情報処理学会論文誌　プログラミング　Vol.6　No.2　45–53 (Aug. 2013)

実行速度を考慮した実装法による細粒度でのコード再利用のためのメソッド内メソッド

平松 俊樹[1,a)]　佐藤 芳樹[2,b)]　千葉 滋[2,c)]

受付日 2012年11月13日, 採録日 2013年5月10日

概要：プログラミングにおいて，ソースコードの再利用は有益である．しかし，Java のようなオブジェクト指向言語の多くではブロック単位でのコードの再利用をサポートしていない．より細かい粒度でコードを再利用するために，あるブロックを別のメソッドとして定義すると，変数受け渡し，副作用の再現のためにコードが煩雑化，実行速度の低下が起こりうる．このような問題を解決するため，細粒度でのコード再利用のためのメソッド内メソッド（内部メソッド）を Java 言語に導入した．メソッド定義内に宣言可能なこの内部メソッドは，外側のスコープで定義されたローカル変数を参照可能で，かつサブクラスから上書きが可能なメソッドである．本研究では，低オーバヘッドで内部メソッドを実現するために，インライン展開やオブジェクト渡しの最適化を行う拡張 Java コンパイラを実装した．内部メソッドを用いたマイクロベンチマークにより，サブクラスでの内部メソッドの上書きがない場合，およびスーパクラスのローカル変数に対する副作用のない上書きである場合について，オーバヘッドが無視できる程度に抑えられることを確認した．

キーワード：オブジェクト指向，Java，クロージャ

Inner Method for Code Reuse in Fine-grained and Its Effective Implementation

Toshiki Hiramatsu[1,a)]　Yoshiki Sato[2,b)]　Shigeru Chiba[2,c)]

Received: November 13, 2012, Accepted: May 10, 2013

Abstract: We introduce a novel language construct called inner method in Java for fine-grained code reuse. Code reuse in software development is important. It can imporove the development cost by preventing users from writing a duplicate code repeatedly. However, typical object-oriented languages like Java only support code reuse in methods, not in code blocks. Inner method is a method that can be declared in method body, overridden by subclasses, and can refer to local variables defined in the outer scope. Using inner methods, we can reuse a piece of code by cut it out as a inner method and reuse the rest of the code by overriding the inner method. However, a naive implementation of inner method causes non-negligible overhead due to extra method calls and object passings between inner and outer methods. To avoid performance degradations, we have implemented a Java compiler that performs inline expansion and optimization of passing objects. Our experimental results show that overhead is negligible if there is no override of the inner method in a subclass or side effects for the local variables of the superclass.

Keywords: object-oriented, Java, closure

1. はじめに

プログラミングにおいて，ソースコードの再利用は有用

[1] 東京工業大学大学院情報理工学研究科数理・計算科学専攻
Department of Mathematical and Computing Sciences, Graduate School of Information Science and Engineering, Tokyo Institute of Technology
[2] 東京大学大学院情報理工学系研究科創造情報学専攻
Department of Creative Informatics, Graduate School of Information Science and Technology, The University of Tokyo, Bunkyo, Tokyo 113–8656, Japan

[a)] hiramatsu@csg.ci.i.u-tokyo.ac.jp
[b)] yoshiki@ci.i.u-tokyo.ac.jp
[c)] chiba@acm.org

である．コードを再利用することで，生産性の向上やコード中における同一コード片の散逸の防止が期待できる．オブジェクト指向言語においては，クラス間の継承関係を利用してメソッドやフィールドを再利用し重複したコードの実装を回避することが可能である．しかし，Java をはじめとする多くのオブジェクト指向言語では，ブロック単位のコードの再利用をサポートしていない．再利用の単位はクラスやメソッドであり，メソッド内の一部のコード片だけを繰り返し呼び出すことや，メソッドの一部分を変更し残りのコードを再利用することは難しい．

メソッドの一部分のコードを別のメソッドとして定義し直すようなリファクタリングも有効な代替案だが，いくつかの問題がある．まず，そのコード領域で使用されるローカル変数をすべて引数として渡すために，大量の引数が必要となり，コードの可読性が著しく低下する場合がある．さらに，そのコード領域にローカル変数への代入，すなわち副作用が含まれる場合，書き換えられたデータをメソッドの返り値や共有変数を介して元のメソッドへ反映させる必要も出てくる．メソッドは単一の返り値しか持てないため，複数の副作用を持つコード片の影響をうまく元のメソッドへ戻せない．また，すべてのローカル変数をグローバル変数やインスタンス変数として共有するのは性能やスコープの問題から現実的ではない．

我々はメソッド内に定義できるメソッド（**内部メソッド**）を Java 言語へ導入し，細粒度でのコードの再利用を実現する．内部メソッドは，内包するメソッドのスコープを継承しローカル変数への読み書きが可能であり，定義クラスのサブクラスから上書き（オーバライド）することもできる新しい言語機構である．これにより，メソッド内の一部のコードを切り出して繰り返し呼び出すような内部メソッド再利用や，さらには内部メソッドを上書きして内包メソッドの残りのコードを再利用するような内包メソッド再利用が可能となる．内包メソッド再利用によって，Template Method パターンに基づく再利用，すなわち内包メソッドに実装されたアルゴリズムの再利用が促進される．Template Method パターンとは，抽象メソッドを用いて実装したアルゴリズムを再利用するためのオブジェクト指向言語のよく知られたテクニックである．

また，内部メソッドを導入した Java 言語から，通常の Java へのコード変換を行うコンパイラを実装し，実行時のオーバヘッドを軽減させた．素朴な内部メソッドの実装では，ローカル変数の受け渡しのためのリストまたはラッパオブジェクトの作成や副作用の同期，多段のメソッド呼び出し等のために余分なオーバヘッドが発生する．一方，我々は，コード変換時の選択的なインライン展開と変数アクセスの最適化によって性能劣化を最小限に抑えるようにコンパイラを実装した．

本論文では，内部メソッド実装の妥当性を評価するために，マイクロベンチマークを用いたオーバヘッドの計測結果を報告する．内部メソッドの上書きがあり，上書きした内部メソッドが副作用を持つときの実行速度は，内部メソッドを用いずに通常のメソッドを用いたときと比較して 35%減少した．変数受け渡しのためのオブジェクトの作成をしないことがその原因であると考えられる．

以下，2 章ではメソッド内の一部分のコード再利用の重要性と難しさを具体的に例示し，3 章では上書き可能な内部メソッドを提案する．4 章で内部メソッドの実装方法について述べ，5 章では内部メソッドのオーバヘッドに関する実験について記す．6 章では関連研究について論じ，7 章で本論文をまとめる．

2. メソッドの部分的な再利用

一般にオブジェクト指向言語では，コードの再利用をそのコピーではなく，メソッド呼び出しによって実現する．メソッドはコードを再利用可能な形で構造化する言語機構である．クラス中に定義されるメソッドの動作は，サブクラスで上書きして変更できる．上書きにより，サブクラスはスーパクラスのコードを再利用しながら，一部の振舞いが異なったものとしてメソッドを拡張することができる．

メソッドを部分的に再利用できれば，より細かい粒度でのコード再利用が可能になる．たとえば，学生全体のリストから卒業年度や試験スコアのような条件を与えて，特定の学生リストを取得するような StudentSelector クラスを考える（**図 1**）．select メソッドには，学生の判定処理やその学生全体への適用処理がハードコードされているため，判定処理だけを合否チェックのために再利用したり，逆に学

```
1  class StudentSelector {
2    void select() {
3      List<Student> students = …
4      List<Student> passed = ,,,
5      double highestScore = 0.0;
6      double lowestScore = 100.0;
7      for (Student s: students) {
8        if (s.gradYear == 2011) {
9          if (s.score > 70)
10           passed.add(s);
11         if (s.score > highestScore)
12           highestScore = s.score;
13         if (s.score < lowestScore)
14           lowestScore = s.score;
15       }
16     }
17   }
18 }
```

図 1　部分的なメソッド再利用が難しいメソッド
Fig. 1　Difficulty in reusing a method code in partial.

生全体への適用処理だけを再利用して判定条件を変更したりするといった細粒度のコード再利用が難しい．そこで，select メソッド内で判定処理を担う for ブロックを filter メソッドとして別に定義すれば，同じ判定処理を社会人学生や夜間学生に対して再利用し，コードの重複コピーを防ぐことができる（図 2）．さらに，図 3 のように，filter メソッドをサブクラスで上書きすれば，判定条件を偏差値や出席回数に切り替えつつ，select メソッドに実装された学生全体への適用処理はサブクラスから再利用できるようになる．このように，スーパクラスが全体のアルゴリズムをテンプレート化し，サブクラスでは具体的な処理を上書き実装するような設計は，オブジェクト指向開発においてコードの再利用性を高めるための Template Method パターンとしてよく知られている．

しかし，Java をはじめとする多くのオブジェクト指向言語ではメソッドより細かい粒度でのコードの再利用をサポートしていないため，メソッドの部分的な再利用は困難である．たとえば，多くの IDE（Integrated Development Environment）は選択したコードを別のメソッドとして切り出して再定義するためのリファクタリング機能を備えている（図 4）．切り出されたメソッドでは，元のメソッドのローカル変数へアクセスできないため，それらは引数とし

```
class StudentSelector {
  void select() {
    List<Student> students = …
    List<Student> passed = ,,,
    double highestScore = 0.0;
    double lowestScore = 100.0;
    for (Student s: students) {
      double[] highAndLow = filter(
      s, passed, highestScore, lowestScore);
      highestScore = highAndLow[0];
      lowestScore = highAndLow[1];
    }
  }
  double[] filter(
  Student student, List<Student> passed,
  double high, double low) {
    if (s.gradYear == 2011) {
      if (s.score > 70)
        passed.add(s);
      if (s.score > high)
        high = s.score;
      if (s.score < low)
        low = s.score;
    }
    return new double[]{high, low};
  }
}
```

図 2　判定処理を切り出したメソッド
Fig. 2　Separating code for student filtering as a new method.

```
class SubSelector extends StudentSelector {
  double[] filter(
  Student student, List<Student> passed,
  double high, double low) {
    if (s.attendance > 0.9) {
      if (s.score > 60)
        passed.add(s);
      if (s.score > high)
        high = s.score;
      if (s.score < low)
        low = s.score;
    }
    return new double[]{high, low};
  }
}
```

図 3　サブクラスからの判定処理の上書き
Fig. 3　Overriding the filter method.

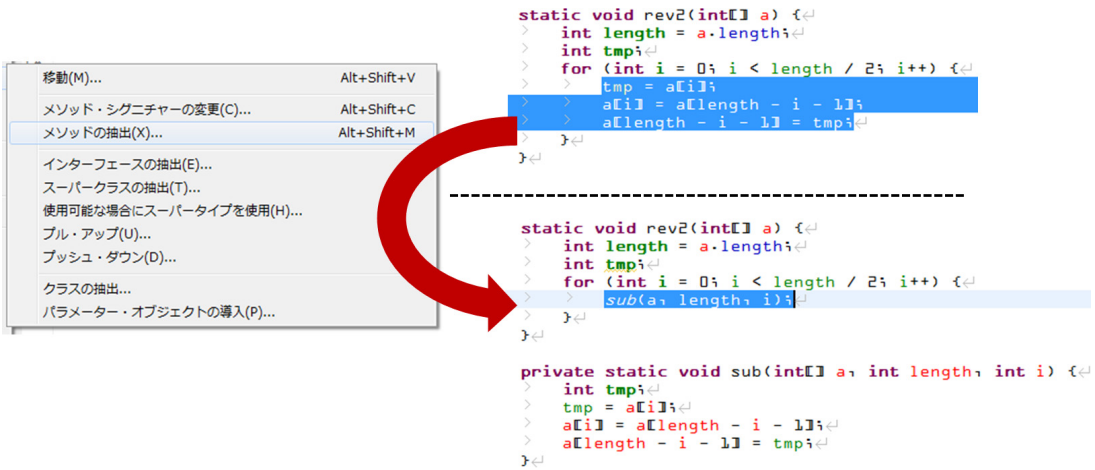

図 4　Eclipse IDE のメソッド抽出機能
Fig. 4　Method extraction in Eclipse IDE.

て渡される．たとえば，図 1 のメソッド select の 8–15 行をメソッド filter として切り出す場合，図 2 のように select のローカル変数である students，passed，high，low を引数として受け渡す必要が出てくる．したがって，ローカル変数が多ければメソッドの引数も大量になり，コードの視認性が落ち，切り出したメソッドの定義も煩雑になる．

切り出されたメソッド中のコードにローカル変数への値代入，いわゆる副作用がある場合，その影響を効率良く復元することも難しい．副作用を受けたローカル変数の値は，返り値や共有変数を介して元のメソッドへ受け渡さなければならない．また，元のメソッドでは変更された値を元のローカル変数へ復元する必要も出てくる．しかし，メソッドは複数の返り値を持てないため，副作用を受ける複数のローカル変数をまとめて受け戻すことはできない．C++ では，ポインタを用いた参照渡しにより副作用の再現を行えるが，呼び出し側から値渡しと区別がつかず，また Java には参照渡しが存在しない．特別なラッパクラスを用意したり，配列やリストに格納して戻したりしたとしても，それらを介した間接的なアクセスがパフォーマンス低下を引き起こすかもしれない．さらに，切り出されたメソッドの呼び出し側でも，値の復元処理やローカル変数の更新をする手間が発生する．たとえば，図 2 では，複数の値を単一オブジェクトに格納する配列を作成し，ローカル変数 high と low の影響を返り値として受け渡している．select メソッドでは，filter メソッド中で副作用を受けた high と low の値を復元するためのコードが現れる．

3. 上書き可能なメソッド内メソッド：内部メソッド

前章で述べた問題を解決するため，我々は**上書き可能なメソッド内メソッド**である内部メソッドを提案する．内部メソッドは，メソッドの中に定義できる局所的なメソッドであり，外側の内包メソッドで宣言されたローカル変数へアクセスできる．メソッド内の重複コードを内部メソッドとして括りだせば，同一メソッド内で細粒度にコードを再利用できる．また，サブクラスから内部メソッドを上書きすることで，内包メソッド側のアルゴリズムを再利用しながら内部メソッドの振舞いを変えられるようになる．

また，内部メソッドの有効性を評価するため，上書き可能なメソッド内メソッドを Java 言語向けの新たな言語機構として設計しコード変換器を実装した．拡張した Java 言語には，内部メソッドの仕様や制約に加え，ローカル変数に対して付加できる public 修飾子を導入している．これらの仕様および実装は，メソッド内コードの内部メソッド化がもたらす，アクセススコープの干渉や性能劣化を回避するように設計した．

```
1  class StudentSelector {
2    void select() {
3      List<Student> students = …
4      public List<Student> passed = ,,,
5      public double highestScore = 0.0;
6      public double lowestScore = 100.0;
7      for (public Student s: students) {
8        void filter() {
9          if (s.gradYear == 2011) {
10           if (s.score > 70)
11             passed.add(s);
12           if (s.score > highestScore)
13             highestScore = s.score;
14           if (s.score < lowestScore)
15             lowestScore = s.score
16         }
17       }
18       filter();
19     }
20   }
21 }
```

図 5　内部メソッドの宣言
Fig. 5　Inner method declaration.

3.1 内部メソッドを利用した細粒度のメソッド再利用

メソッド内に宣言可能で，内包メソッドのアクセススコープを継承するメソッドを内部メソッドと定義する．**図 5** のコード例のように，内部メソッド filter は，内包メソッド select のスコープに含まれるため，ローカル変数の参照や代入を透過的に行え，引数や返り値を使って副作用を再現する必要がない．

内部メソッドのアクセススコープは，ローカル変数にならって設計した．ただし，ローカル変数とは違い，内部メソッドとその内包メソッド間でのスコープ干渉を防ぐ必要がある．また，それと同時に，サブクラスによる変数アクセスや上書きを考慮する必要があるため，定義および呼び出しに次のような制限を設けた．

内部メソッド定義の制限

(1) **通常のメソッドおよびコンストラクタ内でのみ定義可能**：仕様の単純化のため，サブクラスでの拡張が許されているメソッドやコンストラクタに制限し，native メソッドや静的メソッド，静的初期化子，初期化子での定義を現時点ではサポートしない．

(2) **同一メソッド内で同名内部メソッドは 1 つ**：同じメソッド内では，別々の制御文の独立したスコープであったとしても，同名内部メソッドを定義させない．これにより，サブクラスからは内部メソッドを一意に特定し上書きできるようになる．

```
1  class SubSelector extends StudentSelector{
2    void select().filter() {
3      if (s.attendance > 0.9) {
4        if (s.score > 60)
5          passed.add(s);
6        if (s.score > highestScore)
7          highestScore = s.score;
8        if (s.score < lowestScore)
9          lowestScore = s.score;
10     }
11   }
12 }
```

図 6　内部メソッドの上書き
Fig. 6　Inner method overriding.

内部メソッド呼び出し可能な範囲

(1) **内部メソッド定義と同一スコープかつ定義以降**：通常のメソッド定義とは違い，ローカル変数や匿名クラスメソッドと同様に内部メソッドはその定義以降でのみ呼び出しが許される．これにより，内部メソッドからアクセス可能なローカル変数を一意に特定できるようになる．もし定義前の呼び出しを許してしまうと，内部メソッドから内包メソッドの中で未定義のローカル変数へもアクセス可能となってしまう．一方，上位または別のスコープでの呼び出しを許すことは，内部メソッドのスコープを別のスコープでも可視化させることにほかならない．そうすると，呼び出しスコープに同名のローカル変数があった場合，変数名の衝突を回避することは簡単ではない．

(2) **上書きする内部メソッド内**：上記以外では，内部メソッドを上書きしているサブクラスのメソッドから super を目的オブジェクトとして呼び出すことができる．

3.2　内部メソッドの上書き

上書きについては内部メソッドでも通常のメソッドと同様に，サブクラスによる上書き再定義が可能である．**図 6** のように内部メソッド filter の上書きは，内包メソッドと内部メソッドの名前をドットで区切って指定する．上書きした内部メソッドへは，スーパクラスでの内包メソッドのスコープが継承される．そのため，サブクラス SubSelectorの内部メソッド filter からも passed や highestScore，lowestScore へのアクセスが可能となる．

次の条件を満たす内部メソッドが，サブクラスから上書きできるものとする．

上書き可能な内部メソッド

(1) **継承可能なクラスのメソッド内に存在するもの**：final や static クラスを除く，abstract や private，インナークラスを含む通常クラス内のメソッド内での内部メソッドは上書き可能である．一方，明示的に extends を付加しない匿名クラスについても同様に可能となる．

(2) **上書きされていないメソッド内に存在するもの**：内包メソッドがサブクラスから上書きされてない場合に，その内部メソッドを上書き可能とする．たとえば，図 6 で，SubSelector で select() が上書きされないならば，内部メソッド filter() の上書きが許される．同様に，入れ子状に内部メソッドが定義されるときにも，1 度上書きされたメソッド内の内部メソッドの呼び出しや，そのサブクラスでの上書きはサポートしない．

内部メソッドの上書きは内包メソッド（その内部メソッドを宣言しているメソッド）の再利用を促進させる，インナークラスやローカルクラスのメソッドにはない特徴である．Java では，一部のコードをクラス内やメソッド内の局所的なメソッドとして定義するために，インナークラスやローカルクラスを利用できる．そのように定義されたメソッドを呼び出すことでも，細粒度のコード再利用をある程度実現できる．しかし，インナークラスやローカルクラスのメソッドは，内包メソッドのクラスのサブクラスから上書きできない．そのため，内包メソッドのアルゴリズムを再利用するような上書き拡張は許されていない．また，インナークラスはメソッド外に定義されるので，メソッド内のローカルな変数を直接アクセスできない．メソッド内に定義されるローカルクラス中のメソッドも，内包メソッドで final 宣言されたローカル変数しかアクセスできない．

サブクラスで上書きされた内部メソッドからのローカル変数アクセスを制限するために public 修飾子を導入する．public 修飾子はローカル変数に対して付与され，public でない変数は内包メソッド内のローカルな変数となり，サブクラスからアクセスできなくなる．これにより，ローカル変数が意図せずサブクラスへ公開され，誤ってアクセスされる危険を防止できるようになる．一方，スーパクラスのメソッドを修正するときに，サブクラスで上書きしている内部メソッドが参照している変数を知らずに改名，削除してしまう危険も軽減される．

4.　実行性能を考慮した実装

素朴な内部メソッドの実装は，実行性能を著しく落とす可能性がある．たとえば，内部メソッドを通常のメソッドへ変換するような実装では，前述のように引数やラッパオブジェクトを用いてローカル変数を受け渡し，同様のアクセススコープを実現する必要がある．内部メソッド側を呼び出すごとにラッパオブジェクトを生成し，それを介して呼び出し後に副作用のあるローカル変数の値を復元しな

ければならない．また，コードブロックを内部メソッド化することで，そのメソッド呼び出しコストも追加的に発生してしまう．一般に，Java 言語のメソッド呼び出しには，Java 仮想マシン内部で新しいスタックフレームが用意されるだけでなく，メソッドを探索するディスパッチ処理が加わるため性能低下を避けられない．

我々はオーバヘッドを低減させるために，内部メソッド呼び出しの最適化を施したコンパイラを実装した．内部メソッドを導入した拡張 Java 言語は，通常の Java 言語へのソースコード変換により実現する．コード変換には，拡張コンパイラ生成フレームワークである Jastadd を利用した．コンパイラは内部メソッドを通常のメソッドへ置き換える際に，1. 選択的インライン実行，2. ローカル変数オブジェクトの最適化を実行し，メソッドの呼び出しコストを低減させる．以下の節では，それぞれの最適化について説明する．

4.1　選択的インライン実行

実装したコンパイラは，内部メソッドの呼び出しに対して可能な限りインライン展開による最適化を行う．内部メソッドはメソッド内の局所的なメソッドであり，その直接的な呼び出し範囲を内包メソッド（内部メソッドを宣言しているメソッド）に制限している．サブクラスの内部メソッド上書きでの super を介した呼び出しを除けば，呼び出しが内包メソッド以外へ拡散することがない．したがって，サブクラスから上書きされない限り，内包メソッド中で内部メソッド呼び出しのインライン展開が可能となる．インライン展開されたコードでは，追加のメソッド呼び出しや，ローカル変数を受け渡すためのオブジェクト作成が不要になり，実行オーバヘッドを軽減できる．

内部メソッドの上書きがある場合は，内包メソッド側で適切な内部メソッドを呼び出すためのコードを生成する．コンパイラは**図 7** に示すようなコードを生成する．まず，StudentSelector の select メソッドに対し，内部メソッド filter の実装コードをインライン展開する．また，サブクラスによる内部メソッド上書きおよび脱最適化に備え，内部メソッド実装 filter に加えて，その呼び出しをインライン展開しない $select メソッドを生成する．サブクラス SubSelector 側では select メソッドを上書きし，$select を呼び出すことでインライン展開がキャンセルされる．3.2 節で述べたように，内部メソッドが上書き可能であるとき，つねにその内包メソッドは上書きされていないため，コンパイラの生成する SubSelector の select メソッドがユーザ実装と競合することはない．さらにコンパイラは SubSelector には，$select メソッドから呼び出される内部メソッド実装 filter の上書きも通常のメソッドとして生成する．ローカル変数へのアクセスは，後述する最適化を行い実行オーバヘッドを軽減する．

```
1  class StudentSelector {
2    void select(...) {
3      /* インライン展開された
4         内部メソッド呼び出し */
5    }
6    void $select(...) {
7      /* インライン展開されない
8         内部メソッド呼び出し */
9    }
10   void filter(LocalVars lvobj,...) {
11     /* 内部メソッド実装 */
12   }
13 }
14
15 class SubSelector extends StudentSelector{
16   void select(...) {
17     /* 内包メソッドを上書きし
18        インライン展開をキャンセル */
19     $select(...);
20   }
21   void filter(LocalVars lvobj,...) {
22     /* 内部メソッドの上書き実装 */
23   }
24 }
```

図 7　選択的インライン実行
Fig. 7　Inlining of inner method calls.

4.2　ローカル変数オブジェクトの最適化

上書きされた内部メソッドのアクセススコープは，ローカル変数オブジェクトにより実現する．ローカル変数オブジェクトには，内包メソッドと内部メソッドの間で共有してアクセスされるローカル変数を格納する．コンパイラはあらかじめ生成したローカル変数オブジェクトが，内部メソッドの引数と返り値を介して受け渡されるように変換を施す．また，内包メソッドと内部メソッドの双方からのローカル変数へのアクセスも，このオブジェクト内へのアクセスに変換される．そのため，副作用の影響を復元するためのコードを実装したり，ラッパオブジェクトを生成したりする必要もなくなる．ローカル変数オブジェクトを介したヒープアクセスがもたらすオーバヘッドについては，Java 仮想マシンの JIT コンパイラの最適化により，軽微に抑えられることが期待される．

さらに，コンパイラはコード内の変数アクセスを解析し，ローカル変数オブジェクトへのアクセスを最適化する．最適化されたコードの動作例を**図 8** の疑似コードで示す．コンパイラはコード解析により，内部メソッド内のデータフローを調べ，実際に副作用が生じる代入文やオブジェクトアクセスを収集する．その結果，内部メソッド内で使用されない変数はローカル変数オブジェクトへ含めない（3–12 行目）．一方で副作用をともなわない変数については，ロー

```
1  void $select() {
2
3  foreach VAR (VARS)
4    if VAR has write-access in filter() then
5      if LVOBJ is null then
6        create LVOBJ for filter()
7      end
8      push VAR into LVOBJ for filter()
9    elif VAR has read-access in filter() then
10     push VAR into a stackframe for filter()
11   end
12 end
13
14 if LVOBJ is not null then
15   push LVOBJ into a stackframe for filter()
16 end
17
18 filter(LVOBJ,...);
19 }
```

図 8　実行時のローカル変数オブジェクト最適化実装の疑似コード．select() メソッド中の全ローカル変数 VARS のうち，副作用のある変数をローカル変数オブジェクト LVOBJ へ，読み取りアクセスのみならメソッド引数へ渡される（疑似コード内ではスタックへのプッシュ）

Fig. 8　Pseudo code of runtime behavior in optimzied local variable accesses. All local variables (VARS) are checked whether pushed into a local variable object (LVOBJ) or a stack frame created for method call arguments.

カル変数オブジェクトを介さず，メソッド引数を介して渡されるよう変換する（9–11 行目）．疑似コードでは，これを内部メソッド引数でなく，便宜的にメソッドのスタックフレーム（引数渡しのための JVM 内スタック領域）へのプッシュ操作で記述している．実際には，コンパイラは静的解析により実行前に引数の型を決定するため，これらの変数アクセス解析やプッシュ操作によって追加のオーバヘッドが生じるわけではない．

さらに，実行時のオブジェクト生成のオーバヘッドを低減させるため，可能な限りローカル変数オブジェクト生成を抑えられるようコードを変換する．たとえば，内部メソッド内に副作用を持つローカル変数がなければ，ローカル変数オブジェクトの作成をキャンセルする．もし副作用がある場合でも，内包メソッドが再帰的に呼び出されない限りは，ローカル変数オブジェクトをシングルトンオブジェクトとして扱い，無駄なオブジェクト生成を抑える．

5. 実験

内部メソッドの実行オーバヘッドを計測するためにマイクロベンチマークを用いた実験を行った．実験環境は，Windows7 Intel®Core™ i5 CPU 2.67 GHz，メモリ 4.00 GB である．Java VM は Oracle Java HotSpot Client VM 1.6.0_37 を使いデフォルトオプションで実行した．実験では，内部メソッド化による呼び出しオーバヘッドを計測するため，副作用のないコード領域を内部メソッドとして切り出したメソッド（a1），副作用のある領域を切り出したメソッド（a2），それぞれについて内部メソッドが上書きされた場合（a3，a4）について，それぞれ 1,000,000,000（1 G）回呼び出したときの実行時間を計測した．比較対象としては，a1–a4 に対応する領域を通常の Java のメソッドとして切り出した b1–b4 と，メソッドとして切り出さない c1–c4 とを用いた．実験に用いたコンパイラは，変数アクセスを解析してのローカル変数オブジェクトへのアクセスの最適化を実装していないので，生成される Java のコードを直接記述して実験に用いた．

表 1　実行時間（ミリ秒）
Table 1　Elapsed time (ms.).

コードの切り出し方	1	2	3	4
a. 内部メソッド	614	614	3,706	7,373
b. 通常のメソッド	614	7,683	3,709	11,417
c. なし	614	614	614	614

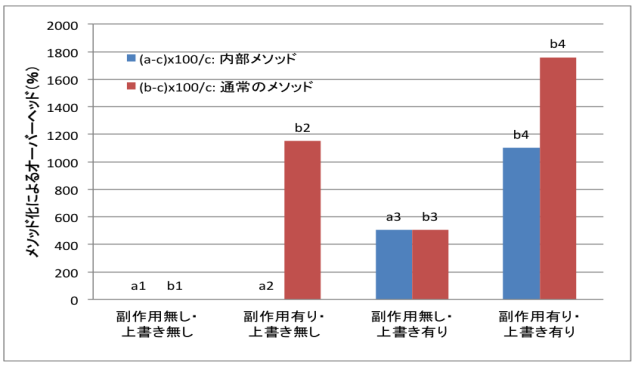

図 9　メソッド切り出しによるオーバヘッド
Fig. 9　Execution overhead due to method extractions.

実験結果として，10 回実行した平均の実行時間を**表 1** に示す．**図 9** には，c1–c4 に対するメソッド切り出しのオーバヘッドの割合を示す．a1，b1 では c1 に対するオーバヘッドは 1%未満になり，a3，b3 はともに c3 の 6 倍以上の時間がかかっている．a2，b2，c2 の比較では，a と c が同じ結果であったのに対して，b は他の 12 倍以上の値が出た．a3，b3，c3 の比較では，a，b がそれぞれ c の約 12 倍，約 18 倍の実行時間を要した．

図 9 から，オブジェクト作成を最適化した a2 と a4 が，b2 および b4 に対して大幅にオーバヘッドを削減していることが分かる．内部メソッドを用いた a のコードの場合，a2 ではインライン展開によってローカル変数受け渡しのためのオブジェクトを作成せず，a4 でも 1 度作成したローカル変数用のオブジェクトを繰り返し利用している．一方，通常の Java のメソッドで素朴に実装した b のコードでは，切り出したメソッドの呼び出しごとにローカル変数を受け

渡すためのオブジェクトを作成している．a1 と b1，a3 と b3 では，副作用を考慮する必要がないため，オブジェクト作成が行われず同程度のオーバヘッドであった．a3 の実行時間が c3 と比べて大きくなっているため，ローカル変数オブジェクトを介したヒープアクセスのオーバヘッドは最適化によって軽微に抑えられてはいない．

6. 関連研究

GCC で拡張された入れ子関数 [7] や Scala [6] でもメソッド内にローカルなメソッドを定義できる．そのような入れ子メソッドはその外側で宣言された変数を参照できるが，サブクラスで上書きすることはできない．オブジェクト指向言語 Beta [1] では入れ子メソッドを，サブクラスで上書き可能であるが，スーパクラスの振舞いを置き換えるような上書きはできない．上書きはスーパクラスのメソッドへの処理を追加することのみ可能である．1 度スーパクラスのメソッド内に現れた振舞いが，サブクラスにおいて上書きされた後にも必ず現れるため，上書きによって完全にメソッドを置き換えることが不可能である．

Closure Joinpoints [2]，regioncut [3] はアスペクト指向プログラミングにおけるジョインポイントを指定するための技術である．この技術を用いればコードのある領域を明示的に指定し，その領域をメソッドとして切り分けることなくジョインポイントとすることができる．指定されたジョインポイントに対してアドバイスを織り込めば，ブロック単位のコードの変更を実現することができるが，アドバイスのコードから元のメソッドのローカル変数へ代入を行うことができない．Closure Joinpoints，regioncut は領域を指定するための機構であるため，指定した領域の内部再利用はできない．

クロージャを用いると，メソッドの一部をブロック単位で分けることが可能である．分けられたクロージャからはその外側のローカル変数を参照できる．クロージャが代入されたフィールドに，サブクラスで別なクロージャを代入するなどして，サブクラスによる上書き相当のことを実現できる．しかし，サブクラスを上書きしたクロージャは，上書き前のクロージャが可能だった外側で宣言されたローカル変数への参照ができない．method slot [4] を用いると，メソッド呼び出しに対してクロージャを上書きすることができるが，同様にローカル変数の参照ができない．

7. まとめ

本論文では細粒度でのコード再利用を可能とする内部メソッドを提案した．内部メソッドはメソッド内に定義できるメソッドであり，単なるクロージャとは違いサブクラスで上書き拡張ができる．内部メソッドからは，内包メソッドで宣言されたメソッドにローカルな変数へもアクセスできる．これらの内部メソッドの特徴によって，メソッド内のコードブロックを内部メソッドとして再利用したり，それを上書きして内部メソッド化したコードブロック以外のコードを再利用したりすることが可能になる．

内部メソッドはコンパイル時のコード変換によって実現した．内部メソッド化によるオーバヘッドを低減させるため，コード変換時には選択的インライン展開，変数アクセスの最適化を行う．それによって，通常のメソッドとしてコードを切り出すような素朴な実装と比較して，実行速度のオーバヘッドが削減されることを示した．

今後は，内部メソッドの有効性の定量的な評価が課題である．我々は，予備的に複数のオープンソースプログラムを対象にした実験で，メソッドから切り出すコード領域の行数を仮定した際のローカル変数宣言とその参照の傾向を調査している．その結果，ブロック等の制御構造や，実装コードの意味を考慮せず，機械的に 20 行のウィンドウで全行走査した場合には，ウィンドウ内外をまたがりアクセスされるローカル変数がその総数の 60〜90%に及ぶコード領域が存在した．このような領域を同一スコープを保持できる内部メソッドで切り出せば，大量のローカル変数を引数で渡したり，副作用を再現するといった手間を軽減できる可能性がある．ただし，走査領域には，制御文をまたがるため切り出しが不可能なコードや，再利用の観点で切り出しが有益でないコードも含まれる．そのため，予備実験から得られたローカル変数アクセスの傾向は，必ずしも内部メソッドの適用可能性および有効性を評価するに足るものではない．さらに実験手法を洗練させ，メソッド切り出し可能かつ有効と見込まれる妥当なコード領域のサンプリング手法の開発を進めていきたい．

参考文献

[1] Knudsen, J.L., Lofgren, M., Lehrmann-Madsen, O. and Magnusson, B.: *Object-Oriented Environments — The Mjolner Approach*, Prentice Hall (1994).
[2] Bodden, E.: Closure joinpoints: Block Joinpoints without Surprises, *AOSD* (2011).
[3] Akai, S. and Chiba, S.: *A Designator for Selecting a Code Region in Aspect-oriented Programming Language*, Information Processing Society of Japan (2011).
[4] Zhuang, Y.Y. and Chiba, S.: Supporting Methods and Events by An Integrated Abstraction, *AOSD* (2013).
[5] Team, T.J.: Jastadd, available from ⟨http://jastadd.org⟩.
[6] available from ⟨http://www.scala-lang.org⟩
[7] Breuel, T.M.: Lexical Closures for C++, *Usenix Proceedings, C++ Conference* (1988).

平松 俊樹

1988 年生．2011 年東京工業大学理学部情報科学科卒業．2013 年同大学大学院情報理工学研究科数理・計算科学専攻修士課程修了．プログラミング言語の研究に従事．

佐藤 芳樹　（正会員）

1977 年生．2000 年東北大学工学部情報工学科卒業．2002 年同大学大学院情報科学研究科情報基礎科学専攻修士課程修了．2005 年東京工業大学大学院情報理工学研究科数理・計算科学専攻博士（理学）取得．（株）三菱総合研究所，（株）日立製作所，日本オラクル株式会社を経て，現在，東京大学大学院情報理工学系研究科特任助教．言語処理系，HPC の研究に従事．

千葉 滋　（正会員）

1991 年東京大学理学部情報科学科卒業．1996 年同大学大学院理学系研究科情報科学専攻博士課程退学．東京大学助手，筑波大学講師，東京工業大学大学院情報理工学研究科教授を経て，2012 年より東京大学大学院情報理工学系研究科教授．博士（理学）．プログラミング言語，システムソフトウェアの研究に従事．