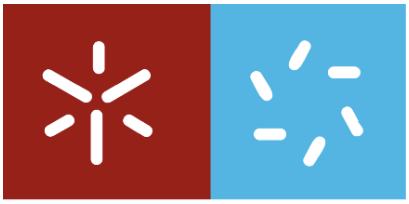


Universidade do Minho

Relatório de Estágio da Licenciatura em Matemática e Ciências de Computação

Strong Types for Relational Data Stored in Databases or Spreadsheets

Alexandra Martins da Silva

Supervisores:

Joost Visser
José Nuno Oliveira

Braga
2006

Resumo

É frequente encontrar inconsistências nos dados (relacionais) armazenados numa base de dados ou numa folha de cálculo. Além disso, verificar a sua integridade num ambiente de programação de baixo nível, como o oferecido pelas folhas de cálculo, é complicado e passível de gerar erros.

Neste relatório, é apresentado um modelo fortemente tipado para bases de dados relacionais e operações sobre estas. Este modelo permite a detecção de erros e a verificação da integridade da base de dados, estaticamente. A sua codificação é feita na linguagem de programação funcional Haskell.

Além das tradicionais operações em bases de dados relacionais, formalizamos dependências funcionais, formas normais e operações para transformação de bases de dados.

Apresentamos também uma aplicação prática deste modelo na área da engenharia reversa de folhas de cálculo. Neste sentido, codificamos operações de migração entre o nosso modelo e folhas de cálculo.

O modelo apresentado pode ser usado para modelar, programar e migrar bases de dados numa linguagem fortemente tipada.

Agradecimentos

O trabalho descrito neste relatório representa para mim o fim de um ciclo. Assim, gostava de aproveitar a oportunidade para agradecer a algumas pessoas que foram extremamente relevantes para o sucesso que tive ao longo do meu percurso académico.

Os meus primeiros agradecimentos vão para os professores Orlando Belo e Luís Barbosa. O primeiro, porque, há cinco anos, com a sua capacidade de persuasão e o seu profissionalismo me convenceu a escolher a licenciatura em Matemática e Ciências de Computação, o que se veio a revelar uma escolha acertada. O segundo, além de me ter trazido para o projecto PURe, o que proporcionou a realização deste trabalho, acompanha-me agora no início de uma nova fase. Muito obrigado por toda a ajuda no começo da minha aventura no mundo da investigação. Ambos são para mim duas grandes referências.

O entusiasmo do professor José Nuno Oliveira foi muito importante para manter sempre em alta a minha vontade de levar este trabalho até ao fim! Obrigado por todas as frutíferas discussões sobre eventuais caminhos a seguir e por toda a paixão que coloca no seu trabalho, que me faz acreditar que uma carreira na investigação pode ser gratificante.

A gratidão que eu tenho em relação ao meu supervisor, Joost Visser, é impossível de expressar por palavras. O seu empenho e dedicação em todos os trabalhos que faz ou supervisiona são admiráveis e inspiradores. Sem ele, este trabalho não teria sido possível e agradeço sobretudo todos os bons conselhos que me deu em todas as fases. O que aprendi ao longo deste ano de trabalho com ele vai ser de um valor inestimável no meu futuro profissional. Por isso e por tudo o resto: muito obrigado, Joost!

Queria agradecer aos meus pais, por todos os sacrifícios que fizeram ao longo da vida para eu poder ter sempre a melhor educação e, em especial, por todo o apoio que me deram no último ano e meio. Obrigado por acreditarem em mim! A minha irmã Irene foi ao longo da minha vida uma referência e a sua preocupação constante comigo é de louvar. Obrigado por tudo, maninha! Gostava também de agradecer ao meu irmão Filipe que, apesar de distante fisicamente, conseguiu transmitir-me a calma com que encara tudo na vida através de muitos telefonemas e *emails*. Finalmente, em termos familiares, gostava de agradecer ás minhas sobrinhas, por todos os sorrisos e tropelias que me enchem a alma de alegria.

O apoio dos meus amigos, não só no estágio, mas também ao longo de toda a licenciatura foi crucial para eu ter mantido a minha sanidade mental até ao fim. Para a Ana, a Carina, o David, o Gonçalo, o Jácome, a Marta e o Tércio, os meus sinceros agradecimentos. Gostava de deixar um agradecimento especial ao Jácome e ao David. O primeiro porque muitas vezes, usando uma expressão dele, me *moeu o juízo* para eu parar de trabalhar; o segundo porque, com toda a sua experiência e sabedoria, me deu conselhos muito preciosos.

Durante o estágio, tive a oportunidade de partilhar o espaço de trabalho com pessoas divertidas, que, desde o início, me ajudaram a seguir o *bom caminho*. Por isso, gostava de expressar um agradecimento por todo o apoio que o João, o Miguel, o Paulo e o Tiago me deram ao longo destes meses. Ao Paulo, em especial, gostava de agradecer as frutíferas discussões (por vezes filosóficas) sobre *type-level programming* e todos os preciosos comentários sobre código e versões prévias deste relatório!

As minhas últimas palavras de agradecimento vão para a pessoa que nos últimos anos tem dado significado à minha vida e porque, na vida, os afectos são o que realmente conta, gostava de expressar a minha gratidão por todo o apoio e amor que o Zé me deu em todas as ocasiões. A ele e ás minhas sobrinhas dedico este trabalho.

Abstract

Relational data stored in databases and spreadsheets often present inconsistencies. Moreover, data integrity checking in a low level programming environment, such as the one provided by spreadsheets, is error-prone.

We present a strongly-typed model of relational databases and operations on them, which allows for static checking of errors and integrity at compile time. The model relies on type-class bounded, parametric polymorphism. We demonstrate its encoding in the functional programming language Haskell.

Apart from the standard relational databases operations, we formalize functional dependencies, normal forms and operations for database transformation.

We also present a first approach on how to use this model to perform spreadsheet refactoring. For this purpose, we encode operations which migrate from spreadsheets to our model and *vice-versa*.

Altogether, this model can be used to design and experiment with typed languages for modeling, programming and migrating databases.

This work was developed as an internship which took place in the second semester of the fifth year of the *Licenciatura em Matemática e Ciências de Computação* degree. This internship was supported by *PURe Project*[1].

[1] FCT under contract POSI/CHS/44304/2002

Contents

Chapter 1

Introduction

Databases play an important role in the technology world nowadays. A lot of our daily actions involve a database access or transaction: when you purchase items in your supermarket it is likely that the stock value for those is automatically updated in a central database; every time you pay with your debit or credit card your bank database will be changed to reflect the transaction made.

Despite its vital importance, work on formal models for designing relational databases is not widespread [27]. The main reason for this is the fact that database theory is considered to be stable. However, recent work by Oliveira [34], develops a *pointfree* [6] formalization of the traditional relational calculus (as presented by Codd [12]), shows that it is possible to present database theory in a more structured, simpler and general form.

A database schema specifies the well-formedness of a relational database. It tells us, for example, how many columns each table must have and what the types of the values for each column should be. Furthermore, some columns may be singled out as keys, some may be allowed to take null values. Constraints can be declared for specific columns, and foreign key constraints can be provided to prescribe relationships between tables.

Operations on a database should preserve its well-formedness. The responsibility for checking that they do lies ultimately with the database management system (DBMS). Some operations are rejected statically by the DBMS, during query compilation. Insertion of ill-typed values into columns, or access to non-existing columns fall into this category. Other operations can only be rejected dynamically, during query execution, simply because the actual content of the database is involved in the well-formedness check. Removal of a row from a table, for instance, might be legal only if it is currently not referenced by another row.

The division of labor between static and dynamic checking of database operations is effectively determined by the degree of precision with which types can be assigned to operations and their sub-expressions. A more precise type has higher information content (larger intent) and is inhabited by less expressions (smaller extent). A single column constraint, such as $0 \leqslant c \leqslant 1$ for example, must be checked dynamically, unless we find a way of capturing these boundaries in its type. If c can be assigned type $\mathit{Probability}$ rather than Real, the constraint would become checkable statically.

In this report, we will investigate whether more precise types can be assigned to database operations than is commonly done by the static checking components of DBMSs. For instance, we will capture key meta-data in the types of tables, and transport such information through the operators from argument to result table types. This allows us to assign a more

precise type, for instance, to the join operator when joining on keys. Joins that are ill-formed with respect to key information can then be rejected statically.

However, further well-formedness criteria might be in vigor for a particular database that are not captured by the meta-data provided in its schema. Prime examples for such criteria are the various *normal forms* of relational databases that have been specified in the literature [38, 27, 12, 14]. Such normal forms are defined in terms of *functional dependencies* [5] between (groups of) columns that are or are not allowed to be present. We will show that functional dependency information can be captured in types as well, and can be carried through operations. Thus, the type-checker will infer functional dependencies on the result tables from functional dependencies on the argument tables. Furthermore, normal form constraints can be expressed as type constraints, and normal form validation can be done by the type checker.

It would be impractical to have the query compiler of a DBMS performing type checking with such precision. The type-checking involved would delay execution, and a user might not be present to review inferred types or reported type errors. Rather, we envision that stronger types can be useful in off-line situations, such as database design, development of database application programs, and database migration. In these situations, more type precision will allow a more rigorous and ultimately safer approach.

The spreadsheet is a kind of interactive tool for data processing which is widely used tool, mainly by the non-professional programmers community, which is 20 times larger than the professional one. However, maintenance, testing and quality assessment of spreadsheets is difficult. This is mainly because the spreadsheet paradigm offers a very low level programming environment – it does not support encapsulation or structured programming. Many spreadsheet users actually use their spreadsheet as a database, although they lose the expressiveness of having keys and attributes relations specified. Migrating spreadsheets to a more structured language becomes essential to do their reverse engineering. For this purpose, we will also link our strongly typed database model to spreadsheets, providing a *map* between our model and the Gnumeric spreadsheet format[1]. This will allow the user to check integrity of plain data stored in a spreadsheet, using the design features offered by our model, and to use the available SQL operations.

1.1 Objectives

The overall goal of this work is to use strong typing to provide more static checks to relational database programmers and designers, including spreadsheet users. More concretely, as specific objectives, we intend to:

1. Provide strong types for SQL operations.

2. Capture database design information in types, in particular functional dependencies.

3. Leverage database types for spreadsheets used as databases.

In Conclusions (chapter 7), we will assess whether these objectives were reached.

[1] http://www.gnome.org/projects/gnumeric/

Report structure

In Sections 1.2 and 1.3, we discuss related work, both on representing databases in Haskell and on spreadsheet understanding. Sections 2.1 and 2.2 explain the basics of the type-class-based programming, or type-level programming, and the Haskell HLIST library of which we make essential use to represent our model.

In Chapter 3, we will present the basic layer of our model: representation of databases and basic relational operations. In Chapter 4, we turn to the SQL layer: a typeful reconstruction of statements and clauses of the SQL language. We also lift traditional table operations to the database level. This part of the model provides static type checking and inference of well-formedness constraints normally specified in a schema. In Chapter 5, we present the advanced layer of our model, which concerns functional dependencies and normal forms. In particular, we show how a new level of operations can be defined on top of the *SQL* level where functional dependency information is transported from argument tables to results. We also go beyond database modeling and querying, by addressing database normalization and denormalization.

A first approach for using our model in spreadsheet understanding is presented in Chapter 6. We conclude and present future work directions in Chapter 7.

Some parts of Sections 1.2, 2.1, 2.2 and Chapters 3, 4, 5 appeared originally in the draft paper [36]. In Section 2.2 description for extra operations defined for heterogeneous lists and records was added. In Chapters 3 and 4 examples were added and more detailed explanation for operations was included. Chapter 5 includes extra normal forms, an algorithm for checking functional dependencies and further examples.

Availability

The source distribution that supports this report is available from the homepage of the author, under the name CODDFISH. Apart from the source code shown here, the distribution includes a variety of relational algebra operators, further reconstructions of SQL operations, database migration operations, and several worked-out examples. The library is presented schematically in figure 1.1. CODDFISH lends itself as a sandbox for the design of typed languages for modeling, programming, and transforming relational databases.

1.2 Related Work

We are not the first to provide types for relational databases, and type-level programming has other applications besides type checking SQL. A brief discussion of related approaches follows.

Machiavelli

Ohori *et al.* extended an ML-like type system to include database programming operations such as join and projection [32]. The extension is necessary to provide types for labeled records, which are used to model databases. They show that the type inference problem for the extended system remains solvable. Based on this type system, the experimental *Machiavelli* language for database programming was developed [33, 9].

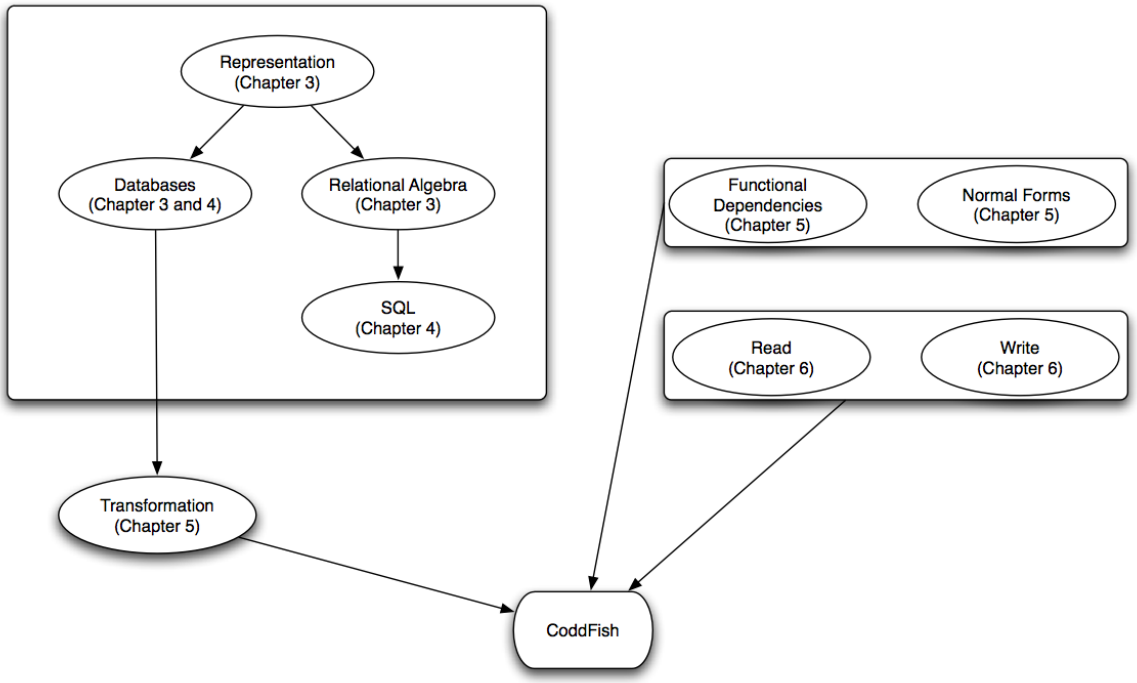


Figure 1.1: Modules which constitute the CoddFish library

The prominence of labeled records is a commonality between the approach of Ohori *et al.* and ours. Their objective, however, is not to provide a faithful model of SQL. Rather, they aim at generalized relational databases and go towards modeling object-oriented databases. Their datatypes for modeling tables are sets of records, with no distinction between key and non-key attributes. Also, meta-information (record labels) are stored with every row, rather than once per table. They do not address the issue of functional dependencies.

Our language of choice, Haskell, can be considered to belong to the ML-family of languages that offer higher-order functions, polymorphism and type inference. However, type-class bounded polymorphism is not a common feature shared by all members of that family. In fact, Ohori *et al.* do not assume these features. This explains why they need to develop a dedicated type system and language, while we can stay inside an existing language and type system.

HaskellDB

Leijen *et al*. present a general approach for embedding domain-specific compilers in Haskell, which they apply to the implementation of a typeful SQL binding, called *Haskell/DB* [26]. They construct an embedded DSL that consists of sub-languages for basic expressions, relational algebra expressions, and query comprehension. Strong types for the basic expression language are provided with phantom types in data constructors to carry type information. For relational algebra expressions, the authors found no solution of embedding typing rules in Haskell, citing the join operator as especially difficult to type. The query comprehension

language is strongly typed again, and is offered as a safe interface to the relational algebra sub-language.

The original implementation of Haskell/DB relies on an experimental extension of Haskell with extensible records supported only by the Hugs interpreter. A more portable improvement of Haskell/DB uses a different model of extensible records [8], which is more restricted, but similar in spirit to the HLIST library of [25] which we rely on.

The level of typing realized by Haskell/DB does not include information to distinguish keys from non-key attributes. Concerning the relational algebra sub-language, which includes restriction (filter), projection, product, and set operators, only syntactic well-formedness checks are offered. Functional dependencies are not taken into account.

From the DSL expressions, Haskell/DB generates concrete SQL syntax as unstructured strings, which are used to communicate with an SQL server via a foreign function interface. The DSL shields the user from the unstructured strings and the low-level communication details.

Our database operations, by contrast, operate on Haskell representations of database tables, and the maps included in them. We have not addressed the topic of persistence (*e.g.* by connection to a database server).

OOHaskell

Kiselyov *et al.* have developed a model of object-oriented programming inside Haskell [24], based on their HLIST library of extensible polymorphic records with first-class labels and subtyping [25]. They rely on type-level programming via the HLIST. For modeling advanced notions of subtyping they also make use of type-level programming directly. The model includes all conventional object-oriented features and more advanced ones, such as flexible multiple inheritance, implicitly polymorphic classes, and many flavors of subtyping. In fact, the authors describe their OOHASKELL library as a laboratory or sandbox for advanced and typed OO language design.

We have used the same basis (the extensible HLIST records) and the same techniques (type-level programming) for modeling a different paradigm, *viz*. relational database programming rather than object-oriented programming. Both models non-trivially rely on type-class bounded and parametric polymorphism, and care has been taken to preserve type inference in both cases.

There are also notable differences between the object-orientation model and the relational database model. Our representation of tables separates meta-data from normal data values and resorts to numerous type-level predicates and functions to relate these. In the OOHASKELL library, labels and values are mostly kept together and type-level programming is kept to a minimum. Especially our representation of functional dependencies explores this technique to a much further extent. Concerns such as capturing advanced design information (such as functional dependencies) at the type-level, carrying such design information in the argument and result types of operations, and thus maintaining at compile-time integrity with respect to design is not present in the object-oriented model.

Point-free relational algebra

Necco *et al.* have developed models of relational databases in Haskell and in Generic Haskell [29, 30]. The model in Haskell is weakly typed in the sense that fixed types are

used for values, columns, tables, and table headers. Arbitrary-length tuples and records are modeled with homogeneous lists. Well-formedness of tables and databases is guarded by ordinary value-level functions. Generic Haskell is an extension of Haskell that supports *polytypic* programming. The authors use these polytypic programming capabilities to generalize from the homogeneous list type constructor to any collection type constructor. The elements of these collections are still fixed types.

Apart from modeling a series of relational algebra operators the authors provide a suite of calculation rules for database transformation. These rules are formulated in *point-free* style, which allow the construction of particularly elegant and concise calculational derivations.

Our model of relational databases can be seen as a successor to the Haskell model of Necco *et al.* where well-formedness checking has been moved from the value level to the type level. We believe that their database transformation calculus can equally be fitted with strong types to guard the well-formedness of calculation rules. In fact, our projection and join operators can be regarded as modeling such rules both at the value and the type level.

Oliveira [34] provides a point-free reconstruction of functional dependency theory, making it concise, simple, and amenable to calculational reasoning. His reconstruction includes the Armstrong axioms and lossless decomposition in *pointfree notation*. We believe that this point-free theory is a particularly promising basis for generalization and extension of our relational database model.

Two-level data transformation

Cunha *et al.* [13] use Haskell to provide a strongly typed treatment of two-level data transformation, such as data mappings and format evolution, where a transformation on the type level is coupled with transformations on the term level. The treatment essentially relies on *generalized* algebraic datatypes (GADT) [35]. In particular, a GADT is used to safely represent types at the term level. These type representations are subject to rewrite rules that record output types together with conversion functions between input and output. Examples are provided of information-preserving and information-changing transformations of databases represented by finite maps and nested binary tuples.

Our representation of databases is similar in its employment of finite maps. The arbitrary-length tuples of the HLIST library, are basically nested binary tuples, with an additional constraint that enforces association to the right and termination with an empty tuple. However, the employment of type-level indexes to model table names and attribute names in headers, rows, and databases goes beyond the maps-and-tuples representation, allowing a nominal, rather than purely structural treatment. Functional dependencies are not represented at all. On the other hand, our representation is limited to databases, while Cunha *et al.* also cover polynomial data structures, involving sums, lists, recursion, and more.

The SQL ALTER statements and our database transformation operations for composition and decomposition have counterparts as two-level transformations on the maps-and-tuples representation. In fact, Cunha *et al.* present two sets of rules, one for data mappings and the other for format evolution, together with generic combinators for composing these rules. We have no such generic combinators, but instead are limited to normal function application on the value level, and to logical composition of constraints at the type level. On the other hand, we have shown that meta-information such as attribute names, nullability and defaults, primary keys, foreign keys, and functional dependencies, can be transported through database transformation operations.

1.3 Related work on Spreadsheets

In this section, we will present a brief discussion on work done to formalize the spreadsheet paradigm.

The UCheck project

In this project, Martin Erwig, Robin Abraham and Margaret Burnett define a unit system for spreadsheets [10, 17, 1] that allows one to reason about the correctness of formulas in concrete terms. The fulcral point of this project is the high flexibility of the unit system developed, both in terms of error reporting [3] and reasoning rules, that increases the possibility of a high acceptance among end users.

The Gencel project

Spreadsheets are likely to be full of errors and this can cause organizations to lose millions of dollars. However, finding and correcting spreadsheet errors is extremely hard and time consuming.

Probably inspired by database systems and how database management systems (DBMS) maintain the database consistent after every update and transaction, Martin Erwig, Robin Abraham, Irene Cooperstein and Steve Kollmansberger designed and implemented Gencel [16]. Gencel is an extension to Excel, based on the concept of a spreadsheet template, which captures the essential structure of a spreadsheet and all of its future evolutions. Such a template ensures that every update is safe, avoiding reference, range and type errors [4, 15, 2].

Simon Peyton Jones et al.

Simon Peyton Jones, together with Margaret Burnett and Alan Blackwell, describe extensions to Excel that allow one to integrate user-defined functions in the spreadsheet grid [23, 7]. This take us from a end-user programming paradigm to an extended system, which provides some of the capabilities of a general programming language.

Chapter 2

Background

In this chapter, we will introduce some of the features of the functional programming language Haskell and a library with operations on heterogeneous lists that we need to build our model.

2.1 Type-level programming

Haskell is a non-strict, higher-order, typed functional programming language [22]. The syntax of Haskell is quite light-weight, resembling mathematical notation. It employs *currying*, a style of notation where function application is written as juxtaposition, rather than with parenthesized lists of comma-separated arguments. That is, $f\ x\ y$ is favored over $f(x, y)$ and functions may be applied partially in a way such that, for example, $f\ x$ is equivalent to $\lambda y \rightarrow f\ x\ y$

We will introduce further Haskell-specific notations as they are used throughout the report, but we start with an explanation of a language construct, a programming style, and a library of which we will make extensive use.

2.1.1 Type classes

Haskell offers nominal algebraic datatypes that may be specified for example as follows:

$\mathbf{data}\ Bool = True \mid False$
$\mathbf{data}\ Tree\ a = Leaf\ a \mid Fork\ [Tree\ a]$

Here $[a]$ denotes list type construction. The datatype constructors can be used to specify complex types, such as $Tree\ (Tree\ Bool)$ and the data constructors can be used in pattern matching or case discrimination:

$depth :: Tree\ a \rightarrow Int$
$depth\ (Leaf\ a) = 0$
$depth\ (Fork\ ts) = 1 + maximum\ (0 : (map\ depth\ ts))$

Here $maximum$ and map are standard list processing functions, and $x : xs$ denotes concatenation of an element to the front of a list.

Data types for which functions with similar interface (signature) can be defined may be grouped into a *type class* that declares an overloaded function with that interface. The type

variables of the class appear in the signature of the function. For particular types, instances of the class provide particular implementations of the functions. For instance:

class *Show a* **where**
  $show :: a \rightarrow String$
instance *Show Bool* **where**
  $show\ True =$ `"True"`
  $show\ False =$ `"False"`
instance $(Show\ a, Show\ b) \Rightarrow Show\ (a, b)$ **where**
  $show\ (a, b) =$ `"("` $+\!\!+\ show\ a +\!\!+$ `","` $+\!\!+\ show\ b +\!\!+$ `")"`

The second instance illustrates how classes can be used in type constraints to put a bound on the polymorphism of the type variables of the class. A similar type constraint occurs in the inferred type of the *show* function, which is $Show\ a \Rightarrow a \rightarrow String$.

Type classes can have more than a single type parameter:

class $Convert\ a\ b \mid a \rightarrow b$ **where**
  $convert :: a \rightarrow b$
instance $Show\ a \Rightarrow Convert\ a\ String$ **where**
  $convert = show$
instance *Convert String String* **where**
  $convert = id$

A *functional dependency* (mathematically similar to, but not to be confused with functional dependencies in database theory) declares that the parameter a uniquely determines the parameter b. This dependency is exploited for type inference by the compiler. Note also that the two instances above are *overlapping*. Both multi-parameter type-classes with functional dependencies and permission of overlapping instances are extensions beyond the Haskell 98 language standard. These extensions are commonly used, supported by compilers, and well-understood semantically.

2.1.2 Classes as type-level functions

Single-parameter type classes can be seen as *predicates* on types, and multi-parameter type classes as *relations* between types. Interestingly enough, when some subset of the parameters of a multi-parameter type class functionally determines all the others, type classes can be interpreted as *functions* on the level of types [18]. Under this interpretation, $Show\ Bool$ expresses that booleans are showable, and $Convert\ a\ b$ is a function that computes the type b from the type a. Rather elegantly, the computation is carried out by the type checker.

Thus, in type-level programming, the class mechanism is used to define functions over types, rather than over values. The arguments and results of these type-level functions are types that model values, which may be termed type-level values. As an example, consider the following model of natural numbers on the type level:

data $Zero; zero = \bot :: Zero$
data $Succ\ n; succ = \bot :: n \rightarrow Succ\ n$

class *Nat n*
instance *Nat Zero*
instance $Nat\ n \Rightarrow Nat\ (Succ\ n)$

```
class Add a b c | a b → c where add :: a → b → c
instance Add Zero b b where add a b = b
instance (Add a b c) ⇒ Add (Succ a) b (Succ c) where
  add a b = succ (add (pred a) b)
pred :: Succ n → n
pred = ⊥
```

The types *Zero* and *Succ* generate type-level values of the type-level type *Nat*, which is a class. The class *Add* is a type-level function that models addition on naturals. Its member function *add*, is the equivalent on the ordinary value-level.

The type-level programming enabled by Haskell's class system is a form of logic programming, such as offered by Prolog [37] (on the value level). In [31], it is proposed that a functional style of type level programming should be added to Haskell instead.

2.2 The HLIST library

Type-level programming has been exploited by Kiselyov *et al.* to model arbitrary-length tuples, or *heterogeneous lists* [25]. These lists, in turn, are used to model extensible polymorphic records with first-class labels and subtyping. We will use these lists and records as the basis for our model of relational databases, an application which Kiselyov *et al.* already hinted.

The following declarations form the basis of the library:

```
data HNil = HNil
data HCons e l = HCons e l

class HList l
instance HList HNil
instance HList l ⇒ HList (HCons e l)

myTuple = HCons 1 (HCons True (HCons "foo" HNil))
```

The datatypes *HNil* and *HCons* represent empty and non-empty heterogeneous lists, respectively. The *HList* class, or type-level predicate, establishes a well-formedness condition on heterogeneous lists, *viz*. that they must be built from successive applications of the *HCons* constructor, terminated with and *HNil*. Thus, heterogeneous lists follow the normal cons-list construction pattern on the type-level. The *myHList* example shows that elements of various types can be added to a list.

Records can now be modeled as heterogeneous lists of pairs of labels and values.

```
myRecord
 = Record (HCons (zero, "foo") (HCons (one, True) HNil))
one = succ zero
```

All labels of a record are required to be pairwise distinct on the type level, and several label types are available to conveniently generate such distinct types. Type-level naturals are a simple candidate. A datatype constructor *Record* is used to distinguish lists that model records from other lists.

The library offers numerous operations on heterogeneous lists and records of which we list a few that we use later:

- Appending two heterogeneous lists

```
class HAppend l l' l'' | l l' → l'' where
  hAppend :: l → l' → l''
```

- Zip two lists into a list of pairs and vice-versa

```
class HZip x y l | x y → l, l → x y where
  hZip :: x → y → l
  hUnzip :: l → (x, y)
```

- Lookup a value in a record (given an appropriate label)

```
class HasField l r v | l r → v
  where hLookupByLabel :: l → r → v
```

Syntactic sugar is provided in the form of infix operators and an infix type constructor synonym.

```
type (:*:) e l = HCons e l
e .*. l = HCons e l
l .=. v = (l, v)
l .!. v = hLookupByLabel l v
myRecord = Record (zero .=. "foo" .*. one .=. True .*. HNil)
```

We have extended the library with some further operations.

For lists, we have defined set operations such as union, intersection, difference and powerset; functions to test if a list is empty, if a list is contained in another list (strictly or not) or if a type only appears once in a list:

```
class (HSet l, HSet l') ⇒ Union l l' l'' | l l' → l'' where
  union :: l → l' → l''
class (HSet l, HSet l') ⇒ Intersect l l' l'' | l l' → l'' where
  intersect :: l → l' → l''
class (HSet l, HSet l') ⇒ Difference l l' l'' | l l' → l'' where
  difference :: l → l' → l''
class PowerSet l ls | l → ls where
  powerSet :: l → ls
class IsEmpty l where
  isEmpty :: l → Bool
class Contained l l' b | l l' → b where
  contained :: l → l' → b
class ContainedEq l l' b | l l' → b where
  containedEq :: l → l' → b
class NoRepeats l where
  noRepeats :: l → Bool
```

Here, *HSet* is a predicate to test if a list does not have repeated elements.

For records, we have encoded operations for checking if a given label is used, for deleting and retrieving multiple record values, for updating one record with the content of another and

for modifying the value at a given label:

```
class HasLabel l r b | l r → b where
  hasLabel :: l → r → b
class ModifyAtLabel l v v' r r' | l r v' → v r' where
  modifyAtLabel :: l → (v → v') → r → r'
class LookupMany ls r vs | ls r → vs where
  lookupMany :: ls → r → vs
class DeleteMany ls r vs | ls r → vs where
  deleteMany :: ls → r → vs
class UpdateWith r s where
  updateWith :: r → s → r
```

Further details can be found in [25].

These elements together are sufficient to start the construction of our strongly typed model of relational databases.

Chapter 3

The basic layer – representation and relational operations

In this chapter, we will show how we represent databases in our model and how we lift classical relational algebra operations to our representation of tables. We will cover attribute types, primary keys, foreign keys and referential integrity constraints. This representation will be the basis for all operations defined in subsequent chapters.

3.1 Tables

A naive representation of databases, based on heterogeneous collections could be the following:

data $HList\ row \Rightarrow Table\ row = Table\ (Set\ row)$
data $TableList\ t \Rightarrow RDB\ t = RDB\ t$

class $TableList\ t$
instance $TableList\ HNil$
instance $(HList\ v, TableList\ t) \Rightarrow TableList\ (HCons\ (Table\ v)\ t)$

Thus, each table in a relational database would be modeled as a set of arbitrary-length tuples that represent its rows. A heterogeneous list in which each element is a table (as expressed by the *TableList* constraint) would constitute a relational database.

Such a representation is unsatisfactory for several reasons:

- Schema information is not represented. This implies that operations on the database may not respect the schema or take advantage of it, unless separate schema information would be fed to them.
- The choice of *Set* to collect the rows of a table does no justice to the fact that database tables are in fact mappings from key attributes to non-key attributes.

For these reasons, we prefer a more sophisticated representation that includes schema information and employs a *Map* datatype:

data $HeaderFor\ h\ k\ v \Rightarrow Table\ h\ k\ v = Table\ h\ (Map\ k\ v)$
class $HeaderFor\ h\ k\ v \mid h \rightarrow k\ v$

```
instance (
  AttributesFor a k, AttributesFor b v,
  HAppend a b ab, NoRepeats ab, Ord k
) ⇒ HeaderFor (a, b) k v
```

Thus, each table contains header information h and a map from key values to non-key values, each with types dictated by the header. The well-formedness of the header and the correspondence between the header and the value types is guarded by the constraint *HeaderFor*. It states that a header contains attributes for both the key values and the non-key values, and that attributes are not allowed to be repeated. The dependency $h \rightarrow k\ v$ indicates that the key and value types of the map inside a table are uniquely determined by its header.

3.2 Attributes

To represent attributes, we define the following datatype and accompanying constraint:

```
data Attribute t name
attribute = ⊥ :: Attribute t name
class AttributesFor a v | a → v
instance AttributesFor HNil HNil
instance AttributesFor a v
 ⇒ AttributesFor (HCons (Attribute t name) a) (HCons t v)
```

The type argument t specifies the column type for that attribute. The type argument *name* allows us to make attributes with identical column types distinguishable, for instance by instantiating it with different type-level naturals. Note that t and *name* are so-called *phantom* type arguments [19, 11], in the sense that they occur on the left-hand side of the definition only (in fact, the right-hand side is empty). Given this type definition we can for instance create the following attributes and corresponding types:

```
data ID;   atID   = attribute :: Attribute Int    (PERS, ID)
data NAME; atName = attribute :: Attribute String (PERS, NAME)
data PERS; person = ⊥ :: PERS
```

Note that no values of the attributes' column types (*Int* and *String*) need to be provided, since these are phantom type arguments. Using these attributes and a few more, a valid example table can be created as follows:

```
myHeader = (atID .*. HNil, atName .*. atAge .*. atCity .*. HNil)
myTable = Table myHeader $
  insert (12 .*. HNil) ("Ralf" .*. 23 .*. "Seattle" .*. HNil) $
  insert (67 .*. HNil) ("Oleg" .*. 17 .*. "Seattle" .*. HNil) $
  insert (50 .*. HNil) ("Dorothy" .*. 42 .*. "Oz" .*. HNil) $
  Map.empty
```

The $ operator is just function application with low binding strength; it allows us to write less parentheses.

The various constraints on the header of *myTable* are enforced by the Haskell type-checker, and the type of all components of the table is inferred automatically. For example, any at-

tempt to insert values of the wrong type, or value lists of the wrong length will lead to type check errors, as we can see in the following wrong length row insertion example:

```
myTable' =
  insert ("Oracle" .*. (1 :: Integer) .*. "Seattle" .*. HNil)
    $ myTable
```

The type-checker will report the error:

```
    No instances for (HDeleteAtHNat n HNil HNil,
                      HLookupByHNat n HNil Int,
                      HFind Int HNil n)
...
```

that reflects the missing attribute of type `Int`. It can be observed that such error messages are not very clear. Improving the readability of the reported type errors is left for future work.

In SQL, attributes can be declared with a user-defined DEFAULT value, and they can be declared not to allow NULL values. Our data constructor *Attribute* actually corresponds to attributes without user-defined default that do not allow null. To model the other variations[1], we have defined similar datatypes called *AttributeNull* and *AttributeDef* with corresponding instances for the *AttributesFor* class.

```
data AttributeNull t name
data AttributeDef t name = Default t

instance AttributesFor a v
    => AttributesFor (HCons (AttributeDef t nr) a) (HCons t v)
instance AttributesFor a v
    => AttributesFor (HCons (AttributeNull t nr) a) (HCons (Maybe t) v)
```

In SQL, there are also attributes with a system default value. For instance, integers have as default value 0. To represent these attributes, we have defined the following class and corresponding instances.

```
class Defaultable x where
  defaultValue :: x

instance Defaultable Int where
  defaultValue = 0

instance Defaultable String where
  defaultValue = ""
```

We can now define a table *yourTable*, that maps city names to country names, where the country attribute is defaultable.

```
data COUNTRY;
data CITIES;

atCity' = attribute :: Attribute String (CITIES, CITY)
atCountry :: AttributeDef String (CITIES, COUNTRY)
atCountry = Default "Afghanistan"

yourHeader = (atCity' .*. HNil, atCountry .*. HNil)
```

[1] We do not consider attributes that can be nullable and defaultable at the same time.

```
yourTable = Table yourHeader $
  insert ("Braga" .*. HNil) ("Portugal" .*. HNil) $
  Map.empty
```

Below, we will see examples of operations (such as insertion in a table) that benefit from the fact that *atCountry* is a defaultable attribute.

3.3 Foreign key constraints

Apart from headers of individual tables, we need to be able to represent schema information about relationships among tables. The *FK* type is used to specify foreign keys:

```
type FK fk t pk = (fk, (t, pk))
```

Here *fk* is the list of attributes that form a (possibly composite) foreign key and t and pk are, respectively, the name of the table to which it refers and the attributes that form its (possibly composite) primary key. As an example, we can specify the following foreign key relationship:

```
myFK = (atCity .*. HNil, (cities, (atCity' .*. HNil) .*. HNil))
```

Thus, the *myFK* constraint links the *atCity* attribute of the first table to the primary key *atCity'* of the second table.

3.4 Well-formedness for relational databases

To wrap up the example, we put the tables and constraint together into a record, to form a complete relational database:

```
myRDB = Record $
  cities  .=. (yourTable, Record HNil) .*.
  people .=. (myTable, Record $ myFK .*. HNil) .*. HNil
```

In this way, we model a relational database as a record where each label is a table name, and each value is a tuple of a table and the list of constraints of that table.

Naturally, we want databases to be well-formed. At schema level, this means we want all attributes to be unique, and we want foreign key constraints to refer to existing attributes and table names of the appropriate types. On the data instance level, we want referential integrity in the sense that all foreign keys should exist as primary keys in the related table. Such well-formedness can be captured by type-level and value-level predicates (classes with boolean member functions), and encapsulated in a data constructor:

```
class CheckRI rdb where
  checkRI :: rdb → Bool
class NoRepeatedAttrs rdb
data (NoRepeatedAttrs rdb, CheckRI rdb) ⇒ RDB rdb = RDB rdb
```

For brevity, we do not show the instances of the classes, nor the auxiliary classes and instances they use. We refer the reader to the source distribution of the report[2] for further

[2] http://wiki.di.uminho.pt/wiki/bin/view/PURe/CoddFish

details. The data constructor *RDB* encapsulates databases that satisfy our schema-level well-formedness demands upon which the *checkRI* predicate can be run to check for dangling references.

As an example, consider *myRDB* defined above. Notice that this relational database meets our well-formedness demands at the type level, since there are no repeated attributes and the foreign key refers to an existing primary key. However, at the value level, the city *Seattle* does not appear in *yourTable*. We can check this by using the data constructor *RDB* and the predicate *checkRI*:

```
> let _ = RDB myRDB
> checkRI myRDB
False
```

Using the data constructor *RDB* we can now define a function to insert a new table in a well-formed database, while guaranteeing that the new database is still well-formed:

$$
\begin{aligned}
&\mathit{insertTableRDB} :: (\\
&\quad \mathit{HExtend}\ e\ db\ db',\\
&\quad \mathit{DB}\ db, \mathit{DB}\ db'\\
&\quad) \Rightarrow e \rightarrow db \rightarrow db'\\
&\mathit{insertTableRDB}\ e\ db = e\ .*.\ db
\end{aligned}
$$

Here the predicate *DB* is used to gather the constraints of well-formedness, in order to avoid long constraints:

$$
\begin{aligned}
&\textbf{class}\ \mathit{DB}\ rdb\\
&\textbf{instance}\ (\mathit{NoRepeatedAttributes}\ rdb, \mathit{CheckRI}\ rdb) \Rightarrow \mathit{DB}\ rdb
\end{aligned}
$$

The *HExtend* constraint guarantees that the produced record is well-formed, *i.e.* does not contain duplicate labels. Then, an attempt to insert a table whose label already exists leads to a type error.

3.5 Row operations

The operations which we will define in the chapters to follow will often resort to auxiliary operations on rows, which we model as a record that has attributes as labels. To compute the type of the record from the table header, we employ a type-level function:

$$
\begin{aligned}
&\textbf{class}\ \mathit{Row}\ h\ k\ v\ r \mid h \rightarrow k\ v\ r\ \textbf{where}\\
&\quad \mathit{row} :: h \rightarrow k \rightarrow v \rightarrow r\\
&\quad \mathit{unRow} :: h \rightarrow r \rightarrow (k, v)\\
&\textbf{instance}\ (\\
&\quad \mathit{HeaderFor}\ (a, b)\ k\ v, \mathit{HAppend}\ a\ b\ ab, \mathit{HAppend}\ k\ v\ kv,\\
&\quad \mathit{HZip}\ ab\ kv\ l, \mathit{HBreak}\ kv\ k\ v\\
&)\ \Rightarrow \mathit{Row}\ (a, b)\ k\ v\ (\mathit{Record}\ l)\\
&\quad \textbf{where}\\
&\qquad \mathit{row}\ (a, b)\ k\ v = \mathit{Record}\ \$\ \mathit{hZip}\ (\mathit{hAppend}\ a\ b)\ (\mathit{hAppend}\ k\ v)\\
&\qquad \mathit{unRow}\ (a, b)\ (\mathit{Record}\ l) = \mathit{hBreak}\ \$\ \mathit{snd}\ \$\ \mathit{hUnzip}\ l
\end{aligned}
$$

Thus, the record type is computed by zipping (pairing up) the attributes with the corresponding column types. The value-level function *row* takes a header and corresponding key and non-key values as argument, and zips them into a row. The converse *unRow* is available as well. For the `Dorothy` entry in *myTable*, for example, the following row would be derived:

```
r = Record $
      atID .=. 50 .*. atName .=. "Dorothy" .*.
      atAge .=. 42 .*. atCity .=. "Oz" .*. HNil
```

Now, we can model value access with record lookup. The projection of the row through a subset of the header will be computed using the record function *LookupMany*. For instance, for the row presented above, if we are only interested in the name and age attributes we can project the row as follows:

```
> lookupMany (atName .*. atAge .*. HNil) r
HCons "Dorothy" (HCons 42 HNil)
```

Another important operation on rows, mainly motivated by semantics of SQL operations, is the reordering of a list of values according to a list of attributes, that we will present next.

3.5.1 Reordering row elements according to a given header

We can reorder the row values according to a given column specification of the row.

If the row list is longer than the attribute list we return not only the reordered list but also the remaining values, which can be either used or ignored in other operations.

```
class LineUp h v v' r | h v → v' r
where
  lineUp :: h → v → (v', r)
instance LineUp HNil vs HNil vs where
  lineUp _ vs = (HNil, vs)
instance (
  AttributeFor at v, HFind v vs n,
  HLookupByHNat n vs v, HDeleteAtHNat n vs vs',
  LineUp as vs' vs'' r
) ⇒ LineUp (HCons at as) vs (HCons v vs'') r where
  lineUp (HCons a as) vs = (HCons v vs'', r)
    where
      n = hFind (mkDummy a) vs
      v = hLookupByHNat n vs
      vs' = hDeleteAtHNat n vs
      (vs'', r) = lineUp as vs'
```

The *mkDummy* function used is just to *extract* the type of an attribute.

```
mkDummy :: AttributeFor at v ⇒ at → v
mkDummy _ = ⊥
class AttributeFor a t | a → t
instance AttributeFor (AttributeNull t nr) (Maybe t)
```

instance *AttributeFor* (*AttributeDef t nr*) *t*
instance *AttributeFor* (*Attribute t nr*) *t*

For the tuple $3 .*. Ralf .*. Seattle .*. HNil$, we can extract the values for *atID* and *atName* as follows:

```
> lineUp (atID .*. atName .*. HNil) ((3::Int) .*. "Ralf" .*. "Seattle" .*. HNil)
(HCons 3 (HCons "Ralf" HNil),HCons "Seattle" HNil)
```

Notice that the order of attributes with the same type is important. For instance if the tuple presented above had `Seattle` before `Ralf`, the former would be returned as being the value for *atName*.

```
> lineUp (atID .*. atName .*. HNil) ((3::Int) .*. "Seattle" .*. "Ralf" .*. HNil)
(HCons 3 (HCons "Seattle" HNil),HCons "Ralf" HNil)
```

To overcome this limitation we will later in this section present an instance of *LineUp* where the argument v will be a record instead of a flat list.

Another important remark is that if the specification list is longer than the row, we can fill those *empty* spaces with *null/default* values (choice which is motivated by semantics of SQL operations), which lead to the following implementation:

class *LineUpPadd b row vs r* | *b row* → *vs r* **where**
  lineUpPadd :: $b \to row \to (vs, r)$

instance *LineUpPadd HNil vs HNil vs* **where**
  $lineUpPadd \_\ vs = (HNil, vs)$

instance (
  AttributeFor at v, *HMember v vs b*,
  LineUpPadd' b at vs v r, *LineUpPadd as r vs' r'*
) ⇒ *LineUpPadd* (*HCons at as*) *vs* (*HCons v vs'*) *r'* **where**
  $lineUpPadd\ (HCons\ a\ as)\ vs = (HCons\ v'\ vs', r')$
    where $v = mkDummy\ a$
      $(v', r) = lineUpPadd'\ (hMember\ v\ vs)\ a\ vs$
      $(vs', r') = lineUpPadd\ as\ r$

The previous class calculates a (type-level) boolean that has value *HTrue*, if there is a value in the row corresponding to the attribute in the specification list, and *HFalse* otherwise. This boolean guides the action of the following auxiliary function.

class *LineUpPadd' b at r v r'* | *b at r* → *v r'*
where
  lineUpPadd' :: $b \to at \to r \to (v, r')$

instance *LineUpPadd' HFalse* (*NullableAttribute v nr*) *r* (*Maybe v*) *r* **where**
  $lineUpPadd'\ b\ at\ r = (Nothing, r)$

instance *LineUpPadd' HFalse* (*DefaultableAttribute v nr*) *r v r* **where**
  $lineUpPadd'\ b\ (Default\ v)\ r = (v, r)$

instance
  Defaultable v
  ⇒ *LineUpPadd' HFalse* (*Attribute v nr*) *r v r* **where**
  lineUpPadd' b at r

$= (defaultValue, r)$

instance (
  AttributeFor at v, *HFind v vs n*,
  HLookupByHNat n vs v, *HDeleteAtHNat n vs vs′*
) ⇒ *LineUpPadd′ HTrue at vs v vs′* **where**
  lineUpPadd′ b at vs = (*v′*, *vs′*)
    where *v* = *mkDummy at*
          n = *hFind v vs*
          v′ = *hLookupByHNat n vs*
          vs′ = *hDeleteAtHNat n vs*

When the value for a given attribute is not present in the list there are three situations where padding can be performed: the attribute is nullable (*Nothing* is inserted); the attribute is user declared defaultable (the default value associated is inserted); the attribute is neither nullable nor user declared defaultable, but it has a system default value (this value is inserted).
As an example consider the tuple *3 .∗. HNil* and imagine we want, as in the previous example, to extract the values for *atID* and *atName*.

```
> lineUpPadd (atID .*. atName .*. HNil) ((3::Int) .*. HNil)
(HCons 3 (HCons "" HNil),HNil)
```

The system default value for Strings was returned as value for *atName*. To wrap up the *LineUpPadd* definition we present the instance for a row presented as a record, where attributes identifiers are linked to values.

instance (
  AttributeFor at v,
  HMember (*at*, *v*) *row b*,
  LineUpPadd′ b at (*Record row*) *v row′*,
  LineUpPadd ats row′ vs r
) ⇒ *LineUpPadd* (*HCons at ats*) (*Record row*) (*HCons v vs*) *r* **where**
  lineUpPadd (*HCons at ats*) (*Record row*) = (*HCons v vs*, *r*)
    where (*v*, *row′*) = *lineUpPadd′* (*hMember* (*at*, *mkDummy at*) *row*) *at* (*Record row*)
          (*vs*, *r*) = *lineUpPadd ats row′*

This instance will overcome limitations in the order of values for attributes with same type, as refeered above in the first example presented.

```
> lineUpPadd (atID .*. atName .*. HNil)
   (Record $ atID .=. (3::Int) .*.atCity .=.  "Seattle" .*. atName .=. "Ralf" .*.HNil)
(HCons 3 (HCons "Ralf" HNil),Record{CITY="Seattle"})
```

For *LineUp* we still need an instance for rows presented as records and the specification list represented as a pair:

instance (
  LookupMany a (*Record row*) *ks*,
  DeleteMany a (*Record row*) (*Record row′*),
  LineUpPadd b (*Record row′*) *vs* (*Record r*)
) ⇒ *LineUp* (*a*, *b*) (*Record row*) (*ks*, *vs*) (*Record r*) **where**
  lineUp (*a*, *b*) *row* = ((*ks*, *vs*), *r*)

where $ks = lookupMany\ a\ row$
$row' = deleteMany\ a\ row$
$(vs, r) = lineUpPadd\ b\ row'$

The first component of the pair corresponds to key attributes, where padding cannot be performed, and the second component to non-key attributes, subject to padding with null or default values.

The key attributes cannot be underspecified in the value list. If we try to line up a row that does not contain values to all key attributes, the type checker will report an error:

```
> lineUp (atID .*. HNil, atName .*. HNil) (Record $ HNil)
<interactive>:1:0:
    No instance for (HasField (Attribute Int (People, ID)) HNil v)
      arising from use of ‘lineUp’ at <interactive>:1:0-5
```

The non-key attributes are subject to padding and an error will only occur when such attribute is neither defaultable nor nullable.

3.6 Relational Algebra

Many operations, known from relational algebra [27], that are typically available for Sets can be lifted to Tables. The main difference in the datatype for tables is that we have explicit separation between keys and non-keys instead of flat tuples.

In this section, we will present the lifting of traditional operations available for Sets to tables.

3.6.1 Single table operations

There are several operations on single tables that are often used, such as projecting all rows of the table accordingly a column specification, adding rows, and selecting specific rows. We will proceed to describe the implementation of such operations in our model.

Projection

Concerning projection, we keep the specified columns. Note that the order of columns can deviate from order in the table header and that the column specification is a flat list. So, the split between key and non-key columns is not exposed in the argument. In the result, all columns are keys and so we present it not in a *Table*, but in a *ResultSet*.

$project :: ($
$\quad HeaderFor\ h\ k\ v, RESULTSET\ ab'\ kv',$
$\quad LookupMany\ ab'\ r\ kv', Row\ h\ k\ v\ r$
$\quad) \Rightarrow ab' \rightarrow Table\ h\ k\ v \rightarrow ResultSet\ ab'\ kv'$
$project\ ab'\ (Table\ h\ m) = ResultSet\ ab'\ m'$
$\quad$ **where** $m' = Map.foldWithKey\ worker\ Set.empty\ m$
$\qquad worker\ k\ v\ mp = Set.insert\ (projectRow\ h\ ab'\ (k, v))\ mp$

Should we want to present the result in a *Table*, we would have as result type $Table\ h\ k\ HNil$ (we can observe that the types $ResultSet\ h\ v$ and $Table\ h\ v\ HNil$ are isomorphic). We define *ResultSet* as follows.

$\textbf{data}\ (AttributesFor\ a\ v, Ord\ v) \Rightarrow ResultSet\ a\ v = ResultSet\ a\ (Set\ v)$

To reduce the number of constraints when using *ResulSet*, we gather the previous constraints in a class, which we have already used in the *project* definition.

$\textbf{class}\ (AttributesFor\ a\ v, Ord\ v) \Rightarrow RESULTSET\ a\ v$

Adding a single row to a table

This is a basic operation, corresponding to a simple insertion in a map.

$$
\begin{aligned}
&add :: HeaderFor\ h\ ks\ vs \\
&\quad \Rightarrow (ks, vs) \rightarrow Table\ h\ ks\ vs \rightarrow Table\ h\ ks\ vs \\
&add\ (k, v)\ (Table\ h\ m) = Table\ h\ m' \\
&\quad \textbf{where}\ m' = Map.insert\ k\ v\ m
\end{aligned}
$$

Due to the *Map* definition in Haskell, if the key already exists in the map, the associated value is replaced with the new value we are inserting.

Filter and Fold

Concerning single table operations context, we will describe two more operations – *filter* and *fold*. All the libraries for collection datatypes in Haskell, such as lists, sets and maps, include these general operations. We can regard our tables as collections and so it makes sense to define them.

Firstly, we describe a function that filters rows in a table using a predicate on keys and values. In the result table only the rows for which the predicate holds are kept. We use the standard *filterWithKey* defined in the *Map* Haskell library. The *filter* for maps defined in the referred library only uses the value information. Since we are interested in performing the filter using all row information we need the function *filterWithKey*.

$$
\begin{aligned}
&filter :: HeaderFor\ h\ ks\ vs \\
&\quad \Rightarrow ((ks, vs) \rightarrow Bool) \rightarrow Table\ h\ ks\ vs \rightarrow Table\ h\ ks\ vs \\
&filter\ p\ (Table\ s\ m) = Table\ s\ m' \\
&\quad \textbf{where}\ m' = Map.filterWithKey\ (\lambda x\ y \rightarrow p\ (x, y))\ m
\end{aligned}
$$

Secondly, we implement the fold over a *Table*, which allows the computation of a value based in all values contained in the tables. We again use the *foldWithKey* defined in the *Map* library.

$$
\begin{aligned}
&fold :: HeaderFor\ h\ k\ v \\
&\quad \Rightarrow (h \rightarrow k \rightarrow v \rightarrow b \rightarrow b) \rightarrow b \rightarrow Table\ h\ k\ v \rightarrow b \\
&fold\ f\ a\ (Table\ h\ m) = a' \\
&\quad \textbf{where}\ a' = Map.foldWithKey\ (\lambda k\ v\ a \rightarrow f\ h\ k\ v\ a)\ a\ m
\end{aligned}
$$

This operation can now be used, for instance, to generate a set with flat tuples from a table. For *myTable* defined in chapter 3 the result would be as follows:

```
> fold (\h k v s -> insert (hAppend k v) s) empty myTable
{
```

```
 HCons 12 (HCons "Ralf" (HCons Nothing (HCons "Seattle" HNil))),
 HCons 50 (HCons "Dorothy" (HCons (Just 42) (HCons "Oz" HNil))),
 HCons 67 (HCons "Oleg" (HCons (Just 17) (HCons "Seattle" HNil)))
}
```

This is a simple operation. The *fold* function can be used to define more complex functions on tables.

3.6.2 Multiple table operations

Union, Intersection and Difference

Union, intersection and difference are classical operations in relational algebra imported from set theory. In our representation, the definition of these operations is just a lift of the corresponding *Map* operations.

$$unionT :: HeaderFor\ h\ k\ v \Rightarrow Table\ h\ k\ v \rightarrow Table\ h\ k\ v \rightarrow Table\ h\ k\ v$$
$$unionT\ (Table\ h\ m)\ (Table\ \_\ m') = Table\ h\ (Map.union\ m\ m')$$
$$intersectionT :: HeaderFor\ h\ k\ v \Rightarrow Table\ h\ k\ v \rightarrow Table\ h\ k\ v \rightarrow Table\ h\ k\ v$$
$$intersectionT\ (Table\ h\ m)\ (Table\ \_\ m') = Table\ h\ (Map.intersection\ m\ m')$$
$$differenceT :: HeaderFor\ h\ k\ v \Rightarrow Table\ h\ k\ v \rightarrow Table\ h\ k\ v \rightarrow Table\ h\ k\ v$$
$$differenceT\ (Table\ h\ m)\ (Table\ \_\ m') = Table\ h\ (Map.difference\ m\ m')$$

Notice that we could define more complex union, intersection and difference operations, if we allowed that the two table arguments could have different headers (as in [29]). Then, we should keep one of the headers and reorder the elements of the other table so they agree in type and could be inserted in the resulting table.

In this chapter we have shown our strongly typed representation of databases and basic operations on tables. In the next chapter, we will present the reconstruction of several SQL statements.

Chapter 4

The SQL layer – data manipulation operations

A faithful model of the SQL language [20] will require a certain degree of flexibility regarding input parameters. When performing *insertion* of values into a table, for example, the list of supplied values does not necessarily correspond 1-to-1 with the columns of the table. Values may be missing, and a list of column specifications may be provided to guide the insertion. We will need to make use of various auxiliary heterogeneous data structures and type-level functions to realize the required sophistication.

The interface provided by the SQL language shields off the distinction between key attributes and non-key attributes which are present in the underlying tables. This distinction is relevant for the behavior of constructs like *join*, *distinct* selection, *grouping*, and more. But at the language level, rows are presented as flat tuples without explicit distinction between keys and non-keys. As a result, we will need to 'marshal' between pairs of lists and concatenated lists, again at the type level.

In this chapter, we will present the reconstruction of several SQL statements, such as select, insert, delete and grouping, and database operations.

4.1 The WHERE clause

```
<where clause> ::= WHERE  <search condition>
```

Table 4.1: Syntax for the WHERE clause (according SQL-92 BNF Grammar)

Various SQL statements can contain a WHERE clause which specifies a predicate on rows. Only those rows that satisfy the predicate are taken into account. The predicate can be formulated in terms of a variety of operators that take row values as operands. These row values are accessed via their corresponding column names.

In section 3.5, we presented how to model a row as a record that has attributes as labels. A predicate is then a boolean function over that record. Recall the row derived for the `Dorothy` entry in *myTable*:

$r = Record\ \$$
$\quad atID\ .=.\ 50\ .*.\ atName\ .=.\ \texttt{"Dorothy"}\ .*.$
$\quad atAge\ .=.\ 42\ .*.\ atCity\ .=.\ \texttt{"Oz"}\ .*.\ HNil$

A predicate over such a row might look as follows:

$isOzSenior = \lambda r \rightarrow (r\ .!.\ atAge) > 65 \wedge (r\ .!.\ atCity) \equiv \texttt{"Oz"}$

The type of the predicate is inferred automatically:

$isOzSenior :: (\\ \quad HasField\ (Attribute\ Int\ \texttt{AGE})\ r\ Int,\\ \quad HasField\ (Attribute\ String\ \texttt{CITY})\ r\ String\\ ) \Rightarrow r \rightarrow Bool$

Interestingly enough, this type is valid for any row exhibiting the *atAge* and *atCity* attributes. If other columns are joined or projected away, the predicate will still type-check and behave correctly. We will encounter such situations below.

4.2 The DELETE statement

```
<delete statement> ::= DELETE   FROM  <table name>
                                [  WHERE  <search condition> ]
```

Table 4.2: Syntax for the DELETE statement (according SQL-92 BNF Grammar)

Now that the WHERE clause is in place, we can turn to our first statement. The DELETE statement removes all rows from a table that satisfy the predicate in its WHERE clause. We model this statement via the folding function for maps:

$delete :: (HeaderFor\ h\ k\ v, Row\ h\ k\ v\ r)\\ \Rightarrow Table\ h\ k\ v \rightarrow (r \rightarrow Bool) \rightarrow Table\ h\ k\ v\\ delete\ (Table\ h\ m)\ p = Table\ h\ m'$
where
$\quad m' = foldWithKey\ del\ Map.empty\ m$
$\quad del\ k\ v\\ \quad\quad \mid p\ (row\ h\ k\ v) = id\\ \quad\quad \mid otherwise = insert\ k\ v\\ foldWithKey :: (k \rightarrow a \rightarrow b \rightarrow b) \rightarrow b \rightarrow Map\ k\ a \rightarrow b$

Here we use Haskell's *guarded* equation syntax. Thus, only rows that fail the predicate pass through to the result map. The following example illustrates this operation. We delete all senior people from Oz in the table *myTable* defined in chapter 3.

```
*Data.HDB.Example Data.HDB.Databases> delete myTable isOzSenior
Table (HCons ID HNil,HCons NAME (HCons AGE (HCons CITY HNil))) {
    HCons 12 HNil:=HCons "Ralf" (HCons Nothing (HCons "Seattle" HNil)),
    HCons 50 HNil:=HCons "Dorothy" (HCons (Just 42) (HCons "Oz" HNil)),
    HCons 67 HNil:=HCons "Oleg" (HCons (Just 17) (HCons "Seattle" HNil))
}
```

Since the only individual from Oz is younger than 65, no rows satisfy the given predicate and the same table is returned.

To illustrate the fact that *isOzSenior* is valid for any row that has *atAge* and *atCity* attributes, consider the following delete applied to *myTable* after projecting out the attribute *atName*.

```
> delete (projectValues (atAge .*. atCity .*. HNil) myTable) isOzSenior
Table (HCons ID HNil,HCons AGE (HCons CITY HNil)) {
    HCons 12 HNil:=HCons Nothing (HCons "Ralf" HNil),
    HCons 50 HNil:=HCons (Just 42) (HCons "Dorothy" HNil),
    HCons 67 HNil:=HCons (Just 17) (HCons "Oleg" HNil)
}
```

4.3 The UPDATE statement

```
<update statement> ::=
         UPDATE <table name> SET <set clause list>
                              [ WHERE <search condition> ]

<set clause list> ::= <set clause> [ { <comma> <set clause> } ... ]
<set clause> ::= <object column> <equals operator> <update source>

<object column> ::= <column name>
<update source> ::= <value expression> | <null specification> | DEFAULT
```

Table 4.3: Syntax for the UPDATE statement (according SQL-92 BNF Grammar)

The UPDATE statement involves a SET clause that assigns new values to selected columns. A record is again an appropriate structure to model these assignments. Updating of a row according to column assignments will then boil down to updating one record with the values from another, possibly smaller record. The record operation *updateWith* can be used for this.

$$
\begin{aligned}
&\mathit{update} :: (\mathit{HeaderFor}\ h\ k\ v, \mathit{Row}\ h\ k\ v\ r, \mathit{UpdateWith}\ r\ s) \\
&\Rightarrow \mathit{Table}\ h\ k\ v \to s \to (r \to \mathit{Bool}) \to \mathit{Table}\ h\ k\ v \\
&\mathit{update}\ (\mathit{Table}\ h\ m)\ s\ p = \mathit{Table}\ h\ (\mathit{foldWithKey}\ \mathit{upd}\ \mathit{empty}\ m) \\
&\quad \textbf{where} \\
&\qquad \mathit{upd}\ k\ v \\
&\qquad\quad \mid p\ r = \mathit{insert}\ k'\ v' \\
&\qquad\quad \mid \mathit{otherwise} = \mathit{insert}\ k\ v \\
&\qquad\quad \textbf{where}\ r = \mathit{row}\ h\ k\ v \\
&\qquad\qquad\quad (k', v') = \mathit{unRow}\ h\ \$\ \mathit{updateWith}\ r\ s
\end{aligned}
$$

Thus, when a row satisfies the predicate, an update with new values is applied to it, and the updated row is inserted into the result. Note that the *UpdateWith* constraint enforces that the list of assignments only sets attributes present in the header, and sets them to values of the proper types. Assignment to an attribute that does not occur in the header of the table, or assignment of a value of the wrong type will lead to a type check error.

4.4 The INSERT statement

```
<insert statement> ::= INSERT INTO <table name>
                                   <insert columns and source>

<insert columns and source> ::=
    [ <left paren> <insert column list> <right paren> ] <query expression>
    |  DEFAULT VALUES

<insert column list> ::= <column name list>
```

Table 4.4: Syntax for the multiple row INSERT statement (according SQL-92 BNF Grammar)

A single row can be inserted into a table by specifying its values in a VALUES clause. Multiple rows can be inserted by specifying a sub-query that delivers a list of suitable rows. In either case, a list of columns can be specified to properly align the values for each row.

Let us first analyze how to define the insert operation in case the column specification is not supplied.

We must reorder the values (using the operation *LineUp* previously defined) and split the row according to key and non-key values, in compliance with the given header. The *split* is defined as follows:

```
class Ord kvs ⇒ Split vs h kvs nkvs | vs h → kvs nkvs
where
  split :: vs → h → (kvs, nkvs)
instance (
  LineUp kas vs kvs r,
  LineUpPadd nkas r nkvs HNil,
  Ord kvs
) ⇒ Split vs (kas, nkas) kvs nkvs where
  split vs (kas, nkas) = (kvs, nkvs)
    where (kvs, r) = lineUp kas vs
          (nkvs, _) = lineUpPadd nkas r
```

We resort to class *Split* in order to avoid repeating the list of constraints in functions that use *split* and add the *Ord* constraint on key types, to ensure they can be used in a map.

We stress that wherever attributes of the same type occur, the values for these attributes need to be supplied in the right order. Having this auxiliary operation available makes it easy to define the insert operation.

```
insert :: (
  HeaderFor h ks vs,
  Split vs' h ks vs
) ⇒ vs' → Table h ks vs → Table h ks vs
insert vs (Table h mp) = Table h mp'
where mp' = Map.insert kvs nkvs mp
      (kvs, nkvs) = split vs h
```

Recall *yourTable* defined in chapter 3, where *atCountry* is a defaultable attribute.

```
yourHeader = (atCity' .*. HNil, atCountry .*. HNil)
yourTable = Table yourHeader $
  insert ("Braga" .*. HNil) ("Portugal" .*. HNil) $
  Map.empty
```

We can now insert the city `Porto`, without specifying its country, and the default value will be inserted.

```
> insert ("Porto".*.HNil) yourTable
Table (HCons CITY HNil,HCons COUNTRY HNil) {
    HCons "Braga" HNil:=HCons "Portugal" HNil,
    HCons "Porto" HNil:=HCons "Afghanistan" HNil
}
```

Now, let us define the insert operation in full, where the column specification is not mandatory. The single-row variant can be specified as follows:

```
class InsertValues t as vs where
  insertValues :: t → as → vs → t
instance (
  HeaderFor h k v,
  Split vs h k v
  ) ⇒ InsertValues (Table h k v) HNil vs where
  insertValues (Table h m) _ vs = Table h m'
    where m' = Map.insert k v m
          (k, v) = split vs h
instance (
  HeaderFor h ks vs,
  HZip (HCons a as) vs' row,
  LineUp h (Record row) (ks, vs) (Record HNil)
  ) ⇒ InsertValues (Table h ks vs) (HCons a as) vs' where
  insertValues (Table h mp) as vs = Table h mp'
    where mp' = Map.insert kvs nkvs mp
          (kvs, nkvs) = fst $ lineUp h (Record $ hZip as vs)
```

The first instance controls the case where no align list of columns is specified, which corresponds to the *insert* operation defined above.

In the second instance, the *hZip* function is used to pair up the list of columns with the list of values. The *ReOrd* constraint expresses the fact that the resulting list of column-value pairs can be permuted into a properly ordered row for the given table.

The multi-row insert accepts as argument the result list of a nested `SELECT` query. Though selection itself will be defined only in the next section, we can already reveal the type of result lists.

```
data AttributesFor a x ⇒ ResultList a x = ResultList a [x]
```

Thus, a result list is a list of rows augmented with the meta-data of that row. Unlike our *Table* datatype, result lists make no distinction between keys and values, and rows may occur more than once.

Now, multi-row insertion can be specified as a fold over rows in a *ResultList*:

$$
\begin{aligned}
&\mathit{insertResultList} :: (\\
&\quad \mathit{AttributesFor}\ a'\ v', \mathit{HZip}\ a'\ v'\ r, \mathit{HeaderFor}\ h\ k\ v,\\
&\quad \mathit{LineUp}\ h\ (\mathit{Record}\ r)\ (k, v)\ (\mathit{Record}\ \mathit{HNil})\\
&) \Rightarrow a' \to \mathit{ResultList}\ a'\ v' \to \mathit{Table}\ h\ k\ v \to \mathit{Table}\ h\ k\ v\\
&\mathit{insertResultList}\ \mathit{tbl}\ \mathit{as}\ (\mathit{ResultList}\ a\ v)\\
&\quad = \mathit{foldr}\ (\lambda vs\ t \to \mathit{insertValues}\ t\ \mathit{as}\ vs)\ \mathit{tbl}\ v
\end{aligned}
$$

Note that the meta-information of the inserted result list is ignored. The specified column list is used instead. An insert operation with a more precise type than that of SQL one can also be defined:

$$
\begin{aligned}
&\mathit{insertTable} :: (\\
&\quad \mathit{HZip}\ b'\ v'\ l', \mathit{LookupMany}\ b\ l'\ v, \mathit{ReOrd}\ k'\ k,\\
&\quad \mathit{HF}\ (a, b)\ k\ v, \mathit{HF}\ (a', b')\ k'\ v'\\
&) \Rightarrow \mathit{Table}\ (a', b')\ k'\ v' \to \mathit{Table}\ (a, b)\ k\ v \to \mathit{Table}\ (a, b)\ k\ v\\
&\mathit{insertTable}\ (\mathit{Table}\ (a', b')\ m)\ (\mathit{Table}\ (a, b)\ m') = \mathit{Table}\ (a, b)\ m''\\
&\quad \textbf{where}\\
&\qquad m'' = \mathit{Map.foldWithKey}\ \mathit{addRow}\ m'\ m\\
&\qquad \mathit{addRow}\ k'\ v'\\
&\qquad\quad = \mathit{Map.insert}\ (\mathit{reOrd}\ k')\ (\mathit{lookupMany}\ b\ \$\ \mathit{hZip}\ b'\ v')
\end{aligned}
$$

This function enforces that keys get inserted into keys (resp. values into values) all with the right types, of course. Only values of the inserted rows may be ignored, while all keys must be preserved. This guarantees that none of the inserted rows gets lost due to keys that coincide after restriction.

4.5 The SELECT statement

```
<query specification> ::=
         SELECT [ <set quantifier> ] <select list> <table expression>

<set quantifier> ::= DISTINCT  |  ALL

<select list> ::=
         <asterisk>
        |<select sublist> [ { <comma> <select sublist> }... ]

<table expression> ::= <from clause> [ <where clause> ]
                       [ <group by clause> ] [ <having clause> ]

<from clause> ::= FROM <table reference> [{<comma> <table reference>}...]
<having clause> ::= HAVING  <search condition>
```

Table 4.5: Syntax (incomplete) for the select statement (according SQL-92 BNF Grammar)

The SELECT statement is a bundle of several functionalities, which include projection (column selection) and cartesian product (exhaustive combination of rows from several tables). In addition to these, clauses may be present for filtering, joining, grouping, and ordering.

Cartesian product on maps can be defined with two nested folds over maps:

```
productM :: (HAppend k k' k'', HAppend v v' v'', Ord v'')
 ⇒ Map k v → Map k' v' → Map k'' v''
productM m m'
 = foldWithKey (λk v m'' → foldWithKey add m'' m') empty m
   where
     add k' v' m'' = insert (hAppend k k') (hAppend v v') m''
```

As the type constraints indicate, the key tuples of the argument maps are appended to each other, and so are the non-key tuples. This operation can be lifted to tables, and then to lists of tables:

```
productT :: (
   HF (a, b) k v, HF (a', b') k' v', HF (a'', b'') k'' v'',
   HA a a' a'', HA b b' b'', HA k k' k'', HA v v' v''
) ⇒ Table (a, b) k v → Table (a', b') k' v' → Table (a'', b'') k'' v''
productT (Table (a, b) m) (Table (a', b') m') = Table h'' m''
   where
     h'' = (hAppend a a', hAppend b b')
     m'' = product' m m'
```

Above we have abbreviated *HeaderFor* and *HAppend* to *HF* and *HA*. Since the SELECT statement allows for any number of tables to be involved in a cartesian product, we lift the binary product to a product over an arbitrary-length tuple of tables, using a type-level function:

```
class Products ts t | ts → t where
   products :: ts → t
instance Products (t :*: HNil) t where
   products (HCons t _) = t
instance (...)
 ⇒ Products ((Table (a, b) k v) :*: (Table (a', b') k' v') :*: ts) t
   where
     products (HCons t ts) = productT t (products ts)
```

Thus, the binary product is applied successively to pairs of tables. For brevity, we elided the lengthy but straightforward type constraints of the second instance of *Products*.

Now that cartesian product over lists of tables is in place, we can specify selection:

```
select :: (
   Products ts (Table h k v), HeaderFor h k v,
   Row h k v r, LookupMany a r x, AttributesFor a x, Ord x,
   HZip a x r', LookupMany b (Record r') y, Ord y, IsEmpty b
) ⇒ Bool → a → ts → (r → Bool) → b → ResultList a x
select distinct a ts p b = ResultList a $ uniq $ sort $ proj $ fltr m
   where
     Table h m = products ts
```

```
    fltr = filterWithKey (λk v → p $ row h k v)
    proj = foldWithKey flt []
    flt k v l = lookupMany a (row h k v) : l
    sort = if isEmpty b then id else (qsort ∘ cmp) b
    cmp b v v' = lkp v < lkp v'
      where lkp x = lookupMany b (Record $ hZip a x)
    uniq = if distinct then rmDbls else id
class IsEmpty l where isEmpty :: l → Bool
rmDbls :: [a] → [a]
qsort :: (a → a → Bool) → [a] → [a]
filterWithKey :: Ord k ⇒ (k → a → Bool) → Map k a → Map k a
```

The first argument corresponds to the presence of the DISTINCT keyword, and determines whether doubles will be removed from the result. The second argument lists the specified columns, to be used in the projection. The third argument represents the FROM clause, from which the *Products* constraint computes a table with type *Table h k v*. The last argument represents the WHERE clause, which contains a predicate on rows from that table. This is expressed by the *Row* constraint. Also, the selected columns must be present in these rows, which is guaranteed by the *LookupMany* constraint for multiple label lookup from a record.

As can be gleaned from the body of the *select* function, the cartesian product is computed first. Then filtering is performed with the predicate. The filtered map is folded into a list where each row is subject to flattening (from pair of keys and non-key values to a flat list of values), and to projection. The resulting list of tuples is passed through *sort* and *uniq*, which default to the do-nothing function *id*. However, if distinct rows were requested, *uniq* removes doubles, and if columns were specified to order by, then *sort* invokes a sorting routine that compares pairs of rows after projecting them through these columns (with *lookupMany*). As an example of using the *select* operation in combination with *insertResultList*, consider the following nested query:

```
insertResultList
  (atCity' .*. HNil)
  (select True (atCity .*. HNil) (myTable .*. HNil) isOzJunior HNil)
  yourTable
```

This produces the following table:

```
Table (CITY .*. HNil, COUNTRY .*. HNil)
  {Braga .*. HNil := Portugal .*. HNil,
   Oz .*. HNil    := Afghanistan .*. HNil}
```

Note that the result list produced by the nested *select* is statically checked and padded to contain appropriate columns to be inserted into *yourTable*. If the attribute AGE would be selected, for instance, the type-checker would complain. Since the nested *select* yields only cities, the declared default gets inserted in the country column.

Although the result list of a selection can be fed into the *insertResultList* operation, it cannot be fed into the key-preserving *insertTable*. For that the closing projection step of the select should be key-preserving and produce a table. The following restricted project does the trick:

```
projectValues :: (
   HF (a, b) k v, HF (a, b') k v', LineUpPadd b' v v' r
) ⇒ b' → Table (a, b) k v → Table (a, b') k v'
projectValues b' (Table (a, b) m) = Table (a, b') m'
   where m' = Map.map (fst ∘ lineUpPadd b') m
```

As the types show, keys will not be lost.

4.6 The JOIN clause

```
<qualified join> ::=
         <table reference>
         [ NATURAL ] [ <join type> ] JOIN
         <table reference> [ <join specification> ]

<join type> ::= INNER
              | <outer join type> [ OUTER ]
              | UNION

<outer join type> ::= LEFT | RIGHT | FULL

<join specification> ::= <join condition> | <named columns join>

<join condition> ::= ON <search condition>
```

Table 4.6: Syntax for the JOIN clause (according SQL-92 BNF Grammar)

The SQL language allows tables to be joined in several different ways, in addition to the cartesian product. We will define the *inner* join, where foreign keys in one table are linked to primary keys of a second table. On maps, the definition is as follows:

```
joinM :: (HAppend k' v' kv', HAppend v kv' vkv', Ord k, Ord k'
) ⇒ (k → v → k') → Map k v → Map k' v' → Map k vkv'
joinM on m m' = foldWithKey worker Map.empty m
   where
      worker k v m'' = maybe m'' add (lookup k' m')
         where
            k' = on k v
            add v' = insert k (hAppend v (hAppend k' v')) m''
lookup :: Ord k ⇒ k → Map k a → Maybe a
```

As the types and constraints indicate, the resulting map inherits its key type from the first argument map. Its value type is the concatenation of the value type of the first argument, and both key and value type of the second argument. A parameter *on* specifies how to obtain from each row in the first table a key for the second. The joined table is constructed by folding over the first. At each step, a key for the second table is computed with *on*. The value for that

key (if any) is appended to the two keys, and stored.

The join on maps is lifted to tables, as follows:

$$
\begin{aligned}
&\mathit{join} :: (\\
&\quad \mathit{HF}\ (a,b)\ k\ v, \mathit{HF}\ (a',b')\ k'\ v', \mathit{HF}\ (a, bab')\ k\ vkv',\\
&\quad \mathit{HA}\ a'\ b'\ ab', \mathit{HA}\ b\ ab'\ bab', \mathit{HA}\ v\ kv'\ vkv', \mathit{HA}\ k'\ v'\ kv',\\
&\quad \mathit{Row}\ (a,b)\ k\ v\ (\mathit{Record}\ r), \mathit{LookupMany}\ a'\ r'\ k'\\
&) \Rightarrow \mathit{Table}\ (a,b)\ k\ v \to \mathit{Table}\ (a',b')\ k'\ v' \to (\mathit{Record}\ r \to r')\\
&\quad \to \mathit{Table}\ (a, bab')\ k\ vkv'\\
&\mathit{join}\ (\mathit{Table}\ h@(a,b)\ m)\ (\mathit{Table}\ (a',b')\ m')\ on = \mathit{Table}\ h''\ m''\\
&\quad \textbf{where}\\
&\qquad h'' = (a, \mathit{hAppend}\ b\ (\mathit{hAppend}\ a'\ b'))\\
&\qquad m'' = \mathit{joinM}\ (\lambda k\ v \to \mathit{lookupMany}\ a'\ (on\ \$\ \mathit{row}\ h\ k\ v))\ m\ m'
\end{aligned}
$$

The header of the resulting table is constructed by appending the appropriate header components of the argument tables. The *on* function operates on a row from the first table, and produces a record that assigns a value to each key in the second table. Typically, these assignments map a foreign key to a primary one, as follows:

$$\mathit{myOn} = \lambda r \to ((\mathit{atPK} \mathbin{.=.} (r \mathbin{.!.} \mathit{atFK})) \mathbin{.*.} \mathit{HNil}))$$

In case of compound keys, the record would hold multiple assignments. The type of *myOn* is inferred automatically, and checked for validity when used to join two particular tables. In particular, the ON clause is checked to assign a value to every key attribute of the second table, and to refer only to keys or values from the second table. Thus, our model of join is typed more precisely than the standard SQL join, since join conditions are not allowed to underspecify the row from the second table.

The following example shows how joins are used in combination with selects:

$$
\begin{aligned}
&\mathit{seniorAmericans}\\
&= \mathit{select}\ \mathit{False}\ (\mathit{atName} \mathbin{.*.} \mathit{atCountry} \mathbin{.*.} \mathit{HNil})\\
&\quad ((\mathit{myTable}\ `\mathit{join}`\ \mathit{yourTable}\\
&\qquad (\lambda r \to ((\mathit{atCity}' \mathbin{.=.} (r \mathbin{.!.} \mathit{atCity})) \mathbin{.*.} \mathit{HNil})))\\
&\qquad \mathbin{.*.} \mathit{HNil})\\
&\quad (\lambda r \to (r \mathbin{.!.} \mathit{atAge}) > 65 \wedge (r \mathbin{.!.} \mathit{atCountry}) \equiv \texttt{"USA"})
\end{aligned}
$$

Recall that *atCity′* is the sole key of *yourTable*. The type-checker will verify that this is indeed the case. The last line represents a where clause that accesses columns from both tables.

Note that our *join* is used as a binary operator on tables. This means that several joins can be performed by nesting *join* calls. In fact, the join and cartesian product operators can be mixed to create join expressions beyond SQL's syntax limits. This is an immediate consequence from working in a higher-order functional language.

4.7 The GROUP BY clause and aggregation functions

When the SELECT statement is provided together with a GROUP BY clause, it can include aggregation functions such as COUNT and SUM in its column specification, and it may have a HAVING clause, which is similar to the WHERE clause but is applied *after* grouping.

On the level of maps, a general grouping function can be defined:

```
<group by clause> ::= GROUP BY  <grouping column reference list>
<grouping column reference list> ::=
        <grouping column reference>
        [ { <comma> <grouping column reference> }... ]

<grouping column reference> ::= <column reference> [ <collate clause> ]
```

Table 4.7: Syntax for the GROUP BY clause (according SQL-92 BNF Grammar)

```
groupByM :: (
  Ord k, Ord k'
) => (k -> v -> k') -> (Map k v -> a) -> Map k v -> Map k' a
groupByM g f m = Map.map f $ foldWithKey grp Map.empty m
  where
    grp k v = insertWith Map.union (g k v) (Map.singleton k v)
Map.map :: (a -> b) -> Map k a -> Map k b
```

Parameter g serves to compute from a map entry a new key under which to group that entry. Parameter f is used to map each group to a single value.

To represent aggregation functions, we define a data type AF:

```
data AF r a b = AF ([a] -> b) (r -> a)
data AVG; atAVG = ⊥ :: Attribute t n -> Attribute Float (AVG, n)
data COUNT; atCOUNT = ⊥ :: n -> Attribute Int (COUNT, n)
myAFs = (atAVG atAge) .=. AF avg (.!.atAge) .*.
        (atCOUNT ()) .=. AF length (const ()) .*. HNil
```

Each aggregation function is a map-reduce pair of functions, where the map function of type $r \to a$ computes a value from each row, and the reduce function of type $[a] \to b$ reduces a list of such values to a single one. As illustrated by $myAFs$, aggregation functions are stored in an attribute-labeled record to be passed as argument to a select with grouping clause.

These ingredients are sufficient to add grouping and aggregation behavior to the selection statements.

We present the resulting function $selectG$, which encodes a select with grouping and having clause.

```
selectG :: (
    ...) => Bool -> af -> ts -> (r -> Bool) -> b -> g -> (r' -> Bool) -> ResultList a xy
selectG distinct f ts p b g q = ResultList h' $ sort $ uniq $ lst gs''
  where
    (l, a) = hUnzip f
    Table h m = products ts
    fm = Map.filterWithKey (λk v -> p $ row h k v) m
    gs = groupByM (λk v -> lookupMany g $ row h k v) flt fm
    flt = Map.foldWithKey (λk v m -> (k, v) : m) []
    gs' = Map.map (λkvs -> aggrs h kvs a) gs
```

$h' = hAppend\ g\ l$
$gs'' = Map.filterWithKey\ (\lambda k\ v \rightarrow q\ \$\ row\ (g,l)\ k\ v)\ gs'$
$uniq =$ **if** $distinct$ **then** $rmDbls$ **else** id
$sort =$ **if** $isEmpty\ b$ **then** id **else** $(qsort \circ cmp)\ b$
$cmp\ b\ v\ v' = lookupMany\ b\ (hZip\ h'\ v) > lookupMany\ b\ (hZip\ h'\ v')$
$lst = Map.foldWithKey\ (\lambda g\ kv\ xs \rightarrow (hAppend\ g\ kv) : xs)\ [\,]$

The arguments of this function have the following meaning: *distinct* represents the distinct flag; f, the list of aggregate functions; ts, the list of tables; p, the where clause; b, the order by clause; g, the group by clause and q, the having clause.

As an example, recall the table *myTable*, defined in chapter 3.

$myHeader = (atID\ .*.\ HNil, atName\ .*.\ atAge\ .*.\ atCity\ .*.\ HNil)$
$myTable = Table\ myHeader\ \$$
$insert\ (12\ .*.\ HNil)\ (\texttt{"Ralf"}\ .*.\ 23\ .*.\ \texttt{"Seattle"}\ .*.\ HNil)\ \$$
$insert\ (67\ .*.\ HNil)\ (\texttt{"Oleg"}\ .*.\ 17\ .*.\ \texttt{"Seattle"}\ .*.\ HNil)\ \$$
$insert\ (50\ .*.\ HNil)\ (\texttt{"Dorothy"}\ .*.\ 42\ .*.\ \texttt{"Oz"}\ .*.\ HNil)\ \$$
$Map.empty$

If we want to count the number of individuals of each city – N – and compute their average age, presenting the results in descending order of N and without having and where clause, we can perform a grouping on this table by the attribute city and then apply the aggregate functions *myAFs* defined above.

```
> selectG False myAFs (myTable.*.HNil) (const True)
            ((atCOUNT ()) .*. HNil) (atCity .*. HNil) (const True)
ResultList (HCons CITY (HCons AVGAGE (HCons NUMBER HNil)))
   [HCons "Seattle" (HCons 18.0 (HCons 2 HNil)),HCons "Oz" (HCons 42.0 (HCons 1 HNil))]
```

4.8 Database operations

Last but not least, we can lift the operations we have defined on tables to work on entire relational databases. These operations then refer by name to the tables they work on. For example, the following models the SELECT INTO statement that performs a select, and stores the result list into a named table:

$selectInto\ rdb\ d\ a\ tns\ w\ o\ tn\ a' = modifyAtLabel\ tn\ f\ rdb$
where
$ts = fst\ \$\ hUnzip\ \$\ lookupMany\ tns\ rdb$
$f\ (t,fk) = (insertResultList\ a'\ (select\ d\ a\ ts\ w\ o)\ t,fk)$

Note that the argument tables are fetched from the database before they are supplied to the *select* function. The *modifyAtLabel* function is an utility on records that applies a given function on the value identified by a given label. We can use this function to insert in table *yourTable* all the cities that appear in table *myTable*. The country associated with each city will be the default one.

```
> selectInto myrdb' True (atCity .*. HNil) (people.*.HNil)
                (const True) HNil cities (atCity'.*.HNil)
Record{
```

```
 cities=Table (HCons CITY' HNil,HCons COUNTRY HNil) {
   HCons "Braga" HNil:=HCons "Portugal" HNil,
   HCons "Oz" HNil:=HCons "Afghanistan" HNil,
   HCons "Seattle" HNil:=HCons "Afghanistan" HNil
 }
...
```

The source code distribution of this report contains liftings of the other table operations as well. Furthermore, database-level implementations are provided of data definition statements, such as CREATE, ALTER, and DROP TABLE.

In this chapter we have presented a reconstruction of several SQL statements. Moreover, we have shown how to provide more precise types to SQL operations such as insertion and join, and we have lifted table operations to the database level. In the next chapter we will go further and we will show how database design information, in particular functional dependencies, can be captured at the type level. We will also present definitions for several normal forms, as well as normalization and denormalization operations.

Chapter 5

Functional dependencies and normal forms

In the preceding chapters we have shown how information about the types, labels, and key-status of table columns can be captured at the type-level. As a consequence, static type-checks guarantee the safety of our tables and table operations with respect to these kinds of meta-data. In this chapter, we will go a step further. We will show how an important piece of database design information, *viz* functional dependencies, can be captured and validated at the type level. Moreover, we will present several normal forms and (de)normalization operations.

5.1 Functional dependencies

The only way to determine functional dependencies that hold in a relation header H is to carefully analyze the semantic meaning of each attribute. They cannot be proved but we might expect them to be enforced in a database.

Definition 1 *Given a header H for a table and X, Y subsets of H, there is a functional dependency (FD) between X and Y ($X \rightarrow Y$) iff X fully determines Y (or Y is functionally dependent on X).*

Functional dependencies play an important role in database design. Database normalization and de-normalization, for instance, are driven by functional dependencies. FD theory is the kernel of the classical relational database design theory developed by Codd [12]. It has been thoroughly studied [5, 21, 28], and is a mandatory part of standard database literature [27, 38, 14].

A type-level representation of functional dependencies is given in Section 5.1.1. We proceed in Section 5.1.2 with type-level predicates that capture the notions of *key* and *superkey* with respect to functional dependencies. These predicates are building blocks for more complex predicates that test whether a given set of functional dependencies adheres to particular normal forms. In Section 5.1.3 we present an algorithm to check functional dependencies in a table and in Section 5.2 type-level predicates are defined for several normal forms. Finally, in Section 5.3 we explore how functional dependency information associated to particular

tables can carry over from the argument tables to the result tables of table operations. In particular, we will show that the functional dependencies of the result tables of projections and joins can be computed at the type-level from the functional dependencies on their arguments.

Note that we are not concerned with mining of functional dependencies from actual data. We assume the presence of functional dependency information, whether declared by a database designer or stemming from another source. Our interest lies in employing the type system to enforce functional dependencies, and to calculate with them.

5.1.1 Representation

To represent functional dependencies, we transpose Definition 1 into the following datatype and constraints:

data $FunDep\ x\ y \Rightarrow FD\ x\ y = FD\ x\ y$

class $FunDep\ x\ y$
instance $(AttrList\ x, AttrList\ y) \Rightarrow FunDep\ x\ y$

class $AttrList\ ats$
instance $AttrList\ HNil$
instance $AttrList\ l \Rightarrow AttrList\ (HCons\ (Attribute\ v\ n)\ l)$

Thus, a functional dependency basically holds two lists of attributes, of which one represents the *antecedent* and the other the *consequent* of the dependency.

A list of functional dependencies for a particular table should only mention attributes from that table. This well-formedness condition can be expressed by the following type-level predicates:

class $FDListFor\ fds\ h$
instance (
  $FDList\ fds, AttrListFor\ fds\ ats,$
  $HAppend\ a\ b\ ab, ContainAll\ ab\ ats$
) $\Rightarrow FDListFor\ fds\ (a, b)$

class $FDList\ at$
instance $FDList\ HNil$
instance $(FunDep\ lhs\ rhs, FDList\ fds)$
  $\Rightarrow FDList\ (HCons\ (FD\ lhs\ rhs)\ fds)$

class $AttrListFor\ fds\ ats \mid fds \rightarrow ats$
instance $AttrListFor\ HNil\ HNil$
instance (
  $Union\ x\ y\ z, FunDep\ x\ y,$
  $AttrListFor\ fds\ ats', Union\ z\ ats'\ ats$
) $\Rightarrow AttrListFor\ (HCons\ (FD\ x\ y)\ fds)\ ats$

Here the *FDList* predicate constrains a heterogeneous list to contain functional dependencies only, and the type level function *AttrListFor* computes the attributes used in a given list of FDs.

5.1.2 Keys

In section 4, we distinguished key attributes from non-key attributes of a table. There is an analogous concept for relations with associated functional dependencies F.

Definition 2 *Let H be a header for a relation and F the set of associated functional dependencies. Every set of attributes $X \subseteq H$ is a key iff $X \rightarrow H$ can be deduced from F and X is minimal. X being minimal means that from no proper subset Y of X we can deduce $Y \rightarrow H$ from F.*

An essential ingredient of this definition is the set of all functional dependencies that can be derived from an initial set. This is called the closure of the FD set. This closure is expensive to compute. However, we can tell whether a given dependency $X \rightarrow Y$ is in the FD closure by computing the set of attributes that can be reached from X via dependencies in F. This second closure is defined as follows.

Definition 3 *Given a set of attributes X, we define the closure X^+ of X (with respect to a set of FDs F) as the set of attributes A that can be determined by X (i.e., $X \rightarrow A$ can be deduced from F).*

The algorithm used to implement the computation of such a closure is described in [38, p.338]. We implemented it on the type level with a constraint named *Closure* (see below).

Another ingredient in the definition of keys is the minimality of a possible key. We define a predicate that expresses this.

```
class Minimal x h fds b | x h fds → b
instance (ProperSubsets x xs, IsNotInFDClosure xs h fds b)
  ⇒ Minimal x h fds b
```

Thus, we compute the proper subsets of X and check (with *IsNotInFDClosure* – implementation not shown) that none of these sets Y is such that $Y \rightarrow H$.

```
class Closure u x fd closure | u x fd → closure
instance
  (
    TypeEq u x b,
    Closure' b u x fd cl
  )
  ⇒ Closure u x fd cl
class Closure' b u x fd closure | b u x fd → closure
instance Closure' HTrue u x fd x
instance
  (GetRHS x fd rhs,
    Union x rhs x',
    Closure x x' fd closure
  ) ⇒ Closure' HFalse u x fd closure
```

With all ingredients defined, we proceed to the specification of the constraint that tests whether a given attribute is a key:

```
class IsKey x h fds b | x h fds → b
instance (
  Closure h x fds cl, Minimal x h fds b'',
  ContainedEq h cl b', HAnd b' b'' b
) ⇒ IsKey x h fds b
```

There may be more than one key for a relation. So, when we use the term *candidate key* we are referring to any minimal set of attributes that fully determine all attributes.

For the definition of normal forms, we additionally need the concept of a *super key*, which is defined as follows:

Definition 4 *$X \subseteq H$, is a superkey for a relation with header H, if X is a superset of a key (i.e., $\exists_{X'}\ X'$ is a key $\wedge\ X' \subseteq X$).*

This concept can be expressed as follows.

```
class IsSuperKey s all fds b | s all fds → b
instance (
  PowerSet s ss, FilterEmptySet ss ss', MapIsKey ss' all fds b
) ⇒ IsSuperKey s all fds b
```

The auxiliary function *MapIsKey* checks if at least one of the subsets of X is a key.

```
class MapIsKey ss all fds b | ss all fds → b
instance MapIsKey HNil all fds HFalse
instance (
  IsKey x all fds b',
  MapIsKey xs all fds b'',
  HOr b' b'' b
  ) ⇒ MapIsKey (HCons x xs) all fds b
```

Note that the power set computation involved here implies considerable computational complexity! We will comment on the need for optimisation in our concluding remarks.

5.1.3 Checking functional dependencies

A functional dependency $X \rightarrow Y$ is satisfied in a relation iff the set of attributes X uniquely determines Y, *i.e.*, if we group the table by X, after projecting by the attributes in $X \cup Y$, the corresponding set to each group X should be singleton. Figure 5.1 schematically explains how we check if table T verifies FD $X \rightarrow Y$.

```
class CheckFD fd t where
  checkFD :: fd → t → Bool
instance (HeaderFor h k v, ..., FunDep x y) ⇒ CheckFD (FD x y) (Table h k v) where
  checkFD (FD x y) t = Map.fold worker True (groupBy x $ project xy t)
    where
      worker set b = b ∧ (Set.size set ≡ 1)
      xy = hAppend x y
```

The lifting of this operation to a list of functional dependencies is simply a map of the previous operation over the referred list combined with a logical $\wedge$.

FD $X \to Y$
Input Table T

x_1	...	x_m	y_1	...	y_n	...	z_k
x'_1	...	x'_m	y'_1	...	y'_n	...	z'_k
x_1	...	x_m	y''_1	...	y''_n	...	z''_k
x''_1	...	x''_m	y''_1	...	y''_n	...	z''_k

$\downarrow \pi_{X \cup Y} T$

x_1	...	x_m	y_1	...	y_n
x'_1	...	x'_m	y'_1	...	y'_n
x_1	...	x_m	y''_1	...	y''_n
x''_1	...	x''_m	y''_1	...	y''_n

$\downarrow$ Group T

$$\begin{array}{lcl} x_1 \ldots x_m & \mapsto & \{y_1 \ldots y_n, y''_1 \ldots y''_n\} \\ x'_1 \ldots x'_m & \mapsto & \{y'_1 \ldots y'_n\} \\ x''_1 \ldots x''_m & \mapsto & \{y''_1, \ldots, y''_n\} \end{array}$$

$\downarrow$ Check $\star$

False

$\star$: The range of the map has only singleton sets

Figure 5.1: Process of checking a functional dependency in a table

```
class CheckFDs fds t where
  checkFDs :: fds → t → Bool
instance CheckFDs HNil t where
  checkFDs HNil t = True
instance (CheckFD (FD x y) (Table h k v),
  CheckFDs fds (Table h k v)
  ) ⇒ CheckFDs (HCons (FD x y) fds) (Table h k v) where
  checkFDs (HCons fd fds) t = (checkFD fd t) ∧ checkFDs fds t
```

5.2 Normal Forms

We will now define normal forms for relation schemas with dependencies defined.

First NF

A relation schema R is in first normal form if every attribute in R is atomic, *i.e.*, the domain of each attribute cannot be lists or sets of values or composite values.

class $Is1stNF\ l\ b \mid l \rightarrow b$
instance $Is1stNF\ HNil\ HTrue$
instance $(Atomic\ at\ b, Is1stNF\ ats\ b', HAnd\ b\ b'\ b'') \Rightarrow Is1stNF\ (HCons\ at\ ats)\ b''$

The definition of atomicity for an attribute is as follows.

class $Atomic\ at\ b \mid at \rightarrow b$
instance $Atomic\ t\ b \Rightarrow Atomic\ (Attribute\ t\ nr)\ b$
instance $(IsList\ t\ b1,$
$IsSet\ t\ b2,$
$IsMap\ t\ b3,$
$HOr\ b1\ b2\ b'',$
$HOr\ b3\ b''\ b',$
$Not\ b'\ b$
$) \Rightarrow Atomic\ t\ b$

The predicates $IsList$, $IsSet$ and $IsMap$ check, respectively, if the attribute domain is of type $[a]$, $Set\ a$ or $Map\ k\ a$. The predicate $IsList$ has the particularity of excluding text (String) that, in Haskell, is represented as $[Char]$.

Second NF

Before defining second normal form we need to define the notions of prime attribute and full dependency between sets of attributes [27].

Definition 5 *Given an header H with a set of FDs F and an attribute A in H, A is prime with respect to F if A is member of any key in H.*

Our encoding of this definition is as follows.

class $IsPrime\ at\ all\ fds\ b \mid at\ all\ fds \rightarrow b$
instance $(Keys\ all\ fds\ lk, MemberOfAnyKey\ at\ lk\ b)$
$\Rightarrow IsPrime\ at\ all\ fds\ b$

Definition 6 *Given a set of functional dependencies F and $X \rightarrow Y \in F^+$, we say that Y is partially dependent upon X if $X \rightarrow Y$ is not left-reduced. That is, there is a proper subset X' of X such that $X' \rightarrow Y$ is in F^+. Otherwise, Y is said to be fully dependent on X.*

Our encoding of *full dependency* is as follows.

class $FullyDep\ fds\ x\ y\ b \mid fds\ x\ y \rightarrow b$
instance $LeftReduced\ fds\ x\ y\ b \Rightarrow FullyDep\ fds\ x\ y\ b$
class $LeftReduced\ fds\ x\ y\ b \mid fds\ x\ y \rightarrow b$
instance $(FunDep\ x\ y,$

$$
\begin{array}{l}
\quad \mathit{ProperSubsets}\ x\ xs, \\
\quad \mathit{LeftReduced}'\ xs\ y\ fds\ b \\
\quad) \Rightarrow \mathit{LeftReduced}\ fds\ x\ y\ b \\
\textbf{class}\ \mathit{LeftReduced}'\ xs\ y\ fds\ b \mid xs\ y\ fds \rightarrow b \\
\textbf{instance}\ \mathit{LeftReduced}'\ \mathit{HNil}\ y\ fds\ \mathit{HTrue} \\
\textbf{instance}\ (\mathit{FunDep}\ x\ y, \\
\quad \mathit{IsInFDClosure}\ y\ x\ fds\ b1, \\
\quad \mathit{Not}\ b1\ b', \\
\quad \mathit{LeftReduced}'\ xs\ y\ fds\ b'', \\
\quad \mathit{HAnd}\ b'\ b''\ b \\
\quad) \Rightarrow \mathit{LeftReduced}'\ (\mathit{HCons}\ x\ xs)\ y\ fds\ b
\end{array}
$$

A relation scheme R is in second NF if (it is in first NF and) every non-prime attribute is fully dependent on every key of R.

More intuitively, this definition means that a scheme (with respect to a set of FDs F) in second NF does not have partial dependencies.

$$
\begin{array}{l}
\textbf{class}\ \mathit{Is2ndNF}\ all\ fds\ b \mid all\ fds \rightarrow b \\
\textbf{instance}\ (\mathit{Is1stNF}\ all\ \mathit{HTrue}, \\
\quad \mathit{NonPrimes}\ all\ lk\ nonprimes, \\
\quad \mathit{Keys}\ all\ fds\ lk, \\
\quad \mathit{MapFullyDep}\ fds\ lk\ nonprimes\ b \\
\quad) \\
\qquad \Rightarrow \mathit{Is2ndNF}\ all\ fds\ b
\end{array}
$$

The class *MapFullyDep* checks if all nonprime attributes are fully dependent upon every key. The class *NonPrimes* computes the nonprime attributes based on the list of keys. We could have defined such function using the predicate *IsPrime*, but this would be very inefficient since the list of keys would be re-calculated several times (which is computationally complex). Since the list of keys is available, we simply perform a set difference, returning a list with every attribute in the header that is not member of any key.

$$
\begin{array}{l}
\textbf{class}\ \mathit{NonPrimes}\ all\ lk\ nonprimes \mid all\ lk \rightarrow nonprimes \\
\textbf{instance}\ (\mathit{Unions}\ lk\ lks, \mathit{Difference}\ all\ lks\ np) \Rightarrow \mathit{NonPrimes}\ all\ lk\ np
\end{array}
$$

We will now discuss the most significant normal forms – third normal form (NF) and Boyce-Codd NF.

These normal forms require the scheme to be in first NF. For economy of presentation, we will present the definitions omitting this verification (in the encoding this verification is performed).

We will also assume that FDs associated with the schemas are represented with a single attribute in the consequent. Notice that all sets of FDs can be reduced to a set in this form [27].

Boyce Codd NF

A table with header H is in Boyce Codd NF with respect to a set of FDs if whenever $X \rightarrow A$ holds and A is not in X then X is a superkey for H.

This means that in Boyce-Codd normal form, the only non-trivial dependencies are those in which a key determines one or more other attributes [38]. More intuitively, no attribute in H is transitively dependent upon any key of H.

Let us start by defining the constraint for a single FD.

```
class BoyceCoddNFAtomic check h x fds b | check h x fds → b
instance BoyceCoddNFAtomic HFalse h x fds HTrue
instance IsSuperKey x h fds b
  ⇒ BoyceCoddNFAtomic HTrue h x fds b
```

The type-level boolean *check* is included because we want to check just if X is a superkey when Y is not in X. Now, we can extrapolate this definition for a set of FDs :

```
class BoyceCoddNF h fds b | h fds → b
instance BoyceCoddNF h HNil HTrue
instance BoyceCoddNF' h (HCons e l) (HCons e l) b
  ⇒ BoyceCoddNF h (HCons e l) b
class BoyceCoddNF' h fds allfds b | h fds allfds → b
instance BoyceCoddNF' h HNil fds HTrue
instance (
  HMember y x bb, Not bb bYnotinX,
  BoyceCoddNFAtomic bYnotinX h x fds b',
  BoyceCoddNF' h fds' fds b'', HAnd b' b'' b
  ) ⇒ BoyceCoddNF' h (HCons (x, HCons y HNil) fds') fds b
```

To illustrate the verification of this NF let us consider the following example [38].

Example 1 *Consider a relation header H containing three attributes: City (C), State (S) and Zip (Z) with non trivial functional dependencies $F = \{CS \rightarrow Z, Z \rightarrow C\}$, meaning that the zip code is functionally determined by city and state; the zip code fully determines the city.*

We can easily see that CS and SZ are the keys for this relation header. This relation is not in Boyce-Codd NF because $Z \rightarrow C$ holds in H and Z is not a superkey.

Let us check this in our model. Having the attributes *city*, *state* and *zip* defined we can define the set of fds as follows:

```
h = city .*. state .*. zip .*. HNil
fds = (FD (city .*. state .*. HNil) (zip .*. HNil)) .*.
      (FD (zip .*. HNil) (city .*. HNil)) .*. HNil
```

Now, we can check whether header *h* is in in Boyce-Codd NF with respect to *fds*.

```
> :t boyceCoddNF h fds
boyceCoddNF h fds :: HFalse
```

More examples of verification of this normal form can be found in [38].

Third NF

A table with header H is in third NF with respect to a set of FDs if whenever $X \rightarrow A$ holds and A is not in X then X is a superkey for H or A is a prime attribute. Notice that this definition

is very similar to Boyce-Codd NF except for the clause “or A is prime”. This NF can therefore be seen as a weakening of Boyce-Codd NF. Intuitively, in third NF we are just demanding that no nonprime attributes are transitively dependent upon a key on H.

As in the previous NF, we will start by defining a constraint for a single FD.

```
class Is3rdNFAtomic check h x y fds b | check h x y fds → b
instance Is3rdNFAtomic HFalse h x y fds HTrue
instance (IsSuperKey x h fds sk, IsPrime y h fds pr, HOr sk pr b)
 ⇒ Is3rdNFAtomic HTrue h x y fds b
```

This definition is as follows for a set of FDs :

```
class Is3rdNF h fds b | h fds → b
instance Is3rdNF h HNil HTrue
instance (Is3rdNF' h (HCons e l) (HCons e l) b)
 ⇒ Is3rdNF h (HCons e l) b
class Is3rdNF' h fds allfds b | h fds allfds → b
instance Is3rdNF' h HNil allfds HTrue
instance (
   HMember y x bb, Not bb bYnotinX,
   Is3rdNFAtomic bYnotinX h x y fds b',
   Is3rdNF' h fds' fds b'', HAnd b' b'' b
   ) ⇒ Is3rdNF' h (HCons (x, HCons y HNil) fds') fds b
```

Notice that any relation schema that is in third normal form (with respect to a set of functional dependencies FD) is also in second normal form with respect to FD. The main idea behind this result is that a partial dependency implies a transitive dependency – a schema that violates 2NF also violates 3NF.

Considering the example 1, we can check that the relation is in third NF, since all attributes are prime (every one is member of some key).

Using these normal form definitions in the form of type constraints, normal form checking can be carried out by the type checker.

5.3 Transport through operations

When we perform an operation over one or more tables that have associated FD information, we can compute new FDs associated to the resulting table. We will consider *project* and *join* as examples. But first we define a representation for tables with associated FD information:

```
data TableWithFD fds h k v
 ⇒ Table' h k v fds = Table' h (Map k v) fds
class (HeaderFor h k v, FDListFor fds h) ⇒ TableWithFD fds h k v
```

Thus, we have an extra component *fds* which is constrained to be a valid set of FDs for the given table.

Project

When we project a table with set F of associated FDs according to a list of attributes B, for every $X \rightarrow Y \in F$ we can do the following reasoning:

- If there is an attribute A , such that $A \in X$ and $A \notin B$ then $X \rightarrow Y$ will not hold (in general) in the new set of FDs;
- Otherwise, we compute $Y' = Y \cap B$ and we have $X \rightarrow Y'$ holding in the new set of FDs.

This simple algorithm is encoded as follows.

```
class ProjectFD b fds fds' | b fds → fds' where
  ProjectB :: b → fds → fds'
instance ProjectFD b HNil HNil where
  projectFD _ _ = hNil
instance (
  FunDep x y, Difference x b x',
  HEq x' HNil bl, ProjectFD' bl b (HCons (FD x y) fds) fds'
) ⇒ ProjectFD b (HCons (FD x y) fds) fds'
where
  projectFD b (HCons (FD x y) fds)
    = projectFD' bl b (HCons (FD x y) fds)
      where x' = difference x b; bl = hEq x' HNil
```

The constraint $HEq\ x'\ HNil\ bl$ is used to control if X only contains attributes that are in B, which is equivalent to check $X - B = \{\}$. The resulting boolean value is passed as argument to an auxiliary function that either will eliminate the FD from the new set or will compute the new FD.

```
class ProjectFD' bl b fds fds' | bl b fds → fds' where
  projectFD' :: bl → b → fds → fds'
instance (FunDep x y, ProjectFD b fds fds')
 ⇒ ProjectFD' HFalse b (HCons (FD x y) fds) fds'
where projectFD' _ b (HCons (FD x y) fds) = projectFD b fds
instance (FunDep x y, Intersect b y y', ProjectFD b fds fds')
 ⇒ ProjectFD' HTrue b (HCons (FD x y) fds) (HCons (FD x y') fds')
where
  projectFD' _ b (HCons (FD x y) fds)
    = HCons (FD x (intersect b y)) (projectFD b fds)
```

This calculation can be linked to the restricted projection function described in Section 4 in the following function.

```
projectValues' :: (
  TableWithFD fds (a, b) k v, TableWithFD fds' (a, b') k v',
  LineUp' b' v v' r, ProjectFD b' fds fds'
) ⇒ b' → Table' (a, b) k v fds → Table' (a, b') k v' fds'
projectValues' b' (Table' (a, b) m fds)
 = Table' (a, b') m' (projectFD b' fds)
  where (Table _ m') = projectValues b' (Table (a, b) m)
```

Thus, this function provides a restricted projection operation that preserves not only keys, but also transports relevant functional dependencies to the result table.

Example 2 *Consider a relation that has information about student's marks for courses with the following header and functional dependencies:*

$$H = \{ \quad numb, name, mark, ccode, cname\}$$

$$\begin{aligned} F = \{ \qquad numb \rightarrow name, & \quad (5.1)\\ ccode \rightarrow cname, & \quad (5.2)\\ numb, ccode \rightarrow mark\} & \quad (5.3) \end{aligned}$$

Let us suppose we want to project a table with header H according to the set of attributes $B = \{mark, ccode, cname\}$. What FDs will hold in the resulting table? (5.1) and (5.3) cannot hold because $numb$ is not present in the resulting table and, therefore, $name$ is not self determined and $code$ alone cannot determine $mark$; 5.2 only involves attributes of B, so it will be the only FD for the new table.

Join

When we join two tables with F and F' sets of FDs associated, then in the new table all the FDs $f \in F \cup F'$ will hold. Therefore, this calculation can be simply linked to the function `sqlInnerJoin` described in 4 in the following function.

```
join' :: (Union fds fds' fds'', ...)
 ⇒ Table' (a, b) k v fds → Table' (a', b') k v fds'
   → (Record row → r) → Table' (a, bab') k vkv' fds''
join' (Table' h m fds) (Table' h' m' fds') r
   = Table' h'' m'' fds''
     where
       Table h'' m'' = join (Table h m) (Table h' m') r
       fds'' = union fds fds'
```

Here we have elided all but the interesting constraint, which performs the union of functional dependency sets.

Example 3 *Let us consider the relation from example 2 and suppose we have a new relation containing the information about students' provenience (city and country names), with header $H' = \{numb, city, country\}$ and FDs $F' = \{City \rightarrow Country\}$. The obvious way to bring together this information is joining the two tables (they have a common attribute $numb$). The resulting table will have header $H'' = \{numb, ccode, cname, mark, numb', city, country\}$. All the FDs from F and F' will hold.*

5.4 Normalization and denormalization

We have shown how strong types, capturing meta-data such as table headers and foreign keys, can be assigned to SQL databases and operations on them. Moreover, we have shown

that these types can be enriched with additional meta-information not explicitly present in SQL, namely functional dependencies. In this section, we will briefly discuss some scenarios beyond traditional database programming with SQL, namely normalization and denormalization, in which strong types pay off.

Normalization and denormalization are database transformation operations that bring a database or some of its tables into normal form, or *vice versa*. Such operations can be defined type-safely with the machinery introduced above.

We have defined a denormalization operation *compose* that turns tables from third into second normal form. In fact, the *compose* operation is a restricted variant of *join*, lifted to the level of a database. Rather than using a 'free-style' ON clause, which assigns values computed from the first table to primary key attributes of the second, *compose* explicitly exploits a foreign key relationship, provided as argument.

$$\begin{aligned}
&compose\ db\ fk\ tn1\ tn2\ newt = (rdb''', union\ (fd\ .*\ .fd'\ .*\ .\ HNil)\ (union\ r'\ r))\\
&\quad \textbf{where}\\
&\qquad (_, fks1) = db\ .!.\ tn1\\
&\qquad (t2@(Table\ (a,b)\ _), fks2) = db\ .!.\ tn2\\
&\qquad (_, pk) = fks1\ .!.\ fk\\
&\qquad on = (\lambda r \rightarrow Record\ \$\ hZip\ pk\ \$\ lookupMany\ fk\ r)\\
&\qquad fks1'' = treatSelfRefs\ tn1\ newt\ fks1'\\
&\qquad (fks2', r') = filterFks\ tn2\ fks2\\
&\qquad fks2'' = treatSelfRefs\ tn1\ newt\ fks2'\\
&\qquad rdb' = hDeleteAtLabel\ tn2\ db\\
&\qquad (rdb'', r) = mapfilterFks\ tn2\ rdb'\\
&\qquad (t1, fks1') = rdb''\ .!.\ tn1\\
&\qquad fd = FD\ a\ b\\
&\qquad fd' = FD\ fk\ pk\\
&\qquad t3 = join\ t1\ t2\ on\\
&\qquad rdb''' = (newt\ .=\ .\ (t3, hLeftUnion\ fks1''\ fks2''))\ .*\ .\ (hDeleteAtLabel\ tn1\ rdb'')
\end{aligned}$$

The functional dependencies encoded in the original database as the given foreign key relation and as meta-data of the second table are lost in the new database. Instead, our *compose* explicitly returns these 'lost' functional dependencies as result, paired with the new database.

Remark that *compose* performs a full treatment of self-references and references (using, respectively, *treatSelfRefs* and *filterFks*) among both tables involved in the join, guaranteeing that only foreign keys that are still holding are kept in the database and that all the others are returned to the user as functional dependencies.

This operation has a limitation. Some information from the second table involved in the *join* can be lost, namely the tuples whose key is not present in the first table (this will be shown in an example of chapter 6). To overcome this problem we have defined a new version of *compose* that instead of deleting both original tables from the database, keeps the second one, filtering the tuples that are redundant.

$$\begin{aligned}
&compose\ db\ fk\ tn1\ tn2\ newt = (rdb''', union\ (fd\ .*\ .fd'\ .*\ .\ HNil)\ r)\\
&\quad \textbf{where}\\
&\qquad \ldots\\
&\qquad t2' = Table\ (a', b')\ m'\\
&\qquad a' = rename\ a\ newt
\end{aligned}$$

$$
\begin{aligned}
&b' = \mathit{rename}\ b\ \mathit{newt}\\
&\mathit{fks2n} = \mathit{renamePartial}\ (\mathit{hAppend}\ a\ b)\ \mathit{fks2}\ \mathit{newt}\\
&\mathit{ResultList}\ \_\ \mathit{ks} = \mathit{select}\ \mathit{True}\ \mathit{fk}\ \mathit{t1}\ (\mathit{const}\ \mathit{True})\\
&m' = \mathit{mapDeleteMany}\ \mathit{ks}\ \mathit{m2}\\
&\mathit{rdb}''' = (\mathit{newt} \mathbin{.=.} (t3, \mathit{hLeftUnion}\ \mathit{fks1}''\ \mathit{fks2}')) \mathbin{.*.}\\
&\qquad (\mathit{tn2} \mathbin{.=.} t2' \mathbin{.=.} \mathit{fks2n}) \mathbin{.*.} (\mathit{hDeleteAtLabel}\ \mathit{tn1}\ \mathit{rdb}'')
\end{aligned}
$$

We omit the code repeated from the original *compose* definition. To maintain the well-formedness of the database we have to rename in the attributes in the header of the table and wherever these attributes were used in the database. For these operations we have defined, respectively, functions *rename* and *renamePartial* that prefix all the attributes with a new identifier. The function *renamePartial* receives the list of attributes that should be renamed in the foreign keys of the database.

Conversely, the normalization operation *decompose* can be used to bring tables into third normal form. It accepts a functional dependency as argument, which subsequently gets encoded in the database as meta-data of one of the new tables produced. Also, an appropriate foreign key declaration is introduced between the decomposed tables.

Thus, *compose* produces functional dependency information that can be used by *decompose* to revert to the original database schema. In fact, our explicit representation of functional dependencies allows us to define database transformation operations that manage meta-data in addition to performing the actual transformations.

An example of application of *compose* will be presented in chapter 6.

In this chapter we have shown how database design information can be captured at the type level. Furthermore, we have defined a new level of operations, which carry functional dependency information from argument to result tables, and we have presented normalization and denormalization operations. In the next chapter, we will present operations which will allow migration between our strongly typed model and spreadsheets.

Chapter 6

Spreadsheets

In previous chapters we have defined a strongly typed model of databases. Reasoning about and expressing data properties in such a model is simple. On the other hand, we should keep in mind that end-users are more familiar with programming environments such as spreadsheets, which have a very intuitive front-end.

For this purpose, in this chapter we will define a *bridge* between our model and Gnumeric[1] spreadsheets. Spreadsheets can be seen as a collection of tables (*i.e.*, a database). Much work, described in appendix 1.3, has been developed in order to identify tables in the sheets of a spreadsheet workbook. For the purpose of our work we will consider that each sheet in a workbook is a table whose first row is the header.

The syntax for a Gnumeric workbook is defined in the UMinho Haskell libraries[2] (module `Language.Gumeric.Syntax`). Roughly, a Gnumeric workbook is a list of sheets, which are lists of cells. Besides this data information, which is of most interest for us, the workbook also carries formating information. Note that the data in each cell is stored as $String$.

We will first define how to write a database from our model to Gnumeric format, where each table becomes a separate sheet in a workbook.

6.1 Migration to Gnumeric format

We will split the task of translating a database in our model to Gnumeric format into four small tasks, as described in figure 6.1.

Writing a single row

To convert a single row to a list of cells we just need to know in which column and row the first cell is located and then we can translate the list of values (our row) to a list of cells. The only restriction is that the values must be showable (so that we can convert them to $String$).

$$\textbf{class } \mathit{WriteRow}\ l\ l' \mid l \to l' \textbf{ where}$$
$$\quad \mathit{writeRow} :: \mathit{Int} \to \mathit{Int} \to l \to l'$$
$$\textbf{instance } \mathit{WriteRow}\ \mathit{HNil}\ [\mathit{Gmr'Cell}] \textbf{ where}$$

[1] Gnumeric (www.gnome.org/projects/gnumeric) is an open source spreadsheet package. It is possible to convert other spreadsheet formats, such as *e.g.* Excel, to this format.

[2] http://wiki.di.uminho.pt/wiki/bin/view/PURe/PUReSoftware

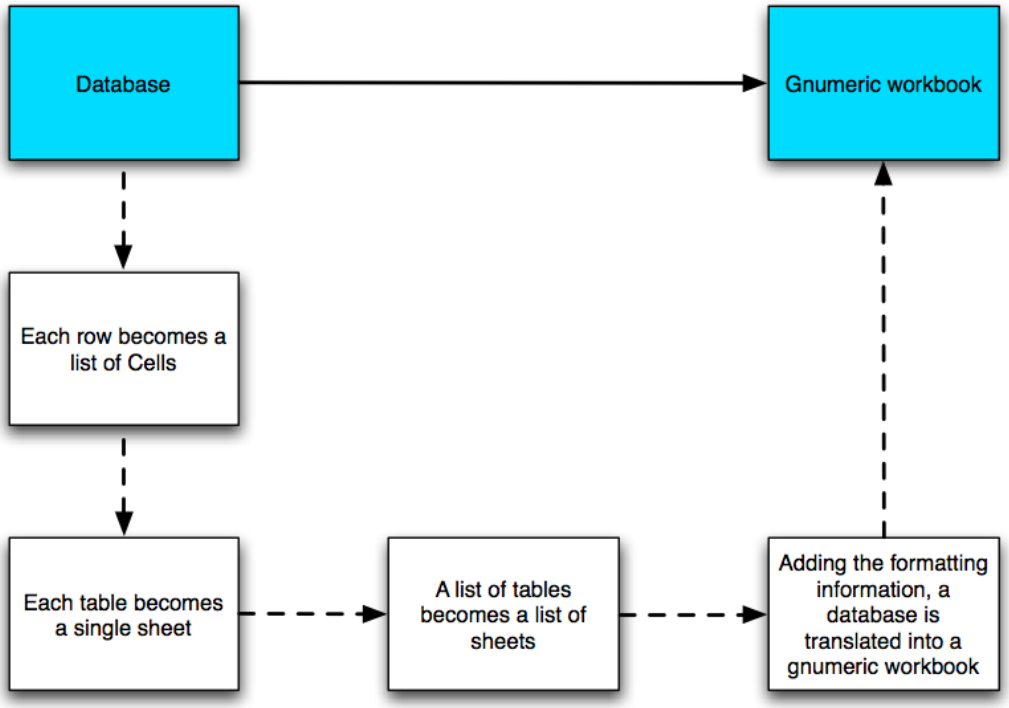


Figure 6.1: *Divide and conquer* for writing database into Gnumeric format

```
  writeRow _ _ _ = []
instance (WriteRow l [Gmr'Cell], ReadShow e) => WriteRow (HCons e l) [Gmr'Cell] where
  writeRow r c (HCons e l) = (cell : (writeRow r (c + 1) l))
    where cell = Gmr'Cell attrs (toString e)
          attrs = Gmr'Cell_Attrs (show c) (show r) ...
```

Here, class *ReadShow* is used to overcome the behavior of show and read functions in Haskell with *Strings*. The former prints extra quote marks when showing a *String* and the latter only parses a *String* if is limited by quote marks.

```
class ReadShow t where
  fromString :: String → t
  toString :: t → String
instance (Read t, Show t) => ReadShow t where
  fromString = read
  toString = show
instance ReadShow String where
  fromString = id
  toString = id
```

The *toString* and *fromString* functions behave as *show* and *read* when the type to show/parse is not of type String. Otherwise, they are the identity function.

For this to fit into our structured rows, divided in keys and non-keys, we have lifted the previous function to pairs.

```
instance (HAppend a b ab, WriteRow ab [Gmr'Cell]) ⇒ WriteRow (a, b) [Gmr'Cell] where
  writeRow i j (a, b) = writeRow i j (hAppend a b)
```

We still need an instance to treat nullable attributes, so that null values (represented by *Nothing*) are transformed in empty *Strings* and the *Just* constructor does not appear in the spreadsheet.

```
instance (
  WriteRow l [Gmr'Cell],
  ReadShow e
  ) ⇒ WriteRow (HCons (Maybe e) l) [Gmr'Cell] where
  writeRow r c (HCons e l) = (cell : (writeRow r (c + 1) l))
    where cell = case e of
                   Just x → Gmr'Cell attrs (toString x)
                   _ → Gmr'Cell attrs ""
          attrs = Gmr'Cell_Attrs (show c) (show r) Nothing ...
```

Writing a single table

To produce a workbook sheet from a table, we have first create an empty sheet, *i.e.* a sheet only with formatting information and an empty cell list (function *emptySheet*, which code is omitted here). Then, we add to this empty sheet the cells resulting from converting all rows in the table to a list of cells and this completes the translation. We added an extra argument to this function, namely the table name, so that we can rename each sheet in the workbook with the table identifier.

```
type TableName = String
writeTable :: (
  WriteRow (ks, vs) [Gmr'Cell],
  WriteRow (a, b) [Gmr'Cell],
  HeaderFor (a, b) ks vs
  ) ⇒ TableName → Table (a, b) ks vs → Gmr'Sheet
writeTable s (Table (a, b) m) = addCells cells emptysht
  where
    emptysht = emptySheet s
    cells = Gmr'Cells (header ++ cs)
    header = writeRow 0 0 (a, b)
    cs = worker 1 0 (Map.toList m)
    worker i j l
      | (List.null l) = []
      | otherwise = (writeRow i j (head l)) ++ (worker (i + 1) j (tail l))
```

In order to decrease the number of needed constraints when using the function *writeTable*, we have defined the following class and instance.

```
class (WriteRow (ks, vs) l, WriteRow h l, HeaderFor h ks vs) ⇒ WriteTable h ks vs l
instance (
  WriteRow (ks, vs) [Gmr′Cell],
  WriteRow (a, b) [Gmr′Cell],
  HeaderFor (a, b) ks vs
  ) ⇒ WriteTable (a, b) ks vs [Gmr′Cell]
```

Using this new definition the function *writeTable* will have the following signature.

```
writeTable :: WriteTable (a, b) ks vs [Gmr′Cell] ⇒ TableName → Table (a, b) ks vs → Gmr′Sheet
```

Writing several tables

To wrap up this functionality, we have lifted the previous operation to a relational database (as defined in Chapter 3), so that we can obtain a list of workbook sheets.

```
class WriteTables l l′ | l → l′ where
  writeTables :: l → ([Gmr′SheetName], l′)
instance WriteTables (Record HNil) [Gmr′Sheet] where
  writeTables _ = ([], [])
instance (
    Show l,
    WriteTables (Record ts) [Gmr′Sheet],
    WriteTable (a, b) ks vs [Gmr′Cell]
  )
    ⇒ WriteTables (Record (HCons (l, (Table (a, b) ks vs, fks)) ts)) [Gmr′Sheet] where
  writeTables (Record (HCons (l, (t, _)) ts)) = (name : names, (writeTable name t) : ts′)
    where name = Gmr′SheetName $ show l
          (names, ts′) = writeTables (Record ts)
```

Producing a Gnumeric workbook

With the previous functions we are now ready to define the function that, given a database in our model, returns the corresponding Gnumeric workbook. Similarly to what is done when translating a single table to a single sheet, we first create an empty workbook, with all formatting information but with an empty list of sheets. Then we add the list of sheets corresponding to the tables of the database.

```
writeRDB db = addSheetNameIndex sheetNameIndex $ addSheets sheets emptywb
  where emptywb = emptyWorkbook
    (sheetNames, shts) = writeTables db
    sheets = Gmr′Sheets shts
    sheetNameIndex = Just $ Gmr′SheetNameIndex sheetNames
```

6.2 Migration from Gnumeric format

We will now describe the converse process of converting a Gnumeric workbook into our database model. We will need a type for each table (sheet) to guide the process. We will additionally have a set of functional dependencies for each table that will be checked on the fly. Then, from the viewpoint of spreadsheet end-users, the database types are spreadsheet specifications which then offer interoperability with our model and functional dependency verification. As before, we will divide this process into simpler tasks. Remark that the workbook will be read as a tridimensional matrix of strings because:

1. A workbook is a list of sheets
2. Each sheet is a list of rows
3. Each row is a list of cells
4. A cell value is a String.

Our aim is to read this matrix into a database. We will first present how to read a single row, a single table and a list of tables.

Reading a single row

The process of reading a row according a list of attribute types is very simple, using the *fromString* function defined above.

```
class ReadRow h v | h → v where
  readRow :: h → [String] → v
instance ReadRow HNil HNil where
  readRow _ _ = HNil
instance (
  ReadShow v, ReadRow ats vs
  ) ⇒ ReadRow (HCons (Attribute v nr) ats) (HCons v vs) where
  readRow (HCons at ats) (s : ss) = HCons (fromString s) (readRow ats ss)
```

If the attribute is nullable or defaultable and the string is empty, *Nothing* or the default value are returned. If the string is not empty, the attribute type is required to be readable and parsing is performed.

```
instance (
  ReadShow v, ReadRow ats vs
  ) ⇒ ReadRow (HCons (NullableAttribute v nr) ats) (HCons (Maybe v) vs) where
  readRow (HCons at ats) ("" : ss) = HCons Nothing (readRow ats ss)
  readRow (HCons at ats) (s : ss) = HCons (Just $ fromString s) (readRow ats ss)
instance (
  ReadShow v, ReadRow ats vs
  ) ⇒ ReadRow (HCons (DefaultableAttribute v nr) ats) (HCons v vs) where
  readRow (HCons (Default v) ats) ("" : ss) = HCons v (readRow ats ss)
  readRow (HCons at ats) (s : ss) = HCons (fromString s) (readRow ats ss)
```

Notice that if the attribute is neither nullable nor defaultable, and the value string in the cell is empty, the instance for *Attribute v nr* defined above will produce an error.

We also need to lift this function to pairs, in order to cope with our key and non key separation.

```
instance (
  ReadRow a k, ReadRow b v,
  HLength a n, HNat2Integral n
  ) ⇒ ReadRow (a, b) (k, v) where
  readRow (a, b) l = let (k, v) = splitAt (hNat2Integral $ hLength a) l
                     in (readRow a k, readRow b v)
```

Reading a single table

In this subtask we produce a table from a matrix of strings according the given header. It is at this level that we perform functional dependency checking. Therefore, if the table violates the required dependencies an empty table is returned.

```
class ReadTable h fds t | h → t where
  readTable :: h → fds → [[String]] → t
instance (
  CheckFDs fds (Table (a, b) k v),
  HeaderFor (a, b) k v, ReadRow (a, b) (k, v)
  ) ⇒ ReadTable (a, b) fds (Table (a, b) k v) where
  readTable (a, b) fds rows | checkFDs fds t = t
                            | otherwise = Table (a, b) empty
    where t = Table (a, b) (fromList $ map (readRow (a, b)) rows)
```

Note that there is no need to have this function inside a class. However, to avoid long constraint headers in other functions we have used this class to gather some of the constraints.

Reading a list of tables

Given the previous operation, reading a list of tables is simply a map of *readTable* over the argument list.

```
class ReadTables h fds t | h fds → t where
  readTables :: h → fds → [[[String]]] → t
instance ReadTables HNil fds HNil where
  readTables _ _ _ = HNil
instance (
  ReadTables hs fds' ts,
  ReadTable (a, b) fds (Table (a, b) k v)
  ) ⇒ ReadTables (HCons (a, b) hs) (HCons fds fds') (HCons (Table (a, b) k v) ts) where
  readTables (HCons (a, b) hs) (HCons fds fds') (s : ss) = HCons (readTable (a, b) fds s) ts
    where ts = readTables hs fds' ss
```

Reading a bidimensional matrix of strings from a list of Cells

Each sheet in a workbook has a list of cells, each one with information about the position (column and row) and the value stored as a string. The following function maps a list of cells into a bidimensional matrix of strings, represented as a map ($r \mapsto (c \mapsto value)$). Note that the cells do not need to be ordered by row/column number. The function *insertWith* performs the insertion in the map at the right position.

```
readTable' (Gmr'Cells cells, max) = elems $ map elems (paddM max $ foldr worker empty cells)
  where worker (Gmr'Cell ats value) m = let (i :: Int) = read $ gmr'CellRow ats
                                            (j :: Int) = read $ gmr'CellCol ats
                                        in insertWith union i (singleton j value) m
```

The argument *max* above represents the number of columns the table should have, so that *paddM* can fill the positions in the map corresponding to empty cells in the spreadsheet with empty strings.

```
paddM max m = map paddRow m
  where paddRow m' = let ks = keys m'
                         ks' = difference [1 .. max] ks
                         newElems = foldr (λa m → insert a "" m) empty ks'
                     in union m' newElems
```

Notice that empty cells are not represented in the list of cells in a gnumeric workbook, so the padding with empty strings is essential to ensure that function *readRow* works as expected.

Producing a database from a Gnumeric workbook

Now that all required auxiliary functions have been defined, we can implement the main function, that reads a gnumeric workbook and produces a database in our model, while checking for well-formedness and referential integrity. Recall that each sheet in the workbook is assumed to hold a single table, the first row of which is the header (and therefore ignored).

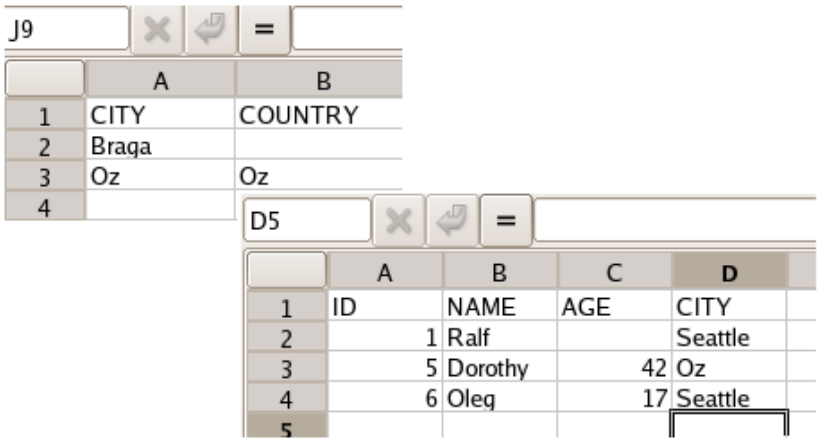
```
readRDB fp h fds tns fks = do (wb :: Gmr'Workbook) ← readGnumeric' fp
                              let cs = map cells (sheets wb)
                              let cols = hLengths h
                              let lCells = map (tail ∘ readTable') $ zip cs cols
                              let db = Record $ hZip tns $ hZip (readTables h fds lCells) fks
                              if (¬ $ checkRI db) then error "Referential integrity error"
                                                  else return $ DB db
```

The function *hLenghts* computes the length of each element in a list of lists. Applied to the list of headers, it computes a list with the number of columns each table should have.

6.3 Example

6.4 Checking integrity

Suppose you have a workbook (figure 6.2) made of two sheets – one mapping city names to country names and another mapping a unique identifier for a person to its name, age and city information.



J9

	A	B
1	CITY	COUNTRY
2	Braga	
3	Oz	Oz
4		

D5

	A	B	C	D
1	ID	NAME	AGE	CITY
2	1	Ralf		Seattle
3	5	Dorothy	42	Oz
4	6	Oleg	17	Seattle
5				

Figure 6.2: Tables contained in the sample gnumeric workbook

We can define a header structure for migrating this information to our strongly typed model and unique identifiers for the two tables[3].

$atAge' = \bot :: AttributeNull\ Int\ (\mathsf{PERS}, \mathsf{AGE})$
$myHeader' = (atID\ .*.\ HNil, atName\ .*.\ atAge'\ .*.\ atCity\ .*.\ HNil)$
$h = yourHeader\ .*.\ myHeader'\ .*.\ HNil$
data *cities*
data *people*
$ids = cities\ .*.\ people\ .*.\ HNil$

We want to associate to table *people* a foreign key relationship with table *cities*. The latter does not have any foreign key information associated (therefore it will be associated with *emptyRecord*). No functional dependencies (fds) are specified to be checked while parsing, since the only fds present are the ones *by construction*, automatically checked when a new row is inserted into the map which carries the data. The encoding of this information is as follows:

$myFK = (atCity\ .*.\ HNil, (cities, atCity'\ .*.\ HNil))$
$fks = emptyRecord\ .*.\ (Record\ \$\ myFK\ .*.\ HNil)\ .*.\ HNil$
$fds = emptyList\ .*.\ emptyList\ .*.\ HNil$
$emptyRecord :: Record\ HNil$
$emptyList :: HNil$

[3]These will be almost the same as those defined for *myTable* and *yourTable* in chapter 3, so some attribute definitions will be omitted.

The city `Seattle` appears in the people's table, but does not appear in the cities' table. Therefore, due to the foreign key relationship, a referential integrity error will be returned.

```
> do DB db <- readRDB "teste.gnumeric" h fds ids fks; putStrLn $ show db
*** Exception: Referential integrity error
```

6.5 Applying database operations

Suppose we have a workbook with the same structure as before, this time with no referential integrity violation (figure 6.3).

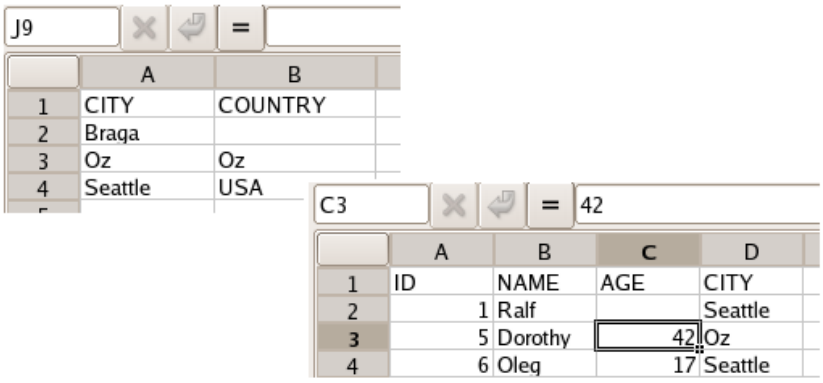


Figure 6.3: Tables contained in the sample gnumeric workbook

This time, we read the file and produce its database representation:

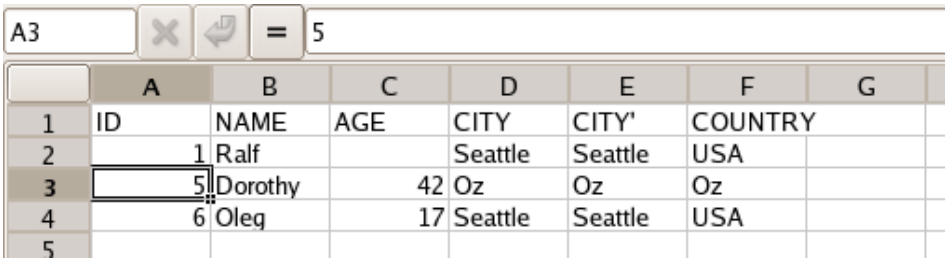
```
> do RDB db <- readRDB "teste.gnumeric" h fds ids fks; putStrLn $ show db
Record{
cities= (Table (HCons CITY' HNil,HCons COUNTRY HNil) {
        HCons "Braga" HNil:=HCons "Afghanistan" HNil,
        HCons "Oz" HNil:=HCons "Oz" HNil,
        HCons "Seattle" HNil:=HCons "USA" HNil},Record{}),
people= (Table (HCons ID HNil,HCons NAME (HCons AGE (HCons CITY HNil))) {
        HCons 1 HNil:=HCons "Ralf" (HCons Nothing (HCons "Seattle" HNil)),
        HCons 5 HNil:=HCons "Dorothy" (HCons (Just 42) (HCons "Oz" HNil)),
        HCons 6 HNil:=HCons "Oleg" (HCons (Just 17) (HCons "Seattle" HNil))},
        Record{CITY.*.HNil=(Cities,HCons CITY HNil)})
}
```

Note that the country associated with `Braga` is now `Afghanistan`, the default value for attribute *atCountry*, and that attribute age for `Ralf` has value *Nothing*, since *atAge* is a nullable attribute and the corresponding cell in the gnumeric sheet was empty.

Having the data stored in a strongly typed relational database we can now apply SQL and database transformation operations and before returning to gnumeric. Note however that the padding with default values will add extra information (that can be undesirable in some situations). Let us assume that we were interested in a *denormalization* operation, namely in replacing the two existing tables by a single one containing all information. The following example illustrates how we could do this in our model.

```
> do DB db <- readRDB "teste.gnumeric" h fds ids fks;
     let (db', fds)  = compose db (atCity .*. HNil) people cities newT;
     putStrLn $ show fds;
     writeFile "result.gnumeric" $ showXml $ writeRDB db'
HCons (FD (HCons CITY' HNil) (HCons COUNTRY HNil))
    (HCons (FD (HCons CITY HNil) (HCons CITY' HNil)) HNil)
```

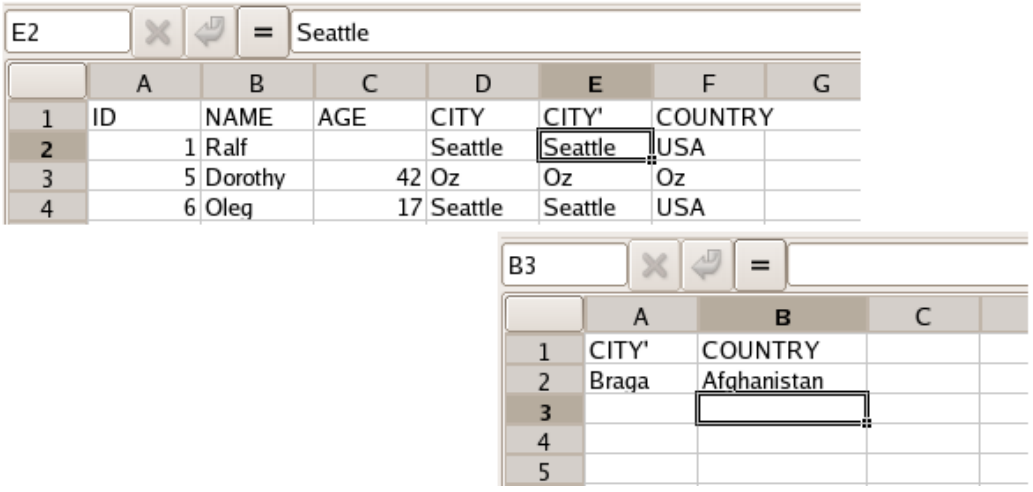
We show the functional dependency information returned together with the new database. The first functional dependency corresponds to the one present *by construction* in the table $cities$ ($atCity' \rightharpoonup atCountry$) and the second one is produced by the foreign key associated with the same table. The resulting workbook can be seen in figure 6.4.



	A	B	C	D	E	F	G
1	ID	NAME	AGE	CITY	CITY'	COUNTRY	
2	1	Ralf		Seattle	Seattle	USA	
3	5	Dorothy	42	Oz	Oz	Oz	
4	6	Oleg	17	Seattle	Seattle	USA	
5							

Figure 6.4: Resulting workbook after database transformation

Notice that the produced table does not have any information about city `Braga`, because no individual from this city could be found in table $people$. Therefore, should we try to recover the original workbook from that of figure 6.4 some information would be missing. In order to avoid this loss of information we should use the version of $compose$ that always keeps the information from the second table involved in the join in the database, namely all the tuples that are not linked by the foreign key relationship. In the previous example the resulting workbook would have two tables, as can be observed in figure 6.5.



	A	B	C	D	E	F	G
1	ID	NAME	AGE	CITY	CITY'	COUNTRY	
2	1	Ralf		Seattle	Seattle	USA	
3	5	Dorothy	42	Oz	Oz	Oz	
4	6	Oleg	17	Seattle	Seattle	USA	

	A	B	C
1	CITY'	COUNTRY	
2	Braga	Afghanistan	
3			
4			
5			

Figure 6.5: Resulting workbook after database transformation

The table mapping cities to countries information only has one tuple now, since all the others in the original table are represented in the new table in the database.

From the point of view of a Haskell programmer, the migration of spreadsheet data to our strongly typed model is an easy way of inserting data into tables.

From the end-user point of view, the fact that the headers must be specified in Haskell is a drawback. However, after having the data stored in our model, the user has available valuable operations on data and refinement laws (*e.g.*, normalization). As presented in the first example, we can use our model to check data integrity in a spreadsheet. This can be useful to guarantee well-formedness of spreadsheets after each modification. For instance, after each insertion in a spreadsheet the data could be migrated to our database model for checking the referential integrity.

The migration defined in this chapter is a first approach to spreadsheet refactoring. The interaction with the user needs to be improved, so that this migration becomes suitable for effective use by end-users.

Chapter 7

Concluding remarks

We have defined a datatype for relational database tables that captures key meta-data, which in turn allows us to define more precise types for database operations. For instance, the join operator on tables guarantees that in the on clause a value is assigned to all keys in the second table, meaning that join conditions are not allowed to underspecify the row from the second table. Furthermore, as an immediate consequence from working in a higher-order functional language, we can mix join and cartesian product operators to create join expressions that go beyond SQL's syntax limits.

We have achieved further well-formedness criteria by the definition of functional dependencies and normal forms in our model. Moreover, we have defined a new level of operations that carry functional dependency information, automatically informed by the type-checker.

We have shown that more type precision allows a more rigorous and ultimately safer approach to off-line situations, such as database design and database migration.

We have shown that the use of our model in spreadsheet refactoring and migration is of interest. After migration, the user has all operations we have defined on tables available and can change the data and check integrity. However, this migration *bridge* is currently only a prototype and leaves room for improvement.

Although our model of SQL is not complete, the covered set of features should convince the reader that a comprehensive model is within reach. The inclusion of functional dependency information in types goes beyond SQL standard, as do operations for database transformation. Below we highlight future work directions.

7.1 Contributions

We have shown that capturing the metadata of a relational table at the type level enables more static checks for database designers. More concretely we have accomplished the following:

1. We have defined a strongly typed representation of relational databases, including metadata such as foreign keys. Furthermore, we have encoded strongly typed (basic) operations derived from relational algebra and several SQL statements. Interestingly enough, we have achieved more precise types for some SQL operations, such as join and insertion.

2. We have captured database design information in types, namely functional dependencies, and we have shown that normal form checking and database transformation operations such as normalization and denormalization are easily expressed in our model.

3. We have shown the uselfuness of our model for spreadsheet users, providing migration operations.

7.2 Future work

Ohori *et al.* model generalized relational databases and some features of object-oriented databases [32, 9]. It would be interesting to see if our approach can be generalized in these directions as well. In particular, a fusion with the model of object-oriented programming inside Haskell given by Kiselyov *et al.* [24] could be attempted to arrive at a unifying model for the object-oriented, functional, and relational paradigms.

The approach of Cunha *et al.* [13] to two-level data transformation and our approach to relational database representation and manipulation have much in common, and their mutual reinforcement is a topic of ongoing study. For instance, our type-level programming techniques could be employed to add sophistication to the GADT and thus allow a more faithful, but still safe representation of relational databases at term level.

Our Haskell model of SQL might be elaborated into a high-level solution for database connection, similar to Haskell/DB. This would require that the Table objects no longer contain maps (unless perhaps as cache), and that table operations are implemented differently than in terms of map operations. Instead, one could imagine that table operations are implemented by functions that generate SQL concrete syntax statements, communicate with the server, and map the answers of the server back to $\mathit{ResultList}$ objects. The header and functional dependency information would meanwhile be maintained as before.

When mainstream programming languages such as Java or C are used to develop database application programs, it is current practice to prepare SQL statements in string representations before sending them over the database connection to the DBMS. Such SQL statements are generally constructed at run-time from parameterized queries and user-supplied input. A drawback is that these queries can not be checked statically, which may lead to run-time errors or security gaps (so-called SQL injection attacks). Our reconstruction of SQL demonstrates that strong static checking of (parameterized) queries is possible, including automatic inference of their types.

We share a number of concerns regarding usability and performance with the authors of the OOHASKELL library. In particular, the readability of inferred types and the problem-specificity of reported type errors, at least using current Haskell compilers, leaves room for improvement. Performance is an issue when type-level functions implement algorithms with non-trivial computational complexity or are applied to large types. Our algorithm for computing the transitive closure of functional dependencies is an example. Encoding of more efficient data structures and algorithms on the type-level might be required to ensure scalability of our model.

We have shown, for particular operations, how we can transport functional dependency information from argument to result tables. It would be interesting to develop a formal calculus that would allow to automatically compute this information for further operations.

Evidence of the effectiveness of our model in spreadsheet reverse engineering needs

some more work. The format we assume for spreadsheets is not a standard and the fact the header structure must be defined in Haskell is a disadvantage. A more intuitive specification front-end for our model and mining functional dependencies from the actual data would be valuable features for the end user.

Bibliography

[1] R. Abraham and M. Erwig. Header and unit inference for spreadsheets through spatial analyses. In *VLHCC '04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing (VLHCC'04)*, pages 165–172, Washington, DC, USA, 2004. IEEE Computer Society.

[2] R. Abraham and M. Erwig. Goal-directed debugging of spreadsheets. In *VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, pages 37–44, Washington, DC, USA, 2005. IEEE Computer Society.

[3] R. Abraham and M. Erwig. How to communicate unit error messages in spreadsheets. In *WEUSE I: Proceedings of the first workshop on End-user software engineering*, pages 1–5, New York, NY, USA, 2005. ACM Press.

[4] R. Abraham, M. Erwig, S. Kollmansberger, and E. Seifert. Visual specifications of correct spreadsheets. In *VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, pages 189–196, Washington, DC, USA, 2005. IEEE Computer Society.

[5] C. Beeri et al. A complete axiomatization for functional and multivalued dependencies in database relations. In *SIGMOD Conference*, pages 47–61, 1977.

[6] R. Bird and O. de Moor. *Algebra of programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[7] A. F. Blackwell, M. M. Burnett, and S. P. Jones. Champagne prototyping: A research technique for early evaluation of complex end-user programming systems. In *VLHCC '04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing (VLHCC'04)*, pages 47–54, Washington, DC, USA, 2004. IEEE Computer Society.

[8] B. Bringert et al. Student paper: HaskellDB improved. In *Haskell '04: Proc. 2004 ACM SIGPLAN workshop on Haskell*, pages 108–115. ACM Press, 2004.

[9] P. Buneman and A. Ohori. Polymorphism and type inference in database programming. *ACM Trans. Database Syst.*, 21(1):30–76, 1996.

[10] M. Burnett and M. Erwig. Visually customizing inference rules about apples and oranges. In *HCC '02: Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)*, page 140, Washington, DC, USA, 2002. IEEE Computer Society.

[11] J. Cheney and R. Hinze. First-class phantom types. Technical report, Cornell University, 2003.

[12] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[13] A. Cunha, J. N. Oliveira, and J. Visser. Type-safe two-level data transformation – with derecursivation and dynamic typing. Accepted for publication in Formal Methods 2006, Lecture Notes in Computer Science, July 2006, Springer©. A preliminary version with additional material appended appeared as technical report DI-PURe-06.03.01, 2006.

[14] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems, 2nd Edition.* Benjamin/Cummings, 1994.

[15] M. Erwig, R. Abraham, I. Cooperstein, and S. Kollmansberger. Automatic generation and maintenance of correct spreadsheets. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 136–145, New York, NY, USA, 2005. ACM Press.

[16] M. Erwig, R. Abraham, I. Cooperstein, and S. Kollmansberger. Gencel - a program generator for correct spreadsheets. *Journal of Functional Programming*, 2005.

[17] M. Erwig and M. M. Burnett. Adding apples and oranges. In *PADL '02: Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages*, pages 173–191, London, UK, 2002. Springer-Verlag.

[18] T. Hallgren. Fun with functional dependencies. In *Proceedings of the Joint CS/CE Winter Meeting*, pages 135–145, 2001. Department of Computing Science, Chalmers, Sweden.

[19] R. Hinze. Fun with phantom types. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*, pages 245–262. Palgrave Macmillan, 2003. ISBN 1-4039-0772-2 hardback, ISBN 0-333-99285-7 paperback.

[20] ISO/IEC. Database language SQL (SQL-92 or SQL2). Technical Report 9075:1992, ISO/IEC, 1992.

[21] A. Jaoua et al. Discovering Regularities in Databases Using Canonical Decomposition of Binary Relations. *JoRMiCS*, 1:217–234, 2004.

[22] S. L. P. Jones. Haskell 98: Language and libraries. *J. Funct. Program.*, 13(1):1–255, 2003.

[23] S. P. Jones, A. Blackwell, and M. Burnett. A user-centred approach to functions in excel. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 165–176, New York, NY, USA, 2003. ACM Press.

[24] O. Kiselyov and R. Lämmel. Haskell's overlooked object system. Draft of 10 September 2005, 2005.

[25] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004.

[26] D. Leijen and E. Meijer. Domain specific embedded compilers. *SIGPLAN Not.*, 35(1):109–122, 2000.

[27] D. Maier. *The Theory of Relational Databases.* Computer Science Press, 1983.

[28] D. Maier, Y. Sagiv, and M. Yannakakis. On the complexity of testing implications of functional and join dependencies. *J. ACM*, 28(4):680–695, 1981.

[29] C. Necco. Procesamiento de datos politípicos (polytypic data processing). Master's thesis, Universidad Nacional de San Luis, Departamento de Informática, Argentina, 2004.

[30] C. Necco and J. Oliveira. Toward generic data processing. In *In Proc. WISBD'05*, 2005.

[31] M. Neubauer, P. Thiemann, M. Gasbichler, and M. Sperber. A functional notation for functional dependencies. In R. Hinze, editor, *Proceedings of the 2001 Haskell Workshop*, 2001.

[32] A. Ohori and P. Buneman. Type inference in a database programming language. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 174–183, New York, NY, USA, 1988. ACM Press.

[33] A. Ohori, P. Buneman, and V. Tannen. Database programming in Machiavelli - a polymorphic language with static type inference. In J. Clifford, B. G. Lindsay, and D. Maier, editors, *Proc. of 1989 ACM SIGMOD Inter. Conference on Management of Data, Portland, Oregon, May 31 - June 2, 1989*, pages 46–57. ACM Press, 1989.

[34] J. Oliveira. First steps in pointfree functional dependency theory. Manuscript in preparation, available from http://www.di.uminho.pt/j̃no.

[35] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. Apr. 2006.

[36] A. Silva and J. Visser. Strong types for relational databases. Draft of 28 March 2006, 2006.

[37] L. Sterling and E. Y. Shapiro. *The Art of Prolog - Advanced Programming Techniques, 2nd Ed.* MIT Press, 1994.

[38] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I*. Computer Science Press, 1988.