Higher-Order Symb Comput (2010) 23:337–370
DOI 10.1007/s10990-011-9078-8

Generic multiset programming with discrimination-based joins and symbolic Cartesian products

Fritz Henglein · Ken Friis Larsen

Published online: 25 November 2011

Abstract This paper presents GMP, a library for generic, SQL-style programming with multisets. It generalizes the querying core of SQL in a number of ways: Multisets may contain elements of arbitrary first-order data types, including references (pointers), recursive data types and nested multisets; it contains an expressive embedded domain-specific language for specifying user-definable equivalence and ordering relations, extending the built-in equality and inequality predicates; it admits mapping arbitrary functions over multisets, not just projections; it supports user-defined predicates in selections; and it allows user-defined aggregation functions.

Most significantly, it avoids many cases of asymptotically inefficient nested iteration through Cartesian products that occur in a straightforward stream-based implementation of multisets. It accomplishes this by employing two novel techniques: symbolic (term) representations of multisets, specifically for Cartesian products, for facilitating *dynamic symbolic computation*, which intersperses algebraic simplification steps with conventional data processing; and *discrimination-based joins*, a generic technique for computing equijoins based on equivalence discriminators, as an alternative to hash-based and sort-merge joins.

Full source code for GMP in Haskell, which is based on generic top-down discrimination (not included), is included for experimentation. We provide illustrative examples whose performance indicates that GMP, even without requisite algorithm and data structure engineering, is a realistic alternative to SQL even for SQL-expressible queries.

Keywords Generic · Programming · Multiset · Bag · Discrimination · Query · Querying · LINQ · SQL · Relational · Algebra · Select · Project · Join · Filter · Map · Lazy · Symbolic · Equivalence · Ordering · Haskell · GADT

This material is based upon work supported by the Danish Strategic Research Council under Project HIPERFIT (hiperfit.dk) and by the Danish National Advanced Technology Foundation under Project 3gERP.

F. Henglein · K.F. Larsen (✉)
Department of Computer Science (DIKU), University of Copenhagen, 2100 Copenhagen, Denmark
e-mail: kflarsen@diku.dk

F. Henglein
e-mail: henglein@diku.dk

1 Introduction

Processing bulk data (collections) stored in external or internal memory is a common task in many programming applications. For externally stored shared data applications, SQL has established itself as the *de facto* interface language for relational database systems.

From a programming perspective, SQL has a number of shortcomings, notwithstanding that some of them are *by design*. Amongst others [8], these are:

- It offers a limited set of operations on tables. In particular, no ad-hoc user-defined operations are allowed in queries.
- Table data are restricted to atomic value types. Neither references nor structured types such as collections or trees are allowed as elements of tables.
- SQL queries are typically not statically type checked, opening the door to SQL injection attacks.
- Application programming language and SQL data types usually do not match one-to-one: They have an "impedance", requiring back-and-forth conversion, which incurs computational costs and is fragile under both program and database schema evolution.

Application language specific libraries for bulk data programming, on the other hand, typically lack the powerful query optimization technology modern RDBMSs provide for SQL, in particular its efficient processing of join queries. It is noteworthy that recent query languages such as GQL[1] for popular key-value ("NoSQL") data stores disallow join queries, exposing application programmers, who do not necessarily have an academic computer science background, to the most challenging part of bulk data programming: Handcrafting efficient join algorithms, or, as a common work-around, storing data in the form of potentially huge and unwieldy prejoined collections.

Language-integrated querying [28, 30] has been proposed to combine the virtues of type-safe queries embedded in an application language with the possibility of powerful query optimization: Application programs do not execute bulk data computations, but instead construct queries dynamically and ship them off to a so-called data provider, which may be a RDBMS with a query optimizer. This encapsulates the impedance mismatch, but does not eliminate it [14].

In this paper we reexamine the bulk data library approach. We provide a library for SQL-style multiset programming that

- supports all the classical features of the data query sublanguage of SQL;
- allows multisets of any element type, including nested multisets and trees;
- permits user-definable predicates in selections, user-definable functions in projections (actually maps), and user-definable equivalences in join conditions;
- admits naïve programming with Cartesian products, without necessarily incurring quadratic execution time cost for computing them;
- and is easy to implement.

To demonstrate the last point we include the complete source code of a prototype implementation in Haskell, present examples of SQL-style programming using GMP, and demonstrate that it performs asymptotically quite acceptably on these examples *vis a vis* MySQL, despite being completely untuned.

[1] See http://code.google.com/appengine/docs/python/datastore/gqlreference.html.

The key to the library is an ostensibly simple idea, which does not seem to have been pursued previously, however: Representing Cartesian products symbolically *at run time*, and performing opportunistic *dynamic symbolic computation* on them.

This idea is combined with a new join algorithm based on *generic top-down discrimination* [22, 23]. The resulting algorithm combines symbolic Cartesian product representation and generic discrimination and computes joins, even for user-defined functions and equivalence relations, in *worst-case* linear time.

This paper emphasizes the programming aspects, in particular

- the key idea of using symbolic Cartesian products to employ dynamic symbolic computation for computational efficiency; and
- discrimination-based joins as a new generic equijoin algorithm for user-definable equivalence relations.

We also briefly discuss the tensions between semantic determinacy, efficient joins, and dependent products; and relate our work to other frameworks for querying in-memory and RDBMS-managed data.

The material is based on Henglein [24], which introduces symbolic ("lazy") Cartesian products and generic discrimination-based joins; and on Henglein and Larsen [25], which extends the techniques to a library for SQL-style multiset programming. The present paper in part simplifies and in part generalizes those results. It furthermore adds a step-by-step development of the method of opportunistic symbolic computation instead of presenting it only as a *fait accompli*,[2] and it expands on the example queries by providing reference formulations in MySQL.

1.1 Required background, notation and terminology

A basic understanding of the relational data model and SQL is required. Any textbook on databases will do; e.g. Ramakrishnan and Gehrke [36].

We use the functional core parts of Haskell [34] as our programming language, extended with Generalized Algebraic Data Types (GADTs) [10, 49], as implemented in Glasgow Haskell [16]. GADTs provide a convenient *type-safe* framework for shallow embedding of *little languages* [3], which we use for symbolic representation of multisets, performable functions, predicates, equivalence and ordering relations. Apart from type safety, all other aspects of our library can be easily coded up in other general-purpose programming languages, both eager and lazy. Hudak and Fasel [26] provide a brief and gentle introduction to Haskell. Since we deliberately do not use monads, type classes or any other Haskell-specific language constructs except for GADTs, we believe basic knowledge of functional programming is sufficient for understanding the code we provide. All code included in the paper is proper Haskell, except if it contains "...", which indicates missing code.

For consistency we primarily use the term *multiset* for sets with positive multiplicities associated with their elements. They are commonly also called *bags*, which shows up occasionally in our code. We use *Cartesian product* or simply *product* to refer to the collection of all pairs constructed from two collections. We do not use the term "cross-product", which is also used in the database literature, because of its common, but different meaning as the Gibbs vector product in mathematics. We call the number of elements in a multiset its *cardinality* or its *count*, which is common in database systems. We reserve the word *size* for

[2] This is at the risk to the authors, but hopefully not to the readers, of making the result look less clever.

a measure of storage space. For example, the size of a multiset representation, an element of the data type `MSet a`, is the number of nodes when viewing it as a tree without node sharing.

1.2 Outline

Section 2 illustrates the idea of adding symbolic Cartesian products to a straightforward list-based implementation of multisets, adding other symbolic representations to the degree they yield performance benefits by exploiting algebraic properties. Section 3 describes discrimination-based joins, which are used in combination with symbolic products in the relational algebra core (Sect. 4) of GMP and its extension to SQL-style query operations (Sect. 5). Section 6 illustrates GMP by example, and Sect. 7 subjects the examples to a performance evaluation and comparison with corresponding formulations in MySQL. We conclude with a discussion of variations, related work, semantic issues in querying, and future work (Sect. 8). Section 2 is an intuitive, example-driven introduction into the step-by-step process of adding more and more dynamic symbolic computing, where we skip most of the details. Sections 3, 4 and 5, on the other hand, present the complete result as a *fait accompli*. There is an intended overlap of material to let the reader choose between a focused reading of the implementation of GMP, skipping the example-driven intuitive explanation; and skipping the three code-laden GMP implementation sections in favor of "getting the idea" from the intuitive explanation in Sect. 2 combined with the examples in Sect. 6. Reading all parts of the paper is, of course, encouraged at the slight risk of exposing the reader to, hopefully limited, irritation over a certain degree of repetitiveness.

2 GMP by example

Assume we have bank accounts represented as unique numeric identifiers, with their associated account holders' names and current account balances stored in separate collections `depositors` and `accounts`, respectively. Assume furthermore we are interested in returning the balance associated with each depositor's name.

In SQL this can be formulated as the conjunctive join query

```
SELECT depName, acctBalance
FROM   depositors, accounts
WHERE  depId = acctId
```

where the attributes `depId` and `depName` belong to the table `depositors`, and `acctId` and `acctBalance` belong to the table `accounts`.

2.1 A naïve list implementation

We can write this query as a functional expression that closely reflects the relational algebra structure of the SQL formulation. To do so, we introduce the following auxiliary function definitions:

```
(f *** g) (x, y) = (f x, g y)
(f .==. g) (x, y) = (f x == g y)
prod s t = [ (x, y) | x ← xs, y ← ys ]
```

They express parallel composition (componentwise function application), equijoin condition, and product on lists, respectively. With `depositors` and `accounts` represented as lists the query can now be expressed as

```
map (depName *** acctBalance)
    (filter (depId .==. acctId)
            (prod depositors accounts) )
```

The list combinators `map` and `filter` correspond to projection and selection, respectively, in relational algebra.

Coding relational algebra operations as straightforward list processing functions makes for an a priori attractive library for language-integrated querying. In practice, however, this is not viable for performance reasons. The culprit is the product function `prod`. In the example above, the computation `prod depositors accounts` *multiplies* the result *out* by nested iteration through `depositors` and `accounts`. Assuming both lists have n elements, this requires $\Theta(n^2)$ time.

2.2 Symbolic products to the rescue

The key idea in this paper is a simple one: Be lazy, really lazy, when evaluating costly functions. Ordinary lazy evaluation of `prod depositors accounts` is not enough: It avoids materializing the product, but eventually n^2 pairs are generated, even though the join condition subsequently filters all but n of them away.

We go one step further. We define a data type with an explicit data type constructor `X` for representing products symbolically. As a starting point, this gives us the following data type definition:

```
data MSet a where
  MSet :: [a] → MSet a
  X    :: MSet a → MSet b → MSet (a, b)
```

A multiset can be constructed from a list using `MSet` and by forming a symbolic product using `X`. Note that `X` not only defers multiplying out, it lets a function on multisets check whether its argument is a symbolic product and extract its component multisets in constant time. The purpose of introducing `X` is that there are algebraic properties of products that can be used to completely avoid multiplying them out in certain application contexts.

In our example there are two application contexts where symbolic products can be exploited:

- The predicate `depId .==. acctId` in the selection is an equijoin condition. When applied to `depositors `X` accounts`, we can employ an efficient join algorithm instead of multiplying out `depositors `X` accounts`.
- The function `depName *** acctBalance` works componentwise on its input pairs. Semantically, the result of mapping it over a product `s `X` t` is the product of mapping `depName` and `acctBalance` over `s` and `t`, respectively, which allows us to avoid multiplying out `s `X` t` altogether. Since the result of an equijoin is always a disjoint union of products, we exploit this by returning its result as a union of *symbolic* products.

To fully exploit symbolic products, we need to add other symbolic representations. For example, representing the result of equijoins efficiently requires representing *unions* of products. We do so by adding a constructor `U` for binary unions. In this fashion we arrive at the following data type for multisets, which we use in this paper:

```
data MSet a where
    MSet :: [a] → MSet a                  -- multiset from list
    U    :: MSet a → MSet a → MSet a      -- union
    X    :: MSet a → MSet b → MSet (a, b) -- product
```

We furthermore assume we have a function that multiplies out products and flattens unions into a list representation of a multiset:

```
list :: MSet a → [a]
```

We also need *symbolic* representations of functions and predicates used in projections and selections. We proceed to illustrate the design process of developing them, driven by our example query.

2.3 Predicates

Consider the subexpression

```
filter (depId .==. acctId) (prod depositors accounts)
```

in our query. We would like to replace `filter` by a function `select` that operates on multisets instead of lists. We need to represent those Boolean-valued functions symbolically, however, for which we can take advantage of symbolic products in second argument position. To that effect we introduce a data type of *predicates*, symbolic Boolean-valued functions, starting with a single constructor for representing arbitrary Boolean-valued functions:

```
data Pred a where
     Pred :: (a → Bool) → Pred a
     ...
```

The dots represent additional predicate constructors, which we subsequently add to achieve computational advantage over predicates represented as "black-box" Boolean-valued functions only.

We also need the function

```
sat :: Pred a → (a → Bool)
```

which returns the Boolean-valued function denoted by a predicate. It is straightforward to define and can be thought of as specifying the denotational semantics of predicates.

One class of predicates we can exploit are *equivalence join conditions*, which we also just call *join conditions*. They generalize equality predicates on attributes in relational algebra. A join condition is a predicate on pairs. It has three components: Two functions f, g and an equivalence relation E. A pair (x, y) satisfies the join condition if and only if $f(x) \equiv_E g(y)$, that is $f(x)$ and $g(y)$ are E-equivalent. To represent a join condition symbolically we introduce a constructor `Is` and add it to `Pred a`:

```
data Pred a where
     Pred :: (a → Bool) → Pred a
     Is   :: (a → k, b → k) → Equiv k → Pred (a, b)
```

The data type `Equiv k` of *equivalence representations* contains symbolic representations of equivalence relations. They are introduced by Henglein [23]. For the present purpose, all we need to know is that there exists an efficient *generic stable discriminator*

```
disc :: Equiv k → [(k, v)] → [[v]]
```

which partitions key-value pairs for an equivalence relation E represented by e in the following sense: `disc e` partitions the input into groups of values associated with E-equivalent keys. For example, for equality on the number type `Int` denoted by

```
eqInt32 :: Equiv Int
```

the expression `disc eqInt32 [(5,10), (8,20), (5,30)]` evaluates to `[[10, 30], [20]]`.

2.4 Selection

So far we have defined data types for multisets and predicates. Let us now define `select`. First we note that `select` takes predicates as inputs, not Boolean-valued functions:

```
select :: Pred a → MSet a → MSet a
```

Its default implementation converts the multiset into a list and filters it using the predicate:

```
select p s = MSet (filter (sat p) (list s)) -- default clause, must be last
```

If it were the only clause defining *select* there would be no point in having symbolic constructors since `list` multiplies out all products and flattens all unions. The additional clauses are special cases which avoid this multiplying out. For example, the clause for a join condition applied to a product is

```
select ((f, g) `Is` e) (s `X` t) = djoin (f, g) e (s, t)
```

where

```
djoin :: (a → k, b → k) → Equiv k → (MSet a, MSet b) → MSet (a, b)
djoin (f, g) e (s, t) = ... disc e ...
```

is an efficient generic join algorithm based on `disc`. The important point of the `select`-clause is not the particular choice of join algorithm invoked, but that `select` discovers *dynamically* when it is advantageous to branch off into an efficient join algorithm. Furthermore, it is important that `djoin` returns its output symbolically, as unions of products.

The selection in our example query can then be formulated as

```
select ((depId, acctId) `Is` eqInt32) (depositors `X` accounts)
```

It returns its result as a multiset of the form `(s1 `X` t1) `U` ... `U` (sn `X` tn)`. This ensures that the output size[3] is linearly bounded by the sum of the cardinalities

[3]Recall that this is not the *cardinality* of the output, but, up to a constant factor, the storage space required for representing an element of `MSet a`.

of the input multisets. This is an important difference to query evaluation in conventional RDBMSs, which follow the System R [37] query engine architecture. There, intermediate results are exclusively represented as streams of records, corresponding to lists in Haskell. To avoid unnecessarily multiplying out the component products of joins consumed by other parts of a query, they employ aggressive query optimization prior to issuing a query plan for execution.

There are more opportunities for exploiting properties of predicates and/or multisets; e.g. representing the constant-true predicate by `TT`, we can add the clause

```
select TT s = s
```

which just returns the second argument, without iterating through it.

2.5 Performable functions

Consider now the outermost projection applied in our query:

```
map (depName *** acctBalance) ...
```

We want to replace `map` with an efficient function `perform` operating on multisets. We call it `perform` instead of `project` since it maps arbitrary functions over multisets, not only projections as in relational algebra.

To represent functions symbolically we introduce, analogous to predicates, a data type of symbolic representations we call *performable functions*

```
data Func a b where
     Func :: (a → b) → Func a b
     ...
```

with an *extension* function

```
ext :: Func a b → (a → b)
```

returning the ordinary function denoted by a performable function. To represent functions operating componentwise, we add to `Func a b` a *parallel composition* constructor:

```
     Par :: Func (a, c) → Func (b, d) → Func (a, b) (c, d)
```

It lets us represent the projection `depName *** acctBalance` in our query symbolically:

```
Func depName `Par` Func acctBalance :: Func (Deposit, Account) (String, Int)
```

2.6 Projection

Let us now define `perform`, which maps a performable function over all elements of a multiset. Analogous to `select`, we note that the first argument should be symbolic, a performable function:

```
perform :: Func a b → MSet a → MSet b
```

Its default implementation converts the multiset into a list and maps the function over it:

```
perform f s = MSet (map (ext f) (list s)) -- default clause, must be last
```

The additional clauses are special cases for avoiding multiplying out products. For example, the clause for performing a parallel composition on a product is

```
perform (f `Par` g) (s `X` t) = perform f s `X` perform g t
```

and the clause for performing a function on a union is

```
perform f (s `U` t) = perform f s `U` perform f t
```

2.7 Query example in GMP

Our query example looks like this in GMP now:

```
perform (Func depName `Par` Func acctBalance)
        (select ((depId, acctId) `Is` eqInt32)
                (depositors `X` accounts))
```

It corresponds to the original list-based formulation. The important difference is that it executes in linear, not quadratic, time in the cardinalities of `depositors` and `accounts`. The `select` subexpression invokes a discrimination-based join, which returns its result in the form `(s1 `X` t1) `U` ... `U` (sn `X` tn)`. The two special clauses for `perform` above ensure that `perform (Func depName `Par` Func acctBalance)` is executed by mapping the component functions `depName` and `acctBalance` without multiplying the products or flattening the unions. The output is of linear *size* and is computed in linear time in the sum of the cardinalities of the inputs `depositors` and `accounts`. This is the case even though, in general, the *cardinality* of the output of the query may be as large as quadratic in the size of the input. Composing the query with `list` results in a query which executes in linear time in the sum of the cardinalities of the inputs and the output of the query.

3 Discrimination-based joins

We introduce a generic notion of joins and show how they can be implemented efficiently. We use neither hash-based nor sort-merge based joins, but introduce *discrimination-based* joins, a new technique for computing joins in worst-case linear time. As the name indicates, they are an application of *generic discrimination* [23], which we describe first.

3.1 Generic discrimination

Definition 1 (Discriminator) A function Δ is an *equivalence discriminator*, or just *discriminator*, for equivalence relation E if

1. (partitioning) Δ maps a list of key-value pairs to a list of groups, where each group is a list containing all the values with E-equivalent keys.

2. (parametricity) Δ is parametric with respect to the values:
   For all binary relations R, if $\mathbf{x}(I \times R)^*\mathbf{y}$ then $\Delta(\mathbf{x})R^{**}\Delta(\mathbf{y})$, where I is the identity relation.[4]

Δ is *stable* if it returns the elements in each group in the same positional order as they occur in the input.

The partitioning property expresses that a discriminator partitions the key-value pairs according to the given equivalence on the keys, but without actually returning the keys themselves. The parametricity property guarantees that a discriminator treats values as *satellite data*. Intuitively, values can be passed as pointers that are guaranteed not to be dereferenced during discrimination. Neither the order in which values are listed in a group nor the order of the groups themselves are fixed by these properties, however.

Discriminators can furthermore be equipped to be *partially* or *fully abstract*. Informally speaking, these are representation independence properties: Implementations of abstract data types such as sets (represented by lists) or pointers (represented by machine addresses) can be changed without changing the outputs of discriminators. Since representation independence is not central to the present paper, we refer to Henglein [23] for a detailed discussion.

Example 1 Consider $[(5, 10), (8, 20), (6, 30), (7, 40), (9, 50)]$ and let two keys be equivalent if and only if they are both even or both odd. A discriminator for this equivalence relation may return $[[10, 40, 50], [20, 30]]$ as a result: 10, 40 and 50 are associated with the odd keys in the input; likewise 20 and 30 are the values associated with the even keys. Another discriminator may return the groups in the opposite order ($[[20, 30], [10, 40, 50]]$) or indeed each group may be ordered differently, e.g. $[[50, 40, 10], [30, 20]]$.

A discriminator that returns $[[10, 40, 50], [20, 30]]$ for $[(5, 10), (8, 20), (6, 30), (7, 40), (9, 50)]$ returns $[[101, 407, 503], [202, 309]]$ for input $[(5, 101), (8, 202), (6, 309), (7, 407), (9, 503)]$ due to its parametricity property.

We are interested in a *generic discriminator*, which takes a specification of an equivalence relation as input and returns a discriminator for it. But how to specify an equivalence relation? Using binary Boolean-valued functions of type `k -> k -> Bool`, as is done in the Haskell library function `nubBy`, is not only unsafe (the Boolean-valued function might not represent an equivalence relation), a discriminator necessarily takes quadratic time if the only operation on keys it has is the Boolean-valued function [23]. Instead, Henglein [23] formulates an expressive domain-specific language of *equivalence representations*.

Equivalence representations provide a compositional way of specifying equivalence relations as symbolic values of type `Equiv k`. More precisely, each element of `Equiv k` denotes an equivalence relation on a *subset* of `k`. Figure 1 shows the definition of `Equiv k` and the generic function `eq`, which maps `e` to the binary Boolean-valued function corresponding to the equivalence relation denoted by `e`. The values v for which `eq e v v` terminates without error constitute the subset on which the equivalence relation is defined.

Recursively defined equivalence representations allow denoting equivalence relations on recursive types, such as lists and trees. Consider for example `listE e` defined by

```
listE :: Equiv t → Equiv [t]
```

[4]The operators $\times$ and $(.)^*$ extend binary relations pointwise to pairs, respectively lists.

```
data Equiv a where
  NatE  :: Int → Equiv Int
  TrivE :: Equiv t
  SumE  :: Equiv t1 → Equiv t2 → Equiv (Either t1 t2)
  ProdE :: Equiv t1 → Equiv t2 → Equiv (t1, t2)
  MapE  :: (t1 → t2) → Equiv t2 → Equiv t1
  BagE  :: Equiv t → Equiv [t]
  SetE  :: Equiv t → Equiv [t]

eq :: Equiv t → t → t → Bool
eq (NatE n) x y =
   if 0 ≤ x && x ≤ n && 0 ≤ y && y ≤ n
   then (x == y)
   else error "Argument out of range"
eq TrivE _ _ = True
eq (SumE e1 _) (Left x) (Left y) = eq e1 x y
eq (SumE _ _) (Left _) (Right _) = False
eq (SumE _ _) (Right _) (Left _) = False
eq (SumE _ e2) (Right x) (Right y) = eq e2 x y
eq (ProdE e1 e2) (x1, x2) (y1, y2) =
   eq e1 x1 y1 && eq e2 x2 y2
eq (MapE f e) x y = eq e (f x) (f y)
eq (BagE _) [] [] = True
eq (BagE _) [] (_ : _) = False
eq (BagE e) (x : xs') ys =
   case delete e x ys of Just ys' → eq (BagE e) xs' ys'
                         Nothing → False
   where
    delete :: Equiv t → t → [t] → Maybe [t]
    delete e v = subtract' []
       where subtract' _ [] = Nothing
             subtract' accum (x : xs) =
                if eq e x v then Just (accum ++ xs)
                            else subtract' (x : accum) xs
eq (SetE e) xs ys =
  all (member e xs) ys && all (member e ys) xs
  where member :: Equiv t → [t] → t → Bool
        member _ [] _ = False
        member e (x : xs) v = eq e v x || member e xs v
```

Fig. 1 Equivalence representations and the (binary Boolean-valued function representations of the) equivalence relations denoted by them

```
listE e = MapE fromList (SumE TrivE (ProdE e (listE e)))

fromList :: [t] → Either () (t, [t])
fromList []       = Left ()
fromList (x : xs) = Right (x, xs)
```

It denotes the elementwise extension of the equivalence denoted by `e` to lists such that $[v_1, \ldots, v_m]$ and $[w_1, \ldots, w_n]$ are `listE e`-equivalent if and only if $m = n$, and v_i and w_i are `e`-equivalent for all $i \in \{1, \ldots, m\}$.

Componentwise equivalence is not the only useful equivalence on lists. Two lists are *multiset-equivalent* under `e`, denoted by `BagE e`, if one of them can be permuted such that it is `listE e`-equivalent to the other. They are *set-equivalent* under `e`, denoted by `SetE e`, if each element in one list is `e`-equivalent to some element in the other list.

Equivalence representations provide an expressive language for defining useful equivalence relations. It is for example possible to define `eqInt32` and `eqString8`, the equalities on 32-bit integers and on 8-bit character strings, respectively. We can also define new equivalence constructors. For example,

```
maybeE :: Equiv k → Equiv (Maybe k)
maybeE e = MapE (maybe (Left ()) Right) (SumE TrivE e)
```

lifts an equivalence `e` to an equivalence on the `Maybe` data type such that two values are equivalent if they both are `Nothing` or they both have the form `Just` v such that the arguments of `Just` are `e`-equivalent.

The *size* of a list is its length plus the sum of the sizes of its elements. The key result about generic discriminators is the following:

Theorem 1 (Linear-time discrimination [23]) *There exists a generic discriminator*

```
disc :: Equiv k -> [(k, v)] -> [[v]]
```

such that for each `e` *from a class* $\mathcal{L}$ *of equivalence representations*, `disc e` *executes in linear time in the size of its input list. Furthermore*, `disc` *can be implemented to return only fully abstract discriminators.*

The class $\mathcal{L}$ is not spelled out here. Suffice it to say that it is quite large, including structural equality on all first-order types and all equivalence representations occurring in this paper.

Combining linear time performance with full abstraction is, at first sight, a surprising result. It means that a discriminator behaves *as if* it has pairwise equivalence tests as the *only* operation on keys, which suggests that it should take quadratic time. In its implementation, however, it uses additional operations before stitching up the output to make their use unobservable. Ensuring full abstraction requires input instrumentation and an extra pass over the output of a simpler discriminator. In our example applications below we have used the simpler discriminator.

The generic discriminator can be used to partition values according to an equivalence relation by associating them with themselves prior to discrimination:

```
part :: Equiv t → [t] → [[t]]
part e xs = disc e [ (x, x) | x ← xs ]
```

Using `part` we can eliminate all equivalence class duplicates in an input list; that is, return exactly one representative from each group of equivalent values.

```
reps :: Equiv t → [t] → [t]
reps e xs = [ head ys | ys ← part e xs ]
```

3.2 Discrimination-based join on lists

Definition 2 A binary predicate P is an *equivalence join condition*, or just *join condition*, if there exist functions f, g and equivalence relation E such that $P(x, y) \iff f(x) \equiv_E g(y)$.

We can now define a generic join function `djoinL` (for discrimination-based join on lists):

```
djoinL :: (a → k, b → k) → Equiv k → ([a], [b]) → [(a, b)]
```

The call `djoinL (f, g) e (xs, ys)` returns all pairs from the Cartesian list product of `xs` and `ys` that satisfy the join condition given by the functions `f, g` and the equivalence representation `e`. It works as follows:

1. Associate each element in `xs` with a key, computed by applying `f` to the value, and tag the value with "Left" to indicate that it comes from the left (first) list argument `xs`.
2. Do the analogous step for `ys`, using `g` and tagging each value with "Right".
3. Concatenate the two lists of key/tagged-value pairs.
4. Discriminate the concatenated list using `disc e`. The result is a list of *groups*, each containing elements from `xs` and `ys` with `e`-equivalent keys.
5. Split each group into those elements coming from `xs` ("Left") and those coming from `ys` ("Right"), and form the Cartesian list product of these two subgroups.
6. Concatenate all these Cartesian list products and return them.

Here is the Haskell code:

```
djoinL :: (a → k, b → k) → Equiv k → ([a], [b]) → [(a, b)]
djoinL (f, g) e (xs, ys) = concat (map mult fprods)
     where us = [ (f x, Left x)  | x ← xs ]    -- [(k, Either a b)]
           vs = [ (g y, Right y) | y ← ys ]    -- [(k, Either a b)]
           bs = disc e (us ++ vs)              -- [[Either a b]]
           fprods = map split bs               -- [([a], [b])]
```

It uses auxiliary functions for multiplying out a Cartesian list product

```
mult :: ([a], [b]) → [(a, b)]
mult (vs, ws) = [ (v, w) | v ← vs, w ← ws ]
```

and splitting tagged values into those tagged `Left` and those tagged `Right`.

```
split :: [Either a b] → ([a], [b])
split [] = ([], [])
split (v : vs) = let (lefts, rights) = split vs in
                 case v of Left v' → (v' : lefts, rights)
                           Right v' → (lefts, v' : rights)
```

If follows from Theorem 1 that `djoinL` executes in worst-case linear time. Since the output can be quadratically larger than the input, linearity is in both the input and the output.

Theorem 2 *Let* `f`, `g` *be constant-time computable and let* `e` *be an equivalence representation with a linear-time discriminator* `disc e`. *Then* `djoinL (f, g) e (xs, ys)` *executes in time* $O(m+n+o)$ *where* n *is the size of* `xs`, m *the size of* `ys`, *and* o *the length of the output*.

Note the subtle difference: o is only the *length* of the output list whereas m and n are the lengths of the input lists, *plus* the *sizes* of their elements. Observe also that, except for the final multiplying out of the pairs of subgroups, `djoinL` runs in time $O(m+n)$, independent of the output length.

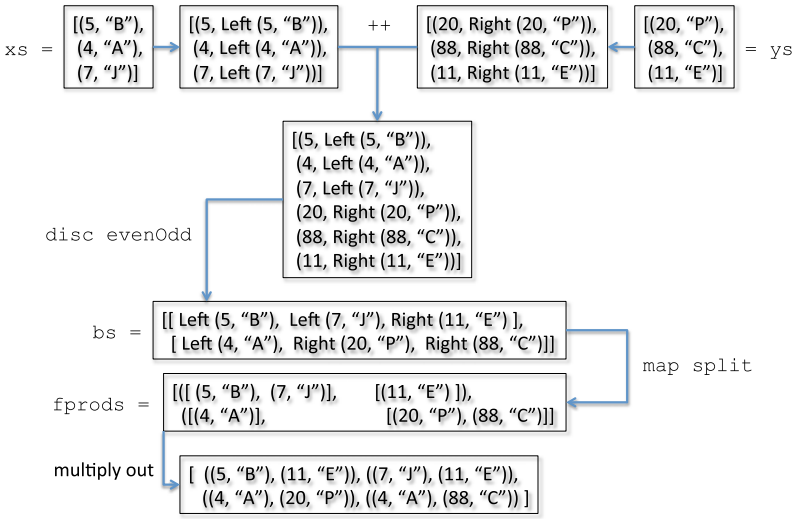

Fig. 2 Example execution of discrimination-based join

3.3 A join example

Figure 2 illustrates execution of the call

```
djoinL (fst, fst) evenOdd
     ([(5, "B"), (4, "A"), (7, "J")], [(20, "P"), (88, "C"), (11, "E")])
```

where two numbers are `evenOdd`-equivalent if and only if they both are even or both are odd. It is specified by the equivalence representation

```
evenOdd :: Equiv Int
evenOdd = MapE (`mod` 2) (NatE 1)
```

4 Relational algebra programming with symbolic products

In this section we present the complete code for adding symbolic products for efficient multiset processing by dynamic symbolic computation. The ideas are the same as in Henglein [24], but the concrete representations are slightly different to make them practically more expressive and compatible with SQL semantics. In particular, our bulk data represent multisets, not sets, and disjunction is added here.

4.1 Multisets

A *multiset*, also called *bag*, is an unordered collection that may contain duplicate elements. We use lists as a basic representation of multisets and add symbolic constructors for union and product.

```
data MSet a where
    MSet  :: [a] → MSet a                    -- multiset from list
    U     :: MSet a → MSet a → MSet a        -- union
    X     :: MSet a → MSet b → MSet (a, b) -- product
```

The empty multiset is denoted by the empty list:

```
empty = MSet []
```

A list representation of a multiset can be prepended to a given argument list

```
prepend :: MSet a → [a] → [a]
prepend (MSet xs) zs = xs ++ zs
prepend (s `U` t) zs = prepend s (prepend t zs)
prepend (s `X` t) zs = foldr (\x us → foldr (\y vs → (x, y) : vs) us ys) zs xs
      where xs = list s
            ys = list t
```

and prepending to the empty list converts a multiset to a list representing it:

```
list :: MSet a → [a]
list s = prepend s []
```

The function `list` multiplies symbolic products out and flattens symbolic unions, resulting in a potentially superlinear blow-up in time and space. Since this is the *only* nonlinear code we use we can easily identify where we incur potentially nonlinear complexity in the subsequent code. Observe that `list` encapsulates not only a computational, but also a semantic problem: it is *nondeterministic* in the sense that two different representations of the *same* multiset may yield different results; e.g. `MSet [1, 2, 3]` and `MSet [3, 2, 1]` denote the same multiset, but `list` returns different lists for them. In particular, representation changes of multisets can be observed by `list`. Since all other operations we use are semantically deterministic, we can identify potential nondeterminism by the presence of `list`.

The *cardinality* or *count* of a multiset is the number of elements it contains:

```
count :: MSet a → Int
count (MSet xs) = length xs
count (s `U` t) = count s + count t
count (s `X` t) = count s * count t
```

Note the last clause: It computes the cardinality without multiplying out the product.

4.2 Performable functions

We define symbolic representations of *performable functions*, functions that can be applied to each element of a multiset:

```
data Func a b where
  Func :: (a → b) → Func a b                          -- from ordinary function
  Par  :: Func a b → Func c d → Func (a, c) (b, d)    -- parallel composition
  Fst  :: Func (a, b) a                               -- first component of pair
  Snd  :: Func (a, b) b                               -- second component of pair
```

Any user-definable function `f` can be turned into a performable function by applying the constructor `Func` to it. The constructors `Fst` and `Snd` represent the corresponding projections on pairs. The *parallel composition* `Par f g` of performable functions `f` and `g` operates on the components of pairs independently.

The function `ext` maps performable functions to their extensions as ordinary functions:

```
ext :: Func a b → a → b
ext (Func f) x         = f x
ext (Par f1 f2) (x, y) = (ext f1 x, ext f2 y)
ext Fst (x, _)         = x
ext Snd (_, y)         = y
```

4.3 Predicates

Similar to performable functions, we allow arbitrary Boolean functions as *predicates*, but maintain symbolic representations for constant-true (`TT`), constant-false (`FF`), sequential conjunction (`SAnd`), parallel conjunction (`PAnd`), sequential disjunction (`SOr`), parallel disjunction (`POr`), and join condition (`Is`).

```
data Pred a where
    Pred :: (a → Bool) → Pred a              -- from Boolean-valued function
    TT   :: Pred a                           -- constant true
    FF   :: Pred a                           -- constant false
    SAnd :: Pred a → Pred a → Pred a         -- sequential conjunction
    PAnd :: Pred a → Pred b → Pred (a, b)    -- parallel conjunction
    SOr  :: Pred a → Pred a → Pred a         -- sequential disjunction
    POr  :: Pred a → Pred b → Pred (a, b)    -- parallel disjunction
    Is   :: (a → k, b → k) → Equiv k → Pred (a, b) -- join condition
```

Their extension as Boolean-valued functions and thus the semantics of predicates is given by `sat`:

```
sat :: Pred a → a → Bool
sat (Pred f) x            = f x
sat TT _                  = True
sat FF _                  = False
sat (p1 `SAnd` p2) x      = (sat p1 x && sat p2 x)
sat (p1 `PAnd` p2) (x, y) = sat p1 x && sat p2 y
sat (p1 `SOr` p2) x       = sat p1 x || sat p2 x
sat (p1 `POr` p2) (x, y)  = sat p1 x || sat p2 y
sat ((f, g) `Is` e) (x, y) = eq e (f x) (g y)
```

Observe that `PAnd` and `POr` are analogous to `Par`: They combine two predicates to a predicate on pairs. The triple `(f, g) `Is` e` represents a *join condition*; see Definition 2.

4.4 Selection

We now define `select`, which corresponds to the relation algebra operation of *selection*:

```
select :: Pred a → MSet a → MSet a
select TT s                    = s
select FF s                    = empty
select p (s `U` t)             = select p s `U` select p t
select (p `SAnd` q) s          = select q (select p s)
select (p `SOr` q) s           = select p s `U` select q s
select (p `PAnd` q)  (s `X` t) = select p s `X` select q t
select (p `POr` q) (s `X` t)   = (select p s `X` t) `U` (s `X` select q t)
select ((f, g) `Is` e) (s `X` t) = djoin (f, g) e (s, t)
select p s                     = MSet (filter (sat p) (list s)) -- default
```

The last clause is a catch-all clause, which performs selection by filtering after multiplying the multiset out to a list. The clauses before it represent efficiency improvements, in some cases resulting in potentially asymptotic speed-ups. For example, the clauses for constant predicates avoid iterating through the multiset argument, and the clauses

```
select (p `PAnd` q)  (s `X` t)   = select p s `X` select q t
select (p `POr` q) (s `X` t)     = (select p s `X` t) `U` (s `X` select q t)
select ((f, g) `Is` e) (s `X` t) = djoin (f, g) e (s, t)
```

avoid naïvely multiplying out products. In the first clause, and similarly in the second clause, we exploit that the results of parallel conjunction and disjunction over products can themselves be expressed as (unions of) products. In the third clause we see that it is possible to dynamically discover when a join condition is applied to a product, which allows us to employ an efficient join-algorithm.

The algorithm `djoin` is *almost* the same as `djoinL` in Sect. 3.2. It operates on multisets instead of lists, however. Most importantly, it avoids multiplying out the subgroups in the final step, returning them as unions of symbolic products instead.

```
djoin :: (a → k, b → k) → Equiv k → (MSet a, MSet b) → MSet (a, b)
djoin (f, g) e (s, t) =
     foldr (\(vs, ws) s → (MSet vs `X` MSet ws) `U` s) empty fprods
     where xs = [ (f r, Left r) | r ← list s ]
           ys = [ (g r, Right r) | r ← list t ]
           bs = disc e (xs ++ ys)
           fprods = map split bs
```

Recall that

```
split :: [Either a b] → ([a], [b])
```

splits tagged values according to their tag.

4.5 Projection

We call the functional that maps a performable function over a multiset `perform`. It corresponds to *projection* in relational algebra. It is defined by the following clauses:

```
perform :: Func a b → MSet a → MSet b
perform f (s `U` t)         = perform f s `U` perform f t
perform (Par f g) (s `X` t) = perform f s `X` perform g t
perform Fst (s `X` t)       = count t `times` s
perform Snd (s `X` t)       = count s `times` t
perform f s                 = MSet (map (ext f) (list s)) -- default clause
```

In the clause

```
perform (Par f g) (s `X` t) = perform f s `X` perform g t
```

the product is not multiplied out, and the result is computed symbolically. Analogous to the `select`-clauses for parallel conjunction and disjunction, this illustrates why it is important to maintain a symbolic representation of *parallel* functional composition. If `Par f g` is only available as a black-box function, there is no alternative to multiplying out the product

before applying the function to each pair in the product one at a time, as seen in the default clause for `perform`.

In a set-theoretic semantics mapping a projection over a symbolic product could be performed by simply returning the corresponding component set, once we have checked that the other set is nonempty. With multisets, however, we need to return as many copies as the other component has elements:

```
perform Fst (s `X` t)       = count t `times` s
perform Snd (s `X` t)       = count s `times` t
```

where the `times` operator defines iterated multiset union:

```
times :: Int → MSet a → MSet a
times 0 s = empty
times 1 s = s
times n s = s `U` times (n - 1) s
```

4.6 Union and product

Operations corresponding to union and Cartesian product in relational algebra are already defined: They are the multiset constructors `U` and `X`, respectively.

5 Generic SQL

We now present the extension of the relational algebra core of Sect. 4 to generic versions of the SQL operations EXCEPT, DISTINCT, GROUP BY, ORDER BY, HAVING, and aggregation functions.

5.1 EXCEPT

The SQL difference operator EXCEPT is based on a notion of equality. We generalize it to an operation, `diff`, that removes all *equivalent* elements from a multiset, where the equivalence is user-definable:

```
diff :: Equiv k → MSet k → MSet k → MSet k
diff e s t = foldr include empty bs
  where
  xs = [ (x, Left x) | x ← list s ]
  ys = [ (y, Right y) | y ← list t ]
  bs = disc e (ys ++ xs)
  untag (Left x) = x
  untag (Right _) = error "Impossible: Erroneously tagged value"
  include b@(Left x : _) s =  MSet (map untag b) `U` s
  include b@(Right y : _) s = s
  include [] s = error "Impossible: Application to empty block"
```

It resembles `djoin` in Sect. 4.4, with only the postprocessing after discrimination being different. Because of stability of `disc` the elements from the multiset `t` to be removed (up to equivalence) from `s` occur first in each group output by the discriminator, if the group contains an element from `t` at all. Consequently, if and only if the first element of a group is an element from `s` we can conclude that it has no equivalent element occurring in `t`, and we can place the group in the output. For example,

```
diff (MapE code eqString8) countries european
```

returns all the countries that are not European (where we assume that we have a multiset `european` of European countries available).

Other operations such as intersection and semijoins can be implemented in a similar fashion.

5.2 DISTINCT

The SQL DISTINCT clause removes duplicate values from a multiset, effectively returning the underlying set. As for EXCEPT this requires a notion of equality on the elements of a multiset. As before we allow arbitrary user-definable equivalence relations. We can code DISTINCT as the binary operation `coalesceBy`, which uses the generic function `reps` defined at the end of Sect. 3.1 to retain only one representative from each equivalence class in a multiset.

```
coalesceBy :: Equiv a → MSet a → MSet a
coalesceBy e s = MSet (reps e (list s))
```

For example,

```
coalesceBy eqString8 (perform (Func language) countryLanguage)
```

first forms the multiset of all languages and then returns the languages without duplicates.

5.3 GROUP BY

The SQL GROUP BY clause partitions a multiset according to an equivalence relation specified by a so-called *grouping list*. All such equivalences—and more—are definable using the equivalence representations in Fig. 1. The GROUP BY clause can thus be defined as a binary operation that takes an equivalence representation and a multiset, and returns a partition represented as a multiset of multisets.

```
groupBy :: Equiv a →  MSet a → MSet (MSet a)
groupBy e s = MSet (map MSet (part e (list s)))
```

For example,

```
groupBy (MapE region eqString8) countries
```

partitions the countries into groups of countries in the same region.

5.4 ORDER BY

The SQL ORDER BY clause is handled completely analogously to GROUP BY. Instead of an equivalence representation it requires the specification of an ordering relation in the form of an *order representation*. The complete definition of order representations (type `Order a`) and their extension as the order-representation generic binary Boolean-valued function `lte` representing ordering relations is given in Fig. 3.

```
data Order a where
   Nat   :: Int → Order Int
   Triv  :: Order t
   SumL  :: Order t1 → Order t2 → Order (Either t1 t2)
   PairL :: Order t1 → Order t2 → Order (t1, t2)
   MapO  :: (t1 → t2) → Order t2 → Order t1
   ListL :: Order t → Order [t]
   BagO  :: Order t → Order [t]
   SetO  :: Order t → Order [t]
   Inv   :: Order t → Order t

lte (Nat n) x y =
    if 0 ≤ x && x ≤ n && 0 ≤ y && y ≤ n
    then x ≤ y
    else error "Argument out of allowed range"
lte Triv _ _ = True
lte (SumL r1 r2) (Left x) (Left y) = lte r1 x y
lte (SumL r1 r2) (Left _) (Right _) = True
lte (SumL r1 r2) (Right _) (Left _) = False
lte (SumL r1 r2) (Right x) (Right y) = lte r2 x y
lte (PairL r1 r2) (x1, x2) (y1, y2) =
    lte r1 x1 y1 &&
    if lte r1 y1 x1 then lte r2 x2 y2 else True
lte (MapO f r) x y = lte r (f x) (f y)
lte (ListL r) xs ys = lte (listL r) xs ys
lte (BagO r) xs ys = lte (MapO (sort r) (listL r)) xs ys
lte (SetO r) xs ys = lte (MapO (usort r) (listL r)) xs ys
lte (Inv r) x y = lte r y x
```

Fig. 3 Order representations and the (characteristic functions of) ordering relations (total preorders) denoted by them

The definition of `lte` uses a number of auxiliary functions. The function

```
listL :: Order t → Order [t]
listL r = MapO fromList (SumL Triv (ProdL r (listL r)))
```

constructs an order representation denoting the lexicographic ordering on lists. Note that its definition is completely analogous to `listE`. The function

```
sort :: Order a → [a] → [a]
```

is an order-representation generic sorting function. It can be implemented using generic order discrimination, which generalizes distributive sorting [22, 23] from finite types and strings to all order-representation denotable orders, or by employing classical comparison-based sorting; e.g.

```
sort r xs = csort (lte r) xs
```

where `csort` is a comparison-parameterized function implemented using Quicksort, Mergesort, Heapsort or similar.

A variant of `sort` is the *unique-sorting* function

```
usort :: Order a → [a] → [a]
```

which sorts the input, but retains only the first element in each run of equivalent elements. (We omit the code here.) It is used in the definition of comparing lists under set ordering `SetO r`: Two lists are compared by first unique-sorting each under `r` and then performing a lexicographic comparison of the resulting lists.

With order representations in place, we can define `orderBy` as an order-representation generic function:

```
orderBy :: Order a → MSet a → [a]
orderBy r s = sort r (list s)
```

For example,

```
orderBy (MapO population (Inv ordInt32)) countries
```

returns the countries as before, but now as a sorted list in descending order of their population. Here `ordInt32` denotes the standard order on 32-bit integers. Note that `MapO` is analogous to `MapE`: it induces an ordering on the domain of a function from an ordering on its codomain.

5.5 HAVING

The SQL HAVING clause is used in combination with the GROUP BY clause in SQL. It retains only those groups of values that satisfy a user-provided predicate (on multisets!). Since our multisets can contain elements of any type, not just primitive ordered types as in SQL, HAVING requires no special attention, but can be coded using `select`:

```
having :: MSet (MSet a) → Pred (MSet a) → MSet (MSet a)
having s p = select p s
```

For example,

```
groupBy (MapE region eqString8) countries
        `having` (Pred (\reg → count reg ≥ 5))
```

computes the regions with at least five countries each.

5.6 Aggregation

Aggregation is the reduction of a multiset by an associative-commutive function such as constant, addition, multiplication, maximum and minimum. It is arguably the most significant extension of SQL over relational algebra in terms of expressiveness.

Since all inputs and outputs of SQL queries are required to be multisets of primitive values, different intermediate data types such as the result of grouping operations need to be coupled with operations yielding primitive values again. This is why aggregation functions are coupled with grouping and ordering clauses in SQL. We are not restricted in the same fashion and can completely separate aggregation from grouping. Notably, any binary function and start value can be extended to an aggregation function on multisets:

```
reduce :: (a → a → a, a) → MSet a → a
reduce (f, n) s = foldr f n (list s)
```

```
CREATE TABLE City (
  ID int(11) NOT NULL,
  Name char(35) NOT NULL,
  PRIMARY KEY (ID)
);
CREATE TABLE Country (
  Code char(3) NOT NULL,
  Name char(52) NOT NULL,
  Population int(11) NOT NULL,
  Region char(26) NOT NULL,
  Capital int(11),
  PRIMARY KEY (Code)
);
CREATE TABLE CountryLanguage (
  CountryCode char(3) NOT NULL,
  Language char(30) NOT NULL,
  IsOfficial char(1) NOT NULL,
  Percentage float(4,1) NOT NULL,
  PRIMARY KEY  (CountryCode, Language),
  KEY CountryCode (CountryCode),
  CONSTRAINT FOREIGN KEY (CountryCode)
              REFERENCES Country (Code)
);
```

```
data City = City {
      cid :: Int,
      cname :: String
    } deriving (Eq, Show, Read)

data Country = Country {
      code :: String,
      name :: String,
      population :: Int,
      region :: String,
      capital :: Maybe Int
    } deriving (Eq, Show, Read)

data CountryLanguage = CountryLanguage {
      countryCode :: String,
      language :: String,
      isOfficial :: Bool,
      percentage :: Float
    } deriving (Eq, Show, Read)
```

Fig. 4 World Database SQL schemas and corresponding Haskell record declarations, some columns of the original data set are left out for clarity of presentation

If the input function f satisfies $f\ v_1\ (f\ v_2\ w) = f\ v_2\ (f\ v_1\ w)$, `reduce` yields a semantically deterministic function; otherwise, the result may depend on the order in which the elements of the multiset are enumerated by `list`. For example,

```
reduce (union, empty) (groupBy (MapE region eqString8) countries
                               `having` (Pred (\reg → count reg ≤ 5)))
```

computes the countries belonging to a region with at most five members each.

6 Examples

To demonstrate how GMP works we present a series of examples. First, we show how to express the same queries both in SQL and by using our library. Second, we show how our library can be used to express queries over non-normalized data.

6.1 World database

For our first series of examples we use a simplified version of the world sample data from the MySQL Project [15]. The database consists of three tables declared by the schemas in Fig. 4, shown together with the corresponding Haskell record declarations.

6.1.1 Finding the capital

The SQL query for finding the capital of each country is:

```
SELECT City.Name, Country.Name
FROM   City, Country
WHERE  ID = Capital
```

To write a similar query with our library, we need to handle an issue that SQL brushes under the carpet. Notice that the `Capital` column does not have a `NOT NULL` constraint in the SQL schema in Fig. 4 and, correspondingly, the `capital` field in the Haskell record has type `Maybe Int`. Thus, what should be done with countries that have a `NULL` value instead of an integer key in the `Capital` column?

Our first solution is a query that uses `PAnd` to only select those countries that have a capital, and we use a join condition to match the remaining identifiers. Finally we use `perform` to project out the `cname` and `name` fields:

```
findCapitals cities countries =
    perform (Func cname `Par` Func name)
            (select ((TT `PAnd` Pred hasCapital) `SAnd`
                    ((cid, value . capital) `Is` eqInt32))
                    (cities `X` countries))
    where hasCapital = maybe False (\_ → True) . capital
          value (Just x) = x
```

While `findCapitals` computes the right result, the predicate used with `select` is not as elegant as we would like. Thus, for our second solution we define a specialized equivalence relation that works similar to the SQL semantics. That is, we lift the matching of fields to take `Nothing` values into account. This is done using the `maybeE` equivalence constructor defined in Sect. 3.1:

```
findCapitals' :: MSet City → MSet Country → MSet (String, String)
findCapitals' cities countries =
    perform (Func cname `Par` Func name)
            (select ((Just . cid, capital) `Is` maybeE eqInt32)
                    (cities `X` countries))
```

This computes the same result as our first formulation of `findCapitals` and, furthermore, it can be thought of as a mechanical translation of the original SQL query. Note that our translation does not need to iterate through the full Cartesian product. Our query only uses time linear in the *sum* of the cardinalities of `cities` and `countries`.

6.1.2 Group by language

To construct aggregate queries over groups of countries with the same official language, we need the skeleton SQL query:

```
SELECT Language, ...
FROM   CountryLanguage, Country
WHERE  IsOfficial = 'T' AND Code = CountryCode
GROUP BY Language
```

where the ellipses denote the computation we want to perform on each group.

With our library we can first compute the grouping we want:

```
groupByLanguage :: MSet Country → MSet CountryLanguage
                                → MSet (MSet (CountryLanguage, Country))
groupByLanguage countries countryLanguage =
    groupBy (MapE (language . fst) eqString8)
            (select ((countryCode, code) `Is` eqString8)
                    (select (Pred isOfficial) countryLanguage `X` countries))
```

Because we now have an explicit representation of the grouping, we can reuse it for various "reports". First, we can generate a multiset of pairs of a language and a list of countries where that language is the official language:

```
countriesWithSameLang :: MSet (MSet (CountryLanguage, Country))
                                          → MSet (String, [String])
countriesWithSameLang groupByLang =
    perform (Func report) groupByLang
    where report bag = let ls = list bag
                           lang = language $ fst $ head ls
                           names = map (name . snd) ls
                       in (lang, names)
```

Second, we can generate a top 10 of the most spoken official languages:

```
languageTop10 :: MSet (MSet (CountryLanguage, Country)) → [(String, Int)]
languageTop10 groupByLang =
    take 10 $ orderBy sumOrder (perform (Func report) groupByLang)
    where report bag = (lang bag, sum $ list $ perform (Func summary) bag)
          lang = language . fst . head . list
          summary (l, c) = percentage l `percentOf` population c
          percentOf p r = round $ (p * fromIntegral r) / 100
          sumOrder = Inv $ MapO snd ordInt32
```

6.2 Working with non-normalized data

Our second series of examples does not have natural SQL counterparts, as we will be working with non-normalized data, namely lists and trees. Our running example is a custom data type of *file system elements*, which represent snapshots of a file system directory structure, without the contents of the files it contains:

```
data FilesystemElem = File FilePath
                    | Directory FilePath [FilesystemElem]
```

where `FilePath` is synonymous with `String`. (How to generate a `FilesystemElem` from an actual file system is outside the scope of this paper.)

6.2.1 Finding duplicate file names

To find files with the same name, but in different places in the file system, we first flatten the file system to a list of files with the full path represented as a list in reverse order. We construct a product of the list with itself, and select those pairs that have the same file name, but different paths:

```
findDuplicates :: FilesystemElem → MSet ([FilePath], [FilePath])
findDuplicates fs = select (((head, head) `Is` eqString8)
                              `SAnd` (Pred (uncurry (/=))))
                           (fsBag `X` fsBag)
    where fsBag = MSet (flatten fs)
```

Recall that `eqString8` denotes equality on strings of 8-bit characters. The function `head` returns the first element of a list, and `uncurry (/=)` tests that the two components of a pair are different.

6.2.2 Finding duplicate directories

To find directories with the same content based on file and directory names, we first flatten the file system into a list of pairs: the full path of the directory and a file system element, one for each `Directory` element in the file system. Then we form a product of the list with itself and select pairs that are duplicates of each other, but with different paths:

```
findDuplicateDirs :: FilesystemElem →
         MSet (([FilePath], [FilesystemElem]), ([FilePath], [FilesystemElem]))
findDuplicateDirs fs = select (((snd, snd) `Is` dirEq)
                                `SAnd` (Pred $ \((d1,_), (d2,_)) → d1 /= d2))
                               (fsNonEmpty `X` fsNonEmpty)
    where fsBag = MSet $ dirs [] fs
          fsNonEmpty = select (Pred (not . null . snd)) fsBag
          dirEq = SetE $ MapE fromFs (SumE eqString8 (ProdE eqString8 dirEq))
          fromFs (File f) = Left f
          fromFs (Directory d fs) = Right(d, fs)
          dirs path (Directory d cont) = (fullpath, cont) : subdirs
              where fullpath = d : path
                    subdirs = [d| c ← cont, d ← dirs fullpath c]
          dirs _ _ = []
```

But what exactly is `dirEq`, which captures what it means for the contents of a (sub)directory to be a "duplicate" of another?

To start with, note that the type `FilesystemElem` is isomorphic to the type `Either FilePath (FilePath [FilesystemElem])`. One direction of the isomorphism can be defined by

```
fromFs (File f) = Left f
fromFs (Directory d fs) = Right(d, fs)
```

With this we can recursively define *structural equality* on file system elements:

```
fsEq0 :: Equiv FilesystemElem
fsEq0 = MapE fromFs (SumE eqString8
                          (ProdE eqString8 (listE fsEq0)))
```

In words, two file system elements are structurally equal if they both are either files with the same name or both are directories with the same name and containing `FilesystemElems` that are, recursively, pairwise structurally equal. As it is not guaranteed that the files and subdirectories in a directory are listed in the same order, however, we need to discover cases where a directory is a duplicate of another even though their contents are listed in different orders. This can be accomplished by replacing `listE` by `BagE`, the multiset-equivalence constructor:

```
fsEq1 :: Equiv FilesystemElem
fsEq1 = MapE fromFs (SumE eqString8
                          (ProdE eqString8 (BagE fsEq1)))
```

Using `SetE` instead of `BagE` we can even allow inadvertently duplicated file names and subdirectories and still identify them as duplicates:

```
fsEq2 :: Equiv FilesystemElem
fsEq2 = MapE fromFs (SumE eqString8
                          (ProdE eqString8 (SetE fsEq2)))
```

Now we have the desired definition of `dirEq`:

```
dirEq = SetE fsEq2
```

6.2.3 Skipping the product

In the previous two examples we have used the SQL idiom of using self-join before filtering pairs for non-duplicates. However, it turns out that for this particular pattern of queries, it is not the most efficient approach to follow the SQL idiom. Instead, we can just use `groupBy` directly on the (flattened) data, without first creating the product. Thus, an alternative formulation of the `findDuplicates` function is:

```
findDuplicates' :: FilesystemElem → MSet (MSet [FilePath])
findDuplicates' fs = select (Pred (\s → count s ≥ 2)) (groupBy fpEq fsBag)
    where fsBag = MSet (flatten fs)
          fpEq = MapE head eqString8
```

This function returns a multiset of multisets, each of cardinality at least 2 and having the same name, but different paths. Note, that this is a more succinct representation of the result compared to `findDuplicates`, which uses the SQL self-join idiom.

7 Performance

To give a *qualitative* illustration of the performance profile of our library we give performance measures for some of the examples presented in the previous sections. For comparison we also present numbers for comparative SQL queries using MySQL. The purpose of this section is *not* to give a rigorous performance evaluation of our library, as the code is not optimized and we use naïve data structures in many places (for instance, we use the standard Haskell representation of strings as lists of characters), nor do we try to claim to be better than a traditional SQL database systems. What we strive to illustrate is (1) that our library displays the predicted asymptotic running time, and (2) that its performance is in the same ballpark as a traditional SQL database system supporting efficient implementation of binary join queries.

All benchmark tests were performed on a lightly loaded Mac Book Pro with a Intel Core i5 CPU and 8 GB of RAM. To orchestrate the benchmarks we use the Haskell library criterion [33], which automatically ensures that the benchmarks are iterated enough times to fit with the resolution of the clock. Furthermore, the criterion performs a bootstrap analysis on the timings to check that spikes in the load from other programs running on the computer do not skew the results. Each timing reported is the average of 100 samplings. Since the standard deviations of these averages are all negligible, we do not report them.

7.1 World database

Our first set of benchmarks are three queries on the world database introduced in Sect. 6.1: find the capital, find groups of countries that have the same official language, and rank languages according to how many people speak the language.

Table 1 shows the performance numbers for the three benchmarks. To find the groups of countries that have the same official language we have used the SQL query

Table 1 Running times (in milliseconds) for queries over the world database

	findCapitals'	countriesWithSameLang	languageRank
Multiset	9.6	1.8	2.1
MySQL	1.1	1.9	2.2

```
SELECT Language, GROUP_CONCAT(Country.Name)
FROM   CountryLanguage, Country
WHERE  IsOfficial = 'T' AND Code = CountryCode
GROUP BY Language
```

with the non-standard MySQL aggregate function `GROUP_CONCAT` that concatenates non-`NULL` values from a group.

To rank languages we use the `languageTop10` query from Sect. 6.1.2 but without limiting the result to ten languages (that is, we leave out the `take 10`) and we use the SQL query:

```
SELECT Language,
       SUM( (CountryLanguage.Percentage * Country.Population) / 100)
       AS Rank
FROM   CountryLanguage, Country
WHERE  IsOfficial = 'T' AND Code = CountryCode
GROUP BY Language
ORDER BY Rank DESC
```

For MySQL we have used the InnoDB engine and the world database unchanged as downloaded. For reference, the `City` table has 4079 rows, the `Country` table has 239 rows, and `CountryLanguage` has 984 rows.

The numbers in Table 1 are not surprising. As expected we see that MySQL is much faster for the highly selective query of finding capitals as this is of the kind of queries that SQL is designed for, where indexes can be used effectively, whereas our library needs to make a full scan of both tables (but only one scan per table). For the aggregate queries there is no significant difference, possibly because the data set is not big enough to really expose any differences.

7.2 Non-normalized data

As our second benchmark we use the queries over `FilesystemElem` from Sect. 6.2. For comparison with MySQL we formulate two versions for finding duplicate files (corresponding to the `findDuplicates` function). In both cases we use a flattened data set where paths are in reverse order:

– First, a naïve schema where we have a table with just the full path in one column:

```
CREATE TABLE FsTest (path VARCHAR(250) NOT NULL)
```

To find (the number of) duplicate files we write a query that is a direct SQL translation of the query we used in the `findDuplicate` function, using the standard SQL functions `SUBSTRING` and `POSITION`:

Table 2 Running times (in milliseconds) for various queries over `FilesystemElems` values

	Files/Dirs	`substring`	`index`	`findDups`	`findDupsPP`	`findDirs`
ghc	2816/485	2026	31	81	22	161
testsuite	5417/315	7644	50	216	59	375
g++	6630/2178	11358	185	974	288	786
gcc-java	24424/1558	169568	102	778	276	3236
full	39287/4537	–	349	2243	697	5335

```
SELECT COUNT(*)
FROM   FsTest AS f1, FsTest AS f2
WHERE  SUBSTRING(f1.path FROM 1 FOR POSITION('/' IN f1.path)) =
         SUBSTRING(f2.path FROM 1 FOR POSITION('/' IN f2.path))
       AND f1.path <> f2.path
```

This setup is likely to make SQL experts cringe as it constitutes highly non-optimal use of SQL: no index is created, and even if we created an index on the sole column the complex string manipulation would inhibit use of the index.

– Second, we use a table schema that does a tiny bit of extra preprocessing, thus enabling more effective use of SQL. We use a table with two columns: the name of the file and the full path:

```
CREATE TABLE FsTest2 (name VARCHAR(250) NOT NULL,
                      path VARCHAR(250) NOT NULL)
```

To find (the number of) duplicate files we first construct an index on the `name` column and then use a straightforward query that compares file names and paths:

```
CREATE INDEX name_index ON FsTest2(name);

SELECT COUNT(*)
FROM   FsTest2 AS f1, FsTest2 AS f2
WHERE  f1.name = f2.name
       AND f1.path <> f2.path;
```

We create the index dynamically to make the query more comparable with our library.

For both SQL setups we use in-memory tables (that is, we use the MySQL engine `MEMORY`).

Table 2 shows the running times for some of the queries from Sect. 6.2. The data used for the experiment are file system elements created from various open source projects:

– *ghc* is the source distribution for GHC version 6.10.4.
– *testsuite* is the source distribution of the test-suite for GHC version 6.10.4.
– *g++* is the source distribution for the C++ part of GCC version 4.6.1.
– *gcc-java* is the source distribution for the Java part of GCC version 4.6.1.
– *full* is a directory containing the above four elements.

The Files/Dirs column gives a count of the number of files and directories found in the various data sets; the `substring` column is the timing for the naïve SQL setup, the `index` column is for the index-based SQL setup; the `findDup` column is for the `findDuplicates` function; the `findDupsPP` column is for the `findDuplicates'`

function from Sect. 6.2.3 with the extra post-processing step of unfolding the result to be equivalent to the result returned by `findDuplicates`; and the `findDirs` column is for the `findDuplicateDirs` function.

The results are mostly as expected: The naïve SQL query runs in quadratic time, and the rest runs roughly in linear time.

8 Discussion

We briefly discuss related work on data structures, language integration, comprehensions, and optimization of relational queries below, as well as variations and future work on semantics, expressiveness and further asymptotic optimizations.

8.1 Lazy data structures

Henglein [24] has shown how to represent collections—there interpreted as sets—using symbolic ("lazy") unions and products, as presented in Sect. 4. The idea of thinking of collections as constructed from a symmetric union operation instead of element-by-element cons operation (lists) is at the heart of the Boom hierarchy and developed in the Bird-Meertens formalism [2] for program calculation. Early on Skillikorn observed its affinity with data-parallel implementations [38], which has received renewed interest in connection with MapReduce style parallelization [12, 39]. Representing products lazily, however, does not seem to have received similar attention despite its eminent simplicity and usefulness in supporting naïve programming with products without necessarily incurring a quadratic run-time blow-up.

8.2 Language-integrated querying

Exploiting the natural affinity of the relational model with functional programming has led to numerous proposals for integrating persistent database programming into a functional language in a type-safe fashion, initiated by Buneman in the early 80s; see e.g. Buneman et al. [5, 6], Atkinson and Buneman [1], Ohori and Buneman [32], Ohori [31], Breazu-Tannen et al. [4], Tannen et al. [41], Buneman et al. [7], Wong [48]. More recently, a number of type-safe embedded query languages have been devised that provide for interfacing with SQL databases: HaskellDB [28], Links [11, 29] and Ferry [20]. The latter two allow formulating queries in a rich query language, which is mapped to SQL queries. The former is the basis of Microsoft's popular LINQ technology [30], which provides a typed language for formulating queries, admitting multiple query implementations and bindings to data sources. Execution of a query consists of building up an abstract syntax representation of the query and shipping it off to the data source manager for interpretation. In particular, the data can be internal (in-memory) collections processed internally, as we do here. However, we provide an actual query evaluation with efficient join implementation, not just an abstract syntax tree generator or a naïve (nested loop) interpreter.

8.3 Comprehensions

Wadler [44] introduces monad comprehensions for expressing queries in "calculus"-style using list comprehensions. Trinder and Wadler [42, 43] demonstrate how relational calculus queries can be formulated in this fashion and how common algebraic optimizations on queries have counterparts on list comprehensions. Peyton Jones and Wadler [35]

extend list comprehensions with language extensions to cover SQL-style constructs, notably ORDER BY and GROUP BY. These and other comprehension-based formulations [47] are based on nested iteration with the attendant quadratic run-time cost, however. Instead we forfeit the possibility of formulating arbitrary dependent products of the form `[ ... | x <- S, y <- f x, ... ]`, but provide an efficient implementation of independent products, which corresponds to `[ ... | x <- S, y <- T, ... ]` where `T` does not depend on `x`. Many queries can be manually transformed into an independent product form. Even though developed for a different purpose, the analysis technique developed by Grust et al. [21] for nested lists may be helpful in developing an automatic technique for doing so. A general investigation of efficient dependent product representation and computation is future work, however.

8.4 Query evaluation and optimization

SQL query evaluation and optimization are well-studied subjects in the database theory and systems. A thorough comparison with the literature [9, 13, 18] is beyond the scope of this paper. The idea of representing relations (tables) using lazy unions and products during query evaluation does not appear to have been investigated systematically, as it clashes with the classical System R-based database system architecture where query evaluation is performed in two separate stages: In the first stage queries are transformed based on algebraic properties, the availability of indexes and statistical information about the data and query history, resulting in a *query plan*. In the second stage the query plan is executed, where all tables, also intermediate ones, are represented as record streams.

In contrast to this, GMP essentially intermixes the symbolic computation steps of the first stage with the second stage "standard" evaluation. This intermixing simplifies achieving the effects of the first stage. For example, GMP achieves most of the effects[5] of classical algebraic optimizations based on join-introduction (from select and cross-product or other joins) as well as commutation of selection and projection with union and with cross-product [36, Sect. 15.3] at run time. Conversely, in the form presented in this paper, we believe GMP does not do better on non-nested queries.

A practical benefit of GMP's dynamic symbolic query evaluation is that it can accommodate user-defined functions and support functional abstractions without forcing them to be represented symbolically for performance reasons. For example, we can define a library function such as

```
combineWith t = depositors `X` t
```

(or something more complicated and impressive-looking) and then easily use it multiple times

```
...
select ((depId, loanId) `Is` eqInt32) (combineWith loans)
...
select ((depId, acctId) `Is` eqInt32) (combineWith accounts)
...
select (((depName, loanName) `Is` lookTheSame)
        `SAnd` (TT `PAnd` (Pred ((> 50000) . loanBalance))))
        (combineWith (perform (Id `Par` Func roundToNearestThousand) loans))
...
```

[5] It does not push projections through selection or the other way round.

and getting linear performance in each case. In a System R-style query optimization setting, the function `combineWith` would have to be represented symbolically to facilitate inlining where it is used so as to produce separate SQL queries, which each can then be optimized and evaluated.

Intermixed symbolic and standard evaluation offers the potential of taking run-time characteristics of computed data into account; e.g., depending on the *actual* cardinality of an *intermediate* result, we may choose to invoke one or the other join algorithm. In System R-style query optimization intermediate data characteristics are only *estimated*.

Performing symbolic computation at run time comes at the cost of having to process the two constructors `X` and `U` in addition to the cons and nil constructors when only streams need to be processed. There are highly tuned implementations for stream processing. How much of any extra cost the two (or, eventually, more) additional constructors cost after a similar tuning effort is an open question. The type system of GMP recognizes when code operates on multisets represented as stream (constructed by `MSet`), which can then be processed by stream-specific code. If large inputs and outputs of queries are stored as tables (essentially streams), *almost all* processing should then be expected to be processed in stream processing mode.

Additional optimizations appear to be possible by pushing the method of introducing symbolic representations in connection with dynamic symbolic computation even further. For example, the clause for `Fst`

```
perform Fst (s `X` t)       = count t `times` s
```

results in producing representations of such *scalar multiplications* in essentially *unary* form: `s `U` ... `U` s`. This suggests introducing a constructor for scalar multiplication to avoid this asymptotic time and space blow-up. Likewise, `reduce` simply multiplies out its multiset argument. In some common cases, however, substantially better performance can be accomplished by representing the result of performing (mapping) a function on a Cartesian product symbolically. This is the subject of ongoing and future work.

A thorough complexity-theoretic analysis and comparison with theoretical results by Willard [45], Goyal and Paige [17], Willard [46] for query sublanguages guaranteeing worst-case linear or quasi-linear running time (data complexity) remains to be done. Off-hand it appears that their techniques are complementary to ours and thus potentially combinable with the symbolic unions and products used here.

8.5 *k*-ary joins

A number of variations and improvements are possible while staying within our implementation framework. For example, decomposing a join on k tables into binary joins is a major part of classical query optimization. An often applicable alternative is to generalize the binary join conditions `(f, g) `Is` e` to k-ary conditions; e.g. `(f, g, h) `Is` e` representing a join condition on triples instead of pairs. This yields linear-time joining on more than 2 tables in the limited, but common case of joining on some field in each of the tables involved in the join, typically a key or foreign key.

In column-oriented databases [40] a table is stored as its decomposition into key- or position-ordered columns that are joined on their keys, respectively positions, to construct the whole table or particular projections of it. They can be viewed as related to the idea of representing tables by symbolic products: Loosely speaking, they correspond to a symbolic product representation with an associated k-ary join condition.

8.6 Querying on infinite lists and with functions as elements

The inputs to our query functions are elements of `MSet a`, interpreted as finite multisets of (possibly recursively typed) first-order values. Fixing `djoin` in Sect. 2.4 as the join algorithm yields a deterministic semantics for `MSet a`, even when interpreted as finite *lists*. Unfortunately, joins in general (and discrimination-based joins in particular) do not easily extend to *infinite* lists. For this reason, querying on *streams* (possibly infinite lists) restricts joining to finite windows over input streams [27]. Since stream processing is highly relevant in practice, it would be interesting to see if and how the techniques in this paper can be extended to streams.

Another input domain extension consists of admitting higher-order values—functions—as input elements. The only equivalence we have for functions is the trivial one, `TrivE`: All functions of a given type are `TrivE`-equivalent. This does allow useful applications, however. If functions (more precisely, function closures) are tagged with dynamically generated names when they are constructed (think of them as elements of type `ref (a -> b)` in ML), we can perform a join on tag equality as a form of "intensional" equality on functions: Only functions with the same tag are recognized as equal. Less intensional equalities are conceivable, but a fully extensional equality is undecidable.

8.7 Semantic nondeterminism

We have on several occasions observed that operations on multisets are not semantically deterministic; that is, applying a function to different representations of the same multiset may yield observably different results (say, different numbers). This is a problem that permeates all of database application development: Relational database systems insist on manipulating "unordered" collections, but then applications iterate over them in a certain, implementation-specific order. The query engine may decide to return a table in one order today and in a different order tomorrow. This puts the onus of ensuring semantic determinacy on the application software developer. Anecdotal evidence suggests that it is not uncommon for programmers to incorrectly rely on a particular order.

But why work with unordered collections at all? Why not simply use lists? The problem is that the product then needs to define a *particular* order for its pairs that each implementation must abide by, but the order is different depending on which join algorithm is used: nested iteration, sort-merge join, hash join or discrimination-based join. It is possible to provide a faithful deterministic list semantics by representing lists as multisets of elements instrumented with their list positions [19]. This requires a postprocessing traversal before list conversion, however, which is often shunned for performance reasons. An alternative, at least for ordered elements, is to maintain all lists in sorted order, which obviates storing the list positions. We leave it to future work as to whether this can be made to be both semantically clean and computationally efficient.

Acknowledgements We would like to thank Phil Wadler, who provided the initial incentive and encouragement for this work. We would like to thank Mike Sperber, Janis Voigtländer, Andreas Rossberg, and Derek Dreyer and the anonymous referees for valuable comments, corrections, suggestions for improvements, and pointers to the literature that we have sought to incorporate as best we could. The alphabetically first author would like to thank Torsten Grust and his team for inspiring discussions and inputs during a visit at the University of Tübingen in November 2009. They provided not only important literature references, but offered insights on the theory and practice of database systems and database programming that cannot be learned from publications alone.

References

1. Atkinson, M.P., Buneman, O.P.: Types and persistence in database programming languages. ACM Comput. Surv. **19**(2), 105–170 (1987). http://doi.acm.org/10.1145/62070.45066
2. Backhouse, R.: An exploration of the Bird-Meertens formalism. In: STOP Summer School on Constructive Algorithmics (1989)
3. Bentley, J.: Programming pearls: Little languages. Commun. ACM **29**(8), 711–721 (1986). http://doi.acm.org/10.1145/6424.315691
4. Breazu-Tannen, V., Buneman, P., Naqvi, S.: Structural recursion as a query language. In: DBPL3: Proceedings of the Third International Workshop on Database Programming Languages: Bulk Types & Persistent Data, pp. 9–19. Morgan Kaufmann, San Francisco (1992)
5. Buneman, P., Nikhil, R., Frankel, R.: A practical functional programming system for databases. In: FPCA'81: Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture, pp. 195–202. ACM, New York (1981). http://doi.acm.org/10.1145/800223.806779
6. Buneman, P., Frankel, R.E., Nikhil, R.: An implementation technique for database query languages. ACM Trans. Database Syst. **7**(2), 164–186 (1982). http://doi.acm.org/10.1145/319702.319711
7. Buneman, P., Naqvi, S., Tannen, V., Wong, L.: Principles of programming with complex objects and collection types. In: ICDT'92: Selected Papers of the Fourth International Conference on Database Theory. Elsevier, Amsterdam, pp. 3–48 (1995). doi:10.1016/0304-3975(95)00024-Q
8. Chamberlin, D.: SQL. In: Encyclopedia of Database Systems, 1st edn. Springer, Berlin (2009)
9. Chaudhuri, S.: An overview of query optimization in relational systems. In: PODS, pp. 34–43. ACM Press, New York (1998)
10. Cheney, J., Hinze, R.: First-class phantom types. CUCIS TR2003-1901. Cornell University (2003)
11. Cooper, E., Lindley, S., Wadler, P., Yallop, J.: Links: Web programming without tiers. In: Proceedings of the 5th International Conference on Formal Methods for Components and Objects, pp. 266–296. Springer, Berlin (2006)
12. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: OSDI, pp. 137–150 (2004)
13. Deshpande, A., Ives, Z., Raman, V: Adaptive query processing. Found. Trends Databases **1**(1), 1–140 (2007). doi:10.1561/1900000001
14. Eini, O.: The pain of implementing LINQ providers. Commun. ACM **54**(8), 55–61 (2011). http://doi.acm.org/10.1145/1978542.1978556
15. Finland, S.: World database from MySQL. Available as example database from MySQL.com (2010). http://downloads.mysql.com/docs/world.sql.zip, the sample data used in the world database is Copyright Statistics Finland, http://www.stat.fi/worldinfigures
16. GHC Team: The Glasgow Haskell Compiler (2011). http://www.haskell.org/ghc/
17. Goyal, D., Paige, R.: The formal reconstruction and speedup of the linear time fragment of Willard's relational calculus subset. In: Proceedings of the IFIP TC 2 WG 2.1 International Workshop on Algorithmic Languages and Calculi, pp. 382–414. Chapman & Hall, London (1997)
18. Graefe, G.: Query evaluation techniques for large databases. ACM Comput. Surv. **25**(2), 73–169 (1993). http://doi.acm.org/10.1145/152610.152611
19. Grust, T., Sakr, S., Teubner, J.: XQuery on SQL hosts. In: Proceedings of the Thirtieth International Conference on Very Large Data Bases, vol. 30, p. 263. VLDB Endowment (2004)
20. Grust, T., Mayr, M., Rittinger, J., Schreiber, T.: Ferry: Database-supported program execution. In: Çetintemel, U., Zdonik, S.B., Kossmann, D., Tatbul, N. (eds.) SIGMOD Conference, pp. 1063–1066. ACM, New York (2009)
21. Grust, T., Rittinger, J., Schreiber, T.: Avalanche-safe LINQ compilation. Proc. VLDB Endow. **3**, 162–172 (2010). http://dl.acm.org/citation.cfm?id=1920841.1920866
22. Henglein, F.: Generic discrimination: Sorting and partitioning unshared data in linear time. In: Hook, J., Thiemann, P. (eds.) ICFP'08: Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming, pp. 91–102. ACM, New York (2008). http://doi.acm.org/10.1145/1411204.1411220
23. Henglein, F.: Generic top-down discrimination for sorting and partitioning in linear time. Tech. rep., Department of Computer Science, University of Copenhagen (DIKU), submitted to Journal of Functional Programming (JFP) (2010)
24. Henglein, F.: Optimizing relational algebra operations using discrimination-based joins and lazy products. In: Proc ACM SIGPLAN 2010 Workshop on Partial Evaluation and Program Manipulation, pp. 73–82. ACM, New York (2010). http://doi.acm.org/10.1145/1706356.1706372, also DIKU TOPPS D-report no. 611
25. Henglein, F., Larsen, K.F.: Generic multiset programming for language-integrated querying. In: Proc. 6th Workshop on Generic Programming (WGP) (2010)

26. Hudak, P., Fasel, J.H.: A gentle introduction to Haskell Version 98. http://www.haskell.org/tutorial, online tutorial (1999)
27. Jain, N., Mishra, S., Srinivasan, A., Gehrke, J., Widom, J., Balakrishnan, H., Çetintemel, U., Cherniack, M., Tibbetts, R., Zdonik, S.: Towards a streaming sql standard. Proc. VLDB Endow. **1**(2), 1379–1390 (2008)
28. Leijen, D., Meijer, E.: Domain-specific embedded compilers. In: Proceedings of the 2nd Conference on Domain-Specific Languages, USENIX Association Berkeley, CA, USA, Austin, Texas, pp. 109–122 (1999)
29. Lindley, S., Wadler, P., Yallop, J., Cooper, E.: Links: Linking theory to practice for the web (2009). http://groups.inf.ed.ac.uk/links/
30. Meijer, E., Beckman, B., Bierman, G.: LINQ: Reconciling objects, relations, and XML in the .NET Framework. In: Proc. SIGMOD (2006)
31. Ohori, A.: Orderings and types in databases. In: Bancilhon, F., Buneman, P. (eds.) Advances in Database Programming Languages, Frontier Series, pp. 97–116. Addison-Wesley, ACM Press, Reading, New York (1990), Chap. 6
32. Ohori, A., Buneman, P.: Type inference in a database programming language. In: Proc. 1988 ACM Conf. on LISP and Functional Programming, pp. 174–183. ACM Press, New York (1988)
33. O'Sullivan, B.: Criterion package (2010). Available from http://hackage.haskell.org/package/criterion
34. Peyton Jones, S.: The Haskell 98 language. J. Funct. Program. **13**(1), 1–146 (2003)
35. Peyton Jones, S., Wadler, P.: Comprehensive comprehensions. In: Proc. 2007 Haskell Workshop, Freiburg, Germany (2007)
36. Ramakrishnan, R., Gehrke, J.: Database Management Systems, 3rd edn. McGraw-Hill, New York (2003)
37. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access path selection in a relational database management system. In: Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data. SIGMOD'79, pp. 23–34. ACM, New York (1979). http://doi.acm.org/10.1145/582095.582099
38. Skillicorn, D.: Architecture-independent parallel computation, presented at IFIP WG2.1 conference, Burton Manor, England (1990)
39. Steele, G.: Organizing functional code for parallel execution; or, foldl and foldr considered slightly harmful. In: Proc. ICFP 2009 (2009), invited talk
40. Stonebraker, M., Abadi, D.J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., O'Neil, P., Rasin, A., Tran, N., Zdonik, S.: C-Store: A column-oriented DBMS. In: Proc. 31st VLDB Conf. Trondheim, Norway (2005)
41. Tannen, V., Buneman, P., Wong, L.: Naturally embedded query languages. In: ICDT'92: Proceedings of the 4th International Conference on Database Theory, pp. 140–154. Springer, London (1992)
42. Trinder, P., Wadler, P.: List comprehensions and the relational calculus. In: Proceedings of the 1988 Glasgow Workshop on Functional Programming, Rothesay, Scotland, pp. 115–123 (1988)
43. Trinder, P., Wadler, P.: Improving list comprehension database queries. In: TENCON'89. Fourth IEEE Region 10 International Conference, Bombay, India, pp. 186–192. IEEE, New York (1989). doi:10.1109/TENCON.1989.176921
44. Wadler, P.: Comprehending monads. Math. Struct. Comput. Sci. **2**, 461–493 (1992)
45. Willard, D.E.: Applications of range query theory to relational data base join and selection operations. J. Comput. Syst. Sci. **52**(1), 157–169 (1996). doi:10.1006/jcss.1996.0012
46. Willard, D.E.: An algorithm for handling many relational calculus queries efficiently. J. Comput. Syst. Sci. **65**(2), 295–331 (2002). doi:10.1006/jcss.2002.1848
47. Wong, L.: Querying nested collections. Ph.D. thesis, University of Pennsylvania (1994)
48. Wong, L.: Kleisli, a functional query system. J. Funct. Program. **10**(1):19–56 (2000)
49. Xi, H., Chen, C., Chen, G.: Guarded recursive datatype constructors. In: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 224–235. ACM, New York (2003)