データベースシステム 99−27
（1994. 7. 22）

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イスの実装

市川 哲彦
お茶の水女子大学　理学部

純粋な関数型プログラミング言語 Haskell 向けに，monad を用いた実行順序制御と多重定義関数を基本操作とするデータベース操作インターフェースを設計・実装した．データベース操作はすべて monad によって制御されているので，参照透過である．また，基本操作を `Persistent` class に関連付けられた多重定義関数として定義することにより，実装の詳細を隠蔽するデータ独立性を実現し，多様関数と静的型付けの利用とを可能にしている．

Implementation of a simple database manipulation interface for the Haskell programming language

Yoshihiko Ichikawa
Faculty of Science, Ochanomizu University

In this report, we describe the implementation of a simple database manipulation scheme for non-strict purely functional programming language Haskell. Evaluation of database actions is controlled in a monadic way. Hence, all database manipulations are referentially transparent. The primitive operations are defined as overloaded functions associated with a newly introduced class `Persistent`. The class hides all the implementation details to achieve data independence, and utilizes polymorphic functions and static type checking, which are the major features of the language.

1 はじめに

これまでに種々の関数型データベースプログラミング言語の研究・開発が行なわれてきた．これらの中で参照透過 [7] な更新機構を持っているものとしては FDBPL [14], Staple [13], PFL [16] などがあるが，対話型データベース言語や最適化のための中間言語としての位置付けではなく，大規模なプログラムの作成までを含めたプログラミングシステムとして考えた場合には，参照透過性の維持は必要であると考えられる．

純粋な関数型言語は参照透過という点で理論的にきれいであるが，一方で入出力操作など外部のシステム状態を扱う機能が手続き的な言語に比べて劣っていた．しかし，近年の研究では，参照透過な入出力機構が開発されている [8] [18].

本研究では，1992 年に標準化がなされた純粋な関数型言語である Haskell [9] にデータベース操作の体系を取り込むことを試みた．Haskell 言語の場合は，対話 (dialogue) による入出力が標準的な言語仕様に入っており，また，現在知られている処理系 (Yale, Chalmers, Glasgow 各大学による) はいずれも継続 (continuation) と monad による入出力機構も提供している．これら参照透過な入出力機構を参考にしてデータベース操作体系を組み込むことで，参照透過な DB 操作の導入が可能になると考えられる．

本稿では，Glasgow 大学で開発された Haskell コンパイラをベースとした，データベース操作インターフェースの実装について報告する．まず，Haskell 言語の簡単な紹介を行い，続いて，データベース操作とその実装，簡単なデータベース操作例について説明する．次に，関連する研究として [8], [13], [16] との比較をした後，まとめを行なう．

2 Haskell 言語に関して

この節では以降の説明において必要な Haskell 言語 の代数型，monad 入出力機構，class 機構について説明をする．

2.1 代数型

データ型には Int, Float, Char などの基本的な型と複合型があり，複合型には タプル，リスト，関数型，代数型 (algebraic type) がある．代数型の例として部品データ [1] 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．この場合，部品を表す代数型 Part は次のように書ける：

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

ここで，Supplier は別な代数型で，供給業者の名前と住所，および供給部品を表している．この宣言により，型構成子 Part および，Supplier が定義され，また，データ構成子 Basic, Composite, Supplier が定義される．

2.2 入出力

ここでは本研究において重要な monad による入出力について説明をする．monad の詳細については [18] [15] を参照されたい．入出力の monad は (IO, thenIO, returnIO) なる 3 つ組で，IO は多様型，thenIO, returnIO は多様関数である：

```
type IO a = IoWorld -> (a, IoWorld)
thenIO   :: IO a -> (a -> IO b) -> IO b
returnIO :: a -> IO a
```

ここで，a および b は型変数である．IO a 型の値は，システム状態を表す IoWorld 型の値をもらって，操作結果と新しい状態を対にして返す関数である．このように状態を陽に制御することから参照透過となる．テキストファイルを読みとる入出力動作を考えると，これは文字列を返す動作であるから IO [Char] なる型を持つ．具体的には，関数

```
readFileIO :: [Char] -> IO [Char]
```

がこの動作を作る．この関数はファイル名をもらい，ファイル内容を結果とする入出力動作を返す．

thenIO は入出力動作の組合せを行なう．式 “thenIO m k” の表す動作を図式的に表すと：

$$w_0 \xrightarrow{m} (x_1, w_1)\ ;\ w_1 \xrightarrow{k\ x_1} (x_2, w_2)$$

つまり，m が状態を遷移させ，次に，m の結果を受けた k が状態を遷移させる．

returnIO は任意の式から入出力動作を作る関数で，returnIO e が表す入出力動作は，図式的には次のようになる：

$$w \longrightarrow (e, w)$$

つまり，状態遷移はさせず操作結果だけを作りだす．

プログラムの実行とは，関数 mainIO :: IO () のシステム状態への適用である．ここで，() は値 () のみを要素とする型である．例として，ファイルを読み取り，各行を reverse するプログラムを考える：

```
mainIO
 = readFileIO "person.dat" `thenIO`\cont ->
```

```
returnIO (lines cont)   `thenIO`\in ->
returnIO (map reverse in) `thenIO` \out ->
appendChanIO stdout (concat out)
```

ここで，`\x -> e` は λ 式 $\lambda.e$ を表している．また，`thenIO` は `infixr 9 `thenIO`` と宣言されており，`` `thenIO` `` 演算子は，λ 式や関数適用よりも優先順位の低い右結合中置き演算子として扱われる．また，`reverse` はリストを逆にする関数，`map` はリストのすべての要素に渡された関数を適用する関数である．この動作を stepwise に見ると次のようである：

(1) `readFileIO` でファイルの内容を読みとる；
(2) `lines` 関数で行のリストにばらす；
(3) `reverse` を `map` 関数で各行に適用する；
(4) `concat` 関数で行をつなぎ合わせたリスト作成し，`appendChanIO` 関数で出力する．

2.3 Class 機構

`+`, `-`, `*`, `/` などの多重定義関数は，class 機構 [17] によって制御されている．ここで，class は幾つかの多重定義関数に関連づけられた型の集合である．例として，多重定義関数 `==` および `/=` に関連づけられた class Eq について説明する．宣言は次のようである：

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

最初の2行は Eq が多重定義関数 (==) と (/=) に関連付けられていることを示し，3行目は `/=` 演算子のデフォルトの振舞いを宣言している．これらの多重定義関数は class 演算と呼ばれる．class instance は

```
instance Eq Int where
    (==) = primEqInt
instance Eq Float where
    (==) = primEqFloat
```

などと宣言される．最初の宣言は，Int が Eq の instance であることと，Int 用の `==` 演算子 (method と呼ばれる) が `primEqInt` であることを示している．どの method が利用されるかは式の型に合わせて自動的に選択される．式 "`x == y`" は，引数の型が Int 型であれば "`primEqInt x y`" に等しく，Float ならば "`primEqFloat x y`" に等しい．

あらかじめ定められた幾つかの class については，instance 宣言の自動的な導出が可能である．これは代数型の宣言時に行なう：

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

これにより，Part 型は自動的に Eq class の instance となり，多重定義関数が利用可能となる．

3 データベース状態と基本操作

関数型データモデルでは，データベースはデータと記号の束縛環境として扱われる．記号に束縛されるデータは，変更できないものかまたは永続性変数である．データベースの更新は，前者では，束縛環境の変更 [6] または新しい束縛環境の生成 [14] として，後者では永続性変数への代入 [2] として定義される．

しかしながら，Haskell 言語ではこれらの枠組は利用できない．これは，(1) 変数への代入が無い；(2) 記号の束縛環境をデータとして扱でない；(3) 静的な型付けをする言語である；という理由からである．何らかの形で "変わり得るもの" を実現するためには，変数の集まりとして定義されるデータベース状態を取り扱えるようにしなくてはならない．

本稿で述べる方法では，型付けされた「識別子 - 値」の関連の集合をデータベース状態と考え，その基本操作を関数として定義することにより，Haskell 言語へのデータベース操作体系の導入を行なう．具体的には，

```
data DbId a = DbId (Int, Int)
```

として宣言された型を a 型のデータ識別子として利用し，識別子とデータのペア (DbId a, a) を個々の変数に相当するものとして扱う．形式的には，型 σ の値全体を V_σ，σ 型識別子の全体を I_σ として表した場合に，σ 型のデータベース s_σ は，$s_\sigma \subset I_\sigma \times V_\sigma$ となる．また，データベースに収められた型全体を Σ で表すと，データベース状態の集合は次のように定義される：

$$\{\{s_\sigma\}_{\sigma\in\Sigma} \mid (\forall\sigma \in \Sigma)(s_\sigma \subset I_\sigma \times V_\sigma)\}$$

ここで，s_σ は有限集合であり，また，識別子は一意でなくてはならない．

データベース状態の操作は変数の参照と代入に相当するが，Haskell 言語は静的型付けをされた言語で動的型付け機構 [3] を持たず，また，データとしての型 [12] は扱えないため，データベース操作関数は多重定義関数で実現する．ここでは，必要最低限の基本操作として次の5種類を用いる：

$\underline{allPairs_\sigma}$ DB に収められている σ 型の識別子とその値をリストにして返す関数；
$\underline{new_\sigma}$ σ 型の値をもらい，新しい識別子と対にしてデータベースに格納する；
$\underline{delete_\sigma}$ 指定された識別子のデータを削除する；

$refer_\sigma$　指定された識別子のデータを検索する．変数の参照に相当する；

$update_\sigma$　指定された識別子のデータを，指定されたデータ値に更新する．変数への代入に相当する．

これらの具体的な型と実現方法は次節で説明する．

4　データベース操作体系の組み込み

4.1　状態遷移の制御

純粋な関数型言語では遅延評価が行なわれ，式評価の順序は処理系が決める．この非決定性を解決するために，I/O monad が制御する状態遷移と同様の状態遷移の制御が必要である．ここでは，DB monad (DB, thenDB, returnDB) を用いる．基本的な考え方は state transformer monad [18] [15] に同じである．データベース状態を表す型を DbWorld とすると，monad の型は次のようである：

```
type DB a = DbWorld -> (a, DbWorld)
thenDB    :: DB a -> (a -> DB b) -> DB b
returnDB  :: a -> DB a
```

IO a が入出力動作を表す型であったのと同様に，DB a はデータベース操作を表す型である．thenDB や returnDB も I/O monad と基本的に同様で，"thenDB m k" および "returnDB e" の表すデータベース操作の図式表現はそれぞれ次のようである：

$$db_0 \xrightarrow{m} (x_1, db_1)\ ;\ db_1 \xrightarrow{k\ x_1} (x_2, db_2)$$

$$db \longrightarrow (e, db)$$

DB monad を用いた場合，データベース操作関数は，操作を（副作用として）行なう関数ではなく，操作を作り出す関数となる．例えば，$allPairs$ 関数は次のような型を持つ：

```
allPairsDB :: DB [(DbId a, a)]
```

多重定義関数の利用は，多様関数の使用を容易にしている．例として，与えられた条件 p を満たすデータを検索する関数 findDB を考える：

```
findDB :: (a -> Bool) -> [(DbId a, a)]
findDB p
  = allPairsDB `thenDB` \pairs ->
    returnDB [ (v,d) | (v,d) <- pairs, p d]
```

これは，allPairsDB が定義された全ての型について利用可能な関数で，実際にどの method が実行されるかは型 a によって異なる．

4.2　エラー処理

上記の DB monad はエラーの処理を考慮していない．そこで，2種類のエラー表現とその取り扱い方法を導入する．第1の方法は，I/O monad におけるエラー処理と同様，代数型を用いて成功・失敗を取り扱うもので，データベース操作の型は次のようになる：

```
type DBE a = DB (DbResult a)
data DbResult a = DbSucc a
                | DbFail DBError
data DBError a = DBNull
               | DBOtherError String
```

つまり，単に操作結果を返すだけではなく，成功・失敗を DbResult 型の値で区別するのである．また，この操作に関連して，新たに次のような関数を導入する：

```
thenDBE :: DBE a -> (a -> DBE b) -> DBE b
```

この関数は，取り扱う DB 操作が DB 型ではなく DBE 型である点と，最初の操作が失敗した場合には以降の操作を行なわないという点で thenDB とは異なる．なお，第5節で，利用されている，succDB は

```
succDB x = returnDB (DbSucc x)
```

と定義されたユーティリティ関数である．failDB も同様である．

第2の方法では結果のリストを用いる．これは，成功の場合は結果からなる単元リストを返し，失敗の場合は空リストを返すもので，dangling reference をエラーとせずに無視したい場合に利用できる．

上記の2種類の処理に対応して，実際の基本演算は以下のような型を持つ：

```
allPairsDBE :: DBE [(DbId a, a)]
newDBE      :: a -> DBE (DbId a)
deleteDBE   :: DbId a -> DBE ()
referDBE    :: DbId a -> DBE ()
updateDBE   :: DbId a -> a -> DB ()
deleteDB    :: DbId a -> DB [()]
referDB     :: DbId a -> DB [a]
updateDB    :: DbId a -> a -> DB [()]
```

4.3　繰り返しとフィックスポイント

return... や then... だけではプログラムの記述が面倒なので，for each に相当する制御演算子を用いる．これは文献 [10] に述べられている Cartesian product と product に相当する：

```
listDBE :: [DBE a] -> DBE [a]
appendDB  :: [DB [a]] -> DB [a]
```

前者は，引数リストの要素を thenDBE で繋げながら順次処理し，処理結果をリストにして返す．後者は，引数リストの要素を thenDB で繋げながら順次実行し，各処理結果のリストとつなぎ合わせて返す関数である．

具体的な利用方法については第5節 で示す.

Hakell 言語は遅延評価をするので フィックスポイント演算子も利用可能である. これは

```
fixDB :: (a -> DB a) -> DB a
```

なる型を持ち, 第1引数の関数は自分自身の出力結果に適用される. 使用例は第5節で示す.

4.4 データベース状態の実装

2次記憶上での各型のデータベース s_σ は識別子を索引とする索引付きファイルとして実装した. ファイル中のデータ表現は, 簡単のために文字列を用いた. このため次のような制約が発生している. まず, 文字列表現可能なデータのみが永続性オブジェクトになり得るため, 関数を含むデータ, 無限リストなどの正規形でないデータ, ループを含むデータは永続にできない. 従って, 永続性とデータ型の直交性 [1] は成立しない.

ファイルを直接取り扱うコードは, 後述するトランザクション制御とあわせてすべて C 言語で書かれ, Haskell プログラムから呼び出されている. そのため, プログラム内でのデータベース状態 DbWorld は, 実質的には IoWorld 型の値のみとなっている.

4.5 データベース操作関数の実装

Haskell 言語における多重定義関数は class によって組織化されているので, 先に示した操作関数を class 演算とする Persistent class を用いる:

```
class (Text a) => Persistent a where
  allPairsDBE :: DBE [(DbId a, a)]
        ....
```

ここで, Text は Persistent の親 class であり, class 演算としては文字列とデータとの相互変換関数を持っている. Text を親 class にしているのは, 既に述べた通り, データの文字列表現が必要だからである.

このような instance 宣言は, データベースの物理レベルに関わるものである. 従って, データ独立性の観点から, また, ソフトウェアの信頼性という観点から, ユーザが記述を行なうのは望ましくない. そこで, 今回は Glasgow 大学で開発された Haskell 処理系 GHC-0.19 に修正を加え, Persistent クラスに関しても, deriving が使えるようにし, データベースの基本操作の定義は処理系が行なうようにした.

4.6 トランザクション

トランザクションは4つの関数で制御される:

```
beginTr :: IO a -> (a -> DBE a) -> IOE a
commitTr :: DB a -> (a -> IOE b) -> DBE b
```

```
data Part
 = Basic String Int Int [DbId Supplier]
 | Composite String Int Int [(DbId Part,Int)]
 deriving (Text, Persistent)
data Supplier
 = Supplier String String [DbId Part]
 deriving (Text, Persistent)

partNameOf (Basic name _ _ _) = name
partNameOf (Composite name _ _ _) = name
....
partNameOfDBE pid
  = referDBE `thenDBE` \part ->
    succDB (parNameOf part)
....
```

図 1: 部品データベースのスキーマ定義

```
abortTr :: DB a -> (a -> IOE b) -> DBE b
endTr   :: DBE a -> (a -> IOE b) -> DBE b
```

ここで, IOE は DBE と同様, 成功・失敗を明示する IO monad を構成する型構成子である. 最初の3つの関数は, トランザクションの開始, コミット, アボートをする. endTr は DB処理の結果が成功ならコミット, 失敗ならアボートする関数である.

プログラムはまた全体で一つのトランザクションとなる. プログラム実行は通常の場合と異なり, 関数 dbMainIOE :: IOE () が表す入出力動作にシステム状態を施すことで行なわれる. この動作が成功した場合には, プログラム全体の処理がコミットされ, 失敗した場合にはアボートされる.

5 データベース操作例

ここでは, 簡単な操作例を示す. 文献 [1] で述べられている, 部品データベースに関する task1 ～4 を用いて説明する. task1 「スキーマ定義」であるが, 永続データは Part と Supplier なので, これらを Persistent class の instance にした上で, フィールドデータを参照・変更するための基本的な関数を用意する (図1) 第1.1節で示した例とは異なり, データの相互参照が通常のポインタではなく, DbId Part などで間接的に行なわれている点に注意されたい.

task2 は「価格が 100 以上の基本部品の検索」である. これは 図2 にように書き表せる.

task3 は「すべての組立部品についてトータルの価格と重さを検索」である. これは図3(a) のように書き表せる. ここでは, リストの内包表記 を用いてデータベース操作のリストを作成し, それを listDBE で評価している. また, fixDB 関数を利用すれば, memo

```
dbMainIOE
 = returnIO () `beginTr` \_ ->
   (let pred p
          = isBasic p && partCostOf p >= 100
    in findDB pred ) `thenDBE` \pairs ->
  succDB (map snd pairs)
  `endTr` \Basic_parts ->
   ... 検索結果の表示 ....
```

図 2: 価格が 100 以上の基本部品の検索

を利用することも比較的簡単である (図 3(b)) .

task4 は「新たな組立工程の登録」である. 図 4 に, ある組立部品の部分部品を登録する関数 assertMadeFromDBE の定義を示す. ここで, tranMadeFromDBE が与えられた部品の部分部品リストを返す関数であること, 図 1 からは省略した関数が一部利用されていることに注意されたい.

6 関連する研究との比較

Staple [13] は永続性記憶を利用した純粋な関数型永続性言語で, 型と永続性の直交性などの永続プログラミングの原理に忠実である. 任意のプログラムモジュールは永続性記憶に置かれる (モジュール永続性). 任意のデータに対して文字列での名前付けができ, ファイルと同様の対話を利用して参照透過に操作できる (ストリーム永続性). ここで, ストリームデータの型は動的にチェックされる. データベース更新は, 古いストリームデータから, 新しいストリームデータを作成し, 同じ文字列名での名前付けを行なうことでなされる.

本研究における実装では, モジュール永続性は無く, また, データベース状態はプログラムの外部と考えているので, ヒープ中のポインタとデータベース内の識別子は別である. そのため, プログラム内で陽に refer 演算を施す必要があり, 複雑なデータを扱う場合には面倒である. また, dangling reference も発生する可能性がある. ヒープは通常のものが利用されるので大量のデータが読み込まれてしまったケースには対処できない. 一方, 更新操作の対象は個々のデータであるので, 更新操作に伴うデータのコピーは Staple のケースよりも少なくすることが (原理的には) 可能である. 永続データとして扱える型は事前に決定しなくてはならないが, ディスクデータと型との対応関係は Persistent クラスの内部的な実装方式に隠蔽されているので動的な型チェックは不要である.

improved Bin file [8] は Staple のストリーム永続性の考え方を Haskell 言語の Binary ファイルに採り入れたものである. 本研究との基本的な差異は Staple のストリーム永続性と同様だが, 言語のサポート・ライブラリとして考えた場合では, 本研究の方法はヒープ構造は一切手を加える必要がなく, ほとんどの Haskell コンパイラにおいて容易に利用可能である.

PFL [16] は, 継続による更新操作制御と例外処理を行う参照透過な永続性プログラミング言語で, 型や関数も更新の対象となる. 抽象実体は 0 項データ構成子で表し, 実体属性は関数で表す. また, これはリレーションに相当する selector があり, パターンを用いて検索できる.

データのモデリング方法が関数型データモデルに忠実でわかりやすいが, 匿名の実体を作成することができないので, 実際に大量のデータを扱う場合には面倒である. また, 継続の利用方法は Haskell などで利用される式継続ではなく命令継続であるため, データの参照が常にグローバルな名前で行なわれる. これに対して, 本方法では, monad ベースの細かな制御を利用している.

7 まとめ

関数型プログラミング言語 Haskell 上に, monad と Persistent クラスを用いたデータベース操作体系の導入を行った. 基本操作は固定であるが, これらを組み合わせてさまざまな操作が作成できる. 前節で述べたように他のシステムに対して劣る点もあるが, 静的な型付けを行なう点, ヒープ構造を改造しなくても様々な Haskell コンパイラに利用できる点, monad を利用した細かな制御が可能な点, 更新操作でのデータコピー量が少ない点などが利点としてあげられる.

現在の問題点としては以下のようなものがある :

(1) 問い合わせが手続き的な様相を呈しており複雑である. また, 通常の計算と混合しているので最適化が難しい ;
(2) DB monad は strict であるため, 余計なデータを参照してしまう可能性がある ;
(3) C で書かれたファイル操作部の効率が悪い.

また今後の課題としては以下のようなものがある :

(1) スキーマ変更のことが考慮されていない. 現在は, 必要なディレクトリやファイルは別途作成し, また, データ変更はプログラムを作成する以外に方法が無い ;
(2) 単一プログラム内部でのトランザクション制御機能はあるが, 複数プログラム間での排他制御と同時実行機能がない ;
(3) バルク型としてはあくまでリストの利用を前提としており, その他の構造上での繰り返し操作は提供していない ;

```
dbMainIOE
 = returnIO ()
  `beginTr_`
   findDB (\part -> isComposite part) `thenDB` \pairs ->
   listDBE [ cAndM pid `thenDBE` \(c, m) -> succDB (part, c, m) | (pid, pt) <- pairs ]
  `endTr` \tuples ->
   ... 検索結果の表示 ....

cAndM :: DbId Part -> DBE (Int, Int)
cAndM pid
  = referDBE pid `thenDBE` \part ->
    if isBasic part then
      succDB (partCostOf part, partMassOf part)
    else
      -- 組立部品の場合
      partMadeFromDBE pid  `thenDBE` \sub_pids ->
      -- 子供部品毎のコストと重さをリストにする
      listDBE [ cAndM sid `thenDBE` \(c, m) -> succDB (s * q, m * q)
                | (sid, q) <- sub_pids ] `thenDBE` \cm_list ->
      let -- 子供のコストと重さを利用して組立部品のコストと重さを計算
        { c_total = sum (map fst cm_list); m_total = sum (map snd cm_list);
          acost   = partCostOf part;       massi   = partMassOf part }
      in succDB (c_total + acost, m_total + massi)
```

(a) 直接的な検索例

```
dbMainIOE
  = returnIO ()
   `beginTr_`
    fixDB ( \ ~(DbSucc memo) ->
      allPairsDB        `thenDB` \pairs ->
      listDBE [ cAndM pid memo `thenDBE` \(c, m) -> succDB (pid, pt, (c, m))
              | (pid, pt) <- pairs ] ) `thenDBE` \memo ->
    succDB (filter (\(_, p, _) -> isComposite p) memo)
   `endTr` \memo ->
   ... 検索結果の表示 ....

cAndM pid memo
  = referDBE pid `thenDBE` \pt ->
    if isBasic pt then
      succDB (partCostOf pt, partMassOf pt)
    else {- 組立部品の場合 -}
      ... memo リストから pid の子供の部品を探し pid のコストを計算する ...
```

(b) memo を利用した検索例

図 3: 組立部品のトータルの価格と重さの検索

```
assertMadeFromDBE  :: DbId Part -> DbId Part -> Int -> DBE ()
assertMadeFromDBE pid ppid quant
  = tranMadeFromDBE ppid 'thenDBE' \ppid_subs ->
    if pid 'elem' ppid_subs then
      partNameOfDBE pid 'thenDBE' \name ->
      partNameOfDBE ppid 'thenDBE' \name' ->
      let mesg = "cycle for asserting "++ name ++ " and " ++ name' ++ "."
      in failDB (DBOtherError ("assertMadeFromDBE: " ++ mesg))
    else
      addPartUsedByDBE ppid pid 'thenDBE_'
      addPartMadeFromDBE pid ppid quant 'thenDBE_'
      succDB ()
```

図 4: ある部品が別な部品から作られていることを登録

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. Atkinson and P. Buneman, "Types and Persistence in Database Programming Langaues," *ACM Comp. Surv.*, **19(2)**, Jun. 1987., pp. 105-190.

2) A. Albano, L. Cardelli and R. Orsini, "Galileo: A Strongly Typed Interactive Conceptual Language," *ACM TODS*, **10(2)**, June 1985, pp. 230-260.

3) M. P. Atkinson and R. Morrison, "Types, Bindings and Parameters in a Persistenet Environment," in [4], 1988, pp. 3-20.

4) M. P. Atkinson, P. Buneman and R. Morrison (eds.), *Data Types and Persistence*, Springer-Verlang, Berlin, 1988.

5) F. Bancilhon and P. Buneman (eds.), *Advances in Database Programming Languages*, ACM Press, NY, 1990.

6) O. Boucelma and J. Le Maitre, "An Extensible Functional Query Language for an Object-Oriented Database System," *DOOD'91*, LNCS 566, Springer-Verlag, 1991, pp. 567-591.

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

8) K. Hamond et al., "Improving Persistent Data Manipulation for Functional Language," in [11], pp. 72-84.

9) 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.

10) D. J. King and P. Wadler, "Combining Monads," in [11], pp. 134-143.

11) J. Launchbury and P. Sansom, eds., *Functional Programming, Glasgow 1992*, Springer-Verlag, 1993.

12) D. C. J. Matthews, "An Overview of the Poly Programming Language," in [4], 1988, pp. 43-50.

13) D. J. McNally, and A. J. T. Davie, "Two Models for Persistence in Lazy Functional Programming Systems," SIGPLAN NOTICES **26**(5), May 1991, pp. 43-52.

14) R. S. Nikhil, "The Semantics of Update in a Functional Programming Language," in [5], 1990, pp. 403-421.

15) S. L. Payton Jones and P. Wadler, "Imperative Functional Programming," *ACM POPL*, 1993, pp. 71-84.

16) C. Small, "A Functional Approach to Database Updates," *Information Systems*, **18**(8), Dec. 1993, pp. 581-595.

17) P. Wadler, "How to Make ad-hoc Polymorphism Less ad hoc," *ACM POPL*, Jan. 1989, pp. 60-76.

18) P. Wadler, "The Essence of Functional Programming," *ACM POPL*, Jan. 1992, pp. 1-14.