ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

Paolo Atzeni and Val Tannen (Eds)

Database Programming Languages (DBPL-5)

Proceedings of the Fifth International Workshop on Database
Programming Languages, Gubbio, Umbria, Italy, 6-8 September 1995

Paper:

Database States in Lazy Functional Programming Languages: Imperative Update and Lazy Retrieval

Yoshihiko Ichikawa

Published in collaboration with the
British Computer Society

Database States in Lazy Functional Programming Languages: Imperative Update and Lazy Retrieval

Yoshihiko Ichikawa

Department of Information Sciences, Ochanomizu University

2-1-1 Otsuka Bunkyo-ku, Tokyo 112, JAPAN

ichikawa@is.ocha.ac.jp

Abstract

This paper proposes a database manipulation interface for the statically typed, purely functional programming language Haskell. The data model uses surrogates to permit direct update of stored objects, and the basic interface is designed based on the state-transformer approach, so that the interface is referentially transparent. This approach requires all the operations to be executed in a single state-transition sequence and thus tends to make queries more imperative than expected. The proposed approach lessens this burden on query construction, by using versioning. Versions can be “frozen” or locked, and a set of locked versions can be supplied as an argument to query operations. This intraprogram versioning permits on-the-fly dereference during query construction, and allows for straightforward implementation of lazy retrieval in strict state-transition sequences.

1 Introduction

Some distinctive features of purely functional programming languages are that their background is declarativeness and mathematical because its computation model includes no side-effects, that they are succinct because of their comprehension syntax, and that they can be efficiently processed by sharing of computation results (lazy computation). These features are also effective in database processing. The mathematical purity ensures that there is much potential in query optimizability and parallel processing, the comprehension syntax has been proven to be effective in query construction [15, 4], and the lazy computation is also a useful tool because databases tend to be very large.

Database updating, however, is not easy to incorporate in purely functional database processing; The lack of side-effecting expressions precludes the assignment operation. An effective and potentially efficient updating mechanism is thus one of the crucial needs in increasing the feasibility of using purely functional languages in database processing.

This paper therefore proposes a suitable database manipulation interface for statically typed, non-strict, purely functional programming languages. The primary target here is Haskell, a standardized non-strict, purely functional programming language [7]. Although Haskell is a standardized language, it has two novel features: a monad of state transformers to implement referentially transparent I/O operations, and the type class system to handle overloaded functions (*ad-hoc polymorphism*). Every I/O operation in the language is a state transformer which, given an I/O state, produces a pair consisting of the operation result and the new state [9].¹ State transformers are constructed using basic combinators and primitive I/O operators, and a program is an I/O state transformer that maps a given initial machine state to the final state by executing the included I/O operations one-by-one.

The proposed approach basically follows this line, and the database operations are referentially transparent state transformers. The state transformer approach, however, is not a panacea. In this basic approach, not only database update operations but also database queries must be coded in a single state-transition sequence. In other words, even simple queries must be coded in an “imperative” style of programming and cannot be coded in a succinct “functional” style. Our approach has a mechanism to relax this inherent imperativeness.

---

¹ Note that the I/O model of Haskell-1.2 is based on *dialogue* and *continuation* instead of state transformers. In this paper, however, we assume the Haskell-1.3 I/O model. At the time of writing, the up-to-date specification of the Haskell-1.3 I/O proposal can be accessed through the URL: <http://www.dcs.gla.ac.uk/~kh/Haskell1.3/IO.html>

Note that the state-transformer approach implies nothing about the model of the states. The database model and its effective Haskell-binding must therefore be newly designed. The proposed approach introduces typed surrogates to represent mutable objects and utilizes the other novel feature of Haskell, the type class system, to make it possible to discriminate between storable types and volatile types and to define overloaded database primitives and database combinators.

The rest of this section will overview these two important points: the database state model and the method to relax imperativeness of the state transformer approach. Two related issues, models of persistency and exception handling, will also be discussed briefly.

To make this paper easier for those who are not familiar with the Haskell language to read, the related Haskell language features are summarized in Appendix A and will also be explained in place (where, however, the explanation must be brief). For more formal and practical aspects of the language features, see Refs. [5] and [7].

1.1 Database states and update model

Referentially transparent update mechanisms have been proposed for a few database programming languages: **FDBPL** [12], **PFL** [14], and **Staple** [10]. In these languages, databases are represented as updatable environments (FDBPL and PFL) or are manipulated through *roots of persistence* with string identifiers (Staple). The former approach is difficult to incorporate into Haskell, simply because the language prohibits run-time modification of symbol bindings.

In the latter approach, when a database is to be updated, a referentially transparent I/O operation is performed to replace an old persistent root with a new one. Although this method is simple and is easy to use, when even a small fragment of a database is updated, all the paths from the persistent roots to the fragment must be duplicated [12]. This not only means that the cost of the update is high, also means that the whole database contents must be considered carefully if they are to be kept consistent. Moreover, *Staple* identifies persistent roots by string names, and therefore uses dynamic type checking by introducing the *Any* type to and from which every data can be coerced. Since Haskell is a statically typed language, the language specification would have to be modified.

The proposed approach, on the other hand, introduces typed artificial identifiers or surrogates to handle object mutability. Intuitively, a database is a family of collections indexed by types. A stored collection with an index, say σ , comprises pairs consisting of a surrogate and a value of type σ . A database object may refer to other objects through surrogates instead of normal direct pointers. Hence, every surrogate-data pair can be modified independently of the rest of the database. Since a database is indexed by types, types themselves are used to identify stored collections. Database operations are polymorphic, and which collection should be manipulated is determined by an explicit or inferred type signature.

Selection of appropriate database operations is performed through the class mechanism of Haskell. In Haskell, a class is a collection of algebraic types (*instances*) associated with a finite number of overloaded operators (*class operators*). Every instance has its specific implementation, or *methods*, of the class operators. At run-time (or, when possible, at compile-time), the class operators select appropriate specific methods from instance method dictionaries. These class constraints can be specified by programmers explicitly in type signatures or can be inferred by the language processors. In the latter case, the processors infer the most general type signatures including class-membership constraints.

The approach proposed here defines a class `Persistent` associated with database operators, and makes database types its instances. Through this *ad-hoc polymorphism*, database types are discriminated from non-database types. Although polymorphic types may also be instances of the `Persistent` class, polymorphic values cannot be stored in databases here, since collections of polymorphic values are not allowed in general. Instead, values of polymorphic types are required to be stored in a fully instantiated form. For example, `Tree a` (a type of trees of polymorphic leaves) may be specified as an instance of the `Persistent` class, but only fully instantiated values such as values of type `Tree Int` and `Tree String` can be stored in databases. Therefore, throughout this paper “database types” are used to denote such storable types instead of “instances of the `Persistent` class.”

Since Haskell is a purely functional language, operations with side effects are prohibited. Hence, the interface utilizes the monad of state transformers to perform referentially transparent update operations. Intuitively, the state transformers are functions which take a database state to a pair consisting of its operation result and a new database state. In a transaction execution, these transformers are executed one-by-one. Theoretically, since the transformers

generate *new* database states, there is no side effect. In practice, however, since the database operations may be executed strictly as in an imperative programming language, database operations can be implemented by side-effecting internal operations.

This monadic style of database state manipulation is also used in Ref. [13], which focuses on an impure and strict database programming language having ML-style references and which gives its denotational description in a pure typed functional language based on the monad of state transformers. The present paper treats the same aspect, but the target languages here are non-strict and purely functional. In such environments, describing database manipulation in the monad of state transformers is not sufficient. This paper therefore also proposes a method to relax the imperativeness of database manipulation.

1.2 Avoiding imperativeness in query formulation

The proposed approach could be implemented simply by defining state representation in the language, primitive database operators, and state combinators to construct complex database operators. Even a very simple query, however, must be coded as a series of primitive operations in a single state-transition sequence. Queries must be coded in an imperative style rather than in the basic coding style of lazy functional programming languages.

This limitation is placed by the loss of on-the-fly dereference of entity values through surrogates. If such dereference is permitted, the value of the dereference would depend on the order of evaluation. That is, it would be unpredictable. To avoid the limitation, the proposed approach maintains for each stored collection a tree of versions. The versions record part of the history of modification during program execution. At any point in state transitions, the current database state or a collection of the current versions may be “frozen,” or locked. The frozen database is read-only and may be supplied as arguments to retrieval operations. In addition, since locked versions can be traversed lazily, this versioning method also accommodates extent retrieval to lazy computation.

The approach proposed here differs from that proposed in FDBPL [12].² In FDBPL, database states cannot be controlled by user programs. Although a transaction may refer to multiple versions, there is no glue to combine transactions and no operation to control versioning. The present approach, on the other hand, treats versioning as a more explicit programming construct. The version granularity is finer, and versions at any point can be marked for later use.

Database freezing here also differs from *array freezing* [9]. Array freezing generates a standard Haskell array from a mutable one that is directly updatable in a state-transition sequence. Frozen databases resemble frozen arrays in that they are read-only, and that on-the-fly dereference is permitted. Database freezing, however, is closely related to transaction control. With or without versioning, because of the possibility of rollback, database systems maintain some update logs or intermediate states. The frozen versions are special intermediate states organized so that they can be accessed as efficiently as normal up-to-date states. Moreover, the new version of databases is generated only when required, whereas array freezing creates a copy without the later use of the frozen array. To clarify the difference between database and array freezing, this paper will briefly describe the monad for lockable array transformers. It is a variation of the monad for array transformers [17]. Although the lockable array monad cannot be used as a complete mathematical model of the database freezing, it clearly accounts for the difference between database freezing and array freezing.

1.3 Models of persistence

The model used here follows the *extension model of persistence*. That is, for each database type, there is a persistent set of entities, or *extent*, of that type, and the extent is maintained automatically by the underlying system. Another persistency model is the *reachability model of persistence*, in which programmers explicitly maintain persistent collections, and entities in the persistent collections are treated as persistency roots. The notable advantages of an approach based on the reachability model are that it is more flexible and with some additional cost of programming can simulate most of the operations for the extension model, and that no reference dangles since entities are deleted only when

---

²In Ref. [12], two formal models of transactions are proposed: one that specifies a new database based on an old one, and one based on a list of database history. This paragraph considers the latter.

there is no path from persistent collections. The disadvantage of such an approach, however, is its complexity in entity deletion that is also found in the persistency root approach described above.

This paper focuses on the extension model of persistence and discusses the issue on imperative update and lazy retrieval in this model. But to make the discussion as general as possible, this paper will also explain how to incorporate the reachability model.

1.4 Exception handling for dangling references

An inherent problem in the extension model of persistency is possibility of dangling references. If programmers had to write error handling codes whenever dereference operators were used, programs would be much more complex. What is worse, when on-the-fly dereference is used, it is more difficult to specify exception handling than it is when explicit state transitions are used. A partial solution to this problem is the proposed use of overloaded operators. Although exceptions cannot be handled as dynamically as in ML [11], the proposed exception handling scheme indicates that purely functional languages can manage exceptions without increasing the burden of programming tasks.

2 Basic Database Manipulation Interface

The interface comprises database state representation, state-transformer combinators, primitive operators, and a transaction model. After this section gives the abstract definition of the model and explains the schema declaration method, it introduces the interface components in that order. A few query examples will be shown at the end of this section.

2.1 Data model

Let Σ be the set of all the types in a database, and let $Ref(\sigma)$ and $Val(\sigma)$, respectively, be the sets of all the surrogates and values of type σ ($\in \Sigma$). As noted previously Σ is a set of ground types. Then a database state is a collection of Σ -indexed triplets $(O_\sigma, s_\sigma, l_\sigma)$ where O_σ is a finite subset of $Ref(\sigma)$, s_σ is a finite map from O_σ to $Val(\sigma)$, and l_σ is a list representation of s_σ . The last item (l_σ) could be omitted if the model were used only at the abstract level. In this research, however, the model is constructed so that it can be represented in Haskell. Concrete and easy-to-manage representation of database states is therefore important. The primitive operations are defined as follows. Let $o \mapsto v$ be a binary association from o to v . Then the operational part comprises five operators:

all_σ	retrieve l_σ ;
$deref_\sigma(o)$	retrieve $o \mapsto v$ from s_σ ;
$upd_\sigma(o \mapsto v)$	replace $o \mapsto w$ in s_σ by $o \mapsto v$;
$new_\sigma(v)$	insert $o_\sigma \mapsto v$ into s_σ for a new o_σ ;
$del_\sigma(o)$	delete $o \mapsto v$ from s_σ

Type signatures of the operations are not given here, since they depend on the way to pass and return database states.

2.2 Database schema

Consider as an example the Part-Supplier database described in Ref. [2]:

- A part is either basic or composite. A basic part has *name*, *cost*, *mass*, *used-by*, and *supplied-by* attributes, and the attributes of for a composite part are *name*, *assembly-cost*, *mass-increment*, *used-by*, and *composed-of*.
- A supplier has *name*, *address*, and *supplies* attributes.

These objects can be represented by algebraic data types, or sum-of-product types, declared thus:

```
> module Parts where
> data Part = Basic      String Int Int [Part] [Supplier]
>             | Composite String Int Int [Part] [(Part, Int)]
```

Figure 1: A database state of the Part-Supplier database.

```

>
> data Supplier = Supplier String String [Part]
    
```

(the leading > signs are used to indicate program fragments.)

The database schema comprises these two types with slight modification. Because stored objects refer to other objects through surrogates instead of direct pointers, the declaration is modified to get a real database schema:

```

> module Parts where
> data Part = Basic      String Int Int [DBRef Part] [DBRef Supplier]
>             | Composite String Int Int [DBRef Part] [(DBRef Part, Int)]
> instance Persistent Part
>
> data Supplier = Supplier String String [DBRef Part]
> instance Persistent Supplier
    
```

Here “DBRef α ” is the Haskell representation of $Ref(\alpha)$, and Persistent is a class of database types. Processing this schema creates a database called “Parts” comprising Part and Supplier types. Figure 1 depicts a state of the Part-Supplier database where ι_i ($i = 1, 2, 3, \dots$) are the surrogates for stored parts and where κ_j ($j = 1, 2, 3, \dots$) are the surrogates for stored suppliers.

2.3 Database states and state-transformer combinators

The database states are of type DBState. This type is an abstract type and the details depend on implementation methods. The type synonym for state transformers and three basic state transformer combinators are declared as follows:³

```

> type DB a = DBState -> (a, DBState)
>
> (>>@=)    :: DB a -> (a -> DB b) -> DB b
> (>>@)      :: DB a ->          DB b   -> DB b
> returnDB :: a -> DB a
    
```

These respectively correspond to IO a, >>=, >>, and return for I/O state transformers. The diagrammatic representations of the database state transformers and the combinators are shown in Figures 2 and 3. Intuitively, $m \gg@= k$ executes m and then executes k passing the result of m to k . $m \gg@ m'$ executes m and m' sequentially. And $\text{returnDB } x$ constructs a database operator from an arbitrary expression.

In these figures, database operations generate new database states, and the operations are thus referentially transparent. To improve the performance of update operations, however, destructive update of the database store is preferred. So the more practical image of database operations is shown in Figure 4. Operations are referentially transparent at

³ :: symbol is read as “has type.”

Figure 2: Database operation as a state transformer.

Figure 3: Combinators of database operations

the logical level, while they may be referentially opaque at the physical level. To make them fully transparent, one constraint must be enforced: *the state transformers must be processed imperatively, or strictly*. In Figure 4, this constraint means that when the state s' is constructed, the intermediate expression e must have been evaluated so that no subexpression depending on the the modifiable parts of the previous state s is suspended. Since only the surrogate-value pairs are modifiable parts of database states, it is necessary only to ensure that the primitive database operators are executed strictly. This restriction is not difficult to enforce, because they are built-in primitive operators.

Figure 4: Imperative manipulation of database states.

2.4 Database primitive

The five primitive operators are declared in Haskell as follows:

```
> type DBAssoc a = Assoc (DBRef a) a
> allDB :: (Persistent a) => DB [DBAssoc a]
> derefDB :: (Persistent a) => DBRef a -> DB (DBAssoc a)
> updDB :: (Persistent a) => DBAssoc a -> DB ()
> newDB :: (Persistent a) => a -> DB (DBAssoc a)
> delDB :: (Persistent a) => DBRef a -> DB ()
```

Assoc is one of the Haskell built-in types, and an element of the type is a binary association represented as $o := v$. “(Persistent a) =>” specifies the constraint that when these functions are used in more specific typing context, the variable a must be replaced with an instance of the Persistent class.

Remember that in the state-transition sequence, state transformers must be processed strictly to make them referentially transparent while allowing for destructive update of the database store. The allDB operator is not an exception. The list of surrogate-value pairs must be constructed immediately at the time of state transition. This is another source of imperativeness.

2.5 Transaction

The transaction function executes a given database operator in a I/O state transition sequence. The function is declared thus

```
> transaction :: DB a -> IO a
```

The modified database state is committed at the end of execution only if all the included operations are processed successfully.

Program execution starts with evaluation of the dbMain function. This function must be bound with an I/O action possibly including database transactions. When there is any trouble in the course of execution, the database state is rolled back to the initial one.

2.6 Examples

Now consider a simple query: retrieve basic parts that cost more than \$100. This can be coded as the following state transformer:

```
> transaction (
>   allDB >>@= \parts ->
>   returnDB [ name | ( _ := (Basic name cost _ _ _)) <- parts,
>                   cost > 100 ] )
```

where $\backslash x \rightarrow e$ represents a lambda expression $\lambda x . e$, and $\_$ is a wild-card pattern or an anonymous variable. The result of allDB is passed to the right-hand side of the $\gg@=$ operator as the argument of the lambda expression.

As noted in the previous subsection, the starting point of program execution is an I/O action bound to dbMain. For instance, if the result of the above query is to be printed on terminals, the full-fledged program may look like this:

```
> module Main(dbMain) where
> dbMain =
>   putStr "Basic parts that cost more than 100\n" >>
>   transaction (
>     allDB >>@= \parts ->
>     returnDB [ name | ( _ := (Basic name cost _ _ _)) <- parts,
>                     cost > 100 ] ) >>= \names ->
>   putStr (lines names)
```

(putStr is an I/O function that prints a given string on terminals.)

The next example updates the addresses of suppliers named “SUP1000” with new_address defined elsewhere:


```

> transaction (
>   allDB >>@= \suppliers ->
>   ( let actions =
>       [ updDB (sid := Supplier name new_address supplies)
>         | (sid := Supplier name _ supplies) <- suppliers,
>           name == "SUP1000" ]
>   in foldr (>>@) (returnDB ()) actions )

```

The list of update actions is bound to the `actions` local variable and is executed sequentially by the `foldr` built-in operation. Since the update operations are performed imperatively, the database store is updated destructively, and no copy is generated in the course of the update sequence.

Consider a query a little bit more complex: retrieve the total mass and total cost of a composite part. Although this can be coded in the basic database interface, the query expression is very complex and the coding style is far from that of lazy functional languages. The code is omitted here, because more readable coding is shown in the next section using on-the-fly dereference.

3 Database Versioning and Lazy Retrieval

Figure 5: Version trees and a version table. A square represents a certain version and the number in it shows the version number. Squares having the same background pattern are generated by a single transaction.

The basic database manipulation interface described in the previous section has two sources of imperativeness: (1) imperative construction of a query because of the loss of on-the-fly dereference through surrogates, and (2) imperative execution of the `allDB` operation. This section introduces the concept of versions into the database state model, so that these two sources of imperativeness are avoided.

3.1 Versions

During program execution, every stored collection is associated with a tree of versions which records some part of the modification history. Figure 5 shows the conceptual representation of such trees. Note that database states are not necessarily committed and that versions may be made current at any point of I/O operations. Hence, the form of version relationships is a tree as shown in the left one in the figure.

At any point in a sequence of I/O operations, the current set of database versions can be frozen or locked by an I/O action called `getDB`. This operation locks the versions and returns the version set as a version table of type

Database as shown in Figure 5. This type is an abstract one, and the details depend on implementation methods. Even though polymorphic `Persistent` instances imply that infinitely many database types are allowed, the table is well-defined since at a certain point of database life cycle there are only finitely many database types. The reverse operation is `restoreDB` that just marks the given version set as current.

3.2 On-the-fly dereference

There are two primitive operators to retrieve data from the frozen databases lazily:

```
> vAllOf    :: (Persistent a) => Database -> [DBAssoc a]
> vValOf    :: (Persistent a) => Database -> DBRef a -> a
```

where `Database` is the type of the version tables. The semantics may be clear from their names and type signatures. A version table, say `init_vt`, including all the initial versions can be created as a part of I/O states. So we can define the more frequently used operators:

```
> allof = vAllOf init_vt
> valof = vValOf init_vt
```

Now that on-the-fly dereference is permitted, we consider the query at the end of the last section again: retrieve the total mass and total cost of a composite part. This query can be written as follows:

```
> cAndM (Basic _ cost mass _ _)
>   = (cost, mass)
> cAndM (Composite _ cost mass _ subparts)
>   = let
>       sub_cm = [ (c * quant, m * quant) |
>                 (sub, quant) <- subparts,
>                 (c,m) <- [cAndM (valOf sub)] ]
>       c_sum = sum (map fst sub_cm)
>       m_sum = sum (map snd sub_cm)
>   in
>       (c_sum + cost, m_sum + mass)
```

Notice the use of `valOf` in the list comprehension. The rest is the same as that for the non-persistent case. If this function must be generalized to one that computes the values according to a certain frozen database, then add an additional argument, say `db`, of type `Database` and replace `valOf sub` with `vValOf db sub`.

3.3 Lazy retrieval of surrogate-value pairs

Versioning allows for straightforward implementation of lazy retrieval in a state-transition sequence. When a certain version is retrieved by `allDB`, the version is also locked so that lazy reading is performed safely. As described in Ref. [3], lazy retrieval of a locked version by `allDB` and `vAllOf` can be implemented using a currency pointer and *find-first* followed by a series of the *find-next* operation. A following update operation to the locked version will generate a new version with the specified modification (Figure 6).

3.4 Version creation and elimination

A new version must be generated only when a locked version is to be updated. In other cases, the current versions may be modified destructively. When a version table or a currency pointer become garbage, the garbage collector can unlock the included versions. If there is no lock on a non-current version, the version may be eliminated safely.

Figure 6: Lazy retrieval of surrogate-value pairs.

3.5 Relation to the lazy state-transformer approach

The present approach is based on imperative state transformers, but there is another approach called the *lazy state transformer approach* [1, 12, 10]. In that approach, states are basically read-only and new states are constructed without modifying the old states. To reduce the cost of state construction, unchanged parts of the new and the old states are shared by backward pointers. And, optionally, when a reference count garbage collector is used, the database element with reference count 1 may be updated destructively.

In the present approach, on the other hand, the order of consideration is reversed. Basically, states are updated destructively, and only when a state must be kept for later use, the state is locked and a new version is generated if necessary.

The difference between the performance obtained with these approaches is not clear and must be investigated further. At least for successive modification of database states, however, the imperative update mechanism may be executed more efficiently. Moreover, even if locking and updating are performed alternately, common parts of versions may be shared as in the lazy state-transformer approach.

3.6 Relation to array freezing

This subsection clarifies the difference between the database versioning and array freezing by showing formalization of the database versioning using the monad of lockable array transformers. The monad is a variation of the monad of array transformers [17]. Here we consider array subscripts as surrogates and associated array values as the associated entity values. A database state is represented by a pair consisting of an array and an identifier generator of type $(Arr, Integer)$, where Arr is the type of arrays with indices of type $Integer$ and values of type Val . For brevity, types, extent management, and error handling are not treated here.

First consider the database monad as an array-transformer monad. The monad can be defined as follows:

```

type DB a    = State → (a, State)
type State   = (Arr, Integer)
returnDB a    = λx.(a, x)
m >>@= k      = λx.let (a, y) = m x in
                  let (b, z) = k a y in
                  (b, z)
    
```

If *index* and *update* are the read and destructive write operations for Arr , the basic operations are defined like

this:

$$\begin{aligned} \text{newDB } v &= \lambda(x, g).(g, (\text{update } g \ v \ x, g + 1)) \\ \text{derefDB } i &= \lambda(x, g).(\text{index } i \ x, (x, g)) \\ \text{updDB } i \ v &= \lambda(x, g).(\text{()}, (\text{update } i \ v \ x, g)) \end{aligned}$$

Note that, for this monad to be single threaded, the `derefDB` operation must be executed strictly: before returning the operation result, it must compute `index i x`.⁴ Let us define the `getDB` and `vValOf` operations following the line of the array-freezing approach. Supposing `dup` is an operation which duplicated an array, a simple definition might be as follows:

$$\begin{aligned} \text{getDB} &= \lambda(x, g).(\text{dup } x, (x, g)) \\ \text{vValOf } x \ i &= \text{index } i \ x \end{aligned}$$

Like `derefDB`, this operation must be strict: before returning the operation result, it must compute `dup x`. This is the basic idea of array freezing.⁵ In this scheme, even if the state array is not going to be modified in future, it is duplicated. In addition, `dup` may generate multiple copies of a single array.

On the other hand, the database versioning simply locks the state and duplication is performed only when necessary. Let us suppose that locking, unlocking, and lock test functions on *Arr* are available:

$$\begin{aligned} \text{lock}, \text{unlock}, \text{clear} &:: \text{Arr} \rightarrow \text{Arr} \\ \text{locked} &:: \text{Arr} \rightarrow \text{Bool} \end{aligned}$$

where `lock` (`unlock`) increments (decrements) the lock counter of the given array and `clear` makes the lock counter 0. Then the database versioning operations can be defined as follows:

$$\begin{aligned} \text{dup}' \ x &= \text{if } \text{locked } x \text{ then } \text{clear } (\text{dup } x) \text{ else } x \\ \text{newDB } v &= \lambda(x, g).(g, (\text{update } g \ v \ (\text{dup}' \ x), g + 1)) \\ \text{derefDB } i &= \lambda(x, g).(\text{index } i \ x, (x, g)) \\ \text{updDB } i \ v &= \lambda(x, g).(\text{()}, (\text{update } i \ v \ (\text{dup}' \ x), g)) \\ \text{getDB} &= \lambda(x, g).\text{let } x' = \text{lock } x \text{ in } (x', (x', g)) \\ \text{vValOf } x \ i &= \text{index } i \ x \end{aligned}$$

where `dup'` tests the given array and duplicates the array only when it is already locked. The implementation has an overhead for lock testing, but unnecessary duplication does not occur in database state-transition sequences.

4 Exception Handling

In the model described so far, it is possible that some reference dangles. The situation is worse if that happens during on-the-fly dereference: even if programmers can check whether or not a reference dangles, that checking will make programs more complex. But a simple exception handling scheme is available in the proposed approach. Recall that database types are declared through instance declarations of the `Persistent` class. So we make the exception handler one of the class operators like this:

```
> class Persistent a where
>   whenDangling :: Database -> DBRef a -> a
>   whenDangling _ _ = error "dangling reference"
```

In this declaration, `whenDangling` is declared with its default method. This method is used when the instances do not declare their own methods explicitly. Recall also that dereference is performed by the `vValOf` operator. Whenever it detects dangling references, it applies `whenDangling` to the current database and the given surrogate.

⁴As noted in Section 2.3, the `index` operation must be a built-in operation having the strict execution semantics.

⁵The type for *mutable* arrays and that for *immutable* ones are different in Haskell, but the basic idea is similar.

For example, let `db` denote a value of type `Database` and `v` denote a surrogate which dangles in `db`. Then the following equivalence holds:

```
vValOf db v  ≡  whenDangling db v
              ≡  error "dangling reference"
```

This means that dereferencing through dangling references stops program execution. When some default value exists, users may specify the value in instance declarations like this:

```
> instance Persistent Part where
>   whenDangling _ _ = Basic "B000" 0 0 []
```

In this case, if a surrogate `v` dangles in a database `db`, the following equivalence holds:

```
vValOf db v  ≡  whenDangling db v
              ≡  Basic "B000" 0 0 []
```

This solution is far from satisfactory because the exception handler is specified in schema definitions rather than in application programs. This issue must be investigated further.

5 Reachability model of persistence

This paper has so far adhered to the extension model of persistence. As described in Section 1, however, the reachability model of persistence is more flexible even though it is subject to a certain complexity in deleting entities. This section therefore briefly describes the way the reachability model can be used and how the database versioning described in this paper works under the reachability model.

We only have to replace `allDB` and `vAllOf` by appropriate extent management operations. Indeed, `newDB`, `updDB`, `derefDB`, and `vValOf` consult or modify surrogate-value relationships and are independent of the model of persistence. The `delDB` operation is not required now, since entities are deleted automatically when they can not be reached from the stored collections.

Let us start with the example of the Part-Supplier database:

```
> data PartsDB = PartExt [DBRef Part]      [DBAssoc Part]
>               | SuppExt [DBRef Supplier] [DBAssoc Supplier]
> instance PersistentExt PartsDB
```

where `PartExt` and `SuppExt` are used to manipulate the explicit extensions of `Part` and `Supplier` entities. The arguments of the data constructors are used to pass the lists of surrogates to and receive surrogate-value pairs from the persistent storage. The usage will become clear in the examples below.

The instance declaration of the `PersistentExt` class gives the following operations to access the explicitly maintained collections:

```
> getExtDB :: (PersistentExt a) => a -> DB a
> addExtDB :: (PersistentExt a) => a -> DB ()
> delExtDB :: (PersistentExt a) => a -> DB ()
> vExtOf   :: (PersistentExt a) => Database -> a -> DB a
> extOf    :: (PersistentExt a) => a -> DB a
> extOf    = vExtOf init_vt
```

where `init_vt` represents the initial version table. For instance, the program to retrieve basic parts that cost more than \$100 can be coded as follows:

```
> transaction (
>   getExt (PartExt [] []) >>@= \(PartExt _ parts) ->
>   returnDB [ name | _ := Basic name cost _ _ _ <- parts,
>               cost > 100 ] )
```

where “PartExt [] []” specifies which extension is to be retrieved. And the result data, “PartExt _ parts,” gives the stored extension list. By using extOf, we can code this query without considering state transitions:

```
> [ name | parts <- let PartExt _ parts = extOf (PartExt [] [])
>   in parts
>   _ := Basic name cost _ _ _ <- pairs,
>   cost > 100 ]
```

Extension modification is simply specified using addExt and delExt. A function to add a new supplier can be written simply by calling addExt like this:

```
> addSupplier supp
> = transaction (
>   newDB supp >>@= \(sid := _) ->
>   addExtDB (SuppExt [sid] [])
```

Note that newDB does not automatically maintain instances. Programmers must explicitly insert the newly created surrogate into the supplier extension.

The database model is defined more formally as follows. A database schema is a triple (Γ, Σ, Φ) where Γ is a finite set of extension tags, Σ is a set of database types, and Φ is a finite map from extension tags to their associated database types. A database state comprises a surrogate pool, an object heap, and a family of extent definitions as follows:

$$(S = \{o_1, o_2, \dots\}, \text{heap} = \{o \mapsto v\}_{o \in S}, \text{ext} = \{e_\tau\}_{\tau \in \Gamma})$$

where the heap should be a type-consistent function:

$$(\forall o \in S)(\exists! \sigma \in \Sigma)(o \in \text{Ref}(\sigma) \wedge \text{heap}(o) \in \text{Val}(\sigma))$$

And, the extent definitions must be consistent with the schema:

$$(\forall \tau \in \Gamma)(\forall o \in e_\tau)(o \in S \wedge o \in \text{Ref}(\Phi(\tau)))$$

Note that because the reachability model of persistence is assumed here, when a database state is committed at the end of program execution, every entity that is not reachable from $\{e_\tau\}_{\tau \in \Gamma}$ is squeezed out of the heap.

Even when we use the extension model of persistence, the versioned database states can be used without modification. That is, when getExtDB is executed, it does not materialize the extent immediately. Instead, it obtains locks on the surrogate-value map and the extent, and then creates a cursor structure to traverse the extent lazily. The scheme to create and delete versions is the same as that for the extension model of persistency.

6 Notes on the Implementation

To clarify implementation dependent topics, this section comments on a sample implementation using C and *Glasgow Haskell Compiler* [6].

Storing closures: A graph of closures is stored as a string which represents the contents. In Haskell, conversions to and from string representation are respectively performed through show and read overloaded functions which are operators of the Text class. Therefore, we impose another restriction on database types: they must be Text class instances. This restriction means that function values, suspensions, and cyclic structures are not allowed to be persistent.⁶ Moreover, shared closures are duplicated.

Identification of stored collections by types: In the present context, it suffices to compute string representation of modules and types. This is done by introducing another class:

⁶ Although a function type can be an instance of Text, “show then read” can not recover the closure contents.

```
> class Representable a where
>   typeRepr :: TypeRepr a
```

The overloaded operator `typeRepr` computes the required representation of `a`. Finally, the database types are declared like this:

```
> data Part = ...
>   deriving (Text, Representable)
> instance Persistent Part
```

where the `deriving` clause specifies that system-supplied default instantiation is performed during compilation.⁷

Versioning: A collection of surrogate-data pairs is stored in an indexed file, and a new version can be generated simply by duplication of the index part. The data part is treated as a heap during program execution. Every version has a counter that records the number of locks, and the locking overhead is very small. Of course, more efficient schemes can be used for version management. For instance, utilizing explicit index page tables would decrease the number of required pages.

7 Conclusion

This paper has described a database manipulation interface for the Haskell programming language. The basic design of the interface is based on the imperative state-transformer approach, so that destructive update is used and operations are referentially transparent. In addition, version maintenance is introduced to permit lazy-retrieval and on-the-fly dereference through surrogates. Since database retrieval can be programmed without considering complicated state transitions, basic programming style of Haskell is retained.

Other than the implementation issues, this approach has a few issues still to be investigated. First, it is possible for programmers to mistake a version table for another. For the initial state, there is no trouble because they can be accessed through specialized functions. But if update and retrieval alternate in a complex way, or if the interface should be used by naive users, more elaborate methods may be required to control the database states like the dedicated query interface proposed in Ref. [15].

Another issue is that using types as collection identifiers may be inappropriate in a situation where types are too coarse to model the real world. Since some entities may have different roles in difference contexts, a single list of entities may fail to capture the real semantics and the addition of more database types could complicate queries. This problem might be avoided to some extent by adopting the reachability model of persistence, but that model makes the extension management tasks (especially the entity deletion task) more complex. Moreover, the proposed approach requires that a stored collection is associated with a ground type. So polymorphic functions, even if we permit them to be stored in databases, must be stored in a specialized form.

References

- [1] G. Argo, J. Hughes, P. Trinder, J. Fairbairn, and J. Launchbury. Implementing functional databases. In F. Bancilhon and P. Buneman, editors, *Advances in Database Programming Language*, pages 165–176, 1990.
- [2] M. P. Atkinson and P. Buneman. Types and persistence in database programming languages. *ACM Comput. Surv.*, 19(2):105–190, Jun. 1987.
- [3] P. Buneman, R. E. Frankel, and R. Nikhil. An implementation technique for database query languages. *ACM Trans. on Database Syst.*, 7(2):164–186, 1982.
- [4] P. Buneman, D. Sucio L. Libkin, V. Tannen, and L. Wong. Comprehension syntax. *ACM SIGMOD RECORD*, 23(1), Mar. 1994.

---

⁷Haskell prohibits user-defined classes to be used in `deriving` clauses. So for the sample implementation, the `Representable` class was added as a built-in class and the language processor was modified.

- [5] A. J. T. Davie. *An Introduction to Functional Programming Systems Using Haskell*. Cambridge University Press, 1992.
- [6] C. Hall, K. Hammond, W. Partain, S. L. Peyton Jones, and P. Wadler. Glasgow Haskell Compiler: A retrospective. In J. Launchbury and P. Sansom, editors, *Functional Programming, Glasgow 1992*, pages 62–71. Springer-Verlag, 1993.
- [7] P. Hudak, S. L. Peyton Jones, and P. Wadler, eds. Report on the functional programming language Haskell, version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.
- [8] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *Proc. of the ACM Symposium on POPL*, pages 71–84, Jan. 1993.
- [9] J. Launchbury and S. L. Peyton Jones. Lazy functional state threads. In *Proc. of ACM SIGPLAN'94 Conf. on PLDI*, pages 24–35, Jun. 1994.
- [10] D. J. McNally and A. J. T. Davie. Two models for persistence in lazy functional programming systems. *SIGPLAN NOTICES*, 25(5):43–52, May 1991.
- [11] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. The MIT Press, 1990.
- [12] R. S. Nikhil. The semantics of update in a functional database programming language. In F. Bancilhon and P. Buneman, editors, *Advances in Database Programming Language*, pages 403–421, 1990.
- [13] A. Ohori. Representing object identity in a pure functional language. In *Proc. of the 3rd Intl. Conf. on Database Theory*, pages 41–55. Springer-Verlag, 1989.
- [14] C. Small. A functional approach to database updates. *Information Systems*, 18(8):581–595, Dec. 1993.
- [15] P. Trinder. Comprehensions, a query notation for DBPLs. In P. Kanellakis and J. W. Schmidt, editors, *Proc. of the 3rd Intl. Workshop on DBPL*, pages 55–68. Morgan Kaufmann, 1991.
- [16] P. Wadler. The essence of functional programming. In *Proc. of the ACM Symposium on POPL*, pages 1–14, Jan. 1992.
- [17] P. Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi, Proc. of the Marktoberdorf Summer School, Jul 30 – Aug 8, 1992*, 1992.

A Brief Introduction to Haskell

This section overviews a few features of Haskell [7] related to this paper. Details can be found in tutorial texts such as Ref. [5] for generic topics, and details on the monadic I/O system can be found in Refs. [16] and [8]. Note that in the following explanation, all the lines in program fragments are for clarity preceded by > signs.

A.1 Data Types

A type is either an algebraic one or a type synonym. Consider as an example data types for the Part-Supplier database [2] comprising *Part* and *Supplier*:

- A part is either basic or composite. A basic part has *name*, *cost*, *mass*, *used-by*, and *supplied-by* attributes, and a composite part has *name*, *assembly-cost*, *mass-increment*, *used-by*, and *composed-of* attributes; and
- A supplier has *name*, *address*, and *supplies* attributes.

These objects can be represented by algebraic data types declared as follows:


```
> data Part
>   = Basic      String Int Int [Part] [Supplier]
>   | Composite String Int Int [Part] [(Part, Int)]
> data Supplier
>   = Supplier String String [Part]
```

where `Part` and `Supplier` (left-hand side) are called *type constructors*, and where `Basic`, `Composite`, and `Supplier` (right-hand side) are called *data constructors*. In the above example, two more type constructors are used: one is the tuple type constructor in `(Part, Int)`, and the other is the list type constructor in `[Supplier]`.

A type synonym is used to name a type. If we want to name “list of parts” type, the following declaration suffices:

```
> type PartList = [Part]
```

A.2 Monadic I/O

In Haskell there are three I/O styles: *dialogue*, *continuation*, and *monadic I/O*. The interface described in this paper is dependent on the monadic I/O system, so that system is described here.

An I/O operation using the monadic I/O system is a state transition function. Consider as an example the `readFile` function. This function constructs an I/O operation from a given filename, and the operation is diagrammatically shown like this:

The input is an I/O state before the action, and the output is a pair consisting of the file contents and the new I/O state. From the viewpoint of types, every I/O action returning a value of type `a` is of type `IO a`, where the implementation is hidden from users. For example, the `readFile "person.dat"` is of type `IO String` because the action returns the contents as a string.

There are two combinators associated with the `IO` type: a function, `return`, and an infix operator, `>>=`. `return` constructs an I/O action from any expression. The diagrammatic representation of the action denoted by `return e` is

The action returns `e` as the result and invokes no state transition. `return` is of type `a -> IO a`.

The other combinator, `>>=`, comprises two I/O actions and is typed as follows:

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

where the `::` symbol is read as “has type.” Consider an expression `m >>= k` which is diagrammatically represented like this:

I/O states are transformed by `m` and `k` sequentially, and the result of `m` is passed to `k`.

For more simpler cases where an intermediate result is not important, the combinator, `>>`, is available:

```
> m >> m' = m >>= \_ -> m'
```

This combinator constructs a sequentially composition of the given operators.

A.3 Class Mechanism

Haskell uses the *class* mechanism to control operator overloading. A class is a family of algebraic types associated with finite number of overloaded operators. The behavior of an overloaded operator for a specific instance is called a *method*. Consider as an example the definition of the `Eq` class:

```
> class Eq a where
>   (==), (/=) :: a -> a -> Bool
>   x /= y = not (x == y)
```

The second line declares that `==` (equality) and `/=` (inequality) are the class operators. The last line gives the *default method* of `/=`.

The following statements declare instances of the `Eq` class:

```
> instance Eq Int where
>   x == y = primEqInt x y
> instance Eq Float where
>   x == y = primEqFloat x y
```

Overloaded operators are resolved at run time (or when possible at compile time). An expression `x == y` is treated as `primEqInt x y` if `x` and `y` have type `Int`, and as `PrimEqFloat x y` if `x` and `y` have type `Float`.

Instances can be derived by a Haskell language processor by adding deriving clauses in algebraic data type declarations:

```
> data Supplier = Supplier String String [Part]
>   deriving Eq
```

By this deriving clause, `Supplier` is made an instance of the `Eq` class. Note that only instances of predefined classes can be derived and that derived methods are predefined in the language specification. In this case, `Supplier` values are equal when each of their three component values is equal.