Coddfish
Functional Pearl: Strong Types for Relational Databases

Alexandra Silva¹ Joost Visser²

¹CWI, The Netherlands
²Universidade do Minho, Portugal

Haskell Workshop, 2006
Motivation

Tables and Operations

Functional Dependencies

Conclusions and Future work
Motivation

**Database**:

<table>
<thead>
<tr>
<th>X_{PK}</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;a&quot;</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

\[ T : T.Y \xrightarrow{FK} S.Z_{PK} \]

\[ S : \begin{array}{|c|c|} \hline Z_{PK} & W \hline "a" & "xpto" \hline \ldots & \ldots \hline \end{array} \]

\[ T \rightarrow^{Y} S \rightarrow^{Z_{PK}} T \rightarrow^{Y} S \rightarrow_{W} Users \]

\[ SQL \]

- `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

**Database:**

\[
\begin{array}{|c|c|}
\hline
X_{PK} & Y \\
\hline
1 & "a" \\
\vdots & \vdots \\
\hline
\end{array}
\]

\[
T : \xrightarrow{FK} S.Z_{PK}
\]

\[
\begin{array}{|c|c|}
\hline
Z_{PK} & W \\
\hline
"a" & "xpto" \\
\vdots & \vdots \\
\hline
\end{array}
\]

\[
T \cdot Y \rightarrow S \cdot Z_{PK}
\]

**SQL**

- 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

**Users**
Motivation

- SQL is very flexible
- but... it could be more precise
  
  ```sql
  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
  ```sql
  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:**

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>class P a</code></td>
<td>Type-level predicate</td>
</tr>
<tr>
<td><code>class R a b</code></td>
<td>Type-level relation</td>
</tr>
<tr>
<td><code>class F a b c</code></td>
<td>Type-level function</td>
</tr>
<tr>
<td><code>where f :: a -&gt; b -&gt; c</code></td>
<td>(with value-level counterpart)</td>
</tr>
</tbody>
</table>

See:

T. Hallgren. Fun with functional dependencies.
O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections.
Extensive use of type level programming and heterogeneous collections

**Recall:**

<table>
<thead>
<tr>
<th>Class</th>
<th>Type-level Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>class P a</code></td>
<td>Type-level predicate</td>
</tr>
<tr>
<td><code>class R a b</code></td>
<td>Type-level relation</td>
</tr>
<tr>
<td><code>class F a b c</code></td>
<td>Type-level function</td>
</tr>
<tr>
<td><code>where f :: a -&gt; b -&gt; c</code></td>
<td>with value-level counterpart</td>
</tr>
</tbody>
</table>

See:
A *table is a set of tuples*

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

But:

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

```haskell
data HeaderFor h k v => Table h k v = Table h (Map k v)
```
A table is a set of tuples

\begin{verbatim}
data HList row => Table row = Table (Set row)
\end{verbatim}

But:

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

\begin{verbatim}
data HeaderFor h k v => Table h k v = Table h (Map k v)
\end{verbatim}
A table is a set of tuples

\[
\text{data HList row } \rightarrow \text{ Table row } = \text{ Table (Set row)}
\]

But:

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

\[
\text{data HeaderFor h k v } \rightarrow \text{ Table h k v } = \text{ Table h (Map k v)}
\]
The constraint \texttt{HeaderFor}

A valid header should not have repeated attributes.

Captured in type level predicate:

\begin{verbatim}
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
\end{verbatim}

The \texttt{fd} \texttt{h -> k v} reflects the fact that the types for the key and non-key values on the table are \textit{uniquely} determined by the header.
The constraint **HeaderFor**

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

```haskell
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 $h \rightarrow k v$ 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:

```haskell
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

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

Let us see some examples:

```haskell
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**

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

Let us see some examples:

```haskell
data ID; atID=attr :: Attribute Int (People ID)
data Name; atName = attr :: Attribute String (People Name)
data People a; people = undefined :: People ()
```
<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Age</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>&quot;Ralf&quot;</td>
<td>23</td>
<td>&quot;Seattle&quot;</td>
</tr>
<tr>
<td>67</td>
<td>&quot;Oleg&quot;</td>
<td>17</td>
<td>&quot;Seattle&quot;</td>
</tr>
<tr>
<td>50</td>
<td>&quot;Dorothy&quot;</td>
<td>42</td>
<td>&quot;Oz&quot;</td>
</tr>
</tbody>
</table>

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
```
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:**

```haskell
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:**

```haskell
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:

<table>
<thead>
<tr>
<th>City</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Braga</td>
<td>Portugal</td>
</tr>
</tbody>
</table>

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

```haskell
data FK fk t pk

myFK = FK (atCity .*. HNil)
    cities
    (atCity’ .*. HNil)
```
Imagine we have the following table:

<table>
<thead>
<tr>
<th>City</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Braga</td>
<td>Portugal</td>
</tr>
</tbody>
</table>

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)
```
Imagine we have the following table:

<table>
<thead>
<tr>
<th>City</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Braga</td>
<td>Portugal</td>
</tr>
</tbody>
</table>

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

```haskell
data FK fk t pk

myFK = FK (atCity .*. HNil) cities (atCity’ .*. HNil)
```
RDB = Tables + Foreign key information

myRDB = Record $
\text{cities} \ = \ (\text{yourTable, HNil}) \ . \ * . \\
\text{people} \ = \ (\text{myTable, myFK . * . HNil}) \ . \ * . \ \text{HNil}$
The join operation

Typical SQL join:

```sql
select *
from People join Cities
on People.City = Cities.City
```
The **join** operation

In Haskell:

- First we define a join for maps

\[(k \rightarrow v) \Join (k' \rightarrow v') = (k \rightarrow vk'v')\]

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

- Then we lift to tables

```haskell
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

\[(k \rightarrow v) \Join (k' \rightarrow v') = (k \rightarrow vk'v')\]

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

- Then we lift to tables

```haskell
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 \textit{join} operation

In Haskell:

- First we define a join for maps
  \[
  (k \mapsto v) \Join (k' \mapsto v') = (k \mapsto vk'v')
  \]

\[
\text{joinM :: ... =>} \\
(k \to v \to k') \to \text{Map } k \, v \to \text{Map } k' \, v' \to \text{Map } k \, vkv'
\]

- Then we lift to tables

\[
\text{join :: ( ... , LookupMany a' r' k' ) => Table} \\
(a,b) \, k \, v \to \text{Table} \, (a',b') \, k' \, v' \to (\text{Record } r \to r') \\
\to \text{Table} \, (a,bab') \, k \, vkv'
\]

- 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:
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:
What about functional dependencies?

In Haskell:

```haskell
data FD X Y = FD X Y
```
What about functional dependencies?

In Haskell:

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

Adding them to tables:

```haskell
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 attributes 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

\[
\begin{pmatrix}
X & Y & Z & W \\
x & y & z & w \\
\cdots & \cdots & \cdots & \cdots \\
\end{pmatrix},
\begin{aligned}
X \rightarrow YZW \\
Z \rightarrow W
\end{aligned}
\]

\[
\begin{pmatrix}
X & Y & W \\
x & y & z \\
\cdots & \cdots & \cdots \\
\end{pmatrix},
\begin{aligned}
X \rightarrow YW
\end{aligned}
\]
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 inferred 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 inferred 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 $\text{fd}$ information from argument to result tables: develop a formal calculus to automatically compute this information for further operations