ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

David N. Turner (Ed)

Functional Programming, Glasgow 1995

Proceedings of the 1995 Glasgow Workshop on Functional
Programming,
Ullapool, Scotland, 10-12 July 1995

Paper:

Database Manipulation in Haskell 1.3

Kevin Hammond and Phil Trinder

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

Database Manipulation in Haskell 1.3

Kevin Hammond*

Division of Computer Science, University of St. Andrews

St. Andrews, Scotland, U.K.

E-mail: `kh@dcs.st-and.ac.uk`

Phil Trinder†

Department of Computing Science, University of Glasgow

Glasgow, Scotland, U.K.

E-mail: `trinder@dcs.gla.ac.uk`

Abstract

This paper investigates using a functional language with a sophisticated type system to implement data-intensive programs. We focus on how bulk data types can be represented rather than on the efficient implementation of those types.

The paper is intended more as a discussion document, recording some thoughts on how to represent bulk data in a functional language, than as a conclusive statement on functional databases. It also serves as a reasonably complex and hopefully realistic example of how some of the new features of Haskell 1.3, including constructor classes, can be exploited to good effect. The limitations of these features are also assessed.

1 Introduction

As part of the EPSRC Parade project, we are studying the implementation of data intensive programs in a purely functional language. Our goal is to make it easy to implement data-intensive programs, rather than to construct an complete DBMS. Representing and manipulating bulk data values are the crucial aspects of data-intensive programming addressed in this paper.

Most functional data languages are primarily query languages, e.g. [AFH+87, HN91, Pou88, PC91]. Updating bulk values is typically restricted to the ‘top-level’, and sometimes not addressed at all in these languages. In contrast, we use Haskell as a data manipulation language, i.e. to insert, delete and update component of bulk values.

In previous work, we described a transaction-processor written in parallel Haskell and running on the GRIP machine [AHPT91, AHPT93]. This work goes beyond that by considering secondary indices, and by generalising our implementation to multiple bulk data types.

A major objective is to explore the new capabilities that have provided in the latest version of Haskell [Has96]. The most important of these features for our application are constructor classes and labelled fields (or records).

2 Data Description using Type Classes

Bulk data structures are used to store the collections of entities that are held in a database. A simple example of a bulk type is a list, but more complex structures such as B-trees are commonly used in database implementation because they provide faster random access to entities through the use of indices.

Entities have keys of some ordered type, which are used to index those entities. When constructing indexed types, it is also useful to be able to determine the range of values that a key may take. In Haskell keys are thus members of

*Supported by an SOED Research Fellowship from the Royal Society of Edinburgh and the EPSRC Parade project

†Supported by the EPSRC Parade project

the `Ix` and `Ord` classes. Normally keys are taken directly from a single data component in an entity, but especially for secondary indices, it is sometimes useful to compute them dynamically from one or more stored values.

```
class (Ix a, Ord a) => Key a

class Entity e where
  key      :: Key k => e k -> k
```

The key for an entity can be obtained by applying the `key` method to that entity. A *constructor class* [Jon95] is used here to allow a given entity type to be parameterised on different concrete types. The variable `e` stands for any type class that takes a single type as its parameter (it has kind `*->*`).

2.1 Operations on Bulk Types

All bulk types support the following familiar operations:

- *lookup* the value of an entity;
- *insert* a new entity into the bulk datatype;
- *delete* an entity from the datatype;
- *update* an existing entity.

where *update* can be seen as a combination of deletion and insertion, if desired.

During database initialisation, it is also necessary to construct new bulk data structures. It is generally easier and more efficient to create a bulk type from the list of entities that should initially form part of that type rather than to create an empty type and then insert entities singly. In fact, for some bulk types (such as lists), the type system makes it rather hard to create a completely empty structure!

Using constructor classes [Jon95], we can describe a simple generic bulk type class `BulkType` which permits these operations. This type class is defined over a datatype `b`, parameterised by key and entity types, `e` and `k`. Entities are members of the `Entity` class, and are thus parameterised on their key values. While the operations described here don't consider the possibility that the operation might fail, it is trivial to extend them to cover this case.

```
class BulkType b where
  initialise ::
    (Key k, Entity e) => (k,k) -> [e k] -> b k e
  lookup    :: (Key k, Entity e) => k      -> b k e -> [e k]
  insert    :: (Key k, Entity e) => e k    -> b k e -> b k e
  delete    :: (Key k, Entity e) => k      -> b k e -> b k e
  update    :: (Key k, Entity e) => e k    -> b k e -> b k e
```

Note that the `lookup` function returns a list of entities that match the key value. We need to return a list since there may be no entities that match a particular key value, or for secondary keys, multiple values may match the given key.

While not a primary requirement of a bulk type, it is probably useful to provide an information function to obtain the legal range of key values.

```
theBounds  :: (Key k, Entity e) => b k e -> (k,k)
```

Other useful operations work on the data structure as a whole. One attractive operation is `toList` which builds a list from the entire bulk type.

```
toList     :: (Key k, Entity e) => b k e -> [e k]
```

This operation allows queries to be constructed using list comprehensions, an elegant solution to the problem of constructing relational queries in a functional language [Tri91]. For example,

```
hughes_followset =
  [x | x <- toList academic_staff,
       Whisky <- prefers x,
       FunctionalProgramming <- teaches x]
```

Unfortunately, this is an extremely inefficient way to construct selections based solely on key values, since large sections of the search space can be pruned if the characteristics of the selection operator are known in advance.

For some bulk data structures the easiest way to build an efficient higher-order selection function is to parameterise on the range of key values that should be selected. This has the advantage of supporting `toList` and `lookup` as special cases. Unfortunately `select` may not be efficient for some bulk data structures, e.g. hashed files or secondary indices.

```
select    :: (Key k, Entity e) => (k,k) -> b k e -> [e k]
toList b  = select (theBounds b) b
lookup k  = select (k,k)
```

2.2 Example: B-Trees

One common bulk data type is the B-tree, whose nodes contain multiple keys and sub-trees. Within a node, the keys are ordered, so that all entities in a sub-tree have key values which are less than or equal to the corresponding key value in the node. There is always one more sub-tree than key value, which contains entities which are greater than the last key value in the node.

It is possible to define a general B-tree structure using lists, but since each node in a B-tree has the same arity, this is over-general, and in order to speed access, tuples or arrays of the appropriate size would normally be used rather than key and sub-tree lists.

```
data (Key k, Entity e) => BTree k e = BNode [k] [BTree k e]
                                   | BEmpty | BLeaf (e k)
```

For maximum efficiency in lookup, B-trees must be balanced following operations which change the structure of the tree. In our context, these operations are insertions and deletions, but not updates or lookups.

By way of example we show how *lookup* can be defined for a B-tree.

```
instance BulkType BTree where
  lookup k (BLeaf e)      | k == key e    = [e]
  lookup k (BNode ks bs) = blookup k ks bs
  lookup _ _              = []

blookup k [] [b] = lookup k b
blookup k (k':ks) (b:bs) | k <= k' = lookup k b
                        | otherwise = blookup k ks bs
```

2.3 The Haskell 1.3 Collection Library

At first sight, the collection library, `LibCollection`, that was proposed for Haskell 1.3 would seem to be ideal for bulk types. Unfortunately, while it provides many operations that are not needed for our functional database, it only supports lookup on complete entities or using inefficient filter-style operations, and doesn't provide update directly (this is important to avoid index rebalancing). We have therefore implemented our operations directly on the base types rather than in terms of this library.

3 The Relational Data Model

While our underlying implementation data model is the standard functional model of algebraic data types and graphs, this is not necessarily the perspective that we would want to present to the database user. In this paper, we show how to implement a standard relational data model using our functional primitives, but other models, e.g. the functional model are also possible.

3.1 Entities

Entities can be best modelled in Haskell 1.3 by data types with labelled fields (which for simplicity, we will call records). As an example, we define the `Academic_Staff` entity with fields `staff_no`, `salary` etc.

```
data Key k => Academic_Staff k =
  Academic_Staff_Rec {
    staff_no :: Integer
    salary   :: Float
    prefers  :: [Beverage_Reln]
    teaches  :: [Course_Reln]
  }
```

The *prefers* and *teaches* fields refer to one-to-many or many-to-many relations between the `Academic_Staff` relation and the *Beverage_Reln* and *Course_Reln* relations.

For each field, there is a selector function of the same name that can be applied to values of type `Academic_Staff`.

```
tax :: Key k => Academic_Staff k -> Float
tax lecturer = salary lecturer * 0.25
```

To specify the key field for an entity, we need to make it an instance of the `Entity` class.

```
instance Entity Academic_Staff where
  key = staff_no
```

3.2 Relations

Our basic approach is to use bulk types as described above to represent relations.

```
data A_S_Reln k e = ASRel (BTree k e)
```

It is possible to define instances for relations so that lookup etc. can be written to work directly on a relation without needing to know about the representation of the internal indices.

```
instance BulkType (A_S_Reln Integer Academic_Staff) where
  insert k (ASRel b) = insert k b
  ...
```

3.2.1 Linking Relations

In Trinder's thesis [Tri89] he proposed to structure relations into cyclic graphs using the standard functional programming techniques. This creates some problems when updating entities, since all links to an entity must also be located and updated.

One solution is to use a fully inverted structure so that each entity records which other entities refer to it (in effect, links between relations become bidirectional). This uses additional storage for the extra pointers, however, and also adds to the database management code – if links are added from new relations, all entities in a relation must be restructured and reconstructed.

An alternative solution is to use keys rather than graph to link relations. This avoids wasting storage and simplifies updates, at the expense of

1. requiring additional *lookups* to follow links, rather than just traversing the data graph, and
2. introducing the risk of inconsistency when entities are deleted from a relation.

We do not address this issue here.

4 Bulk Data Type Representation

We require both fast access to database entities, and fast in-memory update (disk update is another issue). Data may be sparse or dense in a key value. For dense data with key ranges that rarely change, serial arrays are an excellent data representation.

4.1 Version Arrays

One interesting representation for bulk data types on dense structures is to use version arrays. This structure is a functional array representation, intended for in-place update. To simplify the presentation, we show here how to model these structures using Haskell arrays, but we would normally expect these arrays to be implemented via direct state manipulation.

A version array has two components, a unique incrementable version number, and an array of entities that are held in the array. For each key, there is an ordered list of (version,entity) pairs, where the versions indicate when the corresponding entity was created/updated, most recent first. Entities are stored using the `Maybe` type – `Nothing` if the entity was not present in the initial array, or was deleted in a later version; otherwise `Just e`.

```
data (Key k, Entity e) => DBVA k e =
    DBVA Version (Array k [(Version,Maybe (e k))])
type Version = Int
```

The initial version of a version array (the one identified by version *minvers*) contains `Nothing` except the initial values that were given.

```
initialise rng vals =
    DBVA minvers (array rng empty // inited) where
        empty = [(k,[(minvers,Nothing)]) | k <- range rng]
        inited = [( key val, [(minvers,Just val)] )
                    | val <- vals])
```

When inserting a new entity in the database, a new version of the array is created with an incremented version number. The entity is inserted at the appropriate key location, with the new version number. Since it is only possible to insert an entity if it was not already present in the data structure, this needs to be tested, and the previous version of the database returned if there was already an entity with the same key.

```
insert val db@(DBVA v va) =
    case current of
        ((_,Just v):_) -> db
        _ -> DBVA (v+1) (va // [( k, (v+1,Just val) : current) ])
    where k = key val
          current = va ! k
```

The other standard operations are similarly straightforward to define.

4.1.1 Characteristics for Parallel Access

We believe that version arrays work well for both lookup and update in parallel, using techniques similar to those we have already described for linked lists [AHPT93].

The array is implemented as an abstract data type which provides only *lookup*, *insert* etc. operations. In particular, the version number cannot be observed externally.

Internally, the header containing the version number is unique to a particular version of the array, but the array of data is shared by all versions of the array.

When a new version of the array is created, the array of data is modified in the appropriate position to create a new version, value pair at the head of the list of versions for that index. The new value is created as a suspension rather than being evaluated immediately. A new version array header can then be created for this version of the version array.

For example, when inserting v into a version array a whose version is n at index i , the pair $(n+1, v)$ will be prepended to the existing list of version, value pairs at the i th element of a , and a new header (for version $n+1$) returned to the caller. The new value v is created as a suspension.

The advantage of this approach is that the only synchronisations required are on the creation of the new version array header (a fast operation) and when the newly inserted value is demanded at a later point in execution. Later lookup operations can proceed before a newly inserted value has been evaluated, and it is not necessary to complete earlier lookups before inserting a new value. At the same time, the structure supports both fast lookup (close to $O(1)$) and fast insertion ($O(1)$) if the basic array structure is shared.

4.1.2 Hashing

Version arrays could be hashed rather than accessed directly. This is a fairly straightforward change, which we will not discuss here. The use of hashed arrays for functional databases is described by Heytens and Nikhil [HN91].

4.2 Secondary Indices

Secondary indices are used to provide alternative access paths to a relation, so as to improve the speed of common queries. A secondary index may be non-unique. For example, by providing a secondary index for `Academic_Staff` on the `courses` field, we can rapidly answer queries such as who teaches a particular course. The result of this query will be a, possibly empty, list of staff who teach the course.

4.2.1 Relations Supporting Secondary Indices

Because secondary indices share the primary data representation, the general structure of a relation is as a bulk data type with a single primary index, and multiple secondary indices. If a value is inserted or deleted using the primary index,

all secondary indices must also be updated. In general, we want to use different bulk types for primary and secondary indices (and perhaps even for different secondary indices).

Each secondary index in the relation will be represented as a bulk type of primary keys. The primary key or keys corresponding to a secondary key can then be obtained simply by looking up the secondary key in the index. This two-stage lookup is less efficient than the direct-lookup provided by secondary indices in a conventional database.

```
data (Key s, Key p) => SecondaryIndex p s = s -> p
```

```
instance Key p => Entity (SecondaryIndex p) where
  key (s -> p) = s
```

So that, for example, we can define a secondary index which uses strings to refer to an entity whose primary keys are integers by type `SI = BTree String (SecondaryIndex String Integer)`, or more generally

```
type (BulkType b, Key s, Key p) =>
  SI b s p = b s (SecondaryIndex p s)
```

One fairly general way of describing a relation which has multiple secondary indices is thus

```
data (BulkType bp,
      BulkType bs1, ..., BulkType bsn,
      Entity e,
      Key p,
      Key s1, ..., Key sn) =>
  Relation bp bs1 ... bsn e p s1 .. sn =
    Rel {
      rel :: bp p e
      si1 :: SI bs1 s1 p
      ...
      sin :: SI bsn sn p
    }
```

Such a relation could obviously itself be a bulk type, and hence relations could be used for either primary or secondary indices.

4.2.2 Lookup on Secondary Indices

To lookup a value through a secondary index we first need to obtain the list of primary keys corresponding to the secondary key using the `lookup` operation. Each of these primary keys can then be looked up in turn using the `lookup` operation on primary keys; the concatenation of these primary lookups gives the list of entities referenced by the secondary key.

The generic form for this kind of lookup is thus

```
secondary_key_lookup sk si r =
  concat (map (\ k -> lookup k r) pks)
  where pks = lookup sk (si r)
```

Since we would normally expect both primary and secondary indices to be held in primary store, the cost of the extra level of indirection required to access secondary indices in this way is probably not significant.

4.3 Views

It is common to provide more than one *view* of a database relation. A database view restricts the values of a relation that can be seen by a database user, which provides security against accidental change or unauthorised access.

For example, the *Academic_Staff* relation might include sensitive details of the staff members' salaries or beverage preferences, which should not be seen by students considering courses that are run by individual staff members. In a functional setting, a database view of this kind is equivalent to an abstract data type.

```
module Student_View (Academic_Staff,staff_no,teaches) where
import qualified Academic_Staff(Academic_Staff(..))

staff_no    :: Academic_Staff -> Integer
teaches     :: CourseReln

staff_no    = Academic_Staff.staff_no
teaches     = Academic_Staff.teaches
```

5 Further Work

5.1 Transactions

We are planning to work with a transaction processing model. The manager will handle generic transactions on the database as in our previous work. If each transaction returns a pair of a message (indicating the result of the transaction – either success or failure) and a new database, then the manager will have the following structure.

```
manager [] db = ([],db)
manager (t:ts) db =
  let (msg, db') = t db
      (msgs, db'') = manager ts db'
  in (msg:msgs, db'')
```

5.2 Disk Accesses

Our intention is to hold the indices of a relation in-memory, and read entities from disk as they are needed. Changed entities and indices will be written back to disk as a new generation [HG83] when a checkpoint is encountered. Intermediate versions of version arrays can then be discarded, using an operation equivalent to `prune`.

```
prune vp (DBVA v va) =
  DBVA v (amap (filter (\ (ve,e) -> ve >= vp)) va)
```

6 Conclusions

We have investigated using a modern functional language to write data intensive programs. Constructor classes and records prove particularly aid in describing bulk data types. We have defined a generic bulk data type class, which can represent a variety of different structures on different (key,entity) combinations.

Although not fully described here, we believe that the structure of relations which we describe here will help us solve the secondary index problem for parallel update.

References

- [AFH+87] Argo G, Fairbairn J, Hughes RJM, Launchbury EJ, and Trinder PW, “Implementing Functional Databases”, *Proc. Workshop on Database Programming Languages*, Roscoff, France (September 1987), pp. 87-103.
- [AHPT91] Akerholt G, Hammond K, Peyton Jones SL, and Trinder P, “A Parallel Functional Database On GRIP”, *Proc. Glasgow Workshop on Functional Programming*, Portree, Scotland (August 1991).

- [AHPT93] Akerholt G, Hammond K, Peyton Jones SL, and Trinder P, “Processing Transactions on GRIP: a Parallel Graph Reducer”, *Proc. PARLE '93*, Munich, Springer-Verlag LNCS 694, (1993), pp. 634–647.
- [Has96] The Haskell 1.3 Committee, “Report on the Programming Language Haskell, Version 1.3”, In Preparation, (April 1996).
- [HG83] Hecht MS, and Gabbe JD, “Shadowed Management of Free Disk Pages with a Linked List”. *ACM Transactions on Database Systems* 8:4, (December 1983), pp. 503–514.
- [HN91] Heytens, ML and Nikhil R.S. “List Comprehensions in AGNA, a Parallel Persistent Object System” *Proc. FPCA '91*, Cambridge, Mass. (1991).
- [Jon95] Jones, MP, “A system of constructor classes: overloading and implicit higher-order polymorphism”, *J. Functional Programming* 5:1, (1995) pp. 1–37.
- [Pou88] Poulouvasilis A.P. “FDL: An Integration of the Functional Data Model and the Functional Computational Model”, *Proc. 6th British National Conference on Databases (BNCOD 6)* (July 1988), 215-236.
- [PC91] Poulouvasilis A.P. Small C. “A Functional Programming Approach to Deductive Databases”, *Proc. 17th International Conference on Very Large Databases*, G. Lohman, et al. (eds), 491-500, 1991.
- [Tri89] Trinder PW, *A Functional Database*, Oxford University D.Phil. Thesis, (December 1989).
- [Tri91] Trinder, P.W. “Comprehensions: a Query Notation for DBPLs”, *Proc. Third International Workshop on Database Programming Languages*, Nafplion, Greece (August 1991).