データベースシステム 89－9
（1992. 7. 23）

永続的プログラミング言語P３Lの実装方式

鈴木慎司 喜連川優 高木幹雄
東京大学 生産技術研究所

データベースと先端的なアプリケーションの間の型システムのミスマッチを解消する一つの方法は、プログラミング言語が提供する型システムに従って生成されたオブジェクトをプログラマの介在無しにデータベースに格納する機能を言語に付与することである。P3Lにおいてはオブジェクトに永続性という属性を持たせることによって、そのオブジェクトはデータベース内に生成され、後に必要となった時には自動的にデータベース内からアプリケーションの実行空間に読み込まれる。この遅延読み込みを実現するには、ハードウェアを用いる方法からクラスライブラリによる方法までいろいろあるが、P3Lではポインタ書き換え (Pointer Swizzling) と呼ばれる方式に基づき、トランスレータによって遅延読み込みを実現している。本稿ではP3Lのポインタ書き換えの方式を他の方式と対比しながら紹介するとともに、P3Lのオブジェクトストアとのインタフェースを中心に全体構成について述べる。

The implementation of a persistent programming language P3L

Shinji Suzuki† Masaru Kitsuregawa† Mikio Takagi†

The problem of impedance mismatch between advanced application and database is now widely recognized. One solution is to augment an programming language with an ability to store its objects into database without any programmer intervention. In P3L a programmer can declare an object to be persistent so that a group of objects will be stored in database without any extra coding. Those objects will be automatically retrieved from database when they are accessed later on. There are many ways for achieving this incremental loading ranging from pure hardware method to the use of a class library. The P3L implementation employs a software technique called pointer swizzling, which is taken care of by the P3L translator. In this paper the pointer swizzling in P3L is described in comparison with other swizzling methods and structure of P3L implementation is discussed looking closely at the interaction between the translator and a transactional objectstore Kala [1].

†Institute of Industrial Science, Univ of Tokyo

[1]Kala is a trademark of Penobscot Research Center

1 P3L の概要

まず最初に、P3L 言語について述べる。P3L の syntax は通常の C 言語に persistent という storage class を加えるとともに変数宣言を次のように拡張している。変更はオプショナルな db.binder を追加していることである。

```
decl.qualifier.list identifier.declarator [initializer] [db.binder]
type.qualifier.list identifier.declarator [initializer] [db.binder]
default.declaring.list ',' identifier.declarator [initializer] [db.binder]
```

ただし db.binder の指定を伴って宣言される変数は persistent storage class をもつ必要があり、これは意味解析部において検査される。また db.binder 部を持たない persistent 変数の宣言は現在は許されていない。db.binder はリテラル文字列であり、永続変数の宣言は次の例に見られるような形となる。

```
persistent Employee * employee_list : "U-TOKYO.EMPLOYEES"
```

この宣言形式からも明らかなように、P3L における変数（オブジェクト）の永続性はインスタンス毎に設定されるものであり、型とは独立である。(Orthogonal Persistence) オブジェクトに永続性が与えられデータベースに書き込まれるのは、このように宣言時に明示的に指定された場合とプログラムが終了する段階でオブジェクトが他の永続オブジェクトからのポインタによって参照されている場合である。(Persistence by Reachability) このことは、実行中のプログラムの視点からは、永続オブジェクトも揮発オブジェクトも同一に扱われることを意味し、プログラマの負担は多いに軽減される。[Hanson] また、プログラマがオブジェクトの生成時に永続性についての決定をする必要もない。

db.binder はプログラム間でオブジェクトを共有するために用いる鍵であり、いわゆる persistent root を取り出すために用いられる。P3L トランスレータは、宣言された永続オブジェクトがプログラム起動時にデータベースからアプリケーションのプロセス空間内に読み込まれるように出力コードを調整する。プログラムの最初の実行時には、指定されたオブジェクトがデータベース中に見つからない場合がある。その時には、指定されたオブジェクトをデータベース中に生成し C の static 変数と同様に初期化する。これがデータベースにデータを追加する一つの方法だが、例えば <LIST 1> のように生成したオブジェクトを persistent root によって支配される集合型オブジェクト (e.g. 連結リスト) の要素とすることでデータを追加するのがふつうである。

```
typedef struct foo {
   struct foo * next;
   char    key[64];
   char value[128];
} NODE;
persistent NODE * root : "ROOT_NODE";
NODE * new_NODE() {
  ... sizeof( NODE ) バイト分のメモリを
  ... malloc し、メンバを初期化する。
}
int main(int ac, char **av) {
  NODE * n;
  n = new_NODE();
  n->next = root;
  root = n;
}
```

<LIST 1> Creation of Employee

```
typedef struct foo {
   struct foo * next;
   char    key[64];
   char value[128];
} NODE;
persistent NODE * root : "ROOT_NODE";

int main(int ac, char **av) {
  NODE * p;
  for(p=root;p;p=p->next)
    printf("%s:%s\n",p->key,p->value);
}
```

<LIST 2> Retrieval of Employees

`n->next = root; root = n;` の部分で、新しく生成したオブジェクトを root を錨とするリストの先頭に挿入している。<LIST 1> のプログラムで作成された永続オブジェクトのリストを走査し、ノードの内容を表示するプログラムを <LIST 2> に示す。

トランスレータは永続変数の宣言を次のように変換する。

```
persistent NODE  * root : "ROOT_NODE";
     ===>  NODE ** root = p3l_Load("ROOT_NODE",(void *)&root,sizeof(*root),
                                  (void *)p3l_Monadify_PPfoo);
                          <LIST 3>
```

この例からわかるように、P3L トランスレータは P3L ソースを読み込み、C++ を出力する。C++ の関数形式の初期化を用いることによって容易にコンパイラ、リンカ独立に起動時の処理を取り扱うことができる。

トランスレータが行なう処理は、この宣言の書き換えを含め、

1. 永続変数宣言の書き換え
2. Kala(後述) によって管理されるオブジェクトストアへのオブジェクトの書き込みのための関数生成
3. ポインタ書き換え関数の埋め込み

の 3 つである。

つぎのセクションでポインタ書き換えについて解説し、その後のセクションで Kala とのインターフェイスを中心にランタイムの動作について述べる。

2 ポインタ書き換え

複数のプログラム間でデータを共有するためには、プログラムの実行時を越えて有意なアドレス空間にオブジェクトを配置する必要がある。一方実際にオブジェクトをアクセスするには、オブジェクトは CPU にネイティブな空間に存在しなければならない。解決法のひとつはプロセス空間の一部に共有データのための領域を割り当て、この領域に対し Transaction 機能付きの demand paging をすることである。[That] ここではこれを Sigle Space Addressing(SSA) と呼ぶことにする。SSA には以下のような問題がある。

1. ユーザ制御可能なページフォールトハンドラが利用できない場合には、オペレーティングシステムを変更する必要があること
2. データベース全体が仮想メモリ空間にマップされるため Page Table 用の資源を浪費すること
3. 現在の 32bitCPU, オペレーティングシステムでは十分なサイズの空間が提供できないこと

の 3 つである。IBM の System/38, AS/400 シリーズでは 48bit の連想的なアドレス変換によって、2, 3 の問題の解決をはかっている。なお本論文では、仮想メモリ空間は CPU の通常のメモリアクセス命令によって参照可能な空間のことを意味するものとする。

SSA と対照される方式に Multiple Space Addressing(MSA) 方式がある。この方式ではプロセス空間とは別の Universal な空間 (UID 空間) にデータを配置し、ポインタが dereference される時 (あるいはその前) に、ポイントされているオブジェクトを仮想メモリ空間に読み込む。この様子を図 1に示す。UID 空間は必ずしも仮想空間よりも大きい必要はないが、データベースと仮想メモリ空間のサイズの違いや、世界的にユニークなアドレスを与えることを考えると UID の表現には仮想アドレスよりもずっと多くのビットが必要になる。

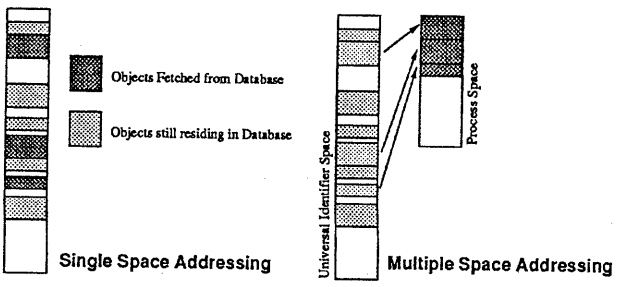

図 1: SSA and MSA

UID空間におけるポインタをUIDと呼ぶことにすると、UIDをどのタイミングで仮想アドレスに変換するかによってMSAにはいくつかのバリエーションが存在する。よく用いられる方式はUID空間内、仮想メモリ空間内にかかわらず、他のオブジェクトへの参照をUIDの形式で保持し、参照がdereference（以後deref）される時に「UID=>仮想アドレスのマッピングテーブル」を検索してオブジェクトをアクセスする方式である。ONTOS, Versant, Objectivityは参照クラス(TRef,LINK,Handle)[1]を用意することで実現している。O_2,Eではコンパイラが参照のderefを検出して、適切なコードを生成する[2]。この方式の問題点は、仮想記憶上でのポインタのコピーのコストとderefのコストが高いことである。一方、利点はマッピングテーブルを介しているため仮想記憶中でのオブジェクトの移動（Copying GCに利用可能）や仮想空間中からのオブジェクトの追いだし(eviction)が容易になるところにある。(ただし追いだしを行なった場合は、マッピングテーブルエントリのGCが問題になる。)

UID=>仮想アドレス変換方式にみられるオーバヘッドを解消する方法にポインタ書き換え(Pointer Swizzling)がある。この方式では、仮想空間中にあるオブジェクト中のポインタをUIDのまま保持するのではなく、仮想アドレスによって書き換える。それによりマッピングテーブルの検索・参照のオーバヘッドを除去することができるし、オブジェクトが仮想メモリ空間に読み込まれた時に書換えを行えばポインタのコピーのコストを抑えることもできる。問題点はポインタの書き換えを行なうためにオブジェクト中のポインタの位置に関する情報を管理する必要のあること、オブジェクトの移動や追いだしが難しくなること、の2点である。オブジェクトの追い出しをしないと使用可能な仮想メモリ空間が減少を続けるが、この方式を採用しているObject Storeは、仮想記憶は年々拡大を続けており一つのトランザクションセッションを実行するには十分なサイズである、と主張している。[CACM]

ポインタ書き換え方式は、さらに書き換えのタイミングによって3つ程に分類される。

2.1 Swizzling at Pointer Dereference Time

PS-Algolの実装において用いられた方式である。これは、後で述べるように他の書き換え方式に比べ非常に効率が悪い。それが問題とされなかったのは仮想メモリ空間内におかれた参照の表現にダイレクトなポインタではなく、PIDLAMと呼ばれるテーブルへのインデクス(LON, Local Object Number)を用いていたからであろう[3]。このテーブルはLONから、オブジェクトのおかれたメモリアドレスおよびUIDを検索するために用意された。ポインタのderefに際しては、常にテーブルがアクセスされるためポインタ書き換えのためのコストは、このテーブルへのメモリアクセスのコストに隠れてしまう。おそらくこの書き換えが仮想マシンの1命令として実行され、ポインタ検査のコストがよく見えなかったことも効率の悪さが見過ごされた原因のひとつだろう。実際i80x86(x>=2)のようにセグメントアーキテクチャを持ったマシンを用いれば、この書き換えのための検査をハードウェアで実行することができるので性能上の問題は解決可能である。以下では、議論をわかりやすくするためPIDLAMによる間接参照を無視することにする。

図2(a)を参照いただきたい。PS-Algolではデータベース(Persistent Heap)中のオブジェクトをアクセスするために、プログラマはPS-Algolの実行時ライブラリ関数であるlookup()を呼び出す。この関数はオブジェクトの名前をキーとして、そのオブジェクトのUIDを取り出す関数である。lookup()の呼びだし後のメモリの様子が図(a)の左側に示されている。左側の長方形がUID空間、右側が仮想メモリ空間である。ポインタPが仮想アドレスではなくUIDを保持している点に注意されたい。PS-Algolの実装では、CPUにネイティブなポインタが表現できるアドレス範囲を二分し、半分をUIDの空間に、残りを仮想アドレス空間に割り付けている。従ってポインタPはUIDと仮想アドレスのどちらでも保持でき、MSBによって両者が区別される。

さらにプログラムの実行が進むと、ポインタが指しているオブジェクト(A)がアクセスされる、つまりポインタPのderefがおこるが、その前にポインタPが指しているオブジェクトが既に仮想空間に読み込まれているか否かが調べられる。この場合は(A)は未だ仮想メモリ空間にないので(A)の読み込みが行なわれる。そして、ポインタPはその読み込みアドレスで書き換えられる[4]。次の(A)へのアクセスでは、ポインタPの検査は行なわれるが、検査の結果ポインタPは仮想アドレスを指していることがわかるので(A)は読み込まれない。

PS-Algolの方式が、充足しようとするinvariantは「derefに使われるポインタは全て仮想アドレスを保持している」というものである。これは直観的にわかりやすい方式だが、上で述べたように、ポインタのderefの度に検査が行なわれるため、ポインタの検査はメモリ参照の回数とおなじ回数だけ行なわれてしまう。共通式除去のようにコンパイラによる静的解析で検査回数を減らす研究も行なわれたが、静的解析には限界がある。

---

[1] いずれもC++のクラスである

[2] Eにおいては、永続オブジェクトの読み込みはコンパイル時に静的にスケジュールされるので実行時にはマッピングテーブルは存在しない。

[3] 一部の実装ではポインタを用いたらしい。

[4] PIDLAMを用いた実装では、LONが割り当てられ、ポインタPがLONで上書きされ、PIDLAMの仮想メモリアドレスのフィールドに(A)のアドレスが書き込まれる。

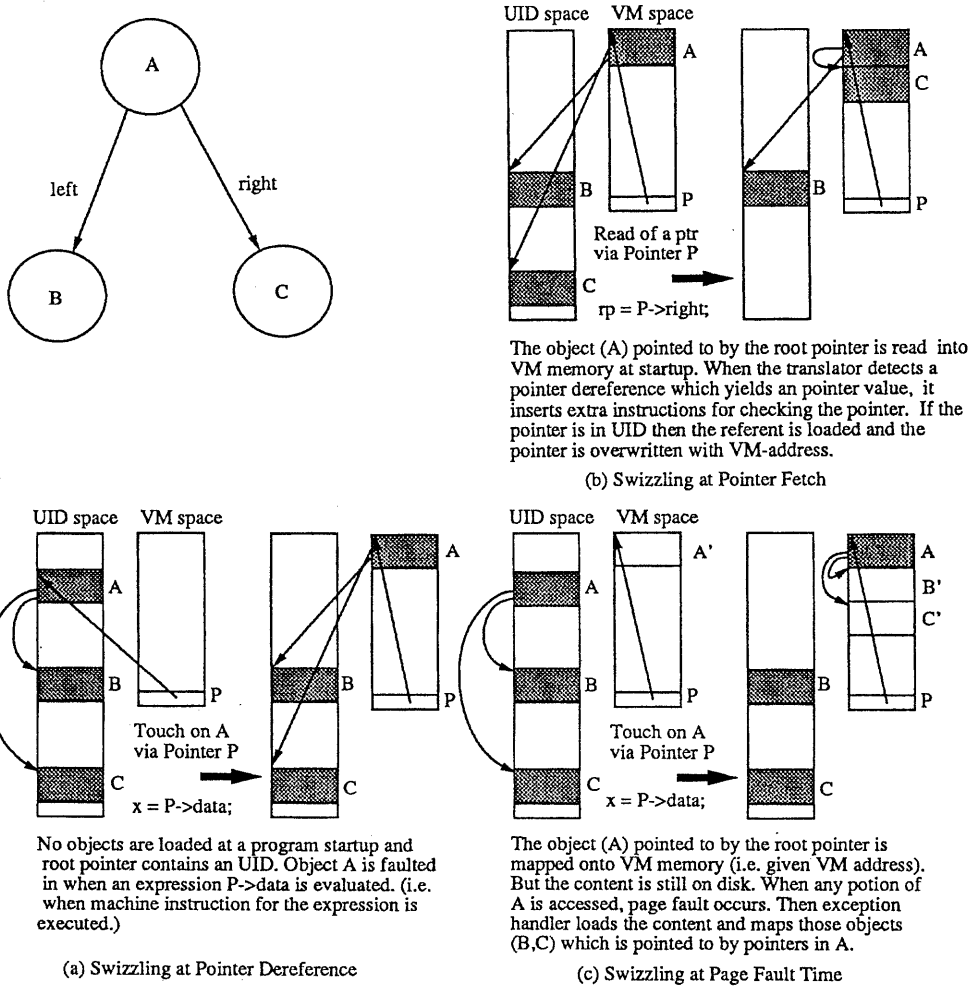

図 2: 各種ポインタ書き換え方式

2.2 Swizzling at Pointer Fetch Time

この方式は P3L に実装において開発された方式である。図 2(b) の中のポインタ P は、P3L のプログラムにおいて persistent と宣言されたオブジェクトを指すポインタである。すでにセクション 1 で述べたように P3L のトランスレータは、persitent と宣言された変数 (x) を同じ型のオブジェクトへのポインタの宣言 (*x) におきかえ、同時にプログラム中の x への参照を *x で書き換える。宣言された永続オブジェクトはプログラム起動時に自動的に読み込まれるので、ユーザの記述した部分のプログラムが実行を始めた段階でのメモリの状況は図 (b) 左のようになっている。

仮想アドレス空間中の (A) の中にあって、UID 空間中の (B),(C) を指しているポインタがあるが、P3L においては UID が仮想アドレス空間よりも遥かに大きい。そのため PS-Algol のように UID をそのまま保持することはできないので、「ポインタの置かれたアドレス =>UID」のマッピングテーブルを用意する。UID 表現のポインタを仮想メモリ空間にロードした時には、ポインタをありえない仮想メモリアドレス (-1) で書き換え、マッピングテーブルにエントリを追加する[5]。

[5]現在の実装では書き換えは Kala によって行なわれ、ここで述べた方法と微妙にことなるが原理的に同じ方法を用いている。

プログラムの実行が始まり、(A) のスカラデータ（ポインタでないデータ）がアクセスされるときには、(A) は読み込み済みなのでポインタの検査をする必要はない。しかし、図 (b) の左側の図からわかるように (A) の中のポインタは必ずしも仮想アドレスを指していない。P3L コンパイラは「ポインタによるポインタの参照」を検出して、その参照が行なわれた際にポインタの書き換えをする。つまり、`rp=P->right` という式を `rp=Swizzle(P->right)` という式に変形する。Swizzle() 関数は参照渡しされた引数のポインタが UID(-1) であるかを検査して、そうであるならば、これをポイントされたオブジェクトの仮想アドレスで書き換える。

上記の処理によって「静的領域内、スタック領域内、CPU レジスタ上に存在するポインタからアクセスできるオブジェクトは全て仮想空間内に存在する」という invariant が充足される。この invariant は PS-Algol のものより小さなコストで充足できる。何故なら、一般に必要なポインタの読みだし回数はメモリアクセスの回数よりずっと少ないからである。この invariant は Baker の incremental copying garbage collector の充足する invariant「to space 内からフェッチされるポインタは全て to space へのポインタである」と同等のものである。

2.3 Swizzling at Page Fault Time

3つめの方式は [Wilson] において提唱された。[CACM] からわかる範囲では、おそらく ObjectStore も同一あるいは極めて類似した方式を用いていると推察される。この方式は Andrew Appel による two space incremental copying garbage collector と同様の invariant を充足するように動作する。それは「仮想メモリ空間にコピーされるポインタは全て仮想アドレスを保持している」である。そして Appel collector の invariant は「to space にコピーされるポインタは全て to space へのポインタである」というものだ。

仮想空間に置かれるポインタが仮想空間を指しているという条件を満たすためには (A) の中のポインタは仮想メモリ空間内に置かれた (B),(C) を指していなければならない。そのためには (B),(C) も読み込む必要があり、その中にはまたポインタがある。結局プログラム起動時に全てのオブジェクトを読み込まなければいけないように思われる。ちょうど、オリジナルの Copying Collector がすべての live object を to space にコピーしなくてはいけないように。しかし、すでに述べたようにデータベースを全て仮想アドレスにロードすることは非現実的である。この問題の解決のために、Wilson の方式も Appel の collector 同様 MMU によるページ保護を利用した遅延ローディングを利用している。原論文では、ローディングをページ単位で行なうように提案しているがここでは簡単のためオブジェクト単位のローディングを仮定する。

図 2(c) の左の図が、Wilson の方式を利用した場合の起動時のメモリの様子である。(A) は UID 空間に置かれたままであるが、仮想アドレス中には (A) を読み込むための空間が用意される。この領域にたいする実メモリはまだ割り当てない。そしてこの空間にはアクセス保護がかけられる。最初にポインタ P を deref して (A) の一部をアクセスした時には、ページ保護例外が発生する。ページフォルトハンドラは (A) の内容を読み込むとともに、(A) の内部のポインタが指しているオブジェクトの格納空間を確保して、それらのポインタを仮想アドレスで書き換える。そして (B),(C) の格納空間にアクセス保護をかけておく。こうして、「読み込まれたオブジェクト内のポインタは仮想アドレスを保持する」という invariant を充足したままオブジェクトをインクリメンタルにロードしてゆく。この方式ではページフォールト時のポインタ書き換えをサポートするために、ページ内のどのワードがポインタであるかを示す補助データを 2 次記憶上に管理する必要がある。また、フォールト時に読み込むべきオブジェクトの UID を知るために P3L と同様なマッピングテーブルをランタイムの補助データとして管理する必要もある。

2.4 上記2方式の比較

ここでは、P3L の方式と Wilson の方式の特徴を考える。まず P3L の方式の利点としては、

1. MMU, オペレーティングシステムあるいはユーザ定義可能なページフォールトハンドラを仮定していないので移植性が高い。
2. 仮想空間の不要な断片化がおこらない。
3. ロードしたオブジェクトの全てのポインタフィールドを書き換える必要がないし、ページフォールトが起きないので cold start 時の立ち上がりが速い。

ということがあげられる。Wilson の方式では、ポイントされるオブジェクトが参照されない場合にも、ポインタは書き換えられる。仮想空間の断片化は、Page Table によるメモリの圧迫や TLB のヒット率の低下につながる。

Wilson の方式の利点 (括弧内は P3L の問題点) は、

1. コード生成には通常のコンパイラが使用できる。(専用のコードジェネレータあるいはトランスレータが必要)

2. ポインタ検査のためのよけいな命令が生成されない。(検査のための命令がオブジェクトサイズの増大、実行速度の低下をまねく)

ということであるが、方式の説明で述べたような補助情報の管理のためには、変数の型やそのアロケーションを管理する必要があり、コンパイラとは別に型情報や永続変数の情報をプログラムから抽出するプログラムを作成しなくてはならない。また、実行速度についても、仮想空間の断片化の悪影響や、P3L の方式の改良 (CSE や Loop Invariant Motion, タグ付きポインタ演算の例外処理ハードウェアの利用) を考えた場合、容易に優劣を決定することはできない。

3 P3L ランタイムと Kala のインターフェイス

3.1 The Kala Basket

「Kala[6]は OODBMS, OMS, 永続オブジェクト指向言語などの実用オブジェクトシステムのための永続オブジェクトストアである。Kala はデータ管理の単位 (monad) を immutable とするとともに、monad の動的グルービングの機構である Basket を導入することにより、単一の基本的なプリミティブに基づいて、様々なトランザクションや、アクセス制御、バージョン制御、コンフィグレーション制御を可能とする。」[Kala]

monad は図 3左の m1 に示すようにビット列と他の monad への参照 (ポインタ) から構成される。monad のグループ化の機構には 2 つの方法があり、一つは kin とよばれる静的な方式である。monad の id(mid) にはその族 (kin) をあらわす情報が埋め込まれるようになっており、族によって静的にグループ化される。もう一つの方式は、Basket と呼ばれる ADT に mid を格納することによって行なわれる動的なグループ化である。

kin のメカニズムは、Multi-version トランザクションの実行における同一オブジェクトの版を表現したり (版毎に、同一の kin に属する別の mid をもった monad を作る)、version 制御における履歴の保持などに用いることができる。一方 Basket のメカニズムは、トランザクションの状態 (item の識別子から monad へのマッピング) を保持したり、バージョン制御におけるコンフィギュレーションの管理などに用いることができる。あるいは、ユーザのデータベースへのアクセス権の制御にも用いることができる。図 3左において、使用者 A の Basket には m1,m2,m3 の monad へのハンドルが格納されいる。そこで使用者 A は m1 内の参照を解決し m2,m3 の内容にアクセスできる。一方使用者 B は m1 へのハンドルを持っているので m1 の内容、ひいては m2, m3 への参照は取り出せるが、それを解決して m2, m3 の内容にアクセスすることはできない。このように Basket 内の mid の授受によって monad へのアクセスを制御することが可能である。

単純なトランザクションは次のように実装できる。Kala は上のアクセス制御で述べたように、ある kid(あるいは mid) が指し示す monad をアクセスする際に、Basket のなかのハンドルを検索してアクセス対象を決定する。この検索順序を決定するために Kala はクライアントごとに Basket のリスト (BSL) を保持している。

トランザクションの開始にあたって、Kala クライアントは新しい Basket(B1) を生成し、BSL の先頭に追加す

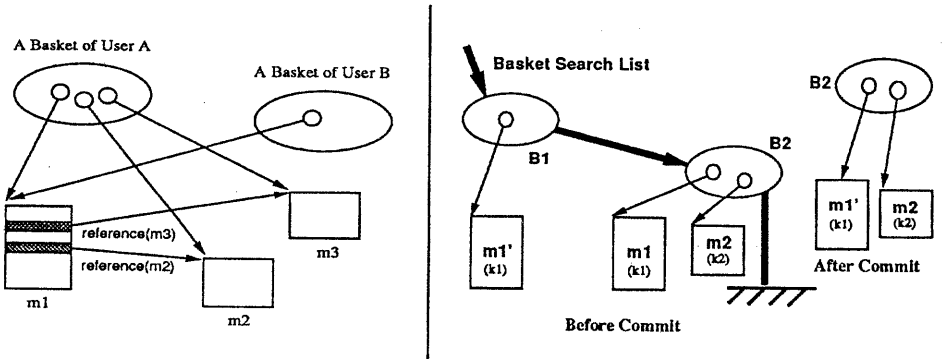

図 3: An Acccess Control(right) Transaction and BSL(left)

[6]Kala is a commercial product from Penobscot Research Center

る。この時B1は空である。クライアントがkid=k1で指示されるデータベースのitemを読み出すときには、k1のkinに属するmonadのハンドルはB2の中にしかないので、m1の内容が与えられる。クライアントがm1を更新しm1の内容をKalaに書き戻すときには、monadはimmutableであるのでm1に上書きするかわりに、kin k1に属する新しいmonad m1'が生成され、そのハンドルがB1の中に作られる。この後、同一のトランザクションにおいてk1で指示されるitemが要求された場合には、今度はB1の中にハンドルが見つかるので、m1'の内容がクライアントに渡される。最後にコミットを行なう時には、Basket B1の中のハンドルがB2に移されるされる。このとき、移されるハンドルと同一のkinに属するハンドルがB2中に存在した場合は、後者は前者で置き換えられる。その結果、Basket B2の内容は右側の図のようになり、データベースの更新が終了する。ハンドルの移動はAtomicに行なわれるので、トランザクションのAtomicityが保証される。Abortする場合には、単にB1を破棄すればよい。そのときm1'を参照するハンドルは全くなくなるのでKalaはm1'に割り当てられていた資源を解放する。

3.2 P3Lランタイム

P3Lのランタイム関数のうち、最初に起動されるのは関数p3l_Load()(<LIST 3>を参照)である。この関数は渡された4つの引数をメンバとする構造体を生成し、それをリストにつなぐ。引数は 永続オブジェクトの名前(char *), 永続オブジェクトを指すポインタのアドレス(void **), 永続オブジェクトのサイズ(int), 永続オブジェクトの書き込みのための関数へのポインタ(void (*pf_monadify)())である。pf_monadifyは永続オブジェクトの型ごとに、トランスレータによって自動生成される。

トランスレータはmain()関数の先頭にp3l_startKala()関数の呼び出しを埋め込む。この関数はKalaの初期化ルーチンを呼び出した後、上記リスト中の各要素に対して次の処理を行なう。

```
オブジェクト名 ==> kidの変換テーブル（monadとして実現されている）を検索する。
if( 検索が成功したら ) {
  kidが指示するmonadを読み込み、ポインタを読み込みアドレスで初期化する。
} else {
  新しいkidをKalaに要求し、オブジェクト名 ==> kidの変換テーブルにエントリを追加する。
  size分のメモリを確保し、内容をゼロクリアして、そのアドレスでポインタを初期化する。
}
```

既に述べたように、P3Lのトランスレータは式のなかにポインタ検査のためのコード(Swizzle関数の呼びだし)を挿入する。この関数は、ポインタが有効な仮想アドレスを含んでいるかを検査し、そうでなければKalaのAPIであるTouchPointer()を呼び出すようになっている。このSwizzle()関数によりオブジェクトはインクリメンタルにディスク上から読み出されていく。Swizzle()関数は効率向上のためにインライン関数として生成される。

プログラムが終了する直前にはp3l_commitKala()が呼び出される。p3l_commitKala()は上で述べたリストのエントリひとつひとつに対してpf_monadifyが指す書き込み関数を呼び出す。この関数は次の2つの処理からなる。

1. 引数で指定されたオブジェクト内のポインタひとつひとつに対し、ポインタの値がNULLでなければ、ポイントされたオブジェクトの型に応じて適切な書き込み関数を、そのポインタを引数として呼び出す。この呼びだしは再帰的に行なわれる。

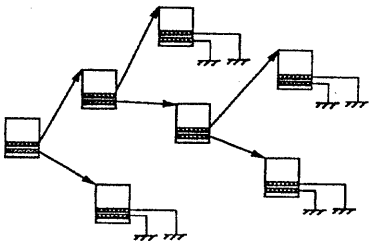

図 4: Monadifying of Persistent Objects

2. Kala API CreateDatum を呼びだし、ポインタが指すオブジェクトから monad を生成する。このときハンドルは BSL の先頭に作られたテンポラリ Basket に収められる。

これは、宣言された永続オブジェクトからポインタにより到達可能なオブジェクトをポストオーダで monad 化する処理である (図 4)。最後にテンポラリ Basket を public Basket にダンプしてトランザクションを終了する。

4 実装環境

現在の処理系は Sun SparcStation の上で動作している。ポインタ書き換えのセクションで述べたように、P3L の処理系はプラットフォーム独立であるので Kala がサポートする環境には容易にポーティングできる。

トランスレータは C++ を用いて記述され、意味検査も行なっているためヘッダファイルを含め 1 万行程度のサイズになっている。P3L で記述されたプログラムは、このトランスレータにより C++ プログラムに変換され、gnu g++ でコンパイルされる。リンク時には、P3L が生成した補助関数と Kala のライブラリが加えられる。

5 今後の課題

現在の実装では Object のディスクへの掃きだしを行なっていないため、データベース内でのナビゲーションを続けると仮想記憶がいつかは尽きてしまう。P3L はテーブルを介した間接参照を行なっていないので、この問題解決のためには、仮想空間に読まれたオブジェクトのうちディスクに書き戻してよいものを検出する必要がある。そのためにはスタック上にあるポインタを全て検出する必要があるが、スタック上のポインタをトレースするコストは非常に高いし、タグをつけることも不可能である。幸いスタック上の値を全てポインタとしてゴミ集めをする、Conversative collector [?] があるので、これを利用して GC と同時に掃きだしを行なうためのランタイムサポートを追加する。また、ポインタ書換えについて、P3L の方式と Wilson の方式の詳細な比較もおこないたい。

参考文献

[Hanson] Eric N. Hanson, Tina M.Harvey and Mark A.Roth, 'Experience in DBMS Implementation Using an Object-Oriented Persistent Programming Language and a Database Toolkit', OOPSLA91

[That] Satish M.Thatte:Persistent Memory;*Intl.* Workshop on Object-Oriented Database systems '86

[CACM] Charles Lamb, Gordon Landis, Jack Orenstein and Dan Weinreb:THE OBJECTSTORE DATABASE SYSTEM

[Wilson] Paul R.Wilson,'Pointer Swizzling at Page Fault Time: Efficiently Supporting Huge Address Space on Standard Hardware', Computer Architecture News, Vol 19, No.4 June 1991

[Kala] Sergiu S. Simme and Ivan Godard, The Kala Basket, OOPSLA91

[Bart] Joel F.Bartlett,'Mostly-Copying Garbage Collection Picks Up Generations and C++', Digital Equipment Corp, Western Research Center, Technical Note, TN-12,1989.