Calculate databases with ‘simplicity’

J.N. Oliveira
DI/U.Minho Braga, Portugal

IFIP 2.1 #59 Meeting,
Nottingham, UK
September 2004

Abstract

- Effort to replace “à la Codd” database schema design (normalization etc) by calculation based on simple (dually, injective) binary relations.
- Simple relations relevant because database entities can be modelled as finite such relations.
- (Pointfree) calculus simpler to use than the standard theory.
- Generic result which enables the refinement of recursive data models
- Prospect of automatic SQL generation (using Strafunski / Haskell) based on results so far.

Motivation

- SQL — data-processing standard “de facto”
- XML — abstract syntax “made popular”
- Can XML be trusted as a data-storage technology?
- Ad hoc XML ↔ SQL conversion
- Need for reliable XML ↔ SQL data exchange technology
Example

(Haskell instead of XML, if you don’t mind):

```haskell
type StringExp = Exp String String

data Exp v o = Var v 
  Term o [Exp v o]
```

How do you SQL-archive `StringExp` data?

Example — SQL

```
CREATE TABLE SYMBOLS ( 
  Symbol CHAR (20) NOT NULL, 
  NodeId NUMERIC (10) NOT NULL, 
  IfVar BOOLEAN NOT NULL 
CONSTRAINT SYMBOLS_pk 
  PRIMARY KEY(NodeId,IfVar) 
);

CREATE TABLE EXPRESSIONS ( 
  FatherId NUMERIC (10) NOT NULL, 
  ArgNr NUMERIC (10) NOT NULL, 
  ChildId NUMERIC (10) NOT NULL 
CONSTRAINT EXPRESSIONS_pk 
  PRIMARY KEY (FatherId,ArgNr) 
);
```

Can you rely on this implementation?
Overall idea

- Calculate implementations from specifications
  
  \[
  Spec = X \\
  \leq X' \\
  \leq X'' \\
  \leq \ldots \\
  \leq Imp
  \]

  by adding details in a controlled manner.

- Define a suitable ordering \( \leq \) on datatypes and develop corresponding data refinement theory

Example of data refinement

Finite sets represented by finite lists:

\[
F = \text{elems} \\
\{1, 2\} \\
[1, 2] \\
\{2, 1\} \\
[2, 1] \\
\{1, 2, 1\}
\]
Refinement inequation

\[ \mathcal{P}_f A \leq_{\text{elems}} A^* \]

meaning that
- sets are “implemented” by lists
- \( A^* \) is able to “represent” \( \mathcal{P}_f A \)
- \( A^* \) is “abstracted” by \( \mathcal{P}_f A \)
- \( A^* \) is a refinement (“refines”) \( \mathcal{P}_f A \)

Refinement inequations

\( A \) is implemented by \( B \), as witnessed by pair \( f, r \), iff

\[ A \leq_{f} B \]

such that
- representation \( r \) is injective
- abstraction \( f \) is surjective
- that is,

\[ f \cdot r = id \]

Not general enough (I)

In the following inequation

\[ A \leq_{i_1 i_1} A + 1 \]

expressing the fact that every element of datatype \( A \) can be represented by a “pointer”,
- \( r = i_1 \) is injective, but
- its converse \( i_1 \) is partial (=not entirely defined)
Not general enough (II)

Representations $r$ need not be functions. Back to

\[ R \quad \mathcal{P} A \quad \subseteq \quad A^* \]

relation $R = \text{elems}^\circ$ will be perfectly acceptable as a representation since

\[ \text{elems} \cdot \text{elems}^\circ = \text{id} \]

because $\text{elems}$ is a surjection.

Binary relation taxonomy

Terminology: simple / entire relation instead of partial / total function (or relation)

Taxonomy basis

<table>
<thead>
<tr>
<th></th>
<th>Reflexive</th>
<th>Coreflexive</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ker R$</td>
<td>entire $R$</td>
<td>injective $R$</td>
</tr>
<tr>
<td>$\text{img} R$</td>
<td>surjective $R$</td>
<td>simple $R$</td>
</tr>
</tbody>
</table>

where

- Reflexive relation: $\text{id} \subseteq R$
- Coreflexive relation (or partial identity): $R \subseteq \text{id}$
- Kernel and image:
  \[ \ker R = R^\circ \cdot R \]
  \[ \text{img} R = R \cdot R^\circ \quad (= \ker (R^\circ)) \]
Principle of data abstraction

where

\[ A \xrightarrow{R} B \]

\[ F \]

- \( A \xleftarrow{F} B \) is a **surjective** + **simple** abstraction relation
- \( R \) is **entire** and \( R \subseteq F^o \) — it is said to be a **representation** for \( F \).

The fact that \( R \) is **injective** follows from \( R \subseteq F^o \).

Summary

| \( \ker R = id \) | entire \( R \) \& injective \( R \) | representation \( R \) |
| \( \text{img } F = id \) | surjective \( F \) \& simple \( F \) | abstraction \( F \) |

It follows that \( R \) is a **right-inverse** of \( F \), that is

\[ F \cdot R = id \]

This is proved by circular inclusion

\[ F \cdot R \subseteq id \subseteq F \cdot R \]

in the next slide.
Right invertibility

\[ F \cdot R \subseteq id \land id \subseteq F^{-1} \cdot R \]

\[ \equiv \{ \text{img} F = id \text{ and } \text{ker} R = id \} \]

\[ F \cdot R \subseteq F \cdot F^o \land R^o \cdot R \subseteq F \cdot R \]

\[ \equiv \{ \text{converse of right conjunct} \} \]

\[ F \cdot R \subseteq F \cdot F^o \land R^o \cdot R \subseteq R^o \cdot F^o \]

\[ \iff \{ (F^{-1}) \text{ and } (R^o \cdot R^{-1}) \text{ are monotonic} \} \]

\[ R \subseteq F^o \land R \subseteq F^o \]

\[ \equiv \{ R \subseteq F^o \text{ is assumed} \} \]

TRUE

Functional abstractions

Quotation from [Mor90], chapter 17, pp. 173–174:

[It is] common for the coupling invariant to be functional from concrete to abstract. […]

The general form for such coupling invariants, called functional abstractions is

\[ \alpha = af c \land dti c \quad , \quad (17.1) \]

where \( af \) is a function, called the abstraction function and \( dti \) is a predicate, in which \( \alpha \) does not appear, called the data-type invariant. […]

That is,

\[ F = af \cdot [dti] \]

Functional abstractions

- **Galois abstractions** — let \( R, F := f^1 \), \( f \) be Galois connected functions where the connection is perfect on the “abstract side”,

\[ f \cdot f^1 = id \]

Example: hash-table representation of a data collection [OR04]

- **Isomorphisms** —

  \[ A \overset{r}{\cong} B \quad \text{such that} \quad r = f^o \]
\( \leq \) is a preorder

### Reflexivity

\[ A \leq A \]

\( \text{id} \)

### Transitivity

\[ A \leq B \land B \leq C \Rightarrow A \leq C \]

\( R \leq F \)

\( S \leq G \)

\( S \cdot R \leq F \cdot G \)

### Proof of transitivity

a) Composition preserves simplicity and surjectiveness:

\[
\text{img} (F \cdot G) = \text{id}
\]

\[
\equiv \quad \{ \text{expand } \text{img} : \text{converses} \}
\]

\[
F \cdot (\text{img} G) \cdot F^\circ = \text{id}
\]

\[
\equiv \quad \{ G \text{ is simple and surjective} \}
\]

\[
\text{img} F = \text{id}
\]

\[
\equiv \quad \{ F \text{ is simple and surjective} \}
\]

\[
id = id
\]

b) \( S \cdot R \leq (F \cdot G)^\circ \) by monotonicity.

### Structural data refinement

For \( F \) a relator,

\[ A \leq B \Rightarrow FA \leq FB \]

\[ F \leq F \]

- Easy proof next slides.
- Also valid for \( n \)-ary relators such as \( \times \), \( + \) etc.
Structural data refinement

a) Abstraction:

\[ \text{img}(F F') \]

\[ = \{ \text{image definition ; relators commute with converse } \} \]

\[ (F(F'^o)) \cdot (F F') \]

\[ = \{ \text{relators commute with composition } \} \]

\[ F(F'^o \cdot F) \]

\[ = \{ \text{F is an abstraction } \} \]

\[ F \text{id} \]

\[ = \{ \text{relators commute with id} \} \]

\[ \text{id} \]

Structural data refinement

b) Representation:

\[ FR \subseteq (F F'^o) \]

\[ \equiv \{ \text{relators commute with converse } \} \]

\[ FR \subseteq F(F'^o) \]

\[ \equiv \{ \text{relators are monotone } \} \]

\[ R \subseteq F'^o \]

\[ \equiv \{ \text{R is a representation for F } \} \]

\[ \text{TRUE} \]

By the way

Datatypes such as \( P \ A \) are \( \leq \) postfix points [Bac00], cf.

\[ P \ A \]

\[ \leq \]

\[ 1 + A \times P \ A \]

\[ R, \text{ins}^\alpha \]

\[ \text{ins} \]

\[ id + id \times [ R, \text{ins}^\alpha ] \]

\[ B \]

\[ 1 + A \times B \]

(hylomorphisms on finite sets).
Abstract database models

- A relational database is a tuple of finite relations
- Finite simple relations model many-to-one (M:1) relationships (inc. primary key relationships)
- Finite simple+injective relations model one-to-one (1:1) relationships

Notation:
- $B \leftarrow A$ : all simple relations from $A$ (the key) to $B$ (the data of interest) —cf. (if also finite) `FiniteMap a b` in Haskell, `map A to B` in VDM-SL.
- $B \leftrightarrow A$ : all injective relations from $A$ to $B$ —cf. (if also finite and simple) `inmap A to B` in VDM-SL,

“Maybe” transpose

Useful isomorphism

\[
(B + 1)^A \cong B \leftarrow A
\]

converts simple relations into ($+ 1$)-valued functions ($\epsilon = i_1^n$):

\[
f = \Gamma R \equiv (b R a \equiv (f a = i_1 b))
\]

NB: generalizes to generic transpose [OR04]

... and exponentials

Multiple-key decomposition / synthesis:

\[
A \leftarrow B \times C
\]

\[
\cong \{ \Gamma \}
\]

\[
(A + 1)^{B \times C}
\]

\[
\cong \{ \text{curry} \}
\]

\[
((A + 1)^C)^B
\]

\[
\cong \{ (\epsilon :)^B \}
\]

\[
(A \leftarrow C)^B
\]
Calculating abs/reps

Altogether, the downwards isomorphism

$$(\varepsilon \cdot)^B \cdot \text{curry} \cdot \Gamma$$

is a convenient shorthand for a less readable pointwise abstraction invariant:

$$\mathcal{S} = (\varepsilon \cdot (\text{curry}(\Gamma \cdot S)))$$

$$\equiv \quad \{ \ldots \text{relational calculus} \ldots \}$$

$$(b, c) \in \text{dom} \mathcal{S} \equiv c \in \text{dom} (\mathcal{S} b) \land S(b, c) = \mathcal{S} b \cdot c$$

NB: thanks to generic transpose, notation $\mathcal{S}$ extends to other classes of relation.

Converse of simple is injective

$$B \times C \leftrightarrow A$$

$$\equiv \quad \{ (\_^B) \text{ isomorphism} \}$$

$$A \leftrightarrow B \times C$$

$$\equiv \quad \{ \text{above} \}$$

$$(A \leftrightarrow C)^B$$

$$\equiv \quad \{ \text{isomorphism is } (\_^B) \}$$

$$(C \leftrightarrow A)^B$$
Refinement by decomposition

**Zip/unzip'ping simple relations:**

\[ B \times C \triangleleft A \leq (B \triangleleft A) \times (C \triangleleft A) \]

where

\[ \text{join} = \langle \cdot \rangle \]

\[ (R, S) \overset{\text{def}}{=} (\pi_1 \cdot R) \cap (\pi_2 \cdot S) \]

\[ \text{unjoin} \overset{\text{def}}{=} \langle \pi_1 \triangleleft \text{id}, \pi_2 \triangleleft \text{id} \rangle \]

where, for injective \( f \),

\[ g \triangleleft f \overset{\text{def}}{=} (g \cdot f^\ast) \]

Refinement by decomposition

\[ \text{uncojoin} \]

\[ (B + C) \triangleleft A \leq (B \triangleleft A) \times (C \triangleleft A) \]

where

\[ \text{cojoin} \]

\[ \text{uncojoin} = \langle (i_1^\ast \cdot), (i_2^\ast \cdot) \rangle \]

\[ \text{cojoin} = \bigcup \cdot ((i_1 \cdot) \times (i_2 \cdot)) \]

**NB:** \( \text{cojoin} \cdot \text{uncojoin} = \text{id} \), since \( \text{img} i_1 \cup \text{img} i_2 = \text{id} \).

Refinement by decomposition

**Nested simplicity:**

\[ \text{unjoin} \]

\[ (D \times (C \triangleleft B)) \triangleleft A \leq (D \triangleleft A) \times (C \triangleleft (A \times B)) \]

**Defi nitions of** \( n\text{join} \) **and** \( \text{unjoin} \) **to follow from next slide's calculation**
Calculation

\[(D \times (C \leftarrow B)) \leftarrow A\]
\[\cong\]
\{ Maybe transpose \}
\[((D \times (C \leftarrow B)) + 1)^A\]
\[\cong\]
\{ Maybe-(right)strength is involved in the abstraction \}
\[((D + 1) \times (C \leftarrow B)) A\]
\[\cong\]
\{ splitting \}
\[(D + 1)^A \times (C \leftarrow B)^A\]
\[\cong\]
\{ Maybe transpose and above \}
\[(D \leftarrow A) \times (C \leftarrow A \times B)\]

Getting away with finite lists

Several other \(\leq\) laws, eg.

\[\text{seq2index}\]

\[A^* \xrightarrow{\leq} A \leftarrow \mathbb{N}\]

\[\text{list}\]

such that, for instance,

\[\text{seq2index} \{a, b, a\} = \{(a, 1), (b, 2), (a, 3)\}\]

\[\text{list} \{(a, 11), (b, 12), (a, 33)\} = [a, b, a]\]

Getting away with recursion

Given

\[\mu F \cong F \mu F\]

one has

\[\mu F \leq (F K \leftarrow K) \times K\]

\[\text{"heap addresses" or "pointers"}\]

for \(K\) a data type of "heap addresses", or "pointers", such that \(K \cong \mathbb{N}\).
Abstraction function

- Main rôle in representation is played by simple F-coalgebra $F K \triangleleft \mathit{K}$, understood as a (finite) piece of “linear storage”, a “heap” or a “database” file.
- $\overline{F}$ (recall $\overline{F}$ notation from above), of type $(\mu F \triangleleft \mathit{K})^{(F K \triangleleft \mathit{K})}$, is nothing but the F-anamorphism combinator:

$$
\begin{align*}
\mu F & \quad \text{in} \quad F(\mu F) \\
\overline{F} H & \quad \text{in} \quad F(\overline{F} H) \\
F K & \quad \text{in} \quad (F X) \cdot H
\end{align*}
$$

Partiality of implementation

Abstraction invariant $t = F(H, k)$ — that is, $t = (\overline{F} H) k$ — will hold only if
- $k \in \text{dom} \mathit{H}$, and
- the accessibility relation for $H$

$$
\begin{align*}
K & \mapsto H \\
K & \mapsto H \quad \text{def} \quad \mathcal{E}_F \cdot H
\end{align*}
$$

is well-founded and closed ($K \mapsto F K$ is the membership of $F$.)
( Many details omitted here!

Back to the **StringExp** example

Since

$$
\text{StringExp} = \mu X. (\text{String} + \text{String} \times X^-)
$$

we have:

$$
\begin{align*}
\text{StringExp} & \leq \{ \text{remove recursion} \} \\
((\text{String} + \text{String} \times K^-) \triangleleft K) \times K \\
& \leq \{ \text{remove finite lists} \} \\
((\text{String} + \text{String} \times (K \triangleleft \mathcal{N})) \triangleleft K) \times K
\end{align*}
$$
Example continued

\[
\begin{align*}
\leq & \quad \{ \text{recall } (B + C) \leftarrow A \leq (B \leftarrow A) \times (C \leftarrow A) \} \\
& \quad (\text{String} \leftarrow K) \times ((\text{String} \times (K \leftarrow \mathbb{N})) \leftarrow K) \times K \\
\leq & \quad \{ \text{remove nested } \leftarrow \} \\
& \quad (\text{String} \leftarrow K) \times (\text{String} \leftarrow K) \times (K \leftarrow (\mathbb{N} \times K)) \times K \\
\equiv & \quad \{ A \times A \cong A^2 \} \\
& \quad (\text{String} \leftarrow K)^2 \times (K \leftarrow (\mathbb{N} \times K)) \times K \\
\equiv & \quad \{ \text{recall } (A \leftarrow C)^m \cong A \leftarrow B \times C \} \\
& \quad (\text{String} \leftarrow (2 \times K))^2 \times (K \leftarrow (\mathbb{N} \times K)) \times K
\end{align*}
\]

Conclusions

- Database schema design as a special case of "do it by calculation" data refinement
- Calculational alternative to state-of-the-art casuistic practice stemming from set-theoretic "normalization theory"
- Many more laws available, eg.
  \[1 \leftarrow A \cong \mathcal{P}A\]
  cf.
  \[
  \text{newtype Set a = MkSet (FiniteMap a ()})
  \]
  in the \texttt{FiniteMap / Set.lhs} Haskell libraries.