Chapter 4

Data representation and refinement

Isomorphism provides us with a notion of data-equivalence. How do we "order" data-structures according to their "size" or "expressive power"? Are there "better" or "worse" data-structures for a given purpose? In this chapter — which is currently in a very drafty version — we address the field of *formal* data processing and refinement by transformation.

4.1 Introduction

The definition of a function $B \xleftarrow{f} A$ can be regarded as a kind of "contract": function f is committed to produce an A-value provided it is supplied with a B-value. Such a "functional contract" can be of two kinds: (a) f intentionally loses information, because B is found too detailed and one wants to capture only the A-aspect of B-values — so, A is an *abstraction* of B (f is non-injective); (b) f faithfully converts data from the B-format to the A-format — so, f is injective and, in the limit, the two formats are the same (f is the identity).

Case (a) above is perhaps more interesting than (b) and supports the following aphorism about a facet of *functional programming*: it is the *art* of transforming or losing information in a controlled and precise way. In this way, it is close to *data-mining* which is nothing but a series of clever, highly scrutinized formal reductions of a complex input structure.

Case (a) also includes the process of *transforming* data from one format to another so that it fits in a particular context or matches a given requirement. One of the aspects of this is *data refinement*, which usually means a change of *data representation*: a problem is abstractly characterized in terms of clever, mathematically nice data models which either don't fit in traditional data structuring techniques, or they do but at a severe cost of efficiency.

Refinement is the process of converting (with a variable concern for formality)

abstract software specifications into real applications intended to run on some available hardware. Formal refinement is one of the most relevant branches of formal methods. It splits into *data refinement* and *algorithmic refinement*, depending on where refinement decisions take place: changing the format of *real* data in the former case, or replacing a *virtual* data-structure [1] by a more efficient (though equally virtual) one in the latter.

Formal refinement is not yet widely used due to the difficulty of scaling its mathematical reasoning to real size problems. By contrast, informal refinement enforces so little discipline in software design that results are very often catastrophic, particularly in the case of very large applications which have a long life-span. As a rule, such applications lack in documentation, have bugs and raise serious maintenance problems. Using or selling them becomes a risky business.

In this chapter we concentrate on data representation and refinement because, in formal software design, data calculation preceeds and paves the way to correct algorithm design by calculation.

4.2 Isomorphism

Isomorfism

$$A \underset{f}{\overset{f^{-1}}{\cong}} B \quad \text{that is} \quad \begin{cases} f \cdot f^{-1} = id_A \\ f^{-1} \cdot f = id_B \end{cases} \tag{4.1}$$

means that f^{-1} is both a *right-inverse* and a *left-inverse* of f. Further to many isomorphism laws studied in previous chapters, we are now interested in exploiting isomorphisms involving quasi-inductive datatypes:

- Powersets vs predicates: recall (3.28).
- Partial mappings vs total functions (for a finite A):

$$A \rightharpoonup B \underset{untot}{\overset{tot}{\cong}} (B+1)^A \tag{4.2}$$

  where

$$\begin{aligned} tot\,\sigma &\overset{\text{def}}{=} \lambda a.if\ a \in \mathsf{dom}\,\sigma\ then\ i_1(\sigma\,a)\ else\ i_2\,NIL \\ untot\,f &\overset{\text{def}}{=} \{a \mapsto b |\ a \in A \wedge f\,a = (i_1 b)\} \end{aligned}$$

- A consequence of the above,

$$\mathcal{P}A \underset{\mathsf{dom}}{\overset{set2fm}{\cong}} A \rightharpoonup 1$$

[1] Recall section 2.5.

cf.

$$
\begin{array}{ll}
 & A \rightharpoonup 1 \\
\cong & \quad \{ \text{converting to total maps (4.2)} \} \\
 & (1+1)^A \\
\cong & \quad \{ \text{basic} \} \\
 & 2^A \\
\cong & \quad \{ \text{by (3.28)} \} \\
 & \mathcal{P}A
\end{array}
$$

- Another example, extending $either$ (1.35) to partial maps:

$$(B + C) \rightharpoonup A \underset{peither}{\overset{unpeither}{\cong}} (B \rightharpoonup A) \times (C \rightharpoonup A) \tag{4.3}$$

Why an isomorphism? Check:

$$
\begin{array}{ll}
 & (B+C) \rightharpoonup A \\
\cong & \quad \{ \text{converting to total maps} \} \\
 & (A+1)^{(B+C)} \\
\cong & \quad \{ \text{exponentials (1.77)} \} \\
 & (A+1)^B \times (A+1)^C \\
\cong & \quad \{ \text{back to partial maps} \} \\
 & (B \rightharpoonup A) \times (C \rightharpoonup A)
\end{array}
$$

Abstraction and representation functions:

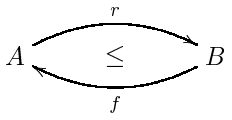
$$peither \quad \stackrel{\text{def}}{=} \quad \cup \cdot ((i_1 \rightharpoonup A) \times (i_2 \rightharpoonup A))$$

and, expressed in CAMILA,

```
; unpeither : ( ( A + B ) -> C )  -->  ( A -> C ) x ( B -
> C )
; The inverse of (unary) unpeither
  unpeither(f) = let (a=[ p2(x) -> f[x] | x <- dom(f): p1(x) == 1 ],
                      b=[ p2(x) -> f[x] | x <- dom(f): p1(x) == 2 ])
                 in < a , b >;
```

4.3 Representation

Representation of A in B means that there exists $A \xleftarrow{f} B$ which has a *right-inverse* r but no *left-inverse*. So, this means that B contains "more information" than A and, as in (3.5), we convey this fact by writing:

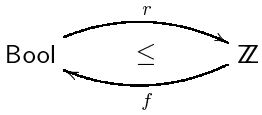


$$\text{that is} \quad \{ \; f \cdot r = id_A \tag{4.4}$$

Inequation (4.4) has several informal interpretations:

- A is "smaller" than B
- B is able to "represent" A
- B is "abstracted" by A
- A is "implemented" by B
- B is a refinement ("refines") A

In set theory one says that f is surjective [2]. Note that the same surjective f can have many right-inverses. Consider, for instance, the following inequation expressing the implementation of Booleans by integers,



where

$$f \quad \stackrel{\text{def}}{=} \quad =_0 \rightarrow \underline{\text{FALSE}}, \underline{\text{TRUE}}$$

(informal meaning: 0 represents FALSE, any other value represents TRUE). It is easy to see that, for every $k \in \mathbb{Z} - \{0\}$,

$$r \quad \stackrel{\text{def}}{=} \quad id \rightarrow \underline{k}, \underline{0}$$

is a right-inverse of f.

Right-inverses are always *left-invertible, i.e. injective* in set theory terminology [3]. This is a very important property of data representation — no two different abstract values $a, a' \in A$ get mixed up along the representation process:

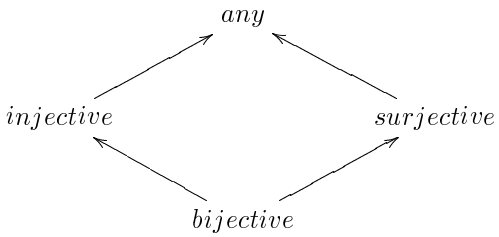
$$a \neq a' \quad \Rightarrow \quad (r\ a) \neq (r\ a')$$

[2] In category theory one would say that f is *split-epic*, a concept stronger than *epic* (*cf.* section 1.6).
[3] *split-monic* in category theory terminology.

An isomorphism f is an arrow which has both a right-inverse r and a left-inverse s — a bijection in set theory terminology. It is easy to show that $r = s = f^{-1}$, as in (4.1) [4]:

$$
\begin{array}{ll}
 & \left\{ \begin{array}{l} f \cdot r = id_A \\ s \cdot f = id_B \end{array} \right. \\
\Leftrightarrow & \quad \{ \text{ composition and identity } \} \\
 & s \cdot (f \cdot r) = s \\
\Leftrightarrow & \quad \{ \text{ associativity } \} \\
 & (s \cdot f) \cdot r = s \\
\Leftrightarrow & \quad \{ \ s \text{ is right-inverse of } f \text{ and identity } \} \\
 & r = s
\end{array}
$$

In summary, we have the following function terminological taxonomy:



In formal data processing, wherever f has a right-inverse r we say that f is an *abstraction* function (also called a *retrieve map*) and r a (particular choice of a) *representation* function.

4.3.1 Examples

Let us recall

$$B^{C \times A} \underset{\textit{uncurry}}{\overset{\textit{curry}}{\cong}} (B^A)^C$$

and try to extend this to partial functions. Rather than an isomorphism, we get an inequation:

$$(C \times A) \rightharpoonup B \underset{unpcurry}{\overset{pcurry}{\leq}} C \rightharpoonup (A \rightharpoonup B)$$

[4] Notation f^{-1} expresses the uniqueness of the inverse function of an isomorphism f.

where

$$\begin{array}{rcl} pcurry & : & (A \times B) \rightharpoonup C \longrightarrow A \rightharpoonup (B \rightharpoonup C) \\ pcurry\, f & \stackrel{\text{def}}{=} & \left(\begin{array}{c} a \\ \left(\begin{array}{c} \pi_2\, p \\ f\, p \end{array} \right)_{p \in \mathsf{dom}\ f \wedge (\pi_1\, p) = a} \end{array} \right)_{a \in 2^{\pi_1}(\mathsf{dom}\ f)} \end{array} \quad (4.5)$$

and

$$unpcurry \quad \stackrel{\text{def}}{=} \quad \lambda\sigma. \left(\begin{array}{c} (a, b) \\ (\sigma\ a)b \end{array} \right)_{a \in \mathsf{dom}\ \sigma \wedge b \in \mathsf{dom}\ (\sigma\ a)}$$

Clearly,

$$pcurry \cdot unpcurry \quad \neq \quad id$$

Doing the same with respect to $split$ (1.18):

$$A \rightharpoonup (B \times C) \underset{\bowtie}{\overset{unjoin}{\leq}} (A \rightharpoonup B) \times (A \rightharpoonup C) \quad (4.6)$$

where the finite mapping "join" operator (3.40) establishes how $\rightharpoonup$ "distributes" over product $\times$. Its right-inverse is

$$unjoin \quad \stackrel{\text{def}}{=} \quad \langle A \rightharpoonup \pi_1, A \rightharpoonup \pi_2 \rangle \quad (4.7)$$

i.e. `unjoin <- split(*->p1,*->p2)` in CAMILA notation.

The $\leq$-subcalculus encompasses other $\rightharpoonup$-laws which cannot be adapted from similar results concerning exponentials. For instance,

$$(B + C)^A \ncong B^A \times B^C$$

and yet we have the following map-decomposition law:

$$A \rightharpoonup (B + C) \underset{\overset{+}{\bowtie}}{\overset{uncojoin}{\leq}} (A \rightharpoonup B) \times (A \rightharpoonup C) \quad (4.8)$$

where

$$\overset{+}{\bowtie} \quad \stackrel{\text{def}}{=} \quad \dagger \cdot ((A \rightharpoonup i_1) \times (A \rightharpoonup i_2))$$

A very important law is

$$A \rightharpoonup D \times (B \rightharpoonup C) \underset{\bowtie_n}{\overset{unnjoin}{\leq}} (A \rightharpoonup D) \times ((A \times B) \rightharpoonup C) \quad (4.9)$$

which is established by an elaboration of $\bowtie$

$$\bowtie_n \quad \stackrel{\text{def}}{=} \quad \bowtie \cdot \langle \pi_1, \dagger \cdot ((A \rightharpoonup \underline{\underline{\perp}}) \times pcurry) \rangle \tag{4.10}$$

such that

$$\begin{aligned} unnjoin \quad &\stackrel{\text{def}}{=} \quad (id \times unpcurry) \cdot unjoin \\ &= \qquad \{ \text{ definition of } unjoin \text{ (4.7)} \} \\ &\qquad \langle A \rightharpoonup \pi_1, unpcurry \cdot (A \rightharpoonup \pi_2) \rangle \end{aligned}$$

is one of its right-inverses. Other standard representations (refinements) follow: for finite A

$$\mathcal{P}A \underset{dom}{\overset{\leq}{\rightleftarrows}} A \rightharpoonup B \tag{4.11}$$

(representation consists of classifying data, $b_0 \in B$).

$$\mathcal{P}A \underset{ran}{\overset{\leq}{\rightleftarrows}} \mathbb{N} \rightharpoonup A \tag{4.12}$$

(representation consists of indexing data).

$$A^\star \overset{\leq}{\rightleftarrows} \mathbb{N} \rightharpoonup A \tag{4.13}$$

(refinement consists of rendering sequence order explicit).

$$A \rightharpoonup B \overset{\leq}{\rightleftarrows} B \rightharpoonup \mathcal{P}A \tag{4.14}$$

(refinement consists of representing a function by its "kernel").

Exercise 4.1 Guess a formal definition of $uncojoin$.
□

Exercise 4.2 Show that

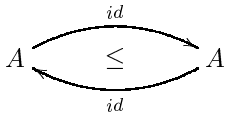
$$A \rightharpoonup (B \rightharpoonup C) \quad \leq \quad (\mathcal{P}A) \times ((A \times B) \rightharpoonup C) \tag{4.15}$$

holds.
□

Exercise 4.3 Infer abstraction functions θ_1 and θ_2 whose righ-inverses are $A \rightharpoonup i_1$ and $A \rightharpoonup i_2$, respectively. Infer the signatures of θ_1 and θ_2. Explain why θ_1 and θ_2 are not the righ-inverses of $A \rightharpoonup i_1$ and $A \rightharpoonup i_2$, respectively.
□

4.3.2 Basic properties of $\leq$:

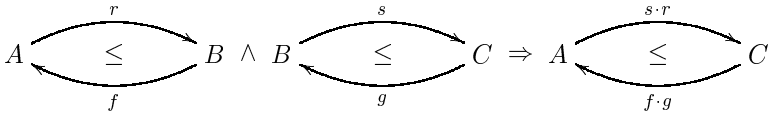
Reflexivity



cf.

$$id \cdot id = id$$

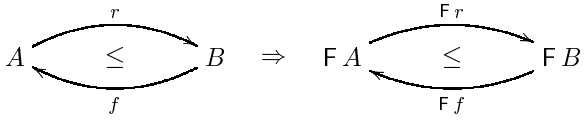
Transitivity



cf.

$$\begin{array}{ll} & (f \cdot g) \cdot (s \cdot r) \\ = & \quad \{ \text{ associativity} \} \\ & f \cdot (g \cdot s) \cdot r \\ = & \quad \{ \ s \text{ is right-inverse of } g \} \\ & f \cdot id \cdot r \\ = & \quad \{ \text{ identity} \} \\ & f \cdot r \\ = & \quad \{ \ r \text{ is right-inverse of } f \} \\ & id \end{array}$$

Monotonicity



where F is an arbitrary functor:

$$\begin{array}{ll} & id \\ = & \quad \{ \text{ functors commute with } id \} \end{array}$$

$$
\begin{array}{ll}
 & \mathsf{F}\,id \\
= & \qquad \{\ r \text{ is right-inverse of } f\} \\
 & \mathsf{F}\,(f \cdot r) \\
= & \qquad \{\ \text{functors commute with composition}\} \\
 & (\mathsf{F}\,f) \cdot (\mathsf{F}\,r)
\end{array}
$$

therefore $\mathsf{F}\,r$ is right-inverse of $\mathsf{F}\,f$. Of course, this result extends to bifunctors, *e.g.*

$$A \underset{f}{\overset{r}{\leq}} B \ \wedge\ C \underset{g}{\overset{s}{\leq}} D \quad \Rightarrow \quad \mathsf{B}(A,C) \underset{\mathsf{B}(f,g)}{\overset{\mathsf{B}(r,s)}{\leq}} \mathsf{B}(B,D)$$

or n-ary functors in general.

4.3.3 Induction (Structural monotonicity)

Given two inductive datatypes $\mu\mathsf{F}$ and $\mu\mathsf{G}$ defined by functors F and G, respectively,

$$\mu\mathsf{F} \underset{in_{\mu\mathsf{F}}}{\overset{\cong}{\leftarrow}} \mathsf{F}\,\mu\mathsf{F}$$

$$\mu\mathsf{G} \underset{in_{\mu\mathsf{G}}}{\overset{\cong}{\leftarrow}} \mathsf{G}\,\mu\mathsf{G}$$

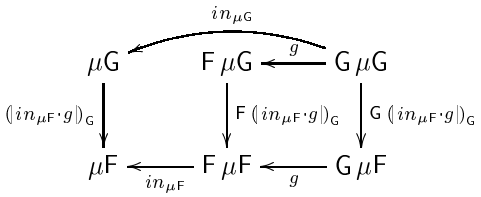
and such that

$$\mathsf{F}\,X \underset{g}{\overset{s}{\leq}} \mathsf{G}\,X \tag{4.16}$$

Then

$$\mu\mathsf{F} \underset{f=(\!|\, in_{\mu\mathsf{F}} \cdot g \,|\!)_{\mathsf{G}}}{\overset{r=(\!|\, in_{\mu\mathsf{G}} \cdot s \,|\!)_{\mathsf{F}}}{\leq}} \mu\mathsf{G} \tag{4.17}$$

holds, *cf.* diagram



This property is very important in data-refinement calculations. Let us see some examples of application:

1. Empty sequences represented by non-empty ones — $\mu\mathsf{F} = A^\star$ and $\mu\mathsf{G} = A^+$, for $1 \leq A$:

$$A^\star \underset{[\,\underline{[\,]},cons\,]}{\overset{\cong}{\leftarrow}} 1 + A \times A^\star$$

$$A^+ \underset{[\,sing,cons\,]}{\overset{\cong}{\leftarrow}} A + A \times A^+$$

   where $sing\, a = [a]$.

   Of course

$$1 + A \times X \underset{!+id}{\overset{\underline{a_0}+id}{\leq}} A + A \times X$$

   holds for some $a_0 \in A$ since

$$1 \underset{!}{\overset{\underline{a_0}}{\leq}} A$$

   by hypothesis. According to (4.17) we infer not only that non-empty lists implement empty lists

$$A^\star \underset{blast}{\overset{embed}{\leq}} A^+$$

   (obvious!) but how they do it:

$$blast = (\!|\,[\,\underline{[\,]}, cons\,] \cdot (! + id)\,|\!)$$
$$embed = (\!|\,[\,sing, cons\,] \cdot (\underline{a_0} + id)\,|\!)$$

   (less obvious). Let us calculate further:

$$\begin{array}{ll} & blast \cdot [\,sing, cons\,] \\ = & \quad \{\text{ cata-cancellation}\} \\ & [\,\underline{[\,]}, cons\,] \cdot (! + id) \cdot (id + id \times blast) \\ = & \quad \{\text{ +-bifunctor and trivia}\} \\ & [\,\underline{[\,]}, cons\,] \cdot (! + id \times blast) \\ = & \quad \{\text{ +-absorption and trivia}\} \\ & [\,\underline{[\,]}, cons \cdot (id \times blast)\,] \end{array}$$

that is

$$\begin{array}{rcl} blast\,[a] & = & [\,] \\ blast\,(cons(a,l)) & = & cons(a, blast\,l) \end{array}$$

($blast$="all but last")

Similarly:

$$\begin{array}{rcl} embed\,[\,] & = & [a_0] \\ embed\,(cons(a,l)) & = & cons(a, embed\,l) \end{array}$$

2. Representing $GenDia$ (genealogical diagrams) by $DecTree$ (decision trees):

Genealogical diagrams:

$$\begin{array}{rlll} GenDia :: & I: & Ind & \textit{/*data about an individual */} \\ & F: & [GenDia] & \textit{/*genealogy of his/her father (if known) */} \\ & M: & [GenDia] & \textit{/*genealogy of his/her mother (if known) */} \end{array}$$

i.e.

$$GenDia \cong Ind \times (GenDia + 1) \times (GenDia + 1) \qquad (4.18)$$

Decision trees:

$$\begin{array}{rcll} DecTree & :: & Q: \; What & \textit{/*Question or Decision */} \\ & & R: \; Answer \rightharpoonup DecTree & \textit{/*Subtrees */} \\ What & = & ... & \\ Answer & = & ... & \end{array}$$

i.e.

$$DecTree \cong What \times (Answer \rightharpoonup DecTree) \qquad (4.19)$$

Clearly,

$$GenDia \quad \cong \quad \mathsf{F}\,(Ind, GenDia)$$

where

$$\mathsf{F}\,(A,X) \stackrel{\text{def}}{=} A \times (X+1) \times (X+1)$$

and

$$DecTree \quad \cong \quad \mathsf{G}\,(What, Answer, DecTree)$$

where

$$\mathsf{G}\,(A, B, X) \stackrel{\text{def}}{=} A \times (B \rightharpoonup X)$$

Reasoning: instantiate $B = 2$. Then

$$\begin{array}{ll} & \mathsf{G}\,(A, 2, X) \\ = & \quad \{ \text{ instantiation} \} \\ & A \times (2 \rightharpoonup X) \\ \cong & \quad \{ \text{ totalizing maps} \} \\ & A \times (X + 1)^2 \\ \cong & \quad \{ \text{ expanding square} \} \\ & A \times ((X + 1) \times (X + 1)) \\ \cong & \quad \{ \text{ flattening (removing parentheses)} \} \\ & A \times (X + 1) \times (X + 1) \\ = & \quad \{ \text{ instantiation} \} \\ & \mathsf{F}\,(A, X) \end{array}$$

Since

$$\mathsf{F}\,(A, X) \underset{}{\overset{}{\rightleftharpoons}} \mathsf{G}\,(A, 2, X) \qquad (\cong)$$

we can deduce:

$$GenDia\ A \underset{}{\overset{}{\rightleftharpoons}} DecTree(A, 2) \qquad (\cong)$$

Challenge: calculate abstraction/representation maps.

3. Last but not least: finite sets represented by finite lists:

$$\mathcal{P}A \underset{elems=(\!|ins|\!)}{\overset{\{\!|in|\!\}}{\rightleftharpoons}} A^\star \qquad (\leq) \tag{4.20}$$

where

$$in = [\, \underline{[\,]}, cons \,]$$

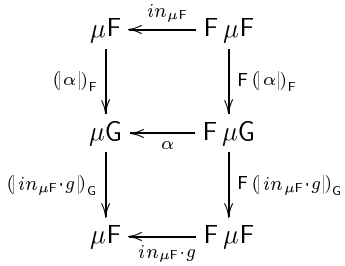
and

$$ins = [\, \underline{\emptyset}, puts \,]$$

— recall (3.2).

4.3.4 Constructive proof

We want to find a right-inverse for $(\!|in_{\mu\mathsf{F}} \cdot g|\!)$ in (4.17) expressible as a catamorphism $(\!|\alpha|\!)$, *cf.* diagram



So α is the unknown. Calculation of α:

$$
\begin{array}{ll}
 & (\!|in_{\mu\mathsf{F}} \cdot g|\!)_{\mathsf{G}} \cdot (\!|\alpha|\!)_{\mathsf{F}} = id \\
\Leftrightarrow & \quad \{ \text{by F-cata-reflexion} \} \\
 & (\!|in_{\mu\mathsf{F}} \cdot g|\!)_{\mathsf{G}} \cdot (\!|\alpha|\!)_{\mathsf{F}} = (\!|in_{\mu\mathsf{F}}|\!)_{\mathsf{F}} \\
\Leftarrow & \quad \{ \text{by F-cata-fusion} \} \\
 & (\!|in_{\mu\mathsf{F}} \cdot g|\!)_{\mathsf{G}} \cdot \alpha = in_{\mu\mathsf{F}} \cdot \mathsf{F}\,(\!|in_{\mu\mathsf{F}} \cdot g|\!)_{\mathsf{G}} \\
\Leftrightarrow & \quad \{ \text{decompose } \alpha = in_{\mu\mathsf{G}} \cdot \alpha' \} \\
 & (\!|in_{\mu\mathsf{F}} \cdot g|\!)_{\mathsf{G}} \cdot in_{\mu\mathsf{G}} \cdot \alpha' = in_{\mu\mathsf{F}} \cdot \mathsf{F}\,(\!|in_{\mu\mathsf{F}} \cdot g|\!)_{\mathsf{G}} \\
\Leftrightarrow & \quad \{ \text{by G-cata-cancellation} \} \\
 & in_{\mu\mathsf{F}} \cdot g \cdot \mathsf{G}\,(\!|in_{\mu\mathsf{F}} \cdot g|\!)_{\mathsf{G}} \cdot \alpha' = in_{\mu\mathsf{F}} \cdot \mathsf{F}\,(\!|in_{\mu\mathsf{F}} \cdot g|\!)_{\mathsf{G}} \\
\Leftrightarrow & \quad \{ g \text{ is natural (4.21)} \} \\
 & in_{\mu\mathsf{F}} \cdot \mathsf{F}\,(\!|in_{\mu\mathsf{F}} \cdot g|\!)_{\mathsf{G}} \cdot g \cdot \alpha' = in_{\mu\mathsf{F}} \cdot \mathsf{F}\,(\!|in_{\mu\mathsf{F}} \cdot g|\!)_{\mathsf{G}} \\
\Leftarrow & \quad \{ g \cdot s = id \} \\
 & \alpha' = s
\end{array}
$$

So

$$\alpha \quad = \quad in_{\mu\mathsf{G}} \cdot s$$

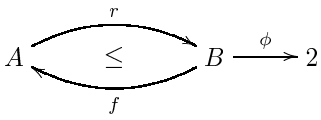
NB: both g and s are natural, ***e.g.***

$$(\mathsf{F}\,f) \cdot g = g \cdot (\mathsf{G}\,f) \qquad \begin{array}{c} A \\ {\scriptstyle f}\downarrow \\ B \end{array} \qquad \begin{array}{ccc} \mathsf{G}\,A & \xrightarrow{g_A} & \mathsf{F}\,A \\ {\scriptstyle \mathsf{G}\,f}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\,f} \\ \mathsf{G}\,B & \xrightarrow[g_B]{} & \mathsf{F}\,B \end{array} \tag{4.21}$$

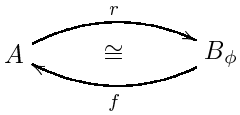
holds.

4.4 Concrete invariants

Given some abstraction map f, the choice of a particular representation r induces a low-level invariant ϕ



called the *concrete invariant* induced by r. It emerges as characteristic predicate of the range of r, that is to say, $\phi\, b$ holds wherever there exists at least one $a \in A$ such that $b = r\ a$. Because r is always injective, one has



where B_ϕ denotes the subset of B which satisfies concrete-invariant ϕ. So the replacement of low-level structure B by abstract structure A is safe provided ϕ is known to hold in the particular situation in hand — a kind of reverse specification via "concrete invariant discharge". That is to say, we have

$$f \cdot r = id$$
$$\phi\, b \stackrel{\text{def}}{=} (r \cdot f)b = b$$

Some examples follow:

$$A \rightharpoonup (B \times C) \underset{\bowtie}{\overset{unjoin}{\cong}} ((A \rightharpoonup B) \times (A \rightharpoonup C))_{eqd}$$

where

$$eqd \stackrel{\text{def}}{=} (=) \cdot (\mathsf{dom} \times \mathsf{dom})$$

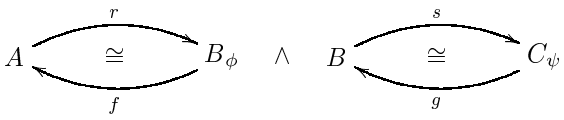
while

$$A \rightharpoonup (B + C) \underset{\overset{+}{\bowtie}}{\overset{uncojoin}{\cong}} ((A \rightharpoonup B) \times (A \rightharpoonup C))_{djd}$$

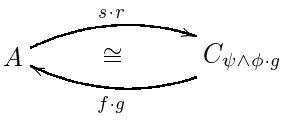
where

$$djd \stackrel{\text{def}}{=} =_\emptyset \cdot \cap \cdot (\mathsf{dom} \times \mathsf{dom})$$

4.4.1 Concrete invariant handling rule



implies



For example, from (3.29) and (4.20) we infer:

$$A \rightharpoonup B \quad \cong \quad (A \times B)^{\star}_{fdp \cdot \mathsf{elems}}$$

4.5 Representing inductive datatypes in ordinary programming languages

4.5.1 The relational database model

A data model is said to be compliant with the *relational database model* if it is a finitary product of expressions of the following kind:

- **relations** — $\mathcal{P}(A_1 \times \times \ldots A_n)$ where every A_i is regarded as an "atomic type" in the target relational environment;
- **mappings** — $(A_1 \times \ldots \times A_n) \rightharpoonup (B_1 \times \ldots \times B_m)$ where every A_i and B_j is regarded as an "atomic type" in the target relational environment.

Then:

- Laws such as (4.6,4.15,4.9,4.8) and (4.13) play an important rôle in derived construct reification. They can be used to "decompose" complex/nested mappings or sequences into tuples of simpler maps.
- Since finite maps are easily refined into binary relations — recall (3.29) — our calculus fragment above is useful in refining elaborate, finite-mapping/sequence based data models into relational database schemata.
- It has been proved elsewhere [3] that **3NF** is guaranteed by mere application of the above laws, among others.

Exercise 4.4 In relational data refinement (*i.e.* the representation of data in the relational database model) one often ignores normalization wherever data storage is not a problem, *i.e.*, one is dealing with small data sets. The following law describes one such refinement step:

$$(A \rightharpoonup B) \times (A \rightharpoonup C) \quad \leq \quad A \rightharpoonup ((B + 1) \times (C + 1)) \tag{4.22}$$

It merges two tables into a single one by, for instance, representing

NÚMERO	NOME
1010	Manuel
11230	Maria

and

NÚMERO	CURSO
11230	LESI
15234	LMCC

by

NÚMERO	NOME	CURSO
1010	Manuel	NIL
11230	Maria	LESI
15234	NIL	LMCC

1. Resort to θ_1 of exercise 4.3 to define an abstration map f for (4.22) in point-free notation.
2. Define the obvious righ-inverse of f.

□

4.5.2 Pointer introduction

For target languages like C/C++ we need a different approach. Recursive data models are coped with by resorting to pointers as shown next. First of all, we have the following basic fact: every non-empty datatype A can be represented by a "pointer":

$$A \underset{f=[\,\underline{a_0},id\,]}{\overset{r=i_2}{\leq}} 1+A \qquad (a_0 \in A)$$

i.e.

$$A \quad \cong \quad (1+A)_{\phi\, x \overset{\text{def}}{=} \exists a \in A : x = (i_2\, a)}$$

(f abstracts both NIL and a_0 into a_0. It is therefore non-injective.) Then we have the following implementation rules:

- Inductive datatypes of form

$$\mathsf{T} \underset{in}{\overset{out}{\cong}} 1+\mathsf{G}\,\mathsf{T}$$

  can be directly implemented in such languages, *e.g.*

$$A^\star \quad \cong \quad 1+\underbrace{A\times A^\star}_{\mathsf{G}\,A^\star}$$

  implemented by

```
typedef struct G {
        A first;
        struct G *next;
        } *T;
```

- Inductive datatypes of form

$$\mathsf{T} \underset{in}{\overset{out}{\cong}} \mathsf{G}\,(1+\mathsf{T})$$

are also directly implementable, *e.g.*

$$GenDia \cong Ind \times (1 + GenDia) \times (1 + GenDia) \qquad (4.23)$$

implemented by

```
typedef struct GenDia {
        Ind individual;        /* data about an individ-
ual */
        struct GenDia *father; /* genealogy of his/her father
                                  (if known) */
        struct GenDia *mother; /* genealogy of his/her mother
                                  (if known) */
        };
```

- All other cases:

$$\mathsf{T} \underset{in}{\overset{out}{\cong}} \mathsf{F}\,\mathsf{T}$$

implemented by forcing a pointer structure:

$$\mathsf{F}\,X \underset{[\ \underline{k}, id\ ]}{\overset{i_2}{\leq}} 1 + \mathsf{F}\,X$$

and therefore (4.17)

$$\mathsf{T} \underset{(\!|\, in \cdot [\ \underline{k}, id\ ]\,|)_{\mathsf{F}}}{\overset{(\!|\, in \cdot i_2\,|)_{\mathsf{G}}}{\leq}} \mathsf{U} \qquad (4.24)$$

where

$$\mathsf{U} \;\cong\; 1 + \mathsf{F}\,\mathsf{U}$$

Example:

$$DecTree \quad \cong \quad What \times (Answer \rightharpoonup DecTree)$$

Via (3.29) this is implemented by

$$DecTree_1 \quad \cong \quad What \times (Answer \times DecTree_1)^\star$$

On its turn, via (4.24), $DecTree_1$ is implemented by

$$DecTree_2 \quad \cong \quad 1 + What \times (Answer \times DecTree_2)^\star$$

that is

$$\begin{cases} DecTree_2 & \cong & 1 + What \times subTrees \\ subTrees & \cong & 1 + (Answer \times DecTree_2) \times subTrees \end{cases}$$

This has a direct encoding in C as follows (for $Answer$ and $What$ = strings):

```
typedef struct DecTree2{
        char *What; /* Question or Decision */
        struct subTrees *R;
        };

typedef struct subTrees {
        struct node *first;
        struct subTrees *next;
        };

typedef struct node {
        char *Answer;
        struct DecTree2 *SubTree;
        };
```

4.5.3 Recursion "removal"

What about SQL-like languages? How do we convert recursive structures in non-recursive tabular ones?

Let

$$X_0 \underset{in}{\overset{\cong}{\rightleftarrows}} \mathsf{F}\, X_0$$

be a solution to

$$X \cong \mathsf{F}\, X \tag{4.25}$$

Implementation step:

$$X_0 \underset{f}{\overset{\cong}{\rightleftarrows}} (K \times (K \rightharpoonup \mathsf{F}\, K))_\phi \tag{4.26}$$

for K a domain of "pointers" such that $K \cong \mathbb{N}$.

Abstraction Function f

We need to define a surjection

$$(K \times (K \rightharpoonup \mathsf{F}\, K))_\phi \xrightarrow{f} X_0$$

that is, to find appropriate f and ϕ.
Temporarily assume that $\sigma \in K \rightharpoonup \mathsf{F}\, K$ is a total function, and draw the diagram of coalgebra (K, σ):

$$\begin{array}{c} K \\ \downarrow{\scriptstyle \sigma} \\ \mathsf{F}\, K \end{array}$$

Assuming a given piece of "linear storage" σ ("database"), let f_σ denote the function which, for each "pointer" $k \in K$, retrieves the value of X_0 corresponding to a σ "scan" starting from k:

$$\begin{array}{ccc} X_0 & \xleftarrow{f_\sigma} & K \\ & & \downarrow{\scriptstyle \sigma} \\ & & \mathsf{F}\, K \end{array}$$

The diagram can be immediately completed,

$$\begin{array}{ccc} X_0 & \xleftarrow{f_\sigma} & K \\ {\scriptstyle in}\uparrow & & \downarrow{\scriptstyle \sigma} \\ \mathsf{F}\, X_0 & \xleftarrow[\mathsf{F}\, f_\sigma]{} & \mathsf{F}\, K \end{array}$$

expressing f_σ as F-hylomorphism $[\![\sigma, in]\!]_{\mathsf{F}}$, that is,

$$f_\sigma = in \cdot (\mathsf{F}\, f_\sigma) \cdot \sigma$$

or

$$\begin{array}{rcl} f & : & K \times (K \rightharpoonup \mathsf{F}\, K) \longrightarrow X_0 \\ f\,(k, \sigma) & \stackrel{\text{def}}{=} & in((\mathsf{F}\, f_\sigma)(\sigma\, k)) \end{array} \tag{4.27}$$

writing $f\,(k, \sigma)$ instead of "curried" $f_\sigma\, k$.

Concrete Invariant ϕ

Face pointer undefinedness in case σ is partial — $\sigma\, k$ in (4.27) is undefined wherever $k \notin \mathit{dom}\, \sigma$ and other pointers k' in the range of σ, reachable from k, may be in the same situation.

- Pointer reachability — transitive closure of

$$k_1 <_{\mathsf{F}} k \quad \stackrel{\text{def}}{=} \quad k \in \mathit{dom}\ \sigma \wedge k_1 \in_{\mathsf{F}} \sigma\ k \tag{4.28}$$

  (recall that $\in_{\mathsf{F}}$ is given in section 3.3), that is

$$\begin{array}{rcl} \phi(k,\sigma) & \stackrel{\text{def}}{=} & let \quad P = \{k\} \cup \{k' \in K \mid k' <_{\mathsf{F}}^{+} k\} \\ & & in \quad P \subseteq \mathit{dom}\ \sigma \end{array}$$

- Well-foundedness — Restrict to least fixpoints, *i.e.* guarantee that $f(k,\sigma)$ does not yield infinite results, forcing P to be well-founded *wrt.* $<_{\mathsf{F}}$:

$$\begin{array}{rcll} \phi(k,\sigma) & \stackrel{\text{def}}{=} & let & P = \{k\} \cup \{k' \in K \mid k' <_{\mathsf{F}}^{+} k\} \\ & & in & P \subseteq \mathit{dom}\ \sigma \wedge \\ & & & \forall \emptyset \subset C \subseteq P : \exists m \in C : \forall k' <_{\mathsf{F}} m : k' \notin C \end{array} \tag{4.29}$$

The proof of surjectiveness of f (4.27) for $(X_0,\ \mathsf{F}\ X_0 \xrightarrow{in} X_0\ )$ a finite or denumerable fixpoint of F, is a lengthy one!

4.5.4 Trivial Example

(C, id_C) is of course a fixpoint of constant functor $\mathsf{F}\ X = C$. Then

$$\begin{array}{rcl} f(k,\sigma) & \stackrel{\text{def}}{=} & in((\mathsf{F}\ f_\sigma)(\sigma\ k)) \\ & = & id_C((C\ f_\sigma)(\sigma\ k)) \\ & = & id_C(id_C(\sigma\ k)) \\ & = & \sigma\ k \end{array}$$

Since $<_{\mathsf{F}}^{+}$ is empty, we have

$$\phi(k,\sigma) \quad \stackrel{\text{def}}{=} \quad k \in \mathit{dom}\ \sigma \tag{4.30}$$

as expected — *cf.* dynamic programming typical of *e.g.* C or PASCAL— instead of handling C data directly (statically stored), the program handles *dynamic* references to them.

4.5.5 Implementing $DecTree$

Recall

$$DecTree \cong What \times (Answer \rightharpoonup DecTree)$$

4.5.6 Data-level Reification

First transformation is recursion removal according to (4.26)

$$DecTree \quad \leq_1 \quad DecTree_1$$

leading to

$$DecTree_1 \quad = \quad K \times (K \rightharpoonup What \times (Answer \rightharpoonup K)) \tag{4.31}$$

What have we achieved? Two interpretations (at least) are admissible:

1. For K a domain of *pointers*,

   $$K \rightharpoonup What \times (Answer \rightharpoonup K) \tag{4.32}$$

   — that is:

   $$K \rightharpoonup What \times (K+1)^{Answer}$$

   — models a heap-segment of dynamic storage. In PASCAL (the heap is "hidden" in the run-time system):

   ```
   type  DecTree1 =  ^DecTree;
         DecTree =   record
                     Q: What;
                     R: array [Answer] of ^DecTree
                     end;
   ```

2. For K a domain of *object names* or "unique identifiers", (4.32) models the object database

   $$name \rightarrow object$$

   implicit in an object-oriented programming environment, *cf.* the following CAMILA:

   ```
   DecTree_1 :: ObjName: K
                Archive: ObjBase ;
   ObjBase   =  K -> Attributes
   Attributes :: Q: What
                 SubObjs: Answer -> K ;
   ```

Proceeding via law (4.9):

$$\begin{array}{rcl} DecTree_1 & = & K \times (K \rightharpoonup What \times (Answer \rightharpoonup K)) \\ & \leq_2 & K \times ((K \rightharpoonup What) \times ((K \times Answer) \rightharpoonup K)) \\ & = & DecTree_2 \end{array} \tag{4.33}$$

For K a state descriptor, $DecTree_2$ accepts the following interpretation:

- $(K \times Answer) \rightharpoonup K$ = *state transition diagram* of a (deterministic) *finite state automaton* ($Answer$ = input stimuli);

- first factor $k \in K$ in (4.33) = current state of the automaton;
- $K \rightharpoonup What$ = semantic table assigning a meaning to each state.

Carrying on our reasoning,

$$\begin{array}{rcl} DecTree_2 & = & K \times ((K \rightharpoonup What) \times ((K \times Answer) \rightharpoonup K)) \\ & \leq_3 & K \times (\mathcal{P}(K \times What) \times \mathcal{P}((K \times Answer) \times K)) \\ & = & DecTree_3 \\ & \cong_4 & K \times (\mathcal{P}(K \times What) \times \mathcal{P}(K \times Answer \times K)) \\ & = & DecTree_4 \end{array}$$

we obtain a final model,

$$DecTree_4 = K \times (\mathcal{P}(K \times What) \times \mathcal{P}(K \times Answer \times K)) \tag{4.34}$$

which is a relational database schema implementing $DecTree$ in terms of two database files where K plays the rôle of a domain of *keys*.
In summary,

$$DecTree \leq_1 DecTree_1 \leq_2 DecTree_2 \leq_3 DecTree_3 \cong_4 DecTree_4$$

4.6 Exercises

Exercise 4.5 Let f and g be two abstraction (*i.e.* surjective) functions. Identify which of the following combination of f and g are also abstraction functions:

$$[\, f, g \,] \tag{4.35}$$
$$f \times g \tag{4.36}$$
$$\langle f, g \rangle \tag{4.37}$$
$$f + g \tag{4.38}$$

□

Exercise 4.6

1. Apply law (4.17) to $\mathsf{T} = \mathbb{N}$ — *i.e.* $in_{\mathsf{F}} = [\underline{0}, succ]$ —, $\mathsf{U} = A^\star$ and $f = id + \pi_2$. Calculate a pointwise definition of abstraction function $l = (\!|in_{\mathsf{F}} \cdot f|\!)$. Which well-known function is l, after all?
2. Knowing that $\mathsf{U} \cong \mathsf{T}$ holds wherever f é iso (bijective) show that

   $$\mathbb{N} \cong 1^\star$$

   and

   $$1 \cong 0^\star$$

   hold.

□

Exercise 4.7 Show that $untot$ can be defined by

$$untot\,\sigma \quad \stackrel{\text{def}}{=} \quad ((\overline{pap}\,\sigma)+!_A)\cdot(\lambda a.a \in \mathit{dom}\,\sigma)?$$

for

$$pap(\sigma, a) \quad \stackrel{\text{def}}{=} \quad \{\!|\ a \in \mathit{dom}\,\sigma \quad \Rightarrow \quad \sigma\,a\ |\!\}$$

□

Exercise 4.8 The *feature bundle* is a parametric datatype

$$\mathsf{FB}(A, B) \quad = \quad A \rightharpoonup (B + \mathsf{FB}(A, B))$$

which is widely useful in specifications, *e.g.*

$$FileSystem \quad = \quad \mathsf{FB}(FileName, File)$$

1. Define algebra

$$\mathcal{P}FileName \xleftarrow{\ g\ } A \rightharpoonup (B + \mathcal{P}FileName)$$

so that $\{\!|g|\!\}$ specifies the function which collects all filenames in a hierarchical $FileSystem$.

2. FBcan be used to specify "universal" data-representation media, *e.g.* the abstract interface data description language IDL:

$$IDL \quad = \quad \mathsf{FB}(STR, STR)$$

Calculate a C/C++ pointer-based implementation of IDL.

3. Knowing that representation

$$B^\star \underset{f}{\overset{r}{\leftrightarrows}} \mathsf{FB}(\mathbb{N}, B) \qquad (\leq)$$

holds, express r as a list-catamorphism.

4. Resort to (4.17 in order to calculate f and r in the obvious facts

$$A \rightharpoonup B \underset{f}{\overset{r}{\leftrightarrows}} \mathsf{FB}(A, B) \qquad (\leq)$$

$$\mathcal{P}A \underset{f}{\overset{r}{\leftrightarrows}} \mathsf{FB}(A, 1) \qquad (\leq)$$

5. Can you represent $A \times B$ or $A + B$ in FB?

□

Exercise 4.9 Let

$$A + A \times X \xleftarrow{g=id+\pi_2} A$$

in (4.16). What abstraction function f do you obtain by (4.17) in this case? Justify.

□

Bibliography

[1] C. J. Rodrigues. Sobre o desenvolvimento formal de bases de dados. Master's thesis, University of Minho, 1993. (In Portuguese).