永続的プログラミング言語のためのコピー方式ゴミ集めの実行時コストについての考察

4G−10

鈴木慎司 喜連川優 高木幹雄
東京大学 生産技術研究所

1 はじめに

[1]において述べたように、Persistent Progrmmming Language(PPL)の実装に際しては、仮想記憶の管理に十分に注意を払う必要がある。これは、プログラマの生産性を向上させるとともに、ページバッファとオブジェクトバッファの統一的な管理を行うことでメモリの効率的利用をはかるためである。この仮想記憶の管理とは、永続オブジェクトのディスクから仮想記憶への移動,および有効に使われていないメモリを自動的に回収することである。プログラムからアクセス不可能なメモリを回収する技術は、ゴミ集め(Garbage Collection)として知られ、これまで多くの方式が考案されている。PPLにおいては、ゴミ集めに加え、参照は残っているもののアクセスが行われないようなオブジェクトを仮想記憶内から追い出すことが重要である。

我々は、現在実装中のコンパイラにおいて2つのポインタの実装方式を検討している。一つは、[2]で紹介した仮想アドレスを基にした実装方式である。この方式はプログラムの実行速度に与える影響が少なく、C言語との親和性も良いのであるが、コピー方式のゴミ集めを行うことが難しい。[3]のMostly Copying Colletionが適応可能ではあるが、この方式は、ヒープ領域を小さなブロックに分割し、明らかにポインタとわかる参照しかされてないブロックのみを移動させるという方式のため、どの程度のコピーの効果が得られるのか不明である。またヒープ上のオブジェクトのポインタの位置は厳密にわからる必要がある。そして2つめの方式は、ポインタをセレクタと呼ぶオブジェクト識別子とオブジェクト内のオフセットとして表現する方法である。この方式では、オブジェクトがおかれた位置は、メモリ内のオブジェクトテーブルに保持される。本稿では、その方式の解説と、実行性能の低下に繋がる原因、Toyプログラムでの性能評価結果について報告する。

2 ゴミ集めの方式

LISP系の言語処理系においては、ポインタ表現(仮想記憶アドレス)を直接用いて、参照が実現されている場合が多い。これは、LISP系の言語では動的な型付けが基本となるためにポインタにタイプタグ付けられており、ポインタ型のデータと、そうでないデータの区別を厳密に行うことができるからである。曖昧なポインタを許さないことで、オブジェクトのコピーを行った時には、それらのポインタを新しい場所にあわせて更新することができる。前章でのべたように、曖昧なポインタを用いながらもコピー方式のゴミ集めをする方法も考えられている。しかし実装の複雑さや、内部ポインタの存在に起因する安全性の問題を考慮し、我々は別の方式を取ることを考えている。

LISPと対象的にSmalltalkの処理系においては、オブジェクト間の参照はオブジェクトテーブルと呼ばれるテーブル(このエントリを以後サンクと呼ぶ)を介して行われる。そして、オブジェクト識別子は、このテーブルへのインデックスとなる。このように、間接参照を設け、オブジェクトが移動したときには対応するサンクを書き換えることで、曖昧なポインタが存在する場合にもコピー方式のゴミ集めを行うことができる。C言語の特性から生じる特徴は2点である。

1. ポインタがオブジェクトの先頭だけでなく、中間をも指しうるので、オブジェクトの識別をするセレクタだけでなく、オブジェクト内でのオフセットもポインタ内に持つ必要がある。
2. スタック上のデータの型(ポインタ/非ポインタ)がわからない。また、型付けのされないデータがヒープ中に存在する。ポインタの更新の問題は間接参照により解決されるが、オブジェクトの生死を厳密に判定することができない。そこで、スタック上のデータやヒープ上の型付けされていないデータが全てオブジェクト識別子だと仮定してゴミ集めを行う。指している対象によって明らかにポインタではないと判断できるものはゴミ集めの対象から除外する。

この方式では、存在するオブジェクト1つあたりに一つのサンクを用意する必要がある。しかし我々の実装では、スタック上につくられたオブジェクトのためのテーブルエントリはオブジェクトと同時に、スタック上に生成する。スタック上のthunkはオブジェクトテーブルと別の場所に存在するため、ゴミ集めの対象からはずれるが、thunkが指しているオブジェクトはスタック上のオブジェクトであるから、ゴミ集めの対象とする必要はない。また、thunkを生成する必要があるのは、スタック上のオブジェクトのアドレスが、関数外部に渡された場合だけであり、関数内のオブジェクトをアクセスする際にはthunkをを介する必要はない。

3 実行性能への影響

このようなポインタの表現形式を採用することで、実行性能への悪影響が生じる。その原因には以下のようなものがある。

3.1 ポインタサイズの増加

すでに述べたように、C言語の処理系をオブジェクトテーブルを用いて行った場合、ポインタはセレクタとオフセットの組として構成する必要がある。すると、関数間でのポインタの受渡しの際に、より多くのCPUサイクルが必要になる。また、データサイズが大きくなり、仮想記憶を圧迫することも考えら

[0]On runtime overhead of copying collection for a persistent programming lanugage
S.Suzuki, M.Kitsuregawa, M.Takagi
Institute of Industrial Science, University of Tokyo

れる。増加量は概して(ポインタのサイズ * ポインタの数 + thunk のサイズ + オブジェクトのサイズ) である。もちろんプログラムコードサイズも増加する。

3.2 間接参照の必要性

ポインタを通してオブジェクトをアクセスする際にメモリサイクルを一つ余分に必要とする。CPU の速度とメモリの速度が益々開きつつある今日この問題は深刻である。しかし、後に述べるコンパイラ最適化によりかなり改善をはかる。

3.3 Thunk の確保、解放

基本的に、オブジェクトが生成されるたびに thunk を確保する必要がある。そのために大きなコストが必要だとすると、それはスタックをデータ領域として多用する C 言語にとって大きなオーバヘッドとなる。スタックフレーム上に生成されたオブジェクトは、当該関数からは直接参照して構わないし、スタックフレーム上に割り振れば良い。そうすれば thunk の生成、消滅必要なコストは殆んどなく、thunk をオブジェクトのアドレスで初期化する必要があるのみである。

4 間接参照のコストの評価

実際にこのポインタの実装方式と採り入れることで、どの程度のオーバヘッドが生じるのかを小さなプログラムで検証してみた。プログラムは、図 1 のような連結リストを作成し、入力した語をリストのなかから探しだして、係数フィールドのカウントを 1 だけ増加するものである。この操作を one,two,three,... という１０語について、１０００回行った時間を図 2 に示す。また、ポインタサイズの影響を調べるために、offset をポインタの中に用意しない場合も計測してみた。(32bit handle) この結果からすると、生じた速度低下のうち 30% 程度がポインタサイズの影響であることがわかる。全体では 5 割程度性能が落ちている。また、命令数でみてみると、通常のポインタを用いたものが 186 命令、間接参照のものが 390 命令となっている。実験には i386 に 64KB のキャッシュを備えた計算機を用いた。

5 最適化

上にのべたように、間接参照をおこなうことで命令数およびメモリアクセス回数が増加する。セグメンテーションの機能を備えた CPU を用いることで、この問題に対処することができるが、多くの CPU はその機能を備えていない。しかし、コンパイラによる最適化により命令数、メモリアクセス回数の両方を減少させることができる。具体的には、

1. 共通式削除のように、同一のセレクタを参照 (Dereference) している式が連続した場合に、最初のアクセスの値を後の式でも再利用する。
2. ループ内にセレクタのアクセスをしている部分があり、ループ内でセレクタの値が変わらない場合には、セレクタのアクセスをループの外に移動する。
3. (2) と同様に、呼び出した関数内でセレクタの値が変わらないことがわかる場合には呼びだし前にセレクタの Dereference を行ない、関数へは通常のポインタを渡す。

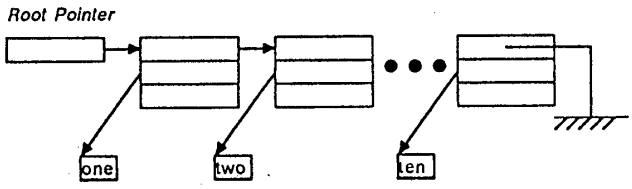
これらの最適化は、よく知られたコンパイラ最適化に似たものであるが以下のことに注意しなくてはならない。それは、同じ式でも状況によって値を再利用できるかどうかが変わることである。なぜならポインタの参照によって永続オブジェクトが読み込まれたときや、プログラムがメモリ要求を出した場合にはゴミ集めが起こるからである。例えば、p がポインタ変数のとき、 *p+2; i=2; *p+2; という並びでは最初の *p と 3 番めの *p の値は共有できるが、 *p+2; i=*q; *p+2; では共有できない。解決策としては、*p と *q のサイズ分だけのメモリが残っていることを確認してから 3 つの式を評価することが考えられる。CPU 内にユーザプログラムから使用可能なフラグビットがあれば、GC が起こったときにこのフラグをセットすることで選択的に値の共有を行うことも考えられる。

6 まとめ

間接参照を利用したコピー方式について述べた。この方式は型付けされていないオブジェクトがヒープ上に存在する場合にも利用できるという利点がある。今後、実行性能上の問題を解決するために上に述べた最適化をコンパイラに実装してゆく。

参考文献

[1] 鈴木, 喜連川, 高木 ' 永続性を備えた C 言語における可動オブジェクトの実装について ' 第 43 回全国大会予稿
[2] 鈴木, 喜連川, 高木 ' 永続オブジェクト参照方式とその評価 ' 第 42 回全国大会予稿
[3] Joel F. Bartlett, 'Compacting Garbage Collection with Ambiguous Roots' Technical Report WRL Research Report 88/2



Linked List in the Sample Program
Fig. 1

STANDARD POINTER	128Bit Handle	64Bit Handle	32Bit Handle
972ms	1690ms	1493ms	1363ms

Execution time of the sample program
Fig. 2