From Spreadsheets to Relational Databases and Back

Jácome Cunha  João Saraiva  Joost Visser

Departamento de Informática Universidade do Minho Portugal
SIG & CWI The Netherlands

PEPM’09, January, 19-20
Each row represents a renting transaction

It includes information about owners, clients, properties

It also contains dates and prices (formulas)

Row 3 “says” that john has rented a property owned by tony, with address 5 Novar Dr., price per day 70, between dates 1/7/00 and 8/31/01 and paid 25550 for it
This unstructured model is valid and serves its purpose

But, it contains data redundancy

For example, the rent per day is repeated several times
As a result, updates can cause data inconsistency.

For example, updating the renting value of property \textit{pg4} must be performed in several places.
Deleting rows can be problematic, too.

For example, deleting the row 5 will remove all the information about property *pg36*.
A normalized (3NF) data representation

- No redundancy
- No update/delete problems
A normalized (3NF) data representation

- No redundancy
- No update/delete problems: the price per day occurs only once
- Tables can be stored in different sheets
We can use information computed during normalization to produce a refactored spreadsheet that helps users to introduce correct data.

Spreadsheet warns the user when errors are introduced.
Infer functional dependencies (Fun)

Normalize (3NF) those functional dependencies (synthesize)

Create a Relational Database (RDB) schema (3NF)

Use data refinements to perform the data migration (to and from)

HaExcel combines all these steps creating a complete system
A *Functional Dependency* (FD) denoted $A \rightarrow B$ means that an element of $A$ is uniquely associated with an element of $B$.

The Fun algorithm (Novelli et al.) calculates the FDs from data.

For our example, the Haskell implementation returns:

- $\text{ownerNo} \rightarrow \text{oName}$
- $\text{totalDays} \rightarrow \text{clientNo}, \text{cName}$
- $\text{propertyNo} \rightarrow \text{pAddress}, \text{rentPerDay}, \text{ownerNo}, \text{oName}$
- ...

Formulas also induce FDs:

$C_0 = f(X_0, \ldots, X_n)$ induces $X_0, \ldots, X_n \rightarrow C_0$
Each tuple in a DB table is uniquely identified by a set of attributes called *Primary Key* (PK).

There may be more than one set suitable for becoming PK. They are designated *Candidate Keys* (CK).

*First Normal Form* (1NF) is respected if each element of all tuples are atomic values.

*Second Normal Form* (2NF) is respected if the 1NF is respected and its non-key attributes are not functionally dependent on part of the key.

*Third Normal Form* (3NF) is respected if the 2NF is respected and if the non-key attributes are only dependent on the key attributes.
Our Approach

Normalize FDs

- Synthesize algorithm (Maier) calculates 3NF set of FDs
- It returns a set of attributes with candidate keys
- It is necessary to choose one CK to be the primary key
- To produce a schema with the lossless decomposition property one must add a FD (Maier):
  \[ \{ \text{all\_atts} \} \setminus \{ \text{atts\_rep\_formulas} \} \rightarrow \{ \text{new\_att}, \text{atts\_rep\_formulas} \} \]
- For our example:
  \[ \text{clientNo, propertyNo, cName, pAddress, rentStart, rentFinish, rentPerDay, ownerNo, oName} \rightarrow \text{totalDays, total rent, new\_att} \]
Our Approach

Calculate a RDB Schema

- Fun produces too many, not necessary and redundant FDs
- Also, attributes representing columns with formulas can not be used in a PK
- We use Synthesize to produce a 3NF set of attributes/CK
  - Its input is the result from Fun without the FDs with formulas in PKs and the FD that guarantees the lossless decomposition property
- We choose the smallest CK to be the PK
For our example,

owners
  ownerNo → oName

clients
  clientNo → cName

properties
  propertyNo → pAddress, rentPerDay, oName

renting
  clientNo, propertyNo, rentStart, rentFinish, ownerNo → total rent, totalDays
We need to migrate data from spreadsheets to the relational model.

We use *data refinements*:

\[
A[rr] \xrightarrow{\text{to}} \leq B[ll] \xleftarrow{\text{from}}
\]

- \( to : A \rightarrow B \) is an injective function;
- \( from : B \rightarrow A \) a surjective function;
- \( from \cdot to = id_A \) (identity function on \( A \));

\( B \) is a refinement of \( A \) witnessed by the \( to \) and \( from \) functions.

In the case where the refinement works in both directions we have an isomorphism \( A \cong B \).
Composition

\[ \text{if } A[rr] \rightarrow to \leq B[ll] \rightarrow from \text{ and } B[rr] \rightarrow to' \leq C[ll] \rightarrow from' \text{ then } A \]
From a Spreadsheet to a Database

Examples: Hierarchical-Relational Data Mapping

\[
\begin{align*}
A^* &\leq \mathbb{N} \rightarrow A \\
2^A &\cong A \rightarrow 1 \\
A? &\cong 1 \rightarrow A \\
A + B &\leq A? \times B?
\end{align*}
\]

- List elimination
- Set elimination
- Optional elimination
- Sum elimination
- Distribute product over sum
- Distribute map over sum (range)
- Distribute map over sum (domain)
To represent database constraints we use type constraints

- Associate a constraint $\phi$ to a type $A$: $A_\phi$ where $\phi : A \rightarrow \text{Bool}$
- Add a second constraint to a constrained datatype: $(A_\phi)_\psi \cong A_\phi \land \psi$
- Functorial pull: $F(A_\phi) \cong (FA)(F_\phi)$
  For example, a constraint on the elements of a list can be pulled up to a constraint on the list: $(A_\phi)^* \cong (A^*)_{\text{list}_\phi}$
Refining a constrained datatype (source)

\[
\text{if } A[rr] \to \leq B[ll] \from \text{ then } A_\phi[rr] \to \leq B_\phi \cdot \from[ll] \from
\]

A law that introduces a constraint to a constrained datatype

\[
\text{if } A[rr] \to \leq B_\psi[ll] \from \text{ then } A_\phi[rr] \to \leq (B_\psi)_\phi \cdot \from[ll] \from
\]
There is an Haskell implementation of data refinement called 2LT – Two Level Transformation (http://code.google.com/p/2lt)

We need a type-safe representation for types, so we use a GADT

```haskell
data Type t where
  Int  :: Type Int
  [·] :: Type a → Type [a]
  · × · :: Type a → Type b → Type (a, b)
  Func :: Type a → Type b → Type (a → b)
  (·). :: Type a → PF (Pred a) → Type a  (Aφ)
  ...

  type Pred a = a → Bool
```
For the function representation we also use a GADT

```haskell
data PF a where
  \pi_1 :: PF ((a, b) \to a)
  \delta :: PF ((a \to b) \to Set a)
  .* :: PF (a \to b) \to PF ([a] \to [b])
  \times :: PF (a \to b) \to PF (c \to d) \to PF ((a, c) \to (b, d))

...```

Function representations can be evaluated to the function that is represented:

```haskell
eval :: Type (a \to b) \to PF (a \to b) \to a \to b```
Data refinement rules in Haskell

\[
\text{type } \text{Rule} = \forall a \cdot \text{Type } a \rightarrow \text{Maybe } (\text{View } (\text{Type } a))
\]

\[
\text{data } \text{View } a \text{ where}
\]

\[
\text{View} :: \text{Rep } a b \rightarrow \text{Type } b \rightarrow \text{View } (\text{Type } a)
\]

\[
\text{data } \text{Rep } a b = \text{Rep}\{ \text{to} = \text{PF } (a \rightarrow b), \text{from} = \text{PF } (b \rightarrow a) \}\]
Constraint-Aware Rewriting

Rewriting composition operators

\[ \text{nop} :: \text{Rule} \quad \text{-- identity} \]
\[ \triangleright :: \text{Rule} \to \text{Rule} \to \text{Rule} \quad \text{-- sequential composition} \]
\[ \varnothing :: \text{Rule} \to \text{Rule} \to \text{Rule} \quad \text{-- left-biased choice} \]
\[ \text{many} :: \text{Rule} \to \text{Rule} \quad \text{-- repetition} \]
\[ \text{once} :: \text{Rule} \to \text{Rule} \quad \text{-- arbitrary depth rule application} \]
Refining a Relational Table to a Spreadsheet Table

\[ \text{Table2sstable} \]

\[ A \rightarrow B \leq (A \times B)^*_fd \]

\[ \text{Sstable2table} \]

**data PF a where**

... 

*Sstable2table :: PF ([([a, b]) \rightarrow (a \rightarrow b))

*Table2sstable :: PF ((a \rightarrow b) \rightarrow [(a, b)])

**table2sstable :: Rule**

*table2sstable (a \rightarrow b) = return (*View rep [a \times b]_{fd}*)

**where**

*rep = Rep{to = Table2sstable, from = Sstable2table}
A Foreign Key (FK) is a set of attributes within one relation that matches the PK of some relation.
Refining Tables where Non-Key Attributes Contain FKS

\[
((A \rightarrow B) \times (C \rightarrow D))_{\pi_B \circ \rho \circ \pi_1 \subseteq \pi_C \circ \delta \circ \pi_2}[dd]^{-} \text{Table2stable } \times \text{Table2stable}
\]

\[
((A \times B)_{fd}^* \times (C \times D)_{fd}^*)_{\pi_B \circ \text{list} \circ \pi_2^* \circ \pi_1 \subseteq \pi_C \circ \text{list} \circ \pi_1^* \circ \pi_2}[uu]^{-} \text{Sstable2table } \times
\]

From Spreadsheets to Relational Databases and Back
The Strategy

\[ rdb2ss :: \text{Rule} \]
\[ rdb2ss = simplifyInv \triangleright \]
\[ (\text{many } (\text{aux tables2table})) \triangleright \]
\[ (\text{many } ((\text{aux tables2sstables}) \triangleright (\text{aux tables2sstables}'))) \triangleright \]
\[ (\text{many } (\text{aux table2sstable})) \]

\text{where}

\[ \text{aux } r = ((\text{once } r) \triangleright \text{simplifyInv}) \triangleright ((\text{many } (\text{once } r)) \triangleright \text{simplifyInv}) \]
HaExcel is a framework to manipulate and transform spreadsheets

- Haskell library: generic and reusable library to transform spreadsheets into RDBs and back
- Front-ends: Excel/Gnumeric (XML) formats can be imported and exported
- Tools:
  - a batch tool, compilable from sources available at http://haskell.di.uminho.pt/websvn
Conclusions

- We extended the 2LT framework with new constraint-aware two-level data refinements.
- We shown how these rules can be combined into a strategic re-write system.
- We constructed HaExcel that transforms a relational schema into a tabular schema and back.
- Our techniques also migrate the formulas between the models.
- We connected importers and exporters for SQL and spreadsheet formats.
- This allows refactoring of spreadsheets to reduce data redundancy and improve error detection and migration of spreadsheet applications.
- An Excel plugin that implements HaExcel is under development.
Questions?