Strong Types for Relational Databases
(Functional Pearl)

Alexandra Silva *
CWI, The Netherlands
ams@cwi.nl

Joost Visser †
Universidade do Minho, Portugal
joost.visser@di.uminho.pt

Abstract

Haskell's type system with multi-parameter constructor classes and functional dependencies allows static (compile-time) computations to be expressed by logic programming on the level of types. This emergent capability has been exploited for instance to model arbitrary-length tuples (heterogeneous lists), extensible records, functions with variable length argument lists, and (homogenous) lists of statically fixed length (vectors).

We explain how type-level programming can be exploited to define a strongly-typed model of relational databases and operations on them. In particular, we present a strongly typed embedding of a significant subset of SQL in Haskell. In this model, meta-data is represented by type-level entities that guard the semantic correctness of database operations at compile time.

Apart from the standard relational database operations, such as selection and join, we model functional dependencies (among table attributes), normal forms, and operations for database transformation. We show how functional dependency information can be represented at the type level, and can be transported through operations. This means that type inference statically computes functional dependencies on the result from those on the arguments.

Our model shows that Haskell can be used to design and prototype typed languages for designing, programming, and transforming relational databases.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; E.1 [*Data Structures*]: Records; H.2.1 [*Database Management*]: Logical design—Data models, normal forms, Schema and subschema; H.2.3 [*Database Management*]: Languages—Data manipulation languages, Query languages

General Terms Algorithms, Design, Languages, Theory

Keywords Type-level programming, Haskell, relational databases, SQL, functional dependency theory

* Supported by the Fundação para a Ciência e a Tecnologia, Portugal, under grant number POSI/ICHS/44304/2002.

† Supported by the Fundação para a Ciência e a Tecnologia, Portugal, under grant number SFRH/BPD/11609/2002.

Haskell'06 September 17, 2006, Portland, Oregon, USA.

1. Introduction

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 in 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 oversized rows, 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 labour between static and dynamic checking of database operations is constrained by the degree of precision with which types can be assigned to operations and their sub-expressions.

In this paper, we show that 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 that 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. For example, the following (inner) joins on a table `T` with foreign key `T.FK` and a table `S` with primary key `S.PK` may all be legal SQL:

```
select X,Y from T join S on T.FK=S.PK
select X,Y from T join S on T.FK=S.Z
select X,Y from T join S on T.Z=S.PK
select X,Y from T join S on T.FK=S.PK and T.FK=S.Z
```

But the latter three, which mis-specify the join conditions, can be statically rejected when assigning more precise types.

But 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 [25, 17, 6, 9]. Such normal forms are defined in terms of *functional dependencies*[1] between (groups of) columns that are or are not allowed to be present [2]. We will show that functional dependency information can be captured in types as well, and

[1] Below we will use *functional dependencies between parameters of type classes*. These are similar, but not to be confused with functional dependencies among table attributes in database theory after which they are named.

can be transported through operations. Thus, the type-checker will infer functional dependencies on the result tables from functional dependencies on the argument tables. For example, if we assume `T.PK` is the primary key of table `T`, then the first of the joins above can include additional functional dependencies in its result type, such as:

```
T.PK -> S.PK        S.PK -> T.FK
```

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 perform 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.

Plan of the paper

Our model of relational databases is specified in Haskell, using a technique called type-class-based programming, or type-level programming. For self-containement, Section 2 explains the basics of Haskell and the technique. We make essential use of the Haskell HLIST library, which offers arbitrary-length tuples and extensible polymorphic records with first-class labels and subtyping [14]. The essentials of this library are introduced in the same section.

In Section 3, we present the first part of our model: a type-ful reconstruction of statements and clauses of the SQL language. This part of the model provides static type checking and inference of well-formedness constraints normally specified in a schema. In Section 4, we turn to the second part of the 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. Finally, in Section 5 we go beyond database modeling and querying, by addressing database transformations, such as normalization and migration. Related work is discussed in Section 6 and Section 7 concludes.

2. Type-level programming

Haskell is a non-strict, higher-order, typed functional programming language [23]. 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, i.e. $f\ x\ y$ is favored over $f(x,y)$. Functions may be applied partially 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 paper, 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 Type classes

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

data *Bool* = *True* | *False*
data *Tree a* = *Leaf a* | *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* → *Int*
depth (*Leaf a*) = 0
depth (*Fork ts*) = 1 + *maximum* (0 : (*map depth ts*))

Here *maximum* and *map* are standard list processing functions.

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* → *String*
instance *Show Bool* **where**
  show True = `"True"`
  show False = `"False"`
instance (*Show a*, *Show b*) ⇒ *Show* (*a*, *b*) **where**
  show (*a*, *b*) = `"("` ++ *show a* ++ `","` ++ *show b* ++ `")"`

The second instance demonstrates 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* ⇒ *a* → *String*.

Type classes can have more than a single type parameter:

class *Convert a b* | *a* → *b* **where**
  convert :: *a* → *b*
instance *Show a* ⇒ *Convert a String* **where**
  convert = *show*
instance *Convert String String* **where**
  convert = *id*

The clause | *a* → *b* denotes a *functional dependency among type parameters* (similar to, but not to be confused with functional dependencies among table attributes in database theory) which declares that the parameter *a* uniquely determines the parameter *b*. This dependency is exploited for type inference by the compiler: when type *a* is instantiated, the instantiation of type *b* is inferred. Note also that the two instances above are *overlapping* in the sense that a particular choice of types can match both instances. The compiler will select the most specific instance in such cases. Both multi-parameter type-classes with functional dependencies and permission of overlapping instances go beyond the Haskell 98 language standard, but these extensions are commonly used, supported by compilers, and well-understood semantically [24].

2.2 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. And interestingly, 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 [10]. 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*. The computation is carried out by the type checker! The execution model for type-level predicates and functions is similar to that of logic programming languages, such as Prolog.

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* = ⊥ :: *Zero*
data *Succ n*; *succ* = ⊥ :: *n* → *Succ n*

```
class Nat n
instance Nat Zero
instance Nat n ⇒ 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. Note the use of the undefined value $\bot$, inhabiting any Haskell type, to create dummy values for types on which we intend to do static computations only.

2.3 The HLIST library

Type-level programming has been exploited by Kiselyov *et al.* to model arbitrary-length tuples, or *heterogeneous lists*[2] [14]. 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. In fact, the authors were motivated by application to database connectivity and already reported progress in that direction (see Section 6).

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 *HNil*. Thus, heterogeneous lists follow the normal cons-list construction pattern on the type-level. The *myTuple* 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 should be pairwise distinct on the type level, and a type-level predicate is supplied to enforce this. Here we use type-level naturals as labels, but other possibilities exist, as we will show later. 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:

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

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

Here, *hAppend* concatenates two heterogeneous lists, the functions *hZip* and *hUnzip*, respectively, turn two lists into a list of pairs and *vice versa*, and *hLookupByLabel* returns the value in a record corresponding to a given label.

Syntactic sugar is provided by infix operators and an infix type constructor synonym, allowing prettier syntax e.g. for *myRecord*:

```
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 deleting and retrieving record values, for updating one record with the content of another, and for modifying the value at a given label:

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

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

3. The SQL layer

The Structured Query Language (SQL) [8, 1] is the most widely used language for programming relational databases. It offers a declarative language, based on relational algebra, that allows information to be retrieved from and stored into tables. We will present our model of the SQL language in two steps: representation of databases, and operations on them.

3.1 Representation of databases

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

```
data HList row ⇒ Table row = Table (Set row)
data TableList t ⇒ RDB t = RDB t

class TableList t
instance TableList HNil
instance (HList v, TableList t) ⇒ 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. Firstly, schema information is not represented. This implies that operations on the database may not respect the schema and can not take advantage of it, unless separate schema information would be fed to them. Secondly, the choice of *Set* to collect the rows of a table does not do justice to the fact that database tables are in fact mappings from key attributes to non-key attributes.

Tables with attributes

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

[2] We will use the terms 'arbitrary-length tuple' and 'heterogeneous list' interchangeably. They are fundamentally different from normal, 'homogenous lists', which hold elements of a single type only.

```
data HeaderFor h k v ⇒ Table h k v = Table h (Map k v)
class HeaderFor h k v | h → 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 that 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.

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

```
data Attribute t name
attr = ⊥ :: 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. Note that *t* and *name* are so-called *phantom* type arguments, 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   = attr :: Attribute Int    (PEOPLE ID)
data NAME; atName = attr :: Attribute String (PEOPLE NAME)
data PEOPLE a; people = ⊥ :: PEOPLE ()
```

Note that no values of the attributes' column types (*Int* and *String*) need to be provided, since these are phantom type arguments. Since we intend to have several tables with similar attributes, we have used a single-argument type constructor to have *qualified* names. Using these attributes and a few more, a valid example table can be created as follows[3]:

```
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 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 attempt to insert values of the wrong type, or value lists of the wrong length will lead to type check errors. We will encounter such situations below.

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, we have defined similar datatypes called *AttrNull* and *AttrDef* with corresponding instances for the *AttributesFor* class.

[3] The $ operator is just function application with low binding force; it allows us to write fewer parentheses.

```
data AttrNull t nm
data AttrDef t nm = Default t

instance AttributesFor a v ⇒
  AttributesFor (HCons (AttrDef t nm) a) (HCons t v)
instance AttributesFor a v ⇒
  AttributesFor (HCons (AttrNull t nm) a) (HCons (Maybe t) v)
```

For brevity, we omit attributes that can both be null and have a declared default.

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

```
class Defaultable x where defaultValue :: x
instance Defaultable Int where defaultValue = 0
instance Defaultable String where defaultValue = ""
```

Examples of such attributes will appear below.

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:

```
data FK fk t pk = FK fk t pk
```

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

```
data COUNTRY;
atCity' = attr :: Attribute String (CITIES CITY)
atCountry :: AttrDef String (CITIES COUNTRY)
atCountry = Default "Afghanistan"

data CITIES a; cities = ⊥ :: CITIES ()

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

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

Thus, the *myFK* constraint links the *atCity* attribute of *myTable* to the primary key *atCity'* of *yourTable*. Note that *atCountry* is an attribute with declared default.

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, HNil) .*.
  people .=. (myTable, myFK .*. HNil) .*. HNil
```

Figure 1 depicts *myRDB*'s schema. Thus, 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. On the schema level, this means we want all attributes to be unique, and we want foreign key constraints to refer to existing attributes and table

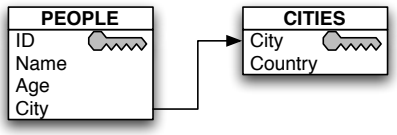

Figure 1. Example database schema diagram.

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. The source distribution of the paper can be consulted for details. The data constructor *RDB* encapsulates databases that satisfy our schema-level well-formedness demands, and on which the *checkRI* predicate can be run to check for dangling references.

The database *myRDB* defined above 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 predicate *checkRI* returns *False*, because the cities *Seattle* and *Oz* do not appear in *yourTable*.

3.2 Table operations

Given the database representation defined above, we can turn our attention to operations on database tables. There are several main challenges to be met.

Firstly, operations that involve more than a single table type will require additional constraints or type-level functions to properly relate or derive the types involved. In the case of *projection*, for example, the type of the result table must be computed from the type of the input table and the selected column types.

Secondly, a faithful modeling of the SQL language will require a certain degree of 'intelligence' 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.

Thirdly, the interface provided by the SQL language shields off the distinction between key attributes and non-key attributes which is present in the underlying tables. This distinction is relevant for the behaviour 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 on the type level.

The WHERE clause

Various SQL statements can contain a WHERE clause that 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.

We can model value access with record lookup. To this end, we model a row as a record that has attributes as its labels. A predicate is then a boolean function over that record. To compute the type of the record from the table header, we employ the following type-level function:

```
class Row h k v r | h → k v r where
  row :: h → k → v → r
  unRow :: h → r → (k, v)
instance (
  HeaderFor (a, b) k v, HAppend a b ab, HAppend k v kv,
  HZip ab kv l, HBreak kv k v
  ) ⇒ Row (a, b) k v (Record l)
  where
    row (a, b) k v = Record $ hZip (hAppend a b) (hAppend k v)
    unRow (a, b) (Record l) = hBreak $ snd $ hUnzip l
```

Here, *HBreak* is the inverse of *HAppend* and breaks a list into two pieces. 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:

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

And a predicate over such a row might look as follows:

```
isOzSenior = λr → (r .!. atAge) > 65 ∧ (r .!. atCity) ≡ "Oz"
```

The type of the predicate is inferred automatically:

```
isOzSenior :: (
  HasField (Attribute Int AGE) r Int,
  HasField (Attribute String CITY) r String
) ⇒ r → Bool
```

Interestingly, this type is valid for any row that has the *atAge* and *atCity* in any order. If other columns are joined or projected away, the predicate will still type-check and behave correctly.

The DELETE statement

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, e.g. `delete from PEOPLE where Age>65 and City="Oz"`. We model deletion via the library function *filterWithKey* for maps:

```
delete (Table h m) p = Table h m'
  where
    m' = filterWithKey keep m
    keep k v = ¬ (p $ row h k v)
filterWithKey :: Ord k ⇒ (k → a → Bool) → Map k a → Map k a
```

Only rows that fail the predicate pass through to the result map.

The UPDATE statement

The UPDATE statement involves a SET clause that assigns new values to selected columns, e.g. `update PEOPLE set City="Oz" where Name="Dorothy"`. A record is again an appropriate structure to model these assignments. Updating of a row according to column assignments then boils down to updating one record with the values from another, possibly smaller record. The record operation *updateWith* (defined in Section 2.3) can be used for this.

```
update (Table h m) s p = Table h (foldWithKey upd empty m)
  where
    upd k v
      | p r = insert k' v'
      | otherwise = insert k v
      where r = row h k v
            (k', v') = unRow h $ updateWith r s
foldWithKey :: (k → a → b → b) → b → Map k a → b
```

Here we use the library function *foldWithKey* on maps, and we use Haskell's *guarded* equation syntax to define a helper function *upd*. 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 *updateWith* 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.

The INSERT statement

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, e.g. `insert CITIES (Country, City) values ("USA","Portland")`. If for certain columns no values are supplied, a default value or NULL is inserted if the attribute concerned has been declared with a default or to allow null, e.g. the query `insert CITIES values ("Amsterdam")` would insert the row `("Amsterdam","Afghanistan")` in the table. We have defined a type-level function *LineUp* to perform such alignment and padding:

```
class LineUp h r kv r' | h r → kv r' where
  lineUp :: h → r → (kv, r')
```

Here, *lineUp* takes meta-information *h* and a row *r* created from a column specification and values to be inserted, and it produces a pair *kv* of reordered and padded lists of keys and values, together with possibly remaining row fragment r'. The *lineUp* function is careful not to perform padding in keys, and for some specific types it can insert system-level defaults, such as the empty string for *String* and 0 for integers.

Now, the single-row variant of INSERT can be specified:

```
insertValues (Table h m) a x = Table h m'
  where
    m' = insert k v m
    (k, v) = fst $ lineUp h (Record $ hZip a x)
```

Here the *hZip* function is used to pair up the list of columns with the list of values. The *lineUp* function permutes and padds the resulting list of column-value pairs into a properly ordered row for the given table.

Any attempt 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 examples:

```
insertValues yourTable (atCity' .*. atCountry .*. HNil)
  ("Paris" .*. "France" .*. 13 .*. HNil)
insertValues myTable (atID .*. atName .*. atAge .*. HNil)
  (1 .*. "Joe" .*. 62 .*. HNil)
insertValues yourTable (atCity' .*. atCountry .*. HNil)
  ("Paris" .*. 11 .*. HNil)
```

All of these examples fail to type-check, as they should.

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 list fold over the rows in a *ResultList*:

```
insertResultList tbl as (ResultList a v)
  = foldr (λvs t → insertValues t as vs) tbl v
```

Note that the meta-information of the inserted result list is ignored. The specified column list is used instead.

The SELECT statement

Several kinds of functionality are bundled into the SELECT statement. The main ones are 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:

```
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 *hAppend* invocations 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:

```
productT (Table (a, b) m) (Table (a', b') m') = Table h'' m''
  where
    h'' = (hAppend a a', hAppend b b')
    m'' = product' m m'
```

Since the SELECT statement allows 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 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]
```

The first argument corresponds to the presence of the DISTINCT keyword, and determines whether duplicates will be removed from the result. The second argument are the specified columns, to be used in projection. The third argument represents the FROM clause, from which the *products* function computes a table with type

Table h k v. The fourth 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* function for multiple label lookup from a record. The last argument represents the ORDER BY clause, in which attributes can be specified by which the list of results should be sorted.

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 subjected 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 duplicates, 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.

The JOIN clause

The SQL language allows tables to be joined in several different ways, in addition to the cartesian product. Here we will show the *inner* join, where values in one table are linked to primary keys of a second table, e.g. `select * from PEOPLE join CITIES on PEOPLE.City=CITIES.City`. 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:

```
join (Table h@(a, b) m) (Table (a', b') m') on = Table h'' m''
  where
    h'' = (a, hAppend b (hAppend a' b'))
    m'' = joinM (λk v → rstrct $
                        lineUp (a', HNil) (on $ row h k v)) m m'
    rstrct ((k', HNil), Record HNil) = k'
```

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 assign a foreign key to a primary one, as follows:

```
myOn = λr → ((atPK . = . (r .!. atFK)) . * . 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 first table. Thus, our join is typed more precisely than the SQL join since join conditions are not allowed to underspecify or overspecify the row from the second table.

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

```
seniorAmericans
 = select False (atName .*. atCountry .*. HNil)
   ((myTable `join` yourTable
     (λr → atCity' .=. (r .!. atCity) .*. HNil))
    .*. HNil)
   (λr → (r .!. atAge) > 65 ∧ (r .!. atCountry) ≡ "USA")
```

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* invocations. In fact, the join and cartesian product operators can be mixed to create join expressions beyond SQL's syntactic limits. This is an immediate consequence from working in a higher-order functional language.

The GROUP BY clause and aggregation functions

When the SELECT statement is provided with a GROUP BY clause, it can have 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 gets applied *after* grouping. For example, the query `select City,count(*) from PEOPLE group by City` would calculate the number of persons that live in each city.

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

```
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
```

The parameter *g* serves to compute from a map entry a new key under which to group that entry. The 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 $r \rightarrow a$ computes a value from each row, and the reduce function of type $[a] \rightarrow b$ reduces a list of such values to a single one. As exemplified 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 behaviour to the select statement. For brevity we do not present the resulting function *selectG* in full, but the interested reader can find details in the source distribution of this paper.

Database operations

We can lift the operations we 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 a utility on records that applies a given function on the value identified by a given label.

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

4. Functional dependencies

In the preceding sections 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 section, 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.

DEFINITION 1. *Given a table header H 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 [6], it has been thoroughly studied [2, 12], and is part of standard database literature [17, 25, 9].

A type-level representation of functional dependencies is given in Section 4.1. We proceed in Section 4.2 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 4.3 type-level predicates are defined for Boyce-Codd normal form and third normal form. Finally, in Section 4.4 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.

4.1 Representation

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

```
data FunDep x y ⇒ FD x y = FD x y

class FunDep x y
instance (AttrList x, AttrList y) ⇒ FunDep x y

class AttrList ats
instance AttrList HNil
instance AttrList l ⇒ 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 predicate:

```
class FDListFor fds h
instance (
  Contains fds (FD a b), FDList fds, AttrListFor fds ats,
  HAppend a b ab, ContainsAll ats ab
  ) ⇒ FDListFor fds (a, b)
```

Here the functional dependency from a table's keys to its values, which holds 'by construction' is required to be in the list of FDs. Further, the *FDList* predicate constrains the list to contain functional dependencies only, and the type level function *AttrListFor* computes the attributes used in a given list of FDs.

4.2 Keys and superkeys

In section 3.2, 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 functional dependencies associated with it. Every set of attributes $X \subseteq H$, such that $X \rightarrow H$ can be deduced from F and X is minimal, is a key. X is minimal if for no proper subset Y of X we can deduce $Y \rightarrow H$ from F.*

An essential ingredient into 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. But, 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 set 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 closure is described in [25, p.338]. We implemented it on the type level with a constraint named *Closure*.

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 $Y \rightarrow H$.

With all ingredients defined, we proceed to the specification of the constraint that tests whether a given set of attributes 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*

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

4.3 Normal forms

There are several normal forms, but we will only discuss the most significant ones – third normal form (NF) and Boyce-Codd NF. For simplicity we will assume that FDs are represented with a single attribute in the consequent.

Boyce Codd normal form

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 nontrivial dependencies are those in which a key determines one or more other attributes [25]. 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 just want to check if X is a superkey when Y is not in X. Now, we can extrapolate this definition to 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*

Examples of verification of this normal form can be found elsewhere [25].

Third normal form

Before defining third normal form we need to define the notion of prime attribute [17].

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.*

The encoding of this definition is as follows.

class *IsPrime at all fds b* | *at all fds* → *b*
instance (*Keys all fds lk*, *MemberOfAnyKey at lk b*)
⇒ *IsPrime at all fds b*

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 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 single-FD constraint is lifted to a constraint on a set of FDs, just as we did in the case of the Boyce-Codd NF.

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

4.4 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. The dependency *FD k v* that holds 'by construction' is always present in that set.

Project

When we project a table with associated FDs F through 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 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**
projectFD :: *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 check that X only contains attributes that are in B, which is equivalent to verifying the equality $X - B = \{\}$. 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 type-level calculation can be linked to a value-level projection operation that restricts a table to a list of selected attributes:

```
projectValues b' (Table' (a, b) m fds)
  = Table' (a, b') m' (projectFD b' fds)
    where m' = Map.map (fst ∘ lineUp b') m
```

Thus, this function provides a restricted projection operation that preserves keys, and transports all relevant functional dependencies to the result table. Such operations can be useful in database transformation scenarios, as we will explain below.

Join

When we join two tables with associated FDs F and F', then in the new table all the FDs $f \in F \cup F'$ will hold. In addition, the attributes from the second table will become functionally dependent on the keys of the first table. This calculation can be simply linked to the function *join* described in Section 3.2:

```
join' (Table' h m fds) (Table' h' m' fds') r = Table' h'' m'' fds''
  where
    Table h''@(a, ba'b') m'' = join (Table h m) (Table h' m') r
    fds'' = HCons (FD a ba'b') (union fds fds')
```

When using this augmented join on the first query example of the introduction (i.e. `select X,Y from T join S on T.FK=S.PK`), the result table will include the functional dependency `T.PK->S.PK` mentioned there. To also obtain the other dependency, `S.PK->T.FK` a further restriction is needed, as we will see below.

5. Database Transformation

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. We will briefly discuss some scenarios beyond traditional database programming with SQL, in which strong types pay off.

Normalization and denormalization

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 an operation *compose* that denormalizes tables that are in third normal form.

```
compose fk pk t1@(Table' h1 m1 fds1) t2@(Table' h2 m2 fds2)
  = (Table' h m ((FD pk fk) .*. fds), Table' h2 m' fds2)
  where
    on = (λr → Record $ hZip pk $ lookupMany fk r)
    Table' h m fds = join' t1 t2 on
    ResultList _ ks = select True fk (Table h1 m1) (const True)
    m' = mapDeleteMany ks m2
```

In fact, the *compose* operation is a further restricted variant of *join'*. 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. This allows us to add a further functional dependency to the composed table, which expresses that the foreign keys from the first table will become functionally dependent on the primary keys from the second table.

Note also that we return a slimmed down copy of the second table. The copy keeps any rows that are not involved in the composition, to make denormalization data-preserving.

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.

Data cleaning and migration

The SQL language is insufficiently expressive for purposes of data cleaning or data migration, and several extensions to the language are provided by specific vendors. Recently, a domain-specific language (DSL) for data migration, called DTL, has been defined and implemented in the Data Fusion tool [5]. A key innovation of DTL is a *mapper* operation that allows the definition of one-to-many data transformations. On the basis of the reconstruction of SQL of Section 3 we have implemented a type-safe *mapper* (included in the source distribution) in a handful of lines. This demonstrates that Haskell can be used to design and prototype DSLs for database cleaning and migration.

The code of *compose*, *decompose*, and *mapper* is included in the source distribution of this paper, together with further database transformation operations. The small set of operations that we defined so far can be extended to construct complete and highly expressive operator suites that cover specific scenarios such as migration, cleaning, and more. These operator suites can be useful by themselves, or can serve as prototypes for strongly typed DSLs.

6. Related work

We are not the first to provide types for relational databases, and type-level programming has other applications besides type checking SQL. We will briefly discuss related approaches.

Machiavelli

Ohori *et al.* extended an ML-like type system to include database programming operations such as join and projection [21]. The extension is necessary to provide types for labeled records, which are used to model databases. They demonstrate 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 [22, 4].

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. But, type-class bounded polymorphism is not a feature shared by all members of that family. In fact, Ohori *et al.* do not assume this feature. This explains why they must develop a dedicated type system and language, while we can stay inside an existing language and type system.

Haskell/DB and HLIST-based database connectivity

Leijen *et al*. present a general approach for embedding domain-specific compilers in Haskell, and its application to the implementation of a typeful SQL binding, called Haskell/DB [16]. They construct an embedded domain-specific language (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.

From 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 users from the unstructured strings and from low-level communication details.

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 uses a different model of extensible records [3], which is more restricted, but similar in spirit to the HLIST library of [14] that we rely on. Conversely, the authors of the HLIST library report on the application of their extensible records to database connectivity, which "adopt[s] concepts from Leijen [*et al*]'s embedding approach".

The most important difference between these approaches and ours is that their tables are stored externally, as is the purpose of database connectivity, while ours are stored internally, as motivated by our wish to model relational databases *inside* Haskell. Furthermore, the level of typing realized by the database connectivity approaches does not include information to distinguish keys from non-key attributes, nor functional dependencies. Permutation and padding of records to approximate SQL's handling of some arguments also seems to be unique to our approach. For the relational algebra sub-language of Haskell/DB, which includes restriction (filter), projection, product, and set operators, only syntactic well-formedness checks are offered. These operators are strongly typed in our approach.

Type-level programming and lightweight dependent types

McBride [18] and Hallgren [10] pioneered the use of Haskell's type system as static logic programming language. Apart from heterogeneous collections [14], the technique has been used for lightweight dependently typed programming [18], implicit configurations [15], variable-length argument lists, formatting [11], and more.

OOHaskell

Kiselyov *et al.* have developed a model of object-oriented programming inside Haskell [13], based on their HLIST library of extensible polymorphic records with first-class labels and subtyping [14]. The model includes all conventional object-oriented features and more advanced ones, such as flexible multiple inheritance, implicitly polymorphic classes, and many flavours of subtyping.

We have used the same basis (HLIST records) and the same techniques (type-level programming) for modeling a different paradigm, *viz.* relational database programming. Both models rely non-trivially 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.

Point-free relational algebra

Necco *et al.* have developed models of relational databases in Haskell and in Generic Haskell [19, 20]. 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 of a single, fixed type.

Apart from modeling relational algebra operators the authors provide a suite of calculation rules for database transformation.

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.

Two-level data transformation

Cunha *et al.* [7] 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 relies on *generalized* algebraic datatypes (GADT). In particular, a GADT is used to safely represent types at the term level. 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. However, the employment of type-level indexes to model table and attribute names in headers, rows, and databases goes beyond the maps-and-tuples representation, allowing a nominal, rather than purely structural treatment. On the other hand, our representation is limited to databases, while Cunha *et al.* also cover hierarchical data structures, involving e.g. sums, lists, recursion.

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, for data mappings and 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.

7. Concluding remarks

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

Future work

Ohori *et al.* model generalized relational databases and some features of object-oriented databases [21, 4]. It would be interesting to see if our approach can be generalized in these directions as well.

The approach of Cunha *et al.* 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 on the term level.

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.

Availability

The source distribution that supports this paper is available from the homepages of the authors, 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. CODDFISH lends itself as a sandbox for the design of typed languages for modeling, programming, and transforming relational databases.

References

[1] American National Standards Institute. *ANSI X3.135-1992: Database Language SQL.* 1992.

[2] C. Beeri, R. Fagin, and J. H. Howard. A complete axiomatization for functional and multivalued dependencies in database relations. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 47–61, 1977.

[3] B. Bringert and A. Höckersten. Student paper: HaskellDB improved. In *Proc. of 2004 ACM SIGPLAN workshop on Haskell*, pages 108–115. ACM Press, 2004.

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

[5] P. Carreira and H. Galhardas. Efficient development of data migration transformations. In G. Weikum et al., editors, *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 915–916. ACM, 2004.

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

[7] A. Cunha, J. N. Oliveira, and J. Visser. Type-safe two-level data transformation. In *Proc. Int. Symp. Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 284–299. Springer, 2006.

[8] C. J. Date. *An Introduction to Database Systems, 6th Edition.* Addison-Wesley, 1995.

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

[10] T. Hallgren. Fun with functional dependencies. In *Proc. of the Joint CS/CE Winter Meeting*, pages 135–145, 2001. Dep.t of Computing Science, Chalmers, Göteborg, Sweden.

[11] R. Hinze. Formatting: a class act. *J. Funct. Program.*, 13(5):935–944, 2003.

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

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

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

[15] O. Kiselyov and C. Shan. Functional pearl: implicit configurations–or, type classes reflect the values of types. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 33–44, New York, NY, USA, 2004. ACM Press.

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

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

[18] C. McBride. Faking it – simulating dependent types in haskell. *J. Funct. Program.*, 12(5):375–392, 2002.

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

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

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

[22] A. Ohori, P. Buneman, and V. Tannen. Database programming in Machiavelli - a polymorphic language with static type inference. In J. Clifford et al., editors, *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 46–57. ACM Press, 1989.

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

[24] P. Stuckey and M. Sulzmann. A theory of overloading. *ACM Trans. Program. Lang. Syst.*, 27(6):1216–1269, 2005.

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