Toward Generic Data Processing

†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 Braga - Portugal

Abstract

Many aspects of *data processing* are functional in nature and can take advantage of recent developments in the area of *functional programming* and calculi.

The work described in this paper is an attempt to contribute to this line of thought, in particular exploiting the Haskell functional language as support tool.

Haskell is used mainly to animate an abstract model of the relational database calculus as defined by Maier, written in the style of model-oriented formal specification.

Parametricity and genericity (polytypism) make room for generic versions of relational standard (type-constructor parametric) and multi-dimensional analysis operations are expressed in Generic Haskell.

Besides animation, the functional model is further subjected to formal reasoning and calculation, paving the way to the eventual polytypic (generic) formulation of the standard relational calculus.
Keywords: Relational Databases, generic programming, polytypic programming, functional programming.
II Workshop de Ingenieria de Software y Bases de Datos (WISBD)

1 Introduction

The relational data model and associated calculus [Mai83] (originating from *e.g.* [Cod71, Cod72]), are today standard matters in computer science curriculæ. The functional programming addressed toward data structuring and calculation is less well–known. The widely accepted fact that *data precede algorithms* in software construction — known since the days of structured programming in the 1970s — has been the subject of recent research in the field of mathematics of program calculation. This has provided further insight into the role — either *real* or *virtual* — of data structuring in programming.

One of the most significant advances of the last decade has been the so-called *functorial* approach to data types which originated mainly from [MA86], was popularized by [Mal90] and reached the text-book format in [BdM97]. A comfortable basis for explaining *polymorphism* [Wad89], the "data types as functors" motto has proved beneficial at a higher level of abstraction, giving birth to *polytypism* [JJ96], *i.e.* higher-order polymorphism.

Polymorphism and polytypism are steps of the same ladder, that of *generic programming* [Bo98]. The main target of this fast evolving discipline is to raise the level of abstraction of the programming discourse in a way such that seemingly disparate programming techniques, algorithms *etc.* are unified into idealized, kernel programming notions.

However heterogeneous a data source may happen to be, if it is "structured" this means that it has the shape of an inductive finite data structure (*e.g.* a finite set or list, a finitely branching tree or a combination of these two). Inductive data types are expressible in generic programming as fix-points 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 following benefits are targets in the "going functional" approach to data processing:

- Animation: development of new formal specification animation standards through the use of advanced functional languages such as Haskell
- Calculation: to exploit the functional programming calculus, in particular with respect to parametric polymorphism, "theorems for free" [Wad89], etc
- Genericity: higher-order polymorphism which makes for truly generic software models and solutions.
- Foundations: search for new foundations for the relational database calculus.

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 countepart of relational operations.[1]. The last section 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
- upgrade it, wherever necessary
- calculate a final solution and implement it.

[1] This has been carried out in the context of the first author's Master's thesis [Nec05].

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* data types 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

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

and combinator $split :: (a \rightarrow b) \rightarrow (a \rightarrow c) \rightarrow a \rightarrow (b, c)$ 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:

$$(either\ f\ g) \cdot Left\ = f$$
$$(either\ f\ g) \cdot Right = g$$

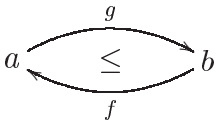
Type constructor *Either* is extended to a bifunctor by providing the following instance of *bmap*:

$$(+) :: (a \to b) \to (c \to d) \to Either\ a\ c \to Either\ b\ d$$
$$(f + g)\ (Left\ a) = Left\ (f\ a)$$
$$(f + g)\ (Right\ b) = Right\ (g\ b)$$

$$instance\ BiFunctor\ Either\ where$$
$$\quad bmap\ f\ g = f + g$$

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:



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 [Oli98], 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 $\phi\ (Set\ l) \stackrel{\text{def}}{=} length\ l = card(elems\ l)$, 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[2]:

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

Table 1 summarizes the Haskell modules defined for these datatypes.

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

	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 :

$$A_1 \times \ldots \times A_n \underset{f}{\overset{r}{\leq}} (\textstyle\sum_{i=1}^{n} A_i)^n$$

under representation function [3] $r\ \ \langle a_j \rangle_{j=1..n} \stackrel{\text{def}}{=} \lambda j.(i_j\ a_j)$ which entails invariant

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

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 [4].

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 [5]:

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

which finally leads to tuples as inhabitants of

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

[3] 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.

[4] 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 $\overline{n}$..

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

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 [Nec05].

	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 Generic Relational Operations

In this section we will extend our model by defining generic versions of relational operators. For instance, the way tuples are structured in our current data model specification, calls for generalization of the collective type which contains tuples in each relation, from:

$$Tuples = \mathcal{P}(IdAttr \rightharpoonup Value) \quad to \quad Tuples = \mathsf{T}(IdAttr \rightharpoonup Value)$$

where T is an arbitrary parametric data type.

The intended generalization step, however, cannot be expressed in standard Haskell and calls for Generic Haskell. We start by over-viewing this language.

5.1 Overview of Generic H∀SKELL

Generic H∀SKELL is based on recent work by Hinze [Hin00] and extends the functional programming language Haskell with, among other features, a construct for defining type-indexed values with kind-indexed types. These values can be specialized to all Haskell data types facilitating wider application of generic programming than provided by earlier systems as eg.POLYP [JJ97].

The Generic H∀SKELL compiler compiles modules written in an enriched Haskell syntax. A Generic H∀SKELL module may contain, in addition to regular Haskell code, definitions of generic functions, kind-indexed types, and type-indexed types, as well as applications of these to types or kinds. The compiler translates a Generic H∀SKELL module into ordinary Haskell by performing a number of tasks:

- translating generic definitions into Haskell code;
- translating calls to generic functions into an appropriate Haskell expressions, and
- specializing generic entities to the types at which they are applied. (Consequently, no type information is passed around at run-time).

In addition, the compiler generates structure types for all data types, together with functions that allow conversion between a data type and its structure type.

A Generic H∀SKELL program may consist of multiple modules. Generic functions defined in one module can be imported into and be reused in other modules. Generic H∀SKELL comes equipped with a library that provides a collection of common generic functions.

In the previous paragraphs we enumerate the basic features of Generic H∀SKELL which are published by Hinze [HJ03]. We will use *generic abstractions* [6], to define generic versions of our relational database operations. Generic abstraction allows generic functions to be defined by abstracting a type variable out of an expression which may involve generic functions. In particular, two generic functions included in the standard distribution of Generic H∀SKELL will be present in deriving all relational operators: *gmap* and *rreduce*.

5.2 Generic Relational Operators

In a full generic function definition, one is forced to be more general than one intends to be. For instance, it is impossible to write a generic function that does not have a function type when applied to a type of kind $* \longrightarrow *$. This is because the specialization mechanism interprets abstraction and application at the type level as abstractions and application at the value level.

To illustrate the later assertion, we reproduce the code of *gmap*'s generic definition.

```
type Map {[ * ]} t1 t2 = t1 -> t2
type Map {[ k -> l ]} t1 t2 = forall u1 u2.
  Map {[ k ]} u1 u2 -> Map {[ l ]} (t1 u1) (t2 u2)

gmap {| t :: k |} :: Map {[ k ]} t t
gmap {| Unit |}     = id
gmap {| :+: |}  gmapA gmapB (Inl a)  = Inl (gmapA a)
gmap {| :+: |}  gmapA gmapB (Inr b)  = Inr (gmapB b)
gmap {| :*: |}  gmapA gmapB (a :*: b)= (gmapA a) :*: (gmapB b)
gmap {| (->) |} gmapA gmapB _        = error "gmap not defined for function types"
gmap {| Con c |}   gmapA       (Con a)  = Con (gmapA a)
gmap {| Label l |} gmapA       (Label a)= Label (gmapA a)
gmap {| Int |}      = id
gmap {| Char |}     = id
gmap {| Bool |}     = id
gmap {| IO |} gmapA = fmap gmapA
gmap {| [] |} gmapA = map gmapA
```

In the Map library, *gmap* is the generic version of *fmap* in the *Functor* class. The type of *gmap* is captured by a kind-indexed type which is defined by induction on the structure of kinds. (The part enclosed in {|.|} is the kind index.)

The rest are equations, one for each type constant, where a type constant is either a primitive type like *Char, Int* etc or one of the three types *Unit*, ": $*$:" and ": $+$:" (null-ary products, binary sums and binary products respectively).

Generic abstraction lifts all restrictions that are normally imposed on the type of a generic function. It enables one to define a function which abstracts a type parameter from an expression, and later apply it generically. The abstracted type parameter is, however, restricted to types of a fixed kind. Generic abstractions can be used to write variations, simplifications and special cases of other generic functions.

The syntax of generic abstractions is similar to ordinary generic definitions, with two important differences:

- the type signature is restricted to a fixed kind, and thus no kind variable is introduced; and
- they consist of just one case which has a type variable as its type argument, rather than a named type.

Generic Project Suppose that we have several kinds of relations, in which the difference is the shape of the structure that contains the tuples, for instance:

[6] one of the extension described by Clarke and Lőh in [CL02], which are implemented in the current version of the Generic H∀SKELL compiler

```
data Relation  = Rel  {schema:: Pfun IdAttr AttrInfo, tuples:: Set (Pfun IdAttr Value) }
data RelationL = RelL {schemaL:: Pfun IdAttr AttrInfo, tuplesL:: [(Pfun IdAttr Value)] }
data RelationLT= RelLT{schemaLT:: Pfun IdAttr AttrInfo, tuplesLT::LTree (Pfun IdAttr Value)}
```

where *LTree* is the "leaf tree" datatype, defined in Haskell as follows:

```
data LTree a = Leaf a | Split (LTree a, LTree a)
```

We want to define a generic function $\pi_{gtuples}$ which, taking a set of attributes names and a generic shape of tuples, returns a generic tuple structure where each tuple is restricted to the same set of attributes:

$$\begin{aligned} \pi_{gtuples} &: \mathcal{P}A \to \mathsf{T}(Tuple) \to \mathsf{T}(Tuple) \\ \pi_{gtuples}\ s\ ts &= \mathsf{T}(\lambda t.\ t \mid s)\ ts \end{aligned}$$

To encode this operator in Generic H∀SKELL , we define a generic abstraction which uses the *gmap* function (provided in Generic H∀SKELL library) to access each tuple of any tuple structure:

```
gprojectTup {| t :: * -> * |} ::(Eq a )=> Set a -> t (Pfun a b) -> t (Pfun a b)
gprojectTup {| t |}  s ts = gmap {| t |} (restp s) ts
```

Generic functions are called by instancing the type-index to a specific type. As an illustration of this, we can specialize *gprojectTup* to different types, for instance:

```
projectR:: Set [Char] -> Relation -> Relation
projectR p (Rel s t)= Rel (restp p s ) (gprojectTup {| Set |} p t )
projectRL:: Set [Char] -> RelationL -> RelationL
projectRL p (RelL s t)= RelL (restp p s) (gprojectTup {| [] |} p t )
projectRLT:: Set [Char] -> RelationLT -> RelationLT
projectRLT p (RelLT s t)= RelLT (restp p s) (gprojectTup {| LTree |} p t)
```

Specialized Function *projectR* can be used to specify the project operation at RDB level as follows:

```
sproject :: Set IdAttr -> IdRel -> RDB -> Error Relation
sproject s id db =
 do {r1 <- aplpf' (relations db) id ;
     result <- inv' ( projectR s r1) ;
     if (s \< dom(schema r1)) then Ok (result)
     else Err "Error in project operation:attrs. are not in relation domain"}
```

Generic Select We proceed to defining $\sigma_{gtuples}$, the generic function that, taking a selection criteria (partial function) and a generic shape of tuples, returns a generic tuple structure where each tuple satisfies the selection criteria presented by the first argument (that is, a tuple structure where each tuple is "coherent" or "compatible" with the first argument):

$$\begin{aligned} \sigma_{gtuples} &: (A \rightharpoonup B) \to \mathsf{T}(Tuple) \to \mathsf{T}(Tuple) \\ \sigma_{gtuples}\ f\ ts &= gfilter\ (coherent\ f)\ ts \end{aligned}$$

To specify this operator in Generic H∀SKELL , we define a generic abstraction which uses the *rreduce* function to implement $gfilter$. The *rreduce* function, provided in Generic H∀SKELL library, is a generic version of $foldr$, typed as follows:

$$\begin{aligned} rreduce\{|t :: *|\} &:: t \to B \to B \\ rreduce\{|t :: * \to *|\} &:: (A \to B \to B) \to tA \to B \to B \end{aligned}$$

Note the reversed order of the last two arguments.

Back to the definition of $\sigma_{gtuples}$, the idea is to access each tuple of a tuple structure and check its with the first argument of the selection function. If the tuple is compatible (coherent), it is "put into" the result structure. We parameterize the operation that permits to add a tuple to the structure and the empty structure (they will be known when the function will be instanced).

```
gselectTup {| t:: * -> * |} :: (Eq a, Eq b)=> Pfun a b
       -> ((Pfun a b) -> t (Pfun a b) -> t(Pfun a b))
       -> t(Pfun a b) -> t(Pfun a b) -> t(Pfun a b)
gselectTup {| t |} p f te xs
       = rreduce {| t |} (\x y -> if (coherent p x) then (f x y) else  y ) xs te
```

The specialization of *gselectTup* to our original model is:

```
selectR :: Pfun [Char] Value -> Relation -> Relation
selectR p (Rel s t) = Rel s (gselectTup {| Set |} p puts (Set[]) t)
```

And the select operation at *RDB*-level becomes:

```
sselect:: Pfun IdAttr Value -> IdRel -> RDB -> Error Relation
sselect f id db =
 do {r1 <- aplpf' (relations db) id ;
     result <- inv' (selectR f r1);
     if ((dom f) \< dom(schema r1))then Ok (result)
     else Err "Error in select operation: attrs. are not in relation domain"}
```

Generic Boolean Operations To specify *generic Boolean operations* of two structures of tuples, we define generic abstractions using *rreduce* and *gany* functions. In the case of *generic union*, *rreduce* is used to add the tuples of the second structure to the first structure. Only the tuples that are not in the structure are added. Function *gany* is provided in the Generic H∀SKELL library. It is a generic version of *any* (existential quantifying over finite lists defined in Haskell Prelude):

$$gany\{|t :: * \to *|\} \quad :: \quad (A \to Bool) \to \mathsf{T}A \to Bool$$

Function *gany* will be used to check if each tuple is in the result structure. We parameterize the operation that adds a tuple to the structure (first parameter of *gunionTup*) because it won't be known until the function is specialized.

```
gunionTup {| t :: * -> * |} :: (Eq a, Eq b, Eq (Pfun a b))
              => ((Pfun a b) -> t(Pfun a b) -> t(Pfun a b))
              -> t(Pfun a b) -> t(Pfun a b) -> t(Pfun a b)
gunionTup {| t |} f t1 t2 = rreduce {| t |}
           (\x y-> if (gany{| t |}(\z -> z==x) t2) then y else (f x y)) t1 t2
```

Once again, what we had before at *RDB*-level stems from a specialization of generic *gunionTup*:

```
unionR  (Rel sx tx) (Rel sy ty) = Rel sx (gunionTup {| Set |} puts tx ty )

sunion :: IdRel -> IdRel -> RDB -> Error Relation
sunion id1 id2 db =
 do {r1 <- aplpf' (relations db) id1 ;
     r2 <- aplpf' (relations db) id2 ;
     result <- inv' (unionR r1 r2) ;
     if (restrEqdom r1 r2) then Ok result
     else  Err "Error in Union Operation: incompatible schemes"}
```

In a similar way, the specification of *generic intersection* of two structures of tuples involves function *rreduce* to construct a structure with the common tuples. We parameterize the operation that adds a tuple to the structure and the empty structure.

```
ginterTup {| t :: * -> * |} :: (Eq a, Eq b, Eq (Pfun a b))
          => ((Pfun a b) -> t(Pfun a b) -> t(Pfun a b))
          -> t(Pfun a b) -> t(Pfun a b) -> t(Pfun a b) -> t(Pfun a b)
ginterTup {| t |} f te t1 t2 =
 rreduce {| t |} (\x y->if (gany{| t |}(\z ->z==x) t2) then (f x y) else y) t1 te
```

The specialization of *ginterTup* and the intersection operation at *RDB*-level is as follows:

```
interR (Rel sx tx)(Rel sy ty)= Rel sx (ginterTup {| Set |} puts (Set[]) tx ty)

sinter :: IdRel -> IdRel -> RDB -> Error Relation
sinter id1 id2 db =
 do {r1 <- aplpf' (relations db) id1 ;
     r2 <- aplpf' (relations db) id2 ;
     result <- inv'(interR r1 r2) ;
     if (restrEqdom r1 r2) then Ok (result)
     else Err "Error in Intersection Operation: incompatible schemes"}
```

The abstraction defined for *generic difference* between two tuple structures is similar to the *generic intersection* function, but with the if's branches inverted:

```
gdiffTup {| t :: * -> * |} :: (Eq a, Eq b, Eq (Pfun a b))
              => ((Pfun a b) -> t(Pfun a b) -> t(Pfun a b))
              -> t(Pfun a b) -> t(Pfun a b) -> t(Pfun a b) -> t(Pfun a b)
gdiffTup {| t |} f te t1 t2 =
  rreduce {| t |} (\x y-> if (gany{| t |}(\z -> z==x) t2)
                          then y else (f x y)) t1 te
```

Generic Renaming Let $\delta_{gtuples}$ denote the generic function that, taking a *rename* function from attribute names to attribute names and a generic shape of tuples, returns a generic tuple structure where attribute names are changed via the rename function. In Haskell, this is function

```
grenameTup {| t :: * -> * |}::(Eq a, Eq b )=>Pfun a a ->t (Pfun a b) ->t (Pfun a b)
grenameTup {| t |} r xs =gmap {| t |} (renpf r) xs
```

Generic Natural Join Let $\bowtie_{gtuples}$ denote the binary operator for combining two tuple structures on all their common attributes. Should they have no common attributes, $\bowtie_{gtuples}$ will return the Cartesian product of them.

First, we define an auxiliary function

$$\begin{array}{rcl} gfilter & : & (A \to Bool) \to (A \to B) \to ((B \times \mathsf{T}B) \to \mathsf{T}B) \to \mathsf{T}B \to \mathsf{T}A \to \mathsf{T}B \\ gfilter\ p\ f\ op\ es & \stackrel{\text{def}}{=} & \{\!|[\underline{es}, (\lambda\, x\ y\ .(p\ \ x) \to op\ \ (f\ \ x)\ \ y, y)]|\!\} \end{array}$$

which filters a structure, retaining only those elements that satisfy p, and applies f to each such element. The third and fourth parameters correspond to the operation that adds a tuple to the structure and the empty structure, respectively [7].

```
gfilter {| t:: * -> * |}::(a -> Bool) -> (a -> b) -> (b -> t b -> t b) -> t b -> t a -> t b
gfilter {| t |} p op f te xs=rreduce {| t |}(\x y->if (p x) then f (op x) y else  y) xs te
```

Then we can define:

```
gnatjoinTup {| t |} ft1 ft2 te1 te2 r1 r2 =
  let a=gmap {| t |} (\x->(gfilter {| t |} (coherent x) (aux x) ft1 te1 r2)) r1
  in rreduce {| t |} ft2 a te2
    where
aux:: Eq a => Pfun a b -> Pfun a b -> Pfun a b
aux x y =(dom(x) /\ dom(y)) <-: (plus x y)
```

Finally, the specialization of *gnatjoinTup* and the natural join operation to our set-based *RDB*-level are, as expected:

```
gnatjoinR  (Rel sx tx) (Rel sy ty)=
 Rel((dom(s1)/\dom(s2))<-:(s1\*/s2))(gnatjoinTup{|Set|} puts (\/)(Set[])(Set[]) tx ty)

snatjoin id1 id2 db = do {r1 <- aplpf' (relations db) id1 ;
                          r2 <- aplpf' (relations db) id2 ;
                          inv'(natjoinR r1 r2) }
```

[7] For instance, instanced to Set, *gfilter* corresponds to ZF-set abstraction: $\{f\ \ a \mid a \in s \wedge p(a)\}$

6 Concluding Remarks

We developed generic relational operators using a language with explicit syntactic constructs for defining polytypic functions. We first experimented with the PolyP [JJ97] system but soon realized that there was a serious shortcoming: only *unary* (regular) data types can be polytypically defined. We have overcome this problem by switching to Generic H∀SKELL , which implements a new approach to generic programming due to Hinze [Hin00]. In particular, we experimented with generic abstractions which are introduced in [CL02]. We use function *gmap* in all cases to apply specific functions that "preserve" Set's invariant.

Our main conclusions are that generic definitions using generic abstractions are simple to write in Generic H∀SKELL and can be applied to Haskell 98 types of all kinds.

We cannot compile a generic function without knowing to which data types it will be called. It is not possible to compile generic functions separately from the code that calls them.

Why such an emphasis on parametricity and genericity?

Surely there is some intellectual reward and conceptual economy in designing solutions to specific problems as the *customization* of *generic* ones. However, there is more. In a famous paper entitled *Theorems for free!* [Wad89] Philip Wadler writes:

> *From the* type *of a polymorphic function we can derive a* theorem *that is satisfy.* (...) *How useful are the theorems so generated?* Only time and experience will tell (...)

This result is a rewording of *Reynolds abstraction theorem* on parametric polymorphism which can be found in a remarkably elegant point-free formulation in [BB03]. This paper and others (eg. [OR04]) present examples of the use of this theorem to calculate useful *fusion-laws* involving polymorphic types. So, it is to be expected that every of our generic, polymorphic models of relational and Olap operators will enjoy one such fusion-law, the corollaries of which — if already known — will be thus proved "for free", and — if still unknown — will add to the theory behind such important areas of computing.

References

[BB03] K. Backhouse and R.C. Backhouse. Safety of abstract interpretations for free,via logical relations and Galois connections. *Science of Computer Programming*, 2003. Accepted for publication.

[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/`).

[CL02] Dave Clarke and Andres Löh. Generic haskell, specifically. In *In Jeremy Gibbons and Johan Jeuring, editors. Generic Programming. Proceedings of the IFIP TC2 Working Conference on Generic Programming, Schloss Dagstuhl, July 2002. ISBN 1-4020-7374-7. Kluwer Academic Publishers*, pages 21–48, 2002.

[Cod71] E. F. Codd. Normalized database structure: A brief tutorial. In *ACM SIGFIDET Workshop on Data Description, Access and Control*, pages 1–17, Nov. 1971.

[Cod72] E. F. Codd. Further normalization of the database relational model. In *Data Base Systems*, pages 33–64. Prentice-Hall, 1972. Courant Inst. Computer Science Symp. 6, Englewood Cliffs, NJ.

[Hin00] Ralf Hinze. Polytypic values possess polykinded types. In *Mathematics of Program Construction*, pages 2–27, 2000.

[HJ03] R. Hinze and J.Jeuring. Generic haskell: Practice and theory, 2003. Lecture notes of the Summer School on Generic Programming, LNCS Springer-Verlang.

[JJ96] J. Jeuring and P. Jansson. Polytypic programming. In *Advanced Functional Programming*, number 1129 in Lecture Notes in Computer Science. Springer, 1996.

[JJ97] Patrik Jansson and Johan Jeuring. PolyP—A polytypic programming language extension. In *Conf. Record 24th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL'97, Paris, France, 15–17 Jan 1997*, pages 470–482. ACM Press, New York, 1997.

[MA86] E. G. Manes and M. A. Arbib. *Algebraic Approaches to Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1986. D. Gries, series editor.

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

[Mal90] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.

[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).

[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).

[Oli98] 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.

[OR04] J.N. Oliveira and C.J. Rodrigues. Transposing relations: from *Maybe* functions to hash tables. In *MPC'07 : Seventh International Conference on Mathematics of Program Construction, 12-14 July, 2004, Stirling, Scotland, UK (Organized in conjunction with AMAST'04)*, Lecture Notes in Computer Science. Springer, 2004. Accepted for publication.

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

[Wad89] Philip L. Wadler. Theorems for free! In *4th International Symposium on Functional Programming Languages and Computer Architecture*, London, Sep. 1989. ACM.