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 $\leftrightarrow$ SQL conversion
- Need for reliable XML $\leftrightarrow$ SQL data exchange technology

Example

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

```
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

$$\begin{array}{rcl} Spec & = & X \\ & \leq & X' \\ & \leq & X'' \\ & \leq & \cdots \\ & \leq & Imp \end{array}$$

  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**:

$$\begin{tikzcd} & {[1,2]} \arrow[dl, bend right, "F = \mathit{elems}"'] \\ \{1,2\} & {[2,1]} \arrow[l] \\ & {[1,2,1]} \arrow[ul, bend left] \end{tikzcd}$$

Refinement inequation

$$
\begin{tikzcd}
\mathcal{P}_f\, A \arrow[r, bend left] \arrow[r, phantom, "\leq"] & A^\star \arrow[l, bend left, "elems"]
\end{tikzcd}
$$

meaning that

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

Refinement inequations

A *is implemented by* B, as witnessed by pair f, r, iff

$$
\begin{tikzcd}
A \arrow[r, bend left, "r"] \arrow[r, phantom, "\leq"] & B \arrow[l, bend left, "f"]
\end{tikzcd}
$$

such that

- **representation** r is injective
- **abstraction** f is surjective
- that is,

$$f \cdot r \quad = \quad id$$

Not general enough (I)

In the following inequation

$$
\begin{tikzcd}
A \arrow[r, bend left, "i_1"] \arrow[r, phantom, "\leq"] & A + 1 \arrow[l, bend left, "i_1^\circ"]
\end{tikzcd}
$$

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

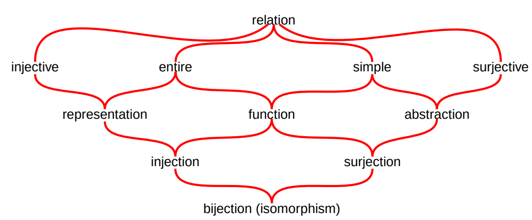
$$\begin{tikzcd} \mathcal{P}_f\, A \arrow[r, bend left, "R"] \arrow[r, phantom, "\leq"] & A^\star \arrow[l, bend left, "elems"] \end{tikzcd}$$

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

$$elems \cdot elems^\circ = id$$

because $elems$ is a surjection.

Binary relation taxonomy



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

Taxonomy basis

	Reflexive	Coreflexive
ker R	entire R	injective R
img R	surjective R	simple R

where

- **Reflexive** relation: $id \subseteq R$
- **Coreflexive** relation (or **partial identity**): $R \subseteq id$
- **Kernel** and **image**:

$$\begin{array}{rcl} \mathit{ker}\ R & = & R^\circ \cdot R \\ \mathit{img}\ R & = & R \cdot R^\circ \quad (= \mathit{ker}\ (R^\circ)) \end{array}$$

Principle of data abstraction

$$\begin{tikzcd} A \arrow[r, bend left, "R"] \arrow[r, phantom, "\leq"] & B \arrow[l, bend left, "F"] \end{tikzcd}$$

where

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

The fact that R is **injective** follows from $R \subseteq F^\circ$.

Summary

ker $R = id$	entire R $\wedge$ injective R	representation R
img $F = id$	surjective F $\wedge$ simple F	abstraction F

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

$$F \cdot R \quad = \quad id$$

This is proved by circular inclusion

$$F \cdot R \subseteq id \subseteq F \cdot R$$

in the next slide.

Right invertibility

$$
\begin{array}{cl}
 & F \cdot R \subseteq id \ \wedge \ id \subseteq F \cdot R \\
\equiv & \quad \{ \ \textit{img } F = id \text{ and } \textit{ker } R = id \} \\
 & F \cdot R \subseteq F \cdot F^{\circ} \ \wedge \ R^{\circ} \cdot R \subseteq F \cdot R \\
\equiv & \quad \{ \ \text{converse of right conjunct} \ \} \\
 & F \cdot R \subseteq F \cdot F^{\circ} \ \wedge \ R^{\circ} \cdot R \subseteq R^{\circ} \cdot F^{\circ} \\
\Leftarrow & \quad \{ \ (F \cdot) \text{ and } (R^{\circ} \cdot) \text{ are monotonic} \ \} \\
 & R \subseteq F^{\circ} \ \wedge \ R \subseteq F^{\circ} \\
\equiv & \quad \{ \ R \subseteq F^{\circ} \text{ is assumed} \ \} \\
 & \textsc{True}
\end{array}
$$

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*
>
> $$a = \mathsf{af}\ c \ \wedge \ \mathsf{dti}\ c \ , \qquad (17.1)$$
>
> *where af is a function, called the abstraction function and dti is a predicate, in which a does not appear, called the data-type invariant. [...]*

That is,

$$F \quad = \quad af \cdot [\![dti]\!]$$

Functional abstractions

- **Galois abstractions** —let $R, F := f^{\flat}, f$ be Galois connected functions where the connection is **perfect** on the "abstract side",

  $$f \cdot f^{\flat} = id$$

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

$$
\begin{tikzcd}
A \arrow[r, bend left, "r"] \arrow[r, phantom, "\cong"] & B \arrow[l, bend left, "f"]
\end{tikzcd}
\qquad \text{such that} \quad r = f^{\circ}
$$

$\leq$ is a preorder

Reflexivity

$$\begin{tikzcd} A \arrow[r, bend left, "id"] \arrow[r, phantom, "\leq"] & A \arrow[l, bend left, "id"] \end{tikzcd}$$

Transitivity

$$\begin{tikzcd} A \arrow[r, bend left, "R"] \arrow[r, phantom, "\leq"] & B \arrow[l, bend left, "F"] \end{tikzcd} \wedge \begin{tikzcd} B \arrow[r, bend left, "S"] \arrow[r, phantom, "\leq"] & C \arrow[l, bend left, "G"] \end{tikzcd} \Rightarrow \begin{tikzcd} A \arrow[r, bend left, "S \cdot R"] \arrow[r, phantom, "\leq"] & C \arrow[l, bend left, "F \cdot G"] \end{tikzcd}$$

Proof of transitivity

a) Composition preserves simplicity and surjectiveness:

$$\begin{aligned} & \mathit{img}\,(F \cdot G) = id \\ \equiv & \quad \{ \text{ expand } img \text{ ; converses} \} \\ & F \cdot (\mathit{img}\, G) \cdot F^{\circ} = id \\ \equiv & \quad \{ \ G \text{ is simple and surjective} \} \\ & \mathit{img}\, F = id \\ \equiv & \quad \{ \ F \text{ is simple and surjective} \} \\ & id = id \end{aligned}$$

b) $S \cdot R \subseteq (F \cdot G)^{\circ}$ by monotonicity.

Structural data refinement

For F a relator,

$$\begin{tikzcd} A \arrow[r, bend left, "R"] \arrow[r, phantom, "\leq"] & B \arrow[l, bend left, "F"] \end{tikzcd} \Rightarrow \begin{tikzcd} \mathsf{F}\,A \arrow[r, bend left, "\mathsf{F}\,R"] \arrow[r, phantom, "\leq"] & \mathsf{F}\,B \arrow[l, bend left, "\mathsf{F}\,F"] \end{tikzcd}$$

- Easy proof next slides.
- Also valid for n-ary relators such as $\times$, $+$ etc.

Structural data refinement

a) **Abstraction:**

$$
\begin{array}{ll}
 & \mathit{img}\,(\mathsf{F}\,F) \\
= & \quad \{ \text{ image defi nition ; relators commute with converse } \} \\
 & (\mathsf{F}(F^\circ)) \cdot (\mathsf{F}\,F) \\
= & \quad \{ \text{ relators commute with composition } \} \\
 & \mathsf{F}(F^\circ \cdot F) \\
= & \quad \{ \ F \text{ is an abstraction } \} \\
 & \mathsf{F}\,id \\
= & \quad \{ \text{ relators commute with } id \} \\
 & id
\end{array}
$$

Structural data refinement

b) **Representation:**

$$
\begin{array}{ll}
 & \mathsf{F}\,R \ \subseteq \ (\mathsf{F}\,F)^\circ \\
\equiv & \quad \{ \text{ relators commute with converse } \} \\
 & \mathsf{F}\,R \ \subseteq \ \mathsf{F}(F^\circ) \\
\Leftarrow & \quad \{ \text{ relators are monotone } \} \\
 & R \ \subseteq \ F^\circ \\
\equiv & \quad \{ \ R \text{ is a representation for } F \ \} \\
 & \textsc{True}
\end{array}
$$

By the way

Datatypes such as $\mathcal{P}_f\,A$ are $\leq$ postfix points [Bac00], cf.

$$
\begin{tikzcd}
\mathcal{P}_f\,A \arrow[rr, bend left, "ins^\circ"] \arrow[d, "{[\![\, R, ins^\circ \,]\!]}"'] \arrow[rr, phantom, "\leq"] & & 1 + A \times \mathcal{P}_f\,A \arrow[ll, bend left, "ins"] \arrow[d, "{id + id \times [\![\, R, ins^\circ \,]\!]}"] \\
B & & 1 + A \times B \arrow[ll, "R"]
\end{tikzcd}
$$

(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:

- $B \leftharpoonup A$: 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.
- $B \hookleftarrow A$: all injective relations from A to B —cf. (if also finite and simple) `inmap` A `to` B in VDM-SL,

"Maybe" transpose

Useful isomorphism

$$\begin{tikzcd}
(B+1)^A \arrow[rr, bend left, "(\in \cdot)"] \arrow[rr, phantom, "\cong"] & & B \leftharpoonup A \arrow[ll, bend left, "\Gamma"]
\end{tikzcd}$$

converts **simple** relations into $(\_ + 1)$-valued functions ($\in = i_1^\circ$):

$$f = \Gamma\, R \quad \equiv \quad (b\, R\, a \quad \equiv \quad (f\, a = i_1\, b))$$

NB: generalizes to generic transpose [OR04]

. . . and exponentials

Multiple-key decomposition / synthesis:

$$\begin{array}{ll}
 & A \leftharpoonup B \times C \\
\cong & \quad \{\ \Gamma\ \} \\
 & (A+1)^{B \times C} \\
\cong & \quad \{\ curry\ \} \\
 & ((A+1)^C)^B \\
\cong & \quad \{\ (\in \cdot)^B\ \} \\
 & (A \leftharpoonup C)^B
\end{array}$$

Calculating abs/reps

Altogether, the downwards isomorphism

$$(\in \cdot)^B \cdot curry \cdot \Gamma$$

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

$$\begin{array}{lcl} & \overline{S} = (\in\cdot) \cdot (curry(\Gamma S)) & \\ \equiv & \quad \{ \text{ ... relational calculus ... } \} & \\ & (b,c) \in \mathit{dom}\, S \;\equiv\; c \in \mathit{dom}\, (\overline{S}\, b) \;\wedge\; S(b,c) = \overline{S}\, b\, c & \end{array}$$

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

Converse of simple is injective

$$\begin{array}{cl} & B \times C \hookleftarrow A \\ \cong & \quad \{ \ (\_^\circ) \text{ isomorphism } \} \\ & A \leftharpoonup B \times C \\ \cong & \quad \{ \text{ above } \} \\ & (A \leftharpoonup C)^B \\ \cong & \quad \{ \text{ isomorphism is } (\_^\circ)^B \ \} \\ & (C \hookleftarrow A)^B \end{array}$$

Refinement by decomposition

$Zip/unzip$'ping simple relations:

$$\begin{tikzcd} B \times C \leftharpoonup A \arrow[rr, bend left, "unjoin"] & \leq & (B \leftharpoonup A) \times (C \leftharpoonup A) \arrow[ll, bend left] \end{tikzcd}$$

where

$$join = \langle \_, \_ \rangle$$

$$\begin{aligned} \langle R, S \rangle &\stackrel{\mathrm{def}}{=} (\pi_1^\circ \cdot R) \cap (\pi_2^\circ \cdot S) \\ unjoin &\stackrel{\mathrm{def}}{=} \langle \pi_1 \leftharpoonup id, \pi_2 \leftharpoonup id \rangle \end{aligned}$$

where, for injective f,

$$g \leftharpoonup f \stackrel{\mathrm{def}}{=} (g\cdot) \cdot (\cdot f^\circ)$$

Refinement by decomposition

$$\begin{tikzcd} (B + C) \leftharpoonup A \arrow[rr, bend left, "uncojoin"] & \leq & (B \leftharpoonup A) \times (C \leftharpoonup A) \arrow[ll, bend left, "cojoin"] \end{tikzcd}$$

where

$$\begin{aligned} uncojoin &= \langle (i_1^\circ \cdot), (i_2^\circ \cdot) \rangle \\ cojoin &= \cup \cdot ((i_1 \cdot) \times (i_2 \cdot)) \end{aligned}$$

NB: $cojoin \cdot uncojoin = id$, since *img* $i_1 \cup$ *img* $i_2 = id$.

Refinement by decomposition

Nested simplicity:

$$\begin{tikzcd} (D \times (C \leftharpoonup B)) \leftharpoonup A \arrow[rr, bend left, "unnjoin"] & \leq & (D \leftharpoonup A) \times (C \leftharpoonup (A \times B)) \arrow[ll, bend left, "njoin"] \end{tikzcd}$$

Defi nitions of $njoin$ and $unnjoin$ to follow from next slide's calculation

Calculation

$$
\begin{array}{cl}
 & (D \times (C \leftharpoonup B)) \leftharpoonup A \\
\cong & \quad \{ \ \mathit{Maybe} \text{ transpose } \} \\
 & ((D \times (C \leftharpoonup B)) + 1)^A \\
\leq & \quad \{ \ \mathit{Maybe}\text{-(right)strength is involved in the abstraction } \} \\
 & ((D + 1) \times (C \leftharpoonup B))^A \\
\cong & \quad \{ \text{ splitting } \} \\
 & (D + 1)^A \times (C \leftharpoonup B)^A \\
\cong & \quad \{ \ \mathit{Maybe} \text{ transpose and above } \} \\
 & (D \leftharpoonup A) \times (C \leftharpoonup A \times B)
\end{array}
$$

Getting away with finite lists

Several other $\leq$ laws, eg.

$$
\begin{tikzcd}
A^\star \arrow[r, bend left, "seq2index"] \arrow[r, phantom, "\leq"] & A \leftharpoonup I\!N \arrow[l, bend left, "list"]
\end{tikzcd}
$$

such that, for instance,

$$
\begin{array}{rcl}
seq2index\ [a, b, a] & = & \{(a, 1), (b, 2), (a, 3)\} \\
list\ \{(a, 11), (b, 12), (a, 33)\} & = & [a, b, a]
\end{array}
$$

Getting away with recursion

Given

$$
\begin{tikzcd}
\mu\mathsf{F} \arrow[r, bend left, "out"] \arrow[r, phantom, "\cong"] & \mathsf{F}\,\mu\mathsf{F} \arrow[l, bend left, "in"]
\end{tikzcd}
$$

one has

$$
\begin{tikzcd}
\mu\mathsf{F} \arrow[r, bend left] \arrow[r, phantom, "\leq"] & \underbrace{(\mathsf{F}\,K \leftharpoonup K)}_{``heap''} \times K \arrow[l, bend left, "F"]
\end{tikzcd}
$$

for K a data type of "heap addresses", or "pointers", such that $K \cong I\!N$.

Abstraction function

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

$$
\begin{tikzcd}
\mu\mathsf{F} & \mathsf{F}(\mu\mathsf{F}) \arrow[l, "in"'] \\
K \arrow[u, "\overline{F}H"] \arrow[r, "H"'] & \mathsf{F}\,K \arrow[u, "\mathsf{F}(\overline{F}H)"']
\end{tikzcd}
\qquad
\begin{array}{rcl}
\overline{F} & = & [\!(\_)\!]_{\mathsf{F}} \\
\overline{F}\,H & = & \mu X.\, in \cdot (\mathsf{F}\,X) \cdot H
\end{array}
$$

Partiality of implementation

Abstraction invariant $t = F(H, k)$ —that is, $t = (\overline{F}H)k$ —will hold only if

- $k \in \mathit{dom}\,H$, and
- the accessibility relation for H

$$
K \xleftarrow{\prec_H} K
$$

$$
\prec_H \;\stackrel{\mathrm{def}}{=}\; \in_{\mathsf{F}} \cdot H
$$

is well-founded and closed ($K \xleftarrow{\in_{\mathsf{F}}} \mathsf{F}\,K$ is the membership of F.)

(Many details omitted here!)

Back to the $StringExp$ example

Since

$$
StringExp = \mu X.(String + String \times X^\star)
$$

we have:

$$
\begin{array}{cl}
 & StringExp \\
\leq & \quad \{\ \text{remove recursion}\ \} \\
 & ((String + String \times K^\star) \leftharpoonup K) \times K \\
\leq & \quad \{\ \text{remove finite lists}\ \} \\
 & ((String + String \times (K \leftharpoonup \mathbb{N})) \leftharpoonup K) \times K
\end{array}
$$

Example continued

$$
\begin{array}{ll}
\leq & \{\ \text{recall } (B+C) \leftharpoonup A \leq (B \leftharpoonup A) \times (C \leftharpoonup A)\ \} \\
 & (String \leftharpoonup K) \times ((String \times (K \leftharpoonup \mathbb{N})) \leftharpoonup K) \times K \\
\leq & \{\ \text{remove nested } \leftharpoonup\ \} \\
 & (String \leftharpoonup K) \times (String \leftharpoonup K) \times (K \leftharpoonup (\mathbb{N} \times K)) \times K \\
\cong & \{\ A \times A \cong A^2\ \} \\
 & (String \leftharpoonup K)^2 \times (K \leftharpoonup (\mathbb{N} \times K)) \times K \\
\cong & \{\ \text{recall } (A \leftharpoonup C)^B \cong A \leftharpoonup B \times C \} \\
 & \underbrace{(String \leftharpoonup (2 \times K))}_{SYMBOLS} \times \underbrace{(K \leftharpoonup (\mathbb{N} \times K))}_{EXPRESSIONS} \times K
\end{array}
$$

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.

$$1 \leftharpoonup A \quad \cong \quad \mathcal{P} A$$

cf.

```
newtype Set a = MkSet (FiniteMap a ())
```

in the FiniteMap / Set.lhs Haskell libraries.