お茶の水女子大学自然科学報告　第48巻　第2号
Natural Science Report, Ochanomizu University, Vol. 48, No. 2 (1998)

Realizing State-Based Database Concepts in a Non-Strict, Statically Typed, Purely Functional Persistent Programming Language

Yoshihiko Ichikawa
Department of Information Sciences, Ochanomizu University

Abstract

This paper proposes a methodology to manipulate the database state in a non-strict, purely functional programming language. The primary target is Haskell, which is the standard for such programming languages and has been known for its use of a state-transformer monad to handle input/output operations and the type class mechanism to incorporate *ad hoc* polymorphism. The main contribution of this paper is to address and propose solutions to the key issues of making the lanaguage *persistent*. While the state-transformer monad naturally structures state-based operations, it complicates programming tasks because of the explicit *single-threadedness*. To lessen this inherent burden of the programming tasks, the proposed method makes use of explicit versioning of the database state, which can be retrieved lazily, even though the primary database state is updated destructively. The ability of multiple state manipulation naturally extends to view maintenance, exception handling, and support for the "what-if" semantics of execution. In addition to this feature, persistent roots are identified by their types instead of by their string- or variable-names. This allows every expression, even including root manipulation, to be typed statically. The supported programming environment also provides programmers with "hooks" to customize primitive operations, and can be generalized to support transaction-boundary rule firing.

1 Introduction

New database applications that require more flexible data structures and more computational power have recently emerged (Jacobs et al. 1995; Flickner et al. 1995; Christophides et al. 1994). The persistent programming system is one of the prominent tools to support the construction of such applications (Atkinson et al. 1983; Atkinson and Morrison 1995; Paton et al. 1996). In contrast to the ordinary database programming style, a persistent programming language extends a volatile programming language so that database access is abstracted by on-memory data access (Fig. 1). Therefore, effectiveness of a persistent programming language is determined to a certain extent by that of the underlying programming paradigm, and by that of the adopted method to identify persistent data. Although "what is the best paradigm" is controversial, the non-strict, statically typed, purely functional programming paradigm naturally exhibits favorable features, summarized as follows:

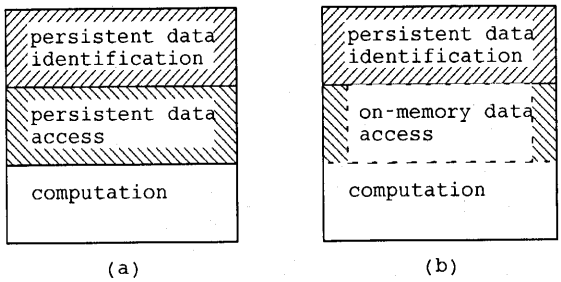


Figure 1: Concept of persistent programming; (a) the structure of ordinary database programs; and (b) persistent programming languages abstract persistent data access through on-memory data access.

- Non-strictness (or laziness) [1] allows the users to write mutually-dependent complicated data structures with no additional cost.
- From a user's perspective, static type checking ensures that all the written programs conform to the expected database schema, *viz.*, the collection of types and operators.
- It is easier to make use of mathematical reasoning because of its purity. Since the purity ensures every piece of programs is free from side effects, the potential optimizability based on equational reasoning is higher than that of others even when restricted database queries and general purpose computations are mixed freely (Poulovassilis and Kind 1990; Poulovassilis and Small 1996).

For the arguments in favor of the functional paradigm itself, refer to Hughes (1989), Bird and Wadler (1988), Holyer (1991), and Davie (1992).

In spite of this potential feasibility of the paradigm, making such a programming language persistent should overcome two key issues with respect to database manipulation:

- Database management tasks include state-based concepts: database update, view maintenance, integrity constraints, and active rules. The paradigm, however, does not have explicit state concepts.
- Prevailing methods to identify persistent roots resort to name- or string-identifiers of them (Dearle et al. 1989; McNally and Davie 1991; McNally 1993; Small and Poulovassilis 1991). Regrettably, name-identifiers would compromise the purity, and string-identifiers would require dynamic type checking.

As for the state-based concepts, the functional programming community has proposed effective techniques, including *continuation*, *linear typing* (Wadler 1990), and *state-transformer monads* (Peyton Jones and Wadler 1993; Gordon 1994; Launchbury and Peyton Jones 1994), to incorporate the state-based programming tasks such as input/output (I/O) operations, and mutable arrays. These techniques are also effective in the context of the persistent functional programming. Indeed, command-continuation, one form of continuation-style, was used by Small (1993) to order database update commands linearly, and linear typing was adopted by Sutton and Small (1995) to ensure the confluence of expressions including side effects. The state-transformer monad was used by Ohori (1990) to formulate the concept of object-identifiers, and its implicit use, i.e., referentially transparent state-based computation, has also been seen in Nikhil (1988) and Nikhil (1990).

The author's previous work (Ichikawa 1995) also made use of the state-transformer monad with multiple versions so that destructive updating and lazy retrieval were mixed in a purely functional programming language. The primary target was Haskell, the standard for such programming languages (Hudak et al. 1992; Peterson and Hammonad, eds. 1996; Peterson and Hammonad, eds. 1997), whose notable features include the use of a state-transformer monad to handle I/O operations and the type class mechanism to incorporate *ad hoc* polymorphism (or *function overloading*).

The present paper completes the previous work by incorporating other important state-based tasks into the methodology. The features of the proposed methodology with respect to the state-based tasks are summarized as follows:

- The database state-transformer monad allows the database state to be updated destructively without compromising the purity of the paradigm.
- Name equivalence on abstract entities can be supported.
- The multiple database versioning reduces the issue of destructive updating and lazy retrieval to a simple concurrency control issue with a single writer and multiple readers.
- The versioning naturally extends to fully lazy, i.e. just on demand and just once, computation of view values.

The proposed methodology identifies values of persistent roots by their types. The key technique is function overloading. An overloaded function changes its behavior according to its types specified by the context, thus the location of a persistent root is associated with the behavior, having another feature:

[1] Non-strictness does not necessarily imply laziness, nor *vice versa*. However, these terms are often used interchangeably.

- Every expression is typed statically, even if it includes manipulation of persistent root values, without compromising the purity of the paradigm.

For the safety of the location manipulation, the proposed methodology is required to restrict root values to be of ground types, since polymorphic mutable locations cause a typing trouble in the Haskell's typing system, as has been seen in the context of ML (Milner et al. 1990; Paulson 1996).

Lastly, it shall be pointed out that the overloaded primitive functions play another role in the proposed methodology. The database access primitives are equipped with "hooks" to customize their behavior. This mechanism with a job queue supports the following features:

- Database designers can customize the behavior of the primitives operators instead of defining dedicated functions to abstract integrity enforcement.
- Transaction-boundary job execution can be incorporated by the ability to access multiple state values simultaneously and the higher-orderness of the paradigm.

This style is flexible enough to support various database manipulation tasks. Among them, this paper exhibits the support for the type extent model of persistency, the management of mutual references, the specification of exception handlers, and the transaction-boundary execution of integrity checkers.

The organization of this paper is as follows. The next section briefly describes the basic features of Haskell including the I/O model, and the class mechanism, which play an important role in the proposed method of database manipulation. While the main aim of the section is to introduce the language features to readers unfamiliar with them, it also describes the structure and operations of the *part-supplier* database (Atkinson and Buneman 1987) that is a running example throughout this paper.

Section 3 explains the core of the methodology which is based on the state-transformer method. The topic of Section 4 is persistency specification. Since this has a close relationship with type inferencing and the safety of state transformers, the section also discusses how the problem inherent in polymorphic location types occurs in this paper's setting. Section 5 proposes an approach which utilizes multiple versioning to relax the imperativeness of the state-transformer approach.

Section 6 shows the customizability of the primitive operators with "hooks." The idea is not new, but it make it possible for the language to support indispensable tasks like integrity enforcement and materialized view management without compromising the properties of the paradigm.

Section 7 describes related work, and the last section concludes with a brief note on the current prototype system, and future work.

2 Haskell Basics and the Running Example

The aim of this section is two-fold. The first is to provide an overview of a few notable features of Haskell (Hudak et al. 1992) related to the proposed methodology. Details can be found in tutorials such as Davie (1992) for generic topics, and details on the monadic I/O system can be found in Wadler (1992b), Peyton Jones and Wadler (1993), and Gordon (1994). In this paper, all the lines in program fragments are preceded by > signs for clarity[2]. Note that the Haskell specification described in Hudak et al. (1992) is Haskell 1.2, while the latest specification is Haskell 1.4, on which this paper's work is based.

The other aim is to explain the structure and example tasks on the part-supplier database used by Atkinson and Buneman (1987) to compare miscellaneous languages in the context of database management. This example was also used by Gamerman et al. (1992) to compare the object-oriented approach with other existing ones to programming database applications. This section introduces the Haskell features using the volatile version of this example, and the following sections will give the corresponding persistent version.

2.1 Data types, expressions, and bindings

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

[2]Punctuations are also excluded from the program fragments to avoid confusion. "..." is sometimes used to indicate a part which is eliminated from the code to hide cumbersome or implementation-dependent details.

- 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 `[Part]` and the like.

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]
```

In Haskell, a string is a list of characters, thus the type `String` is defined (internally) as follows:

```
> type String = [Char]
```

The data constructor arguments can be labeled to introduce explicit field names. The above algebraic data types, `Part` and `Supplier`, can be alternatively defined with labeled records as follows:

```
> data Part
>   = Basic    { pName::String,   pCost, pMass::Int,
>                pUsedBy::[Part], pSuppliedBy::[Supplier] }
>   | Composite{ pName::String,   pCost, pMass::Int,
>                pUsedBy::[Part], pComposedOf::[(Part, Int)] }
> data Supplier = Supplier{sName, sAddress::String, sSupplies::[Part]}
```

Field names are also used as *selectors* for selecting fields from algebraic data. The name space of field names is the same as that for functions and variables. The above declarations define nine selector functions in total, where `pName`, `pCost`, and `pMass` are shared by `Basic` and `Composite` values.

The pattern matching also handles labeled records. For example, the following function selects names of basic parts from the given list of part values:

```
> basicParts :: [Part] -> [String]
> basicParts parts = [ n | Basic{pName = n} <- parts ]
```

where `::` is read as "has type."

In connection with this example, we note a few aspects of the language. The Haskell type system permits parametric polymorphism (using a traditional Hindley-Milner type structure like ML), extended with *ad hoc* polymorphism, or *overloading* (using *type classes* described later). Thus the principal type or most generic type of an arbitrary expression can be inferred from the type system, and the first line of the above example that declares the type of the function is optional[3]. Nevertheless, we will often include the type annotations of functions in the remainder of this paper, even when the function bodies are also included. This conventional style is followed mainly for readability.

The above definition uses the *list comprehension syntax* (Wadler 1987) in its right-hand side of the definition. A list comprehension abstracts iteration over a list. In general, a list comprehension takes the following form:

[3] *Ad hoc* polymorphism may result in ambiguous typing. We do not enter into this problem for brevity.

```
> massAndCost :: Part -> (Int, Int)
> massAndCost Basic{pCost, pMass}
>   = (pMass, pCost)
> massAndCost Composite{pCost,pMass,pComposedOf}
>   = (pMass + subm, pCost + subc)
>   where
>     submcs
>       = [ (m * q, c * q) | (p, q) <- pComposedOf,
>                            (m, c) <- [ massAndCost p ] ]
>     subm = sum (map fst submcs)
>     subc = sum (map snd submcs)
```

Figure 2: Volatile version of the function to compute the total mass and cost of a part

$$[\ e \mid q_1,\ q_2,\ \cdots,\ q_n\ ],$$

where q_i $(1 \leq i \leq n)$ is either of a generator, $pattern_i$<-e_i, or a guard, $pred_i$. The generator specifies iteration on the list represented by e_i with $pattern_i$ bound for each element of the list. If the element does not match the pattern, the loop "goes" to the next element. $Pred_i$ specifies that a certain loop on a list element is worth continuing. When the predicate evaluates as *false*, then the loop also "goes" to the next element. The following is another example which computes all the pairing from two lists, `xs` and `ys`:

```
[ (x,y) | x <- xs, y <- ys ]
```

The expression `Basic{pName = n}` in the `basicParts` function has two roles: (1) checking if the element of the list `part` is a basic part, and (2) binding the value of the `pName` field to `n`. This pattern matching is often used to tear apart a labeled record without using selector functions. A field name may be used as the variable name to which the field value is bound. Using this shorthand, the above function could have been simpler like this:

```
> basicParts parts = [ pName | Basic{pName} <- parts ]
```

The extended style of pattern matching is also used to define functions. To illustrate, Fig. 2 shows the function that recursively computes the total mass and cost of a part. `fst` and `snd` are library functions that compute the first and second elements of a binary tuple, respectively. `map` and `sum` are also library functions. `sum` computes the sum of a list of numeric values, and `map` applies its first argument to each of the elements of the second argument. The result is the list of the applications.

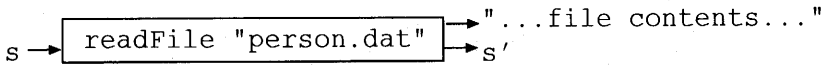
There is a short-hand notation for field updates for labeled records. For example, a function that updates the address of the given supplier can be coded as follows:

```
> updateSupAddr :: Supplier -> String -> Supplier
> updateSupAddr s newAddr = s{sAddr = newAddr}
```

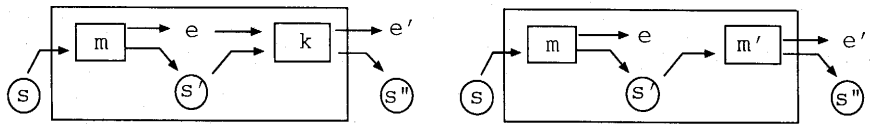
Note that the "update" is *not* destructive, but constructs a *new* value in which all the field values except `sAddr` are the same as those of `s` and the value of `sAddr` is `newAddr`.

2.2 Monadic I/O

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:

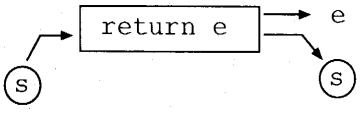


The action takes the I/O state to a pair consisting of the file contents and the new I/O state. From the viewpoint of types, every I/O action returning a value of type `a` is of type `IO a`, where the implementation is hidden from users. For example, the `readFile "person.dat"` is of type `IO String`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



(a) `m >>= k`

(b) `m >> m'`



(c) `return e`

Figure 3: Combinators of state transformer monads

There are three combinators associated with the `IO` monad[4]. Fig. 3 is a diagrammatical representation of these functions. The simplest function `return` constructs a state-transformer from an arbitrary expression, and the `>>=` composes two state-transformers while passing the intermediate result from the first to the second. Another one `>>` simply composes I/O actions. The infix operator `>>=` is of type `IO a -> (a -> IO b) -> IO b` which is equivalent to `IO a -> ((a -> IO b) -> IO b)` as the `->` operator is right associative.

2.3 Classes and overloaded functions

Haskell uses the *class* mechanism to control operator overloading. For mathematical fundamentals, see Wadler (1989) and Jones (1994b). A class is a family of algebraic types associated with overloaded operators. A class member is called an *instance*, and the operators are called *class 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 two 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`" ("`PrimEqFloat x y`") if `x` and `y` have type "`Int`" ("`Float`").

2.4 Constructor classes and do-notation

The concept of constructor classes was originally proposed by Jones (1994b) to incorporate the monad concept (Moggi 1989; Wadler 1992a) into Gofer, a language akin to Haskell. Instances of a constructor class are general data type constructors instead of (null-ary) data types. The previous section explains

[4]The constructor class will be described in Section 2.4.

>>=, >>, and return as if they were I/O specific operators. However, these are the operators of the Monad class, declared as follows:

```
> class Monad m where
>    return :: a -> m a
>    (>>=)  :: m a -> (a -> m b) -> m b
>    (>>)   :: m a ->       m b  -> m b
>    m >> m' = m >>= \_ -> m'
```

The list type constructor, [], is also an instance of the Monad class, thus the above definition using the list comprehension syntax is equivalent to the following definition in terms of the list type:

```
> basicParts :: Monad m => m a -> m a
> basicParts parts = parts >>= \Basic{pName} ->
>                    return pName
```

The context "Monad m" requires that this function should be used with any of the Monad instances.

The companion of the Monad class is the *do-notation*, which is the generalization of the list comprehension syntax. For example, the above definition is also defined using the notation as follows:

```
> basicParts parts = do Basic{pName} <- parts
>                       return pName
```

The do-notation abstracts computation regarding Monad, while the list comprehension syntax abstracts computations of only lists. The following are the simplified translation rules from do-expression to the corresponding one using the monad combinators:

$$
\begin{array}{lcl}
\textbf{do}\ \{e\} & = & e \\
\textbf{do}\ \{e;stmts\} & = & e >> \textbf{do}\ \{stmts\} \\
\textbf{do}\ \{p <- e;\ stmts\} & = & e >>= \backslash p -> \textbf{do}\ \{stmts\} \\
\textbf{do}\ \{\textbf{let}\ decllist;\ stmts\} & = & \textbf{let}\ decllist\ \textbf{in do}\ \{stmts\}.
\end{array}
$$

To improve the readability, the remainder of this paper will use the do-notation and the monad combinators only for the database and the I/O monads, and will use the list-comprehension syntax for lists.

3 Database State Monad

To prevent purity from being disrupted, the proposed methodology utilizes a monad of state transformers to perform referentially transparent update operations. As the I/O mechanism of Haskell is based on the state-transformer monad for the I/O state type, we can introduce another state-transformer monad for the database state. Theoretically, there is no side effect, since the transformers generate a *new* database state value. In practice, on the other hand, state-based operations can be implemented by side effects on the internal state, since the database operations are executed linearly as in an imperative programming language.

We shall mention the non-triviality of this approach here to avoid confusion. An arbitrary state type defines its state-transformer monad. This ensures the generality of the state-transformer approach, but this fact also implies that every property of a monad is determined by the structure of the state type. Therefore, we need to design the state type so that it meets our requirements: destructive updatability, type safety, lazy retrieval, customizable primitives and so on.

The database monad comprises the database state type, primitive operators, state-transformer combinators, and a transaction model. The database state in turn comprises two parts:

- A collection of entities that are represented by introducing surrogates to model their identities and mutability; and
- A collection of persistent roots that gives the access points of the stored database state.

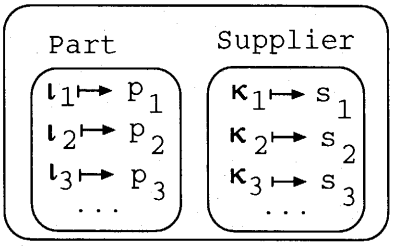


Figure 4: Entity part of a database state

This section describes only the first part of the state, while the next section will describe the second part in conjunction with the safety.

The primitive operators are associated with the `Entity` class (Sections 2.3 and 2.4), and the entity types are declared by making them the instances of the class. Through this *ad hoc* polymorphism, entity types are discriminated from others. Hence, user-defined polymorphic database operations can be built without difficulty, and the compilation system can prevent programmers from applying database operations to irrelevant types[5]. Although polymorphic *types* may also be instances of the `Entity` class, polymorphic *values* cannot be permanently included in the database state to avoid a problem inherent of mutable locations described in the next section. For example, `[a]` may be specified as an instance of the `Entity` class, but only fully instantiated values such as values of type `[Int]` and `[String]` can be stored. Therefore, throughout this paper "entity types" are used to denote such types of storable values instead of "instances of the `Entity` class."

3.1 Database state and entities

Let Σ be the set of all the entity types in a database, and let $Ref(\sigma)$ and $Val(\sigma)$, respectively, be the sets of all the surrogates and values of type σ ($\in \Sigma$). Then the entity-related part of the database state is a collection of Σ-indexed maps s_σ. Note that $Val(\sigma)$ is a set of values of Haskell type σ. A value in $V(\sigma)$ may contain a value of type $Ref(\sigma)$ which refers to other entities directly through pointers and/or indirectly through entity surrogates. The primitive operations are defined as follows. Let $o \mapsto v$ be a binary association from o to v. Then the operational part comprises three operators:

$read_\sigma(o)$	retrieve $o \mapsto v$ from s_σ;
$write_\sigma(o \mapsto v)$	replace $o \mapsto w$ in s_σ with $o \mapsto v$; and
$new_\sigma(v)$	insert $o_\sigma \mapsto v$ into s_σ for a new o_σ.

Note that due to the restriction applied to the persistent roots types, Σ is usually a finite set of ground types.

To illustrate, consider the Part-Supplier database explained in Section 2. The objects were represented by algebraic data types, or sum-of-product types, declared as follows:

```
> data Part
>    = Basic     { pName::String,   pCost, pMass::Int,
>                  pUsedBy::[Part], pSuppliedBy::[Supplier]}
>    | Composite{ pName::String,   pCost, pMass::Int,
>                  pUsedBy::[Part], pComposedOf::[(Part, Int)]}
> data Supplier
>   = Supplier{sName, sAddress::String, sSupplies::[Part]}
```

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 should be modified as such.

[5]Since a database entity is a kind of mutable location, we could use it to implement the *undo-able* mutable variables to reduce the computational order of algorithms. This issue, however, is out of the scope of this paper.

```
> data Part
>   = Basic    { pName::String,          pCost, pMass::Int,
>                pUsedBy::[DBRef Part], pSuppliedBy::[DBRef Supplier]}
>   | Composite{ pName::String,          pCost, pMass::Int,
>                pUsedBy::[DBRef Part], pComposedOf::[(DBRef Part, Int)]}
> instance Entity Part
>
> data Supplier
>   = Supplier{sName, sAddress::String, sSupplies::[Part]}
> instance Entity Supplier
```

Here "DBRef α" is the Haskell representation of $Ref(\alpha)$, and Entity is the class to designate database types. Fig. 4 depicts an abstract view of the state of the Part-Supplier database where ι_i $(i = 1, 2, 3, ...)$ are the surrogates for stored parts and κ_j $(j = 1, 2, 3, ...)$ are the surrogates for stored suppliers.

3.2 State transformers and basic combinators

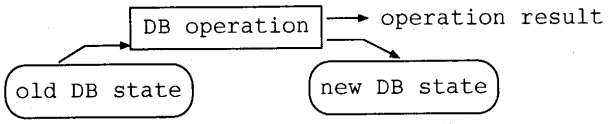


Figure 5: Database operation as a state transformer.

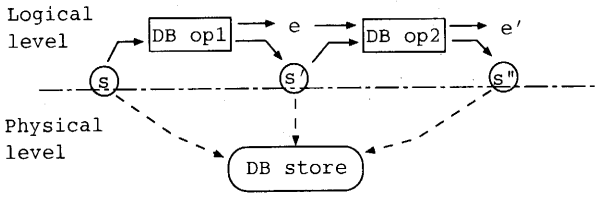


Figure 6: Imperative manipulation of database state.

Provided that the database state values are of type DBState, the type and the monad of database state-transformers may be defined as follows:

```
> data DB a = DB ( DBState -> (a, DBState) )
>
> instance Monad DB where
>    DB m >>= k  = DB (\d -> let (x,d') = m d
>                                DB m'  = k x
>                            in m' d')
>    return x    = DB (\d -> (x, d))
```

where \x -> e represents a lambda abstraction $\lambda x\,.\,e$. These correspond to IO a, >>=, and return for the I/O state-transformer monad, respectively. Complicated implementation details are hidden from users, but the above simplified and explicit definitions suffice to show the skeleton of the state-transformer monad. The diagrammatic representation of a database state-transformer is shown in Fig. 5. The combinators can be as shown in Fig. 3, even though the definitions of the combinators are different from those of the I/O monad.

In these figures, database operations are drawn assuming that they construct a new database state value for every step. This is required to ensure the referential transparency. To improve the performance of update operations, however, the database store should be updated destructively. Therefore, a more desirable schematic drawing is shown in Fig. 6. Operations are referentially transparent at the logical level, while they may be referentially opaque at the physical level. To make them fully transparent, the

state transition should be *hyper-strict*. In terms of Fig. 6, this requirement means that when state s' is constructed, the intermediate expression e must have been evaluated so that its subexpressions depending on the the modifiable part of the previous state s are fully evaluated. This restriction is not difficult to enforce, because we only have to ensure it for the built-in primitive operators.

3.3 Primitive operations and transactions

The primitive operators are declared in Haskell as follows:

```
>   readDB  :: Entity a => DBRef a       -> DB a
>   writeDB :: Entity a => DBRef a -> a  -> DB ()
>   newDB   :: Entity a => a             -> DB (DBRef a)
```

"`Entity a =>`" specifies the constraint that when these functions are used in a more specific typing context, the variable `a` must be replaced with an instance of the `Entity` class.

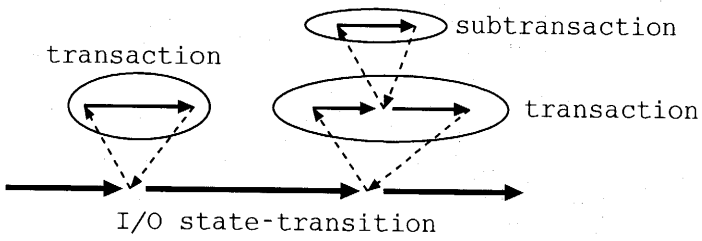


Figure 7: Interaction between the I/O world and the database world; the solid arrows at the bottom depict the state-transition sequence in the I/O world, and the other solid ones correspond to database state-transitions.

Every transaction expression is of type `DB` τ, where τ is the type of the result. This expression is taken to the I/O world by the `transaction` function for execution, or is wrapped up in the `subtransaction` function to be a nested transaction of another one. These functions are typed as follows:

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

The diagrammatical representation of these functions is exhibited in Fig. 7. The single-threadedness of I/O operations also ensures that of the database transactions. The modification by a transaction is committed at the end of it by default, whenever all the included database operations are processed successfully.

3.4 Examples

To illustrate the usage of the above database primitives, we show a few examples. The first one is a simple query: retrieve basic parts that cost more than $100. Provided that all the surrogates of parts are held in a list bound to `parts`, the query can be coded using the following state transformer:

```
>    mapM readDB parts >>= \partValues ->
>    return [ pName | Basic{pName,pCost} <- partValues, pCost > 100 ]
```

where `_` is a wild-card pattern or an anonymous variable. `mapM` is one of the library functions regarding monads. It applies the first argument to the elements of the second argument, and gathers the application results into a list.

As mentioned in the previous subsection, database transactions are "enabled" only after they are taken to the I/O world through the `transaction` function. Hence, the above database query may be executed like this:

```
> massAndCost :: Part -> DB (Int, Int)
> massAndCost Basic{pCost,pMass}
>   = (pMass, pCost)
> massAndCost Composite{pCost,pMass,pComposedOf}
>   = do submcs <- computeSubmcs
>        let subm = sum (map fst submcs)
>            subc = sum (map snd submcs)
>        return (pMass + summ, pCost + sumc)
>   where
>     computeSubmcs
>       = accumulate [ do part   <- readDB p
>                         (m, c) <- massAndCost part
>                         return (m * q, c * q)
>                    | (p, q) <- pComposedOf ]
```

Figure 8: "Imperative" query function to compute the total mass and cost of a part

```
> selectExpensiveParts parts =
>    putStr "Basic parts that cost more than 100\n" >>
>    transaction (
>      mapM readDB parts >>= \partValues ->
>      return [ pName | Basic{pName,pCost} <- partValues, pCost > 100 ]
>        ) >>= \names ->
>    putStr (lines names)
```

where `lines` is a predefined function which, given a list of strings, concatenates the strings with a newline character appended to every element of the list, and `putStr` is also a predefined I/O function to construct an I/O action that prints the given string on the terminal screen.

The next example updates the addresses of suppliers named "SUP1000" with `newAddr` defined elsewhere, provided that all the surrogates of the stored suppliers can be accessed by the variable `suppliers`:

```
>  transaction (
>    mapM readPairDB suppliers >>= \supPairs ->
>    sequence [ writeDB sid sval{sAddr=newAddr}
>                 | (sid, sval@Supplier{sName}) <- supPairs,
>                   sName == "SUP1000" ]
```

where `readPairDB` is a function defined as

```
> readPairDB s = readDB >>= \v -> return (s, v)
```

Hence "`mapM readPairsDB`" is a function that converts a list of surrogates into a list of surrogate-value pairs. The list comprehension "`[ writeDB .... | ...]`" constructs a list of database actions to perform the required update operations. These actions are combined by the built-in function, `sequence`. The combined action executes the update actions in the order given in the list. Since the update operations are performed strictly, the database store can be updated destructively so that no copy is generated.

Consider a slightly more complicated query: retrieve the total mass and cost of a composite part. The volatile version has already been shown in Fig. 2. The persistent version can be written as shown in Fig. 8, but the query expression is more complicated than the volatile one. Note that the utility function `accumulate` combines the given list of monad actions into a bigger action that returns the list of the results. The imperativeness stems from the difference between the structural data access through pattern matching and list comprehension, and the sequential execution of database operators. A more readable definition will be shown later using on-the-fly dereferencing.

4 Models of Persistence

There are two models of persistence specification: *type extent* and *reachability* models. The proposed approach follows the latter one basically, and supports the former by the help of triggering mechanism described in the next section. Before discussing the details, we briefly overview features of these two models.

In the type extent model of persistence, every database type is associated with a persistent set of entities, or *extent*, of that type. The type extent is maintained automatically by the underlying storage manager: whenever a new entity is created, it is stored in the corresponding type extent. When this model is used to specify persistency, another primitive operation for deletion, say `delDB`, is necessary. Without a certain mechanism to enforce referential integrity, however, a delete operation may result in dangling references that refer to a stale entity.

On the other hand, in the reachability model of persistence, programmers explicitly maintain persistent roots. Values reachable from persistent roots through direct pointers or indirect surrogate references are considered to be persistent. The notable advantage of the reachability model is that it is more flexible and can simulate most of the operations for the type extent model with some additional programming cost. Besides, references dangle less often since entities are deleted only when there is no path from any of the persistent roots. The typical disadvantage of such an approach is its complexity in deleting entities, that is, all the references to the entity must be modified. If some of the paths should fail to be modified, the database might include an entity that refers to the entity to be deleted. Even though such references never dangle physically, they should be considered to dangle logically since they refer to an obsolete entity. Note that, in the proposed approach, the situation is not so bad, since a database can be updated on a "surrogate-value pair" basis.

At a more abstract level, the persistent roots are just associations from root identifiers to their corresponding root values. The identifiers may be symbols as in Napier88 (Dearle et al. 1989) or strings as in Staple (McNally and Davie 1991). These two approaches, however, do not comply with the purity rule or the static typing rule. Indeed, symbol values cannot be modified without sacrificing purity, and string identifiers require dynamic typing since there is no clue to infer the root types in arbitrary string identifiers. In the approach proposed here, the types of persistent roots are used as root identifiers, and selection of a certain root is implemented naturally through overloaded access functions.

The rest of this section focuses on the support of the reachability model, and also addresses the problem in persistent roots of polymorphic types. Due to this problem, we require every root value to be of ground type. This indirectly requires every permanent values including entities to be of ground type, guaranteeing the safety of state capturing and on-the-fly dereference introduced in the next section.

4.1 Persistent root declaration

persistent roots are specified using the `PerRoot` class as the `Entity` class is used to define entity types. Provided that two persistent roots for parts and suppliers are maintained for the part-supplier database, the following declarations suffice:

```
> data PartExt = PartExt{parts::[DBRef Part]}
> instance PerRoot PartExt where
>     initValue _ = PartExt{parts=[]}
>
> data SuppExt = SuppExt{suppliers::[DBRef Supplier]}
> instance PerRoot SuppExt where
>     initValue _ = SuppExt{suppliers=[]}
```

The first and the fifth lines declare the root types for parts and suppliers, respectively. The rest of the code fragment declares `PerRoot` instances. The above instance declarations include the specification of initial values in the third and seventh lines. In these cases, they are composed of an empty list. Notice that there is one parameter for the `initValue` method. The parameter is used to define non-materialized views which will be explained in the next section. Throughout this section, these values are always "unused."

For each `PerRoot` instance, the underlying storage manager maintains the value of that type. The values of persistency roots are read and written via two overloaded functions:

```
> readRootDB  :: PerRoot a =>      DB a
> writeRootDB :: PerRoot a => a -> DB ()
```

Fig. 9 diagrammatically represents the behavior of these functions. Since `readRootDB` and `writeRootDB` are overloaded, they have different implementations per instance. In the figure, *readRootDB*$[\tau_i]$ denotes a certain implementation of `readRootDB` for type τ_i, and so does *writeRootDB*$[\tau_i]$. An appropriate implementation is selected automatically through the class mechanism of the Haskell programming language. A context of the root access functions determines the type of the functions, so the type itself determines the corresponding root value location.

4.2 Ground type restriction

The Haskell language does not prevent a polymorphic type from being an instance of the `PerRoot` class. For example, `[a]` and `a -> b` may be instances of `PerRoot`. However, supporting a polymorphic root location causes a subtle typing problem that is similar to the one regarding *references* in ML. Connor et al. (1991) also has pointed out that the same trouble arises when subtyping is taken into account. Provided, for example, that `a -> b` be an instance of `PerRoot`, the type be associated with its unique location, and a function `incI` be declared as follows:

```
> incI :: Int -> Int
> incI x = x + 1
```

Then, the following expression would be correctly typed by the type checker: [6]:

```
>       writeRootDB incI >>
>       readRootDB       >>= \f ->
>       return (f (2::Float))
```

Note that `>>=` and `>>` are right-associative with the same priority, and that their priority value is lower than that of lambda abstractions denoted by `->` which are also right-associative. Evaluating this expression may lead to a run-time error. In this example, `writeRootDB` stores `incI` for the root associated with $\forall\alpha.\forall\beta.\alpha \to \beta$, and the stored value is retrieved and used later as a function of type $\forall\alpha.\text{Float} \to \alpha$. The result is not predictable, since the implementation of `Int` is different from that of `Float` in general. The most plausible result is a "segmentation fault" error.

Haskell allows for instance declarations only in the following form:

$$C\ (T\ a_1\ a_1\ \cdots\ a_n),$$

where C is a class name, T is a type constructor name of arity n, and a_i $(1 \leq i \leq n)$ are mutually distinct type variables. Programmers may use any specialized forms of the instance type, but the type checker does not discriminate them. As we associate a location with every instance instead of with every specialized form of the instance in the above discussion, the location was shared by all the specialized types. This sharing caused the above problem.

We adopt a simple solution that distinguishes locations of persistent roots at their finest level: every one of them should be associated with a ground specialization of the declared instance type. This section first clarifies the issue in a more formal setting using the technique of Connor et al. (1991), and then mentions the related issues found in ML's references, and in the mutable variable extension of Haskell by Launchbury and Peyton Jones (1994). Then we show how the restriction is enforced by the slightly modified declaration of the `PerRoot` instance.

More formal view of the problem

The above problem can be clearly seen using the denotational description of storage and the safety condition proposed by Connor et al. (1991). For any location denoted by i, written $loc(i)$, three kinds of types are attributed:

[6] $e :: \tau$ is an expression that explicitly specifies that e is of type τ.

- The creation type, written $T_{creation}(loc(i))$;
- The right-hand or stored value type, written $T_{r\text{-}value}(loc(i))$; and
- The view types, $T_{view(j)}(loc(i))$, for every expression, j, associated with the location $loc(i)$.

Note that in Connor et al. (1991), $T_{r\text{-}value}(loc(i))$ is called *r-value minimum type*, since record-based subtyping or inclusion polymorphism is treated as the major topic in it. However, we treat parametric typing, and the most general type of a value (inferred from its surrounding context) is assumed unless otherwise explicitly noted. Connor et al. (1991) proposed the following invariant to evaluate the accuracy of static type descriptions:

$$\forall i.\forall j.T_{view(j)}(loc(i)) \leq T_{r\text{-}value}(loc(i)), \tag{1}$$

where $\leq$ denotes the parametric subtyping relation; $\tau_1 \leq \tau_2$ holds between the types iff there is a type variable specialization and/or substitution θ such that $\tau_1 = \theta(\tau_2)$. Intuitively, the stored value must be used as-is or in its more specific form. Note again that in Connor et al. (1991) $\leq$ denotes record-based subtyping. Thus, we have exchanged the right- and left-hand sides of the inequality.

Let us view the problem described above using this denotational semantic model of locations. For every `PerRoot` instance τ, the associated storage, say $loc(\tau)$, is of type τ. In the above example, τ was $\forall\alpha.\forall\beta.\alpha \to \beta$. Because of the specialization rule of typing, `readRootDB` may be of type $\theta(\tau)$, where θ is a valid specialization of τ. In the above example, this specialized type is $\forall\alpha.\texttt{Float} \to \alpha$. This means that for every view of $loc(\tau)$ the following holds:

$$\forall\tau.\forall\tau'.(\tau' \leq \tau \Longrightarrow (T_{view(readRootDB[\tau'])}(loc(\tau)) \leq T_{creation}(loc(\tau)))), \tag{2}$$

where $\Longrightarrow$ represents "implication". Since the root value is read only through this operator, the above inequality implies that

$$\forall\tau.\forall j.T_{view(j)}(loc(\tau)) \leq T_{creation}(loc(\tau)). \tag{3}$$

In addition, a similar discussion gives another inequality:

$$\forall\tau.\forall e :: \tau'.(\tau' \leq \tau \Longrightarrow (T_{view(writeRootDB[\tau']\,e)}(loc(\tau)) \leq T_{creation}(loc(\tau)))). \tag{4}$$

In the above example, τ' is `Int` $\to$ `Int`. The value stored in the location is modified only through `writeRootDB`, so this inequality implies

$$\forall\tau.T_{r\text{-}value}(loc(\tau)) \leq T_{creation}(loc(\tau)). \tag{5}$$

Finally, the required inequality (1) does not necessarily hold. Indeed, this has already been shown by the counter example above.

We shall mention that the above problem does not arise from the inherent properties of the Haskell type class system. The Haskell language itself does not have the *mutable-location* concept.

Related issues and techniques

There are two known methods for avoiding the location problem in statically typed functional programming languages. ML avoids it by using special type variables called *weak type variables*. Type variables appearing in a location type are weak[7]. Unlike usual type variables, weak type variables are not candidates for generalization (or $\forall$ quantification). If this restriction had been applied to our case, the two inequalities (2) and (4) would have been replaced by the following equalities:

$$\forall\tau'.T_{view(readRootDB[\tau'])}(loc(\tau')) = T_{creation}(loc(\tau')) \tag{6}$$

$$\forall\tau'.T_{view(writeRootDB[\tau']\,e)}(loc(\tau')) = T_{creation}(loc(\tau')), \tag{7}$$

which in combination would have led to the desired equality

$$\forall\tau'.\forall j.T_{view(j)}(loc(\tau')) = T_{r\text{-}value}(loc(\tau')). \tag{8}$$

[7] According to the ML terminology, a location is called a reference, and is associated with three operators, `ref`, `!`, and `:=` for creation, reading, and assignment, respectively.

Notice that we use τ' instead of τ, because the weakness of the type variables requires that every creation type for a location coincide with its usage.

The mutable variable extension of Haskell (Launchbury and Peyton Jones 1994) also adopts the same idea, but implements it by using the monomorphic typing of lambda-bound variables and assigning a special type for the *wrapper*. Although the monomorphic typing is not directly relevant to our current problem, we address it here because it clearly explains why the above problem never occurs in relation to entity locations. Let a polymorphic type $\forall\alpha.\forall\beta.\alpha \to \beta$ be an instance of the `Entity` class:

```
> instance Entity (a -> b)
```

Then the following expression does not incur the problem described above, even though it resembles the above root access manipulation code:

```
> createVar :: DB (DBRef (a -> b))
> createVar = newDB (\x -> bottom)
>
> action = createVar      >>= \v ->
>          writeDB v incI >>
>          readDB  v      >>= \f ->
>          return (f (2::Int))
```

Before discussing further, we shall note the following properties:

1. `createVar` does not create a new location by itself. A new location is created only in a certain sequence of the database state transitions, which in turn is executed only in a certain sequence of the I/O state transitions.

2. Lambda-bound variables are always of monomorphic types according to the Hindley-Milner typing rule.

The first property ensures that *a newly created entity location is accessed only through lambda-bound variables*, as shown in the above example, and the second point ensures that *the type of the location is under the monomorphic typing discipline.* These properties ensure the same typing restriction as was found in ML's weak type variables:

$$\forall\tau'.T_{view(readDB[\tau'])}(loc(\tau')) \;=\; T_{creation}(loc(\tau')) \tag{9}$$

$$\forall\tau'.T_{view(writeDB[\tau']\,e)}(loc(\tau')) \;=\; T_{creation}(loc(\tau')). \tag{10}$$

Provided that only entity locations are considered, these equalities in combination imply the desired equality as follows:

$$\forall\tau'.\forall j.T_{view(j)}(loc(\tau')) \;=\; T_{r\text{-}value}(loc(\tau')). \tag{11}$$

In the above example, the τ' is `Int` $\to$ `Int` throughout the creation, reading, and writing processes, even though the initial value of the location is `\x -> bottom` whose principal type is $\forall\alpha.\forall\beta.\alpha \to \beta$. In other words, this lambda abstraction is treated as if it were of type `Int` $\to$ `Int` at the creation time.

Now consider another slightly modified example:

```
> action = createVar      >>= \v ->
>          writeDB v incI >>
>          readDB  v      >>= \f ->
>          return (f (2::Float))
```

Using this function incurs a *typing* error instead of a *run-time* error. As noted in the second property above, the location type carried by `v` and `f` must be unifiable without generalization. This requires that `Int` $\to$ `Int` be unified with `Float` $\to \beta$, thus a typing error occurs as expected.

Lastly, we shall point out that there is *no* "wrapper" for the database monad. In general, a state-transformer monad abstracts *computation processes*, while an ordinary lambda term abstracts *values* to be computed. The difference also requires that a state-transformer monad must be facilitated with an

evaluator which executes the abstracted computation process. Launchbury and Peyton Jones (1994) facilitate their lazy state-transformer monad with `runST`, which executes or "wraps" a state-transformer to produce the result value:

$$\texttt{runST}:: \forall a.(\forall s.\texttt{ST}\ s\ a) \rightarrow a.$$

We might have facilitated the database monad with an executor like

$$\texttt{runDB}:: \forall a.\texttt{DB}\ a \rightarrow a.$$

However, this gives rise to two problems. The first problem is the one that we have been discussed so far. If we allow a programmer to use the wrapper, (s)he could *name* the entity location with its polymorphic type like this:

```
> createVar' :: DBRef (a -> b)
> createVar' = runDB (newDB (\x -> bottom))
```

Since this function would assign a polymorphic type to the location, we would face the same problem as discussed above. Another problem is loss of linearity of the database state thread. To ensure the *well-definedness* of the database state, we must also ensure the linearity of all the related actions. An example of non-linear expression would be:

```
>     let v = createVar'
>     in (runDB (writeDB v incI), runDB (readDB v))
```

The result depends on the order of evaluation of the tuple elements. Because of the first property of the database monad and the fact that the I/O monad does not have a wrapper either, we can ensure the linearity of the database monad. Note that the technique proposed by Launchbury and Peyton Jones (1994) avoids this illegal "capture" of locations by assigning a special type to `runST`[8]. Briefly speaking, their technique never allows a location to be used outside the local state-transition sequence where the location was created.

4.3 Avoiding illegal root manipulation

The second technique is valid only if locations are created and used in a lambda abstraction. On the other hand, the locations of persistent roots are implicitly generated *in advance*. Our proposed solution thus imposes a restriction that stored root values should be of ground type. Note that this does not necessarily prevent database designers from using polymorphic root instances. Instead, even though a polymorphic type is declared as an instance of `PerRoot`, only its ground specializations are treated as the types of the stored roots locations.

Following this setting, every branch shown in Fig. 9 has one or more of sub-branches according to the ground specialization of instance type (Fig. 10). *writeRootDB*$[\tau_i]$ denotes the branch of `writeRootDB` for τ_i that is an instance of the `PerRoot` class. *writeRootDB*$[\tau_{ij}]$ is a one-step-further refinement of *writeRootDB*$[\tau_i]$ where $\tau_{ij} = \theta(\tau_i)$ for some ground specialization, θ. The appropriate implementation method is automatically selected by the class mechanism of Haskell.

This restriction requires that the `PerRoot` class is declared in a slightly different way:

```
> class Ground a => PerRoot a where ...
```

where "`Ground a`" gives the context of this class declaration, and requires that only if a is an instance of `Ground`, can it be an instance of `PerRoot`. This implies that `readRootDB` and `writeRootDB` implicitly require their related types be instances of `Ground`. Therefore, an appropriate sub-branch in Fig. 10 is selected automatically.

It is trivial to see that this restriction is sufficient to ensure inequality (1). Indeed, since all locations are of ground types, inequalities (2) and (4) are reduced to simple equalities respectively as follows:

$$\forall\tau'.T_{view(readRootDB[\tau'])}(loc(\tau')) = T_{creation}(loc(\tau')) \tag{12}$$

[8]The type is not a Hindley-Milner type, because the quantifiers are not all at the top level.

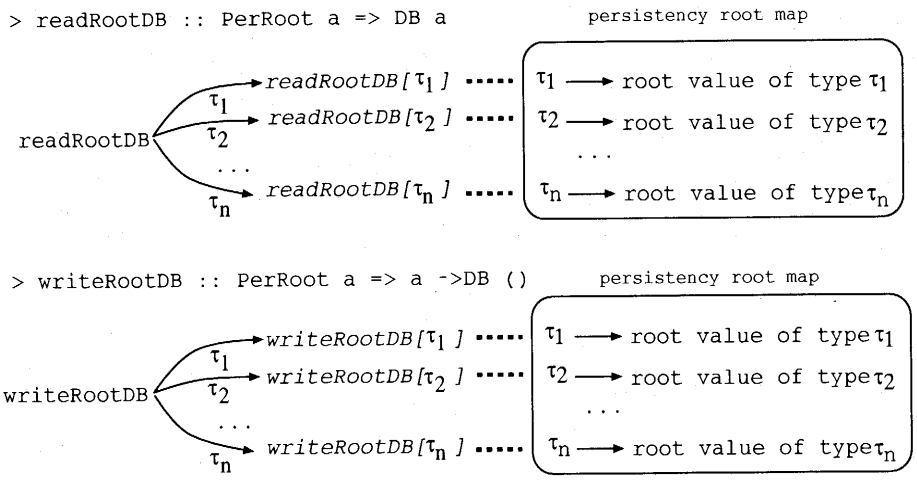


Figure 9: Behavior of `readRootDB` and `writeRootDB`

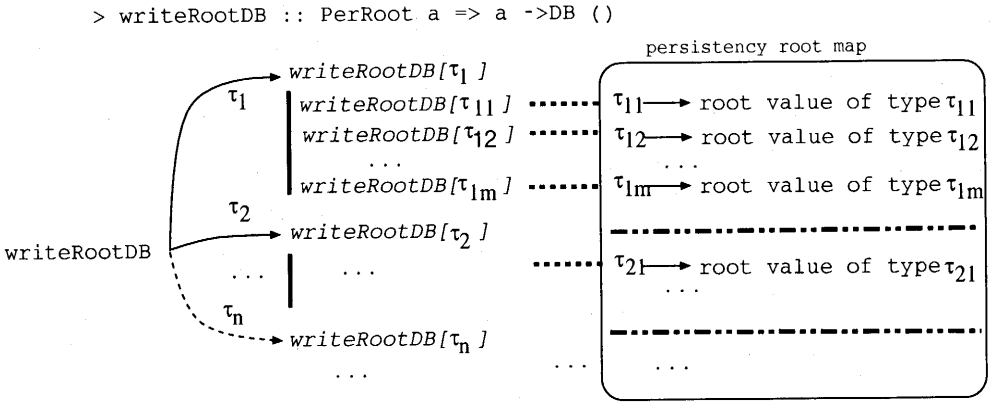


Figure 10: Behavior of `writeRootDB`

$$\forall \tau'.\forall e :: \tau'.T_{view(writeRootDB[\tau']\, e)}(loc(\tau')) \;=\; T_{creation}(loc(\tau')), \tag{13}$$

which in combination imply the desired equality

$$\forall \tau'.T_{view(\tau')}(loc(\tau')) \;=\; T_{r\text{-}value}(loc(\tau')). \tag{14}$$

We also use τ' instead of τ to indicate that the type is restricted to being a ground specialization of a certain root type τ.

Although how to enforce this restriction is an implementation issue, we describe the basic strategy here to make the above intuitive explanation more concrete. Instances of `Ground` are categorized into two groups. The first group simply includes ground types like `Bool`, `Char`, and `Int`:

```
> instance Ground Bool
> instance Ground Char
> instance Ground Int
```

The other group comprises (algebraic) data type constructors with arity of more than one. Remember that the Haskell class mechanism allows an instance declaration to have its specific context. For instance, the `Ground-[a]` instantiation is declared like

```
> instance Ground a => Ground [a]
```

where `Ground a` is the context of this instance declaration, which is to say that iff a type α is an instance of `Ground`, then so is the type $[\alpha]$.

The first group gives the base cases of the structural induction, and the second one gives the induction step. Henceforth, it suffices to ensure that all the predefined and user-defined algebraic data types are

correctly made instances of the `Ground` class. Note also that it is impossible for users to give their own instance declaration for the class: Haskell prohibits overwrapped instance declarations[9], and allows only instances in their most general form[10]. Any attempt by users to declare their own `Ground` instances invokes an error during compilation.

The above restriction requires that a user to be careful in declaring the persistent root types. For example, a user can define $\forall\alpha$.`Bag` α to be an instance of the `PerRoot` class, but they must now declare it as follows:

```
> data Bag a = Bag [a]
>
> instance Ground a => PerRoot (Bag a)
```

As pointed out by Jones (1994b), this may infer a confusing context. To illustrate, consider a function defined like this:

```
> incBag x = readRootDB >>= \(Bag s) ->
>            writeRootDB (Bag (x:s))
```

This is a transaction that adds an element to a persistent root containing a "bag" value. Because of the restriction described above, the inferred type of this function is

```
> incBag ::  Ground a => a -> DB ()
```

instead of the more intuitive declaration:

```
> incBag ::  PerRoot (Bag a) => a -> DB ()
```

This is an inherent property of the Haskell language, and we cannot avoid it.

Finally, we must mention that the `Ground` instantiation does not require any special technique for its implementation. To support the language specification, every Haskell processor has already been equipped with an automatic instantiation mechanism in a particular compilation phase. Modifying this phase so that the above restriction is automatically generated for all algebraic data types is straightforward[11].

5 Lazy Retrieval and Imperative Update

As pointed out in Section 3.4, even complicated recursive functions must resort to step-by-step, or imperative, execution in the core of the monadic database manipulation. The source of this imperativeness is the lack of one-the-fly dereferencing from a surrogate. This complication was necessary for the referential transparency not to be compromised, but it sacrificed the declarativeness and terseness of Haskell programs.

This issue is closely related to the parallel nature of Haskell. Because of its declarativeness, the language naturally exhibits parallelism in execution. For instance, the two subexpressions in e_1 + e_2 may be evaluated in three different orders: e_1 then e_2, e_2 then e_1, or e_1 and e_2 in parallel. The third, parallel execution, does not affect the result of execution because of the referential transparency. Moreover, in non-strict programming languages, expressions are evaluated on demand. Evaluating an expression constructs the datum that represents the computation process instead of the one that represents the result of computation. This computation datum is often called a *closure*. Thus, expressions of non-strict functional languages are evaluated by two interleaving steps: (1) constructing a closure, and (2) performing (or reducing) a closure. Even though the evaluation is performed in a single-CPU system, the underlying evaluator must include the scheduler and the evaluator of closures.

Since Haskell is parallel in nature as explained just above, we must control the execution of closures at least so that the on-the-fly dereference through surrogates does not break the referential transparency of

[9] Two instance declarations, $C\ \tau_1$ and $C\ \tau_2$, overwrap, if τ_1 and τ_2 are unifiable.
[10] Instance declaration $C\ (T\ a_1\ \cdots\ a_n)$ requires all the type expressions, a_i $(1 \leq i \leq n)$, to be distinct type variables.
[11] Exceptions are constructor variables.

the programming system. Although simply implying a certain order of evaluation might solve the issue, it would depend on the particular system implementation and hence reduce the portability of the system. In this section, therefore, we pursue a method that is easy to port and does not affect the language specifications of Haskell.

The basic idea is the same as the "notorious" *state capture* which may disrupt the linearity of a state-transformer monad by introducing a function like this[12]:

```
getState = DB (\s -> (s, s)).
```

To ensure the linearity, we have two choices:

1. To invalidate the state capture operation, or
2. To duplicate the state value.

Thus this issue is reduced to a transaction control issue among one *writer* and multiple *readers*. The first choice implies that the writer gets an exclusive lock on the state to ensure linearity. The other choice leads to the mechanism called *multiple versions concurrency control* (Bernstein et al. 1987). In this mechanism, a transaction manager keeps track of modification histories of data items. When a data item is modified, the history is updated so that the old value may be retrieved later. Even when a transaction issues a read request for an older value, the request is successfully processed if an appropriate old value is found in the modification history. The technique proposed in the following sections uses an idea similar to this transaction mechanism, but uses explicit version-controlling primitives to make it easier for programmers to handle multiple versions simultaneously. Every transaction is a *writer* that may modify the database state, and expressions that retrieve data from the captured state values are *readers*. All the *writers* are linearly ordered, but the *readers* can access captured state values on-the-fly.

5.1 State capture

Fig. 11 shows a conceptual view of of a database state history. The companion figure (Fig. 12) shows the same aspect of database modification from the viewpoint of an object. An object may have more than one values corresponding to its modification history. In Fig. 11 (a), the number i represents a version generated at time t_i. Fig. 11 (b) shows the same version history plotted in the time domain. Similar plotting can be used to exhibit an object history (Fig. 12). The object is created at t_1. The values of the object are modified at t_3 and t_5, respectively with v_3 and v_5. The values at other points are inherited according to the version generation history. For instance, the object value is v_5 at t_6, and v_1 at t_7. Dereferencing the value of the object at t_0 incurs an error. Remember that while objects are "born" explicitly, they "die" implicitly when they become garbage.

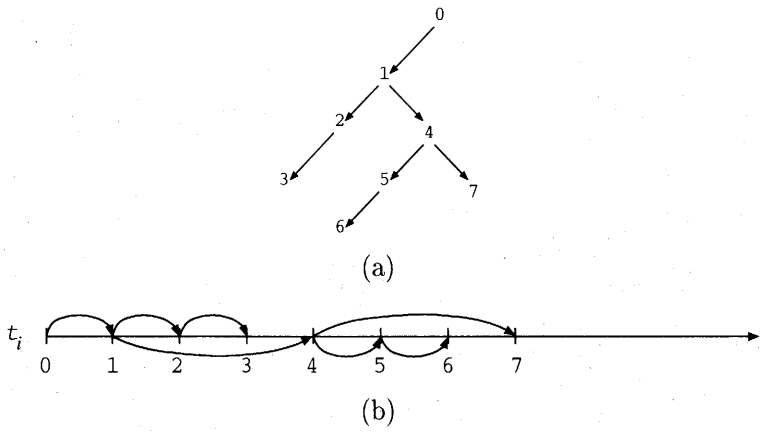


Figure 11: Tree form history of versions: (a) shows the history, and (b) shows an alternative view of the version in the time domain.

At any point of a database state-transformer execution, the current state can be captured by the action `getDB`. Provided that `Database` is the abstract type that represents the captured database state,

[12]Since the database state-transformer monad is defined by an abstract data type, users are prohibited to define this functions.

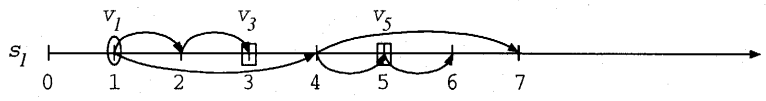


Figure 12: History of an object: The circle designates the birth of the object, and two squares are put on the points where the object is modified. The values at the non-marked places after the birth are inherited according to the version history.

the operation is typed as follows:

```
> getDB :: DB Database
```

Since `getDB` implies the existence of readers, it gets a read lock on the current database state. The details of the captured value depends on the implementation technique, but it is argued that conceptually that the value can be considered to represent the entire database state. Thus, we can restore the previously captured state by the following function:

```
> restoreDB :: Database -> DB ()
```

Every writer's operation should check to see if it conflicts with a read lock on the expected version. Whenever the read lock is obtained by a reader expression, the writer does not modify the state itself, but generates a new state value. This conservatism is required to ensure that all the readers and writers are failure-free. While a lock on a version is explicitly got by `getDB`, the lock should be released by a garbage collector. When an expression that holds a lock on a certain version becomes garbage, the garbage collector can release the lock.

In addition to these operators, the original state of a transaction is useful for functions that manipulate more than one database versions. For now, we only show the primitive operator to access the original state:

```
> getOrigDB :: DB Database
```

This operator does not increase the cost of database state manipulation. Since the original database state is always kept (at least virtually) intact during a database transaction to support the rollback operation, even when this operator is executed in a transaction, the number of database state values required in the transaction does not increase.

5.2 On-the-fly location access

Before showing on-the-fly access operators to the captured state, we shall mention again that the entity locations do not incur any typing trouble. An entity location is accessed only through traversal from a persistent root value or is newly created by `newDB`. The first case is not dangerous because of the ground type restriction noted in the previous section, and the newly created entity is manipulated consistently because of the monomorphism restriction put on the lambda-bound variables. Moreover, the single-threadedness is always satisfied, since there is no *wrapper* that runs a state-transformer on-the-fly.

There are two primitive operators for retrieving data from the captured database state lazily:

```
> readRef :: Entity a  => Database -> DBRef a -> a
> readRoot :: PerRoot a => Database -> a
```

The former dereferences a surrogate in the given version, and the latter retrieves a persistent root from it.

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:

```
> massAndCost :: Database -> Part -> (Int, Int)
> massAndCost db Basic{pCost,pMass}
>   = (pMass, pCost)
> massAndCost db Composite{pCost,pMass,pComposedOf}
>   = (pMass + subm, pCost + subc)
>   where
>     submcs
>       = [ (c*q, m*q) | (p, q) <- pComposedOf,
>                        (m, c) <- [ massAndCost(db) (readRef(db) p) ]
>     subm = sum (map fst submcs)
>     subc = sum (map snd submcs)
```

The differences from the function shown in Fig. 2 are that `readRef` is used and that `db` is passed to `massAndCost`.

5.3 The "what-if" semantics

Multiple versions allow us to define an action that is equivalent to `return` except that it forces the surrounding (sub-)transaction to restore the original state:

```
> markAbortDB :: a -> DB a
```

Note that this function affects only the *current* state value left at the end of the transaction. Even when the original state is restored, the computation process performed in the transaction is still valid. This property and the explicit versioning support the "what-if" scenario effectively. For example, suppose that the total mass and cost of a particular composite part have to be evaluated under a hypothesis that a certain database update is performed. Section 5.2 has already defined `massAndCost`, which computes the cost and mass of a particular part, thus the following simple function suffices:

```
> whatIfCandM :: DB () -> DBRef Part -> IO (Int, Int)
> whatIfCandM updateParts pid
>    = transaction (
>        do updateParts
>           db <- getDB
>           markAbortDB (massAndCost(db) (readRef(db) pid)) )
```

This function can be generalized so that the hypothetical update is performed on a particular database state. The key technique is to use `restoreDB`, which makes the given database state current. By temporarily making the given state current and then discarding the modification applied to the state, the "what if" scenario is written as

```
> hWhatIfCandM :: Database -> DB () -> DBRef Part -> IO (Int, Int)
> hWhatIfCandM db updateParts pid
>    = transaction (
>        do restoreDB db
>           updateParts
>           db <- getDB
>           markAbortDB (massAndCost(db) (readRef(db) pid)) )
```

5.4 Non-materialized views

One of the important features of database management systems is the support of views, or derived values. Views can be categorized into two types: materialized views, and non-materialized views, where view materialization is the computation of view values. The materialized views are associated with

```
> data BParts = BParts {basicParts::[DBRef Part]}
> instance PerRoot BParts
>   initValue db
>     = let PartExt{ parts } = getRoot(db)
>       in BParts{ basicParts =
>            [ pid | pid     <- parts,
>                    Basic{} <- readRef(db) pid ] }
>   isView _ = True
>
> data CParts = CParts {compositeParts = [DBRef Part]}
> instance PerRoot CParts
>   initValue db
>     = let PartExt{ parts } = getRoot(db)
>       in CParts{ basicParts =
>            [ pid | pid         <- parts,
>                    Composite{} <- readRef(db) pid ] }
>    isView _ = True
```

Figure 13: Declarations of non-materialized view roots.

their pre-computed values, so the maintenance of them reduces to an integrity enforcement issue, which will be described in the next section. On the other hand, reading a non-materialized view invokes the computation routine associated with the view. The former requires more storage, while the latter incurs a computation overhead. Which to use should be decided case-by-case; thus, database management systems must support both.

In the proposed Haskell-based database management environment, both materialized and non-materialized views are supported. The remainder of this section describes the support of non-materialized views only. The materialized views will be discussed later in Section 6.4. To illustrate, we define two views of the stored parts: one for basic parts and the other for composite parts.

The idea of implementing non-materialized views is straightforward. Remember that the initial values of persistent roots are specified by the `initValue` method. This means that when there are no explicitly stored root values, this method is automatically invoked by the underlying system. Hence, if we define the view definition in the right-hand side of the method, it becomes the default value of the root. Thus, providing a *switch* to prohibit the modification of the root values is enough to define non-materialized views. Following this idea, the non-materialized views for the basic and composite parts can be specified as shown in Fig. 13.

The argument supplied to `initValue` is the current database state. The right-hand side of the method computes the initial value using the current database value. The `isView` method works as the switch to control the updatability of the root value. Only if it evaluates to `False`, the root update action `writeRootDB` is allowed to be executed on this root. Otherwise, the action invokes a run-time error. `isView` is typed thus

```
>     isView :: [a] -> Bool
```

where `a` is assumed to be the type of the persistent root. Because of the restriction imposed by the class mechanism of Haskell, the type of the persistent root must appear in the type signature of the class method. In the underlying implementation, the actually supplied value is `[]`, and the type consistency is manipulated by an explicitly specified type signature.

In contrast to the materialized views, non-materialized views are dynamically computed on demand. The notable advantage over the materialized views is the minimal cost of view maintenance for programmers and the systems, but the view values may be repeatedly computed. There is a simple solution to this problem. As the type signature of the `initValue` implies, the view values are computed based on the database state supplied as the first argument. If the computed value is cached before it is returned to the user code requiring the view value, later access to the same view can be performed simply by fetching

the value in the system cache associated with the current database state. Thus, the internal view access code may look like this:

```
do b <- ''check if the current state caches the required value''
   if b && ''cached value is that for the current state''
    then return ''the cached value''
    else do db <- getDB
            let val = initValue db
            ''store val in the cache''
            return val
```

To keep the cache contents consistent, the view values are recomputed after the database state has been updated. The above pseudo-code uses `getDB`. This locks the current version of the database state, thus the lazy computation of view values is performed safely.

6 Triggers for Integrity Enforcement

A database model has as an integral part a mechanism to specify integrity constraints of the database state. Persistent programming languages should also have certain primitives to help the designers specify the constraints, but the general purposeness of the languages makes it difficult for a fixed set of built-in primitives to support arbitrary integrity constraints. Thus, instead of facilitating the Haskell language with such extensions, we make use of automatic triggering of functions.

As has been explained, most primitive operations are overloaded, and entity types and persistent roots are associated with `Entity` and `PerRoot`, respectively. This leads to a technique that associates the operators with "hooks" to customize the operations. The idea of using hooks to customize predefined functions prevails in the Lisp community. The same idea is also adopted in Common Lisp Object System (CLOS) (Keene 1989). Thus, the same customizability is inherited by persistent programming languages based on Lisp or CLOS (Fishman et al. 1987; Paepcke 1988; Barbedette 1992). The main aim of this section is to show that the same kind of customizability is achieved in the context of purely functional database programming, and that the ability to handle multiple versions naturally extends to support the transaction boundary rule execution.

This section first summarizes the concept of active databases, and then proposes a method of incorporating a triggering mechanism into the methodology using the simulation of the type extent model of persistence as an example. The remainder explains how other integrity constraints, i.e., including mutual references, materialized views, and dangling reference exceptions, are supported by the mechanism, and also shows that the transaction-boundary rule-firing is supported.

6.1 Active database technology

Active database technology was originally proposed to strengthen the power of integrity enforcement by database management systems (Eswaran and Chamberlain 1975). Since then the concept was generalized to handle more general rules in HiPAC (McCarthy and Dayal 1989), POSTGRES (Stonebraker et al. 1990), Starburst (Widom 1996), and ODE (Agrawal and Gehani 1989). Among these, the HiPAC project proposed Event-Condition-Action (ECA) rules, as a general formalism of active database management functions. An ECA rule, as the name suggests, comprises three parts. The first part specifies the relevant events, such as update of a certain relation, which enable the rule. The second part prescribes the Boolean condition that specifies when to invoke the action given in the third part. In an abstract syntax, the ECA rule can be written as

$$\text{on} < \textit{event} > \text{if} < \textit{condition} > \text{then} < \textit{action} >.$$

Ghandeharizadeh et al. (1996) characterized execution semantics of ECA rules into two aspects: the *coupling modes*, which were identified in the HiPAC project, and the number of state values included in condition checking and action invocation. The coupling modes specify the relative timing among event detection, condition checking, and action execution. The timing is either *immediate*, *deferred*, or *separate*. Immediate coupling means that the second activity follows the first immediately, while deferred coupling means that the second activity is postponed until some later time but still falls within the

```
> instance Entity Part where
>   afterNew pid part
>     = do PartExt{parts} <- readRootDB
>          writeRootDB PartExt{parts=pid:parts}
```

Figure 14: Hook to automatically insert a part into the root

same transaction. Separate coupling means that a concurrent process is spawned to perform the second activity.

This and the next sections consider only *immediate* and *deferred* coupling modes, since the transaction model considered here does not support concurrent transaction execution. It might be possible to generalize the environment to support concurrent threads as proposed in (Akerholt et al. 1993; Trinder 1995). This paper, however, puts more stress on the applicability of the purely functional programming paradigm to flexible control of database state management. The issues on transaction formalism and architectures are out of its scope.

6.2 Simulating the type-extent model of persistency

The reachability model of persistence requires that every piece of data should be reachable from at least one of the persistent roots. This is the source of flexibility, but also the source of complexity of the extent management. In manipulating a database entity, programmers must always take into account the set of related persistent roots. In addition, even though a program works at a particular point, modification of the set of persistent roots would invalidate the program; the newly introduced roots are not necessarily taken into account in the original program design phase.

Remember that a database type is specified by making it an instance of the `Entity` class:

```
> module Parts where
> data Part
>   = Basic    { pName::String,         pCost, pMass::Int,
>                pUsedBy::[DBRef Part], pSuppliedBy::[DBRef Supplier]}
>   | Composite{ pName::String,         pCost, pMass::Int,
>                pUsedBy::[DBRef Part], pComposedOf::[(DBRef Part, Int)]}
> instance Entity Part
```

This example specifies no instance methods for `Part`. This implies that the default entity management strategy suffices for this type. If programmers decide to invoke actions at the entity management operations like *new*, *read*, and *write*, appropriate actions can be given as instance methods. Since it would be dangerous to allow for such direct modification of primitive functions, we design the primitive functions so that they call customizable overloaded functions or *hooks*. By default such companion functions do nothing. If the users define the hooks, they are called automatically according to the database modification.

To illustrate, we define a hook that automatically inserts a newly created `Part` entity into the persistent root associated with `PartExt`. We only have to declare the action as an instance method for `afterNew` as shown in Fig. 14. The first argument is the surrogate of the newly created entity, and the second one is the associated value. The specified action gets the root value through `readRootDB`, constructs a new root value, and then updates the root with the new list value. The underlying system automatically calls this hook for application programs. Therefore, the simple expression

```
newDB Basic{pName="part0010", pCost=100, ...}
```

implies that

```
do pid             <- newDB' Basic{pName="part0010", pCost=100, ...}
   PartExt{parts} <- readRootDB
   writeRootDB PartExt{parts=pid:parts}
```

where `newDB'` designates the built-in primitives of the surrogate creation[13].

This style of entity management is very flexible. For instance, we can split the persistent root into two roots: one for `Basic` parts and the other for `Composite` parts. In this case, the persistent roots regarding the `Part` entities may be declared thus:

```
> data BParts = BParts{bparts::[DBRef Part]}
> data CParts = CParts{cparts::[DBRef Part]}
>
> instance PerRoot BParts
> instance PerRoot CParts
```

Now that the root has been split, the `afterNew` method must be modified as follows:

```
> instance Entity Part where
>   afterNew pid Basic{}
>     = do BParts{bparts} <- readRootDB
>          writeRootDB BParts{bparts = pid:bparts}
>   afterNew pid Composite{}
>     = do CParts{bparts} <- readRootDB
>          writeRootDB CParts{bparts = pid:bparts}
```

Splitting the root might affect programs that have already been developed, but the view mechanism solves this problem to some extent. Remember that Fig. 13 defines two views for basic and composite parts. A similar definition specifies a view that automatically merges the two lists to construct the full list of stored parts.

Another important difference between the type-extent model of persistence and the reachability model is the ability to allow for explicit deletion of entities. Since simulating this operation is closely related to the mutual references, it will be discussed just below.

6.3 Relationship management

Insertion and update of an entity must be properly handled, since relationships between entities may be implemented through mutual references. The above "hooking" technique is also applicable to maintain such mutual references in a terse and declarative way.

Mutual references have been seen in the part-supplier database. A part entity refers to other parts through the `pUsedBy` and `pComposedOf` fields. Similar relationships also exist between `Part` and `Supplier` values through `pSuppliedBy` and `sSupplies`. Whenever a supplier stops to supply a part, the corresponding `Supplier` entity must be modified accordingly. In addition, the supplied `Part` entity must also be modified to make the mutual references consistent. To support the schema level specification of mutual reference management, two `Entity` class operations, `beforeUpdate` and `afterUpdate` are used as the hooks. These hooks are respectively called before and after `writeDB`.

Fig. 15 shows the `afterUpdate` hook to maintain the mutual references between parts and suppliers. The hook is called using three arguments: the surrogate, the old value, and the new value. This hook simply checks the "delta" between the old and new values, and modifies the affected parts accordingly. Note that \\ is the list difference operator, and `delete` deletes the first argument from the second list if it exists.

A similar action may be specified on the `Part` type side. In this case, the `afterUpdate` method for `Part` is defined so that the related supplier entities are modified accordingly. To suppress the infinite

[13] Of course, the name of the built-in primitive depends on the implementation. We name it just to discriminate it from the `newDB` primitive function.

```
> instance Entity Supplier where
>   afterUpdate sid                    -- modified entity's surrogate
>         Supplier{sSupplies=oldParts}  -- oldValue
>         Supplier{sSupplies=newParts}  -- newValue
>
>     | newParts == oldParts            -- if no difference
>     = return ()                       --    do nothing
>
>     | otherwise                       -- else
>     = let added   = newParts \\ oldParts
>           deleted = oldParts \\ newParts
>       in do modifyParts (:)    added
>             modifyParts delete deleted
>    where
>      modifyParts f [] = return ()
>      modifyParts f (p:ps)
>        = do Part{pSuppliedBy=oldValue} <- readDB p
>             let newValue = f sid oldValue
>             writeDB p newValue
>             modifyParts f ps
```

Figure 15: The `afterUpdate` method for `Supplier`

calling of hooks, the hook shown above is defined with two *guards*. A guarded right-hand side comprises a condition and an expression written in the following form:

$$f\, pat_1 \cdots pat_n \quad \begin{array}{ll} | & condition_1 = expression_1 \\ & \vdots \\ | & condition_m = expression_m. \end{array}$$

If there are more than one guarded right-hand sides, the first one whose guard is evaluated as being true is selected. In the program shown in Fig. 15, the action to maintain the consistency is taken only if the delta between the old and new values is not empty.

The safety of the action is the responsibility of the programmers. While this is not satisfactory, it is theoretically impossible to analyze the programs completely, since the language considered here is general-purpose.

The hooks associated with `writeDB` also make entity deletion easier. For instance, deleting a part from a database requires us only to clear the fields of the related entities, and to delete the part from the persistent root value:

```
> deletePart pid
>   = do part <- readDB pid
>        case part of
>          Basic{}     ->
>            writeDB pid part{UsedBy=[],pSuppliedBy=[]}
>          Composite{} ->
>            writeDB pid part{pUsedBy=[],pComposedOf=[]}
>        PartExt{parts} <- readRootDB
>        writeRootDB PartExt{parts=delete pid parts}
```

When the `writeDB` function is executed, the `afterUpdate` hook automatically deletes the references to this part in `Part` and `Supplier` entities, so there is no need for the programmers to take into account mutual references associated with the part to be deleted.

```
> instance Entity Part where
>   afterNew pid Basic{}
>     = do BParts{ basicParts  = parts }    <- readRootDB
>          writeRootDB BParts{ basicParts = pid:parts }
>   afterNew pid Composite{}
>     = do CParts{ compositeParts = parts } <- readRootDB
>          writeRootDB CParts{ compositeParts = pid:parts }
>   afterUpdate pid Basic{} Composite{} -- basic to composite
>     = do BParts{ basicParts = parts }     <- readRootDB
>          writeRootDB BParts{ basicParts = delete pid parts }
>          CParts{ compositeParts = parts } <- readRootDB
>          writeRootDB CParts{ compositeParts = pid:parts }
>   afterUpdate pid Composite{} Basic {} -- composite to basic
>     = do BParts{ basicParts = parts }     <- readRootDB
>          writeRootDB BParts{ basicParts = pid:parts }
>          CParts{ compositeParts = parts } <- readRootDB
>          writeRootDB CParts{ compositeParts = delete pid parts }
>   afterUpdate pid _ _
>     = return ()
```

Figure 16: Materialization management routines for the basic and composite parts views.

6.4 Materialized views

A materialized view is implemented by combining a persistent root and related hooks: the root stores the view value, and the hook maintains the value. For example, the materialized views for persistent roots for basic and composite parts are declared through the following instances:

```
> data BParts = BParts {basicParts::[DBRef Part]}
> instance PerRoot BParts
>   initValue _ = BParts{ basicParts = [] }
>
> data CParts = CParts {compositeParts = [DBRef Part]}
> instance PerRoot CParts
>   initValue _ = CParts{ compositeParts = [] }
```

The materialized value management is specified through the triggered routines for the `Entity-Part` instantiation, as shown in Fig. 16. The `afterNew` function is the hook that is executed after `newDB` is executed. In this case, it inserts the newly created entity into either of the materialized views according to the kind of the part. The `afterUpdate` function handles the case where the kind of a part is changed from basic to composite or *vice versa*. When the kind is not changed by the part entity update, no action is taken.

The usage of materialized views is not different from that of ordinary roots. For example, the selecting expensive basic parts, say ones costing more that 100 dollars, can be selected simply by traversing the `BParts` persistent roots:

```
> expensiveBasicParts
>   = do BParts{basicParts} <- readRootDB
>        parts              <- mapM readDB basicParts
>        return [ pName | Basic{pName, pCost} <- parts, pCost > 100 ]
```

Note that this code does not differ much from a code which directly selects required values from the list of all the parts. Since the scan is performed only on the basic values, however, the run-time performance should be better especially if the ratio of basic parts to all parts is low.

6.5 Exception handling

Even though the reachability model of persistence is adopted, an invalid-reference exception occurs when a newly created entity surrogate is looked up in an *older* database state. The following trivial example exhibits a trivial case:

```
> staleDereference name cost mass
>   = do db <- getDB
>        s  <- newDB Basic{ pName = name, pCost = cost, pMass = mass,
>                           pSuppliedBy = [], pUsedBy = [] }
>        return (readRef(db) sid)
```

While the newly created surrogate is valid only after the `newDB`, the first argument of `readRef` is created before the operation. What is worse, the call-by-need semantics reveals this erroneous situation only when the value is required, not when the value is defined.

Facilitating hooks for database operations can be applied to this case. So we make the exception handler one of the class operators like this:

```
> class Entity 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 `readRef` operator. Whenever it detects an invalid surrogate, it applies `whenDangling` to the current database and the given surrogate. For example, let `db` denote a value of type `Database`, and `v` denote a surrogate that is invalid in `db`. Then the following equivalence holds:

```
readRef db v  ≡  whenDangling db v  ≡  error "dangling reference".
```

When some default value exists, users may specify the value in instance declarations like this:

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

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

```
readRef db v  ≡  whenDangling db v  ≡  Basic "B000" 0 0 [].
```

6.6 Transaction boundary rule execution

In this subsection, we generalize the above idea in order to support the *deferred* coupling mode. The strategy adopted here introduces a queue of jobs. A job comprises the condition part and the action part. The coding strategy of the job gives the coupling mode of condition checking and action execution. If the job checks the condition and then takes a certain action immediately, the condition-action coupling mode is "immediate". On the other hand, the job may enqueue the action again as a non-conditional job. In this case, the coupling mode is "deferred". The queue is organized according to a certain queuing discipline. Under the simplest discipline, the queue may be organized as a first-come, first-served structure. A more flexible discipline may associate precedence values with the jobs in the queue. To support both flexibility and simplicity, we use a queue sorted by precedence values and then by the order of enqueuing.

The queue is controlled by the following primitive operation:

```
> enqueueDB :: Int -> (Database -> DB ()) -> DB ()
```

The first argument specifies the precedence of the job. The second argument specifies the job itself. Note that the action is of type `Database -> DB ()` instead of `DB ()`. This reflects the semantics of the queue execution. We follow the *transaction boundary rule firing*, under which the enqueued jobs are executed

at the end of the surrounding transaction. The relevant database state values may be shown like this

$$db_{orig}, \cdots, db_{prop}, \cdots, db_{cur},$$

where db_{orig} is the original database state at the transaction start time, db_{prop} is the proposed state when the commit procedure starts, and db_{cur} is the current database state. In the examples in the rest of this section, these values are usually bound to `db0`, `db1` and `db2`, respectively.

As an example, consider a situation where the type-extent model of persistency is simulated through the `afterNew` class method. We have already showed the immediate coupling in Fig. 14. On the other hand, if we want to defer the action, the following rewriting of the code is enough.

```
> instance Entity Part where
>   afterNew pid part
>     = enqueueDB 0 (\db1 ->
>         do PartExt{parts} <- readRootDB
>            writeRootDB PartExt{parts=pid:parts} )
```

After the surrounding transaction starts the commit procedure, the enqueued actions are executed. This example shows the method of writing a non-conditional deferred action. In this coding, the root update can be observed only after the current transaction has been committed.

The enqueued job may refer to related values to see if its action should be taken. To illustrate, let us consider the case where a supplier who supplies no parts should be deleted. This cannot be simply executed by `afterUpdate`. Even if a supplier supplies no parts at a certain point of a transaction, the supplier value may be modified so that it supplies some other parts in the same transaction. A possible solution is to enqueue an appropriate job whenever a relationship between a supplier and a part is modified. The typical specification may look like this:

```
> instance Entity Supplier where
>   afterUpdate sid oldVal newVal
>     = enqueueDB 0 (aJob sid)
>   where
>     aJob sid db1
>       = do Supplier{sSupplies} <- readDB sid
>            when (sSupplies == []) ( do
>               SuppExt{suppliers=supps} <- readRootDB
>               let supps' = delete sid supps
>               writeRootDB SuppExt{supplies=supps'} )
```

where `when` is a library function defined as

```
> when p a = if p then a else return ()
```

Even though it is not syntactically clear, the condition and the action part of the ECA rule are encoded in this job.

So far the several parameters supplied to the jobs or the methods have not been fully utilized. The following example makes use of the original state of the transaction: when the cost of a basic part is reduced by more than 20%, stops the program with an error message. The program looks like this:

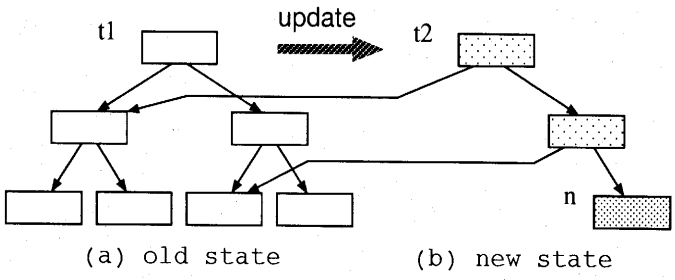


Figure 17: Lazy state transformers with database state in a tree form: (a) *t1* is the root of the old state, and (b) *t2* is that of the new one. *n* is the newly created node by the update operation. The path from the root to the updated node *n* is also duplicated so that the old state is intact.

```
> instance Entity Part where
>   afterUpdate pid oldVal newVal
>     = enqueueDB 0 (aJob pid)
>     where
>       pred pid Basic{pCost=cost1} Basic{pCost=cost2}
>         = fromInteger cost1 / fromInteger cost2 < 0.8
>       pred pid oldVal             newVal
>         = False
>       aJob pid db1
>         = do db0 <- getOrigDB
>              when (pred pid (readRef(db0) pid) (readRef(db1) pid))
>                error ("Part " ++ pName (readRef(db0) pid)
>                    ++ ": cost reduction error")
```

In this program, the original state `db0` and the proposed state `db1` are used to compute the difference between the original and proposed costs. If `db1` were replaced with the current database retrived by `getDB`, the difference between the original and current values would be computed.

Queued jobs are executed at the commit time until the job queue becomes empty. More specifically, the state transition sequence in a transaction looks like this

$$db_{orig}, \cdots, db_{prop_0}, db_{prop_1}, \cdots, db_{cur},$$

where db_{prop_0} is the state at the commit time, and db_{prop_1} is the state after the rules in the queue at the commit time have been executed, and so on. Remember that the type of `enqueueDB` is

```
> enqueueDB :: Int -> (Database -> DB  ()) -> DB ()
```

The db_{prop_i} is passed to all the jobs in phase $i+1$ through the argument value.

7 Related Work

7.1 Shadow paging

The present approach is based on imperative state transformers, but there is another approach based on the *shadow paging* technique (Argo et al. 1990; Nikhil 1990; Nikhil 1988; Maier and Stein 1987). In that approach, the database state is immutable, and it is updated by tearing apart the old value and constructing a new one. To reduce the cost of state construction, unchanged parts of the new and old values are shared by backward pointers.

Fig. 17 shows the database state structured in a tree form. The left tree is the old state with *t1* as its root. The right tree exhibits the new state after the node labeled *n* has been updated. Notice that the old state is completely intact, and the newly created state include three new nodes: one for the updated node, and two for the path from the root to the updated node.

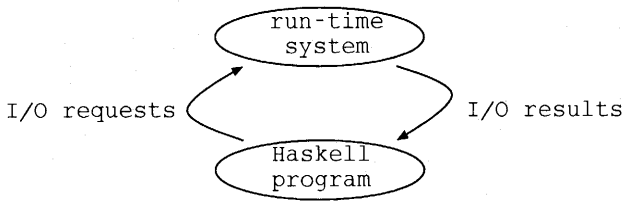


Figure 18: Stream-based communication for a program and an external run-time system

The cost of duplicating the path to the updated node could be reduced by modifying it directly instead of creating a new node if possible. This condition can be easily checked if a reference count garbage collector is utilized to implement the persistent pages[14].

The shadow paging chooses duplication by default, and only when applicable, is an in-place update performed. On the other hand, the methodology proposed in this paper modifies the current version destructively by default, and only when it must be kept for later use, a new version is generated.

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

7.2 Persistent streams

Another model of the mutable state uses persistent streams (McNally and Davie 1991; Hammond et al. 1993). While Haskell 1.3 and 1.4 specify monadic I/O operations, the older language definition, Haskell 1.2 (Hudak et al. 1992), adopted dialogue-based I/O operations. In this model, a Haskell process is connected with an external run-time process through two command channels (Fig. 18). The left channel is used to send operation requests from the program to the run-time system, and the other is used by the program to receive the results of the processed requests.

Staple (McNally and Davie 1991) extends this I/O system to support persistent streams. A persistent stream is similar to a file except for the contents. While a file contains a list of characters, a persistent steam stores a value of type `any`, which has its type annotation in its representation. Programmers can convert a normal value into the corresponding value of type `any` through `mkany`. The reverse conversion is performed through `coerce`. In this conversion process, the dynamic type checking is performed at run-time to see if the expected type is equal to the type associated with the `any` value. The requests in a program are organized in a lazily generated list, and the run-time process works as a sequential transaction manager that processes the incoming requests in the order in the list. Hence, the side effects by the run-time process does not ruin the referential transparency of the language.

In spite of the difference between streams and monads, the persistent stream can support the approach described here. Indeed, Peyton Jones and Wadler (1993) show the equivalence between the stream-based I/O system and monad-based one. The difficulty arises, however, in the implementation of the database state versioning. Since the external run-time system manipulates all the input operations including file and terminal I/O operations, creating a new version leads to duplication of the complete system environment. To support the versioning, the persistent stream model must be extended so that the run-time process is composed of multiple request handlers to which requests are appropriately dispatched.

Another difference between Staple and the proposed approach exists in the typing principle. As explained above, Staple is a strongly typed language, while our proposal adheres to static typing, which is the basic typing principle of Haskell. Although strong typing is more flexible, it lessens the reliability of database programs. A persistent stream is identified by its string names. Programs can change the association between the name and the contents. Since the contents are a value of type `any`, the actual type of the value may change in future. Hence, even though a program runs correctly at a particular time, it may incur run-time type errors in future. On the other hand, a statically typed correct program never goes wrong in future unless the database schema is invalidated.

[14]Note that the reference count garbage collector is claimed to be too slow for realistic applications. What is worse, it is difficult to scale up to multi-user programming environments.

7.3 Checking linearity

Some language systems detect the "serializability" of impure constructs in an expression. A well-known functional programming language is Clean (Achten et al. 1993). Its type checker detects the side effects of expressions and ensures the linearity of the effects not be compromised. Similar linearity analysis is also found in PFL (Sutton and Small 1995). An example of such erroneous expressions is thus

$$(\mathit{include}\ t\ r,\ \mathit{exclude}\ t\ r),$$

where the first subexpression inserts a tuple t to the relation named r, and the second one deletes the same tuple. The result, of course, depends on the order of evaluation. PFL ensures that every expression is confluent by making use of *linear typing*. Hence the above expression is *type-incorrect*, and is detected as illegal before execution. Note that linear typing ensures that mutable values are never duplicated nor discarded, and thus results in the language being *confluent*. Even though confluence ensures that every expression is unambiguous, the referential transparency of the language is compromised.

8 Conclusion

A persistent programming environment for the standard, non-strict, purely functional language, Haskell, has been proposed. This section briefly describes the current status of the prototype, and then addresses further research issues.

8.1 Current status of the Prototype

To show the feasibility of the environment, a prototype has been implemented by porting Hugs 1.3, a Haskell-compliant successor of Gofer (Jones 1994a), to Texas Persistent Store 0.5 (Singhal et al. 1992)[15]. The current prototype implements all the proposed features. Although the garbage collection procedure in connection with the object management is not sufficient from the viewpoint of performance and strictness, this prototype can demonstrate the minimum feasibility of the lazy functional programming for database manipulation.

8.2 Further research issues

Several important issues are yet to be explored to make the programming environment more practical for database management.

Relationship to database modeling methodologies

The first phase of database management process is to model the real world using a semantically rich data model. We have not yet facilitated the proposed environment with any conceptual design methodology. The database research community, however, has seen intensive research activities in this area. The semi-automatic translation of the entity-relationship model to the relational model has been studied intensively (Chen 1976; Teorey 1994; Elmasri and Navathe 1994). The recent trend in object-orientation has led to the development of Object Definition Language (ODL) (Cattel and Barry 1997), which models the real world in object-oriented terms independent of specific implementation programming languages. Ceri and Fraternali (1997) cover various topics in the conceptual design phase with the object-oriented or semantic database models and mapping abstract schemas to those in implementation models. FDL (Poulovassilis and Kind 1990) and its successor PFL (Small and Poulovassilis 1991; Small 1993) directly support the functional data model proposed by Shipman (1981).

Active rule formalization

The proposed manipulation of active rules is not based on a mathematical foundation or formal model. Instead, incorporating the functionality has been made possible through the ability to handle multiple versions, the inherent computational completeness, and the class mechanism to support customizable

[15]The first prototype (Ichikawa 1995) was developed using Glasgow Haskell Compiler 0.29 (Hall et al. 1993) with C procedures to manage database type extent.

overloaded functions. There are some known proposals of formal active database models based on delta-state (Ghandeharizadeh et al. 1996; Doherty et al. 1996), logic (Fraternali and Tanca 1995), and functions (Reddi et al. 1995; Poulovassilis et al. 1996). Reddi et al. (1995) and Poulovassilis et al. (1996) address the technique that allows execution of any user-defined function to be treated as an event. Even though this direction is practical for dedicated functional database programming languages, it is not necessarily clear that this is so in the persistent programming environment extending a volatile programming language.

Storage management and concurrency control

The storage manager of the prototype is not tuned for a persistent programming environment. In particular, the garbage collection algorithm for multiple versions has not been fully investigated yet. Another issue is the concurrency control. Even if the prototype were developed in a multi-user object-oriented programming environment such as Shore (Carey et al. 1994), the high update-rate due to garbage collection and lazy evaluation would make it difficult to allow for concurrent updating of the database storage. Remember that every suspended expression is updated whenever the value of the expression is reduced, and that partial application also may allocate heap cells.

Optimization

One of the preferable features of the purely functional computation paradigm is its optimizability based on equational reasoning. As a result of this property, the functional programming language community has developed many optimization techniques for volatile functional languages. (See e.g., Peyton Jones and Lester (1992) for the pointers). On the other hand, the database community has made use of this property to devise optimization techniques that can typically be seen in Trinder (1991), Buneman et al. (1994), and Poulovassilis and Small (1996). We have to devise an optimization procedure that not only improves the performance of functional computation but also takes into account the physical data independence and the properties of storage devices. These two aspects are discussed in the database and functional programming language communities, and have not been explored well in conjunction with each other.

Schema evolution and reflection

The current prototype does not handle schema evolution. The only method of modifying schema is to initialize the database state and the script defining the database schema, even though utility functions and query functions are freely redefinable. Staple (McNally and Davie 1991) handles schema evolution by retaining old modules as they are. Although this is a plausible approach, users must always be aware of which modules are accessed through persistent roots. Another approach uses dynamic typing and dynamic module binding as in Napier88 (Dearle et al. 1989).

Another aspect of the persistent programming is the reflection mechanism. As has been noted in the context of Napier88 by Kirby (1992), and has also been noted in a number of standard database texts, run-time accessibility to the meta level information, or schema information, is useful for users and for certain applications like graphical database query interfaces.

Feasibility for advanced database applications

Practical targets of the persistent programming languages would be more complex than the running example, the part-supplier database. Multimedia applications would require more sophisticated user interfaces, and put more stress on computation resources. These could be considered as the issues for applying purely functional programming for realistic applications. Unless these issues are explored, however, the persistent programming environment based on the paradigm will not become a practical approach.

References

Achten, P., J. van Groningen, and R. Plasmeijer (1993). High level specification of I/O in functional languages. In J. Launchbury and P. Sansom (Eds.), *Functional Programming, Glasgow 1992*, pp. 1–17. Springer-Verlag.

Agrawal, R. and N. H. Gehani (1989, May). ODE (object database and environment): The language and the data model. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pp. 36–45.

Akerholt, G., K. Hammond, S. L. Peyton Jones, and P. W. Trinder (1993). Processing transactions on GRIP: A parallel graph reducer. In *Proceedings of PARLE '93*, pp. 634–647. Lecture Notes in Computer Science 694, Springer-Verlag.

Argo, G., J. Hughes, P. Trinder, J. Fairbairn, and J. Launchbury (1990). Implementing functional databases. In F. Bancilhon and P. Buneman (Eds.), *Advances in Database Programming Language*, pp. 165–176.

Atkinson, M. and R. Morrison (1995, July). Orthogonally persistent object systems. *The VLDB Journal 4*(3), 319–402.

Atkinson, M. P., P. J. Bailey, K. J. Chisholm, W. P. Cockshott, and R. Morrison (1983). An approach to persistent programming. *Computer Journal 26*(4), 360–365.

Atkinson, M. P. and P. Buneman (1987, June). Types and persistence in database programming languages. *ACM Computing Surveys 19*(2), 105–190.

Bancilhon, F., C. Delobel, and P. Kanellakis (Eds.) (1992). *Building an Object-Oriented Database System: The Story of O_2*. Morgan Kaufmann.

Barbedette, G. (1992). Lisp O_2: A persistent object-oriented lisp. In *Bancilhon et al. (1992), Chapter 10*, pp. 215–233.

Bernstein, P. A., V. Hadzilacos, and N. Goodman (1987). *Concurrency Control and Recovery in Database Systems*. Addison Wesley.

Bird, R. and P. Wadler (1988). *Introduction to Functional Programming*. Prentice Hall.

Buneman, P., L. Libkin, D. Suciu, V. Breazu-Tannen, and L. Wong (1994, March). Comprehension syntax. *ACM SIGMOD RECORD 23*(1), 87–96.

Carey, M. J., D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling (1994, May). Shoring up persistent applications. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pp. 383–394.

Cattel, R. G. G. and D. K. Barry (Eds.) (1997). *Object Database Standard: ODMG 2.0* (2nd ed.). Morgan Kaufmann.

Ceri, S. and P. Fraternali (1997). *Designing Database Applications with Objects and Rules: the IDEA Methodology*. Addison-Wesley.

Chen, P. P.-S. (1976, March). The entity-relationship model — toward a unified view of data. *ACM Transactions of Database Systems 1*(1), 9–36.

Christophides, V., S. Abiteboul, S. Cluet, and M. Scholl (1994, May). From structured documents to novel query facilities. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pp. 313–324.

Connor, R., D. McNally, and R. Morrison (1991). Subtyping and assignment in database programming language. In P. Kanellakis and J. W. Schmidt (Eds.), *Proceedings of the Third International Workshop on Database Programming Languages*, pp. 363–382. Morgan Kaufmann.

Davie, A. J. T. (1992). *An Introduction to Functional Programming Systems Using Haskell*. Cambridge University Press.

Dearle, A., R. Connor, F. Brown, and R. Morrison (1989). Napier88 — a database programming language? In R. M. Richard Hull and D. Stemple (Eds.), *Proceedings of the Second International Workshop on Database Programming Languages*, pp. 179–195. Morgan Kaufmann.

Doherty, M., R. Hull, and M. Rupawalla (1996, June). Structures for manipulating proposed updates in object-oriented databases. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pp. 306–317.

Elmasri, R. A. and S. B. Navathe (1994). *Fundamentals of Database Systems* (2nd ed.). CA: Benjamin/Cummings.

Eswaran, K. P. and D. D. Chamberlain (1975, September). Functional specifications of a subsystem for data base integrity. In *Proceedings of the First International Conference on Very Large Data Bases*, pp. 48–68.

Fishman, D. H., D. Beech, H. P. Cate, E. C. Chow, T. Connors, J. W. Davis, N. Derrett, C. G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. A. Neimat, T. A. Ryan, and M. C. Shan (1987, January). Iris: An object oriented database management system. *ACM Transactions on Office Information Systems 5*(1), 48–69.

Flickner, M., H. Sawhney, W. Niblack, J. Ashley, Q. Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, and P. Yanker (1995, September). Query by image and video content: The QBIC system. *IEEE Computer 28*(9), 23–32.

Fraternali, P. and L. Tanca (1995, December). A structured approach for the definition of the semantics of active databases. *ACM Transactions on Database Systems 20*(4), 414–471.

Gamerman, S., C. Lanquette, and F. Vélez (1992). Using database applications to compare programming languages. In *Bancilhon et al. (1992), Chapter 13*, pp. 278–324.

Ghandeharizadeh, S., R. Hull, and D. Jacobs (1996, September). Heraclitus: Elevating deltas to be first-class citizens in a database programming language. *ACM Transactions on Database Systems 21*(3), 370–426.

Gordon, A. D. (1994). *Functional Programming and Input/Output.* Cambridge University Press, Distinguished Dissertations in Computer Science.

Hall, C., K. Hammond, W. Partain, S. L. P. Jones, and P. Wadler (1993). Glasgow Haskell Compiler: A retrospective. In J. Launchbury and P. Sansom (Eds.), *Functional Programming, Glasgow 1992*, pp. 62–71. Springer-Verlag.

Hammond, K., D. McNally, P. M. Sansom, and P. Trinder (1993). Improving persistent data manipulation for functional language. In J. Launchbury and P. Sansom (Eds.), *Functional Programming, Glasgow 1992*, pp. 72–84. Springer-Verlag.

Holyer, I. (1991). *Functional Programming with Miranda.* Pitman.

Hudak, P., S. L. P. Jones, and P. Wadler, eds. (1992, May). Report on the functional programming language Haskell, version 1.2. *ACM SIGPLAN Notices 27*(5).

Hughes, J. (1989). Why functional programming matters. *The Computer Journal 32*(2), 98–107.

Ichikawa, Y. (1995, September). Database states in lazy functional programming languages: Imperative update and lazy retrieval. In *Proceedings of the Fifth International Workshop on Database Programming Languages*, pp. 150–163. (The final revision is available via `http://www.springer.co.uk/eWiC/DBPL5.html`).

Jacobs, C. E., A. Finkelstein, and D. H. Salesin (1995, August). Fast multiresolution image querying. In *Proceedings of SIGGRAPH 95*, pp. 277–286.

Jones, M. P. (1994a, May). The implementation of Gofer functional programming languages. Technical Report RR-1030, Yale University.

Jones, M. P. (1994b). *Qualified Types: Theory and Practice.* Cambridge University Press, Distinguished Dissertations in Computer Science.

Keene, S. E. (1989). *Object-Oriented Programming in COMMON LISP: A Programmer's Guide to CLOS.* Addison Wesley.

Kirby, G. N. C. (1992). *Reflection and Hyper-Programming in Persistent Programming Systems.* Ph. D. thesis, University of St. Andrews.

Launchbury, J. and S. L. Peyton Jones (1994, June). Lazy functional state threads. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming language Design and Implementation*, pp. 24–35.

Maier, D. and J. Stein (1987). Development and implementation of an object-oriented dbms. In B. Shriver and P. Wegner (Eds.), *Research Directions in Object-Oriented Programming.* MIT Press.

McCarthy, D. R. and U. Dayal (1989, June). The architecture of an active data base management system. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pp. 215–224.

McNally, D. J. (1993). *Models for Persistence in Lazy Functional Programming Systems.* Ph. D. thesis, University of St. Andrews.

McNally, D. J. and A. J. T. Davie (1991, May). Two models for persistence in lazy functional programming systems. *SIGPLAN NOTICES 25*(5), 43–52.

Milner, R., M. Tofte, and R. Harper (1990). *The definition of Standard ML.* The MIT Press.

Moggi, E. (1989, June). Computational lambda-calculus and monads. In *Proceedings of Symposium on Logic in Computer Science*, pp. 14–23.

Nikhil, R. S. (1988). Functional databases, functional languages. In M. P. Atkinson, P. Buneman, and F. Bancilhon (Eds.), *Data Types and Persistence*, pp. 51–68.

Nikhil, R. S. (1990). The semantics of update in a functional database programming language. In F. Bancilhon and P. Buneman (Eds.), *Advances in Database Programming Language*, pp. 403–421.

Ohori, A. (1990). Representing object identity in a pure functional language. In *Proceedings of the Third International Conference on Database Theory*, pp. 41–55. Springer-Verlag.

Paepcke, A. (1988, August). PCLOS: A flexible implementation of CLOS persistence. In S. Gjessing and K. Nygaard (Eds.), *ECOOP'88: Proceedings of the Second European Conference on Object-Oriented Programming*, pp. 374–389. Lecture Notes in Computer Science 322, Springer-Verlag.

Paton, N., R. Cooper, H. Williams, and P. Trinder (1996). *Database Programming Languages.* Prentice Hall.

Paulson, L. C. (1996). *ML for the Working Programmer* (2nd. ed.). Cambridge University Press.

Peterson, J. and K. Hammonad, eds. (1996, May). *Report on the Functional Programming Language Haskell, Version 1.3.* `http://haskell.org`.

Peterson, J. and K. Hammonad, eds. (1997, April). *Report on the Functional Programming Language Haskell, Version 1.4.* `http://haskell.org`.

Peyton Jones, S. L. (1987). *The Implementation of Functional Programming Languages.* Prentice-Hall.

Peyton Jones, S. L. and D. Lester (1992). *Implementing Functional Languages: A Tutorial.* Prentice-Hall.

Peyton Jones, S. L. and P. Wadler (1993, Jan.). Imperative functional programming. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 71–84.

Poulovassilis, A. and P. Kind (1990, March). Extending the functional data model to computational completeness. In *Proceedings of the International Conference on Extending Database Technology*, pp. 75–91. Lecture Notes in Computer Science 416, Sprinter-Verlag.

Poulovassilis, A., S. Reddi, and C. Small (1996, October). A formal semantics for an active functional DBPL. *Journal of Intelligent Information Systems 7*(2), 151–172.

Poulovassilis, A. and C. Small (1996, April). Algebraic query optimization for database programming languages. *VLDB Journal 5*(2), 119–132.

Reddi, A., A. Poulovassilis, and C. Small (1995, September). Extending a functional DBPL with ECA-rules. In T. Sellis (Ed.), *Proceedings of the Second International Workshop on Rules in Databases (RIDS-2)*, pp. 101–115. Lecture Notes in Computer Science 985, Springer-Verlag.

Shipman, D. W. (1981, March). The functional data model and the data language DAPLEX. *ACM Transactions on Database Systems 6*(1), 140–173.

Singhal, V., S. Kakkad, and P. Wilson (1992, September). Texas: An efficient, portable persistent store. In *Proceedings of the Fifth International Workshop on Persistent Object Systems*, pp. 11–33.

Small, C. (1993, December). A functional approach to database updates. *Information Systems 18*(8), 581–595.

Small, C. and A. Poulovassilis (1991, August). An overview of PFL. In *Proceedings of the Third International Workshop on Database Programming Languages*, pp. 96–111. Morgan Kaufmann.

Stonebraker, M., A. Jhingran, J. Goh, and S. Potamianos (1990, May). On rules, procedures, caching and views in data base systems. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pp. 281–290.

Sutton, D. and C. Small (1995, July). Extending functional database languages to update completeness. In C. A. Goble and J. A. Keane (Eds.), *Proceedings of the Thirteenth British National Conference on Databases*, pp. 12–14. Lecture Notes in Computer Science 940, Springer-Verlag.

Teorey, T. J. (1994). *Database Modeling & Design: The Fundamental Principles* (2nd ed.). Morgan Kaufmann.

Trinder, P. (1991). Comprehensions, a query notation for DBPLs. In P. Kanellakis and J. W. Schmidt (Eds.), *Proceedings of the Third International Workshop on Database Programming Languages*, pp. 55–68. Morgan Kaufmann.

Trinder, P. W. (1995). Data dependent concurrency control. In *Functional Programming, Glasgow 1994*, pp. 231–244. Springer-Verlag.

Wadler, P. (1987). List comprehensions. In *Peyton Jones (1987), Chapter 3*, pp. 127–138.

Wadler, P. (1989, January). How to make ad-hoc polymorphism less ad-hoc. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pp. 60–76.

Wadler, P. (1990). *Linear Types Can Change the World!* North Holland.

Wadler, P. (1992a, June). Comprehending monads. *Mathematical Structures in Computing Science 2*(4).

Wadler, P. (1992b, January). The essence of functional programming. In *Proc. ACM Symposium on POPL*, pp. 1–14.

Widom, J. (1996). The Starburst active database rule system. *IEEE Transactions on Knowledge and Data Engineering 8*(4), 593–595.