ポインタタグ（ＭＲＢ）による多重参照管理方式

近山 隆　木村 康則

（ＩＣＯＴ）

1．はじめに

PrologやPure Lispのような副作用を持たない言語では，同じ構造体データの一部を副作用的に書き替えることによって何回も利用することを，陽にソースプログラム中から指定することはない．これを単純に実装すると，もともとあった構造体と一部だけが異なる構造体データが必要になるごとに次々に新しいメモリ領域に構造体を割り付ける事になる．このため，ゴミ集めが頻繁に起きるのみならず，メモリアクセスの局所性が悪くなることからキャッシュのミスヒットやページフォールトが多くなり，実行速度面での損失も大きい．

LispでRPLACDなどの副作用のある機能を用いるのが常道になっているのは，メモリ消費を抑えるのが主な理由のひとつである．Prologでは，バックトラック時に不要になった構造体はスタック手法を用いて容易に解放／再利用できるし，変数の束縛が解けることによって暗黙のうちに同じ構造体の一部分を書き替えて再使用することもできる．同じ論理型言語でも，GHCのようなバックトラッキングのない言語では，メモリ消費の問題になんらかの方策を講じない限り，実用的性能を得ることは難しい．

メモリアクセスの局所性を確保するには，データ領域が不要になったことをその都度判定し，即座に再利用することが望ましい．このための手法としては，参照カウンタによる方法が一般的である．しかし，参照カウンタの管理にはメモリの読み書きが必要であり，そのオーバヘッドは小さくない．また，参照カウンタはデータオブジェクトそのものに付随するものなので，オブジェクトへのポインタを渡すだけで本体にアクセスする必要のないときにも，オブジェクト本体にアクセスしなければならない．マルチプロセサ環境でデータが多数のプロセサに共有されている場合は，共有データへのアクセスはできる限り避けるべきである．したがって，GHCのような並列処理を前提とした言語の処理系には参照カウンタによるゴミ集め方式は不適当である．

本稿では，オブジェクトではなくポインタの側に簡単な情報を持たせ，これを管理することによって，小さなオーバヘッドで多くの場合に良好な回収効率を実現する方式について述べる．

Multiple Reference Management by Pointer Tags (MRB)
T.Chikayama and Y.Kimura (ICOT)

2．ＭＲＢ方式

参照カウンタの問題点は，参照数の情報がポインタではなくポインタが指すデータの側にあることだった．そこで，ポインタの側に１ビットの余分な情報（多重参照ビット，Multiple Reference Bit，ＭＲＢ）を置き，それに以下のような意味を持たせて，参照カウンタと同様のインクリメンタルなゴミ集めを行なうことにする．

○：このデータへのポインタはこれ１本である

●：このデータへのポインタが他にもあるかもしれない．

構造体データへのポインタのＭＲＢは，図１のどちらいずれかのようになっているように管理する．

図１ 構造体へのポインタのＭＲＢ

ＭＲＢを正しく管理していれば，○ポインタが不要になった場合，他に同じデータを参照しているポインタはないことがわかる．そこでポインタの指す先のメモリ領域をただちに解放／再利用することができる．

構造体データだけについてなら，ＭＲＢの管理は容易である．新たに構造データを生成したときには，○で指す．引数に渡されたポインタを複数のゴールに，または同じゴールでも別々の引数として複数回渡す場合（ポインタの複製時）は●にしてから渡す．もともと●ならそのままでよい．下記のようなGHCプログラム中の変数ＸやＹについては，引数に渡されたままのデータをボディ部のゴールに渡してもよいが，ＺやＷについては●に変えてから渡さなくてはならない．

```
p(X, [Y|Z], W) :- true |
        q(X, W), r([Y|Z], Z, W).
```

構造データを未定義変数とユニファイする場合，その変数が他に複数の参照パスを持っていると，そのデータ構造にもこのユニフィケーションによって複数の参照パスが生じることになる．○の指すものが回収可能であるという性質を保つには，具体化時にその変数を指しているのは他には○が１本だけ，他にはない，という時にだけ○に具体化，

そうでなければ●にして具体化するようにすればよい．そこで未定義変数へのポインタには：

○：他に○があれば，それが他の唯一のポインタ

●：他に○があっても，まだ他に●があるかも知れないという意味のＭＲＢを持たせる．

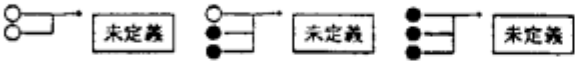

図２ 未定義変数へのポインタのＭＲＢ

引数に受け取らなかったような新しい変数を，２本以下のポインタだけで指すように作る場合は，変数は○で指してよい．最初から３本以上のポインタを持つように生成する変数は，最初から●で指す．下記のプログラム中の変数Ｘ，Ｙは○で指すように生成し，Ｚは●で指すように生成する．

```
p :- true | q([X|Y], Y, Z), r([X|Z], Z).
```

ポインタの複製時に○を●にするのは，データ構造へのポインタと同様である．したがって，ポインタの複製時には複製するものが具体化済かどうかをデリファレンスして調べる必要はない．

変数の具体化の際には，変数へのポインタ／データへのポインタの双方が共に○の場合だけ○で，それ以外の場合は●で具体化する．

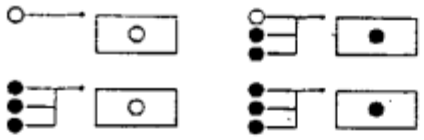

図３ 具体化済変数へのポインタのＭＲＢ

図３からわかるように，●で指される間接語セルには複数の参照パスがあるかも知れない．そこで，デリファレンス時に途中の間接語にひとつでも●のセルがあったならば，デリファレンス結果は●とし，他のパスから参照されるときのために回収を控えなくてはならない．

構造体の要素の取り出しの際も同様で，構造体全体が●で指されていれば，要素を取り出した後にも構造体全体への参照パスが残っているかもしれないので，その要素も構造体経由で後に参照される可能性がある．このため，構造体から取り出した要素へのポインタは，構造体全体が○で指されており，要素自体も○になっているときのみ○にしてよいことになる．

３．ＭＲＢ管理のオーバヘッド

ＭＲＢの管理には以下のようなオーバヘッドがある．

（１）余分なリソースが必要であること

１語に付き１ビットづつの余分なビットが必要になる．これはメモリ語のみならず，レジスタについても同様である．

（２）余分な管理が必要であること

デリファレンス，構造体要素の取り出し，変数の具体化，ポインタの複製などの際に実行時管理オーバヘッドがある．

４．ＭＲＢ管理のメリット

ＭＲＢ管理のメリットには以下のようなものがある．

（１）インクリメンタルＧＣ

① ワーキングセット縮小：インクリメンタルにゴミ集めができるためワーキングセットが小さくなり，キャッシュヒット率の向上などの利益をもたらす．これはメモリ共有型のマルチプロセサシステムでは特に重要である．

② 並列ＧＣ：メモリ共有型並列処理システムで，ＧＣも並列に行なうのは簡単ではない．ＭＲＢによるＧＣは自然に並列に行われる．

③ レスポンスの向上：疎結合マルチプロセサで，１台のＰＥがＧＣを始めると，その間他のＰＥからのアクセスに答えることはできないことがある．これが処理のネックになって並列性が上がらなくなる可能性がある．ＭＲＢによるＧＣを行なえば全体的なＧＣの頻度を減らせるので，長時間に渡って返答ができないことが少なくなる．

（２）同じ構造体の再利用

① 書き替え可能な配列：副作用的でない書き替え可能な配列を実現する場合，通常はマルチバージョン配列構造（連想リストと配列を組み合わせたような形式）を用いる必要があるが，元の配列が○で指されていれば他に参照パスはないのだから，直接を書き替えてよい．

② 部分的な構造体の再利用：回収可能な○で指された構造体の一部だけを変更して再利用することもできる．たとえば，リストのcar部が同じでcdr部だけが異なるようなものが必要でもとのリストセルが回収できる場合は，cdr部のみを書き替えて再利用することができる．

５．おわりに

ポインタに１ビットの多重参照管理情報を持たせることによって，インクリメンタルに不要な構造体データを回収する方式を示した．現在この方式に基づく実験処理系を用いて，評価を行なっている．

【参考文献】

T. Chikayama and Y. Kimura, "Multiple Reference Management in Flat GHC", Proc. of ICLP 87, Melbourne, 1987.