Bulletin of Aichi Univ. of Education, 59（Natural Sciences）, pp. 19–22, March, 2010

Objective-C による Lisp 言語処理系のメソッド呼び出しの最適化

安本太一

情報教育講座

Optimization of Method Calls on Lisp System by Objective–C

Taichi YASUMOTO

Department of Information Sciences, Aichi University of Education, Kariya 448 - 8542, Japan

1 はじめに

Objective-C 言語 [1, 2, 3, 4, 5] を用いて Common Lisp 言語処理系 [6, 7, 8] を実装している [9]. Objective-C 言語のオブジェクト指向の機能と Lisp 言語は，双方とも，動的な特徴をもっている．束縛を実行時に解決するというこの特徴は，柔軟性という点で評価されているが，一般には実行前に束縛が解決されている C 言語プログラムなどに比べて実行効率は必ずしも良くない．

Objective-C 言語 (以下簡潔のために Objective-C という) のオブジェクト指向の機能を生かし，Objective-C のインスタンス変数やメッソッドを使って，Lisp 言語処理系を実装することは，コンパイラの実現，処理系の保守，OS が提供する API との親和性という点では望ましい．しかしながら，組み込み関数やコンパイルされたユーザ定義関数のコードにおいて，生真面目に動的に束縛を解決するのは実行効率の点で懸念が残る．

本稿では．Objective-C の柔軟性の利点を享受しつつも，試験的なものではあるが，この実行効率の低下を抑える方法を一つ提案する．その後，簡単な評価実験とその結果を報告し，提案する方法を実装するのに適した C 言語の構文の拡張について言及する．

2 Objective-C のオブジェクト指向機能における束縛の解決

Objective-C では，メソッドやインスタンスの束縛は，実行時における探索によって解決される [10]．例えば，メソッドの実行に際しては，文字列表現されたセレクタの名前をキーにして，クラスが保持しているメソッドのリストからメソッド実装を探索し，メソッドが存在したらその実装を取り出すということを行っている．この文字列は唯一性が保証された文字列 (C 言語の演算子 ==を用いてアドレスで比較可能な文字列)，一度探索した結果はキャッシュされるといった，実行効率低下への配慮はされている．しかしながら，このような配慮があっても，C 言語などにおける関数直接呼び出しに比べれば，メッセージ送信は遥かに遅い．

Objective-C では，実行効率の低下と引き換えに柔軟性を確保していて，実際に定義されていないメソッドを呼び出すコードを記述してコンパイルできたり，(メソッド実装を置き換えることによって) メソッドを実行する際にフックを仕掛けることが可能といった，かなり自由なことが許される．

現実の Objective-C のプログラミングでは，プログラムの全てを徹底的に Ojbective-C のオブジェクト指向の機能を用いて記述する必要はなく，柔軟性を必要としない部分は C 言語で静的に記述することができる．標準の C 言語ライブラリも利用できる．MacOSX が提供する API [11] を利用する場合にオブジェクト指向機能を使うことを要求される場合を除き，プログラマは，柔軟性と実行効率を必要に応じて選択できる．

3 Objective-C による Lisp 言語処理系の実装

筆者が行っている Objective-C による Lisp 言語処理系の実装 [9] では，Lisp データへのアクセスと，記号が保持している変数束縛や関数束縛へのアクセスは，Objective-C のオブジェクト指向の機能を用いて実現されている．

前者は，Lisp のデータを NSObject の (間接的な) サブクラスとして実現することにより，Objective-C の方からも Lisp のデータが操作できるようにするためである．一つのプログラムを，Lisp 言語と Objective-C を使い分けて記述することができることを目的としている．

後者は，コンパイラの実現のためである．コンパイルされて機械語に翻訳された Lisp のコードの実行に必要な変数や関数の束縛の解決を，リンカではなく，Objective-C のプログラムのランタイム環境を使って実現する．

図 1 に，組み込み関数の実装例として，関数+の実装を示す．`LPlus` が+の本体である．処理の制御につい

ては，C 言語で記述しているので，Objective-C のオブジェクト指向の機能を使うことによる実行効率の低下はない．加算の部分は，最終的に [x add: y] というメッセージ送信で実現されているので，素朴に実装するならば，毎回メソッド実装探索を行うことになる．このメッセージ送信を行うために，number_plus という関数呼び出しが行われているが，これは，マクロで実現してもよいので，関数呼び出しのコストは本質的な問題ではない．

```
#import "object.h"

LObject* number_plus(LObject *x, LObject *y)
{
  return([x add: y]);
}

@interface LPlus : LCfun
@end

@implementation  LPlus
- (void)cf_self{
  int i,j;

  j = vs_top - vs_base;
  if(j == 0){
    vs_push(small_fixnum(0));
    return;
  }

  for(i = 0; i < j; i++)
    check_type_number(vs_base[i]);
  for(i = 1; i < j; i++)
    vs_base[0] = number_plus(vs_base[0],
                             vs_base[i]);
  vs_top = vs_base+1;
}
@end
```

図 1: 組み込み関数+の定義

4 メソッド呼び出しの高速化

メソッド呼び出しの高速化を行う方法としては，キャッシュ，すなわち，メソッド実装を取り出して，その実装を使い回す (メソッド実装を直接呼ぶ) ことがあげれられる．メソッド実装探索は 1 回だけ行えばよい．

ただし，単純にこのようにすると，関数の再定義が行われた場合に，対応できない (再定義が反映されない) ので，Lisp 言語の対話性を損なう．

組み込み関数 (やコンパイルされたユーザ定義関数) であれば，その関数の入り口で，使用されているメソッド実装の探索を行い，その結果をキャッシュするという対応が考えられる．組み込み関数 (やユーザ定義関数) の実行中は，これらの関数が利用している関数の定義は不変という前提である．関数定義や関数の再定義は，トップレベルのみで行われるということであれば，合理性はある．ユーザ定義関数が再定義されたり，組込み関数が保守のために再定義されたりした場合は，キャッシュを破棄すればよい．

しかし，この方法だと，組み込み関数 (やユーザ定義関数) の入り口で，毎回，その関数が使用している (関数に関連する) 全てのメソッド実装の探索を行うので，必ずしも効率が良いとはいえない．関数定義に出現している関数が全て使われるとは限らないことから，全てのメソッドが使われるとは限らないからである．

この欠点を克服する提案は後で述べることにし，本方法の素朴な性能評価を先に行う．

5 評価

メソッド実装の直接呼び出しの効果を評価するために，Apple 社製コンピュータ Mac 上で，下記の Lisp 言語の式を 10000000 回評価した時の実行時間を計測した．

```
(+ 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)
```

+は組み込み関数であり，先述のように Objective-C で記述され，add:というメッセージ送信をしている．上記の式では，メッセージの送り先は Fixnum(固定長整数) に限定されることが実行前にわかる．実行時間を計測するのに用いた環境は，MacPro (Intel Dual Core Xeon 2.66GHz × 2, MacOSX10.5.8) と MacBook(Intel Core2 Duo 2.0GHz × 1, MacOSX10.6) である．その結果を，表 1 に示す．表中のメッセージ送信というのは，素朴に LFixnum のオブジェクト (Objective-C で実現されている Lisp データ Fixnum のオブジェクト) に add:のメッセージ送信を行うことを示す．直接呼出というのは，いったんメソッドを探索したら，その結果得られたメソッド実装を直接呼び出し続けることを示す．

双方の実行環境においても，直接呼出は，メッセージ送信より，実行時間が短縮されていて，それぞれ 90%，95%になっている．

MacOSX10.5.8 は 32 ビットモード実行形式 (i386 用実行形式)，MacOSX10.6 は 64 ビットモード実行形式

表 1: メッセージ送信と直接呼出の実行時間の比較

実行環境	メッセージ送信	直接呼出	比
MacPro MacOSX10.5.8	61.7	55.8	0.90
MacBook MacOSX10.6	44.4	42.3	0.95

(単位は秒)

(x86_64 用実行形式) を基本としている．x86_64 の方が使えるレジスタが多かったり，後者の OS は前者の改良版となっていて Objective-C で記述されたプログラムのランタイム環境が最適化されている．CPU の能力が低いのに MacBook の方が絶対的に実行時間が短いのはこのような状況によるものと推測される．

ランタイム環境が最適化されている MacOX10.6 の環境でも，実行時間の短縮がみられることから，ランタイム環境そのものにメソッド実装のキャッシュが用意されているといってもメッセージ送信のコストは高く，メソッド実装の直接呼び出しを行う効果があることがわかる．

6 考察

高級言語で実現可能かどうかは別として，本稿の提案には最適化の余地が残っている．

先述の評価実験を行ったときの，メソッド実装の直接呼び出しの実装では，add:メソッドの実装を C 言語の static 変数 (型は関数へのポインタ) に格納している．この static 変数は NULL に初期化されていて，メソッド実装を探索したら，メッソド実装へのポインタを格納する．メソッド実装の再定義 (関数の再定義などによって生じる) が行われたときは，この static 変数は，NULL に初期化されることになっている．

したがって，関数+の入り口で，メソッド add:の実装が得られているかどうかを，毎回チェックしている．上記の static 変数が NULL なら，メソッド実装を探索した後に．+の本来の処理に進む．さもなければ，探索はせず，+の本来の処理に進む．加算を行う際は，static 変数に格納されたメソッド実装を呼び出す．メソッド実装がいったん求められれば，入り口におけるこのチェックは不要である．少しでも実行効率を向上させる観点からは，このチェックによるオーバヘッドを省きたい．

2 回目以降の+の呼び出しにおける，+の入り口でのチェックを省く方法として，メソッドの実装を格納する static 変数の初期値を NULL ではなく，次のようなブロックにすることが考えられる．このブロックは，

1. セレクタ名 (唯一性のためにコンパイルされたメソッドの名前) からメソッド実装を探索する．
2. 探索したメソッド実装を直接呼び出す．
3. 探索したメソッド実装を static 変数に代入する．

というようなことを行う．

Apple 社が C 言語の標準を検討する ISO ワーキンググループに提案している Blocks という構文 [12, 13] を用いると，このブロックは，図 2 のように表現できる．

```
//impはメソッド実装へのポインタ
//idはメッセージ送信先，SELはセレクタ
//...はメッセージの引数
static IMP (^imp)(id, SEL, ...)
  = ^IMP(id, SEL, ...)
    { セレクタ名からメソッド実装を探索する;
      メソッド実装を直接呼び出す;
      imp = メソッド実装;
    };
```

図 2: Blokcs を用いたメソッド実装を格納する変数の初期化

imp は，一度呼び出された後は，メソッド実装を単に指すだけなので，先述のメッソッド実装が求まっているか否かのチェックは不要となる．

Blocks という構文を用いたこのような実装では，組み込み関数やコンパイルされたユーザ定義関数の入り口で，その関数に含まれている全てのメソッド呼び出しのメソッド実装を求める必要はなくなる．(制御の流れの結果として) 呼び出されないメソッドについては，メソッド実装の探索は全く行われず，CPU 時間の浪費はない．同一のメソッドが，複数の箇所で呼び出されていても，最初にどの箇所でメソッド実装の探索を行うか検討する必要もない．

MacOSX10.6 が 2009 年 8 月 28 日に発売され，最近，Blocks が実際に使えるコンパイラが一般に提供された．実際に Blocks を使った実装を試そうとしたが，Blocks の中でセレクタ名のための`@selector()` ディレクティブを使うと，(特別な宣言なしに)Blocks の外の変数への代入は許されないといった Blocks のいくつかの制限に抵触するらしく，コンパイルエラーになった．Blocks は，名前のない関数という側面の他に，(Blocks が定義された時点での自動変数へのアクセスが可能といった) クロージャーの機能を実現していることもあり，このような制限があるのであろう．本稿で言及している Blokcs を使った実装は，Blocks に完全なクロージャーまでの機能を求めていないので，残念である．

7 まとめ

Lisp 言語における関数呼び出しが，最終的にObjective-C におけるメッセージ送信になる場合，メソッド実装をキャッシュして直接呼び出せば実行効率が向上するというデータを示した．本稿で提案するキャッシュの方法は，Lisp 言語の関数の実行状況に即したものである．Lisp 言語には，+や map といった列として与えられた引数の要素に対して，繰り返し同じ処理を行う組込み関数が多くあるので，メソッド実装をキャッシュする効果が寄与するケースは多いと推測される．

また，メソッド実装のキャッシュを効率良く実現するための構文として，Apple 社が提案中の C 言語の文法拡張 Blocks を検討した．現在提供されている Blokcs では，Blocks の制限により，メソッド実装をキャッシュして直接呼び出しすることの実装には使えない．しかし，無名の関数だけに機能を絞った “軽量な Blocks” が用意されれば，メソッド実装のキャッシュに使える見込みがあることを示した．Blocks は提案されてから日が浅く，Objective-C のベースとなっている C 言語標準に取り入れらるかどうかは未定である．本稿で述べた他にも，クロージャとしての機能までは要求しない事例はあると思われるので， Blocks が C 言語標準に取り入れられるときに，軽量な Blocks も用意されることを期待したい．

参考文献

[1] B. J. Cox, and A. J. Novobilski : *Object-Oriented Programming Approach 2nd Edition*, Addison-Wesley (1991).

[2] B.J. コックス, A.J. ノボビルスキ 共著, 松本正雄訳 : オブジェクト指向のプログラミング 改訂第 2 版 ソフトウェア再利用の方法, 新紀元社 (2004).

[3] *Object-Oriented Programming with Objective-C*, Apple (2007).

[4] *The Objective-C 2.0 Programming Language*, Apple (2008).

[5] 萩原剛志 : 詳解 Objective-C 2.0, ソフトバンククリエイティブ (2008).

[6] Steele, G. L. Jr. : *Common Lisp the language*, Digital Press (1984).

[7] Yuasa, T. and Hagiya, M. : *Kyoto Common Lisp Report*, Teikoku Insatsu Publishing (1985).

[8] Yuasa, T. : Design and Implementation of Kyoto Common Lisp, *Journal of Information Processing*, Vol.13, No.3, pp.284–295 (1990).

[9] 安本太一 : Objective-C による Lisp 言語処理系の試験的実装, 愛知教育大学研究報告 自然科学編, 五十八輯, pp.21-25 (2009).

[10] 木下誠 : Dynamic Objective-C, ビー・エヌ・エヌ新社 (2009).

[11] 木下誠 : たのしい Cocoa プログラミング [Leopard 対応版], ビー・エヌ・エヌ新社 (2008).

[12] Grast, B. : *APPLE'S EXTENSIONS TO C*, http://open-std.org/jtc1/sc22/wg14/www/docs/n1370.pdf, March 10,2009.

[13] 千種菊里 : Mac 解体新書 第 2 回 Blocks—GCD の根幹を担う文法拡張, 月刊アスキードットテクノロジース, Vol.14, No.9, pp.156–pp.164(2009).

（2009年 9 月17日受理）