データベースシステム 109－8
（1996．7．24）

非正格関数型言語におけるデータベース実現値の永続性モデル

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 を対象としたデータベース操作インターフェースの永続性モデルについて報告する。先行研究 [3] では、状態遷移による参照透過なデータベース操作、クラス機構による永続性と例外処理の指定、内部版管理による on-the-fly データベースアクセス、といった特徴を有しているが、永続性モデルとしては型外延モデルを用いていたため、モデリング能力やプログラミングの面での柔軟性に欠けていた。そこで、本研究では、より柔軟な永続性モデルである到達可能性モデルの先に提案した手法への適用を試みた。提案手法では、永続変数は用いず、型によって永続ルートの識別をしており、これにより静的な型づけと参照透過性の維持を可能にしている。また、型クラスの機能を用いることにより、型外延モデルと同様の永続性管理が行なえると同時に、ビューに相当する永続ルートの利用も可能である。

Database States in Lazy Functional Programming Languages: Incorporating the Reachability Model of Persistence by the Class Mechanism.

Yoshihiko Ichikawa
Faculty of Science, Ochanomizu University

This report proposes a database manipulation interface for the statically typed, purely functional programming language, Haskell. While our previous work described in [3] has desirable properties such as referential transparency, static typability, and accommodation of imperative update to lazy retrieval, the persistency can be specified only on a type extent basis. In contrast, the other typical persistency model, the reachability model of persistence, gives us more flexibility in database modeling and also is suitable for functional computation. The approach proposed here incorporate the reachability model into the previously proposed one through the type class mechanism of Haskell without sacrificing the basic properties. Instead of using persistent variables to specify persistency roots, the proposed approach uses types to identify the roots, so that the referential transparency is retained. This approach also allows us to simulate the type extent model of persistence declaratively. Moreover, to facilitate view-like usage of persistency roots, a persistency root having a derived value can be easily specified.

1 はじめに

静的な型付けを行ない非正格な評価を行なう純粋な関数型言語を対象としたデータベース操作インタフェースについて報告を行なう。基本的ターゲットは Haskell 言語 [6] であるが、その他の言語であっても、状態遷移に基づく入出力とクラス機構を有していれば適用可能である。

純粋な関数型言語からのデータベース操作は宣言的であること、数学的な性質が解析しやすいこと、さらに多相型，高階関数，内包表記による簡潔な記述が可能であるなどの望ましい性質を持つ反面、数学的な等式推論を保証する参照透過性があるために、逆にデータベース更新の取り扱いは難しくなっている。そこで、筆者は文献 [3] において、状態遷移に基づく参照透過なデータベース操作体系の実現と、プログラム内版管理に基づく命令的更新と遅延検索との混合を可能にする手法を提案した。しかしながら、この手法では永続性は型外延単位で指定する方式 (**型外延モデル**) を用いていた。永続性を指定された型についてはその型の値の集合 (型外延) が存在すると考え、データ作成時に自動的に型外延への挿入が行なわれる。

このモデルは従来の関数型データモデル ([2] など) の考え方に従って採用したものである。しかし、利用者がデータの型外延への挿入をしなくても良いという利点がある反面、実世界のモデリングにおいて柔軟性に欠ける。また関数型プログラミングという側面から見た場合には、永続型の一時的な値が使えない、型外延は一つであるため役割や属性で別々な集合が構成できない、型外延の値としてユーザ定義のデータ構造を用いることが難しい、通常の関数型プログラミングと違って明示的なデータ削除が必要、という問題点が挙げられる。

永続性のモデルとしては上記の型外延モデルの他に**到達可能性モデル** がある。このモデルではあらかじめ指定された変数などから到達可能なすべてのデータが永続性を持つ。到達可能性判断の基準になるデータは永続ルートと呼ばれる。永続ルートの指定には、永続ストリーム、永続環境、永続変数が知られている。この手法では型外延モデルのような問題が生じないという利点があるが、参照透過で静的な型付けの言語においては、永続ルートの指定は必ずしも自明ではない。実際、永続環境や永続変数では、実行時に名前束縛の環境や名前自身の値が変わってしまうため参照透過性を損なってしまうし、また、永続ストリームでは動的な型チェックが必要となる。

この点に加え、到達可能性モデルでは永続性管理が面倒になるため、型外延モデル的な使い方に対応可能かという点も重要である。データ挿入用の関数を必要に応じて作成するということも可能であるが、利用者にとって繁雑であるし、また安全面でも問題がある。

永続ルートを適宜用いると、外延集合の部分集合を複数作成できるので、ビューに相当する永続ルートを管理することも原理的には可能である。しかし、実行時に値を導出するわけでないため、更新時の処理コストが大きく、また、空間効率も良くない。したがって、導出値を用いる本来のビュー的な永続ルートも選択的に指定できることが望ましい。

これらの点に鑑み、本報告では文献 [3] の基本的なアプローチを到達可能性モデルに適合させる試みについて報告する。以下、アプローチの概略を述べた後、データベース操作インターフェース、型外延モデルへの適用、ビュー永続ルートについて説明をする。また、本方式では永続ルートとしては高階の型は扱えるが多相型は扱えない。この点についても簡単に言及する。

2 基本方針 — 型クラスの利用

本報告で提案する手法では、Haskell 言語の型クラス機能 を用いてデータベース状態や、永続ルートの管理を行っている。本小節では、型クラスに関して簡単に触れた後、実際にどのようにして型クラスが用いられているかを説明する。

型クラスは型の集まりであり、クラスに属する型はインスタンスと呼ばれる。クラスにはそれぞれクラス演算と呼ばれる関数が関係付けられている。例えば Eq クラスはクラス演算として、等値演算 (==) と非等値演算 (/=) を持ち、インスタンスはぞれぞれの型に応じた== や /= の定義を持つ。このようなインスタンス毎のクラス演算の実装はインスタンスメソッドと呼ばれる。Eq クラスのインスタンスには Int や Bool があるが、等値比較はそれぞれ異なっており、演算子の型に応じて自動的にインスタンスメソッドが選択される。その他のクラスとしては、文字列形式への変換が可能な型のクラス Show や、算術演算 +、-、* が使える型のクラス Num がある。

このようにクラス機構はインスタンスとなる型の性質を表現するのに適したメカニズムである。この考え方を用い、ここでは永続ルートとなりえ

る型やデータベース中で識別子を持ち得る型などを型クラスを用いて記述する。より具体的には以下のような目的で型クラスを用いている。

データベース型の指定 データは直接更新が可能なように (識別子, 値) 対の形で格納される。この時データベース中の (識別子, 値) 対は、トランザクション管理や版管理の対象として扱われるため、単純な変更可能変数 [5] とは、異なった操作が必要である。このことを明示するために `DBType` クラスを導入し、このインスタンスのみをデータベース型として扱う。また、後述する永続ルートの自動管理、一貫性制約の明示などの目的でもこのクラスを用いる。

データベース操作の組合せ演算子を与える Haskell の入出力演算は**状態遷移子**によって記述される。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。これと同様データベース操作は基本演算と組合せ演算子から構成される。ここでは Haskell-1.3 から新しく導入された型構成子クラス [4] を用いて組み合わせ演算子を記述している。

永続ルートの指定 本研究では永続ルートは型によって識別されるという考え方をする。指定された各々の型のただ一つの値がその型で識別される永続ルートとなる。この方式を実現するために、永続ルートを構成する型については `PerRoot` クラスのインスタンスとする。また、同じクラスのインスタンスメソッドの宣言を通して、ビューに相当する永続ルートの取り扱いにも宣言的に対応可能である。

3 状態遷移に基づくデータベース操作と永続ルートの指定

以下では、部品・供給者データベース [1] を例としてスキーマ定義と操作体系を説明する。

抽象的にはスキーマは三つ組 (Γ, Σ, Φ) である。Γ は永続ルートを識別するためのラベル、Σ は識別子を持ち得るデータベース型の集合、Φ は Γ から永続ルートの型への写像である。$\sigma\ (\in \Sigma)$ 型の値を V_σ で、また、$\sigma\ (\in \Sigma)$ 型の識別子を I_σ で表すとする。データベースの実現値は $\iota \in I_\sigma$ から $v \in V_\sigma$ への有限マップを与える M と、各タグ $\tau \in \Gamma$ からその値へのマップを与える R から構成される。

3.1 基本的なデータベース操作

この部分の詳細は文献 [3] で論じているので、ここでは概略の説明と、スキーマ定義と基本操作を中心に説明する。まず、部品·供給者データベースのスキーマを考える。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、`insntance DBType` σ は σ が Σ の要素であることを宣言している。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。また、$[a]$ は要素が a 型のリスト型である。

データベース型に関する基本演算には

$new(v)$	未使用の o について $M(o)$ を v とする
$ref(o)$	$M(o)$ の検索
$upd(o,v)$	$M(o)$ を v に更新

があり、Haskell 言語の関数としては次のように宣言されている：

```
> newDB:: DBType a => a  -> DB (DBRef a)
> refDB:: DBType a => DBRef a -> DB a
> updDB:: DBType a => DBRef a -> a -> DB ()
```

ここで、`DBType a =>` は型変数 `a` の値として、`DBType` クラスのインスタンスのみを取ることを示している。また `DB` は状態遷移の型である。データベース状態の型を `DBState` とすれば、

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

と表される[1]。つまり、状態を受けとって、計算結果と新しい状態を返す関数となっている。

データベース演算はこのような状態を次々と受け渡しながら変化させる状態遷移子であり、組合せ演算子 `>>=`、`return` を用いて複雑な演算が構成される。これは型構成子クラスを用いた形で実現されているので、様々な状態遷移の取り扱いにも適用可能であるが、敢えてデータベース操作に特化して書くと

[1] 実際にはエラー処理が含まれるのもう少し複雑な型をしている。

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

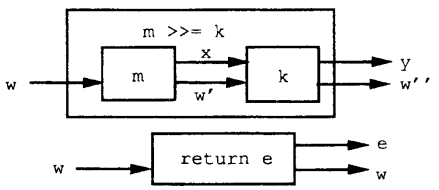
図 1: 部品・供給者データベースのスキーマ

```
> expensiveBasicParts partIds
>  = getDB >>= \db ->
>    let parts = map (vValOf db) partIds
>    in return [ name | Basic name cost _ _ _ <- parts, cost >= 100 ])
> updateAddress sid newAddress
>  = refDB sid >>= \(Supplier name address supplies) ->
>    updDB sid (Supplier name newAddress supplies) )
```

図 2: 基本的な操作の例: 価格が 100 以上の基本部品名を取り出すフィルタと業者の住所変更関数

```
> (>>=) :: DB a -> (a -> DB b) -> DB b
> return :: a -> DB a
```

のように宣言されているものとして扱われる。図式的には各々次のように表される。



この考え方だけでは操作が命令的になってしまうので、データベース状態を適宜取り出して、状態遷移とは別に on-the-fly で用いることも可能である。基本演算は

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

であり、それぞれデータベース状態の取得と状態の復旧を行なう。また refDB に対応する演算に

```
> vValOf :: DBType a =>
>           Database -> DBRef a -> a
```

があり、指定されたデータベース状態からの on-the-fly での値参照ができる。

図 2 に操作例を示す。ここで、`\x -> e` は λ 式 $\lambda x.e$ を表す。最初の例は部品の識別子のリストから、価格が 100 以上の基本部品の名前を検索する操作であり、on-the-fly での値参照を用いている。次の例は与えられた業者の住所を変更する関数である。

```
> data PartExt = PartExt [DBRef Parts]
> instance PerRoot PartExt where
>     initValue = return (PartExt [])
> data SuppExt = SuppExt [DBRef Parts]
> instance PerRoot SuppExt where
>     initValue = return (SuppExt [])
```

図 3: 永続ルートの指定

3.2 永続ルートの操作

本アプローチでは永続ルートはルートとなる型によって指定されると考える。さらに、その型自身がルートの識別子として利用されると考える。従って、R は各々の永続ルートの型に対して、その型の値を対応させる写像である。本方式ではクラス PerRoot を用い、そのインスタンス全体が Γ を構成すると考える。例として、部品・供給者データベースにおける永続ルート指定の様子を図 3 に示す。この図では、部品の永続ルートになる PartExt と、供給業者の永続ルートになる SuppExt が指定されている。initValue メソッドの指定はデータベース中での永続ルートの初期値を与えている。永続ルートの操作は、状態遷移性の基本操作 2 つと、on-the-fly でのルート参照のための関数によってなされる:

```
> getRoot::PerRoot a=> DB a
> setRoot::PerRoot a=> a -> DB ()
> vRootOf::PerRoot a=> Database -> a
```

```
> instance DBType Part where
>  atNew pid =
>    getRoot >>= \(PartExt pids) ->
>    setRoot (PartExt (pid:pids))
>  beforeUpdate pid newValue =
>      ... 部品間と部品・供給者間の ...
>      ... 参照構造の整合性維持       ...
```

図 4: 自動的な型外延管理

したがって、図 2 に示したフィルタは実際には次のように利用することができる:

```
>   transaction (
>       getRoot >>= \(PartExt partIds) ->
>       expensiveBasicParts partIds)
```

ここで getRoot は多相関数であり、指定された型 (ここでは PartEXt) のルートを与える。また、transaction 関数は、データベース処理の入出力処理へのマップを行なうと同時に atomicity の保証をする関数である。

4 自動的な型外延管理と一貫性制約チェック

上記の永続ルートを用いることで、型外延モデルが実現可能である。最も単純な方法は、型外延管理用に専用の関数を用意し、それを常に利用するものである。しかし、呼びだし形式が異なって不便であるし、また安全面でも問題がある。そこで、データベース型を指定する DBType クラスのクラス演算として、データ生成あるいは更新をトリガーとして自動的に実行されるアクションを記述する方式を採用した。例として、Part の生成に当たって自動的に PartExt のリストにデータを追加することを考える (図4)。この宣言では単に Part を DBType クラスのインスタンスにするだけではなく、atNew メソッドとして PartExt への追加が指定されている。atNew メソッドは newDB で新たな識別子が生成された時に、自動的に実行される。引数は新しく生成された識別子である。

他にも beforeUpdate、aftrerUpdate クラス演算があり、それぞれ値の updDB による更新の前後で自動的に実行され、条件チェックや、相互参照の更新などに使用可能である。図 4 では詳細は省略したが部品間の使用関係の更新や部品・供給者間の供給関係の更新をインスタンスメソッドとして指定している。これにより、専用の関数を別途作成して行なう方式に比べて宣言性と安全性が高められている。

```
> data BasicExt = BasicExt [DBRef Parts]
> instance PerRoot BasicExt where
>   initValue =
>    getDB >>= \db ->
>    let
>      PartExt pids = vRootOf db
>      ps = zip pids (map (vValOf db) pids)
>      bs = [p|(p, Basic _ _ _ _ _)<-ps]
>    in return (BasicExt bs)
>  isView _  = True
```

図 5: ビュールートの指定

5 ビュー永続ルート

図 3 で示した例では永続ルートの初期値計算は、単に空のデータを返している。実際には、ここで任意のデータベース状態依存の処理を指定できるので、ビューとして利用できる。例として基本部品のビューを考えてみる (図5)。新たに永続ルート BasicExt が定義されており、初期値指定で PartExt の値から基本部品のみを検索している。また、isView の定義はビューであることの指定で、誤って setRoot で更新しないことを保証している。

6 多相型の PerRoot インスタンス

多相型の PerRoot インスタンスには注意が必要である。例として instance PerRoot (a->b) を考えてみる。これは正しい宣言だが、より具体的な関数も同じインスタンスメソッドが利用され、例えば $Int \to Int$ と $String \to Int$ は区別されない。そのため、$Int \to Int$ 型の値を格納しておいて、それを $String \to Int$ 型として使っても型エラーにならない[2] 。従って、永続ルートのデータタイプを実行時に調査し、型チェックの問題をルートの有無の問題に置き換えることが必要である。動的な型チェックを導入する方法も考えられるが、言語仕様と処理系の大幅な変更が必要である。

[2] 型クラスでは型を永続ルートタグとして使うには荒い分類しか与えないことを意味しており、Haskell が静的な型チェックをする言語であることとは矛盾しない。

ここではより単純な手法として、永続ルートの型を型変数を含まないグラウンド型に限定する方法を採用している。この影響でデータベース中には多相型の値は格納できない[3]。この制約は、新たに Ground クラスを導入し、PerRoot のインスタンスを Ground のインスタンスに制限することによって実現できる。Ground 型のインスタンスは Int などの基本的な型から構成される変数を含まない型に限定され、自動的に多相型の永続ルートは排除される。

しかしながら、この制約の中においても次のような宣言は有効である：

```
> data Fun a = Fun (a->a)
>       deriving Ground
> instance PerRoot (Fun a)
```

ここで deriving 節はインスタンスメソッドの自動導出を指定しており、Ground を新たに既定義クラスとして加えることで実現できる。deriving 節の制約は再帰的に適用されるため、a が Ground である場合にのみ、Fun a も Ground となる[4]。この例の場合は、Fun Int や Fun String などの永続ルートは構築可能であるが、一般の Fun a という形の永続ルートは存在しえない。

7 まとめ

純粋な関数型で非正格な言語上で、到達可能性モデルに基づく操作体系の導入を行なった。型クラスを用いた手法により、静的な型付けを壊すことなく、かつ永続変数のような参照透過性を破壊する構造を入れることなく実現可能である。また単なる永続ルート方式であるだけではなく、クラスメソッドとしてデータベース中のデータ生成や更新をタイミングとしたデータベース操作を記述することで、型外延永続性モデルへの適応や一貫性制約の維持に宣言的に対応できること、および、ビューに相当する永続ルートの実現も可能であることを示した。

最後に示した多相型永続ルートの禁止は、型によるルート識別によるデメリットである。データの構造面でのモデリングを考えるのであればそれほど大きな支障はないとも言えるが、対象の振舞いのモデリングや、操作環境の動的な変更などが要求される場合には問題となり得る。

このような型システムの問題に加え、実装面においても到達可能性モデルの効率の良い実装が今後の課題である[5]。

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.*, Vol. 19, No. 2, pp. 105–190, Jun. 1987.

[2] P. Buneman, R. E. Frankel, and R. Nikhil. An implementation technique for database query languages. *ACM Trans. on Database Syst.*, Vol. 7, No. 2, pp. 164–186, 1982.

[3] Y. Ichikawa. Database states in lazy functional programming languages: Imperative update and lazy retrieval. *Proc. of the 5th Intl. Workshop on DBPL*, pp. 150–163, Sep. 1995.

[4] M. P. Jones. *Qualified Types: Theory and Practice.* Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.

[5] 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.

[6] J. Peterson and K. Hammond, eds. Report on the functional programming language Haskell, version 1.3. *Yale University Research Report YALEU / DCS / RR-1106*, May 1995.

[3] 可能ではあるが永続ルートから到達可能にできない

[4] 通常の Haskell 言語の deriving のセマンティックスはより制限がきつくなっているので、型チェックの部分は幾分変更が必要である。

[5] 現在は Haskell 解釈系 Hugs1.01 の記憶構造をファイルに mmap() でマップして永続性を実現し、mark-scan 型ガーベッジ・コレクタによって到達可能性モデルを実現している。現在は操作関数などのテスト段階である。