Generic Data Processing: A Normalization Exercise

†Claudia Necco, ‡J. Nuno Oliveira,

† Departamento de Informática - Facultad de Cs. Físico Matemáticas y Naturales
Universidad Nacional de San Luis
Ejército de los Andes 950 - 5700 San Luis - Argentina

‡ Departamento de Informática, Universidade do Minho
4700-320 Braga - Portugal

Abstract

This paper describes an exercise in generic data *normalization theory* using a data reification calculus based on the categorial approach to datatypes.

We develop a relational data model in a functional language and then use transformations to refine it. The exercise removes data redundancy in close similarity to conventional relational techniques, which extract functional dependences by schema decomposition [Mai83]. Finally the model is extended using the principles of generic programming, suggesting how to scale up normalization theory to arbitrary data.

Keywords: generic programming, polytypic programming, functional programming, program calculation, data reification.

1 Introduction

In relational database programming [Cod72] one models real-life facts as tuples which are recorded in large, mutually dependent persistent data-sets which are subject to intensive search and processing in order to gather knowledge about a particular application domain. The need for larger and larger data-sets calls for the integration of disparate data-models (*data warehousing*); routine data inspection gives place to *data-mining*, and sophisticated on-line analytical processing (OLAP) replaces manual consolidation of data.

In the past, *textual data* has by-and-large been out of this data processing trend, due to lack of structure and a too strong technology bias [1]. This has changed recently when, with the advent of the INTERNET, open mark-up textual standards eventually gained wide acceptance and world-wide prominence. Text processing has always been a privilege of the grammar theorist, the language analyst or compiler writer, as well as fertile ground for PERL and AWK scripting. However, how does one combine this with "flat" data mining and OLAP technologies? Can these be scaled-up to arbitrarily structured textual documents? Is there room for a single, generic theory of data calculation able to cope with such *heterogeneous* data sources (*e.g.* text, semi-structured data, tabular data, *etc.*)?

These questions call for the unification of (from the relational side) normalization, browsing, analytical processing with the universal-morphism approach which underlies the calculation theory of generic programming. For instance, the relational *join/unjoin* operators can be regarded as relational instantiations of polytypic functions *zip* and *unzip*, respectively, see [JJ98] and [Oli98a, NSO99].

However heterogeneous a data source may happen to be, if it is "structured" this means that it has the shape of an inductively finite data structure (*e.g.* a finite set or list, a finitely branching tree or a combination thereof). Inductive datatypes are expressible in generic programming as fixpoints of appropriate (regular) functors [BdM97]. So the main task appears to be that of generalizing the functorial constructs which describe relational database types to arbitrary regular functors and *see what happens.* Of course we have to broaden our view of functional programming to that of generic (polytypic) programming [Bo98].

The main contribution of this paper is to show an illustrative example of generic normalization, using generic functional programming in a *pointfree* style and program calculation based on categorial datatype theory.

The structure of the paper as follows: Section 2 contains a very brief introduction to formal modeling. Section 3 introduces elementary concepts and terminology which are used throughout the paper. Section 4 sketches a formal model for database relational data. Section 5 presents the generic normalization exercise [2]. The last two sections present some conclusions and future work.

2 Formal Methods and Program Calculation

Formal methods aim at driving software production into good engineering standards by splitting software production into a *specification phase*, in which a mathematical model is built of the contractor requirements, followed by an *implementation phase* in which such a model is somehow converted into a runnable software artifact. Formal methods research shows that implementations can be effectively *calculated* from specifications [Mor90, Oli90, Oli92]. So, in a sense, software technology is becoming a mature discipline in its adoption of the "universal problem solving" strategy which one is taught at school:

- understand the problem
- build a mathematical model of it
- reason in such a model

[1] Think of the variety of text editors still in existence today.
[2] This has been carried out in the context of the first author's Master's thesis [Nec02].

- upgrade it, wherever necessary
- calculate a final solution and implement it.

The sophistication of this strategy is only dependent on the underlying mathematics. In the context of software calculi, data manipulation is based on solving systems of (recursive) equations on domain spaces, up to isomorphism. This entails the definition of data transformations which can be expressed functionally and animated using a functional programming language such as Haskell [Tho96].

There are two basic styles for expressing functions: the *pointwise* style and the *pointfree* style. In the former, functions are described by applying them to arguments ("points"). In the latter one describes functions exclusively in terms of functional combinators. Thanks to the algebra of such combinators [BdM97], the pointfree style leads to a very effective method for reasoning about functions, which is based on elementary category theory and is adopted in this paper. A few concepts in the field are summarized below.

3 Categorical Support

Categories. A category consists of a collection of objects and a collection of arrows. Each arrow $f :: a \rightarrow b$ has a source object a and a target object b. Two arrows f and g can be composed to form a new arrow $g \cdot f$, if f has the same target object as the source object of g. This composition operation is associative. Furthermore, for each object a there is a so-called identity arrow $id_a :: a \rightarrow a$, which is the unit of composition.

Our base category is called *Types* and has types as objects and functions as arrows. Arrow composition is function composition (.) and the identity arrows are represented by the polymorphic function *id*.

Functors. Functors are structure-preserving mappings between categories. *Polymorphic* datatypes are functors from *Types* to *Types*. In Haskell, functors can be defined by a type constructor f of kind $* \rightarrow *$, mapping objects to objects, together with a higher-order function *fmap*, mapping arrows to arrows. This is provided as a constructor *class* in the Haskell *Prelude* (the standard file of primitive functions) as follows:

$$\begin{array}{l} class\ Functor\ f\ where \\ \quad fmap :: (a \rightarrow b) \rightarrow (f\ a \rightarrow f\ b) \end{array}$$

The arrow action of a functor must preserve identity arrows and distribute over arrow composition. For functors from *Types* to *Types*, this means that the following equations must hold:

$$\begin{array}{l} fmap\ id = id \\ fmap\ (f \cdot g) = (fmap\ f) \cdot (fmap\ g) \end{array}$$

Bifunctors. The *product category Types* $\times$ *Types* consists of pairs of types and pairs of functions. We can define functors from *Types* $\times$ *Types* to the base category *Types* in Haskell. These functors are called *bifunctors*. A (curried) bifunctor in Haskell is a type constructor of kind $* \rightarrow * \rightarrow *$, together with a function *bmap*. The following constructor class *Bifunctor* was made available:

$$\begin{array}{l} class\ Bifunctor\ f\ where \\ \quad bmap :: (a \rightarrow c) \rightarrow (b \rightarrow d) \rightarrow (f\ a\ b \rightarrow f\ c\ d) \end{array}$$

Products. Categorical *products* are provided in Haskell by the type constructor for pairs (a, b) (usually written as Cartesian product $a \times b$ in mathematics) and projections fst and snd (resp. π_1 and π_2 in standard mathematical notation). Type constructor $(,)$ is extended to a bifunctor in the obvious way:

$$\begin{array}{l} instance\ BiFunctor\ (\ ,)\ where \\ \quad bmap\ f\ g = f \times g \end{array}$$

where

$$\begin{array}{l} (\times) :: (a \to b) \to (c \to d) \to (a, c) \to (b, d) \\ (f \times g) = split\ (f \cdot fst)\ (g \cdot snd) \end{array}$$

and combinator $split :: (a \to b) \to (a \to c) \to a \to (b, c)$ behaves as follows: $split\ f\ g\ x = (f\ x, g\ x)$.

Sums. Categorical *sums* are defined in the Haskell *Prelude* by means of type constructor

$$data\ Either\ a\ b = Left\ a \mid Right\ b$$

together with a function $either :: (a \to b) \to (c \to b) \to Either\ a\ c \to b$ satisfying the following equations:

$$\begin{array}{l} (either\ f\ g) \cdot Left\ \ = f \\ (either\ f\ g) \cdot Right = g \end{array}$$

Type constructor *Either* is extended to a bifunctor by providing the following instance of *bmap*:

$$\begin{array}{l} (+) :: (a \to b) \to (c \to d) \to Either\ a\ c \to Either\ b\ d \\ (f + g)\ (Left\ a) = Left\ (f\ a) \\ (f + g)\ (Right\ b) = Right\ (g\ b) \end{array}$$

$$\begin{array}{l} instance\ BiFunctor\ Either\ where \\ \quad bmap\ f\ g = f + g \end{array}$$

The popular notations $\langle f, g\rangle$, $[f, g]$ and $\mathsf{F}\ f$ (where F is a functor) will be adopted interchangeably with $split\ f\ g$, $either\ f\ g$ and $fmap\ f$, respectively.

Invertible arrows. An arrow $f :: b \to a$ is said to be *right-invertible* (vulg. surjective) if there exists some $g :: a \to b$ such that $f \cdot g = id_a$. Dually, g is said to be *left-invertible* (vulg. injective) if there exists some f such that the same fact holds. Then type b is said to "represent" type a and we draw:

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

where g and f are called resp. the *representation* and *abstraction* functions. An isomorphism $f :: b \to a$ is an arrow which has both a right-inverse g and a left-inverse h — a *bijection* in set theory terminology. It is easy to show that $g = h = f^{-1}$. Type a is said to be *isomorphic* to b and one writes $a \cong b$.

Isomorphisms are very important functions because they convert data from one "format" to another format losing information. These formats contain the same "amount" of information, although the same datum adopts a different "shape" in each of them. Many isomorphisms useful in data manipulation can be defined [Oli98a], for instance function $swap\ :: (a\ ,\ b) \to (b\ ,\ a)$ which is defined by $swap = \langle \pi_2, \pi_1\rangle$ and establishes the *commutative property* of product, $a \times b \cong b \times a$.

4 Modeling Relational Data

Collective datatypes. Our model of relational data will be based on several families of abstractions, including collective datatypes such as finite *powersets* ($\mathcal{P}a$) and finite partial *mappings* ($a \rightharpoonup b$). These are modeled as Haskell *polymorphic algebraic types* (that is, algebraic type definitions with type variables) based on finite lists $[a]$, see $Set\ a$ and $Pfun\ a\ b$ in Table 1, respectively. Both abstractions contain an equality relation and an ordering relation. The latter instantiates to set inclusion ($\subseteq$) and partial function definedness, respectively.

The finite sets model assumes invariant $\phi\ (Set\ l) \stackrel{\mathrm{def}}{=} length\ l = card(elems\ l)$, where $length$ is Haskell standard and $card$(inal) and $elem$(ent)s have the usual set-theoretical meaning. Partial mappings require an extra invariant ensuring a functional dependence on sets of pairs [3]:

$$fdp \stackrel{\mathrm{def}}{=} (\subseteq \{1\}) \cdot rng \cdot (id \rightharpoonup card) \cdot collect \qquad (1)$$

Table 1 summarizes the Haskell modules defined for these datatypes.

	Finite Sets	**Partial Functions**
Datatypes:	$data\ Set\ a = Set\ [a]$	$data\ Pfun\ a\ b = Map[(a,b)]$
Constructors:	$emptyS, sings, puts, prods$ $ltos$	$bottom, singpf, putpf$ $collect$
Deletions:	$gets$	$getpf$
Observers:	$ins, nins, incls, card$ $allS$	$compatible, incompatible$ $allPf$
Filters:	$filterS$	
Operations:	$inters, unions, diffs, plus, pfzip$ $flatr, flatl, slstr, srstr, sextl, sextr$ $zipS, zipWithallS$	$plus, pfinv, restn, restp$ $pfzip, pfzipWith$
Folds:	$foldS$	$foldPf$
Functor:	$fmapS$	
Bifunctor:		$bmapPf$
Others:	$the, stol, elems, card$ $unzipS$	$dom, rng, aplpf$ $tnest, discollect, mkr, bpfTrue, bpfFalse$ $pfunzip$

Table 1: Finite sets and partial functions: datatypes and functions implemented.

Relational Database Model. An n-ary relation in mathematics is a subset of a finite n-ary product $A_1 \times \ldots \times A_n$, which is inhabited by n-ary vectors $\langle a_1, \ldots, a_n \rangle$. Each entry a_i in vector $t = \langle a_1, \ldots, a_n \rangle$ is accessed by its position's projection $\pi_i : A_1 \times \ldots \times A_n \to A_i$. This, however, is not expressive enough to model relational data as this is understood in database theory [Mai83]. Two ingredients must be added, whereby *vectors* give place to *tuples*: attribute names and NULL values. Concerning the former, one starts by rendering vectorial indices explicit, in the sense of writing *e.g.* $t\,i$ instead of $\pi_i\,t$. This implies merging all datatypes A_1 to A_n into a single coproduct type $A = \sum_{i=1}^{n} A_i$ and then represent the n-ary product as :

$$\begin{tikzcd} A_1 \times \ldots \times A_n \arrow[r, bend left, "r"] \arrow[r, phantom, "\leq"] & (\sum_{i=1}^{n} A_i)^n \arrow[l, bend left, "f"] \end{tikzcd}$$

[3] $collect :: \mathcal{P}(a \times b) \to (a \rightharpoonup \mathcal{P}b)$ converts a relation into a set-valued partial function and $rng :: (a \rightharpoonup b) \to \mathcal{P}b$ is the usual *range* function.

under representation function [4] $r\,\langle a_j\rangle_{j=1..n} \overset{\text{def}}{=} \lambda j.(i_j\ a_j)$ which entails invariant

$$\phi\, t \quad \overset{\text{def}}{=} \quad \forall j = 1, \ldots, n, t\, j = i_j\, x \ : \ x \in A_j$$

Note that $j = 1, \ldots, n$ can be written $j \in \overline{n}$, where $\overline{n} = \{1, \ldots, n\}$ is the initial segment of the natural numbers induced by n. Set $\overline{n}$ is regarded as the *attribute* name-space of the model [5].

As a second step in the extension of vectors to tuples, we consider the fact that some attributes may not be present in a particular tuple, that is, NULL values are allowed [6]:

$$(\sum_{i\in\overline{n}} A_i + 1)^n$$

which finally leads to tuples as inhabitants of

$$Tuple \quad = \quad (\overline{n} \rightharpoonup \sum_{i\in\overline{n}} A_i)$$

thanks to isomorphism $A \rightharpoonup B \ \cong \ (B+1)^A$ [Oli90]. This models tuples of arbitrary arity (up to n attributes), including the empty tuple. For notation economy, for every $X \subseteq \overline{n}$, we will write $Tuple_X$ as a shorthand for $X \rightharpoonup \sum_{i\in X} A_i$.

Tuple is the basis for the Haskell model of database relations presented in Table 2. Relations (*Relation*) are sets of tuples sharing a common attribute schema (*SchemaR*). A rather complex invariant ensuring that tuples are well and consistently typed is required, which is omitted here for economy of presentation. This and other details of this model can be found in [Nec02].

	Relations
Datatypes:	*type Tuple = Pfun IdAttr Value* *type SchemaR = Pfun IdAttr AttrInfo* *type IdAttr = String* *type Tuples = Set Tuple* *data Relation = Rel { schema::SchemaR, tuples::Tuples}* *data AttrInfo = InfA { ifKey::Bool, defaultV::Value }* *data Value = Int Int | String String | Date String | Time String*
Constructors:	*emptyR*
Operations:	$unionR, interR, diffR$ $projectR, selectR, natjoinR, equijoinR, renameR, divideR$

Table 2: Relations: datatypes and functions implemented.

5 Exercise in Generic Normalization

Our main target in the exercise which follows is to illustrate the scaling up of *normalization theory* when it is applied under the principles of generic programming.

Normalization through Decomposition. Consider the following relational database schema

$$S \quad = \quad \{\langle r = \{\alpha, \beta, \gamma, \delta\}, \alpha\beta\rangle\} \tag{2}$$

[4]Injections $i_{j=1,n}$ are associated to the n-ary coproduct. *Left* and *Right* in Haskell correspond to i_1 and i_2, respectively.

[5]The fact that this can be replaced by any isomorphic collection of attribute names of cardinality n has little impact in the modelling, so we stick to $\overline{n}$..

[6]Think of 1 as the singleton type {NULL}.

consisting of one table or relation r involving four attributes α, β, γ and δ. $\alpha\beta$ is a key for this relation and it must also satisfy functional dependence $\alpha \to \delta$, such as in, *e.g.*, the following tabular illustration:

α	β	γ	δ
$a1$	$b1$	$c1$	$d1$
$a1$	$b2$	$c2$	$d1$
$a2$	$b3$	$c1$	$d2$

(3)

We would like to update the relation by specifying values for the key and then giving values for the remaining attributes. However, if we perform [7]

$$CH(r; a1, b1; \gamma = c1, \delta = d3)$$

the relation will violate FD $\alpha \to \delta$. To avoid violations of this kind, every time an update is made, one has to scan the whole relation and update the δ value everywhere the α value occurs, despite the fact that only one tuple was to be changed. This happens because the (α-value, δ-value) information is duplicated in the relation, thus making the data redundant. We are better off, with respect to updates and redundancy, if we represent the same information as a database of two relations, r_1 and r_2, as shown below:

α	δ
$a1$	$d1$
$a2$	$d2$

$+$

α	β	γ
$a1$	$b1$	$c1$
$a1$	$b2$	$c2$
$a2$	$b3$	$c1$

(4)

We can retrieve the original relation r by taking $r_1 \bowtie r_2$. The update anomaly no longer exists, since only one tuple needs to be updated to change a δ assignment. We have also removed some data redundancy, since (α-value, δ-value) pairs are recorded once. This process which extracts functional dependences is known as *Normalization through Decomposition* [Mai83, RO97].

In general, the set-theoretical model of S (2) using vector types is

$$\mathcal{P}(A \times B \times C \times D)_\phi \tag{5}$$

where $\mathcal{P}$ is the powerset functor and $A, \ldots, D$ are the types which are inhabited by the values of attributes $\alpha, \ldots, \delta$. Invariant ϕ records the $\alpha \to \delta$ functional dependence, recall (1):

$$\phi \stackrel{\text{def}}{=} fdp \cdot \mathcal{P}\langle \pi_1, \pi_4 \rangle$$

Our task now is as follows. First of all, we want to show how to extract the $\alpha \to \delta$ functional dependence from (5) by calculation. That is, we want to calculate a pair (rf, af) of abstraction/representation functions witnessing the inequality which follows:

$$\begin{tikzcd}
\mathcal{P}(A \times B \times C \times D)_\phi \arrow[r, bend left, "rf"] \arrow[r, phantom, "\leq"] & (A \rightharpoonup D) \times \mathcal{P}(A \times B \times C) \arrow[l, bend left, "af"]
\end{tikzcd} \tag{6}$$

In the illustration above, this means to decompose table (3) in two tables (4).

Secondly, we want to show that such a decomposition is not a privilege of the relational (tabular) information model and that it can in fact be generalized to any other collective datatype T,

$$\mathsf{T}(A \times B \times C \times D)_\phi \ \leq \ (A \rightharpoonup D) \times \mathsf{T}(A \times B \times C) \tag{7}$$

[7] $CH(...)$ is the *change* operation, used to modify only part of a tuple (see [Mai83], page 8).

under a generalized invariant

$$\phi \stackrel{\text{def}}{=} fdp \cdot setify \cdot \mathsf{T}\langle \pi_1, \pi_4 \rangle$$

where $\mathcal{P}X \xleftarrow{setify} \mathsf{T}X$ is the polytypic operation which collects all data from the nodes of a T-structure (of course, $setify = id$ for $\mathsf{T} = \mathcal{P}$).

The transformation of the left-hand side of (6) into its right-hand side was carried out for *vectors* and then extended to *tuples* using the categorial structure of the model [Nec02]. In order to carry out calculations at the generic functor level (7), it is necessary to extend the model to support generic functions.

For economy of presentation we shall skip the first step (vector model) and focus on the calculation performed over *tuples* (Table 2) and its generalization.

Calculating with Tuples. The calculation which is sketched below is a pointfree version of the conventional relational technique which extracts functional dependences by schema decomposition [Mai83]. We want to show that

$$\mathcal{P}(Tuple_{\overline{n}})_{\phi_{k,s}} \leq (Tuple_{k} \rightharpoonup Tuple_{s-k}) \times \mathcal{P}(Tuple_{\overline{n}-(s-k)}) \tag{8}$$

holds, where s is the substructure s that contains a functional dependence, k is the corresponding key (thus $k \subseteq s \subseteq \overline{n}$ holds) and invariant $\phi_{k,s}$ is parametric on s and k.

The calculation goes as follows (see below an account of the functions involved):

$$\begin{array}{ll}
 & \mathcal{P}(Tuple_{\overline{n}})_{\phi_{k,s}} \\
\cong & \qquad \{\ \mathbf{rf_1} = \mathcal{P}(tnest\ k); \quad \mathbf{af_6} = \mathcal{P}(plus)\} \\
 & \mathcal{P}(Tuple_{k} \times Tuple_{\overline{n}-k}) \\
\leq & \qquad \{\ \mathbf{rf_2} = collect; \quad \mathbf{af_5} = discollect\} \\
 & Tuple_{k} \rightharpoonup \mathcal{P}(Tuple_{\overline{n}-k}) \\
\cong & \qquad \{\ \ \mathbf{rf_3} = id \rightharpoonup (sextl \cdot \mathcal{P}(tnest\ s)); \quad \mathbf{af_4} = \mathcal{P}(plus) \cdot slstr\ ;\ (\overline{n}-k) \cap s = s-k\ \} \\
 & Tuple_{k} \rightharpoonup Tuple_{s-k} \times \mathcal{P}(Tuple_{\overline{n}-s}) \\
\leq & \qquad \{\ \mathbf{rf_4} = pfunzip; \quad \mathbf{af_3} = pfzip\} \\
 & (Tuple_{k} \rightharpoonup Tuple_{s-k}) \times (Tuple_{k} \rightharpoonup \mathcal{P}(Tuple_{\overline{n}-s})) \\
\cong & \qquad \{\ \mathbf{rf_5} = id \times discollect; \quad \mathbf{af_2} = id \times collect\ \} \\
 & (Tuple_{k} \rightharpoonup Tuple_{s-k}) \times \mathcal{P}(Tuple_{k} \times Tuple_{\overline{n}-s}) \\
\cong & \qquad \{\ \mathbf{rf_6} = id \times \mathcal{P}(plus); \quad \mathbf{af_1} = id \times tnest\ k\ ;\ (\overline{n}-s) \cup k = \overline{n}-(s-k)\ \} \\
 & (Tuple_{k} \rightharpoonup Tuple_{s-k}) \times \mathcal{P}(Tuple_{\overline{n}-(s-k)})
\end{array}$$

The *collect* $:: \mathcal{P}(a \times b) \rightarrow (a \rightharpoonup \mathcal{P}b)$ injection and its left-inverse *discollect* are as defined in [Oli90]. The tuple splitting operation *tnest* $:: \mathcal{P}a \rightarrow (a \rightharpoonup b) \rightarrow (a \rightharpoonup b) \times (a \rightharpoonup b)$ is central to the calculation in performing isomorphism $A \rightharpoonup B \cong (K \rightharpoonup B) \times ((A-K) \rightharpoonup B)$ which holds wherever $K \subseteq A$ [Oli90]. The calculation also involves *slstr* $:: a \times \mathcal{P}b \rightarrow \mathcal{P}(a \times b)$, the *left-strength* of the powerset functor, and its inverse *sextl*. Finally, $pfzip, pfunzip$ are the finite mapping counterparts of $zip, unzip$ [JJ98]. The Haskell implementation of all these functions can be found in [Nec02].

Going generic. Let us now see how to generalize $\mathcal{P}$ in the calculation above to a generic (strong) functor T and how to model this in Haskell by using higher-order polymorphism and constructor classes. First, the class of strong functors is defined as a subclass of *Functor*,

```
class Functor f ⇒ Strong f where
        rstr :: (f a, b) → f(a, b)
        lstr :: (b, f a) → f(b, a)
```

exporting both a left and a right strength (resp. *rstr* and *lstr*) whose default implementations are

```
rstr(t, x) = fmap (split id (const x )) t
lstr(x, t) = fmap (split (const x ) id) t
```

Then subclass *Poly* of *Strong* is defined, whose operations include those required by the process of normalization (note the character "p" (=polytypic) prefixing in each function symbol):

```
class Strong t ⇒ Poly t where
  – signatures
  pzipWith :: (a → b → c) → t a → t b → t c
  punzip :: t(a, b) → (t a , t b)
  pzip :: (t a, t b) → Maybe(t(a, b))
  pflatten  :: t a → [a]
  pelems   :: Eq a ⇒ t a → Set a
  pdiscollect :: (Eq (t (Maybe (b, c))), Eq b) ⇒ Pfun b (t (Maybe c)) → t (b, c)
  pcollect  :: Eq a ⇒ t (a, b) → Pfun a (t (Maybe b))
  – default implementations
  pelems   = Set · nub · pflatten
  pdiscollect x = foldS (pzipWith (plus))u y
     where y = (fmapS(λ(z, y) → fmap (splus z) y) · mkr)x
           u = (the · (fmapS (fmap (const (Map [ ])))) · rng)x
  pcollect t  = Map (map (split id (flip extr t)) l)
    where l = (nub · pflatten · (fmap fst)) t
  punzip = split (fmap fst) (fmap snd)
```

Note that *pzip* is among the functions which have no default implementation, because the "zip" process fails (cf. *Maybe*) wherever the shape of both arguments is not the same [Jan00] and this can only be checked when the structure of the instance type is known. It has thus to be provided for every instance of the class, a fact which motivates the brief discussion which follows.

6 Discussion

We have chosen to implement our generic (polytypic) normalization operations using higher-order polymorphism and constructor classes in Haskell. The class system allows for overloaded functions but programs become cluttered with instance declarations and type declarations become cluttered with contexts.

Two other ways to implement polytypic programs in a typed language are: *(a)* to use a universal datatype; *(b)* to use special syntactic constructs.

In the first alternative, an universal datatype is chosen on which we define the functions we want to have available for large classes of datatypes. These polytypic functions can be used on a specific datatype if we provide translation functions to and from the universal datatype. An advantage of this approach is that we do not need a language extension for writing polytypic programs. However, it has several disadvantages: type information is lost in the translation phase to the universal datatype, and type errors can occur when programs run. Furthermore, different people will use different universal datatypes, which will make program reuse more difficult.

PolyP [Jan00] is an example of the second alternative, that of extending the language with explicit syntactic constructs for defining polytypic functions. We have ported our experiment to the PolyP system but soon realized that there was a serious shortcoming: only *unary* (regular) datatypes can be polytypically defined. We hope to overcome this problem by switching to Generic Haskell [CHJ+01].

7 Concluding Remarks and Future Work

Productivity and scientific progress in the software development technology is often hindered by artificial, "application domain" border-lines which prevent cross-fertilization of results and the even spread of novelty. Such frontiers often have an academic, social or cultural bias. For instance, the average database programmer will regard *functional programming* as too academic and perhaps useless. Conversely, a functional programmer will regard *database programming* as a too specific and not sufficiently exciting topic.

However, these two research areas have more in common than it appears at first sight. Both put emphasis on the rÙle of *data structuring* in software development and both have developed their own calculus. Can these two seemingly disparate notations and calculi me merged together? This is the question which has motivated the present paper, which describes research in the intersection between formal methods and relational database theory. Our experiments (which include a generic version of *mda*, the multi-dimensional analysis Olap functionality [Nec02]) suggest that there is a more general theory of data normalization of which the standard relational database theory appears to be a particular case.

Of course, a lot of work remains to be done in this evolution of standard normalization theory towards genericity, in particular concerning a proper formalization of the intuitions presented in this paper. The interplay between datatype construction and datatype constraining (cf. datatype *invariants*) still requires a proper formalization in the pointfree style. And, as can be found in our calculation and is noted in [Oli90], many transformation rules are invariant-sensitive. The "pullback approach" of [Oli98b] is an attempt in this direction which proves to be insufficient in practice. Currently we are exploiting a far more promising approach, that of modelling constrained datatypes (subject to invariants) by *coreflexive* relations and performing the reasoning in the (pointfree) *relational theory of datatypes* [BdM97, Bac00].

References

[Bac00] R. C. Backhouse. Fixed point calculus, 2000. Summer School and Workshop on Algebraic and Coalgebraic Methods in the Mathematics of Program Construction, Lincoln College, Oxford, UK 10th to 14th April 2000.

[BdM97] R. Bird and O. de Moor. *Algebra of Programming*. Series in Computer Science. Prentice-Hall International, 1997. C. A. R. Hoare, series editor.

[Bo98] R.C. Backhouse and T. Sheard (org.). WGP'98 — Workshop on Generic Programming, 1998. Marstrand, Sweden, 18th June, 1998 (`http://www.cse.ogi.edu/PacSoft/conf/wgp/`).

[CHJ+01] D. Clarke, R. Hinze, J. Jeuring, A. Löh, and J. de Wit. The generic haskell user's guide, November 2001. Technical Report UU-CS-2001-26, Universiteit Utrecht.

[Cod72] E. F. Codd. Relational completeness of database sublanguages. In *Data Base Systems*, pages 65–98. Prentice-Hall, 1972. Courant Inst. Computer Science Symp. 6, Englewood Cliffs, NJ.

[Jan00] P. Jansson. *Functional Polytypic Programming.* PhD thesis, Chalmers University og Technology and Götebord University, 2000.

[JJ98] P. Jansson and J. Jeuring. Polylib — a library of polytypic functions. In *Workshop on Generic Programming (WGP'98), Marstrand, Sweden*, 1998.

[Mai83] D. Maier. *The Theory of Relational Databases.* Computer Science Press, 1983. ISBN 0-914894-42-0.

[Mor90] C. Morgan. *Programming from Specification.* Series in Computer Science. Prentice-Hall International, 1990. C. A. R. Hoare, series editor.

[Nec05] C. Necco. Polytypic data processing, may 2005. Master's thesis (Facultad de Cs. Físico Matemáticas y Naturales, University of San Luis, Argentina).

[NSO99] F. L. Neves, J. C. Silva, and J. N. Oliveira. *Converting Informal Meta-data to VDM-SL: A Reverse Calculation Approach* . In *VDM in Practice! A Workshop co-located with FM'99: The World Congress on Formal Methods, Toulouse, France, 20-21 September*, September 1999.

[Oli90] J. N. Oliveira. A reification calculus for model-oriented software specification. *Formal Aspect of Computing*, 2(1):1–23, April 1990.

[Oli92] J. N. Oliveira. *Software Reification using the SETS Calculus* . In *Proc. of the BCS FACS 5th Refinement Workshop, Theory and Practice of Formal Software Development, London, UK*, pages 140–171. Springer-Verlag, 8–10 January 1992. (Invited paper).

[Oli98a] J. N. Oliveira. A data structuring calculus and its application to program development, May 1998. Lecture Notes of M.Sc. Course Maestria em Ingeneria del Software, Departamento de Informatica, Facultad de Ciencias Fisico-Matematicas y Naturales, Universidad de San Luis, Argentina.

[Oli98b] J. N. Oliveira. 'Fractal' Types: an Attempt to Generalize Hash Table Calculation. In *Workshop on Generic Programming (WGP'98), Marstrand, Sweden*, June 1998.

[RO97] C. J. Rodrigues and J. N. Oliveira. *Normalization is Data Reification.* Technical Report UMDITR9702, University of Minho, Dec. 1997.

[Tho96] Simon Thompson. *Haskell – The Craft of Functional Programming.* Addison-Wesley, 1st edition, 1996. ISBN 0-201-40357-9.