Coddfish

Functional Pearl: Strong Types for Relational Databases

Alexandra Silva[1] Joost Visser[2]

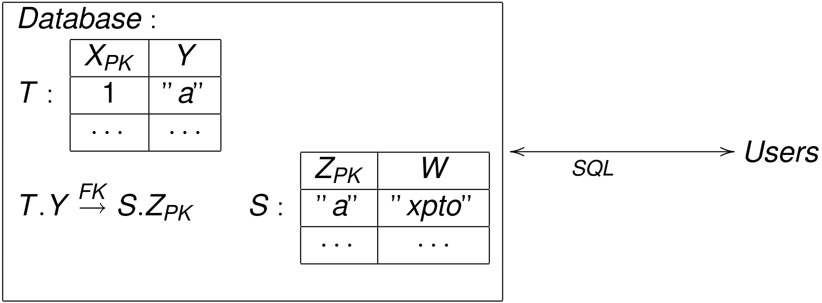
[1]CWI, The Netherlands

[2]Universidade do Minho, Portugal

Haskell Workshop, 2006

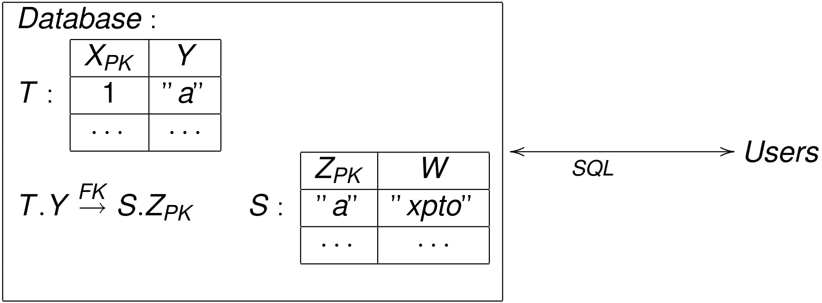
Outline

Motivation




```
insert into T values (2,"s")
insert into T values (3)
select * from T join S on T.Y=S.Z
select * from T join S on T.Y=S.W
```

Motivation




```
insert into T values (2,"s")
insert into T values (3)
select * from T join S on T.Y=S.Z
select * from T join S on T.Y=S.W
```

Motivation

- SQL is very flexible
- but... it could be more precise
  `select * from T join S on T.Y=S.Z`
  could be statically rejected because it mis-specifies the join condition.
- Haskell types look like a good way of making SQL more precise
- but we do not want to provide an SQL data binding (such as Haskell/DB)

Motivation

- SQL is very flexible
- but... it could be more precise
  `select * from T join S on T.Y=S.Z`
  could be statically rejected because it mis-specifies the join condition.
- Haskell types look like a good way of making SQL more precise
- but we do not want to provide an SQL data binding (such as Haskell/DB)

Motivation

- SQL is very flexible
- but... it could be more precise
  `select * from T join S on T.Y=S.Z`
  could be statically rejected because it mis-specifies the join condition.
- Haskell types look like a good way of making SQL more precise
- but we do not want to provide an SQL data binding (such as Haskell/DB)

What will we show?

We will show how to...

- ... capture key meta-data in the types of tables
- ... encode standard (type-safe) SQL operators
- ... capture functional dependency information on the type level and ensure normal forms
- ... transport meta-data information through the operations

Type-level Programming

Extensive use of type level programming and heterogeneous collections

Recall:

`class P a` **Type-level predicate**
`class R a b` **Type-level relation**
`class F a b c | a b -> c` **Type-level function**
`  where f ::  a -> b -> c` (**with value-level counterpart**)

See:
T. Hallgren. Fun with functional dependencies.
O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections.

Type-level Programming

Extensive use of type level programming and heterogeneous collections

Recall:

`class P a` **Type-level predicate**

`class R a b` **Type-level relation**

`class F a b c | a b -> c` **Type-level function**

`  where f ::  a -> b -> c` (**with value-level counterpart**)

See:

T. Hallgren. Fun with functional dependencies.

O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections.

Tables

A table is a set of tuples

```
data HList row => Table row = Table (Set row)
```

But:

- We miss schema information;
- Tables are in reality **mappings** from key to non-key attributes.

```
data HeaderFor h k v => Table h k v = Table h (Map k v)
```

Tables

A table is a set of tuples

```
data HList row => Table row = Table (Set row)
```

But:

- We miss schema information;
- Tables are in reality **mappings** from key to non-key attributes.

```
data HeaderFor h k v => Table h k v = Table h (Map k v)
```

Tables

A table is a set of tuples

```
data HList row => Table row = Table (Set row)
```

But:

- We miss schema information;
- Tables are in reality **mappings** from key to non-key attributes.

```
data HeaderFor h k v => Table h k v = Table h (Map k v)
```

The constraint `HeaderFor`

A valid header should not have repeated attributes.
Captured in type level predicate:

```
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
```

The fd `h -> k v` reflects the fact that the types for the key and non-key values on the table are **uniquely** determined by the header.

The constraint `HeaderFor`

A valid header should not have repeated attributes.
Captured in type level predicate:

```
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
```

The fd `h -> k v` reflects the fact that the types for the key and non-key values on the table are **uniquely** determined by the header.

The constraint `HeaderFor`

A valid header should not have repeated attributes.
Captured in type level predicate:

```
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
```

The fd `h -> k v` reflects the fact that the types for the key and non-key values on the table are **uniquely** determined by the header.

Attributes

How did we model attributes?

Phantom types working

```
data Attribute t name
attr = undefined :: Attribute t name
```

Let us see some examples:

```
data ID; atID=attr :: Attribute Int (People ID)
data Name; atName = attr :: Attribute String (People Name)
data People a; people = undefined :: People ()
```

Attributes

How did we model attributes?

Phantom types working

```
data Attribute t name
attr = undefined ::  Attribute t name
```

Let us see some examples:

```
data ID; atID=attr ::  Attribute Int (People ID)
data Name; atName = attr ::  Attribute String (People Name)
data People a; people = undefined ::  People ()
```

people:

ID	Name	Age	City
12	"Ralf"	23	"Seattle"
67	"Oleg"	17	"Seattle"
50	"Dorothy"	42	"Oz"

Ok, we now have ingredients to construct our first table:

```
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
```

people:

ID	Name	Age	City
12	"Ralf"	23	"Seattle"
67	"Oleg"	17	"Seattle"
50	"Dorothy"	42	"Oz"

Ok, we now have ingredients to construct our first table:

```
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
```

What about default and null attributes?

Easy:

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

atCountry ::  AttrDef String (Cities Country)
atCountry = Default "Afghanistan"
```

We have also modelled default system attributes.

What about default and null attributes?

Easy:

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

atCountry ::  AttrDef String (Cities Country)
atCountry = Default "Afghanistan"
```

We have also modelled default system attributes.

What about default and null attributes?

Easy:

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

atCountry ::  AttrDef String (Cities Country)
atCountry = Default "Afghanistan"
```

We have also modelled default system attributes.

Foreign keys

Imagine we have the following table:

cities:

City	Country
Braga	Portugal

How do we model a foreign key from the previous table to this one?

```
data FK fk t pk

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

Foreign keys

Imagine we have the following table:

cities:

City	Country
Braga	Portugal

How do we model a foreign key from the previous table to this one?

```
data FK fk t pk

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

Foreign keys

Imagine we have the following table:

cities:

City	Country
Braga	Portugal

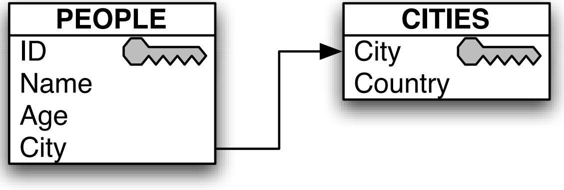
How do we model a foreign key from the previous table to this one?

```
data FK fk t pk

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

RDB = Tables + Foreign key information

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



The `join` operation

Typical SQL join:

```
select *
from People join Cities
on People.City = Cities.City
```

The `join` operation

In Haskell:

- First we define a join for maps

$$(k \rightharpoonup v) \bowtie (k' \rightharpoonup v') = (k \rightharpoonup vk'v')$$

```
joinM ::  ...  =>
(k -> v -> k') -> Map k v -> Map k' v' -> Map k vkv'
```

- Then we lift to tables

```
join :: ( ... , LookupMany a' r' k' ) => Table
(a,b) k v -> Table (a',b') k' v' -> (Record r -> r')
-> Table (a,bab') k vkv'
```

- Notice how we do not allow the key of the second table to be underspecified

The `join` operation

In Haskell:

- First we define a join for maps

$$(k \rightharpoonup v) \bowtie (k' \rightharpoonup v') = (k \rightharpoonup vk'v')$$

```
joinM ::  ...  =>
(k -> v -> k') -> Map k v -> Map k' v' -> Map k vkv'
```

- Then we lift to tables

```
join ::  ( ...  , LookupMany a' r' k' ) => Table
(a,b) k v -> Table (a',b') k' v' -> (Record r -> r')
-> Table (a,bab') k vkv'
```

- Notice how we do not allow the key of the second table to be underspecified

The `join` operation

In Haskell:

- First we define a join for maps

$$(k \rightharpoonup v) \bowtie (k' \rightharpoonup v') = (k \rightharpoonup vk'v')$$

```
joinM ::  ...  =>
(k -> v -> k') -> Map k v -> Map k' v' -> Map k vkv'
```

- Then we lift to tables

```
join ::  ( ...  , LookupMany a' r' k' ) => Table
(a,b) k v -> Table (a',b') k' v' -> (Record r -> r')
-> Table (a,bab') k vkv'
```

- Notice how we do not allow the key of the second table to be underspecified

What about functional dependencies?

Why FD's?

- Database normalization and de-normalization, for instance, are driven by functional dependencies
- Kernel of the classical relational database design theory (Codd, Maier, . . .)

See:
C. Beeri, R. Fagin, and J. H. Howard. A complete axiomatization for functional and multivalued dependencies in database relations. SIGMOD, 1977.

What about functional dependencies?

Why FD's?

- Database normalization and de-normalization, for instance, are driven by functional dependencies
- Kernel of the classical relational database design theory (Codd, Maier, . . .)

See:
C. Beeri, R. Fagin, and J. H. Howard. A complete axiomatization for functional and multivalued dependencies in database relations. SIGMOD, 1977.

What about functional dependencies?

In Haskell:

```
data FD X Y = FD X Y
```

What about functional dependencies?

In Haskell:

```
data FD X Y = FD X Y
```

Adding them to tables:

```
data TableWithFD fds h k v => Table' h (Map k v) fds
```

- `TableWithFD fds h k v` ensures `HeaderFor h k v` and that `fds` does not refer to attributtes not present in the header.

What can we do with functional dependencies?

- We can improve the design of a database
  - Given a header and the corresponding set of fds we can determine the possible table keys
  - Given a database we can check several normal forms (and thus avoid data redundancy and update anomalies)
- We can transport (and transform) them in the operations **(cool!)**

What can we do with functional dependencies?

- We can improve the design of a database
  - Given a header and the corresponding set of fds we can determine the possible table keys
  - Given a database we can check several normal forms (and thus avoid data redundancy and update anomalies)
- We can transport (and transform) them in the operations **(cool!)**

What can we do with functional dependencies?

- We can improve the design of a database
  - Given a header and the corresponding set of fds we can determine the possible table keys
  - Given a database we can check several normal forms (and thus avoid data redundancy and update anomalies)
- We can transport (and transform) them in the operations (cool!)

What can we do with functional dependencies?

- We can improve the design of a database
  - Given a header and the corresponding set of fds we can determine the possible table keys
  - Given a database we can check several normal forms (and thus avoid data redundancy and update anomalies)
- We can transport (and transform) them in the operations **(cool!)**

Transport through project

X	Y	Z	W
x	y	z	w
...	...	...	...

$$, \; X \to YZW, \; Z \to W$$

↓ *project XYW*

X	Y	W
x	y	z
...	...	...

$$, \; X \to YW$$

Conclusions

What I did not show

- Default system attributes
- Lifting of table operations to databases (*e.g.* selectInto)
- Database transformation operations (normalization and denormalization)

Conclusions

- Haskell can be used to assign more precise types to SQL operations
- The join operator on tables guarantees that in the *on* clause a value is assigned to all keys in the second table
- We have defined a new level of operations that carry functional dependency information, automatically infered by the type-checker.

Haskell can be used for the design of typed languages for modeling, programming, and transforming relational databases.

Conclusions

- Haskell can be used to assign more precise types to SQL operations
- The join operator on tables guarantees that in the *on* clause a value is assigned to all keys in the second table
- We have defined a new level of operations that carry functional dependency information, automatically infered by the type-checker.

Haskell can be used for the design of typed languages for modeling, programming, and transforming relational databases.

Future work

- Use our model for spreadsheet transformation
- We have shown how we can transport **fd** information from argument to result tables: develop a formal calculus to automatically compute this information for further operations