決定木を用いたメソッドディスパッチャのメタレベルでの実装

A Meta-level Implementation of Method Dispatcher Using Decision Trees

池田博之† 楢崎修二† 吉田紀彦‡
Hiroyuki Ikeda Shuji Narazaki Norihiko Yoshida

1. はじめに

オブジェクト指向言語における、メソッド照合の基本戦略は線形探索であるが、これではマルチメソッドの場合、計算量が大きくなってしまいシステム全体における処理速度に大きな影響を与えてしまう。そのために、メソッドキャッシュを用いたり、コールサイトを別の形で表現したりすることによって効率の良いメソッド照合を行うための様々な研究がなされてきた。

本稿では、メソッド照合を木の上で行い、シグネチャに類似性が認められるあるメソッド列に対して、シグネチャの同一部分から次に呼び出されるメソッドの木をあらかじめ計算しておき、次のメソッドを呼び出すときはその木の残りの部分のみを計算することによって、メソッド照合の計算量を軽減する手法を提案し、実装する。

2. 背景

マルチメソッドの場合、総称関数から目的のメソッド本体を得るためには、線形探索で一つ一つメソッドを取り出し、適用可能かどうかチェックし、適用可能であればそれまでに見つかった適用可能なメソッドと比較して、より特定的な方を残し、最後まで見ていって残ったものを使うという方法が使われている。しかし、これでは計算量が大きく、メソッドの総数を n とおくと、O(n)となってしまう。そこで、以下のような方法が提案され使用されている。

2.1 メソッドキャッシュ

一度呼び出したメソッドをソフト的なキャッシュに格納し、次に呼ばれるメソッドはまずキャッシュ内を照合し、失敗すれば総称関数への照合を行う[1]。

- ハッシュドキャッシュ
  上記のキャッシュの方法だと、キャッシュの大きさが１つしかなかった場合、異なるメソッドが交互に呼び出されるようなことがあれば、全く無意味になってしまう。また、ある程度の大きさがあったとしても線形に照合していったのでは、計算量がキャッシュの大きさに比例してしまう。ハッシュドキャッシュはハッシュ法を利用し、メソッドの名前やシグネチャをキーとした呼び出しおよび格納を行う。この方法ではキャッシュ内に目的のメソッドがあればその計算量は O(1)である。
- Polymorphic Inline Cache(PIC)[2]
  ハッシュドキャッシュはメソッド全体に対するキャッシュであるが、PIC はレシーバごとにキャッシュを用意する。また、これは、ハードウェアキャッシュのように固定長ではなく、新たに呼び出されたメソッドはそのつどキャッシュへ追加されてゆく。

2.2 二分木

2.1 節で挙げた方法は、一度呼び出されたメソッドは再び使用される確率が高いので、それを覚えておいて、メソッドが呼ばれたときには計算量の大きい照合ルーチンを回避し、キャッシュ内のそれを利用しようという考え方であった。

総称関数が線形で表現されているために計算量が大きくなるので、総称関数を二分木で表現することによってメソッド照合の計算量を軽減しようとする方法がある[3]。これによる計算量は木が平衡していれば O(log n)となる。

3 概要

今回の提案する手法では、総称関数の表現に注目し、さらにそれを利用した最適化の手法を考えた。

```
Foo(int, char)
Boo(int, int)
```

例１ シグネチャに類似性のある例

あるプログラムにおいて、シグネチャに類似性のあるメソッド列が出現しているとする。ここでシグネチャの類似性とは例えば以下のような特徴を有するものとする。

上記の例では、第一引数のクラスが同じなので、それぞれのメソッドを独立に配分するのではなく、そのシグネチャの類似性を利用して、照合の計算量をさらに減少させることによって、更なる高速化が可能であると考えた。どのように利用するかというと、あらかじめ生成されているそれぞれの総称関数のメソッド決定木をシ

†長崎大学 Nagasaki University
‡埼玉大学 Saitama University

グネチャの共通部分まで、あらかじめ辿っておき、ポインタを配列へ格納しておく。次のメソッドが呼ばれた時点で、ポインタが指す残りの部分木のみを実行してゆくことで、本来なら木全体を実行するよりも計算量を軽減することができる。先ほどの例でいえば、2 番目に呼ばれるメソッド Boo は第一引数の部分の照合を省略することができる。

3. 実装

今回提案する手法を、Tiny-CLOS[4]に実装した。Tiny-CLOS は scheme で書かれており、Metaobjct protocol(MOP)[5]の概念が導入されている。Tiny-CLOS の MOP はシンプルな作りになっていて、メタオブジェクトクラスのサブクラスを作り、それを扱うメソッドを MOP が示して 2 いる総称関数に追加するだけで、標準的な振舞いを拡張することができる。

まず、メソッド決定木を扱えるように、総称関数クラスを拡張した。メソッドが定義されるたびに、そのシグネチャとメソッドを総称関数に二分木の形で登録してゆくための関数 generate-decision-tree を作った。ここで、このメソッド決定木のメソッドを決定するための手順は、1) 登録したクラスの継承関係をみる、2)登録したクラスと等しいか比較する、3)シグネチャの終端かチェックする、という手順で行う。generate-decision-tree はこの手順に基づいて、クラスの継承関係をもとにソートしながら、If 文を生成するようになっている。

類似度の高いメソッド列に対しては、コンパイル時に、最初に使われたメソッドのシグネチャを使用して、次に呼ばれる総称関数を辿れるところまで辿り、残った部分木へのポインタを、最初のメソッドとペアにして配列へ格納しておく。そして、実行時には、最初のメソッドおけ違算したあと、次のメソッドを呼び出すときには、総称関数を利用するのではなく、配列内の部分木を利用すれば木全体を辿るよりも速くメソッドを決定することができる。

4. まとめ

類似度の高いメソッド列のシグネチャの共通部分をあらかじめ計算しておくことで、メソッド照合の計算量を軽減させる方法を提案し、実装した。

今後の課題として、今回インタプリタに実装したので、コンパイラへの実装。今回の手法が適応可能なメソッド列がどのくらいあるかその検証、共通部分の計算方法の拡張。などがある。

参考文献

[1] D. Unger and D. Patterson, “Barkeley Smalltalk: Who Knows Where the Time Goes?” in G. Krasner, ed., Smalltalk-80: Bits of History and Words Advice. Addison-Wesley, Reading, MA, 1983.

[2] Holzle U., Chambers C. and D. Ungar., Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches. ECOOP'91.

[3] O. ZENDRA, D. COLNET and S. COLLIN, Efficient Dynamic Dispatch without Virtual Function Tables The SmallEiffel Compiler. ECOOP'97.

[4] http://www2.parc.com/csl/groups/sda/projects/oi/

[5] http://www.elwoodcorp.com/alu/mop/index.html

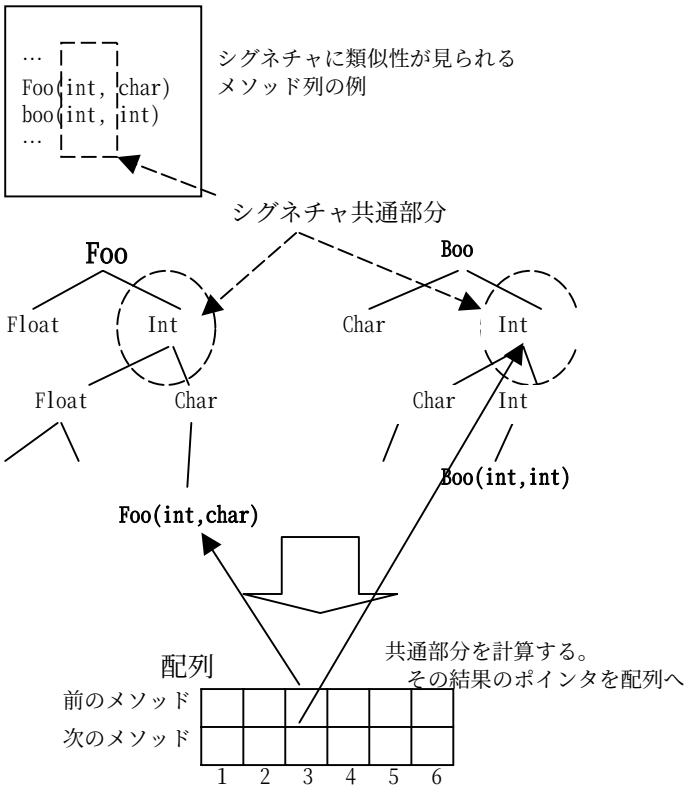

図 1 メソッド決定木を用いたメソッド配分