using the `GROUP BY` construct, SQL can also create a table of many aggregate values indexed by a set of attributes. For example, the following query reports the average temperature for each reporting time and altitude:

```
SELECT   Time, Altitude, AVG(Temp)
FROM     Weather
GROUP BY Time, Altitude;
```

`GROUP BY` is an unusual relational operator: It partitions the relation into disjoint tuple sets and then aggregates over each set.

Chapter 3

The Relational Data Model in Haskell

The main purpose of this chapter is to build a precise model, expressed in Haskell, of the basic operations of the relational data model as defined by Maier [Mai83].

This chapter is split into six sections. In the first section we develop Haskell support for some basic concepts arising from category theory [Bac95] which will be thoroughly use in the modeling. In the second section we introduce and model the concept of a *constrained* data type. Thereafter the third describes two families of abstractions including *collective data types* such as finite *power sets* ($\mathcal{P}A$) and finite partial *mappings* ($A \rightharpoonup B$), while section four considers the specification of tuples and basic relation operations leading to relation algebra. The fifth section deals with monads and their application to capturing errors.

Finally, a sixth section describes the definition of the *relational database level* (RDB) where concepts such as key and integrity rule are integrated to complete the definition of the invariant property of the adopted relational data model.

3.1 Categorical Basis

This section presents the module `Cat_Impl.hs` (see Appendix A.1) which models a few basic generic concepts borrowed from category theory.

Categories. A category consists of a collection of *objects* and a collection of *arrows*. Each arrow $f :: a \to b$ has a source object a and a target object b. Two arrows f and g can be composed to form a new arrow $g \cdot f$, if f has the same target object as the source object of g. This composition operation is associative. Furthermore, for each object a there is a so-called identity arrow $id_a :: a \to a$, which is the unit of composition.

Our base category is called $Types$ and has types as objects and functions as arrows. Arrow composition is function composition (.) and the identity arrows are represented by the polymorphic function id.

Functors. Functors are structure-preserving mappings between categories. *Polymorphic* data types are functors from $Types$ to $Types$. In Haskell, functors can be defined by a type constructor f of *kind* $* \to *$ (recall section 2.2.1) mapping objects to objects, together with a higher-order function $fmap$, mapping arrows to arrows. This is provided as a constructor *class* in the Haskell *Prelude* (the standard file of primitive functions) as follows:

```
class Functor f where
   fmap :: (a -> b) -> (f  a -> f  b)
```

The arrow action of a functor must preserve identity arrows and distribute over arrow composition. For functors from $Types$ to $Types$, this means that the following equations must hold:

$$\begin{aligned} fmap\, id &= id \\ fmap\, (f \cdot g) &= (fmap\, f) \cdot (fmap\, g) \end{aligned}$$

Bifunctors. The *product category* $Types \times Types$ consists of pairs of types and pairs of functions. We can define functors from $Types \times Types$ to the base category $Types$ in Haskell. These functors are called *bifunctors*. A (curried) bifunctor in Haskell is a type constructor of kind $* \to * \to *$, together with a function $bmap$. The constructor class *Bifunctor* is made available as follows:

```
class Bifunctor f where
  bmap :: (a -> c) -> (b -> d) -> (f  a  b -> f  c  d)
```

Products. Categorical *products* are provided in Haskell by the type constructor for pairs (a, b) (usually written as Cartesian product $A \times B$ in mathematics) and projections fst and snd (resp. π_1 and π_2 in standard mathematical notation). Type constructor $(,)$ is extended to a bifunctor in the obvious way:

```
instance BiFunctor ( ,) where
   bmap f g = f >< g
```

where

```
(><) :: (a -> b) -> (c -> d) -> (a,c) -> (b,d)
(f >< g) = split (f . fst) (g . snd)
```

and combinator $split :: (a \to b) \to (a \to c) \to a \to (b, c)$ behaves as follows: $split\, f\, g\, x = (f\, x, g\, x)$. Outfix notation $\langle f, g \rangle$ is often used instead of $split\, f\, g$.

Sums. Categorical *sums* (or coproducts) are defined in the Haskell *Prelude* by means of type constructor

```
data Either a  b = Left a | Right  b
```

where `Left` and `Right` constructors correspond to left and right injections (resp. i_1 and i_2 in standard mathematical notation), together with a function $either :: (a \to b) \to (c \to b) \to Either\ a\ c \to b$ satisfying the following equations:

$$(either\ f\ g) \cdot Left = f$$
$$(either\ f\ g) \cdot Right = g$$

Type constructor *Either* is extended to a bifunctor by providing the following instance of *bmap*:

```
(-|-)::(a -> b) ->(c -> d) ->Either a c ->Either b d
(f -|- g) (Left  a) = Left (f a)
(f -|- g) (Right b) = Right (g b))

instance BiFunctor Either where
    bmap f g = f -|- g
```

Out-fix notation $[f, g]$ is often used instead of $either\ f\ g$.

Isos. Isomorphisms are very important functions because they convert data from one "format" to another format without losing information. These formats contain the same "amount" of information, although the same datum adopts a different "shape" in each of them. Isomorphic data domains are regarded as "abstractly" the same.

The module $Cat\_Impl.hs$ contains isomorphisms useful in data manipulation that reflect properties of functorial operations. For instance, the *swap* function $swap :: (a\,, b) \to (b\,, a)$ is defined by $swap = \langle \pi_2, \pi_1 \rangle$ and establishes the *commutative property* of product, $A \times B \cong B \times A$. In Haskell syntax it is written as follows:

```
swap :: (a , b)-> (b , a)
swap (a , b) = (b , a)
```

Another well-known isomorphism, known as the *associative property* of product is:

$$A \times (B \times C) \quad \cong \quad (A \times B) \times C$$

This is established by $(A \times B) \times C \xrightarrow{assoc} A \times (B \times C)$ which is defined by

```
assoc = split (p1 . p1) (split (p2 .p1) p2)
```

Other useful functions related with this property are:

```
assocr (a, b , c)  = (a,(b , c))
unassocr(a,(b , c))= (a, b , c)

assocl (a , b , c) = ((a , b) , c)
unassocl ((a , b) , c) = (a , b , c)
```

McCarthy's Conditional. The point free version of conditional expressions of the form

$$if\ (p\ x)\ then\ (f\ x)\ else\ (g\ x)$$

for some given predicate $A \xrightarrow{p} Bool$ (Bool is the primitive data type containing truth values FALSE and TRUE), some "then"-function $A \xrightarrow{f} B$ and some "else"-function $A \xrightarrow{g} B$ is given by the *cond* combinator, which is defined in Haskell as follows,

```
cond :: (a -> Bool) -> (a -> b) -> (a -> b) -> a -> b
cond p f g = (either f g) . (grd p)
```

where *grd* implements the guard associated to a given predicate $A \xrightarrow{p} Bool$:

```
grd :: (a -> Bool) -> a -> Either a a
grd p x = if p x then Left x else Right x
```

This combinator is the well-known "McCarthy conditional" functional, which is often written as follows: $(p \to g, h)$.

Table 3.1 summarizes the mathematical and Haskell notations which will be used interchangeably in the rest of this thesis.

Math. Notation	*Haskell Language*	*Description*
$A \times B$	(a, b)	*product datatype*
π_1	fst	*product left projection*
π_2	snd	*product right projection*
$A + B$	$Either\ a\ b$	*coproduct datatype*
i_1	$Left$	*coproduct left injection*
i_2	$Right$	*coproduct right injection*
$\langle f, g \rangle$	$split\ f\ g$	*pair f and g*
$[f, g]$	$either\ f\ g$	*either f g (combinator)*
F f	$fmap\ f$	*functor f (where* F *is a functor)*
$p \to f, g$	$cond\ p\ f\ g$	*McCarthy's conditional*
$(p?)$	$grd\ p$	*Guard*
$\overline{f}$	$curry\ f$	*currying*
$\underline{k}$	$const\ k$	*constant function*

Table 3.1: Mathematics notation versus Haskell notation.

Some useful structures. We define the *monoid* type class with one (closed) operator, and the *Boolean algebra* type class with the corresponding Boolean operators as follows:

```
class Monoid a where
    (&) :: a -> a -> a

class BoolAlg a where
    (\/) :: a -> a -> a
    (/\) :: a -> a -> a
```

3.2 Constrained Data Types

The specification of a design normally involves data types subject to properties (usually called *invariants* or *integrity constraints*) which formalize real-life conventions, laws, rules, norms or natural constraints. For instance, a finite mapping can be modeled as a finite binary relation where no two pairs with the same first element can be found [1].

Notation A_ϕ will be used to denote datatype A subject to restriction $Bool \xleftarrow{\phi} A$ meaning:

$$A_\phi \stackrel{\text{def}}{=} \{a \in A \mid \phi\, a\}$$

We define the *CData* type class to model constrained abstract types in Haskell, as follows:

```
class CData a where
    inv :: a -> Bool
    inv a = True
    inv' :: a -> Error a
    inv' a = if (inv a) then Ok a
             else Err "Invariant violation"
```

exporting the invariant property in two ways: as a predicate (inv) which, by default, is true, and as its extension inv' resorting to the "error data type"

```
data Error a = Err String | Ok a
```

providing an error wherever inv is violated. (Dynamic error handling will be dealt with in detail in section 3.5).

This type class allows us to define a set of types (*CData*) which contains precisely those types for which a suitable definition of integrity constraint (*inv*) will be given using instance declarations. We define below some obvious instances of class $CData$:

```
instance (CData a, CData b) => CData (a,b) where
    inv(a,b) = (inv a) && (inv b)
```

[1] Section 3.3.2 describes this data type.

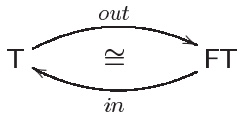
```
instance (CData a, CData b) => CData (Either a b) where
    inv (Left a) = inv a
    inv (Right b) = inv b

instance CData a => CData [a] where
    inv = all inv
```

These enable the structural propagation of invariant properties along structured datatypes. Wherever a data type a is of class $CData$, all functions which deliver output of type a should be invariant preserving. This can be ensured by standard proof obligations of invariant preservation, and dynamically checked by use of inv', as shown later on.

3.3 Modeling Finite Structures

An *inductive data type* is defined up to isomorphism

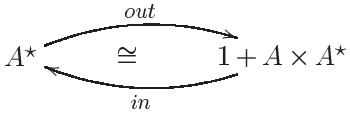


and from its definition two isomorphisms emerge — algebra in and co-algebra out — which are each other inverses:

$$\begin{aligned} in \cdot out &= id_{\mathsf{T}} \\ out \cdot in &= id_{(\mathsf{FT})} \end{aligned}$$

Algebra in provides data type constructors upon which one may build inductive definitions, algorithms, etc. It also sets up the basis for inductive proofs about the data type. Conversely, co-algebra out provides a "parser" for observing the data type in a "recursive descent" fashion. Because in and out are each other inverses, structuring algorithms around in or out is simply a matter of taste: the former leads to an "axiomatic" (structural, inductive) style, the latter to a more algorithmic (recursive, interpretative) style.

A leading example of inductive data type are lists over type A ($A^\star$). A list is either the empty list or an element appended to a list, that is:



Algebra in can be defined $in = [\underline{[\,]}, cons]$, where $\underline{[\,]}$ will express the "NIL pointer" (1 in the equation) by the empty sequence and $cons$ is the standard "left append" sequence constructor:

$$\begin{aligned} cons &: A \times A^\star \to A^\star \\ cons(a,[a_1,a_2,...,a_n]) &= [a,a_1,...,a_n] \end{aligned}$$

Co-algebra *out* can be defined $out = (\underline{[\,]} + < hd, tl >) \cdot (=_{[\,]}?)$, where sequence operators hd (*head of nonempty sequence*) and tl (*tail of nonempty sequence*) are described as follows:

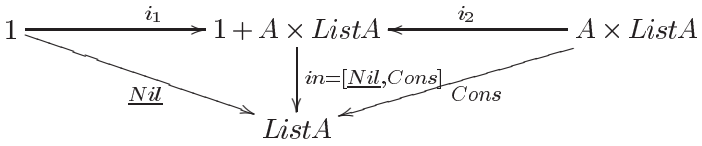
$$\begin{aligned} hd &: A^\star \to A \\ hd[a_1,a_2,...,a_n] &= a_1 \end{aligned}$$

$$\begin{aligned} tl &: A^\star \to A^\star \\ tl[a_1,a_2,...,a_n] &= [a_2,...,a_n] \end{aligned}$$

Data type $A^\star$ correspond to the Haskell `[a]` and operator ":" correspond to *cons* constructor. The Haskell built-in type of lists can be written, after the declaration of type A:

```
data ListA = Nil  | Cons (A, ListA)
```

Data type `ListA` is inhabited by *expressions* involving *Nil* and *Cons*, like: Nil, $Cons(a_1, Nil)$, $Cons(a_1, Cons(a_2, Nil))$, *etc*, and generates the following diagram:



3.3.1 Finite Sets

The power set data type $\mathcal{P}A$ containing all subsets of a finite set A:

$$\mathcal{P}A = \{s \mid s \subseteq A\}$$

is not a regular or inductive data type in this way. We know that the empty set $\emptyset$ is a subset of A — and therefore an inhabitant of $\mathcal{P}A$ — and that, given a subset s of A and a particular $a \in A$, then $\{a\} \cup s$ is another (possibly larger) subset of A. So a function $1 + A \times \mathcal{P}A \xleftarrow{ins} \mathcal{P}A$ can be defined, $ins = [\underline{\emptyset}, \cdot\lambda(a,s) \cdot \{a\} \cup s]$, which is an algebra for synthesizing $\mathcal{P}A$ data values. We will write

$$ins = [\underline{\emptyset}, puts]$$

by introducing

$$puts \stackrel{\text{def}}{=} \cup \cdot (sings \times id)$$

where $sings\ x = \{x\}$ ("singleton set"). Conversely, if $s \subseteq A$ is nonempty, it can always be decomposed into pair $(a, s - \{a\})$ for some $a \in A$. So we can think of $\mathcal{P}A \xrightarrow{outs} 1 + A \times \mathcal{P}A$ where

$$outs = (! + gets) \cdot (=_\emptyset)?$$

and where, for nonempty s,

$$\begin{array}{lll} gets\ s & \stackrel{\text{def}}{=} & let\ a \in s \\ & & in\ (a, s - \{a\}) \end{array}$$

From the outset, $\mathcal{P}A$ is a data type similar to $A^\star$: both share the same generative "grammar", or inductive "shape", defined by functor $\mathsf{X} = 1 + A \times X$. However, a major distinction between the two types can be identified at once: ins is not a right inverse of $outs$ on $\mathcal{P}A$, that is,

$$outs \cdot ins = id$$

does not hold. Because of the unordered structure of a set s one cannot reconstruct the steps along which it was built. So a powerset is not inductively generated, *i.e.*it is *non initial* in category-theoretical terms. This explain why *outs* above could not be defined inductively and why set-theoretical equality had to become explicit. Nevertheless,

$$ins \cdot outs = id$$

holds ($outs$ is a right inverse of in). Altogether, this means that $1 + A \times \mathcal{P}A$ contains "more information" than $\mathcal{P}A$ alone. We convey this fact by writing

$$\mathcal{P}A \underset{ins}{\overset{outs}{\leq}} 1 + A \times \mathcal{P}A \tag{3.1}$$

Data type Definition. Finite sets are modeled by finite lists constrained by the fact that elements in sets are not repeated,

$$\mathcal{P}A \leq A^\star{}_{nd}$$

where nd is defined as follows

$$nd(Set\ l) \quad \stackrel{\text{def}}{=} \quad length\ l = card(elems\ l) \tag{3.2}$$

where $length$ is Haskell standard and $card$(inal) and $elem$(ent)s have the usual set-theoretical meaning. Its data type definition in Haskell resorts to the predefined list data type:

```
data Set a =  Set [a]
```

This declaration introduces a new *type constructor* Set, with one *data constructor* Set (in this case the names are the same) whose data type invariant definition is:

```
instance  Eq a => CData (Set a) where
   inv (Set l) = length l == card(elems l)
   inv' s      = if (inv s) then Ok s
                            else Err "Set invariant violation"
```

The *trivial set* is a set having no elements:

```
emptyS:: Set a
emptyS = Set []
```

Sets are ordered by inclusion,

```
instance Eq a => Ord (Set a)
        where (Set s) <= r = all (-| r) s
```

where `all` is Haskell standard in the list data type and "`-|`" is a notation shortcut for set membership, `a -| s = ins a s`, where

```
ins :: Eq a => a -> Set a -> Bool
ins a (Set s) = elem a s
```

and `elem` is standard list membership. Finally, set-theoretical equality is defined in the usual way:

```
instance Eq a => Eq (Set a)
        where s == r = s <= r && r <= s
```

Cata, Ana, Hylo. The main consequence of the lack of in/out invertibility is that one cannot build catamorphisms over $\mathcal{P}A$: only *outs*-based hylomorphisms

$$\begin{array}{ccc} \mathcal{P}A & \xrightarrow{outs} & 1 + A \times \mathcal{P}A \\ {\scriptstyle [\![\ outs, g\ ]\!]}\downarrow & & \downarrow{\scriptstyle id+id\times[\![\ outs, g\ ]\!]} \\ B & \xleftarrow[g]{} & 1 + A \times B \end{array}$$

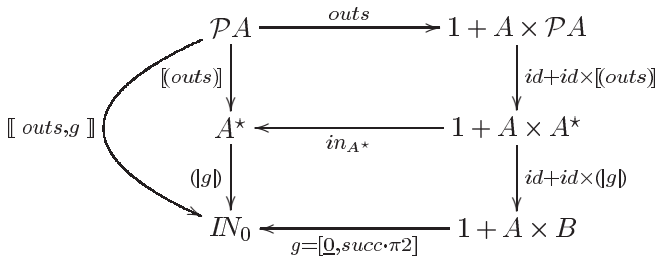
can be defined over some suitable target $(1 + A \times \_)$-algebra g. The set hylomorphism combinator in Haskell is defined as follows:

```
outSet:: Set a -> Either (Set a) (a,Set a)
outSet (Set [])= i1 (Set [])
outSet  s = i2 (gets s)

inSet:: Eq a => Either (Set a) (a, Set a)-> Set a
inSet= either (const emptyS) uputs

hyloSet ::(Either (Set a) (a,b) -> b) -> Set a -> b
hyloSet a = a . ( rec (hyloSet a)) . outSet where
            rec f = id -|- (id >< f)
```

For instance, the $card$ operator, that counts the number of elements of a finite set, is hylomorphism $[\![outs, g]\!]$, for $B = I\!N_0$ and $g = [\underline{0}, succ \cdot \pi 2]$, where $\underline{0}$ is constant function zero.



We will adopt notation $\{\!|g|\!\}$ as an abbreviation of $[\![\ outs, g\ ]\!]$, itself an alternative notation for $(\!|g|\!) \cdot (\!|outs|\!)$ over $A^\star$ finite sequences:

$$\{\!|g|\!\} \stackrel{\text{def}}{=} (\!|g|\!) \cdot (\!|outs|\!)$$

Then, we can define:

$$\begin{array}{lcl} card & : & \mathcal{P}A \to I\!N_0 \\ card & \stackrel{\text{def}}{=} & \{\!|[\underline{0}, succ \cdot \pi_2]|\!\} \end{array}$$

that is,

```
card:: Set a -> Int
card = hyloSet (either (const 0) (succ . p2))
```

Folding. The recursion operator *fold* encapsulates a common pattern for defining programs that *consume* values of a *least* fix-point type such as finite lists [GHA01]. In other words, when using *foldS*, a programmer writes functions consuming values of a data type D in terms of a *fold* function which captures the recursive traversal scheme for D. These functions are meant to replace the constructors in the traversal. We define $foldS\ f\ u$ as hylomorphism $\{\!|[\underline{u}, uncurry\ f]|\!\}$, that is,

```
foldS ::(a -> b -> b) -> b -> Set a -> b
foldS g u (Set [])= u
foldS g u    s = hyloSet (either (const u) (uncurry g)) s
```

in Haskell.

Universal quantification $\forall a \in s, p(a)$ is written $allS\ p\ s$ where $allS$ is another *hylomorphism* that can be seen as *set folding*:

```
allS :: (a -> Bool) -> Set a -> Bool
allS p (Set [])= False
allS p s = ((foldS (/\) True) . (nmap p) ) s
```

Functor. The powerset (type) functor

$$\begin{array}{rcl} \mathcal{P} & : & (A \to B) \to \mathcal{P}A \to \mathcal{P}B \\ \mathcal{P}f & = & \{\!|ins \cdot (id + f \times id)|\!\} \end{array}$$

is equivalent to $\{\![[\underline{\emptyset}, uncurry\ puts \cdot (f \times id)]]\!\}$ and boils down to point wise set-theoretical comprehension:

$$(\mathcal{P}fs) \quad = \quad \{fa \,|\, a \in s\}$$

Recall that a *functor* class with a *polymorphic map* (typed `fmap :: Functor a => (b -> c) -> a b -> a c`) is provided in the Haskell Prelude. However, our Set type cannot be made an instance of this class because $fmap$ must build sets that satisfy the set invariant (no duplicate elements) and, to do so, we need a context constraint: equality on the result type. That is to say, if we define

```
instance Functor (Set a) where
    fmap f = hyloSet (either (const (Set[]))
                             (uncurry puts.(f><id)))
```

the trouble is that type variables b and c in the signature of `fmap` are universally quantified. This means we can neither use equality or greater-than on the argument's elements nor on the resulting set's element [2].

To implement the power-set functor, we define a new three-parameter type class $NFunctor$ [3] as follows:

```
class NFunctor f a b where
     nmap :: (a -> b) -> (f a -> f b)
```

Then we define the three functors *Maybe, []* and *IO* that are provided in the standard distribution (*Prelude.hs*) as instances of this class at the obvious way: $nmap = fmap$. And finally we define the power-set functor as:

```
instance (Eq b) => NFunctor Set a b where
    nmap f = hyloSet (either (const (Set[]))
                     (uncurry puts.(f >< id)))
```

[2] Categorically speaking, Set is a functor over a *sub-category* of the category of Haskell types and functions, but Haskell gives us no way to express that [Hug99].

[3] One reason for this choice is that our model defines basically restricted types and a multi-parameter class allows us to make instance declarations that constrain the element type or the resulting element type *on a per-instance basis*.

Filtering. ZF-set abstraction,

$$\{f\ \ a \mid a \in s \wedge p(a)\}$$

corresponds to power-set *filtering*:

$$\begin{array}{rcl} filterS & : & (B \to A) \to (B \to Bool) \to \mathcal{P}(B) \to \mathcal{P}(A) \\ filterS\ f\ p & \stackrel{\text{def}}{=} & \{\!|[\underline{\emptyset}, \cup]|\!\} \cdot \mathcal{P}(p \to sings \cdot f, \underline{\emptyset}) \end{array}$$

In Haskell, we write (after power-set absortion):

```
filterS :: Eq a => (b -> a) -> (b -> Bool) -> Set b -> Set a
filterS f p
    = foldS (unions . cond p (sings . f) (const emptyS))
            (emptyS)
```

where power-set-absorption

$$\{\!|g|\!\} \cdot (\mathcal{P} f) \quad = \quad \{\!|g \cdot (id + f \times id)|\!\} \tag{3.3}$$

is applicable to algebras g satisfying the following property [4] :

$$\{\!|g|\!\} \cdot ins \quad = \quad g \cdot (id + id \times \{\!|g|\!\})$$

Union and Intersection Next we define infix operators and functions for intersection and union sets.

```
instance Eq a => BoolAlg (Set a) where
        r /\ s = filterS id (-| r) s
        (\/)  = foldS puts

unions :: (Eq a) => Set a -> Set a ->Set a
unions = (\/)

inters :: (Eq a) => Set a -> Set a -> Set a
inters r s= (r/\s)
```

Function *the* — returning the inhabitant of a singleton set —

$$\begin{array}{rcl} the & : & \mathcal{P}A \to A \\ the\ \{a\} & \stackrel{\text{def}}{=} & a \end{array}$$

is a partial function

```
the :: Set a -> a
the (Set [a]) = a
```

[4] see [Oli01b]

which is made total by resorting to the $Error$ data type:

```
the' :: Set a -> Error a
the' (Set[]  ) = Err "Empty Set - there is nothing in it"
the' (Set [a]) = Ok a
```

Function the can be used safely in contexts which ensure its application to nonempty arguments. We provide further details about this and other partial functions in section 3.5.

The following list of function signatures involving finite sets are defined in the model (the complete listing of the $Set\_Impl.hs$ module is given in appendix 7.4.4, A.2).

```
sings       :: a -> Set a
puts        :: Eq a => a -> Set a -> Set a
ltos        :: [a]->Set a
uputs       :: Eq a => (a,Set a) -> Set a
nins, ins   :: Eq a => a -> Set a -> Bool
incls       :: (Eq a) => Set a -> Set a -> Bool
unions      :: (Eq a) => Set a -> Set a ->Set a
inters      :: (Eq a) => Set a -> Set a -> Set a
diffs       :: (Eq a) => Set a -> Set a -> Set a
prods       :: Set a-> Set a -> Set (a,a)
dunion      :: Eq a => Set (Set a) -> Set a
zipS        :: Set a -> Set b -> Set (a,b)
unzipS      :: (Eq b, Eq a) => Set (a,b) -> (Set a,Set b)
zipWithallS :: Set a -> Set b -> Set (a,Set b)
stol        :: Set a-> [a]
elems       :: Eq a => [a] -> Set a
distls      :: a -> Set b -> Set (a,b)
```

Notation shortcuts. The following infix operators are provided as shortcuts:

```
a -| s = ins a s
a -||s = nins a s
r \< s = incls r s
```

3.3.2 Finite Mappings

There is a specialization of the power-set data type which is very useful in formal specification: the *finite mapping* data type

$$A \rightharpoonup B \quad \cong \quad \mathcal{P}(A \times B)_{fdp}$$

A finite mapping is thus a finite binary relation [5] subject to a data type invariant establishing a "$A \to B$" functional dependency,

$$fdp\, r \quad \equiv \quad \forall t, t' \in r.\, (\pi_1 t = \pi_1 t') \Rightarrow t = t'$$

[5] A finite binary relation is an inhabitant of data type $\mathcal{P}(A \times B)$, for finite A and B.

This predicate can also be written in point-free style as follows,

$$fdp \stackrel{\text{def}}{=} (\subseteq \{1\}) \cdot rng \cdot (id \rightharpoonup card) \cdot collect$$

where $collect :: \mathcal{P}(A \times B) \to (A \rightharpoonup \mathcal{P}B)$ is a function (to be defined later in this section) which converts a relation into a set-valued partial function and $rng :: (A \rightharpoonup B) \to \mathcal{P}B$ is the usual *range* function.

$A \rightharpoonup B$ is a finite "approximations" of the exponential type B^A, which is inhabited by (total or entire) functions from A to B. They can be "totaled" by transposition [OR04] by introduction of an explicit "error" value. Thus the basic isomorphism

$$A \rightharpoonup B \quad \cong \quad (B+1)^A$$

Data type Definition Finite mappings can also be modeled by the following (quasi) inductive data type

$$A \rightharpoonup B \quad \leq \quad 1 + (A \times B) + (A \rightharpoonup B)$$

leading to restricted lists of pairs,

$$A \rightharpoonup B \quad \leq \quad ((A \times B)^{\star}_{nd})_{fdp} \tag{3.4}$$

— where nd has been defined by (3.2) — and to the following Haskell type definition involving two type variables, products and lists:

```
data Pfun a b = Map [(a,b)]
```

This declaration introduces the new *type constructor* `Pfun` and the corresponding *data constructor* `Map`. Thus *domain* and *range*, two usual operators defined over finite mappings,

```
dom :: (Eq a) => Pfun a b -> Set a
dom (Map f) = (Set . nub . fst . unzip) f

rng :: (Eq b) => Pfun a b -> Set b
rng (Map f ) = (Set . nub . snd . unzip) f
```

and fold combinator

```
foldPf :: ((a,b) -> c -> c) -> c -> Pfun a b -> c
foldPf f u (Map s) = foldr f u s
```

In the limit, a partial function can be totally undefined, that is, $dom(f) = \phi$. There is one such function, the *empty function*:

```
emptyPf = Map []
```

Definition 13 (Coherence) *Two given partial functions $f : A \rightharpoonup B$ and $g : C \rightharpoonup D$ are said to be* coherent *iff*

$$\forall a \in dom\ (f) \cap dom\ (g) \quad : \quad f(a) = g(a)$$

Clearly, if $dom\ (f) \cap dom\ (g) = \emptyset$, then f and g are coherent. In Haskell this leads to predicate

```
coherent::(Eq a, Eq b)=> Pfun b a -> Pfun b a -> Bool
coherent f g = all (uncurry(==)) r
                 where Set r = rng(pfzip f g)
```

Partial mapping equality and ordering relations stem from this predicate:

```
instance (Eq a, Eq b) => Ord (Pfun a b) where
   f <= g = (dom f) <= (dom g) && coherent f g

instance (Eq a, Eq b) => Eq (Pfun a b) where
   f == g = f <= g && g <= f
```

For f and g two coherent partial functions, we can define their union as the following partial function:

```
unionpf: (Eq b, Eq a) => Pfun a b -> Pfun a b -> Pfun a b
unionpf f g = f \*/ g
```

and its total counterpart:

```
unionpf' :: (Eq b, Eq a) => Pfun a b -> Pfun a b -> Error (Pfun a b)
unionpf' f g = case (coherent f g) of
         True -> Ok (f \*/ g)
         False -> Err "map union: incompatible maps"
```

where operator $\backslash * /$, unrestricted union, is defined by

```
r \*/ s= foldPf putpf r s
```

For arbitrary partial functions f and g, it is possible to define a union operator generalization, the "override" operator († in VDM notation). This is function *plus* in our model:

```
plus :: (Eq b, Eq a) => Pfun a b -> Pfun a b -> Pfun a b
plus f g = (restn f (dom g) ) \*/ g
```

In the special case that the two maps are disjoint, the override operator reduces to the union operator. Hence we say that the override operator *covers* the union operator.

Function *pfzip* is another glueing operator that combines two partial functions as follows:

```
pfzip :: Eq b => Pfun b c -> Pfun b d -> Pfun b (c,d)
pfzip (Map f)(Map g) =
   Map [ a |-> (b,c) | (a,b) <- f , (d,c) <- g, a==d ]
```

that is, when the maps are non-disjoint, this function returns a function from the common domain to pairs that contain the corresponding ranges.

Function *pfzipWith op* works like *pzip* but resorts to an operator *op* to combine the values from the two maps, instead of just pairing them:

```
pfzipWith :: (Eq a, Eq b) =>
             (c -> d -> a) -> Pfun b c -> Pfun b d -> Pfun b a
pfzipWith f a b = (id *-> (uncurry f))(pfzip a b)
```

where infix operator `*->` corresponds to a bifunctor to be defined shortly.

We provide another function (*monpf*) for maps that yields a monoid operator combining the the override operator and *pfzipWith* function as follows:

```
instance (Eq a, Eq b, Monoid b) => Monoid (Pfun a b) where
        (&) = monpf (&)

monpf :: (Eq b, Eq a) =>
         (b -> b -> b) -> Pfun a b -> Pfun a b ->  Pfun a b
monpf f m n = plus m (plus n (pfzipWith f m n))
```

This monoidal operator will prove to be of great significance in some of our later specifications.

Filtering. Two useful filtering operators are positive ($\mid$) and negative ($\setminus$) domain restriction, both with signature

$$\mid, \setminus \quad : \quad \mathcal{P}A \times (A \rightharpoonup B) \to (A \rightharpoonup B) \tag{3.5}$$

The corresponding Haskell definitions are as follows:

```
restp :: Eq a => Set a -> Pfun a b -> Pfun a b
restp (Set s)(Map f) = Map ([ p | p <- f ,(elem (fst p) s )])

restn :: Eq a => Set a -> Pfun a b -> Pfun a b
restn (Set s)(Map f) = Map([ p | p <- f not (elem (fst p) s)])
```

Mapping look-up. Recall the operation of *applying* a function to an argument:

$$\begin{array}{rcl} ap & : & B^A \times A \to B \\ ap(\sigma, a) & \stackrel{\text{def}}{=} & \sigma a \end{array}$$

We define operators for applying finite partial and total functions to their arguments (modeled using *Pfun* data type) as

```
aplpf :: Eq a => Pfun a b -> a -> Maybe b
aplpf (Map f) a = lookup a f
```

where *lookup* is the list look-up function defined in the Haskell Prelude, and

```
aplft :: (Eq a, Eq b) => Pfun a b -> a -> b
aplft f = the . rng . (flip restp f) . sings
```

wherever f is a total mapping.

Mapping composition. The following combinator can be used to model partial function composition:

```
compf :: Eq a => Pfun a b -> Pfun c a -> Pfun c b
compf (Map f) (Map g) =
       Map [ a |-> c | (a,b) <- g, (b',c) <- f, b'==b ]
```

Bifunctors. Given two partial functions $f : A \rightharpoonup B$ and $g : C \rightharpoonup D$ such that f is injective, we can define the functorial construct $f \rightharpoonup g$ as follows:

$$\begin{array}{rcl} f \rightharpoonup g & : & (A \rightharpoonup C) \to (B \rightharpoonup D) \\ (f \rightharpoonup g)\ \sigma & \stackrel{\text{def}}{=} & \{\!|ins \cdot (id + (f\ \times\ g)\ \times id)|\!\} \end{array}$$

In Haskell, we introduce infix operator `*->`

```
g *-> f = bmapPf g f
```

which corresponds to function

```
bmapPf::(Eq c, Eq d)=>(a->c)->(b->d)->Pfun a b->Pfun c d
bmapPf f g  h = foldPf (\p ->putpf ((f >< g) p)) (Map[]) h
```

where

```
putpf::(b,a) -> Pfun b a -> Pfun b a
```

is the function that adds a pair of values to a partial function if the pair is coherent with the function.

Relations versus finite mappings. The well-known isomorphism between binary relations into set-valued functions

$$\mathcal{P}(A \times B) \underset{discollect}{\overset{collect}{\cong}} (\mathcal{P}B)^A$$

is approached, for finite such structures, as follows,

```
collect :: (Eq a, Eq b) => Set (a,b) -> Pfun a (Set b)
collect (Set r) =
    Map (nub [a |-> Set [b | (c,b) <- r ,a == c ] | (a,b) <- r])

discollect:: Pfun a (Set b) -> Set (a,b)
discollect (Map f) = Set [(a,b) | (a, (Set s)) <- f , b <-s ]
```

provided the empty set is not considered in the range of the involved finite mappings.

Data type Invariant. Now, we can define the finite mapping data type invariant (see 3.4):

```
instance (Eq a, Eq b) => CData (Pfun a b) where
   inv (Map pf) =inv (Set pf)&& fdp(Set pf)
   inv' f       = if (inv f) then Ok f
                      else Err "Partial function invariant violation"
```

where

```
fdp :: (Eq a, Eq b) => Set (b,a) -> Bool
fdp  = ((<= sings 1). rng . (id *-> card) . collect)
```

Finite mapping algebra. The list of functions involving finite mappings defined in the model is as follows (the complete code is shown in section A.3 of appendix 7.4.4):

```
aplft     :: (Eq a, Eq b) => Pfun a b -> a -> b
aplpf     :: Eq a => Pfun a b -> a -> Maybe b
aplpf'    :: Eq a => Pfun a b -> a -> Error b
bmapPf    :: (Eq a, Eq b) =>
                 (c -> b) -> (d -> a) -> Pfun c d -> Pfun b a
bpfFalse  :: (Eq (Set a), Eq a) => Pfun a Bool -> Set a
bpfTrue   :: (Eq (Set a), Eq a) => Pfun a Bool -> Set a
collect   :: (Eq a, Eq b) => Set (a,b) -> Pfun a (Set b)
compat    :: (Eq a, Eq b) => Pfun a b -> Pfun a b -> Set a
coherent  :: (Eq a, Eq b) => Pfun b a -> Pfun b a -> Bool
compf     :: Eq a => Pfun a b -> Pfun c a -> Pfun c b
discollect:: Pfun a (Set b) -> Set (a,b)
dom       :: (Eq a) => Pfun a b -> Set a
fdp       :: (Eq a, Eq b) => Set (b,a) -> Bool
filterPf  :: (Eq a, Eq b) =>
                 ((b,a) -> Bool) -> Pfun b a -> Pfun b a
foldPf    :: ((a,b) -> c -> c) -> c -> Pfun a b -> c
get       :: Eq a => b -> a -> Pfun a b ->  b
incompat  :: (Eq a, Eq b) => Pfun a b -> Pfun a b -> Set a
mkr       :: Pfun a b -> Set (a,b)
monpf     :: (Eq b, Eq a) =>
              (b -> b -> b) -> Pfun a b -> Pfun a b ->  Pfun a b
pfinv     :: (Eq a, Eq b) => Pfun a b -> Pfun b (Set a)
pfunzip   ::(Eq a, Eq b, Eq c) =>
                Pfun b (c,a) -> (Pfun b c,Pfun b a)
pfzip     :: Eq b => Pfun b c -> Pfun b d -> Pfun b (c,d)
pfzipWith :: (Eq a, Eq b) =>
                (c -> d -> a) -> Pfun b c -> Pfun b d -> Pfun b a
plus      :: (Eq b, Eq a) => Pfun a b -> Pfun a b -> Pfun a b
putpf     :: (Eq a, Eq b) => (b,a) -> Pfun b a -> Pfun b a
renpf     :: (Eq a, Eq b) => Pfun a a -> Pfun a b -> Pfun a b
restn     :: Eq a => Pfun a b -> Set a -> Pfun a b
restp     :: Eq a => Pfun a b -> Set a -> Pfun a b
rng       :: (Eq b) => Pfun a b -> Set b
```

```
singpf    :: (a,b) -> Pfun a b
tcollect  :: (Eq a, Eq b) => Set a -> Set (Pfun a b) ->
                               Pfun (Pfun a b) (Set (Pfun a b))
tnest     :: Eq a => Set a -> Pfun a b -> (Pfun a b,Pfun a b)
tot2      :: (a -> b -> b) -> b -> Set (c,a) -> b
ttot      :: (Eq a, Eq b) =>
             b -> (a -> a -> a) -> a -> Set (Pfun b a) -> Pfun b a
unionpf   :: (Eq b, Eq a) => Pfun a b -> Pfun a b -> Pfun a b
unionpf'  :: (Eq b, Eq a) =>
                   Pfun a b -> Pfun a b -> Error (Pfun a b)
```

Notation Shortcuts. We provide infix alternatives for the most common finite mapping operators:

```
x |-> y = (x,y)
s <:  f = restp f s
s <-: f = restn f s
g *-> f = bmapPf g f
r \*/ s = foldPf putpf r s
a <.  f = aplft f a
```

3.4 Modeling Relational Data in Haskell

The concept of a relation in the relational model is slightly more abstract than its counterpart in mathematics. The relational model deals with *tuples* by their information content. The tuple specification requires a careful explanation which is given in the following sections.

3.4.1 Embedding Products into Tuples

An n-ary relation in mathematics is a subset of a finite n-ary product $A_1 \times .. \times A_n$, which is inhabited by n-ary vectors $t = \langle a_1, \ldots, a_n \rangle$. Each entry a_i in vector t is accessed by its position's projection $\pi_i : A_1 \times \ldots \times A_n \to A_i$. This, however, is not expressive enough to model relational data as this is understood in database theory [Mai83]. Two ingredients must be added, whereby *vectors* give place to *tuples*: attribute names and NULL values. Concerning the former, one starts by rendering vectorial indices explicit, in the sense of writing *e.g.* $t\ \ i$ instead of $\pi_i\ \ t$. This implies merging all data types A_1 to A_n into a single coproduct type $A = \sum_{i=1}^{n} A_i$ and then representing the n-ary product as

$$A_1 \times \ldots \times A_n \underset{f}{\overset{r}{\leq}} (\textstyle\sum_{i=1}^{n} A_i)^n$$

under representation function [6] $r\ \langle a_j \rangle_{j=1..n} \stackrel{\text{def}}{=} \lambda j.(i_j\ a_j)$ which entails invariant

$$\phi\ t \stackrel{\text{def}}{=} \forall j = 1, \ldots, n, t\ j = i_j\ x\ :\ x \in A_j$$

Note that $j = 1, \ldots, n$ can be written $j \in \overline{n}$, where $\overline{n} = \{1, \ldots, n\}$ is the initial segment of the natural numbers induced by n. Set $\overline{n}$ is regarded as the *attribute* name-space of the model [7].

As a second step in the extension of vectors to tuples, we consider the fact that some attributes may not be present in a particular tuple, that is, NULL values are allowed [8]:

$$(\sum_{i \in \overline{n}} A_i + 1)^n$$

This finally leads to tuples as inhabitants of

$$Tuple \quad = \quad (\overline{n} \rightharpoonup \sum_{i \in \overline{n}} A_i)$$

thanks to isomorphism $A \rightharpoonup B \ \cong\ (B + 1)^A$ [Oli90]. This models tuples of arbitrary aridity (up to n attributes), including the empty tuple.

More descriptively, we define

$$(\overline{n} \rightharpoonup \sum_{i \in \overline{n}} A_i) \quad \cong \quad ((IdAttr \times Value)^{\star}_{nd})_{fdp}$$

$$Tuple \quad = \quad IdAttr \rightharpoonup Value$$

where $IdAttr$ is a data type of attribute names and $Value$ is a data type that models a set of domains. Domains are sets of values from which the attributes draw their values. The idea is to prevent comparisons of attributes that are not based on the same domain by strongly type-checking domains based on their names.

The type checking can be generalized assuming some elementary set of value types. In this specification, we model five different *Value* domains: Integers, Reals, Text, Date and Time. In Haskell:

```
data Value = Int Int       |
             Float Float   |
             String String |
             Date String   |
             Time String    deriving (Show,Eq)

type IdAttr  = String
type Tuple =  Pfun IdAttr Value
```

[6] Injections $i_{j=1,n}$ are associated to the n-ary coproduct. $Left$ and $Right$ in Haskell correspond to i_1 and i_2, respectively.

[7] The fact that this can be replaced by any isomorphic collection of attribute names of cardinality n has little impact in the modeling, so we stick to $\overline{n}$.

[8] Think of 1 as the singleton type $\{\text{NULL}\}$.

3.4.2 Modeling Relations

Relations can be modeled as pairs of relation schemes and sets of tuples, see definition 1. The relation scheme (a set of attribute names, each one with one default value and an indication whether it is a key or not) can be defined as a partial function from attribute names to attribute information:

$$Relation = (IdAttr \rightharpoonup AttrInfo) \times \mathcal{P}(IdAttr \rightharpoonup \sum Value)$$

Writing the specification in Haskell:

```
type SchemaR = Pfun IdAttr AttrInfo
type Tuples  = Set Tuple

data AttrInfo = InfA {ifKey :: Bool,
                      dom   :: Value } deriving (Show,Eq)

data Relation = Rel { schemaR:: SchemaR,
                      tuples :: Tuples }
```

The specification of $Tuples$ as a set of $Tuple$ implies that only proper relations which contain no duplicate tuples are modeled.

Mappings at tuple level implies that relations are at least in the first normal form (1NF). Note that the default value implicitly defines the underlying type information. We can define constants of type $Value$ to be used in relation schemes, eg.:

```
domvalueInt   = Int 0
domvalueFloat = Float 0.0
domvalueString= String ""
domvalueDate  = Date ""
domvalueTime  = Time ""
```

The *trivial relation* is a relation having no attributes and no tuples. This relation is the only relation having an empty set of attribute names so it is also the *universal relation* of that type.

```
emptyRel= Rel (Map[])(Set[])
```

Note, however, that a relation with no tuples can have a non-empty set of attributes. As a matter of fact, a rather complex data type invariant has to be imposed on $Relation$ for this data type to be meaningful. Because it requires further notational machinery, its discussion is deferred to section 3.6.

3.4.3 Basic Relation Operations

The basic operators of relational algebra include *union, intersection, difference, selection, projection, natural-join, equi-join, division*, and *renaming*. "Boolean operations" [Mai83] union, intersection and difference require that the operand relations have the same scheme (see 2.5.1 definition 3).

Two relations on the same scheme can be considered sets over the same universe, the set of all possible tuples on the relation scheme. We relax this equality precondition to "schemes compatibility" (see definition 1

of section 3.3.2), defining operators more tolerant and leaving to the next level operators (the Database level) the responsibility of controlling schemes equality. The Haskell code for Boolean operations is:

```
-- Union
unionR ::   Relation -> Relation -> Relation
unionR (Rel s1 t1) (Rel s2 t2) =
        let a = compat s1 s2
            t1' = fmapS (a<:) t1
            t2' = fmapS (a<:) t2
        in (Rel ((a <: s1) \*/ (a <: s2)) (t1' \/ t2'))

-- Intersection
interR ::   Relation -> Relation -> Relation
interR (Rel s1 t1) (Rel s2 t2) =
        let a = compat s1 s2
            t1' = fmapS (a<:) t1
            t2' = fmapS (a<:) t2
        in (Rel (dom(a <: s1)<:(a <: s2) )(t1' /\ t2'))

-- Difference
diffR ::   Relation -> Relation -> Relation
diffR (Rel s1 t1) (Rel s2 t2) =
        let a = compat s1 s2
            t1' = fmapS (a<:) t1
            t2' = fmapS (a<:) t2
        in (Rel(dom(a <: s2)<-:(a<:s1))(diffs t1' t2'))
```

Concerning selection, *select* (denoted σ_R), is a un-ary operator on relations. When applied to a relation r, it yields another relation that is the subset of tuples of r with certain values on pre-specified attributes (see 2.5.1 definition 7). The argument and result relation schemes are the same.

$$\begin{aligned} \sigma_R &: (IdAttr \rightharpoonup Value) \to Relation \to Relation \\ \sigma_R\ f\ (Rel\ s\ t) &= Rel\ s\ (filterS\ id\ (coherent\ f)\ t) \end{aligned}$$

Select operators commute under composition. Let $r(R)$ be a relation and let A and B be attributes of R, with $a \in dom(A)$ and $b \in dom(B)$. The identity:

$$\sigma_R\ \{A = a\}(\sigma_R\ \{B = b\}(r)) \quad = \quad \sigma_R\ \{B = b\}(\sigma_R\ \{A = a\}(r))$$

always holds. This property can be calculated from our specification [9]:

[9] Power set-absorption (3.3) is useful in showing that power set-filtering commutes with composition.

$$
\begin{aligned}
& ((\sigma_R\ f_1) \cdot (\sigma_R\ f_2))\ (Rel\ s\ t) \\
= & \quad \{ \text{composition definition} \} \\
& \sigma_R\ f_1\ (\sigma_R\ f_2\ (Rel\ s\ t)) \\
= & \quad \{ \textit{select} \text{ definition } \} \\
& \sigma_R\ f_1\ (Rel\ s\ (filterS\ id\ (coherent\ f_2) t)) \\
= & \quad \{ \textit{select} \text{ definition again } \} \\
& Rel\ s\ (filterS\ id\ (coherent\ f_1)\ (filterS\ id\ (coherent\ f_2)\ t)) \\
= & \quad \{ \text{ composition} \} \\
& Rel\ s\ ((filterS\ id\ (coherent\ f_1) \cdot filterS\ id\ (coherent\ f_2))\ t) \\
= & \quad \{ (filterS\ id\ p\ \cdot filterS\ id\ q\ ) \text{ commutes } \} \\
& Rel\ s\ ((filterS\ id\ (coherent\ f_2) \cdot filterS\ id\ (coherent\ f_1))\ t) \\
= & \quad \{ \text{ composition definition, } \textit{select} \text{ definition twice } \} \\
& ((\sigma_R\ f_2) \cdot (\sigma_R\ f_1))\ (Rel\ s\ t)
\end{aligned}
$$

Projection (written π_R) is also a unary operator on relations. Where select chooses a subset of the rows in a relation, project chooses a subset of the columns (see 2.5.1 definition 8). We can define:

$$
\begin{aligned}
\pi_R & : \mathcal{P}(Attr) \to Relation \to Relation \\
\pi_R\ a\ (Rel\ s\ t) & = Rel\ (s \mid a)\ (nmap\ (\lambda x.\ x \mid a)\ t)
\end{aligned}
$$

that is, the projection of a relation onto a set s, is the relation obtained by simultaneously restricting (positive restriction) the relation scheme and each tuple in the relation to attributes in s. In Haskell:

```
projectR :: Set IdAttr -> Relation -> Relation
projectR a (Rel s t) = Rel ( a <: s ) (nmap ( a<: ) t)
```

Natural join ($\bowtie_R$) is a binary operator for combining two relations (see section 2.5.1 definition 9). In general, join combines two relations on all their common attributes. Otherwise, join returns the Cartesian product of them.

```
natjoinR (Rel s1 t1) (Rel s2 t2) =
 let t=(nmap ( \x ->
               (nmap (split (const x) id)
                     (filterS id (coherent x) t2)))t1)
     ta=(foldS (unions) (Set[]) ) t
   in (Rel ( (dom(s1) /\ dom(s2)) <-: ( s1 \*/ s2))
           (nmap (\x ->
                  (dom(s1)/\dom(s2)) <-: (plus (p1 x)(p2 x))) ta))
```

The *divide* operator ($\div_R$), has a rather complex definition, but it does have some application in natural situations (see section 2.5.1 definition 10). Another way to define the divide operator follows: let $r(R)$ and $s(S)$ be relations with schemas such that $S \subseteq R$, and let $R' = R - S$. Then $r \div s$ is the maximal subset r' of $\pi_R\{R'\}(r)$ such that $r' \bowtie_R s$ is contained in r. The natural join in this case is a Cartesian product.

```
divideR ::   Relation ->   Relation -> Relation
divideR r s =
  let y= diffs (dom(schema r)) (dom(schema s))
      x= nmap (\t -> projectR y (selectR t  r))(tuples s)
      u = projectR y  r
  in foldS interR u x
```

Finally, the *renaming* operator (δ_R) takes a rename function from attribute names to attribute names ($IdAttr$ in our model) and one relation and returns a relation where attribute names have been changed according to the rename function (see section 2.5.1, definition 11). Attribute domains are not changed.

```
renameR ::  Pfun IdAttr IdAttr -> Relation -> Relation
renameR f (Rel s t)=(Rel(renpf f s)( nmap (renpf f) t))
```

3.5 Monadic Error Handling

The RDB-Model contains partial functions. These cannot be applied to all of their inputs. In order to avoid run-time errors one has two alternatives, either by ensuring that every call to a function g with parameter x is *protected - i.e.*, the context which wraps up such calls ensures *pre-condition* $x \in dom\ \ g$, or one *raises* exceptions, *i.e.* explicit error values. In the former case, mathematical proofs need to be carried out in order to guarantee *safety* (that is, *pre-condition* compliance). The overall effect is to *restrict* the domain of the partial function. In the latter case one goes the other way round, by extending the co-domain of the function so that it accommodates exceptional outputs. We implement the second alternative with an $Error$ monad (see 2.3.1). This monad (also called the $Exception$ monad) embodies the strategy for combining computations that can throw exceptions by passing bound functions from the point an exception is thrown to the point that it is handled.

Our Error monad is based upon the data type:

```
data Error a = Err String | Ok a
                    deriving (Show, Eq)
```

that extends an arbitrary data type `a` with its (polymorphic) exception (or error) value. The constructor `Ok` encloses a normal expression of type `a` and the constructor `Error` models an evaluation failure as a data value plus an error string. Clearly, one has

$$ErrorA \quad \cong \quad 1 + A$$

So, in abstract terms, one may regard as *partial* every function of signature

$$1 + A \xleftarrow{g} B$$

The next step is to define total(ized) versions of partial functions in the model. We term the new functions *<name function>'*. Let us see some examples.

In the *Set data type* model, we add the *gets'* and *the'* alternatives to $gets$ and the:

```
gets' :: Set a -> Error (a,Set a)
gets' (Set [])= Err "Empty Set-Nothing to get"
gets' (Set (a:x) )= Ok (a,Set x)

the' :: Set a -> Error a
the' (Set[]) = Err "Empty Set - there is nothing in it"
the' (Set [a]) = Ok a
```

To the *partial function data type* model we add *aplpf'* [10] and *unionpf'* functions:

```
aplpf' :: Eq a => Pfun a b -> a -> Error b
aplpf' (Map f) a =
   case (lookup a f) of
   Nothing -> Err "Apply Empty Partial Function"
   Just x  -> Ok x

unionpf' :: (Eq b, Eq a) => Pfun a b -> Pfun a b -> Error (Pfun a b)
unionpf' f g = case (coherent f g) of
         True -> Ok (plus f g)
         False -> Err "map union: incompatible maps"
```

Finally, to the *relation datatype* model we add:

```
valType' :: Value -> Error String
valType' (Int _   ) = Ok "Int"
valType' (String _) = Ok "String"
valType' (Date _  ) = Ok "Date"
valType' (Time _  ) = Ok "Time"
valType'  _     = Err "unknown ValueType"
```

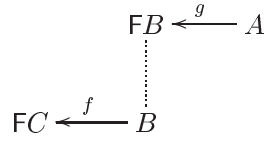
Now we need to extend the domain of functions that compose with the "new" *partial functions* such that they are able to accept arguments from $Error\ A$ and propagate them. That is, for g' a partial function that produces an $Error\ A$ value and f a total function which we want to compose with g', we need to "lift" f to $f^{\#}$ such that it accepts arguments from g'. Graphically:

$$C \xleftarrow{f} B \xleftarrow{g} A \tag{3.6}$$

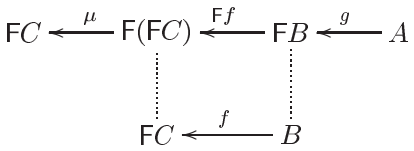
$$\begin{array}{ccc} & & Error\ B \xleftarrow{g'} A \\ & & \vdots \\ Error\ C & \xleftarrow{f^{\#}} & Error\ B \end{array}$$

[10] Note that the "error message" exception handling data type extends $Maybe$ ($Maybe\ A \leq Error\ A$).

Note that the composition scheme for the Error data type (3.6) generalizes to the following polytypic pattern [Oli01c]: the output of the producer function is "F-times" more elaborate than the input of the consumer function, where F is some parametric data type – $\mathsf{F}X = 1 + X$.



Then a composition scheme is devised for such functions, which is depicted as follows:



and is given by

$$f \cdot^{!} g \quad \stackrel{\text{def}}{=} \quad \mu \cdot \mathsf{F} f \cdot g$$

where $\mathsf{F}A \xleftarrow{\mu} \mathsf{F}^2 A$ is a suitable polymorphic function. Together with a unit function $\mathsf{F}A \xleftarrow{u} A$ and μ, data type F will form a so-called *monad* type, of which *Error A* is an example.

Arrow $\mu \cdot \mathsf{F} f$ is called the *extension* of f. Functions μ and u are referred as the monad's *multiplication* and *unit*, respectively. The monadic composition scheme $f \cdot^{!} g$ is called *Kleisli composition*.

In the *Standard Prelude*, Haskell defines the $Monad$ class as follows:

```
class Monad m where
   return :: a -> m a
   (>>=)  :: m a -> ( a -> m b) -> m b
```

where return refers to the unit of m, and the binding operator (>>=) is used to define the μ multiplication operator, function `join` in module `Monad.hs`.

We define Error Data type as instance of Haskell Monad Class and overload the monadic operators "return" and "bind" as follows:

```
instance Monad Error where
    return b = Ok b
    (Err e) >>=  f =  Err e
    (Ok  a) >>=  f = f a
```

We define two functions to extend the domain of functions to accept arguments from $Error\ a$, one for unary functions and another for binary functions:

```
lift :: Monad a => (b -> c) -> b -> a c
lift f n = return(f n)

lift2 :: Monad a => (b -> c -> d) -> b -> c -> a d
lift2 f n m = return (f n m)
```

Two functions in monadic form `f::a -> m b` and `g::b -> m c` can be composed via Kleisli composition:

```
(.!)::Monad a =>(b -> a c) ->(d -> a b)-> d -> a c
(f .! g) x  = (g x) >>= f
```

The application of a "lifted" function is the following operator:

```
(!):: Monad a => (b -> a c) -> a b -> a c
f ! x = x >>= f
```

Haskell provides an alternative syntax for bind ($>>=$) called "do notation" which is defined as follows:

```
do { y <- x ; f } = (x >>= (\ y -> f))
```

We use this notation to implement monadic folds on lists (*i.e.*, folding lists within a monad) and sets:

```
mfold :: Monad a => (b -> c -> a c) -> a c -> [b] -> a c
mfold f k [] = k
mfold f k (h:t) = do { b <- mfold f k t ;
                       f h b }

mfoldS :: Monad a => (b -> c -> a c) -> a c -> Set b -> a c
mfoldS f u (Set l) = mfold f u l
```

All operations at level of Relational Data Base level will be defined in the sequel using `do` notation. Prior to providing such definitions, we present the invariant of the $Relation$ data type, which also requires the $Error$ machinery.

3.6 The Relation Data type Invariant

A complex invariant is required to ensure that $Relation$ meets the integrity constraints of partial mappings and set compound types such that tuples are well and consistently typed. In the latter case, we need to ensure that:

1. all tuple schemes are mutually coherents, that is:

   $\forall t_i, t_j \in r \,.\, coherent(t_i, t_j)$

2. all tuples are type correct, and

3. the key-property is valid in the relation.

We can reason about the first property using the relation abstract data type in the following way:

$$
\begin{array}{ll}
 & Relation = (IdAttr \rightharpoonup AttrInfo) \times \mathcal{P}(IdAttr \rightharpoonup Value) \\
Step\,1 & \quad \{\ f_1 = id \times \mathcal{P}(id \rightharpoonup valType) \text{ where } valType : Value \to String\} \\
 & (IdAttr \rightharpoonup AttrInfo) \times \mathcal{P}(IdAttr \rightharpoonup String) \\
Step\,2 & \quad \{\ f_2 = id \times (mfoldS\ \ unionpf'\ \ (Ok\ \ emptyPf))\ \} \\
 & (IdAttr \rightharpoonup AttrInfo) \times Error(IdAttr \rightharpoonup String)
\end{array}
$$

In order to detect if some attribute has different types in different tuples, in Step 1 we apply the *valType* function to obtain the type of attributes in each tuple (tuple schemes). Then, we check for "scheme coherence" using function *unionpf'* to reduce the set of tuples schemes. Function *mfoldS* is the monadic version of *foldS* (see section 3.5).

We obtain the complete invariant on relations, using this method to formalize each property. The Haskell code for the invariant is as follows:

```
instance CData Relation where
 inv (Rel s t) =
    inv s &&
    inv t &&
    (m /= emptyPf ) &&
    relSchOk (Rel s t)&&
    fdpOkv (Rel s t)
      where
        m= foldS (\x y -> if (coherent x y) then unionpf x y else y)
                 emptyPf (nmap (id *-> valType) t)
        relSchOk r = m <= (id *-> (valType . defaultV)) (schema r)
        fdpOkv (Rel s t) = fdp(nmap (tnest (getKeyAtts s)) t)

 inv' (Rel s t) =
       do {
            inv' s
                 `xtherwise` "Relation schema is not a partial function" ;
            inv' t
                 `xtherwise` "Relation tuples set is not valid";
            m <- mfoldS unionpf' (Ok emptyPf) (nmap (id *-> valType) t)
                 `xtherwise` "tuple schemas are not mutually compatible" ;
            check (relSchemaOk m) (Rel s t)
                 "At least one tuple type does not match relation schema" ;
            check fdpOk (Rel s t)
                 "The key-property is not valid in the relation"
            }
            where
              relSchemaOk m r = m <= (id *-> (valType . defaultV)) (schema r)
              fdpOk (Rel s t) = fdp(nmap (tnest (getKeyAtts s)) t)
```

```
check p v s = if (p v) then (Ok v) else Err s

xtherwise (Ok  x) s = Ok x
xtherwise (Err _) s = Err s
```

3.7 The RDB Level

A *Relational Database* is a collection of named relations. Any given database, populated with data, is an instance of a *database scheme*, a specification of the names and types of data in each relation attribute. A name is just a string. Names are used to identify particular attributes relations and databases. We define:

```
data RDB = RDB { relations :: Pfun IdRel Relation }

type SchemaRDB = Pfun IdRel SchemaR
```

All operations already defined at *Relation*-level are lifted to RDB-level by adding error handling.

Boolean Operations

"Boolean operations" relational union, intersection and difference operations, require that operands relations have the same scheme (recall section 2.5.1, definition 3).

```
--      Union
union :: IdRel -> IdRel -> RDB -> Error Relation
union id1 id2 db =
  do {r1 <- aplpf' (relations db) id1 ;
      r2 <- aplpf' (relations db) id2 ;
      result <- inv' (unionR r1 r2) ;
      if (restrEqdom r1 r2)
      then Ok result
      else  Err "Error in Union Op.:incompatible schemes"}

--      Intersection
inter :: IdRel -> IdRel -> RDB -> Error Relation
inter id1 id2 db =
  do {r1 <- aplpf' (relations db) id1 ;
      r2 <- aplpf' (relations db) id2 ;
      result <- inv'(interR r1 r2) ;
      if (restrEqdom r1 r2)
      then Ok (result)
      else Err "Error in Intersection Op.:incompatible schemes"}

--      Difference
diff :: IdRel -> IdRel -> RDB -> Error Relation
diff id1 id2 db =
  do {r1 <- aplpf' (relations db) id1 ;
      r2 <- aplpf' (relations db) id2 ;
```

```
      result <- inv'(diffR r1 r2) ;
      if (restrEqdom r1 r2)
      then Ok (result)
      else Err "Error in Difference Op.:incompatible schemes"}
```

where *restrEqdom* predicate is as follows:

```
restrEqdom:: Relation -> Relation -> Bool
restrEqdom r1 r2 = schema r1 == schema r2
restrEqdom _  _ = False
```

Select Operation

This operator requires that all attributes specified in its first argument (selection function) belong to the relation scheme to be filtered.

```
select ::  Pfun IdAttr Value  -> IdRel -> RDB -> Error Relation
select f id db =
  do {r1 <- aplpf' (relations db) id ;
      if ((dom f) \<  (dom (schema r1)) )
      then inv' ( selectR f r1)
      else Err "Error in select Op.:attr.are not in rel.domain"}
```

Project Operator

Let r be a relation on scheme R, the *projection of r onto X*, $\pi_X(r)$ requires that X be a subset of R.

```
project :: Set IdAttr -> IdRel -> RDB -> Error Relation
project s id db =
  do {r1 <- aplpf' (relations db) id ;
      if (s \<  (dom (schema r1)) )
      then inv' ( projectR s r1)
      else Err "Error in project Op.:attr. not in rel. domain"}
```

Natural Join Operation

```
natjoin:: IdRel -> IdRel -> RDB -> Error Relation
natjoin id1 id2 db =
  do {r1 <- aplpf' (relations db) id1 ;
      r2 <- aplpf' (relations db) id2 ;
      inv'(natjoinR r1 r2) }
```

Divide Operation

Let $r(R)$ and $s(S)$ be relations, r **divided** by s ($r \div s$) requires that $S \subseteq R$.

```
divide :: IdRel -> IdRel -> RDB -> Error Relation
divide id1  id2 db =
  do {r1 <- aplpf'(relations db) id1 ;
```

```
      r2 <- aplpf'(relations db) id2 ;
      if ( (dom (schema r2)) \<  (dom (schema r1)) )
      then inv'(divideR r1 r2)
      else Err"Error in divide op.:second rel.domain is not
           include in first relation's domain"}
```

Renaming Operation

Let r be a relation on scheme R, then r with A **renamed** to B, denoted $\delta_{A \leftarrow B}(r)$, requires that A be an attribute in R and B be an attribute not in $R - A$.

```
rename::Pfun IdAttr IdAttr -> IdRel -> RDB -> Error Relation
rename f id1 db =
  do {r1 <- aplpf' (relations db) id1 ;
      if ((dom f) \< (dom (schema r1)) &&
           not((rng f) \< (dom (schema r1))))
      then inv'(renameR f r1)
      else Err "Error in rename op.: attr. not in rel.domain"}
```

3.8 Summary

In this chapter we have developed a model, written in the Haskell notation, of the basic operations of the relational algebra. The model is built in layers, corresponding to different families of abstractions. Each layer is associated with a data type and the operations required to deal with it.

We have defined a type class to model constrained data types. Once the model is built, appropriate monad-based techniques are introduced for error handling. This allows for the lifting to the RDB-level of the operations already defined at the relation-level.

Chapter 4

Toward On Line Analytical Processing

In this chapter, our goal is to extend the RDB model presented in the previous chapter in order to provide the functionality necessary for OLAP-based applications.

4.1 Relational versus Multidimensional Tables

The fundamental data structure of a multidimensional database is what we call an *n-dimensional table*. Let us start by giving some intuition behind the concept. We wish to be able to see values of certain attributes as "functions" of others, in whichever way suits us, exploiting possibilities of multi-dimensional rendering. Drawing on the terminology of statistical databases [Sho82], we can classify the attribute set associated with the scheme of a table into two kinds: *parameters* and *measures*. There is no a priory distinction between parameters and measures, so that any attribute can play either role [1]. An example of a two-dimensional table is given in Table 4.1 (adapted from [GL97]).

We want to work with the relational model we have defined in the previous chapter. A natural way to achieve this is to regard the multidimensionality of tables as an inherently *structural* feature, which is most significant when the table is rendered to the user. The actual *contents* of a table are essentially orthogonal to the associated structure, i.e., the distribution of attributes over dimensions and measures. Separating both features leads to a *relational* view of a table. For instance, the entry in the first (i.e., top left-most) "cell" in Table 4.1 containing the entry $(5, 6)$ corresponds to the tuple $(PC, Mendoza, 1996, Jan, 5, 6)$ over the scheme

$$\{Part, City, Year, Month, Cost, Sale\} \tag{4.1}$$

in a relational view of table $SALES$.

[1] Needless to say, the data type of a measure attribute must have some kind of metrics or algebra associated with it.

SALES			*TIME*						
			Year	1996			1997		
			Month	Jan	Feb	...	Jan	Feb	...
CATEGORY	*Part*	*City*	*(Cost, Sale)*						
	PC	Mendoza		(5, 6)	(5, 7)	...	(4, 6)	(4, 8)	...
		Córdoba		(5, 7)	(5,8)	...	(4, 8)	(4, 9)	...
		⋮		⋮	⋮		⋮	⋮	
	Inkjet	Mendoza		(7, 8)	(7, 9)	...	(6, 9)	(6, 8)	...
		Bs. As.		(6, 9)	(6, 9)	...	(5, 8)	(5, 9)	...
		⋮		⋮	⋮		⋮	⋮	
	⋮	⋮		⋮	⋮		⋮	⋮	

Table 4.1: $SALES$ — a sample two dimensional table with dimensions $Category$ and $Time$. The associated parameter sets are $\{Part, City\}$ and $\{Year, Month\}$, respectively. The measure attributes are $Cost$ and $Sale$.

To provide for OLAP, we need to define operations concerned with the following kinds of functionality:

- *Classification:* Ability to classify or group data sets in a manner appropriate for subsequent summarization.
- *Reduction/Consolidation:* Generalization of the aggregate operators in standard SQL. In general, reduction maps multi-sets of values of a numeric type to a single, "consolidated", value.

Classification is a generalization of the familiar SQL **group by** operator. The following example presents a typical query involving classification.

Example 4.1 *Consider the relation $RSALES$ with scheme (4.1) mentioned before. A typical query would be:* "find, for each part, the total amount of annual sales". *Even though this query involves* aggregation*, notice that it also involves* classifying *the data into various groups according to certain criteria, before aggregation is applied. Concretely, the above query involves classification by attributes $Part$ and $Year$.*

4.2 Classification

In our model, relations are sets of tuples (tables) with a scheme, while tuples are finite partial functions. First, we define a function for partial function decomposition (or "tuple classification"). Then we extend the notion of classification, applying it in the context of tables.

Partial function decomposition (Classification on Tuples) Let t be a tuple ($t \in Attribute \rightharpoonup Value$), and let $X = \{A_1, ..., A_k\}$ be an arbitrary subset of $dom(t)$. A

classification over X of tuple t, is the pair of tuples defined by $tnest\ X$, where $tnest$ is polymorphic function

$$tnest \quad : \quad \mathcal{P}A \to (A \rightharpoonup B) \to ((A \rightharpoonup B) \times (A \rightharpoonup B)) \tag{4.2}$$

$$tnest\ s \quad \stackrel{\text{def}}{=} \quad \langle \mid s, \setminus s \rangle \tag{4.3}$$

The idea of this function is to decompose a partial map into a pair of maps of the same type

$$A \rightharpoonup B \underset{\dagger}{\overset{tnest\ s}{\leq}} (A \rightharpoonup B) \times (A \rightharpoonup B)$$

which, together, rebuild the original map. In Haskell:

```
tnest :: Eq a => Set a -> Pfun a b -> (Pfun a b,Pfun a b)
tnest s f = (s <: f,s <-: f)
```

Tabular Decomposition (Classification on Tables) Let t be a set of tuples (typed $\mathcal{P}(Attribute \rightharpoonup Value)$), and let $X = \{A_1, ..., A_k\}$ be an arbitrary set of attributes of t. A $classification$ over X, of table t, is given by $tcollect\ X\ t$, where $tcollect$ is polymorphic function

$$tcollect\ s \quad \stackrel{\text{def}}{=} \quad collect \cdot \mathcal{P}(tnest\ s)$$

that is,

```
tcollect :: (Eq a, Eq b)
            => Set a
            -> Set (Pfun a b)
            -> Pfun (Pfun a b) (Set (Pfun a b))
tcollect s t = collect (nmap (tnest s) t)
```

in Haskell.

Classification essentially maps tuples of a relation to different groups (necessarily disjoint). Intuitively, we can think of the attributes in the first argument of $tcollect$ as corresponding to the "group id".

Example 4.2 *The classification part of the query of Example 4.1 can be expressed as follows:* tcollect {"Part","Year"} (tuples RSALES). *Table 4.2 illustrates the result of this operation in concrete Haskell syntax.*

```
Map[    ( Map [(”Part”, ”PC”),(”Year”, ”1996”)] ,
        Set[ Map[(”City”, ”Mendoza”),(”Month”, ”Jan”),(”Cost”, 5), (”Sale”, 6)],
             Map [(”City”, ”Mendoza”),(”Month”, ”Feb”),(”Cost”, 5), (”Sale”, 7)],
             .....,
             Map [(”City”, ”Cordoba”),(”Month”, ”Jan”),(”Cost”, 5), (”Sale”, 7)],
             .....] ),
        ( Map [(”Part”, ”PC”),(”Year”, ”1997”)] ,
        Set[ Map [(”City”, ”Mendoza”),(”Month”, ”Jan”),(”Cost”, 4), (”Sale”, 6)],
             Map [(”City”, ”Mendoza”),(”Month”,”Feb”),(”Cost”, 4), (”Sale”, 8)],
             .....,
             Map [(”City”, ”Cordoba”),(”Month”, ”Jan”),(”Cost”, 4), (”Sale”, 8)],
             .....] ),
        ( Map [(”Part”, ”Inkjet”),(”Year”, ”1996”)] ,
        Set [ Map [(”City”, ”Mendoza”),(”Month”, ”Jan”),(”Cost”, 7), (”Sales”, 8)],
             Map [(”City”, ”Mendoza”),(”Month”, ”Feb”),(”Cost”, 7), (”Sale”, 9)],
             .....,
             Map [(”City”, ”Buenos Aires”),(”Month”, ”Jan”),(”Cost”, 6), (”Sale”, 9)],
             .....] ),
        ( Map [(”Part”, ”Inkjet”),(”Year”, ”1997”)] ,
        Set [ Map [(”City”, ”Mendoza”),(”Month”, ”Jan”),(”Cost”, 6), (”Sales”, 9)],
             Map [(”City”, ”Mendoza”),(”Month”, ”Feb”),(”Cost”, 6), (”Sale”, 8)],
             .....,
             Map [(”City”,”Buenos Aires”),(”Month”, ”Jan”),(”Cost”, 5), (”Sale”, 8)],
             .....] )
        ]
```

Table 4.2: Output of the expression $tcollect\{"Part","Year"\}(tuples\ RSALES)$.

4.3 Reduction/Consolidation

Next, we consider reduction/consolidation, which includes not only applications of functions such as *max, min, avg, sum, count* to multi-sets of values defined by groups of tuples, but also statistical functions such as *variance* and *mode*, and business calculations such as *proportions* and *quarterlies*.

In our model, reduction functions map sets of tuples of values to individual values. We first define some necessary auxiliary functions.

Relational Reduction

Function

$$\begin{aligned} tot2 \quad &: \quad (A \times B \to B) \to B \to \mathcal{P}(C \times A) \to B \\ tot2 \ f \ u \quad &\stackrel{\text{def}}{=} \quad \{\!|[\underline{u}, f \cdot (\pi_2 \times id)]|\!\} \end{aligned}$$

reduces a binary relation on the second projection according to a reduction structure $A \times B + 1 \xrightarrow{\langle f, \underline{u} \rangle} B$ which, in most cases, is a monoid algebra. In Haskell:

```
tot2 :: (a -> b -> b) -> b -> Set (c,a) -> b
tot2 f = foldS (curry (uncurry f . (p2 >< id)))
```

Partial Function Application with a Default Value

Let apl be the isomorphism

$$A \rightharpoonup B \underset{apl^{-1}}{\overset{apl}{\cong}} (B+1)^A$$

in

$$\begin{array}{rcl} get & : & B \to A \to (A \rightharpoonup B) \to B \\ get\ u\ a\ f & \stackrel{\text{def}}{=} & [id, \underline{u}] \cdot (apl\ f\ a) \end{array}$$

In Haskell:

```
get :: Eq a => b -> a -> Pfun a b ->  b
get u a f = aux (aplpf' f a)
            where aux (Ok b) = b
                  aux (Err s) = u
```

Tabular Reduction

Finally, function

$$\begin{array}{rcl} ttot & : & B \to ((A \times A) \to A) \to A \to \mathcal{P}(B \rightharpoonup A) \to (B \rightharpoonup A) \\ ttot\ b\ f\ u\ s & \stackrel{\text{def}}{=} & \{b \mapsto tot2\ f\ u\ (\mathcal{P}(g\ u\ b))s\} \end{array}$$

performs tabular reduction, where $g\ u\ b \stackrel{\text{def}}{=} (id \times (get\ u\ b)) \cdot swap \cdot (tnest\ \{b\})$. Argument b specifies the measure attribute over which reduction will take place while arguments f and u provide the required reduction algebra. The output is packaged into a one-attribute tuple mapping the measure attribute name to the final result.

The corresponding Haskell code follows the above definition very closely:

```
ttot :: (Eq a, Eq b)
        => b
        -> (a -> a -> a)
        -> a
        -> Set (Pfun b a)
        -> Pfun b a
ttot b f u s = Map [ b |-> (tot2 f u (g u b s)) ]
     where g u b = nmap ((id >< (get u b)) . swap .
                         tnest (sings b))
```

Example 4.3 *Consider again the query of Example 4.1. We illustrate in this example how, from the classified set of tuples computed in Example 4.2, it is possible to obtain the final answer to the query.*

Let $fclass$ *be the mapping arising from the classification step (*$fclass$ *type is* $(Attr \rightharpoonup Value) \rightharpoonup \mathcal{P}(Attr \rightharpoonup Value)$*) computed in Example 4.2. We can use ttot to summarize over the* Sales *attribute, with a particular binary operation (monoid* $(+, 0)$ *in this example), in the range of* $fclass$*. The last step is to transform the resulting structure in a table (set of tuples). Diagram (4.4) depicts the required computations.*

$$\begin{array}{c} \mathcal{P}(Attr \rightharpoonup Value) \\ \Big\downarrow {\scriptstyle tcollect\ \{Part,Year\}} \\ (Attr \rightharpoonup Value) \rightharpoonup \mathcal{P}(Attr \rightharpoonup Value) \\ \Big\downarrow {\scriptstyle id \rightharpoonup (ttot\ Sales\ (+)\ 0)} \\ (Attr \rightharpoonup Value) \rightharpoonup (Attr \rightharpoonup Value) \\ \Big\downarrow {\scriptstyle mkr} \\ \mathcal{P}((Attr \rightharpoonup Value) \times (Attr \rightharpoonup Value)) \\ \Big\downarrow {\scriptstyle \mathcal{P}(uncurry\ plus)} \\ \mathcal{P}(Attr \rightharpoonup Value) \end{array} \tag{4.4}$$

Altogether, we have evaluated the expression

$$(\mathcal{P}plus) \cdot mkr \cdot (id \rightharpoonup (ttot\ Sale\ (+)\ 0)) \cdot (tcollect\ \ \{Part, Year\}) \tag{4.5}$$

4.4 Multidimensional Analysis

Next we define a "multidimensional analysis" function that generalizes the algebraic structure of (4.5) above:

$$mda\ \ s\ \ a\ \ f\ \ u \ \stackrel{\text{def}}{=}\ \ (\mathcal{P}plus) \cdot mkr \cdot (B \rightharpoonup ttot\ \ a\ \ f\ \ u) \cdot (tcollect\ \ s)$$

In the context of our relational model in Haskell, we provide the mda function defined over the $Relation$ data type, as follows:

```
mdaR::     Set IdAttr
        -> IdAttr
        -> (Value -> Value -> Value)
        -> Value
        -> Relation
        -> Relation
```

```
mdaR s a f u  r =
        Rel ((unions s (sings a)) <:(schema r))
            (nmap (uncurry plus)(mkr y))
        where y = (id *-> (ttot a f u)) x
              x = tcollect s (tuples r)
```

Table 4.3 illustrates the application of the "multidimensional analysis" operation $mdaR$ to our running example. Operation $mdaR$ produces a relation with scheme $Part, Year, Sale$ which is depicted two-dimensionally.

PartYearSales		*Year*		
		1996	1997	...
Part	PC	320	455	...
	Inkjet	298	450	...
	⋮	⋮	⋮	

Table 4.3: Output of the expression `mdaR (Set ["Part","Year"] ) "Sale" fadd f0 RSALES` applied to the input relation *RSALES* of Example 4.1.

4.5 Summary

In this chapter we have developed in Haskell a collection of functions that capture some of the functionality currently provided by multidimensional database products. This is done by defining operations which allow for classifying and reducing relations (tables). Suitably combined, these operations will make possible to carry out the multidimensional analysis of a relational database.

It should be stressed that the operations defined do not intend to address the issue of restructuring information from the perspective of the dimensionality of the data.

Chapter 5

Toward Generic Data Processing

5.1 Introduction

So far, Haskell has been used in animating an abstract model of the standard relational database calculus plus some OLAP extensions, written in the style of model-oriented formal specifications. However, several functions were specified which were more *general* than strictly required by the intended data-processing functionality. Genericity is a central topic of this thesis. In that follows, we address this topic by showing how parametericity and genericity make room for further extensions of the relational database model. For instance, the way tuples are structured in our current data model specification,

$$\begin{aligned} SchemaR &= IdAttr \rightharpoonup AttrInfo \\ Tuples &= \mathcal{P}(IdAttr \rightharpoonup Value) \\ Relation &= SchemaR \times Tuples \\ SchemaRDB &= IdRel \rightharpoonup SchemaR \\ RDB &= IdRel \rightharpoonup Relation \end{aligned}$$

calls for generalization of the collective type which contains tuples in each relation,

$$Tuples = \mathcal{P}(IdAttr \rightharpoonup Value)$$

to

$$Tuples = \mathsf{T}(IdAttr \rightharpoonup Value)$$

where T is an arbitrary parametric data type.

In this chapter we will extend our model by defining generic versions of relational operators. The intended generalization step, however, cannot be expressed in standard Haskell and calls for Generic Haskell. We start by over-viewing this language.

5.2 Overview of Generic H∀SKELL

Generic H∀SKELL is based on recent work by Hinze [Hin00b] and extends the functional programming language Haskell with, among other features, a construct for defining type-indexed values with kind-indexed types. These values can be specialized to all Haskell data types facilitating wider application of generic programming than provided by earlier systems as eg.POLYP [JJ97].

The Generic H∀SKELL compiler compiles modules written in an enriched Haskell syntax. A Generic H∀SKELL module may contain, in addition to regular Haskell code, definitions of generic functions, kind-indexed types, and type-indexed types, as well as applications of these to types or kinds. The compiler translates a Generic H∀SKELL module into ordinary Haskell by performing a number of tasks:

- translating generic definitions into Haskell code;
- translating calls to generic functions into an appropriate Haskell expressions, and
- specializing generic entities to the types at which they are applied. (Consequently, no type information is passed around at run-time).

In addition, the compiler generates structure types for all data types, together with functions that allow conversion between a data type and its structure type.

A Generic H∀SKELL program may consist of multiple modules. Generic functions defined in one module can be imported into and be reused in other modules. Generic H∀SKELL comes equipped with a library that provides a collection of common generic functions.

In the previous paragraphs we enumerate the basic features of Generic H∀SKELL which are published by Hinze [HJ03]. Some extensions described by Clarke and Löh in [CL02], which are implemented in the current version of the Generic H∀SKELL compiler, are:

- *copy lines*, a method to ease the programming of similar generic functions. A new generic function may be based on a previous one, inheriting the definitions for all cases that are not explicitly overwritten.
- *constructor cases*, which allow for generic function cases to be defined not only for named types, but also for particular constructors. This, at a first glance, may seem not particularly generic, but it happens to be useful in situations similar to those which require special cases to be defined for particular types.
- *generic abstractions* which allow generic functions to be defined by abstracting a type variable out of an expression which may involve generic functions.

We will use the last extension to define generic versions of our relational database operations. In particular, two generic functions included in the standard distribution of Generic H∀SKELL will be present in deriving all relational operators: $gmap$ and $rreduce$.

5.3 Generic Relational Operators

In a full generic function definition, one is forced to be more general than one intends to be. For instance, it is impossible to write a generic function that does not have a function type when applied to a type of kind $* \longrightarrow *$. This is because the specialization mechanism interprets abstraction and application at the type level as abstractions and application at the value level.

To illustrate the later assertion, we reproduce the code of $gmap$'s generic definition.

```
type Map {[ * ]} t1 t2 = t1 -> t2
type Map {[ k -> l ]} t1 t2 = forall u1 u2.
  Map {[ k ]} u1 u2 -> Map {[ l ]} (t1 u1) (t2 u2)

gmap {| t :: k |} :: Map {[ k ]} t t
gmap {| Unit |}     = id
gmap {| :+: |}  gmapA gmapB (Inl a)  = Inl (gmapA a)
gmap {| :+: |}  gmapA gmapB (Inr b)  = Inr (gmapB b)
gmap {| :*: |}  gmapA gmapB (a :*: b)= (gmapA a) :*: (gmapB b)
gmap {| (->) |} gmapA gmapB _        = error
                "gmap not defined for function types"
gmap {| Con c |}   gmapA       (Con a)  = Con (gmapA a)
gmap {| Label l |} gmapA       (Label a)= Label (gmapA a)
gmap {| Int |}      = id
gmap {| Char |}     = id
gmap {| Bool |}     = id
gmap {| IO |} gmapA = fmap gmapA
gmap {| [] |} gmapA = map gmapA
```

In the Map library, $gmap$ is the generic version of $fmap$ in the $Functor$ class. The type of $gmap$ is captured by a kind-indexed type which is defined by induction on the structure of kinds. (The part enclosed in $\{|.|\}$ is the kind index.)

The rest are equations, one for each type constant, where a type constant is either a primitive type like *Char, Int* etc or one of the three types *Unit*, ": $*$:" and ": $+$:" (null-ary products, binary products and binary sums respectively).

Generic abstraction lifts all restrictions that are normally imposed on the type of a generic function. It enables one to define a function which abstracts a type parameter from an expression, and later apply it generically. The abstracted type parameter is, however, restricted to types of a fixed kind. Generic abstractions can be used to write variations, simplifications and special cases of other generic functions.

The syntax of generic abstractions is similar to ordinary generic definitions, with two important differences:

- the type signature is restricted to a fixed kind, and thus no kind variable is introduced; and
- they consist of just one case which has a type variable as its type argument, rather than a named type.

Generic Project Suppose that we have several kinds of relations, in which the difference is the shape of the structure that contains the tuples, for instance:

```
data Relation   = Rel   {schema:: Pfun IdAttr AttrInfo,
                        tuples:: Set (Pfun IdAttr Value) }

data RelationL  = RelL  {schemaL:: Pfun IdAttr AttrInfo,
                        tuplesL:: [(Pfun IdAttr Value)] }

data RelationLT = RelLT {schemaLT:: Pfun IdAttr AttrInfo,
                        tuplesLT:: LTree (Pfun IdAttr Value) }
```

where $LTree$ is the "leaf tree" datatype, defined in Haskell as follows:

```
data LTree a = Leaf a | Split (LTree a, LTree a)
```

We want to define a generic function $\pi_{gtuples}$ which, taking a set of attributes names and a generic shape of tuples, returns a generic tuple structure where each tuple is restricted to the same set of attributes:

$$\begin{aligned} \pi_{gtuples} \quad &: \quad \mathcal{P}A \to \mathsf{T}(Tuple) \to \mathsf{T}(Tuple) \\ \pi_{gtuples}\ s\ ts \quad &= \quad \mathsf{T}(\lambda t.\, t \mid s)\ ts \end{aligned}$$

To encode this operator in Generic H∀SKELL , we define a generic abstraction which uses the *gmap* function (provided in Generic H∀SKELL library) to access each tuple of any tuple structure:

```
gprojectTup {| t :: * -> * |} ::
        (Eq a )
            => Set a
            -> t (Pfun a b)
            -> t (Pfun a b)
gprojectTup {| t |}  s ts = gmap {| t |} (restp s) ts
```

Generic functions are called by instancing the type-index to a specific type. As an illustration of this, we can specialize *gprojectTup* to different types, for instance:

```
projectR:: Set [Char] -> Relation -> Relation
projectR p (Rel s t)=
    Rel (restp p s ) (gprojectTup {| Set |} p t )

projectRL:: Set [Char] -> RelationL -> RelationL
projectRL p (RelL s t)=
    RelL (restp p s) (gprojectTup {| [] |} p t )

projectRLT:: Set [Char] -> RelationLT -> RelationLT
projectRLT p (RelLT s t)=
   RelLT (restp p s) (gprojectTup {| LTree |} p t)
```

Specialized Function *projectR* can be used to specify the project operation at RDB level (recall function $projectR$ defined in section 3.4.3) as follows:

```
sproject :: Set IdAttr -> IdRel -> RDB -> Error Relation
sproject s id db =
     do {r1 <- aplpf' (relations db) id ;
         result <- inv' ( projectR s r1) ;
         if (s \< dom(schema r1))
            then Ok (result)
            else Err "Error in project operation:
                      attributes are not in relation domain"
        }
```

Generic Select We proceed to defining $\sigma_{gtuples}$, the generic function that, taking a selection criteria (partial function) and a generic shape of tuples, returns a generic tuple structure where each tuple satisfies the selection criteria presented by the first argument (that is, a tuple structure where each tuple is "coherent" or "compatible" with the first argument):

$$\begin{aligned} \sigma_{gtuples} &: (A \rightharpoonup B) \to \mathsf{T}(Tuple) \to \mathsf{T}(Tuple) \\ \sigma_{gtuples}\ f\ ts &= gfilter\ (coherent\ f)\ ts \end{aligned}$$

To specify this operator in Generic H∀SKELL , we define a generic abstraction which uses the $rreduce$ function to implement $gfilter$. The $rreduce$ function, provided in Generic H∀SKELL library, is a generic version of $foldr$, typed as follows [1]:

$$\begin{aligned} rreduce\{|t :: *|\} &:: t \to B \to B \\ rreduce\{|t :: * \to *|\} &:: (A \to B \to B) \to tA \to B \to B \end{aligned}$$

Back to the definition of $\sigma_{gtuples}$, the idea is to access each tuple of a tuple structure and check its with the first argument of the selection function. If the tuple is compatible (coherent), it is "put into" the result structure. We parameterize the operation that permits to add a tuple to the structure and the empty structure (they will be known when the function will be instanced).

```
gselectTup {| t:: * -> * |} ::
       (Eq a, Eq b)
           => Pfun a b
           -> ((Pfun a b) -> t (Pfun a b) -> t(Pfun a b))
           -> t(Pfun a b)
           -> t(Pfun a b)
           -> t(Pfun a b)
gselectTup {| t |} p f te xs
           = rreduce {| t |}
```

[1] Note the reversed order of the last two arguments.

```
               (\x y -> if (coherent p x)
                        then (f x y)
                        else  y )
               xs te
```

The specialization of *gselectTup* to our original model is:

```
selectR :: Pfun [Char] Value -> Relation -> Relation
selectR p (Rel s t) =
            Rel s (gselectTup {| Set |} p puts (Set[]) t)
```

And the select operation at RDB-level becomes:

```
sselect ::  Pfun IdAttr Value
          -> IdRel
          -> RDB
          -> Error Relation
sselect f id db =
    do {r1 <- aplpf' (relations db) id ;
        result <- inv' (selectR f r1);
        if ((dom f) \< dom(schema r1))
           then Ok (result)
           else Err "Error in select operation:
                     attributes are not in relation domain"
       }
```

Generic Boolean Operations To specify *generic Boolean operations* of two structures of tuples, we define generic abstractions using $rreduce$ and $gany$ functions. In the case of *generic union*, $rreduce$ is used to add the tuples of the second structure to the first structure. Only the tuples that are not in the structure are added. Function $gany$ is provided in the Generic H∀SKELL library. It is a generic version of any (existential quantifying over finite lists defined in Haskell Prelude):

$$gany\{|t :: * \to *|\} \quad :: \quad (A \to Bool) \to \mathsf{T}A \to Bool$$

Function $gany$ will be used to check if each tuple is in the result structure. We parameterize the operation that adds a tuple to the structure (first parameter of $gunionTup$) because it won't be known until the function is specialized.

```
gunionTup {| t :: * -> * |} ::
         (Eq a, Eq b, Eq (Pfun a b))
              => ((Pfun a b) -> t(Pfun a b) -> t(Pfun a b))
              -> t(Pfun a b)
              -> t(Pfun a b)
              -> t(Pfun a b)
gunionTup {| t |} f t1 t2 =
  rreduce {| t |}
           (\x y-> if (gany{| t |}(\z -> z==x) t2)
                   then y
                   else (f x y)) t1 t2
```

Once again, what we had before at RDB-level stems from a specialization of generic *gunionTup*:

```
unionR  (Rel sx tx) (Rel sy ty) =
           Rel sx (gunionTup {| Set |} puts tx ty )
```

and

```
sunion :: IdRel -> IdRel -> RDB -> Error Relation
sunion id1 id2 db =
    do {r1 <- aplpf' (relations db) id1 ;
        r2 <- aplpf' (relations db) id2 ;
        result <- inv' (unionR r1 r2) ;
        if (restrEqdom r1 r2)
           then Ok result
           else  Err "Error in Union Operation:
                      incompatible schemes"
       }
```

In a similar way, the specification of *generic intersection* of two structures of tuples involves function $rreduce$ to construct a structure with the common tuples. We parameterize the operation that adds a tuple to the structure and the empty structure.

```
ginterTup {| t :: * -> * |} ::
     (Eq a, Eq b, Eq (Pfun a b))
          => ((Pfun a b) -> t(Pfun a b) -> t(Pfun a b))
          -> t(Pfun a b)
          -> t(Pfun a b)
          -> t(Pfun a b)
          -> t(Pfun a b)
ginterTup {| t |} f te t1 t2 =
  rreduce {| t |} (\x y-> if (gany{| t |}(\z -> z==x) t2)
                          then (f x y)
                          else y ) t1 te
```

The specialization of *ginterTup* and the intersection operation at RDB-level is as follows:

```
interR (Rel sx tx) (Rel sy ty)=
          Rel sx (ginterTup {| Set |} puts (Set[]) tx ty )
```

and

```
sinter :: IdRel -> IdRel -> RDB -> Error Relation
sinter id1 id2 db =
    do {r1 <- aplpf' (relations db) id1 ;
        r2 <- aplpf' (relations db) id2 ;
        result <- inv'(interR r1 r2) ;
        if (restrEqdom r1 r2)
           then Ok (result)
           else Err "Error in Intersection Operation:
                     incompatible schemes"
       }
```

The abstraction defined for *generic difference* between two tuple structures is similar to the *generic intersection* function, but with the if's branches inverted:

```
gdiffTup {| t :: * -> * |} ::
         (Eq a, Eq b, Eq (Pfun a b))
              => ((Pfun a b) -> t(Pfun a b) -> t(Pfun a b))
              -> t(Pfun a b)
              -> t(Pfun a b)
              -> t(Pfun a b)
              -> t(Pfun a b)
gdiffTup {| t |} f te t1 t2 =
  rreduce {| t |} (\x y-> if (gany{| t |}(\z -> z==x) t2)
                             then y
                             else (f x y)) t1 te
```

The specialization of *gdiffTup* and the difference operation at RDB-level for our model are

```
diffR  (Rel sx tx) (Rel sy ty)=
   Rel sx (gdiffTup {| Set |} puts (Set[]) tx ty )
```

and

```
sdiff :: IdRel -> IdRel -> RDB -> Error Relation
sdiff id1 id2 db =
    do {r1 <- aplpf' (relations db) id1 ;
        r2 <- aplpf' (relations db) id2 ;
        result <- inv'(diffR r1 r2) ;
        if (restrEqdom r1 r2)
           then Ok (result)
           else Err "Error in Difference Operation:
                     incompatible schemes"
       }
```

respectively.

Generic Renaming Let $\delta_{gtuples}$ denote the generic function that, taking a *rename* function from attribute names to attribute names and a generic shape of tuples, returns a generic tuple structure where attribute names are changed via the rename function. In Haskell, this is function

```
grenameTup {| t :: * -> * |} :: (Eq a, Eq b ) =>
                  Pfun a a -> t (Pfun a b) -> t (Pfun a b)
grenameTup {| t |} r xs =gmap {| t |} (renpf r) xs
```

As before, this instances to $Relation$ and then to RDB-level:

```
renameR :: Pfun [Char] [Char] -> Relation -> Relation
renameR p (Rel s t) =
            Rel (renpf p s) (grenameTup {| Set |} p t )
```

```
srename ::  Pfun IdAttr IdAttr
          -> IdRel
          -> RDB
          -> Error Relation
srename f  id1 db =
    do {r1 <- aplpf' (relations db) id1 ;
        result <- inv'(renameR f r1);
        if ((dom f) \< dom(schema r1))
          then Ok (result)
          else Err "Error in Rename Operation:
                 attribute to rename are not in relation domain"}
```

Generic Natural Join Let $\bowtie_{gtuples}$ denote the binary operator for combining two tuple structures on all their common attributes. Should they have no common attributes, $\bowtie_{gtuples}$ will return the Cartesian product of them.

First, we define an auxiliary function

$$\begin{array}{rcl} gfilter & : & (A \to Bool) \\ & & \to (A \to B) \\ & & \to ((B \times \mathsf{T}B) \to \mathsf{T}B) \\ & & \to \mathsf{T}B \\ & & \to \mathsf{T}A \\ & & \to \mathsf{T}B \\ gfilter\ p\ f\ op\ es & \stackrel{\text{def}}{=} & \{\!|[\underline{es}, (\lambda\, x\, y\, .(p\ x) \to op\ (f\ x)\ y, y)]|\!\} \end{array}$$

which filters a structure, retaining only those elements that satisfy p, and applies f to each such element. The third and fourth parameters correspond to the operation that adds a tuple to the structure and the empty structure, respectively. For instance, instanced to Set, $gfilter$ corresponds to ZF-set abstraction

$$\{f\ \ a \mid a \in s \wedge p(a)\}$$

```
gfilter {| t:: * -> * |} ::
          (a -> Bool)
       -> (a -> b)
       -> (b -> t b -> t b)
       -> t b
       -> t a
       -> t b
gfilter {| t |} p op f te xs =
     rreduce {| t |}
          (\x y -> if (p x)
                   then f (op x) y
                   else  y ) xs te
```

Then we can define:

```
gnatjoinTup {| t |} ft1 ft2 te1 te2 r1 r2 =
 let a=
```

```
    gmap {| t |}
         (\x->
            (gfilter {| t |}
                 (coherent x) (aux x) ft1 te1 r2)
         ) r1
 in rreduce {| t |} ft2 a te2
```

where

```
aux:: Eq a => Pfun a b -> Pfun a b -> Pfun a b
aux x y =(dom(x) /\ dom(y)) <-: (plus x y)
```

Finally, the specialization of *gnatjoinTup* and the natural join operation to our set-based RDB-level are, as expected:

```
gnatjoinR  (Rel sx tx) (Rel sy ty)=
  Rel ( (dom(s1) /\ dom(s2)) <-: ( s1 \*/ s2))
      ( gnatjoinTup {| Set |} puts (\/) (Set[]) (Set[]) tx ty)

snatjoin id1 id2 db =
    do { r1 <- aplpf' (relations db) id1 ;
         r2 <- aplpf' (relations db) id2 ;
         inv'(natjoinR r1 r2) }
```

5.4 Generic Olap Operators

We are now ready to define the generic counterparts of the OLAP functions described in section 4.1.

Partial Function Decomposition (Classification on Tuples) Let t be a tuple ($t \in Attribute \rightharpoonup Value$), and let $X = \{A_1, ..., A_k\}$ be an arbitrary subset of $dom(t)$. A *classification* over X of tuple t, is the pair of tuples defined by $tnest\ X$, where $tnest$ is the polymorphic function that decompose a partial map into a pair of maps of the same type (see 4.3).

Tabular Decomposition (Classification on Tables) Let t be a structure of tuples (typed $\mathsf{T}(Attribute \rightharpoonup Value)$), and let $X = \{A_1, ..., A_k\}$ be an arbitrary set of the attributes of t. A $classification$ over X, of structure t, is given by $gtcollect\ X\ t$, where $gcollect$ is polymorphic function

$$gtcollect\ s \stackrel{\text{def}}{=} gtcollect \cdot \mathsf{T}(tnest\ s)$$

where $gcollect$ is the function that converts a structure of pairs into a structure-valued finite mapping,

$$\mathsf{T}\ (A \times B) \underset{gdiscollect}{\overset{gcollect}{\rightleftarrows}} A \rightharpoonup \mathsf{T}B$$

written in Generic H∀SKELL as follows:

```
gcollect {| t |} ts =
       Map (nmap (split id (flip gextr {| t |} ts)) l )
           where l= (nub .
                     flatten {| t |} .
                     gmap {| t |} fst  ) ts
```

The *gextr* auxiliary function

$$gextr\ t \stackrel{\text{def}}{=} \mathsf{T}(\lambda(x,y).\ if\ (t = x)\ then\ y\ else\ EmptyPf)$$

takes a partial function and a structure with pairs of tuples (or "classified tuples", see *tnest* in section 4.1), and returns a structure with the same shape of the second argument, where each pair is replaced either by this second projection or by an "empty tuple". In case the first projection of a pair matches the partial function, the pair is replaced by its second projection.

For tuples, *gtcollect* is written as follows:

```
gtcollect{| t:: * -> * |} ::
    (Eq a, Eq b)
        => Set a
        -> t (Pfun a b)
        -> Pfun (Pfun a b) (t (Pfun a b))
gtcollect {| t |} s ts =
       gcollect {| t |} (gmap {| t |} (tnest s) ts)
```

Generic Relational Reduction. This is provided by function *gtot*2 which is defined by

$$\begin{aligned} gtot2 &: (A \times B \to B) \to B \to \mathsf{T}(A \times B) \to B \\ gtot2\ f\ u &\stackrel{\text{def}}{=} \{|[\underline{u}, f \cdot (\pi_2 \times id)]|\} \end{aligned}$$

and expressed in Generic H∀SKELL as follows:

```
gtot2 {| t:: * -> * |} ::
                  (a -> b -> b)
               -> b
               -> t (c,a)
               -> b
gtot2 {| t |} f u ts =
         rreduce {| t |} f y u
                 where y= gmap {| t |} snd ts
```

Generic Tabular Reduction We define *gttot* function, that performs tabular reduction, in a way similar to the *ttot* function (see section 4.1), simply by generalizing the power-set functor to an arbitrary regular functor:

```
gttot {| t:: * -> * |} ::
              (Eq a, Eq b)
               => b
               -> (a -> a -> a)
               -> a
               -> t(Pfun b a)
               -> Pfun b a
gttot {| t |} a f u ts =
     Map [ (a, (gtot2 {| t |} f u y))]
         where y = ( gmap {| t |} ((id >< (get u a)) .
                     swap .
                     tnest (sings a)) ts
```

Generic Multidimensional Analysis Finally, the specification of "generic multidimensional analysis" of tuple structures is given by:

```
gmda {| t:: * -> * |}::
              (Eq a, Eq b)
               => Set a
               -> a
               -> (b -> b -> b)
               -> b
               -> t (Pfun a b)
               -> Set (Pfun a b)
gmda {| t |} s a f u ts =
      nmap (uncurry plus) (mkr y)
         where y= (id *-> (gttot {| t |} a f u )) x
               x= gcollect {| t |} s ts
```

Note that $gmda$ always produces a "table" as result, that is, a "set" of tuples. This is because multi-dimensional analysis has to do with data reduction. Its output is always an association of such a reduction to the values of the preserved dimensions, irrespectively of the shape of the input structure.

5.5 Example

Generic functions are specialized at compile time and specializations are always generated locally per module. In order to specialize $gmda$ to "Sets of tuples", we define $smdaR$ and $smda$ functions, that specialize $gmda$ to Relations and RDB level respectively. In Haskell:

```
smdaR:: Set IdAttr
      -> IdAttr
      -> (Value->Value->Value)
      -> Value
      -> Relation
      -> Relation
smdaR  s a f u  r = Rel ((unions s (sings a)) <:(schema r))
```

```
                        (gmda{| Set |} s a f u (tuples r))

smda::  Set IdAttr
      -> IdAttr
      -> (Value->Value->Value)
      -> Value
      -> IdRel
      -> RDB
      -> Error Relation
smda s a f u id db = do { r1<- aplpf' (relations db) id ;
                          inv'(smdaR s a f u r1 ) }
```

We use the Generic Generic H∀SKELL compiler to produce an ordinary Haskell module with the "specialized" OLAP operations $smdaR$ and $smda$. Consider again the query of Example 4.1 in chapter 4, given the relation $RSALES$ with scheme $\{$*Part, City, Year, Month, Cost, Sale*$\}$, *"find, for each part, the total amount of annual sales"*. Expression `smdaR (Set["Part","Year"]) "Sales" valadd val0 "RSALES"` applied to the input Relation $RSALES$ of Example 4.1 produces the same output depicted in Table 4.3.

5.6 Summary

In this chapter we have provided definitions for generic versions of the standard relational operators and studied the effects of this generalization.

Among several possible generalizations, we chose to generalize the "shape" of the parametric type which collects all tuples in a relation.

We have also investigated the consequences of defining classification and reduction operations on generic tuple structures.

Chapter 6

An Approach to (Generic) Normalization

6.1 Motivation

In relational database design practice, starting relation schemes tend often to be *too large*, *unstructured* and affected by functional dependencies. The goal of the so-called *normalization theory* [Mai83], [Cod71b], [Cod72a] is to find a set of relation schemes — also called a *database scheme* — which is free of information redundancy (*i.e.* "is normalized") and yet able to represent the same information as the initial relation scheme. This chapter is devoted to illustrate the scaling up of *normalization theory* when it is applied under the principles of generic programming and generic information models developed in the previous chapters.

To provide a complete generalization of *normalization theory* would be a non-trivial task far beyond the scope and plan of this thesis. Instead, we develop an exercise which shows how such generalization can or could be performed [1].

Our example is as follows. Let us — back to the conventional, set-based (tabular) model — consider the following relational database scheme

$$S \quad = \quad \{\langle r = \{\alpha, \beta, \gamma, \delta\}, \alpha\beta\rangle\} \tag{6.1}$$

consisting of one table or relation *r* involving four attributes α, β, γ and δ such that $\alpha\beta$ is a key for this relation and that it satisfies functional dependence (FD) $\alpha \rightarrow \delta$. Such is the case of the following tabular illustration:

α	β	γ	δ
$a1$	$b1$	$c1$	$d1$
$a1$	$b2$	$c2$	$d1$
$a2$	$b3$	$c1$	$d2$

(6.2)

[1] A preliminary version of this exercise can be found in [NO02].

We would like to update the relation simply by specifying values for the key and then giving values for the remaining attributes. However, should we perform [2]

$$CH(r; a1, b1; \gamma = c1, \delta = d3)$$

then the relation will violate FD $\alpha \to \delta$.

To avoid violations of this kind, every time an update is made, one has to scan the whole relation and update the δ value everywhere the α value occurs, despite the fact that only one tuple is to be changed. This happens because the (α-value, δ-value) information is duplicated in the relation, thus making the data redundant. We are better off, with respect to updates and redundancy, if we represent the same information as a database of two relations, namely r_1 and r_2, as shown below:

$$(\ r_1 = \begin{array}{|c|c|}\hline \alpha & \delta \\ \hline a1 & d1 \\ \hline a2 & d2 \\ \hline \end{array} \quad , \quad r_2 = \begin{array}{|c|c|c|}\hline \alpha & \beta & \gamma \\ \hline a1 & b1 & c1 \\ \hline a1 & b2 & c2 \\ \hline a2 & b3 & c1 \\ \hline \end{array}\ ) \tag{6.3}$$

We can retrieve the original relation r by taking $r_1 \bowtie r_2$. The update anomaly no longer exists, since only one tuple needs to be updated to change a δ assignment. We have also removed some data redundancy, since (α-value, δ-value) pairs are recorded only once.

This process which extracts functional dependences is known as *normalization through decomposition* [Mai83, RO97]. In general, the set-theoretical model of S (6.1) using vector types is

$$\mathcal{P}(A \times B \times C \times D)_{\phi} \tag{6.4}$$

where $\mathcal{P}$ is the power-set functor and $A, \ldots, D$ are the types which are inhabited by the values of attributes $\alpha, \ldots, \delta$, respectively. Invariant ϕ

$$\phi \stackrel{\text{def}}{=} fdp \cdot \mathcal{P}\langle \pi_1, \pi_4 \rangle$$

records the $\alpha \to \delta$ functional dependence, where fdp is the predicate defined in section 3.3.2.

Our work plan concerning this example is as follows. First of all, we want to show how to extract the $\alpha \to \delta$ functional dependence from (6.4) by calculation. That is, we want to calculate a pair (rf, af) of abstraction/representation functions witnessing the inequality [3] which follows:

$$\mathcal{P}(A \times B \times C \times D)_{\phi} \underset{af}{\overset{rf}{\leq}} (A \rightharpoonup D) \times \mathcal{P}(A \times B \times C) \tag{6.5}$$

[2] $CH(...)$ denotes the *change* operation, used to modify only part of a tuple (see [Mai83], page 8).
[3] See section 3.1.

For instance, representation function rf applied to table (6.2) should decompose it in pair of tables of (6.3).

Secondly, we want to show that such a decomposition is not a privilege of the relational (tabular) information model and that it can in fact be generalized to any other collective data type T,

$$\mathsf{T}(A \times B \times C \times D)_{\phi} \quad \leq \quad (A \rightharpoonup D) \times \mathsf{T}(A \times B \times C) \tag{6.6}$$

under a generalized invariant

$$\phi \quad \stackrel{\text{def}}{=} \quad fdp \cdot pelems \cdot \mathsf{T}\langle \pi_1, \pi_4 \rangle$$

where $\mathcal{P}X \xleftarrow{pelems} \mathsf{T}X$ is the polytypic operation which collects all data from the nodes of a T-structure (of course, $pelems = id$ for $\mathsf{T} = \mathcal{P}$). The transformation of the left-hand side of (6.5) into its right-hand side will be carried out first for *vectors* and then extended to *tuples* as defined in the model.

In order to express the outcome of the calculations at the generic functor level (6.6), it will be necessary to extend the model to support generic functions. We will experiment with higher-order polymorphism and constructor classes as a means of adding genericity to standard Haskell and showing a simple, alternative way of extending the model's expressive power.

6.2 Pure Relation Level

The calculation which is sketched below is a point-free version of the conventional relational technique which extracts functional dependences by scheme decomposition [RO97]:

$$\begin{array}{ll} & \mathcal{P}(A \times B \times C \times D)_{\phi} \qquad (6.7) \\ \cong & \left\{ \begin{array}{lcl} rf_1 & = & \mathcal{P}(assocr4) \\ af_1 & = & \mathcal{P}(unassocr4) \end{array} \right\} \\ & \mathcal{P}(A \times (B \times C \times D))_{\phi} \qquad (6.8) \\ \leq & \left\{ \begin{array}{lcl} rf_2 & = & collect \\ af_2 & = & discollect \end{array} \right\} \\ & A \rightharpoonup \mathcal{P}(B \times C \times D) \qquad (6.9) \\ \cong & \left\{ \begin{array}{lcl} rf_3 & = & id \rightharpoonup extld3 \cdot \mathcal{P}(shiftr3) \\ af_3 & = & id \rightharpoonup \mathcal{P}(unassocl \cdot swap) \cdot lstrS \end{array} \right\} \\ & A \rightharpoonup (D \times \mathcal{P}(B \times C)) \qquad (6.10) \\ \leq & \left\{ \begin{array}{lcl} rf_4 & = & pfunzip \\ af_4 & = & uncurry\, pfzip \end{array} \right\} \\ & (A \rightharpoonup D) \times (A \rightharpoonup \mathcal{P}(B \times C)) \qquad (6.11) \\ \leq & \left\{ \begin{array}{lcl} rf_5 & = & id \times discollect \\ af_5 & = & id \times collect \end{array} \right\} \end{array}$$

$$
\begin{array}{ll}
 & (A \rightharpoonup D) \times \mathcal{P}(A \times (B \times C)) \qquad (6.12) \\
\cong & \left\{ \begin{array}{lcl} rf_6 & = & id \times \mathcal{P}(unassocl) \\ af_6 & = & id \times \mathcal{P}(assocr) \end{array} \right\} \\
 & (A \rightharpoonup D) \times \mathcal{P}(A \times B \times C) \qquad (6.13)
\end{array}
$$

The injective function $collect :: \mathcal{P}(A \times B) \rightarrow (A \rightharpoonup \mathcal{P}B)$ and its left-inverse $discollect$ are defined in module `Pfun_Impl.hs` (see 3.6 in section 3.3.2).

The calculation also involves $lstrS :: A \times \mathcal{P}B \rightarrow \mathcal{P}(A \times B)$, known as the *left-strength* [4] of the power-set functor, and its inverse $extlS$. Finally, $pfzip, pfunzip$ the finite mapping counterparts of $zip, unzip$ [BdM97] are defined in section 3.3.2.

6.3 Going Polytypic

Let us now see how to generalize $\mathcal{P}$ in the calculation above to a generic (strong) functor T and how to model this in Haskell by using higher-order polymorphism and constructor classes.

Let us deal with the Haskell model first. To begin with, we need to extend the *strength* of the power-set functor to regular functors (class $NFunctor$). That is, we need to define the function that pairs all elements of a structure with one particular element. We define the following functions in Haskell that correspond to left and right polytypic strength:

```
prstr :: NFunctor f a (a,b) => (f a,b) -> f (a,b)
prstr(s,x) = nmap (split id (const x)) s

plstr :: NFunctor f a (a,b) => (b,f a) -> f (b,a)
plstr(x,s) = nmap (split (const x) id) s
```

6.3.1 Adding Ad-Hoc Genericity

The next step is to define the class $Poly$ whose operations include those required by the process of normalization (note the character "p" *(=polytypic)* prefixing in each function symbol):

```
class Poly t where

   pzipWith :: (a -> b -> c) -> t a -> t b -> t c

   punzip :: (NFunctor t (Maybe (a,b)) (Maybe a),
              NFunctor t (Maybe (a,b)) (Maybe b)) =>
                   t (a,b) -> (t a , t b)

   pzip :: (NFunctor t (a,b) a,
            NFunctor t (a,b) b) =>
```

[4] In general the strength of a functor F is a natural transformation from $(\mathsf{F}A) \times B$ to $\mathsf{F}(A \times B)$, which has useful links with the concept of *membership* [RBH96].

```
                (t a , t b) -> Maybe(t(a,b))

pflatten :: t a -> [a]

pelems :: (Eq a , NFunctor t a (Set a)) =>
                t a -> Set a

pdiscollect ::(Eq (t (Maybe (b,c))), Eq b,
            NFunctor t (b,Maybe c) (Maybe (b,c)),
            NFunctor t (Maybe c) (b,Maybe c),
            NFunctor t (Maybe c) (Maybe (b,c))
            NFunctor t (Maybe (b,c)) (b,c))  =>
                Pfun b (t (Maybe c)) -> t (b,c)

pcollect :: (NFunctor t (a,b) a,
             NFunctor t (a,b) (Maybe b), Eq a) =>
                t (a,b) -> Pfun a (t (Maybe b))

pextl ::   (NFunctor t (Maybe b,Maybe c) (Maybe b),
            NFunctor t (Maybe b,Maybe c) (Maybe c),
            NFunctor t (Maybe (b,c)) (Maybe b,Maybe c),
            Eq (Maybe b),
            NFunctor t (Maybe b) (Set (Maybe b))) =>
                t (Maybe (b,c)) ->  (b,t (Maybe c))
```

Some default implementations are provided as follows:

```
pelems  = Set . nub . pflatten

pdiscollect x = nmap g . (foldS (pzipWith (&)) u y)
    where y   = (nmap (( nmap plstr) . plstr) . mkr) x
          u = (the . rng .
                (id *-> (nmap (const Nothing)))) x
          g (Just a) = a

pcollect t   = Map (nmap (split id (flip extr t)) l)
    where  l = (nub . pflatten . (nmap fst) ) t

punzip = split (nmap fst) (nmap snd)

pextl = ((fromJust. (foldS (\/) Nothing) .
          pelems) >< id) .
          punzip
```

Some comments are on demand:

- *pcollect* resorts to the $Maybe$ data type [5] in order to introduce explicit $Nothing$

[5] That is, data type $1 + A$, which corresponds to the `Maybe` type constructor of the *Standard Prelude*.

values to fill in the empties in a generic T-structure. Because $Maybe$ is an instance of $Nfunctor$, it is also a T-structure.

- The function $pextl$ that extracts a value of a structure of pairs (the inverse of left-strength) is defined for structures with $Maybe$ components.
- Function $punzip$ takes a structure containing pairs and splits it into a pair of structures containing the first and the second component respectively. Function $pzip$ is a partial inverse of $punzip$: it takes a pair of structures and "zips" them together to *just* a structure of pairs if the two structures have the same shape, or to *Nothing* otherwise.
- The traditional function zip [BdM97]:

$$zip \quad :: \quad [a] \to [b] \to [(a, b)]$$

  combines two list and does not need the `Maybe` type in the result as the longer list can always be truncated. (In general such truncation is possible for all types that have a null-ary constructor, but not for all regular types.)
- $pzip$, $pzipWith$, and $pflatten$ are among the functions which have no default implementation. They have thus to be provided for every instance of the class.

The next step is to define some well-known functors as instances of this class. Some examples follow:

```
instance Poly [] where
        pflatten = id
        pzipWith = zipWith
        pzip (a, b) = Just (zip a b)

instance Poly Maybe where
        pflatten (Just a) = [a]
        pflatten (Nothing)= []
        pzipWith f (Just a)(Just b)  = Just (f a b)
        pzipWith f (Just a)(Nothing) = Nothing
        pzipWith f (Nothing)(Just b) = Nothing
        pzipWith f (Nothing)(Nothing)= Nothing
        pzip (Just a,Just b)  = Just (Just (a ,b))
        pzip (Just a,Nothing) = Nothing
        pzip (Nothing,Just b) = Nothing
        pzip (Nothing,Nothing)= Nothing

instance Poly Set where
        pflatten= flattenS
        pzipWith f (Set l1) (Set l2)=
                    Set (zipWith f l1 l2)
        pzip (a, b)
             | (card a) == (card b)  = Just (zipS a b)
             | otherwise             = Nothing
```

6.4 Polytypic Normalization by Calculation

We are now ready to reproduce the calculation between steps (6.7) and (6.13) now at polytypic level. The calculation goes as follows:

$$
\begin{array}{ll}
 & \mathsf{T}(A \times B \times C \times D) \\
\supseteq & \quad \left\{ \begin{array}{lcl} rf_1 & = & \mathsf{T}(assocr4) \\ af_1 & = & \mathsf{T}(unassocr4) \end{array} \right\} \\
 & \mathsf{T}(A \times (B \times C \times D)) \\
\subseteq & \quad \left\{ \begin{array}{lcl} rf_2 & = & pcollect \\ af_2 & = & pdiscollect \end{array} \right\} \\
 & A \rightharpoonup \mathsf{T}(1 + B \times C \times D) \\
\supseteq & \quad \left\{ \begin{array}{lcl} rf_3 & = & id \rightharpoonup \mathsf{T}(id + assocr \cdot shiftr3) \\ af_3 & = & id \rightharpoonup \mathsf{T}(id + unassocr \cdot swap) \end{array} \right\} \\
 & A \rightharpoonup \mathsf{T}(1 + (D \times (B \times C))) \\
\subseteq & \quad \left\{ \begin{array}{lcl} rf_4 & = & id \rightharpoonup pextl \\ af_4 & = & id \rightharpoonup ((\mathsf{T}\, lstr) \cdot lstr) \end{array} \right\} \\
 & A \rightharpoonup (D \times \mathsf{T}(1 + (B \times C))) \\
\subseteq & \quad \left\{ \begin{array}{lcl} rf_5 & = & pfunzip \\ af_5 & = & uncurry\, pfzip \end{array} \right\} \\
 & (A \rightharpoonup D) \times (A \rightharpoonup \mathsf{T}(1 + (B \times C))) \\
\subseteq & \quad \left\{ \begin{array}{lcl} rf_6 & = & id \times pdiscollect \\ af_6 & = & id \times pcollect \end{array} \right\} \\
 & (A \rightharpoonup D) \times \mathsf{T}(A \times (B \times C)) \\
\supseteq & \quad \left\{ \begin{array}{lcl} rf_7 & = & id \times \mathsf{T}(unassocr) \\ af_7 & = & id \times \mathsf{T}(assocr) \end{array} \right\} \\
 & (A \rightharpoonup D) \times \mathsf{T}(A \times B \times C)
\end{array}
$$

Example

Let us see a brief example of polytypic normalization of an arbitrary structure of tuples.

Consider the relation $assign$ [6] shown in Table 6.1:

$assign$	(FLIGHT	DAY	PILOT	GATE)
	112	6 June	Bosley	7
	112	7 June	Brooks	7
	203	9 June	Bosley	12

Table 6.1: The relation $assign$.

[6] This example is taken from [Mai83], page 98.

$FLIGHT, DAY$ is a key for assign, and the relation must also satisfy the FD

$$FLIGHT \longrightarrow GATE$$

To avoid violating the FD, every time an update is made, we have to scan the relation and update the gate number every place the flight number appears. We would like to change only one tuple. Furthermore, the flight number–gate number information is duplicated in the relation, thus making the data redundant.

Suppose that we model relations as pairs of relation schemes and "Leaf Trees" of products, that is:

$$RelationBT = (IdAttr \rightharpoonup AttrInfo) \times BTree(Value \times Value \times Value \times Value)$$

where the $BTree$ data type is defined in Haskell:

```
data BTree a = Empty | Node(a, (BTree a, BTree a))
```

We illustrate the process of normalization by showing the data transformation over the second projection of the data type (the generalized part):

```
Node ((Int 112,Date "7-june",String "Brooks",Int 7),
      (Node ((Int 112,Date "6-june",String "Bosley",Int 7),
             (Empty,Empty)),
       Node ((Int 203,Date "9-june",String "Bosley",Int 12),
             (Empty,Empty))
      )) :: BTree (Value,Value,Value,Value)
```

$$\downarrow \mathsf{T}(assocr4)$$

```
Node ((Int 112,(Date "7-june",String "Brooks",Int 7)),
      (Node ((Int 112,(Date "6-june",String "Bosley",Int 7)),
             (Empty,Empty)),
       Node ((Int 203,(Date "9-june",String "Bosley",Int 12)),
             (Empty,Empty))
      )) :: BTree (Value,(Value,Value,Value))
```

$$\downarrow pcollect$$

```
[Int 112
   -> Node (Just (Date "7-june",String "Brooks",Int 7),
            (Node (Just (Date "6-june",String "Bosley",Int 7),
                   (Empty,Empty)),
             Node (Nothing,(Empty,Empty))
            )) ,
 Int 203
   -> Node (Nothing,
            (Node (Nothing,(Empty,Empty)),
             Node (Just (Date "9-june",String "Bosley",Int 12),
                        (Empty,Empty))
            ))] :: Pfun Value (BTree (Maybe (Value,Value,Value)))
```

$\downarrow id \rightharpoonup \mathsf{T}(id + assocr \cdot shiftr3)$

```
[Int 112
   -> Node (Just (Int 7,(Date "7-june",String "Brooks")),
            (Node (Just (Int 7,(Date "6-june",String "Bosley")),
                   (Empty,Empty)),
             Node (Nothing,(Empty,Empty))
            )) ,
 Int 203
   -> Node (Nothing,
            (Node (Nothing,
                   (Empty,Empty)),
             Node (Just (Int 12,(Date "9-june",String "Bosley")),
                   (Empty,Empty))
            ))] :: Pfun Value (BTree (Maybe (Value,(Value,Value))))
```

$\downarrow id \rightharpoonup pextl$

```
[Int 112
   -> (Int 7,
       Node (Just (Date "7-june",String "Brooks"),
             (Node (Just (Date "6-june",String "Bosley"),
                    (Empty,Empty)),
              Node (Nothing,
                    (Empty,Empty)))
            )) ,
 Int 203
   -> (Int 12,
       Node (Nothing,
             (Node (Nothing,
                    (Empty,Empty)),
              Node (Just (Date "9-june",String "Bosley"),
                    (Empty,Empty))
            ))))] :: Pfun Value (Value,BTree (Maybe (Value,Value)))
```

$\downarrow pfunzip$

```
([Int 112 -> Int 7 , Int 203 -> Int 12],
 [Int 112 -> Node (Just (Date "7-june",String "Brooks"),
                   (Node (Just (Date "6-june",String "Bosley"),
                          (Empty,Empty)),
                    Node (Nothing,
                          (Empty,Empty))
                   )) ,
  Int 203 -> Node (Nothing,
                   (Node (Nothing,
```

```
                        (Empty,Empty)),
                  Node (Just (Date "9-june",String "Bosley"),
                        (Empty,Empty))
                ))
 ]) :: (Pfun Value Value,Pfun Value (BTree (Maybe (Value,Value))))
```

$$\downarrow id \times pdiscollect$$

```
([Int 112 -> Int 7 , Int 203 -> Int 12],
 Node ((Int 112,(Date "7-june",String "Brooks")),
       (Node ((Int 112,(Date "6-june",String "Bosley")),
              (Empty,Empty)),
        Node ((Int 203,(Date "9-june",String "Bosley")),
              (Empty,Empty))
       ))) :: (Pfun Value Value,BTree (Value,(Value,Value)))
```

$$\downarrow id \times \mathsf{T}(unassocr)$$

```
([Int 112 -> Int 7 , Int 203 -> Int 12],
 Node ((Int 112,Date "7-june",String "Brooks"),
       (Node ((Int 112,Date "6-june",String "Bosley"),
              (Empty,Empty)),
        Node ((Int 203,Date "9-june",String "Bosley"),
              (Empty,Empty))
       ))) :: (Pfun Value Value,BTree (Value,Value,Value))
```

After these calculations, we are better off, with respect to updates and redundancy, because we have the same information (relation $assign$) as a pair of structures.

6.4.1 Vectors Give Place to Tuples

To develop this section we recall that data type $Tuple$ is the basis for the Haskell model of database relations presented in chapter 3, (section 3.4.1) and that relations ($Relation$ data type) are sets of tu-ples sharing a common attribute scheme (the $SchemaR$ data-type).

For notation economy, let

$$Tuple \quad = \quad (\overline{n} \rightharpoonup \sum_{i \in \overline{n}} A_i)$$

For every $X \subseteq \overline{n}$, we will write $Tuple_X$ as a shorthand for $X \rightharpoonup \sum_{i \in X} A_i$. We want to show that, for $k \subseteq s \subseteq \overline{n}$, inequation

$$\mathcal{P}(Tuple_{\overline{n}})_{\phi_{k,s}} \quad \leq \quad (Tuple_k \rightharpoonup Tuple_{s-k}) \times \mathcal{P}(Tuple_{\overline{n}-(s-k)}) \quad (6.14)$$

holds, where invariant $\phi_{k,s}$ ensures that the substructure s contains a functional dependence where k is the corresponding key.

The calculation goes as follows:

$$
\begin{array}{ll}
 & \mathcal{P}(Tuple_{\overline{n}})_{\phi_{k,s}} \\
\cong & \quad \left\{ \begin{array}{rcl} rf_1\ k & = & \mathcal{P}(tnest\ k) \\ af_1 & = & \mathcal{P}(uncurry\ plus) \end{array} \right\} \\
 & \mathcal{P}(Tuple_{\,k} \times Tuple_{\overline{n}-k}) \\
\leq & \quad \left\{ \begin{array}{rcl} rf_2 & = & collect \\ af_2 & = & discollect \end{array} \right\} \\
 & Tuple_{\,k} \rightharpoonup \mathcal{P}(Tuple_{\overline{n}-k}) \\
\cong & \quad \left\{ \begin{array}{rcl} rf_3\ s & = & id \rightharpoonup (extlS \cdot \mathcal{P}(tnest\ s)) \\ af_3 & = & \mathcal{P}(uncurry\ plus) \cdot lstrS \\ & & [\text{ NB: } (\overline{n}-k) \cap s = s-k\ ] \end{array} \right\} \\
 & Tuple_{\,k} \rightharpoonup Tuple_{\,s-k} \times \mathcal{P}(Tuple_{\overline{n}-s}) \\
\leq & \quad \left\{ \begin{array}{rcl} rf_4 & = & pfunzip \\ af_4 & = & uncurry\ pfzip \end{array} \right\} \\
 & (Tuple_{\,k} \rightharpoonup Tuple_{\,s-k}) \times (Tuple_{\,k} \rightharpoonup \mathcal{P}(Tuple_{\overline{n}-s})) \\
\cong & \quad \left\{ \begin{array}{rcl} rf_5 & = & id \times discollect \\ af_5 & = & id \times collect \end{array} \right\} \\
 & (Tuple_{\,k} \rightharpoonup Tuple_{\,s-k}) \times \mathcal{P}(Tuple_{\,k} \times Tuple_{\overline{n}-s}) \\
\cong & \quad \left\{ \begin{array}{rcl} rf_6 & = & id \times \mathcal{P}(uncurry\ plus) \\ af_6\ k & = & id \times \mathcal{P}(tnest\ k) \\ & & [\text{ NB: } (\overline{n}-s) \cup k = \overline{n} - (s-k)\ ] \end{array} \right\} \\
 & (Tuple_{\,k} \rightharpoonup Tuple_{\,s-k}) \times \mathcal{P}(Tuple_{\overline{n}-(s-k)})
\end{array}
$$

In generalizing sets of tuples to a generic functor T, we reuse the $Poly$ class without changes. The calculation is a smooth generalization of the one above.

$$
\begin{array}{ll}
 & \mathsf{T}(Tuple_{\overline{n}})_{\phi_{k,s}} \\
\cong & \quad \left\{ \begin{array}{rcl} rf_1\ k & = & \mathsf{T}(tnest\ k) \\ af_1 & = & \mathsf{T}(uncurryplus) \end{array} \right\} \\
 & \mathsf{T}(Tuple_{\,k} \times Tuple_{\overline{n}-k}) \\
\leq & \quad \left\{ \begin{array}{rcl} rf_2 & = & pcollect \\ af_2 & = & pdiscollect \end{array} \right\} \\
 & Tuple_{\,k} \rightharpoonup \mathsf{T}(1 + Tuple_{\overline{n}-k}) \\
\cong & \quad \left\{ \begin{array}{rcl} rf_3\ s & = & id \rightharpoonup (pextl \cdot \mathsf{T}(1 + tnest\ s)) \\ af_3 & = & \mathsf{T}(1 + uncurry\ plus) \cdot \mathsf{T}(plstr) \cdot plstr) \\ & & [\text{ NB: } (\overline{n}-k) \cap s = s-k\ ] \end{array} \right\} \\
 & Tuple_{\,k} \rightharpoonup (Tuple_{\,s-k} \times \mathsf{T}(1 + Tuple_{\overline{n}-s})) \\
\leq & \quad \left\{ \begin{array}{rcl} rf_4 & = & pfunzip \\ af_4 & = & uncurry\ pfzip \end{array} \right\} \\
 & (Tuple_{\,k} \rightharpoonup Tuple_{\,s-k}) \times (Tuple_{\,k} \rightharpoonup \mathsf{T}(1 + Tuple_{\overline{n}-s}))
\end{array}
$$

$$
\begin{array}{ll}
 & \left\{ \begin{array}{lcl} rf_5 & = & id \times pdiscollect \\ af_5 & = & id \times pcollect \end{array} \right\} \\
\cong & (Tuple_{\,k} \rightharpoonup Tuple_{\,s-k}) \times \mathsf{T}(Tuple_{\,k} \times Tuple_{\,\overline{n}-s}) \\
 & \left\{ \begin{array}{lcl} rf_6 & = & id \times \mathsf{T}(uncurry\ plus) \\ af_6\ k & = & id \times \mathsf{T}(tnest\ k) \\ & & [\text{ NB: } (\overline{n}-s) \cup k = \overline{n} - (s-k) \text{ }] \end{array} \right\} \\
\cong & (Tuple_{\,k} \rightharpoonup Tuple_{\,s-k}) \times \mathsf{T}(Tuple_{\,\overline{n}-(s-k)})
\end{array}
$$

6.5 Summary

In this chapter, we have presented the process of normalization through decomposition, from a calculational approach. We have shown how the data can be "transformed" using appropriate morphisms (reification laws) in order to extract functional dependences.

Next we have sketched a discipline of generic normalization of inductive datastructures. XML document re-factoring is an area where this discipline could be applicable. (Think for instance of the inductive types as representing the generative grammar implicit in a DTD and of the tuples as representing the attributive contents of the elements themselves.)

The availability of Haskell $\leftrightarrow$ XML conversion tools (eg. HaXML, see `http://www.cs.york.ac.uk/fp/HaXml`)) provides a stimulus toward the direct use of our Haskell model in such a context. However — as discussed in this chapter — we need support for polytypism, or generic programming and, as has been noted here, this can be achieved in more than one way.

Chapter 7

Conclusions and Future Work

The research carried out in the work reported in this dissertation belongs to the intersection between *formal methods*, relational *database theory* and *functional programming*. In particular, the Haskell functional language has been used to animate an abstract model of the standard relational database calculus, written in the style of model-oriented formal specification.

Parametricity and genericity have been shown to make room for further extensions of the model, eventually leading to the use of Generic Haskell and to ad-hoc polymorphism. Besides animation, the (generic) functional model is further subject to formal reasoning and calculation, paving the way to a more generic formulation of the standard relational calculus. Our experiments have gone as far as formulating a generic version of mda, the OLAP multi-dimensional analysis functionality, and furthermore, suggesting a theory of data normalization which is more general than the standard relational database theory (of which it appears to be a particular case).

The topic of generic (or type-constructor parametric) programming has become a central one in this thesis. Our incursion in this field has shown that there is more than one way to achieve genericity, each with its merits and disadvantages. In this chapter we will draw some conclusions on this "hot topic" of today's computer programming discipline.

Despite their somewhat speculative nature, the results of this thesis are likely to be useful in pragmatic contexts such as, for example, the normalization (or re-engineering) of large and complex XML data, a topic described below in the prospect for future work. Prior to that, let us present some comments and conclusions about the work developed so far.

7.1 Overall Comments

In devising a new data model (or variants thereof) the most important aspect of the design is concerned with the semantic objects (or states) — *"those things we wish to talk about"*.

The objects modeled by data type definitions in our (standard or generic) database

models are *representationally abstract* mathematical domains. That is: we have abstracted from any implementation concerns to focus on what the user wishes to see. But the domain definitions sometimes define too little. It is, in general, not possible to express, in the form of domain equations, all the internal consistency constrains which usually must hold within and/or among sub-parts of objects. So data model integrity constraints must be expressed in the form of predicates called data type *invariants*. Invariants apply to objects of defined domains and are intended to hold for those objects which are said to be *well-formed*.

Many database design approaches all too often begin by explaining concrete syntax, trying to cover the domain of applicable commands as completely as possible, and failing, invariably, to explain, to any acceptable depth, the semantic domains. As a matter of fact, such denotational modeling of data models very soon brings one into interesting generalizations. The abstraction level of declarative languages, such as Haskell, relieves one from many unwanted clerical details, and thereby enables one to better exploit one's mental capability. Among the abstraction tools available in Haskell, monads deserve a special mention.

Monads provide us with a convenient notion of a *computation* or *effect*. By using them we can enrich our computational model by distinguishing between the values and the effects. For example, two programs that calculate the same answer, but generate different screen outputs, should certainly be considered different. This is hard to achieve in a setting where computations are viewed statically, as pure functions. By using monads, one can precisely — and generically — specify the desired level of distinction between data and computations. This balances the trade-off between impure and pure functional languages from the implementer's language point of view. On the one hand, pure languages such as Haskell benefit from the power of equational reasoning. On the other hand, many desired features seem to be very hard to implement without using impure constructs, such as error handling. Monads provide solutions that combine the best of both approaches. We should note that monads are not special programming language constructs — they are simply an example of a good data abstraction technique. One of the big advantages of monads, apart from the support for higher order functions, is that they almost don't impose restrictions to the underlying programming language environment. The monadic style is just a simple methodology that can capture individual properties while keeping the abstraction level appropriately high, thus saving us from being too concerned about technical details.

7.2 About Genericity

In brief, one can implement polytypic programs in three different ways (recall section 2.4):

- using a universal data type;
- using higher-order polymorphism and constructor classes;
- using a special syntactic construct.

In this work we have chosen to experiment with the last two alternatives. We started from using the Haskell 98 type class [1] mechanism in order to gain experience in defining generic functions. First we defined the *constructor class* "NFunctor" (see section 3.3.1, paragraph 3.3.1) in order to provide our model with a polytypic *mapping function*. The use of a multi-parameter class allowed us to make instance declarations that constrain the arguments type *on a per instance basis*.

Then, we extended the *strength* of the power-set functor to regular functors (class $NFunctor$) by defining *prstr* and *plstr* functions in Haskell (these correspond to left and right polytypic strength). Finally, we defined a *constructor class* — $Poly$ — whose operations include those required by the process of generic (polytypic) normalization.

The next step was to develop generic relational operators using a language with explicit syntactic constructs for defining polytypic functions. We first experimented with the POLYP [JJ97] system but soon realized that there was a serious shortcoming: only *unary* (regular) data types can be polytypically defined. We have overcome this problem by switching to Generic H∀SKELL , which implements a new approach to generic programming due to Hinze [Hin00b]. In particular, we experimented with generic abstractions which are introduced in [CL02]. It should be stressed that function *gmap* defined in the Map library of Generic H∀SKELL (reproduced in section 5.3), is the generic version of $fmap$ in the $Functor$ class, but not of our $nmap$ in the $NFunctor$ class. Should we have used *gmap* function to define "the mapping function" over Set data type, the resulting specialization could produce erroneous sets. Again, we need a context constraint: equality on the result type (see section 3.3.1, paragraph 3.3.1). We use function *gmap* in all cases to apply specific functions that "preserve" Set's invariant.

In this context our main conclusions are as follows:

- By using the Haskell 98 class system, programs become cluttered with instance declarations and type declarations become cluttered with contexts (see type declarations of poly class functions in section 6.3.1).

- Different instances declarations can constrain the argument type in different ways (see instance definition of $NFunctor$ for Set type in section 3.3.1).

- Multi-parameter type classes could be used to represent an arbitrary relation between types (see type of *prstr* and *plstr* functions in section 6.3).

- Generic definitions using generic abstractions are simple to write in Generic H∀SKELL and can be applied to Haskell 98 types of all kinds.

- We cannot compile a generic function without knowing to which data types it will be called. It is not possible to compile generic functions separately from the code that calls them.

[1] The term "type class" includes both the original Haskell 1.0 type system and the "constructor classes" introduced by Jones [Jon93].

7.3 Some Difficulties

The main difficulties in carrying out the work described in this dissertation have been found, basically, in the areas of computer science and mathematics: the lack of experience with functional programming on one hand, and some unfamiliarity with category theory on the other.

A paradigmatic example of evolution of our modeling originated in this lack of experience with these mathematical subjects is provided by the morphism associated with the *finite set* data type. This has undergone significant transformations, evolving from a first (naive) form:

```
foldS f u (Set s) = foldr f u s
```

— as instanced for instance in

```
card = foldS (\a -> succ) 0
```

— to the more sophisticated and useful form,

```
foldS g u  = hyloSet (either (const u) (uncurry g))
```

— now instanced in a more algebraic way, using the *either* combinator,

```
card = hyloSet (either (const 0) (succ . p2))
```

which includes the hylomorphism combinator

```
hyloSet ::(Either (Set a) (a,b) -> b) -> Set a -> b
hyloSet a = a . ( rec (hyloSet a)) . outSet where
            rec f = id -|- (id >< f)
```

which unveils the polynomial structure of its base functor in a much clear way.

7.4 Prospects for Future Developments

The research theme proposed in this M.Sc. thesis project is expected to evolve toward a Ph.D. project in a natural way. Based on some intuitions that has been gathered from the experimental work of this M.Sc. thesis, we are ready for moving from notation to calculus, as sketched in *e.g.* [BH95], and from there to concrete applications.

The target of generalizing the whole relational calculus will entail complex reasoning. Fortunately, there is now a standard and solid approach to program calculation which seems to have the potential to accommodate this project, and this is the *relational theory of data types* [BH93].

Some specific topics for future work are listed below.

7.4.1 Toward Generic Normalization Theory

A lot of work remains to be done in the evolution of standard *normalization theory* toward genericity which is suggested in this thesis. In particular, a proper formalization of the intuitions presented in section 6 is on demand. This includes the interplay between data type *construction* and data type *constraining* (cf. data type *invariants*), which still requires proper formalization in the point-free style.

It should be stressed that — as can be found in our calculations and is pointed out in [Oli90] — many transformation rules are invariant-sensitive. The "pull-back approach" of [Oli98] is an attempt toward point-free invariant reasoning which proves to be insufficient in practice. Currently, a more promising approach can be exploited [Oli04]: that of modeling constrained data types by *co-reflexive* relations and performing the reasoning in the (point-free) *relational theory of data types* [BH93, BdM97, Bac00]. The latter reference is a very lucid account of program calculation pointing toward an effective (and desirable) "algebrization of logics".

7.4.2 Toward Generic Multi-Dimensional Data Processing

In chapter 4, we introduced a set of operations that extended the functionality provided by the relational data model proposed in chapter 3 to support OLAP-based applications. We worked under the assumption that relational systems can model N-dimensional data as a relation with N-attribute domains.

Function mda, which creates a table with an aggregated value indexed by a set of attributes, operates on relations and produces relations. It could be composed with the basic operators of relational algebra to build other OLAP operators in order to provide constructs such histograms, cross-tabulations, subtotals, roll-up and drill-down. For instance, mda could be used to compute the following table (roll up using totals report):

PartYearSales		*Year*			
		1996	1997	...	Total
Part	PC	320	455	...	1256
	Inkjet	298	450	...	987
	⋮	⋮	⋮	⋮	⋮
	Total	1788	1450	...	—

Table 7.1: Sales Roll Up by Part by Year (applied to the input relation *RSALES* of Example 4.1.

The rightmost column corresponds to the output of the expression:

```
mdaR (Set["Part"]) "Sale" fadd f0 RSALES
```

while the bottom row could be computed using:

```
mdaR (Set["Year"]) "Sales" valadd val0 r1olap
```

Composition (and others categorical combinators) provides a powerful tool to compose operators and allows for complex multidimensional queries to be built.

In Chapter 5, section 5.4 we developed the generic counterparts of OLAP operations. We have presented the basic ideas of a comprehensive, generic conceptual model for multidimensional data processing, with symmetric treatment of dimensions and measures and the ability to pose powerful ad-hoc queries through a simple and declarative interface.

The next step, which goes beyond the scope of the present work, should include the development of a complete algebra and a rigorous calculus based upon the algebraic operators of the model.

7.4.3 Free Theorems for Generic Data Processing

As stressed earlier on, the topic of generic (or type-constructor parametric) programming is central to this thesis. Why such an emphasis on parametricity and genericity?

Surely there is some intellectual reward and conceptual economy in designing solutions to specific problems as the *customization* of *generic* ones. However, there is more. In a famous paper entitled *Theorems for free!* [Wad89] Philip Wadler writes:

> *From the* type *of a polymorphic function we can derive a* theorem *that is satisfy.* (...) *How useful are the theorems so generated? Only time and experience will tell* (...)

This result is a rewording of *Reynolds abstraction theorem* on parametric polymorphism which can be found in a remarkably elegant point-free formulation in [BB03]. This paper and others (eg. [OR04]) present examples of the use of this theorem to calculate useful *fusion-laws* involving polymorphic types. So, it is to be expected that every of our generic, polymorphic models of relational and OLAP operators will enjoy one such fusion-law, the corollaries of which — if already known — will be thus proved "for free", and — if still unknown — will add to the theory behind such important areas of computing.

7.4.4 Application Area: XML Re-factoring

The widespread adoption of XML as an universal, textual data-definition format is regarded as a positive step in computing in the last decade. However, such a positive step may be endangered by too much freedom and informality in designing and/or using such XML-dialects, as is pointed out in two recent attempts to formalize XML-based domain-specific languages, namely UIML [Fer04] and GML [Hen04].

As a consequence, it can be anticipated that many XML documents are likely to become legacy code sooner than expected. One of the deficiencies found in the works just mentioned has to do with lack of referential integrity and lack of constraints — two of our concerns in this thesis.

The discipline of generic normalization of inductive data-structures will surely be applicable to such XML document re-factoring — think, as we have already stressed at the end of chapter 6, of the inductive types as representing the generative grammar implicit in a DTD and of the tuples as representing the attributive contents of the elements themselves.

The availability of Haskell-based *grammar-engineering* tools such as STRAFUNSKI [LV03] adds to our interest and curiosity in this line of applied research.

Appendix A

Haskell Source Code

A.1 Cat_Impl.hs

```
-------------------------------------------------------------
-- RDB model: Category Theory Support
-------------------------------------------------------------
module Cat_Impl where
import Types
infix 4  ><
infix 4  -|-

-- (1) Product ----------------------------------------------
(><) :: (a -> b) -> (c -> d) -> (a,c) -> (b,d)
(f >< g)(a,b) = (f a, g b)

split :: (a -> b) -> (a -> c) -> a -> (b,c)
split f g x = (f x, g x)

-- (2) Coproduct --------------------------------------------
(-|-) :: (a -> b) -> (c -> d) -> Either a c -> Either b d
(f -|- g) (Left  a) = Left (f a)
(f -|- g) (Right b) = Right (g b)

-- (3) Exponentiation ---------------------------------------
expn :: (a -> b) -> (c -> a) -> c -> b
expn f = \g -> f . g

-- (4) Others -----------------------------------------------
-- McCarthy's conditional
cond :: (a -> Bool) -> (a -> b) -> (a -> b) -> a -> b
cond p f g = (either f g) . (grd p)
grd :: (a -> Bool) -> a -> Either a a
grd p x = if p x then Left x else Right x
```

```
-- (5) Error Monad ------------------------------------------
data Error a = Err String | Ok a deriving (Show, Eq)

instance Monad Error where
    return b = Ok b
    (Err e) >>=  f =  Err e
    (Ok  a) >>=  f = f a

mfold :: Monad a => (b -> c -> a c) -> a c -> [b] -> a c
mfold f k [] = k
mfold f k (h:t) = do { b <- mfold f k t ;
                       f h b }
mfoldS :: Monad a => (b -> c -> a c) -> a c -> Set b -> a c
mfoldS f u (Set l) = mfold f u l

lift   :: Monad a => (b -> c) -> b -> a c
lift f n = return(f n)

lift2  :: Monad a => (b -> c -> d) -> b -> c -> a d
lift2 f n m = return (f n m)

-- (6) Monadic extensions  ----------------------------------
--    Kleisli composition
(.!) :: Monad a => (b -> a c) -> (d -> a b ) -> d -> a c
(f .! g) x  = (g x) >>= f

(!):: Monad a => (b -> a c) -> a b -> a c
f ! x = x >>= f

-- (7) Basic algebraic signatures ----- ----------------------
class Monoid a where
    (&) :: a -> a -> a
instance Monoid (Maybe a) where
        (Just a) & Nothing = Just a
        Nothing & (Just a) = Just a
        Nothing & Nothing = Nothing
        (Just a) & (Just b) = Nothing

class BoolAlg a where
    (\/) :: a -> a -> a
    (/\) :: a -> a -> a
instance BoolAlg Bool where
        (\/)  = (||)
        (/\)  = (&&)
instance BoolAlg (Maybe a) where
        (Just a) \/ Nothing  = Just a
        Nothing  \/ (Just a) = Just a
        Nothing  \/ Nothing  = Nothing
        (Just a) \/ (Just b) = Nothing
```

```
-- (8) Basic bifunctors ---------------------------------------
class BiFunctor f where
      bmap :: (a -> c) -> (b -> d) -> (f a b -> f c d)
instance BiFunctor Either where
    bmap f g = f -|- g
instance BiFunctor (,) where
    bmap f g (a,b) = (f >< g)(a,b)

-- (9) New Functor --------------------------------------------
class NFunctor f a b where
     nmap :: (a -> b) -> (f a -> f b)
instance NFunctor [] a b where
     nmap=fmap
instance NFunctor Maybe a b where
    nmap f Nothing  = Nothing
    nmap f (Just x) = Just (f x)
instance NFunctor IO a b where
    nmap f x = x >>= (return . f)

-- Strength of a functor
prstr ::  NFunctor a b (b,c) => (a b,c) -> a (b,c)
prstr(s,x)      = nmap (split id (const x)) s
 plstr :: NFunctor a b (c,b) => (c,a b) -> a (c,b)
plstr(x,s)      = nmap (split (const x) id) s

-- (10) Invariants --------------------------------------------
class CData a where
    inv :: a -> Bool
    inv a = True
    inv' :: a -> Error a
    inv' a = if (inv a) then Ok a
                        else Err "Invariant violation"
instance (CData a, CData b) => CData (a,b) where
    inv(a,b) = (inv a) && (inv b)
instance (CData a, CData b) => CData (Either a b) where
    inv (Left a) = inv a
    inv (Right b) = inv b
instance CData a => CData [a] where
    inv = all inv
 -- (11) Natural isomorphisms ----------------------------------
swap :: (a,b) -> (b,a)
swap (a,b) = (b,a)
shiftr3:: (a,b,c) -> (c,a,b)
shiftr3 (a,b,c) = (c,a,b)
assoc = split ( fst . fst ) (split ( snd . fst ) snd )
assocr (a,b,c) = (a,(b,c))
unassocr (a,(b,c)) = (a,b,c)
assocl (a,b,c) = ((a,b),c)
unassocl ((a,b),c) = (a,b,c)
```

A.2 Set_Impl.hs

```
-------------------------------------------------------------
-- RDB model: Sets Implementation
-------------------------------------------------------------
module Set_Impl  where
import Types     --Datatype definition
import List
import Cat_Impl
import Exp

--(1) Hylo ------------------------------------------------
outSet:: Set a -> Either (Set a) (a,Set a)
outSet (Set [])= i1 (Set [])
outSet  s = i2 (gets s)

inSet:: Eq a => Either (Set a) (a, Set a)-> Set a
inSet= either (const emptyS) uputs

hyloSet ::(Either (Set a) (a,b) -> b) -> Set a -> b
hyloSet a = a . ( rec (hyloSet a)) . outSet where
            rec f = id -|- (id >< f)

-- (2) Instances ------------------------------------------
instance Eq a => Eq (Set a)
        where s == r = s <= r && r <= s

instance Eq a => Ord (Set a)
        where (Set s) <= r = all (-| r) s

instance Eq a => BoolAlg (Set a) where
        r /\ s = filterS id (-| r) s
        (\/)  = foldS puts

instance (Eq b) => NFunctor Set a b where
 nmap f =
    hyloSet (either (const (Set[])) (uncurry puts . (f >< id)))

instance  Eq a => CData (Set a) where
   inv (Set l) = length l == card(elems l)
   inv' s      = if (inv s) then Ok s
                            else Err "Set invariant violation"

-- (3) Total Functions ------------------------------------
-- Membership
ins :: Eq a => a -> Set a -> Bool
ins a (Set s) = elem a s
nins :: Eq a => a -> Set a -> Bool
nins a (Set s)   = not (elem a s)
```

```
-- Folding
foldS ::(a -> b -> b) -> b -> Set a -> b
foldS g u = hyloSet (either (const u) (uncurry g))

-- ZF filter
filterS :: Eq a => (b -> a) -> (b -> Bool) -> Set b -> Set a
filterS f p=
   foldS(unions . cond p (sings . f) (const emptyS) )(emptyS)

-- constructors
sings :: a -> Set a
sings a = Set [a]

-- Union-intersection-diff-inclusion
unions :: (Eq a) => Set a -> Set a ->Set a
unions = (\/)
inters :: (Eq a) => Set a -> Set a -> Set a
inters = (/\)
dunion :: Eq a => Set (Set a) -> Set a
dunion = foldS (unions) (Set [])
diffs :: (Eq a) => Set a -> Set a -> Set a
diffs r s= filterS id (-|| s) r
incls :: (Eq a) => Set a -> Set a -> Bool
incls     r       s    =(diffs r s) == Set []

-- elems
elems :: Eq a => [a] -> Set a
elems = foldr puts (Set [])

-- cardinal
card:: Set a -> Int
card = hyloSet (either (const 0) (succ . p2))

-- Power set strength
lstrS :: (a, Set b) -> Set (a,b)
lstrS (a, Set b) = Set [(a,c) | c<-b ]
rstrS :: (Set b, a) -> Set (b,a)
rstrS (Set b, a) = Set [(c,a) | c<-b ]
 -- extract the leftmost element (inverses of strength)
extlS :: (Eq a, Eq b) =>  Set (a,b) -> (a,Set b)
extlS = (the >< id) . unzipS

--  Zip's-unzip
zipS :: Set a -> Set b -> Set (a,b)
zipS (Set a) (Set b)= Set (zip a b)
zipWithallS:: Set a -> Set b -> Set (a,Set b)
zipWithallS (Set a)  b = Set ([(x,b) | x <- a ])
unzipS:: (Eq b, Eq a) => Set (a,b) -> (Set a,Set b)
unzipS = split (nmap p1) (nmap p2)
```

```
-- flatten
flattenS :: Set a -> [a]
flattenS =  hyloSet (either (const [])(uncurry (:)))

flatr:: (Eq a, Eq b, Eq c) => Set (a,(b,c))-> Set (a,b,c)
flatr = nmap (unassocr)

flatl:: (Eq a, Eq b, Eq c) => Set ((a,b),c)-> Set (a,b,c)
flatl= nmap (unassocl)

-- Others
puts :: Eq a => a -> Set a -> Set a
puts a (Set s) = Set (nub (a : s))

uputs::Eq a => (a,Set a) -> Set a
uputs = uncurry(puts)

allS :: (a -> Bool) -> Set a -> Bool
allS p s = ((foldS (/\) True) . (nmap p) ) s

prods:: Set a-> Set a -> Set (a,a)
prods (Set a) (Set b)= Set([(x,y) | x <- a,  y <- b])

ltos:: [a]->Set a
ltos l    =Set l

stol:: Set a-> [a]
stol (Set s)  =s

-- (4) Partial Functions -------------------------------------
gets :: Set a -> (a,Set a)
gets (Set (a:x))= (a,Set x)

gets' :: Set a -> Error (a,Set a)
gets' (Set []    )= Err "Empty Set - Nothing to get"
gets' (Set (a:[]))= Ok (a,Set [])
gets' (Set (a:x) )= Ok (a,Set x)

the :: Set a -> a
the (Set [a]) = a

the' :: Set a -> Error a
the' (Set[]  ) = Err "Empty Set - There isn't the"
the' (Set [a]) = Ok a

-- (5) Notation Shortcuts ------------------------------------
a -| s = ins a s
a -||s = nins a s
r \< s = incls r s
```

A.3 Pfun_Impl.hs

```
-----------------------------------------------------------------
-- RDB model: Partial Functions Implementation
-----------------------------------------------------------------
module Pfun_Impl where
import Cat_Impl
import List
import Combinadores
import Types
import Set_Impl

-- (1) Instances ------------------------------------------------
instance (Eq a, Eq b) => Ord (Pfun a b) where
        f <= g = (dom f) <= (dom g) && coherent f g
instance (Eq a, Eq b) => Eq (Pfun a b) where
        f == g = f <= g && g <= f
instance (Eq a, Eq b, Monoid b) => Monoid (Pfun a b) where
        (&) = monpf (&)
instance (Eq a, Eq b) => CData (Pfun a b) where
   inv (Map pf) =inv (Set pf)&& fdp(Set pf)
   inv' f = if (inv f) then Ok f
               else Err "Partial function invariant violation"

-- (2) Total Functions ------------------------------------------
dom :: (Eq a) => Pfun a b -> Set a
dom (Map f) = (Set . nub . fst . unzip) f
rng :: (Eq b) => Pfun a b -> Set b
rng (Map f ) = (Set . nub . snd . unzip) f

putpf :: (Eq a, Eq b) => (b,a) -> Pfun b a -> Pfun b a
putpf p (Map s) = if (coherent (singpf p) (Map s))
                  then Map (union [p] s)
                  else (Map[])

-- functional dependency
fdp :: (Eq a, Eq b) => Set (b,a) -> Bool
fdp (Set[])=False
fdp  s = ((<= sings 1). rng . (id *-> card) . collect) s

--constructor
singpf:: (a,b) -> Pfun a b
singpf p = Map [p]

-- coherence
coherent  :: (Eq a, Eq b) => Pfun b a -> Pfun b a -> Bool
coherent f g = all(uncurry(==)) r where Set r= rng (pfzip f g )
coherents :: (Eq a, Eq b) => Pfun a b -> Pfun a b -> Set a
coherents f g = diffs ((dom f) \/ (dom g)) (incoherents f g)
```

```
incoherents :: (Eq a, Eq b) => Pfun a b -> Pfun a b -> Set a
incoherents f g = bpfFalse ((id *-> (uncurry(==)))(pfzip f g))

-- zip-unzip
pfzip :: Eq b => Pfun b c -> Pfun b d -> Pfun b (c,d)
pfzip (Map f)(Map g) =
   Map [ a |-> (b,c) | (a,b) <- f , (d,c) <- g, a==d ]

pfunzip ::(Eq a, Eq b, Eq c) =>
          Pfun b (c,a) -> (Pfun b c,Pfun b a)
pfunzip = split (id *-> p1) (id *-> p2)

pfzipWith :: (Eq a, Eq b) => i
             (c -> d -> a) -> Pfun b c -> Pfun b d -> Pfun b a
pfzipWith f a b = (id *-> (uncurry f))(pfzip a b)

--Overwrite operator
plus :: (Eq b, Eq a) => Pfun a b -> Pfun a b -> Pfun a b
plus f g = (restn (dom g) f ) \*/ g

--positive- negative restriction
restp :: Eq a => Set a -> Pfun a b -> Pfun a b
restp (Set s) (Map f)= Map ([ p | p <- f , ( elem (fst p) s ) ])

restn :: Eq a => Set a -> Pfun a b -> Pfun a b
restn (Set s) (Map f)=
      Map ([ p | p <- f , not ( elem (fst p) s ) ])

-- Folding
foldPf :: ((a,b) -> c -> c) -> c -> Pfun a b -> c
foldPf f u (Map[]) = u
foldPf f u (Map s) = foldr f u s

-- "reverses" f.
pfinv :: (Eq a, Eq b) => Pfun a b -> Pfun b (Set a)
pfinv  f  = (collect . (nmap swap) . mkr) f


--make relation
mkr :: Pfun a b -> Set (a,b)
mkr (Map f) = Set f

--apply total function
aplft :: (Eq a, Eq b) => Pfun a b -> a -> b
aplft f = the . rng . (flip restp f) . sings

-- rename
renpf :: (Eq a, Eq b) => Pfun a a -> Pfun a b -> Pfun a b
renpf r f=plus (dom r <-: f) (compf f ((id *-> the) (pfinv r)))
```

```
-- compound total functions
compf :: Eq a => Pfun a b -> Pfun c a -> Pfun c b
compf (Map f) (Map g) =
         Map [ a |-> c | (a,b) <- g, (b',c) <- f, b'==b ]

-- Bifunctor
bmapPf :: (Eq a, Eq b) =>
          (c -> b) -> (d -> a) -> Pfun c d -> Pfun b a
bmapPf f g  h  = foldPf (\p -> putpf ((f >< g) p)) (Map []) h

--apply partial function
aplpf :: Eq a => Pfun a b -> a -> Maybe b
aplpf (Map f) a = lookup a f

--Union operator
unionpf :: (Eq b, Eq a) => Pfun a b -> Pfun a b -> Pfun a b
   unionpf f g = f \*/ g

-- filtering
filterPf::(Eq a, Eq b) => ((b,a) -> Bool)-> Pfun b a-> Pfun b a
filterPf p=foldPf (plus . cond p singpf (const emptyPf))(Map[])

-- Others
collect :: (Eq a, Eq b) => Set (a,b) -> Pfun a (Set b)
collect (Set r) =
 Map (nub [ a |-> Set[ b | (c,b) <- r , a==c ] | (a,b)  <- r ])
discollect:: Pfun a (Set b) -> Set (a,b)
discollect (Map f) = Set [(a,b) | (a, (Set s)) <- f , b <-s ]

bpfTrue :: (Eq (Set a), Eq a) => Pfun a Bool -> Set a
bpfTrue g = let f = pfinv g
   in if True -| (dom f) then aplft f True else (Set[])
bpfFalse :: (Eq (Set a), Eq a) => Pfun a Bool -> Set a
bpfFalse g =let f = pfinv g
   in if False -| (dom f) then aplft f False else (Set[])

monpf :: (Eq b, Eq a) =>
           (b -> b -> b) -> Pfun a b -> Pfun a b ->  Pfun a b
monpf f m n = plus m (plus n (pfzipWith f m n))

-- (3) Partial Functions ------------------------------------
aplpf' :: Eq a => Pfun a b -> a -> Error b
aplpf' (Map f) a=case (lookup a f) of
                  Nothing -> Err "Apply Empty Partial Function"
                  Just x  -> Ok x
unionpf'::(Eq b, Eq a) => Pfun a b->Pfun a b-> Error (Pfun a b)
unionpf' f g = case (coherent f g) of
         True -> Ok (f \*/ g)
         False -> Err "map union: incompatible maps"
```

```
-- (4) Auxs. for Olap operations ------------------------------
tnest :: Eq a => Set a -> Pfun a b -> (Pfun a b,Pfun a b)
tnest s f = (s <: f,s <-: f)
tcollect :: (Eq a, Eq b) =>
     Set a -> Set (Pfun a b) -> Pfun (Pfun a b) (Set (Pfun a b))
tcollect s t = collect (nmap (tnest s) t)
tot2 :: (a -> b -> b) -> b -> Set (c,a) -> b
tot2 f b    r= foldS (curry (uncurry f . (p2 >< id))) b r
ttot :: (Eq a, Eq b) =>
        b -> (a -> a -> a) -> a -> Set (Pfun b a) -> Pfun b a
ttot a f u s = Map [ a |-> (tot2 f u y) ]
   where y =nmap ((id >< (get u a)) . swap . tnest (sings a)) s

-- get totalizes a partial function (f) with a default value (u)
get :: Eq a => b -> a -> Pfun a b ->  b
get u a f = aux (aplpf' f a)
            where aux (Ok b) = b
                  aux (Err s) = u

-- (5) Notation shortcuts -------------------------------------
x |-> y = (x,y)
s <:  f = restp s f
s <-: f = restn s f
g *-> f = bmapPf g f
r \*/ s = foldPf putpf r s
a <.  f = aplft f a
```

A.4 Rel_Impl.hs

```
-------------------------------------------------------------------
-- RDB model: Relations Implementation
-------------------------------------------------------------------

module Rel_Impl where
import Types
import Cat_Impl
import Set_Impl
import Pfun_Impl

-- (1) Instance definitions ----------------------------------

instance Eq Relation where
    r1==r2 =
      (schema(r1)==schema(r2)) && (tuples(r1)==(tuples(r2)))

instance Ord Relation where
    r1<=r2 =
      (schema(r1)<=schema(r2)) && (tuples(r1)<=(tuples(r2)))
```

```
instance CData Relation where
 inv (Rel s t) = inv s && inv t && (m /= emptyPf ) &&
   relSchOk (Rel s t) && fdpOkv (Rel s t)
   where
     m=foldS (\x y ->if (coherent x y) then unionpf x y else y)
              emptyPf (nmap (id *-> valType) t)
     relSchOk r = m <= (id *-> (valType . defaultV)) (schema r)
     fdpOkv (Rel s t) = fdp(nmap (tnest (getKeyAtts s)) t)
 inv' (Rel s t) =
  do {inv' s
       `xtherwise` "Relation schema is not a partial function";
      inv' t
       `xtherwise` "Relation tuples set is not valid";
      m <-mfoldS unionpf' (Ok emptyPf)(nmap (id *-> valType) t)
       `xtherwise` "tuple schemas are not mutually compatible";
      check (relSchemaOk m) (Rel s t)
       "At least one tuple type does not match relation schema";
      check fdpOk (Rel s t)
         "The key-property is not valid in the relation"
      }
  where
   relSchemaOk m r= m<=(id *-> (valType . defaultV)) (schema r)
      fdpOk (Rel s t) = fdp(nmap (tnest (getKeyAtts s)) t)

check p v s = if (p v) then (Ok v) else Err s

xtherwise (Ok  x) s = Ok x
xtherwise (Err _) s = Err s

-- (2) Total Functions ----------------------------------------
--Observers
getMSch :: Relation -> SchemaR
getMSch (Rel s r )=s
getKeyAtts :: SchemaR -> Set IdAttr
getKeyAtts = bpfTrue . (id *->ifKey)

valType :: Value -> String
valType (Int _) = "Int"
valType (String _) = "String"
valType (Date _) = "Date"
valType (Time _) = "Time"

-- get all attributes actually in relation
actAtts :: Relation -> Set IdAttr
actAtts = dunion . (nmap dom) . tuples

-- get all attributes declared in relation schema
dclAtts :: Relation -> Set IdAttr
dclAtts = dom . schema
```

```
-- Basic Relation operators
-- Union
unionR ::   Relation -> Relation -> Relation
unionR (Rel s1 t1) (Rel s2 t2) =
        let a = coherents s1 s2
            t1' = nmap (a<:) t1
            t2' = nmap (a<:) t2
        in (Rel ((a <: s1) \*/ (a <: s2)) (t1' \/ t2'))

-- Intersection
interR ::   Relation -> Relation -> Relation
interR (Rel s1 t1) (Rel s2 t2) =
        let a = coherents s1 s2
            t1' = nmap (a<:) t1
            t2' = nmap (a<:) t2
        in (Rel ( dom(a <: s1) <: (a <: s2) ) (t1' /\ t2'))

-- Difference
diffR ::   Relation -> Relation -> Relation
diffR (Rel s1 t1) (Rel s2 t2) =
        let a = coherents s1 s2
            t1' = nmap (a<:) t1
            t2' = nmap (a<:) t2
        in (Rel (dom(a <: s2) <-: (a <: s1) ) (diffs t1' t2'))

-- Projection
projectR :: Set IdAttr -> Relation -> Relation
projectR a (Rel s t) = (Rel ( a <: s ) (nmap (a<:) t))

-- Selection
selectR :: Pfun IdAttr Value  ->   Relation -> Relation
selectR i (Rel s t) =  (Rel s (filterS id (coherent i) t))

-- Natural join
natjoinR ::   Relation ->   Relation -> Relation
natjoinR (Rel s1 t1) (Rel s2 t2) =
 let t=(nmap ( \x -> (nmap (split (const x) id)
                     (filterS id (coherent x) t2)))t1)
     ta=(foldS (unions) (Set[]) ) t
 in (Rel ( (dom(s1) /\ dom(s2)) <-: ( s1 \*/ s2))
   (nmap (\x ->(dom(s1)/\dom(s2)) <-: (plus (p1 x)(p2 x))) ta))


-- Cartesian Product
cproductR :: Relation -> Relation -> Relation
cproductR (Rel s1 t1) (Rel s2 t2) =
   let ta= nmap (split (const t1) id) t2
       tb= nmap(\(l,r)-> nmap(split id (const r)) l) ta
   in Rel (s1\*/s2) (nmap(uncurry plus) (dunion tb))
```

```
-- Renaming
renameR :: Pfun IdAttr IdAttr -> Relation -> Relation
renameR f (Rel s t)= (Rel (renpf f s) ( nmap (renpf f) t))

-- Divide
divideR :: Relation -> Relation -> Relation
divideR r s =
  let y= diffs (dom(schema r)) (dom(schema s))
      x= nmap (\t -> projectR y (selectR t  r)) (tuples s)
      u = projectR y  r
  in foldS interR u x

-- (3) Partial Functions ------------------------------------
valType' :: Value -> Error String
valType' (Int _   ) = Ok "Int"
valType' (String _) = Ok "String"
valType' (Date _  ) = Ok "Date"
valType' (Time _  ) = Ok "Time"
valType'  _         = Err "unknown ValueType"

-- (4) Olap operators ---------------------------------------
mdaR:: Set IdAttr ->IdAttr-> (Value->Value->Value) ->Value ->
             Relation-> Relation
mdaR s a f u  t =
 Rel ((unions s (sings a))<:(schema t))
     (nmap (uncurry plus)(mkr y))
  where y = (id *-> (ttot a f u)) x
        x = tcollect s (tuples t)
```

A.5 RDB_Impl.hs

```
-----------------------------------------------------------------
-- RDB model: Relational Database Implementation
-----------------------------------------------------------------
module RDB_Impl where
import Cat_Impl
import Types
import Set_Impl
import Pfun_Impl
import Rel_Impl

-- (1) Instance definitions ---------------------------------
instance Eq RDB where
        db1==db2 = (relations(db1)==relations(db2))
instance Ord RDB where
        db1<=db2 = (relations(db1)<=relations(db2))
instance CData RDB where
    inv (RDB f) = ( allS inv) (rng f)
```

```
-- (2) Observers -----------------------------------------------
dbschema :: RDB -> SchemaRDB
dbschema = (id *-> schema) . relations

-- (3) RDB operators -------------------------------------------

-- Union
union :: IdRel -> IdRel -> RDB -> Error Relation
union id1 id2 db =
 do {r1 <- aplpf' (relations db) id1 ;
     r2 <- aplpf' (relations db) id2 ;
     result <- inv' (unionR r1 r2) ;
     if (restrEqdom r1 r2)
      then Ok result
      else Err "Error in Union Operation: incompatible schemas"
    }

-- Intersection
inter :: IdRel -> IdRel -> RDB -> Error Relation
inter id1 id2 db =
 do {r1 <- aplpf' (relations db) id1 ;
     r2 <- aplpf' (relations db) id2 ;
     result <- inv'(interR r1 r2) ;
     if (restrEqdom r1 r2)
      then Ok (result)
      else Err "Error in Intersection Op.:incompatible schemas"
    }

-- Difference
diff :: IdRel -> IdRel -> RDB -> Error Relation
diff id1 id2 db =
 do {r1 <- aplpf' (relations db) id1 ;
     r2 <- aplpf' (relations db) id2 ;
     result <- inv'(diffR r1 r2) ;
     if (restrEqdom r1 r2)
      then Ok (result)
      else Err "Error in Difference Op.: incompatible schemas"
    }

-- Projection
project :: Set IdAttr -> IdRel -> RDB -> Error Relation
project s id db =
 do {r1 <- aplpf' (relations db) id ;
     result <- inv' ( projectR s r1) ;
     if (s \< dom(schema r1))
      then Ok (result)
      else Err "Error in Project op.:
                  attributes are not in relation domain"
    }
```

```
-- Selection
select ::  Pfun IdAttr Value  -> IdRel -> RDB -> Error Relation
select f id db =
 do {r1 <- aplpf' (relations db) id ;
     result <- inv' (selectR f r1);
     if ((dom f) \< dom(schema r1))
      then Ok (result)
      else Err "Error in select op.:
                  attributes are not in relation domain"
    }
-- Natural join
natjoin:: IdRel -> IdRel -> RDB -> Error Relation
natjoin id1 id2 db =
 do {r1 <- aplpf' (relations db) id1 ;
     r2 <- aplpf' (relations db) id2 ;
     inv'(natjoinR r1 r2)
    }
-- Renaming
rename :: Pfun IdAttr IdAttr -> IdRel -> RDB -> Error Relation
rename f  id1 db =
 do {r1 <- aplpf' (relations db) id1 ;
     result <- inv'(renameR f r1) ;
     if ((dom f) \< dom(schema r1))
      then Ok (result)
      else Err "Error in rename op.:
              attributes to rename are not in relation domain"
    }
-- Divide
divide :: IdRel -> IdRel -> RDB -> Error Relation
divide id1  id2 db =
 do {r1 <- aplpf'(relations db) id1 ;
     r2 <- aplpf'(relations db) id2 ;
     inv'(divideR r1 r2)
    }

--(4) Olap operators ------------------------------------------
mda::Set IdAttr->IdAttr->(Value->Value->Value)->Value->IdRel
     ->RDB->Error Relation
mda s a f u id db =
 do {r1<- aplpf' (relations db) id ;
     inv'(mdaR s a f u r1 )
    }

-- (5) Restrictions -------------------------------------------
restrEqdom:: Relation -> Relation -> Bool
restrEqdom r1 r2 = schema r1 == schema r2
restrEqdom _  _ = False
```

Bibliography

[ABNO97] J. J. Almeida, L. S. Barbosa, F. L. Neves, and J. N. Oliveira. Dwq consortium. deliverable d1.1, data warehouse quality requirements and framework. technical report dwq-ntua-1001. Technical report, NTUA Athens, Greece (1997)., 1997.

[AGS97] Rakesh Agrawal, A. Gupta, and Sunita Sarawagi. Modeling multidimensional databases. In Alex Gray and Per-Åke Larson, editors, *Proc. 13th Int. Conf. Data Engineering, ICDE*, pages 232–243. IEEE Computer Society, 7–11 1997.

[Bac78] J. Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *CACM*, 21(8):613–639, August 1978.

[Bac95] José Carlos Bacelar. Introdução à teoria das categorias. Relatório Técnico UMDITR9507, Departamento de Informática, Universidade do Minho, Braga, Portugal, 1995.

[Bac00] R. C. Backhouse. Fixed point calculus, 2000. Summer School and Workshop on Algebraic and Coalgebraic Methods in the Mathematics of Program Construction, Lincoln College, Oxford, UK 10th to 14th April 2000.

[Bal95] Dorel D. Baluta. A formal specification in z of the relational data model, version 2, of e. f. codd. M. sc. thesis, Concordia University, Montreal, QC, Canada, 1995.

[BB03] K. Backhouse and R.C. Backhouse. Safety of abstract interpretations for free,via logical relations and Galois connections. *Science of Computer Programming*, 2003. Accepted for publication.

[BD77] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *JACM*, 24(1):44–67, January 1977.

[BdM97] R. Bird and O. de Moor. *Algebra of Programming*. Series in Computer Science. Prentice-Hall International, 1997. C. A. R. Hoare, series editor.

[BH93] R. C. Backhouse and P. F. Hoogendijk. Elements of a relational theory of datatypes. In S.A. Schuman, B. Möller, and H.A. Partsch, editors, *Formal Program Development*, number 755 in Lecture Notes in Computer Science, pages 7–42. Springer, 1993.

[BH95] Eerke Boiten and Paul Hoogendijk. A database calculus based on strong monads and partial functions. Technical report, Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, 1995.

[BJ82] D. Bjorner and C. B. Jones. *Formal Specification and Software Development*. Series in Computer Science. Prentice-Hall International, 1982. C. A. R. Hoare.

[Bo98] R.C. Backhouse and T. Sheard (org.). WGP'98 — Workshop on Generic Programming, 1998. Marstrand, Sweden, 18th June, 1998 (`http://www.cse.ogi.edu/PacSoft/conf/wgp/`).

[CD97] Surajit Chaudhuri and Umesh Dayal. An overview of data warehousing and olap technology. In *Proc. ACM SIGMOD Record 26(1)*, march, 1997.

[CL02] Dave Clarke and Andres Löh. Generic haskell, specifically. In *In Jeremy Gibbons and Johan Jeuring, editors. Generic Programming. Proceedings of the IFIP TC2 Working Conference on Generic Programming, Schloss Dagstuhl, July 2002. ISBN 1-4020-7374-7. Kluwer Academic Publishers*, pages 21–48, 2002.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *CACM*, 13(6):377–387, June 1970.

[Cod71a] E. F. Codd. A database sublanguage founded on the relational calculus. In *ACM SIGFIDET Workshop on Data Description, Access and Control*, pages 35–61, Nov. 1971.

[Cod71b] E. F. Codd. Normalized database structure: A brief tutorial. In *ACM SIGFIDET Workshop on Data Description, Access and Control*, pages 1–17, Nov. 1971.

[Cod72a] E. F. Codd. Further normalization of the database relational model. In *Data Base Systems*, pages 33–64. Prentice-Hall, 1972. Courant Inst. Computer Science Symp. 6, Englewood Cliffs, NJ.

[Cod72b] E. F. Codd. Relational completeness of database sublanguages. In *Data Base Systems*, pages 65–98. Prentice-Hall, 1972. Courant Inst. Computer Science Symp. 6, Englewood Cliffs, NJ.

[Cod74] E. F. Codd. Recent investigations in relational database systems. In *1974 IFIP Conf.*, pages 1017–1021, 1974.

[Cod90] E. F. Codd. *Missing Information*. Addison-Wesley Publishing Company, Inc., 1990.

[Cou97] OLAP Council. The OLAP glossary, 1997. (`http://www.olapcouncil.org`).

[Fer04] L.G. Ferreira. Formalizing markup languages for user interface. Master's thesis, University of Minho, 2004. Submitted.

[FJ90] J.S. Fitzgerald and C.B. Jones. *Modularizing the formal description of a database system*, volume 428 of *Lecture Notes in Computer Science*. Springer, 1990.

[GCB+97] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and subtotals. *J. Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[GHA01] Jeremy Gibbons, Graham Hutton, and Thorsten Altenkirch. When is a function a fold or an unfold? In Andrea Corradini, Marina Lenisa, and Ugo Montanari, editors, *Proceedings 4th Workshop on Coalgebraic Methods in Computer Science, CMCS'01, Genova, Italy, 6–7 Apr. 2001*, volume 44(1). Elsevier, Amsterdam, 2001.

[GL97] Marc Gyssens and Laks V. S. Lakshmanan. A foundation for multi-dimensional databases. In *The VLDB Journal*, pages 106–115, 1997.

[Gra96] Paul Graham. *ANSI Common Lisp*. Prentice Hall, 1996. GRA p4 96:1 1.Ex.

[Hen04] M.R. Henriques. pGML : estudo de um subconjunto "preciso" do GML2.12. Master's thesis, Universidade do Minho, Nov. 2004. In Portuguese.

[Hin99] Ralf Hinze. Polytipic functions over nested datatypes. *Discrete Mathematics and theoretical Computer Science*, pages 159–180, 1999.

[Hin00a] Ralf Hinze. A new approach to generic functional programming. In Thomas W. Reps, editor, *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposiumon Principles of Programming Languages (POPL' 00), Boston, Massachusetts, January 19-21*, pages 119–132, January 2000.

[Hin00b] Ralf Hinze. Polytypic values possess polykinded types. In *Mathematics of Program Construction*, pages 2–27, 2000.

[HJ00] R. Hinze and S. Jones. Derivable type classes, 2000. Haskell Workshop 2000.

[HJ03] R. Hinze and J.Jeuring. Generic haskell: Practice and theory, 2003. Lecture notes of the Summer School on Generic Programming, LNCS Springer-Verlang.

[Hug99] J. Hughes. Restricted datatypes in Haskell, 1999. In Third Haskell Workshop. Utrecht University technical report.

[Jan00] P. Jansson. *Functional Polytypic Programming*. PhD thesis, Chalmers University og Technology and Götebord University, 2000.

[Jay95a] C.B. Jay. Polynomial polymorphism. In *Computing: The Australasian Theory Symposium Proceedings, Melbourne, Australia, 29–30 January, 1995*, pages 237–243, 1995.

[Jay95b] C.B. Jay. A semantics for shape. *Science of Computer Programming*, 1995. In press, 27 pp.

[Jeu95] J. Jeuring. Polytypic pattern matching. In S. Peyton Jones, editor, *Proceedings of Functional Programming Languages and Computer Architecture (FPCA95)*, pages 238–248, 1995.

[JJ96] J. Jeuring and P. Jansson. Polytypic programming. In *Advanced Functional Programming*, number 1129 in Lecture Notes in Computer Science. Springer, 1996.

[JJ97] Patrik Jansson and Johan Jeuring. PolyP—A polytypic programming language extension. In *Conf. Record 24th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL'97, Paris, France, 15–17 Jan 1997*, pages 470–482. ACM Press, New York, 1997.

[JJ98] P. Jansson and J. Jeuring. Polylib — a library of polytypic functions. In *Workshop on Generic Programming (WGP'98), Marstrand, Sweden*, 1998.

[Jon80] C. B. Jones. *Software Development — A Rigorous Approach*. Series in Computer Science. Prentice-Hall International, 1980. C. A. R. Hoare.

[Jon86] C. B. Jones. *Systematic Software Development Using VDM*. Series in Computer Science. Prentice-Hall International, 1986. C. A. R. Hoare.

[Jon93] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *FPCA '93: Conference on Functional Programming and Computer Architecture, Copenhagen, Denmark*, pages 52–61, New York, N.Y., 1993. ACM Press.

[Jon03] S.L. Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, Cambridge, UK, 2003. Also published as a Special Issue of the Journal of Functional Programming, 13(1) Jan. 2003.

[LM99] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *Domain-Specific Languages*, pages 109–122, 1999.

[LV03] R. Lämmel and J. Visser. A Strafunski Application Letter. In V. Dahl and P. Wadler, editors, *Proc. of Practical Aspects of Declarative Programming (PADL'03)*, volume 2562 of *LNCS*, pages 357–375. Springer-Verlag, January 2003.

[LW96] Chang Li and Xiaoyang Sean Wang. A data model for supporting on-line analytical processing. In *CIKM*, pages 81–88, 1996.

[MA86] E. G. Manes and M. A. Arbib. *Algebraic Approaches to Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1986. D. Gries, series editor.

[Mai83] D. Maier. *The Theory of Relational Databases*. Computer Science Press, 1983. ISBN 0-914894-42-0.

[Mal90] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.

[Mil84] Robin Milner. A proposal for standard ML. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 184–197. ACM Press, 1984.

[Mor90] C. Morgan. *Programming from Specification*. Series in Computer Science. Prentice-Hall International, 1990. C. A. R. Hoare, series editor.

[Muk97] P. Mukherjee. Automatic translation of VDM-SL specifications into Gofer. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*, number 1313 in Lecture Notes in Computer Science, pages 258–277. Springer, 1997.

[NO02] C. Necco and J.N. Oliveira. Generic data processing: A normalization exercise. In *CACIC'02* , pages 751–761, October 2002. 8th Argentinian Computer Science Congress, Univ. Buenos Aires, 15-18th October.

[NSO99] F. L. Neves, J. C. Silva, and J. N. Oliveira. *Converting Informal Meta-data to VDM-SL: A Reverse Calculation Approach* . In *VDM in Practice! A Workshop co-located with FM'99: The World Congress on Formal Methods, Toulouse, France, 20-21 September*, September 1999.

[Oka00] Chris Okasaki. An overview of edison, 2000. In ICFP 2000 (Haskell Workshop).

[Oli90] J. N. Oliveira. A reification calculus for model-oriented software specification. *Formal Aspect of Computing*, 2(1):1–23, April 1990.

[Oli92] J. N. Oliveira. *Software Reification using the SETS Calculus* . In *Proc. of the BCS FACS 5th Refinement Workshop, Theory and Practice of Formal Software Development, London, UK*, pages 140–171. Springer-Verlag, 8–10 January 1992. (Invited paper).

[Oli98] J. N. Oliveira. 'Fractal' Types: an Attempt to Generalize Hash Table Calculation. In *Workshop on Generic Programming (WGP'98), Marstrand, Sweden*, June 1998.

[Oli99] J.N. Oliveira. An introduction to pointfree programming, 1999. Departamento de Informática, Universidade do Minho. 37p., chapter of book in preparation.

[Oli01a] J. N. Oliveira. "Bagatelle in C arranged for VDM SoLo". *Journal of Universal Computer Science*, 7(8):754–781, 2001. Special Issue on *Formal Aspects of Software Engineering* (*Colloquium in Honor of Peter Lucas* , *Institute for Software Technology*, Graz University of Technology, May 18-19, 2001).

[Oli01b] J.N. Oliveira. Data processing by calculation, 2001. Lectures at the *6th Estonian Winter School in Computer Science* March 2001, Palmse, Estonia.

[Oli01c] J.N. Oliveira. A quick look at monads, 2001. Departamento de Informática, Universidade do Minho. Chapter of book in preparation.

[Oli04] J. N. Oliveira. Constrained datatypes, invariants and business rules: a relational approach, 2004. PUReCafé, DI-UM, 2004.5.20 [talk], PURE PROJECT (POSI/CHS/44304/2002).

[OR04] J.N. Oliveira and C.J. Rodrigues. Transposing relations: from *Maybe* functions to hash tables. In *MPC'07 : Seventh International Conference on Mathematics of Program Construction, 12-14 July, 2004, Stirling, Scotland, UK (Organized in conjunction with AMAST'04)*, Lecture Notes in Computer Science. Springer, 2004. Accepted for publication.

[RBH96] Oege de Moor Richard Bird and Paul Hoogendijk. Generic functional programming with types and relations. *Journal of Functional Programming*, pages 1–28, 1996.

[RO97] C. J. Rodrigues and J. N. Oliveira. *Normalization is Data Reification.* Technical Report UMDITR9702, University of Minho, Dec. 1997.

[Rue92] Fritz Ruehr. *Analitical and Structural Polymorphism Expressed Ussing Patterns Over Types*. PhD thesis, University of Michigan, 1992.

[Sho82] Arie Shoshani. Statistical databases: Characteristics, problems, and some solutions. In *Eigth International Conference on Very Large Data Bases, September 8-10, 1982, Mexico City, Mexico, Proceedings*, pages 208–222. Morgan Kaufmann, 1982.

[SN95] T. Sheard and N. Nelson. Type safe abstractions using program generators. Technical report, Oregon Graduate Institute of Science and Technology, 1995.

[SW92] W. B. Samson and A. W. Wakelin. *Algebraic Specification of Databases: A Survey from a Database Perspective*, volume Glasgow of *Workshops in computing series*. Springer Verlang, 1992.

[Tho96] Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley, 1st edition, 1996. ISBN 0-201-40357-9.

[Tur85] David A. Turner. Miranda—a non-strict functional language with polymorphic types. In J.P. Jouannaud, editor, *Functional programming languages and Computer Architectures*, volume 201 of *Lecture Notes in Computer Science*. Springer Verlag, 1985.

[Wad89] Philip L. Wadler. Theorems for free! In *4th International Symposium on Functional Programming Languages and Computer Architecture*, London, Sep. 1989. ACM.

[Wad92] Philip Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Albequerque, New Mexico, 1992.

[Wal90] A. Walshe. *NDB: The Formal Specification and Rigorous Design of a Single-User Database System*. Prentice Hall, ISBN 0-13-116088-5, 1990.