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: generic programming, polytypic programming, functional programming.

II Workshop de Ingeniería de Software y Bases de Datos (WISBD)
1 Introduction

The relational data model and associated calculus [Mai83] (originating from \textit{e.g.} [Cod71, Cod72]), are today standard matters in computer science curricula. The functional programming addressed toward data structuring and calculation is less well-known. The widely accepted fact that \textit{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 \textit{real} or \textit{virtual} — of data structuring in programming.

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

Polymorphism and polytypism are steps of the same ladder, that of \textit{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 \textit{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 (\textit{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 \textit{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], \textit{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 counterpart 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 \textit{specification phase}, in which a mathematical model is built of the contractor requirements, followed by an \textit{implementation phase} in which such a model is somehow converted into a runnable software artifact. Formal methods research shows that implementations can be effectively \textit{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 \( f :: a \to b \) 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 \( \text{id}_a :: a \to a \), which is the unit of composition.

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

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

```
class Functor f where
  fmap :: (a -> b) -> (f a -> f b)
```

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

```
fmap id = id
fmap (f \cdot g) = (fmap f) \cdot (fmap g)
```

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

```
class Bifunctor f where
  bmap :: (a -> c) -> (b -> d) -> (f a b -> f c d)
```

**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. \( \pi_1 \) and \( \pi_2 \) in standard mathematical notation). Type constructor \((\cdot)\) is extended to a bifunctor in the obvious way:

```
instance Bifunctor (,) where
  bmap f g = f \times g

where

  (\times) :: (a -> b) -> (c -> d) -> (a, c) -> (b, d)
  (f \times g) = split (f \cdot fst) \cdot (g \cdot snd)
```

and combinator \( \text{split} :: (a \to b) \to (a \to c) \to a \to (b, c) \) behaves as follows: \( \text{split} f g x = (f x, g x) \).
Sums. Categorical sums are defined in the Haskell Prelude by means of type constructor

\[
\text{data Either } a \ b = \text{Left } a \mid \text{Right } b
\]

together with a function either :: (a -> b) -> (c -> b) -> Either a c -> b satisfying the following equations:

\[
\begin{align*}
\text{either } f \ g \cdot \text{Left } & = f \\
\text{either } f \ g \cdot \text{Right } & = g
\end{align*}
\]

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

\[
\begin{align*}
(+) \ :: \ (a \rightarrow b) \rightarrow (c \rightarrow d) \rightarrow \text{Either } a \ c \rightarrow \text{Either } b \ d \\
(f + g) \ (\text{Left } a) & = \text{Left } (f \ a) \\
(f + g) \ (\text{Right } b) & = \text{Right } (g \ b)
\end{align*}
\]

\[
\text{instance BiFunctor Either where}
\]
\[
\text{bmap } f \ g = f + g
\]

The popular notations \( \langle f, g \rangle \), \([f, g]\) and \( F \ f \) (where \( F \) is a functor) will be adopted interchangeably with split \( f \ g \), either \( f \ g \) and \( f \map g \), 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 \( f \cdot g = \text{id}_a \). 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:

\[
\begin{array}{c}
\text{g} \\
\downarrow \\
\text{f}
\end{array}
\begin{array}{c}
\leq \\
\text{a} \\
\uparrow \\
\text{b}
\end{array}
\]

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) -> (b,a) which is defined by swap = \( (\pi_2, \pi_1) \) 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 \( P(a) \) and finite partial mappings \( a \rightarrow b \). 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 \( \phi\ (\text{Set } l) \) \( \equiv \) length \( l \) = card(\text{elem s 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\(^2\):

\[
f dp \ \overset{\text{def}}{=} \ (\subseteq \ 1) \cdot \text{rng} \cdot (\text{id} \rightarrow \text{card}) \cdot \text{collect}
\]

Table 1 summarizes the Haskell modules defined for these datatypes.

---

\(^2\)collec:: \( P(a \times b) \rightarrow (a \rightarrow Pb) \) converts a relation into a set-valued partial function and \( \text{rng} :: (a \rightarrow b) \rightarrow Pb \) is the usual range function.
<table>
<thead>
<tr>
<th>Datatypes:</th>
<th>Finite Sets</th>
<th>Partial Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>data Set a = Set [a]</td>
<td>data Pfun a b = Map([a,b])</td>
<td></td>
</tr>
</tbody>
</table>

| Constructors: | emptyS, sings, puts, prods | bottom, singpf, putpf |
| Deletions:    | gets                     | getpf |
| Observers:    | ins, nins, incls, card   | compatible, incompatible |
| Filters:      | filterS                  | allPf |
| Operations:   | interS, unions, diffS, plus, pfzip | plus, pfinv, restn, restp |
|               | flatS, flatl, slstr, srstr, sextl, sextr | pfzip, pfzipWith |
|               | zipS, zipWithAllS        | |
| Folds:        | foldS                    | foldPf |
| Functor:      | fmapS                    | bmapPf |
| Bifunctor:    |                         | |
| Others:       | the, stol, elems, card   | dom, rng, aplf |
|               | unzipS                   | 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 $(a_1, \ldots, a_n)$. Each entry $a_i$ in vector $t = (a_1, \ldots, a_n)$ 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. $i \times t$ 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:

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

under representation function $^3 r \ (a_j)_{j=1..n} \overset{\text{def}}{=} \lambda j. (i_j a_j)$ which entails invariant

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

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 $^5$:

$$\left( \sum_{i \in \pi} A_i + 1 \right)^n$$

which finally leads to tuples as inhabitants of

$$\text{Tuple} = (\overline{n} \to \sum_{i \in \pi} A_i)$$

$^3$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 \{\text{NULL}\}.  

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

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].

<table>
<thead>
<tr>
<th>Relations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datatypes:</td>
</tr>
<tr>
<td>type Tuple = Pfun IdAttr Value</td>
</tr>
<tr>
<td>type SchemaR = Pfun IdAttr AttrInfo</td>
</tr>
<tr>
<td>type IdAttr = String</td>
</tr>
<tr>
<td>type Tuples = Set Tuple</td>
</tr>
<tr>
<td>data Relation = Rel { schema::SchemaR, tuples::Tuples }</td>
</tr>
<tr>
<td>data AttrInfo = InjA { iKey::Bool, defValue::Value }</td>
</tr>
<tr>
<td>data Value = Int Int</td>
</tr>
<tr>
<td>Constructors:</td>
</tr>
<tr>
<td>emptyR</td>
</tr>
<tr>
<td>Operations:</td>
</tr>
<tr>
<td>unionR, interR, diffR</td>
</tr>
</tbody>
</table>

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:

\[ \text{Tuples} = \mathcal{P}( \text{IdAttr} \rightarrow \text{Value} ) \quad \text{to} \quad \text{Tuples} = \mathsf{T}( \text{IdAttr} \rightarrow \text{Value} ) \]

where \( \mathsf{T} \) 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 \( \mathsf{H\&SKELL} \)

Generic \( \mathsf{H\&SKELL} \) 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 e.g. \texttt{POLYP} [JJ97].

The Generic \( \mathsf{H\&SKELL} \) compiler compiles modules written in an enriched Haskell syntax. A Generic \( \mathsf{H\&SKELL} \) 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 \( \mathsf{H\&SKELL} \) 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 \( \mathsf{H\&SKELL} \) program may consist of multiple modules. Generic functions defined in one module can be imported into and be reused in other modules. Generic \( \mathsf{H\&SKELL} \) comes equipped with a library that provides a collection of common generic functions.
In the previous paragraphs we enumerate the basic features of Generic HVSKELL 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 HVSKELL will be present in deriving all relational operators: \texttt{gmap} and \texttt{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 \(\ast \longrightarrow \ast\). 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 \texttt{gmap}'s generic definition.

```haskell
type Map {( k )} t1 t2 = t1 -> t2
Map {( k )} u1 u2 = Map {( 1 )} (t1 u1) (t2 u2)

gmap {t :: k} :: Map {( k )} t t

gmap {Unit} = id

gmap {++} gmapA gmapB (In1 a) = In1 (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 1} 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, \texttt{gmap} is the generic version of \texttt{fmap} in the \texttt{Functor} class. The type of \texttt{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 \texttt{Char}, \texttt{Int} etc or one of the three types \texttt{Unit}, \texttt{*: :} and \texttt{*: +:} (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 Löh in [CL02], which are implemented in the current version of the Generic HVSKELL 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 \( \pi_{\text{gtuple}} \) 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:

\[
\pi_{\text{gtuple}} : \mathcal{P}A \to T(Tuple) \to T(Tuple)
\]

\[
\pi_{\text{gtuple}} s t s = T(\lambda t. t | s) t s
\]

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

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

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

\[
\text{projectR} :: \text{Set } [\text{Char}] \to \text{Relation} \to \text{Relation}
\]

\[
\text{projectR} p (\text{Rel } s t) = \text{Rel } (\text{restp } p s) (\text{gprojectTup } \{ \text{Set } l \} p t)
\]

\[
\text{projectR} :: \text{Set } [\text{Char}] \to \text{RelationL} \to \text{RelationL}
\]

\[
\text{projectR} p (\text{RelL } s t) = \text{RelL } (\text{restp } p s) (\text{gprojectTup } \{ \text{[] | l} \} p t)
\]

\[
\text{projectR} :: \text{Set } [\text{Char}] \to \text{RelationLT} \to \text{RelationLT}
\]

\[
\text{projectR} p (\text{RelLT } s t) = \text{RelLT } (\text{restp } p s) (\text{gprojectTup } \{ \text{LTree } l \} p t)
\]

Specialized Function \( \text{projectR} \) can be used to specify the project operation at RDB level as follows:

\[
\text{project} :: \text{idAttr} \to \text{IdRel} \to \text{RDB} \to \text{Error Relation}
\]

\[
\text{project } s id db =
\begin{array}{l}
do \{ r1 <- \text{aplf}\, (\text{relations db})\, id \}; \\
\quad \text{result } <- \text{inv' } (\text{projectR } s r1)\
\quad \text{if } (s \not\text{\in dom(schema r1)})\text{ then Ok (result)} \\
\quad \text{else Err "Error in project operation:attrs. are not in relation domain"}
\end{array}
\]

**Generic Select** We proceed to defining \( \sigma_{\text{gtuple}} \), 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):

\[
\sigma_{\text{gtuple}} : (A \to B) \to T(Tuple) \to T(Tuple)
\]

\[
\sigma_{\text{gtuple}} f t s = gfilter (\text{coherent } f) t s
\]

To specify this operator in Generic H<SVKELL>, we define a generic abstraction which uses the \( rreduce \) function to implement \( gfilter \). The \( rreduce \) function, provided in Generic H<SVKELL> library, is a generic version of \text{foldr}, typed as follows:

\[
rreduce\{ t :: * \}\ :: t \to B \to B
\]

\[
rreduce\{ t :: * \to * \}\ :: (A \to B \to B) \to tA \to B \to B
\]

Note the reversed order of the last two arguments.

Back to the definition of \( \sigma_{\text{gtuple}} \), 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).
The specialization of `gselectTup` to our original model is:

```
selectR :: Pfun [Char] Value -> Relation -> Relation
selectR p (Rel s t) = Rel s (gselectTup {t | l} 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: the domain is not in relation domain"
```

**Generic Boolean Operations** To specify generic Boolean operations of two structures of tuples, we define generic abstractions using `reduce` and `gany` functions. In the case of generic union, `reduce` 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 Haskell library. It is a generic version of `any` (existential quantifying over finite lists defined in Haskell Prelude):

```
gany{[t :: * -> *]} :: (A -> Bool) -> TA -> 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 | l} :: (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 | l} f t1 t2 = rreduce {t | l}
   ([x y -> if (gany{[t | l]} (\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 {t | l} puts tx ty)
```

In a similar way, the specification of generic intersection of two structures of tuples involves function `reduce` 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 | l} :: (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 | l} f te t1 t2 =
   rreduce {t | l} (\x y -> if (gany{[t | l]} (\z -> z == x) t2) then y else y) t1 te
```
The specialization of $\text{ginterTup}$ and the intersection operation at $RDB$-level is as follows:

\[
\text{sinter} :: \text{IdRel} \rightarrow \text{IdRel} \rightarrow \text{RDB} \rightarrow \text{Error Relation}
\]

\[
\text{sinter id1 id2 db} =
\]

\[
do \{r1 <- \text{aplpf}' \text{ (relations db) id1} ;
\]

\[
r2 <- \text{aplpf}' \text{ (relations db) id2} ;
\]

\[
\text{result} <- \text{inv'} \text{ (interR r1 r2)} ;
\]

\[
\text{if (restrEqDom r1 r2) then Ok (result)}
\]

\[
\text{else Err "Error in Intersection Operation: incompatible schemes"}
\]

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

\[
\text{gdiffTup} \{! t :: * \rightarrow * !\} :: (\text{Eq a, Eq b, Eq (Pfun a b)})
\]

\[
\rightarrow ((\text{Pfun a b} \rightarrow t(\text{Pfun a b}) \rightarrow t(\text{Pfun a b})))
\]

\[
\rightarrow t(\text{Pfun a b} \rightarrow t(\text{Pfun a b}) \rightarrow t(\text{Pfun a b}) \rightarrow t(\text{Pfun a b}))
\]

\[
\text{gdiffTup} \{! t !\} f te t1 t2 =
\]

\[
\text{rreduce} \{! t !\} (\langle x y \rangle \rightarrow \text{if (gany}(! t !)(\langle z \rightarrow z=x \rangle) t2)
\]

\[
\text{then y else (f x y)) t1 te}
\]

**Generic Renaming** Let $\delta_{\text{tuples}}$ 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

\[
\text{grenameTup} \{! t :: * \rightarrow * !\} :: (\text{Eq a, Eq b}) \rightarrow \text{Pfun a a} \rightarrow t(\text{Pfun a b}) \rightarrow t(\text{Pfun a b})
\]

\[
\text{grenameTup} \{! t !\} r xs = \text{gmap} \{! t !\} (\text{renpf r}) xs
\]

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

First, we define an auxiliary function

\[
\text{gfilter} : (A \rightarrow \text{Bool}) \rightarrow (A \rightarrow B) \rightarrow ((B \times TB) \rightarrow TB) \rightarrow TB \rightarrow TA \rightarrow TB
\]

\[
\text{gfilter p f op es} \overset{\text{def}}{=} \{[e, (\lambda x y . (p x) \rightarrow \text{op} \ (f x) y, y)]\}
\]

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.  

\[
\text{gfilter} \{! t :: * \rightarrow * !\} :: (\text{a} \rightarrow \text{Bool}) \rightarrow (\text{a} \rightarrow \text{b}) \rightarrow (\text{b} \rightarrow \text{t b} \rightarrow \text{t b}) \rightarrow \text{t b} \rightarrow \text{t a} \rightarrow \text{t b}
\]

\[
\text{gfilter} \{! t !\} p \text{ op f te xs}=\text{rreduce} \{! t !\}(\langle x y \rangle \rightarrow \text{if (p x) then f (op x) y else y}) \text{ xs te}
\]

Then we can define:

\[
\text{gnatjoinTup} \{! t !\} ft1 ft2 te1 te2 r1 r2 =
\]

\[
\text{let a}=\text{gmap} \{! t !\} (\langle x \rightarrow \text{gfilter} \{! t !\} (\text{coherent x}) (\text{aux x}) \text{ ft1 te1 r2}) \text{ r1}
\]

\[
in \text{rreduce} \{! t !\} \text{ ft2 a te2}
\]

\[
\text{where}
\]

\[
\text{aux x y}=(\text{dom}(x) \land \text{dom}(y)) \rightarrow \text{if (plus x y)}
\]

Finally, the specialization of $\text{gnatjoinTup}$ and the natural join operation to our set-based $RDB$-level are, as expected:

\[
\text{gnatjoinR} \text{ (Rel sx tx) (Rel sy ty)=}
\]

\[
\text{Rel}((\text{dom(s1)} \land \text{dom(s2)})) \leftarrow ((\text{s1} \land \text{s2}))\text{ (gntatjoinTup} \{! \text{Set} !\} \text{ puts} \ (\land \text{(Set}) \text{)(Set}) \text{ tx ty)}
\]

\[
\text{snatjoin id1 id2 db} = \text{do (r1} \rightarrow \text{aplpf'} \text{ (relations db) id1 ;}
\]

\[
r2 \leftarrow \text{aplpf'} \text{ (relations db) id2 ;}
\]

\[
\text{inv'} \text{(natjoinR r1 r2)}
\]

\[
\text{\text{7For instance, instanced to Set, gfilter corresponds to ZF-set abstraction: } \{f a \mid a \in s \land p(a)\}}
\]
6 Concluding Remarks

We developed generic relational operators using a language with explicit syntactic constructs for defining polymorphic 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 polymorphically defined. We have overcome this problem by switching to Generic HYSKELL, 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 HYSKELL 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 \textit{Theorems for free!} [Wad89] Philip Wadler writes:

\begin{quote}
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 (...)
\end{quote}

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


