Strong Types for Relational Databases

CoddFish

Alexandra Silva Joost Visser

Departamento de Informática
Universidade do Minho

PURe Cafe, December 2005

- A database schema specifies the well-formedness of a relational database
- Operations on a database should preserve its well-formedness.

- Type-level programming
- HList library
- A typeful reconstruction of statements and clauses of the SQL language
- Functional dependencies and normal forms

Single parameter type classes $\rightsquigarrow$ Predicates on types

```
class Show a ...
instance Show Char ...
```

$\longrightarrow$ Char is showable

Single parameter type classes $\rightsquigarrow$ Predicates on types

```
class Show a ...
instance Show Char ...
```

$\longrightarrow$ Char is showable

Multiple parameter type classes $\rightsquigarrow$ Relations between types

$$
\begin{array}{l}
\textbf{class}\ \mathit{Convert}\ a\ b\ |\ a \rightarrow b\ \textbf{where} \\
\quad \mathit{convert} :: a \rightarrow b \\
\textbf{instance}\ \mathit{Show}\ a \Rightarrow \mathit{Convert}\ a\ \mathit{String}\ \textbf{where} \\
\quad \mathit{convert} = \mathit{show} \\
\textbf{instance}\ \mathit{Convert}\ \mathit{String}\ \mathit{String}\ \textbf{where} \\
\quad \mathit{convert} = \mathit{id}
\end{array}
$$

More important

The Convert class is not just a relation, but because of the functional dependency it is a **function**

Type-checker is used for computing

Multiple parameter type classes $\rightsquigarrow$ Relations between types

$$
\begin{array}{l}
\textbf{class}\ \mathit{Convert}\ a\ b \mid a \rightarrow b\ \textbf{where} \\
\quad \mathit{convert} :: a \rightarrow b \\
\textbf{instance}\ \mathit{Show}\ a \Rightarrow \mathit{Convert}\ a\ \mathit{String}\ \textbf{where} \\
\quad \mathit{convert} = \mathit{show} \\
\textbf{instance}\ \mathit{Convert}\ \mathit{String}\ \mathit{String}\ \textbf{where} \\
\quad \mathit{convert} = \mathit{id}
\end{array}
$$

More important

The Convert class is not just a relation, but because of the functional dependency it is a **function**

Type-checker is used for computing

Multiple parameter type classes $\rightsquigarrow$ Relations between types

$$
\begin{array}{l}
\textbf{class}\ \mathit{Convert}\ a\ b \mid a \rightarrow b\ \textbf{where} \\
\quad \mathit{convert} :: a \rightarrow b \\
\textbf{instance}\ \mathit{Show}\ a \Rightarrow \mathit{Convert}\ a\ \mathit{String}\ \textbf{where} \\
\quad \mathit{convert} = \mathit{show} \\
\textbf{instance}\ \mathit{Convert}\ \mathit{String}\ \mathit{String}\ \textbf{where} \\
\quad \mathit{convert} = \mathit{id}
\end{array}
$$

$\boxed{\text{More important}}$

The Convert class is not just a relation, but because of the functional dependency it is a **function**
Type-checker is used for computing

$$\textbf{data}\ \mathit{Zero};\ \mathit{zero} = \bot :: \mathit{Zero}$$
$$\textbf{data}\ \mathit{Succ}\ n;\ \mathit{succ} = \bot :: n \to \mathit{Succ}\ n$$

$$\textbf{class}\ \mathit{Nat}\ n$$
$$\textbf{instance}\ \mathit{Nat}\ \mathit{Zero}$$
$$\textbf{instance}\ \mathit{Nat}\ n \Rightarrow \mathit{Nat}\ (\mathit{Succ}\ n)$$

How to define sum on Nat?

$$\textbf{class}\ \mathit{Add}\ a\ b\ c \mid a\ b \to c\ \textbf{where}\ \mathit{add} :: a \to b \to c$$
$$\textbf{instance}\ \mathit{Add}\ \mathit{Zero}\ b\ b\ \textbf{where}\ \mathit{add}\ a\ b = b$$
$$\textbf{instance}\ (\mathit{Add}\ a\ b\ c) \Rightarrow \mathit{Add}\ (\mathit{Succ}\ a)\ b\ (\mathit{Succ}\ c)\ \textbf{where}$$
$$\quad \mathit{add}\ a\ b = \mathit{succ}\ (\mathit{add}\ (\mathit{pred}\ a)\ b)$$

$$\textbf{data}\ Zero;\ zero = \perp :: Zero$$
$$\textbf{data}\ Succ\ n;\ succ = \perp :: n \rightarrow Succ\ n$$

$$\textbf{class}\ Nat\ n$$
$$\textbf{instance}\ Nat\ Zero$$
$$\textbf{instance}\ Nat\ n \Rightarrow Nat\ (Succ\ n)$$

How to define sum on Nat?

$$\textbf{class}\ Add\ a\ b\ c \mid a\ b \rightarrow c\ \textbf{where}\ add :: a \rightarrow b \rightarrow c$$
$$\textbf{instance}\ Add\ Zero\ b\ b\ \textbf{where}\ add\ a\ b = b$$
$$\textbf{instance}\ (Add\ a\ b\ c) \Rightarrow Add\ (Succ\ a)\ b\ (Succ\ c)\ \textbf{where}$$
$$add\ a\ b = succ\ (add\ (pred\ a)\ b)$$

$$\textbf{data}\ \mathit{Zero};\ \mathit{zero} = \bot :: \mathit{Zero}$$
$$\textbf{data}\ \mathit{Succ}\ n;\ \mathit{succ} = \bot :: n \rightarrow \mathit{Succ}\ n$$

$$\textbf{class}\ \mathit{Nat}\ n$$
$$\textbf{instance}\ \mathit{Nat}\ \mathit{Zero}$$
$$\textbf{instance}\ \mathit{Nat}\ n \Rightarrow \mathit{Nat}\ (\mathit{Succ}\ n)$$

$\boxed{\text{How to define sum on Nat?}}$

$$\textbf{class}\ \mathit{Add}\ a\ b\ c \mid a\ b \rightarrow c\ \textbf{where}\ \mathit{add} :: a \rightarrow b \rightarrow c$$
$$\textbf{instance}\ \mathit{Add}\ \mathit{Zero}\ b\ b\ \textbf{where}\ \mathit{add}\ a\ b = b$$
$$\textbf{instance}\ (\mathit{Add}\ a\ b\ c) \Rightarrow \mathit{Add}\ (\mathit{Succ}\ a)\ b\ (\mathit{Succ}\ c)\ \textbf{where}$$
$$\quad \mathit{add}\ a\ b = \mathit{succ}\ (\mathit{add}\ (\mathit{pred}\ a)\ b)$$

HList library (Lämmel *et al.*)

data $HNil = HNil$
data $HCons\ e\ l = HCons\ e\ l$

class $HList\ l$
instance $HList\ HNil$
instance $HList\ l \Rightarrow HList\ (HCons\ e\ l)$

HList library (Lämmel *et al.*)

$\boxed{\text{More convenient notation}}$

$$\textbf{type}\ (:\!*\!:)\ e\ l = HCons\ e\ l$$
$$e\ .\!*\!.\ l = HCons\ e\ l$$

$$l\ .\!=\!.\ v = (l, v)$$
$$l\ .!.\ v = hLookupByLabel\ l\ v$$

HList library (Lämmel *et al.*)

Example

$$myHList :: Int :*: Bool :*: String :*: HNil$$
$$myHList = (1 :: Int) .*. True .*. \texttt{"foo"} .*. HNil$$

$$mySugar = Record\ (zero .=. \texttt{"foo"} .*. one .=. True .*. HNil)$$

HList API

$$
\begin{array}{l}
\textbf{class}\ HAppend\ l\ l'\ l''\ |\ l\ l' \rightarrow l''\ \textbf{where} \\
\quad hAppend :: l \rightarrow l' \rightarrow l'' \\
\textbf{class}\ HZip\ x\ y\ l\ |\ x\ y \rightarrow l, l \rightarrow x\ y\ \textbf{where} \\
\quad hZip :: x \rightarrow y \rightarrow l \\
\quad hUnzip :: l \rightarrow (x, y) \\
\textbf{class}\ HasField\ l\ r\ v\ |\ l\ r \rightarrow v \\
\quad \textbf{where}\ hLookupByLabel :: l \rightarrow r \rightarrow v
\end{array}
$$

Representation of databases

- Table $\equiv$ set of arbitrary-length tuples
- Database $\equiv$ heterogeneous list of tables

$$\textbf{data}\ \mathit{HList\ row} \Rightarrow \mathit{Table\ row} = \mathit{Table}\ (\mathit{Set\ row})$$
$$\textbf{data}\ \mathit{TableList\ t} \Rightarrow \mathit{RDB\ t} = \mathit{RDB\ t}$$

$$\textbf{class}\ \mathit{TableList\ t}$$
$$\textbf{instance}\ \mathit{TableList\ HNil}$$
$$\textbf{instance}\ (\mathit{HList\ v}, \mathit{TableList\ t}) \Rightarrow \mathit{TableList}\ (\mathit{HCons}\ (\mathit{Table\ v})\ t)$$

Drawbacks lead to new approach

- Schema information is not represented
- Operations may not respect the schema (!)
- No distinction between key attributes and non-key attributes.

$$\textbf{data}\ \mathit{HeaderFor}\ h\ k\ v \Rightarrow \mathit{Table}\ h\ k\ v = \mathit{Table}\ h\ (\mathit{Map}\ k\ v)$$

$$\begin{aligned}
&\textbf{class}\ \mathit{HeaderFor}\ h\ k\ v \mid h \rightarrow k\ v\\
&\textbf{instance}\ (\\
&\quad \mathit{AttributesFor}\ a\ k, \mathit{AttributesFor}\ b\ v,\\
&\quad \mathit{HAppend}\ a\ b\ ab, \mathit{NoRepeats}\ ab, \mathit{Ord}\ k\\
&\ ) \Rightarrow \mathit{HeaderFor}\ (a, b)\ k\ v
\end{aligned}$$

Drawbacks lead to new approach

- Schema information is not represented
- Operations may not respect the schema (!)
- No distinction between key attributes and non-key attributes.

data $\mathit{HeaderFor}\ h\ k\ v \Rightarrow \mathit{Table}\ h\ k\ v = \mathit{Table}\ h\ (\mathit{Map}\ k\ v)$

class $\mathit{HeaderFor}\ h\ k\ v \mid h \rightarrow k\ v$
instance (
$\quad \mathit{AttributesFor}\ a\ k, \mathit{AttributesFor}\ b\ v,$
$\quad \mathit{HAppend}\ a\ b\ ab, \mathit{NoRepeats}\ ab, \mathit{Ord}\ k$
$) \Rightarrow \mathit{HeaderFor}\ (a, b)\ k\ v$

Attributes

data *Attribute t nr*

$$attribute = \perp :: Attribute\ t\ nr$$

class *AttributesFor a v* $\mid a \rightarrow v$
instance *AttributesFor HNil HNil*
instance *AttributesFor a v*
$\Rightarrow AttributesFor\ (HCons\ (Attribute\ t\ nr)\ a)\ (HCons\ t\ v)$

$$\mathbf{data}\ \mathit{ID}; \mathit{atID} = \mathit{attribute} :: \mathit{Attribute}\ \mathit{Int}\ \mathit{ID}$$
$$\mathbf{data}\ \mathit{NAME}; \mathit{atName} = \mathit{attribute} :: \mathit{Attribute}\ \mathit{String}\ \mathit{NAME}$$
$$\mathbf{data}\ \mathit{CITY}; \mathit{atCity} = \mathit{attribute} :: \mathit{Attribute}\ \mathit{String}\ \mathit{CITY}$$

Example

$$\begin{array}{l}
\mathit{myHeader} = (\mathit{atID} \mathbin{.*.} \mathit{HNil}, \mathit{atName} \mathbin{.*.} \mathit{atAge} \mathbin{.*.} \mathit{atCity} \mathbin{.*.} \mathit{HNil}) \\
\mathit{myTable} = \mathit{Table}\ \mathit{myHeader}\ (\\
\quad \mathit{insert}\ (123 \mathbin{.*.} \mathit{HNil})\ (\texttt{"Ralf"} \mathbin{.*.} 23 \mathbin{.*.} \texttt{"Seattle"} \mathbin{.*.} \mathit{HNil})\$ \\
\quad \mathit{insert}\ (678 \mathbin{.*.} \mathit{HNil})\ (\texttt{"Oleg"} \mathbin{.*.} 17 \mathbin{.*.} \texttt{"Seattle"} \mathbin{.*.} \mathit{HNil})\$ \\
\quad \mathit{insert}\ (504 \mathbin{.*.} \mathit{HNil})\ (\texttt{"Dorothy"} \mathbin{.*.} 42 \mathbin{.*.} \texttt{"Oz"} \mathbin{.*.} \mathit{HNil})\$ \\
\quad \mathit{Map.empty})
\end{array}$$

The Join Operator

$$
\begin{array}{ccc}
K \rightharpoonup V & \bowtie & K' \rightharpoonup V' \\
 & \downarrow & \\
 & K \rightharpoonup VK'V' &
\end{array}
$$

Extrapolating to Table

$$
\begin{aligned}
& join :: (\\
& \qquad \ldots \\
& \quad) \Rightarrow Table\ (a, b)\ k\ v \rightarrow Table\ (a', b')\ k'\ v' \rightarrow (r \rightarrow r') \\
& \quad \rightarrow Table\ (a, bab')\ k\ vkv' \\
& join\ (Table\ h@(a, b)\ m)\ (Table\ (a', b')\ m')\ on = Table\ h''\ m'' \\
& \quad \mathbf{where} \\
& \qquad h'' = (a, hAppend\ b\ (hAppend\ a'\ b')) \\
& \qquad m'' = joinM\ (\lambda k\ v \rightarrow lookupMany\ a'\ (on\ \$\ row\ h\ k\ v))\ m\ m'
\end{aligned}
$$

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

$$\begin{array}{l}
seniorAmericans \\
\quad = select\ False\ (atName \mathbin{.{*}.} atCountry \mathbin{.{*}.} HNil) \\
\qquad ((myTable \text{ `}join\text{` } yourTable \\
\qquad\quad (\lambda r \rightarrow ((atCityID \mathbin{.{=}.} (r \mathbin{.!.} atCity)) \mathbin{.{*}.} HNil))) \\
\qquad\quad \mathbin{.{*}.} HNil) \\
\qquad (\lambda r \rightarrow (r \mathbin{.!.} atAge) > 65 \wedge (r \mathbin{.!.} atCountry) \equiv \texttt{"USA"})
\end{array}$$

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, . . .)

What are FD's?

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

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

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

$\boxed{\text{example}}$

Let H be a header for a relation and F the set of funcional dependencies associated. 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.

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

$$
\begin{array}{l}
\textbf{class}\ \mathit{IsKey}\ x\ h\ \mathit{fds}\ b \mid x\ h\ \mathit{fds} \rightarrow b \\
\textbf{instance}\ (\\
\quad \mathit{Closure}\ h\ x\ \mathit{fds}\ \mathit{cl}, \mathit{Minimal}\ x\ h\ \mathit{fds}\ b'', \\
\quad \mathit{ContainedEq}\ h\ \mathit{cl}\ b', \mathit{HAnd}\ b'\ b''\ b \\
\ ) \Rightarrow \mathit{IsKey}\ x\ h\ \mathit{fds}\ b
\end{array}
$$

Why are NFs important?

- Avoid data redundancy
- Avoid update anomalies

In the libraries we have 1^{st}, 2^{nd}, 3^{rd} and Boyce-Codd NFs

example

Why are NFs important?

- Avoid data redundancy
- Avoid update anomalies

In the libraries we have 1^{st}, 2^{nd}, 3^{rd} and Boyce-Codd NFs

example

Why are NFs important?

- Avoid data redundancy
- Avoid update anomalies

In the libraries we have 1^{st}, 2^{nd}, 3^{rd} and Boyce-Codd NFs

example

Why are NFs important?

- Avoid data redundancy
- Avoid update anomalies

In the libraries we have 1^{st}, 2^{nd}, 3^{rd} and Boyce-Codd NFs

example

Why are NFs important?

- Avoid data redundancy
- Avoid update anomalies

In the libraries we have 1^{st}, 2^{nd}, 3^{rd} and Boyce-Codd NFs

example

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.

$\therefore$ The only non-trivial dependencies are those in which a key determines one or more other attributes.

$\boxed{\text{No attribute in } H \text{ is transitively dependent upon any key}}$

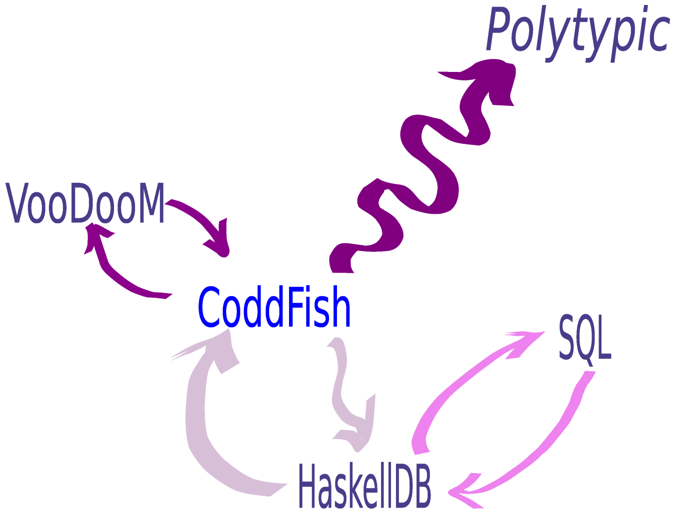
$\boxed{\text{Single FD}}$

class *BoyceCoddNFAtomic check h x fds b* $|$ *check h x fds* $\rightarrow$ *b*
instance *BoyceCoddNFAtomic HFalse h x fds HTrue*
instance *IsSuperKey x h fds b*
$\Rightarrow$ *BoyceCoddNFAtomic HTrue h x fds b*

Set of FDs

$$
\begin{array}{l}
\textbf{class } \mathit{BoyceCoddNF}\ h\ \mathit{fds}\ b \mid h\ \mathit{fds} \rightarrow b \\
\textbf{instance } \mathit{BoyceCoddNF}\ h\ \mathit{HNil}\ \mathit{HTrue} \\
\textbf{instance } \mathit{BoyceCoddNF}'\ h\ (\mathit{HCons}\ e\ l)\ (\mathit{HCons}\ e\ l)\ b \\
\qquad\qquad \Rightarrow \mathit{BoyceCoddNF}\ h\ (\mathit{HCons}\ e\ l)\ b
\end{array}
$$

$$
\begin{array}{l}
\textbf{class } \mathit{BoyceCoddNF}'\ h\ \mathit{fds}\ \mathit{allfds}\ b \mid h\ \mathit{fds}\ \mathit{allfds} \rightarrow b \\
\textbf{instance } \mathit{BoyceCoddNF}'\ h\ \mathit{HNil}\ \mathit{fds}\ \mathit{HTrue} \\
\textbf{instance } (\\
\qquad \mathit{HMember}\ y\ x\ \mathit{bb}, \mathit{Not}\ \mathit{bb}\ \mathit{bYnotinX}, \\
\qquad \mathit{BoyceCoddNFAtomic}\ \mathit{bYnotinX}\ h\ x\ \mathit{fds}\ b', \\
\qquad \mathit{BoyceCoddNF}'\ h\ \mathit{fds}'\ \mathit{fds}\ b'', \mathit{HAnd}\ b'\ b''\ b \\
\quad) \Rightarrow \mathit{BoyceCoddNF}'\ h\ (\mathit{HCons}\ (x, \mathit{HCons}\ y\ \mathit{HNil})\ \mathit{fds}')\ \mathit{fds}\ b
\end{array}
$$


Polytypic
VooDooM
CoddFish
SQL
HaskellDB