データベースシステム 104−25
（1995. 7. 19）

非正格関数型言語におけるデータベース更新と遅延検索

市川 哲彦
お茶の水女子大学　理学部

非正格な関数型プログラミング言語 Haskell 向けに，状態モナドによる実行順序制御を用いたデータベース操作インターフェースを実装した．このアプローチの最も単純な実装では，データベース操作が参照透過になる反面，基本処理が状態遷移の過程で行なわれるために，検索の記述が命令的になるという欠点がある．本稿ではトランザクション制御の自然な拡張としての版管理機能を用いることにより，この問題を軽減する試みについて報告する．

Database States in Lazy Functional Programming Languages: Intra-Program Versioning for Imperative Update and Lazy Retrieval

Yoshihiko Ichikawa
Faculty of Science, Ochanomizu University

We propose a database manipulation interface for the statically typed, purely functional programming language, Haskell. Our data model uses surrogates to permit direct update of stored objects, and the basic interface is designed based on the state transformer approach [8], so that the interface is referentially transparent. The state transformer approach requires all the operations to be executed in a single state transition sequence, and thus tends to make queries more imperative than expected. In our approach, to lessen this burden of query construction, versioning is utilized. Versions can be "frozen" or locked, and a set of locked versions can be supplied as an argument to query operations. This intra-program versioning permits on-the-fly dereference while query construction, and allows for straightforward implementation of lazy retrieval in a state thread.

1 はじめに

本稿では静的な型を持つ非正格で純粋な関数型言語を対象としたデータベースインターフェースを提案する．ここでは，標準化されたそのような言語 Haskell [6, 5] を対象としているが，**状態遷移子 (state transformer)** に基づく入出力機構と**クラス機構 (class mechanism)** を備えたものにも同様に適用可能である．

1.1 状態遷移に基づくデータベース操作

非正格な純粋な関数型言語 Haskell の入出力演算は**状態遷移子**によって記述される [8, 4]．これは I/O 状態を受け取り，演算結果と新しい I/O 状態を対にして返す関数で，ファイル操作などを行なう**基本演算**と，基本演算や通常の式から状態遷移子を構成する**組合せ演算子**とを用いて構成される．プログラム自体も状態遷移子として定義され，その実行は与えられた計算機の初期状態から最終状態まで指定された演算子を順次実行していく状態遷移となる．これによりいわゆる副作用を状態遷移の明示的な指定として吸収し，参照透過性を保っている[1]．

筆者は文献 [13] において，状態遷移子に基づくデータベース操作の Haskell 言語への導入について考察を行なっている．この方式の特徴は以下のとおりである：

1. データベースは型毎にグループ化された “識別子 — 値” の対として考える．データベース内におけるデータ参照を通常のポインタによるものではなく，識別子を用いたものとすることで，**永続ルート (persistent root)** [9] 方式に比してデータ更新がより直接的で簡潔になる．

2. データベース演算子は多相的であり，陽に指定された型あるいは型推論によって求められた型から操作対象が決定される．これにより動的な型チェックは不要で，静的な型が利用できる．

3. データ操作は状態遷移子として定義され，単一の状態遷移過程で実行される．そのため更新操作も含めてすべて参照透過である．

[1] ここで，入出力状態の遷移は，あくまで単一の（状態を表す）値がその場で更新されていくものであり，遷移過程が履歴として保存されるのでは無いことに注意されたい．

1.2 版管理の導入による命令性の緩和

上記の第 3 の点は参照透過性を保つために必要な措置であったが，一方で，データ検索式の構成も含めてすべてのデータベース演算が状態遷移と共に行なわれることから 2 つの問題点が発生している．一つは，データベースの検索式が命令的な形式をとり，通常の式の形式とは異なったものになるという点である．ある程度は，**不動点演算子 (fixpoint operator)** の導入で解決しているものの，内包表記や自己参照データ定義による簡潔な検索記述の利用が困難となった．基本的な問題点は，識別子からの値の参照が状態遷移の過程で行なわれ，**非状態遷移性の参照 (on-the-fly dereference)** ができないことに由来している．

また，言語が非正格であるのに対し，状態の遷移に伴って行なわれる基本演算に対しては**超正格評価 (hyperstrict evaluation)** がなされなくてはならないという性質がある．例として，特定の型の**外延 (extension)** データ集合を検索する演算 *allDB* を考えてみよう．このような大量データのフェッチ演算の場合は，文献 [3] で提案されているような *find_first* に続いて *find_next* を要求駆動で評価していくストリーム処理が有効である．しかし，状態遷移子として考える場合は，*allDB* が現在状態に依存する計算であるために，この演算を実行する状態遷移が終った段階で，すべての外延が計算され尽くしていなくてはならない．さもないと，以降の状態遷移で発生する更新演算に伴って検索結果が変わってしまう可能性がある．

本研究で提案する手法では，これらの問題を解決するため文献 [13] の枠組に加え，プログラム実行中の更新履歴（の一部）を反映した版構造を採り入れている[2]．データベース状態はその遷移過程で読みとり専用の施錠することが可能であり，施錠された状態に対する更新操作は新たな版を生成する．この考え方により

1. 非状態遷移性の参照演算子
2. プログラム内部のトランザクション演算子
3. 状態遷移性外延集合検索演算子の遅延評価

が同一の枠組の元で定義・実装可能となる．

[2] ここで言う版は，モデリングにおいて利用される版構造 [7] とは異なっている．つまり，明示的に作成したり，検索したりするものではなく，データベース演算を定義する上での構成要素として，また，実際の実装方式を与えるためのものとして利用しているもので，多版並列実行制御 [2] に近い．

1.3 例外処理への対処

文献 [13] で提案した識別子を用いたデータ関係の表現では，識別子の削除（あるいは実体の削除）に伴う dangling 参照が発生する可能性がある．状態遷移に伴う実行のみの場合は，エラー処理への対処は容易であるが，非状態遷移性演算子を用いる場合，式の中での例外処理の記述は検索式を複雑にする．本論文では，Haskell 言語のクラス機構を利用した解法を提案する．これは，ML [11] が備えているような柔軟な例外処理機構では無いが，例外処理の可能性を示しているものと考えられる．

1.4 関連研究との比較

版管理の考え方は FDBPL [10] でも述べられている．これはトランザクションを版のリストからの新しいデータベースの生成と考えるものであるが，版をどのような基準で選択する可能性があるかを述べるに留まっており，トランザクション制御の演算子や版管理の演算子などは提供されていない．

読みとり専用データを生成するという意味では**配列の凍結 (array freezing)** [8] でも同様の考え方をしている．これは，変更可能な配列からそのコピーを作成し，以後読みとり専用として扱うという点で本手法と良く似ている．しかしながら，本手法での版生成はトランザクション制御機構によってなされている．版の凍結は版の施錠によって行なわれ，版生成は施錠されたデータの更新処理に伴ってシステム側の責任で行なわれる．一方配列の凍結はアルゴリズムの効率などを考慮してプログラマ自身が明示的に行なう処理である．

1.5 本報告の構成

以下，第 2 節では基本的なデータベース操作インターフェースについても説明した後，第 3 節で版概念による非状態遷移性演算子や遅延検索の導入について述べる．続いて，クラス機構を用いた dangling 参照の処理について第 4 節で，実装面第 5 節で簡単に触れる．第 6 節はまとめである．

2 状態遷移子によるデータベース操作

基本的なデータベース操作部は，(1) データベース状態の内部表現，(2) データベース操作を行なう基本演算子，(3) 基本演算や通常の式から状態遷移子を構成する組合せ演算子，から構成される．以下，部品・供給者データベース [1] を例として用いながら，スキーマ定義の方法と操作部を概説し，その操作例を示す．

データベースに収められた型全体を Σ で表し，型 σ の値全体を V_σ，σ 型識別子の全体を I_σ とする．この時，データベース状態は Σ で索引づけられた 3 つ組 $(O_\sigma, s_\sigma, l_\sigma)$ の集まりである．ここで，O_σ は I_σ の有限部分集合，s_σ は O_σ から V_σ への対応を与える 2 項関係，l_σ は s_σ のリスト表現である．

部品・供給者データベースのスキーマを考えてみよう．部品には基本部品 (basic part) と組立部品 (composite part) の 2 種類があり，前者の属性は名前，価格，重さ，供給業者，後者の属性は名前，組立工費，重量増分，利用される部品とその数量である．スキーマは図 1のように定義される．ここで，`DBRef` σ は I_σ の Haskell での表現であり，`Persistent` はデータベースを構成する型のクラス，つまり Σ である[3]．行の始めの '`>`' 記号はプログラムの一部であることを明示するために入れられている．

データベース状態の型を `DBState` とすると，状態遷移子の型は，

```
> type DB a = DBState -> (a, DBState)
```

である．ここで，`a` は型変数で，`DB a` 型の値は，データベース状態を受け取り，`a` 型の操作結果と新しいデータベース状態を対にして返す関数である．$o \mapsto v$ で o から v への対応を表し，$\sigma \in \Sigma$ とすると，基本演算は次の 5 つよりなる:

all_σ	l_σ の検索
$ref_\sigma(o)$	s_σ からの $o \mapsto v$ の検索
$upd_\sigma(o \mapsto v)$	s_σ 中の $o \mapsto w$ を $o \mapsto v$ に更新
$new_\sigma(v)$	未使用の o を用いて $o \mapsto v$ を s_σ に挿入
$del_\sigma(o)$	$o \mapsto v$ を s_σ から削除

これら基本演算は Haskell 言語内部では図 2に示す型宣言をされている．ここで，`Assoc` α β は Haskell の組み込み型で，α から β への対応を表し，その値は `o := v` と書かれる．`(Persistent a) =>` は型変数 `a` が`Persistent` クラスに属する型，つまりデータベース型で具体化されなくてはならないことを表す．

[3] クラスの本来の目的は型毎に振舞の異なる多重定義関数を導入することであるが，ここではデータベース型を指定するために利用している．

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

図 1: 部品・供給者データベースのスキーマ

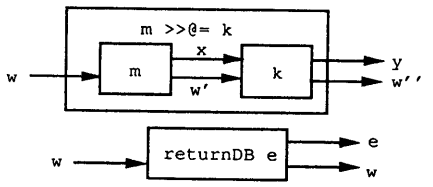
```
> type DBAssoc a = Assoc (DBRef a) a
> allDB :: (Persistent a) =>              DB [DBAssoc a]
> refDB :: (Persistent a) => DBRef a   -> DB (DBAssoc a)
> updDB :: (Persistent a) => DBAssoc a -> DB ()
> newDB :: (Persistent a) => a         -> DB (DBAssoc a)
> delDB :: (Persistent a) => DBRef a   ->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

図 2: データベース操作の基本演算

状態遷移子の組合せ演算子は，入出力に用いられる`>>=`, `>>`, `return` のそれぞれに対応して以下のものを用いる：

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

図式で表現すれば，式 `m >>@= k` と`returnDB e` の表す状態遷移はそれぞれ次のようである



また，式 `m >>@ m'` は `m` と `m'` の順次実行である．

データベース状態遷移子は `transaction` 関数によって実行され，すべての処理が成功した場合にのみコミットする．特に明示的にアボートする場合は，`markAbortDB` をトランザクション中で実行すれば良い．プログラム全体は，入出力状態遷移を定義する `dbMain` 関数として定義され，途中でエラーが発生すれば，含まれる全てのトランザクションは遡って初期状態までアボートされる．

ここで簡単な例を考えて見る．基本部品で価格が１００以上のものを検索するプログラムは図3のように書ける．ここで，`\x -> e` は λ 式 $\lambda x.e$ を表し，`_` はワイルドカードパターンである．基本演算 `allDB` で検索されたリストは，`parts` 変数にバインドされ，`returnDB` 内のリスト内包表記によって必要な検索が行なわれる．更新操作も同様である．例として，供給業者"SUP100"のアドレスを `new_address` に変更する処理を考えると，図4のように書ける．これは `let` 式の `actions` として更新操作のリストを生成し，それを組み込み関数 `foldr` を用いて実行している．

より複雑な問い合わせ，例えば，組立部品のトータルの価格と重さ，などもこの枠組で検索できる．また不動点演算子を用いればメモを利用した効率的な検索も記述可能ではある [13]．しかし，問い合わせの表現は通常の Haskell 言語の記法とは異なり非常に命令的である．次節では版の導入により，より自然な記法でそれらの検索が書けることを示す．

3 版機構の利用

これまでデータベース状態はその場で更新されるものとして扱ってきた．しかしながら，検索が命令的になること，外延集合検索において遅延評価ができないという問題がある．そこで本手法では，更新履歴（の一部）を記録した版の木がデータベース状態の一部として保持されているものと

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

図 3: 価格が１００以上の基本部品

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

図 4: 業者の住所変更

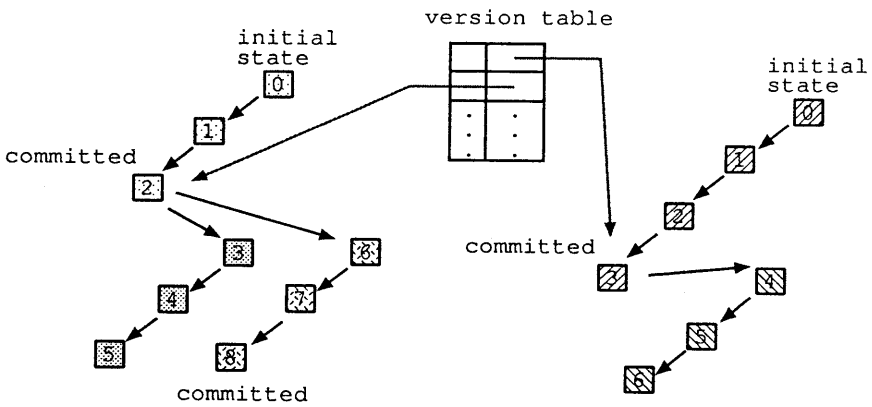


図 5: 版の木構造

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

図 6: 版表に対する基本演算

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

図 7: 部品の価格と重さを計算する関数

考える (図 5). 図中, 四角は版を, 中の数字は版番号を表し, また, 同じパターンを背景に持つ版は同じトランザクションで生成されている. なおトランザクションはロールバックされる可能性があるため, 版の関係は一般に木構造となっている.

ユーザが版に関して利用できる基本演算には getDB と restoreDB の 2 つがある. 前者は入出力状態の遷移過程において "現在の" 版集合に読み取り専用の施錠をし, その版集合を版表として返す (図 5). 版表の型は Database である. また, 逆の演算として restoreDB があり, これは引数で指定した版表中の版を "現在の" 版とする. 施錠された版集合, つまり版表に対する基本演算は 図 2に示す 2 つである. その意味は型と名前が示すとおり, allDB と refDB にそれぞれ相当する. またデータベース状態の一部として初期状態の版からなる版表を用意することが可能であるから, それを init_vt で表すと, 初期状態の検索演算は

```
> allOf = vAllOf init_vt
> valOf = vValOf init_vt
```

と書くことができ, 検索のみのプログラムは全て非状態遷移性の参照で, しかも版表を明示的に用いることなく記述することが可能となる.

再度, 組立部品のトータルの価格・重さを検索することを考えよう. これは図 7 のように記述できる. プログラム自体は通常の Haskell のものとほとんど変わりがなく, 違うのは valOf によって識別子から値を参照していることだけである.

版を用いることで, 外延集合検索は遅延評価可能になる. vAllOf 施錠された版を対象とした検索であるから, 当然遅延評価は可能であるが, 状態遷移に伴って実行される allDB の処理も次のようにしてして遅延評価できる :

```
> allDB = createCursor >>@= \cursor ->
>         returnDB (findAll cursor)
>   where
>     findAll c
>       = case findNext c of
>           Just (v, c') -> v: findAll c'
>           Nothing      -> []
```

ここで, createCursor は操作対象の版に施錠をした上で, 外延を操作するためのカーソル構造を生成する. findNext はそれを利用して, データと残りを検索するためのカーソル構造を返す. ここで上記のプログラムはあくまで説明のためのものであることに注意されたい. カーソル構造は線形に用いられているため (つまりカーソルへのポインタは高々一つであるため), 実際の実装では単一データのその場書き換えを用いてプログラム可能である.

トランザクション処理も版概念で記述される.

つまり，最初に getDB で版表を作成しておき，エラーが発生した場合には restoreDB で元に戻せば良い．

最後に版の生成と削除に関してまとめておく．新しい版は施錠された版に対する更新操作がなされた時にだけ生成すればよい．施錠されていない版は直接更新される．版表やカーソル構造がゴミになった場合には，ゴミ集めの過程で，含まれる版が解錠される．この時，他の版表やカーソルからの施錠がなく“現在の”版でもない場合はその版自体は削除することができる．また，restoreDB で“現在の”版が v から v' へ変化する場合，もしも v が施錠されてなければ，v は削除できる．

4 dangling 参照の処理

ここで利用しているデータモデルでは，“識別－値”の対は明示的に削除することが可能であるため，dangling 参照が発生する可能性がある．状態遷移に沿った操作のみを利用する時には，処理は困難では無い．しかし，非状態遷移性参照を行なう時には，エラーがいつ発生するかはわからないため，**例外処理 (exception handling)** [11] が必要になる．Haskell 言語自身は例外処理機構を持たないが，データベース型が全て Persistent クラスのインスタンスであることと，ここで取り扱いたい例外があくまでデータベース操作に対応するものであることから，例外処理関数を Persistent クラスの多重定義関数としておくことで，ある程度の対処が可能である：

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

これは whenDangle 関数のデフォルトを指定するもので，dangling 参照が発生すると即座にプログラムを停止させる．

これに対し各データベース型で個別の処理を与えるのであれば，インスタンス宣言のところで，

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

のようにして適当な値や処理を入れるようにしておけば良い．

なお，この方法では例外処理はスキーマ定義に伴って定義され，アプリケーション毎の定義ができないという点でまだ不十分である．

5 実装に関して

実装は Glasgow 大学で開発された Haskell コンパイラと C 言語を用いて行なっている．この節では，実装に関して概略を説明する．

格納データの表現

ヒープ中のデータはその文字列表現を格納している．データと文字列表現との相互変換は Text クラスの演算 red と show を用いて行なわれるので，データベース型は Text クラスのインスタンスで無くてはならないという制限が発生している．また，このため，関数，サイクルを含むデータなどはデータベースに入れられない[4]．

型情報の計算

明示された，あるいは推論された型から基本演算の対象が決定されるため，実行時に型情報が計算できなくてはならない．そのため，モジュール名と型名を与えるために

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

というクラスを導入する．ここで，typeRepr 関数は型 a の表現を与えるもので，従って，データベース型はこのクラスのインスタンスで無くてはならない．実際に実装で使われたスキーマは次のように宣言されている[5]：

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

なお，この枠組は多相型にも対応している．例えば，Tree a なる型があれとすると，同じスキーマ定義であっても，Tree Int や Tree String 型の外延などは別なものとして扱われる．

版管理

“識別子－値” 対の集合は索引付ファイルで記憶する．現在の実装では，新しい版の生成は索

[4] 関数型は Text クラスのインスタンスにできるが，show の結果は<<function>> などの文字列でヒープ中の内容が表現できるわけではない．

[5] deriving 節はコンパイル時のインスタンス自動生成を指定する．本来は組み込みクラスに関してのみ適用できるものなので，コンパイラを変更して，Representable に関しても可能にした．

引部のコピーによって行ない，また，データ部はプログラム実行中は単一ものがヒープ的に利用され，プログラム全体のコミット操作時にゴミ集めがなされる．また，各版には施錠の数が記録されており，施錠と解錠はカウンタの増減によってなされる．

6 まとめ

状態遷移を利用したデータベース操作インターフェースを Haskell 言語を対象として実装した．基本的な考え方は [13] で既に提案したものと同じであるが，問い合わせ記述が命令的になってしまうという問題点を，新たに，版概念を利用することで解決した．この方法では，非状態遷移性の検索，状態遷移性外延集合検索の遅延評価，トランザクション処理，が全て説明でき，かつ直接的な実装方法を与えることが可能である．

データの格納方式など実装面での問題もあるが，それに加えていくつかの問題が未解決である．まず，版集合をプログラマーが操作可能であるため，この柔軟性の向上に伴ってバグの可能性が増してしまう．初期状態に関しては問題はないが，更新・検索が入り組んだプログラムの場合では版集合パラメタを隠蔽する *read-only monad* [12] などの手段が必要である．次に，本手法ではデータベース操作は全て型毎にまとめられているが，モデリング能力と言う点では一つの型の値が複数の役割を持ってくる可能性があり，その意味ではモデルが粗過ぎるとも言える．また，多相型の値は原理的にデータベースに入れることができないという問題点もある．つまり，多相型の関数は（たとえ物理的にデータベースに記憶する手段を用意しても），型変数を全て具体化させた形でしかデータベースに格納することができない．これらは本手法が持つ本質的な問題点であり，記憶域管理手法や実行時の型情報の取り扱いなどと併せて更なる研究が必要である．

謝辞

本研究に関して有益な助言を与えて下さった藤代一成氏に感謝いたします．

参考文献

1) M. P. Atkinson and P. Buneman. Types and persistence in database programming langaues. *ACM Comput. Surv.*, 19(2):105–190, Jun. 1987.

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

3) P. Buneman, R. E. Frankel, and R. Nikhil. An implementation technique for database query languages. *ACM Trans. Database Syst.*, 7(2):164–186, 1982.

4) Haskell Commitee. The definition of monadic I/O for Haskell 1.3. *http://www.dcs.gla.ac.uk-/~kh/Haskell1.3/IO.html*, Dec 1994.

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

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

7) W. Kim. *Introduction to Object-Oriented Databases.* MIT Press, 1990.

8) J. Launchbury and S. L. Peyton Jones. Lazy functional state threads. In *Proc. of ACM SIGPLAN'94 Conf. on PLDI*, pp. 24–35, Jun. 1994.

9) D. J. McNally. *Models for Persistence in Lazy Functional Programming Systems.* PhD thesis, Unversity of St. Andrews, 1993.

10) R. S. Nikhil. 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, 1990.

11) R. Stansifer. *ML Primer.* Prentice-Hall, 1992.

12) P. Wadler. The essence of functional programming. In *Proc. of the ACM Symposium on POPL*, pp. 1–14, Jan. 1992.

13) 市川 哲彦. プログラミング言語 Haskell 上のデータベース操作インターフェースの実装. 情報処理学会研究報告, 94(62):201–208, 1994.