平成23年度 学士論文

細かい粒度でコードの再利用を可能とするメソッド内メソッドのJava言語への導入

東京工業大学 理学部 情報科学科

学籍番号 07-2233-1

平松 俊樹

指導教員

千葉 滋 教授

平成23年2月7日

概要

本稿では、メソッド内にメソッドを定義でき、さらにローカル変数のスコープの制限を緩める言語機構を Java 言語に導入することを提案する。この機構を導入することで、より細かい粒度でコードの再利用が可能になると考える。

プログラミングにおいて、コードの再利用は非常に有益である。コードを再利用することによって、生産性の向上、コピー&ペーストによる同一コードの散逸の防止などが期待できる。しかし、既存の Java 言語においては、コードの再利用の最小単位はメソッドであり、ブロック単位でのコードの再利用は不可能である。より細かい粒度でのコードの再利用を行うために、後に変更されるブロックを別のメソッドとして定義することは可能であるが、その場合には、新たに分けられたメソッドが必要なローカル変数にアクセスするために、該当するローカル変数をメソッドの引数に渡す、等の方法をとらなくてはならなくなる。さらに、新たに分けられたメソッドから元のメソッドのローカル変数に代入を行う際には、分けられたメソッドの返り値を元に、元のメソッド側で代入を行わなければならなくなるなど、コードが煩雑になってしまう。

本研究では、自身が定義されたメソッドのローカル変数にアクセスできるメソッド内メソッドを Java 言語に導入することを提案する。本システムでは、メソッド内にメソッドの定義を記述することが可能である。そのクラス内においては、メソッド内メソッドはその外側のメソッドで宣言された全てのローカル変数にアクセス可能である。また、そのクラスのサブクラスにおいて、メソッド内メソッドのみをオーバーライドすることが可能である。その際にメソッド内メソッドからアクセスできるローカル変数はスーパークラスのメソッド定義時に指定したもののみである。そして、サブクラスのメソッド内メソッドからアクセス可能であることを指定するために、本システムでは引数、ローカル変数に public 修飾子を指定することが可能である。サブクラスのメソッド内メソッドにおいてアクセス可能なローカル変数は、この public が指定されたもののみである。このため、サブクラスからスーパークラスのローカル変数に無制限にアクセスすることは不可能であり、カプセル化は破壊されていないと考える。

本研究の実装は、コンパイラ実装フレームワークである JastAdd を用い

て実装された Java コンパイラである JastAddJ を拡張することによって行った。コンパイル時に、メソッドボディ内にメソッド定義を持つメソッドに対して、対応するクラスを生成し、メソッド内のローカル変数をそのクラスのフィールドとすることで、本システムにおける、メソッド内メソッドからの外部のローカル変数への参照を実現した。また、性能を調べるために実験を行い、ローカル変数への代入が頻繁に行われるプログラムに対しては、ほぼオーバーヘッドなく本システムが実現できていることを確かめた。

謝辞

本研究を進めるにあたり、研究の方針や進め方についての数々の助言をして下さった指導教員の千葉滋先生に深く感謝いたします。

また、赤井駿平氏には論文の書き方、研究の進め方についての多くの助言を頂き、大久保貴司氏には実装上の指導、助言をして頂きました。心から感謝いたします。

最後に、ともに研究を行い、多くの助言をして頂いた千葉研究室の皆様方に感謝いたします。

目 次

図 目 次

表 目 次

第1章　はじめに

プログラミングにおいて、コードの再利用はとても有益である。コードを再利用することで、生産性の向上、同一コードのコード中における散逸の防止などが期待できる。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などを再利用することが可能である。

しかし、Java 言語においてはコードの再利用の最小単位はメソッドであり、それより小さな単位、つまりブロック単位でのコードの再利用ができない。このため、メソッドの一部分を再利用し、一部分を変更する、といったことを既存の Java 言語において実現することは困難である。

このようなメソッドの一部の変更を行うために、メソッドの一部を別のメソッドとして定義しなおし、それをサブクラスにおいてオーバーライドすることでメソッドの一部を上書きするという方法が取られうる。しかし、あるメソッドが別のメソッド内で宣言されたローカル変数にアクセスすることは不可能である。そのため、メソッドの一部を別のメソッドとして定義しなおした場合には、そして、そのメソッド内で、元のメソッドで宣言されたローカル変数が必要とされる場合には、必要なローカル変数を引数として定義しなおされたメソッドに渡さなければならない。さらに、定義しなおされたメソッドの内部から、元のメソッドのローカル変数への代入がある場合には、定義しなおされたメソッドの返り値を用いて、元のメソッドにおいてローカル変数への代入を行うなどの作業が必要となる。以上の様な作業が必要となるため、メソッド内の一部を別のメソッドとした場合にはコードが煩雑になる恐れがある。結果として、プログラムの一つの変更に対して必要なコード上の修正点が多くなり、プログラムの変更・修正に対して弱くなる。

そこで、本研究では、Java 言語にメソッド内に定義可能なメソッドを導入するシステムを提案する。本システムにおいては、メソッド内に定義されたメソッドは、自身の定義をその内部に含むメソッドのローカル変数にアクセスすることができる。そのため、本システムを用いた場合には、既存の Java 言語においてメソッドの一部を別のメソッドとした場合の様な、引数としてのローカル変数の引き渡し、返り値を元にしたローカル変数への代入といった作業を行う必要がなくなる。

本稿の残りは、次のような構成からなっている。第 2 章では既存の技術とその問題点について、第 3 章では本システムの仕様を、第 4 章では本システムの実装について、第 5 章では性能を調べるための実験について、第 6 章ではまとめと今後の課題について述べる。

第2章　既存技術とその問題点

本章では、既存のプログラミング言語におけるコードの再利用に関する問題点と、本研究の関連研究について述べる。

2.1　既存のプログラミング言語とその問題点

2.1.1　Java

Java 言語におけるコードの再利用の最小単位はメソッドである。クラス間の継承関係を利用することで、再利用したいメソッドが、あるクラスだけでなく、そのサブクラスにおいても再利用できる。この例を図 2.1 に示す。この図では、クラス C のメソッド method1() が、C のサブクラスである Child においても引き継がれる。

しかし、Java 言語においてはメソッドより細かい粒度でのコードの再利用、つまりブロック単位でのコードの再利用ができない。このため、メソッドの一部を再利用し、一部を変更したい場合には、変更したい部分を新たに別のメソッドとして切り離し、定義しなおすという作業が必要となる。

メソッド内の一部を別のメソッドとして定義しなおすことで、メソッドボディの部分的な再利用、変更は可能となるが、別のメソッドを用いることで新たな問題が発生する。Java 言語においては、メソッド内のローカル変数は他のメソッドからアクセスすることができない。このため、単純にメソッド内の一部を別のメソッドとして切り分けた場合には、切り分けられたメソッドから元のメソッドのローカル変数を参照できず、また、ローカル変数への代入も不可能である。この例を図 2.2 に示す。

この例では、図 2.1 におけるクラス C のメソッド method1 の for 文の内部を、別のメソッド method2 として切り分けている。method2 からは method1 内のローカル変数 i, j を参照することができず、また method2 から method1 のローカル変数 localVar1, localVar2 への代入も不可能である。

この問題を解消するためには、必要なローカル変数を新たに切り分けられたメソッドの引数として渡す、等の手段を取らなくてはならない。また、もとのメソッドのローカル変数への代入を再現するためには、切り分けら

```
1  //スーパークラス
2  public class C {
3    public void method1() {
4      int localVar1 = 0;
5      int localVar2 = 0;
6                  int localVar3 = 0;
7                  String localVar4 = "";
8      for (int i = 0; i < 100; i++) {
9        for (int j = 0; j < 100; j++) {
10         localVar1 = i + j;
11         localVar2 = i * j;
12       }
13     }
14   }
15 }
16 //サブクラス
17 public class Child extends C{
18   public static void main(String[] args) {
19     new Child().method1();
20   }
21 }
```

図 2.1: 継承によるメソッドの再利用

れたメソッドの返り値を元に、元のメソッド側でローカル変数への代入を行わなければならない。この際、メソッドの返り値は一つであるため、複数の値をローカル変数へ代入する場合には、切り分けられたメソッド内において、必要な値を一つのオブジェクトにまとめるなどの手段をとる必要がある。これらの変更を図 2.2 のコードに加えたものを、図 2.3 に示す。

これらの変更を行い、サブクラスにおいて切り分けたメソッドをオーバーライドすることにより、メソッドの一部を再利用、一部に変更を加えることができる。しかし、この状態ではサブクラスのメソッドにおいて、引数として渡されないローカル変数（図 2.3 における localVar3, localVar4）を参照する必要がある場合、または元のメソッドの新たなローカル変数への代入を行う場合に、サブクラスにおけるコードの変更だけでは対処できない。このため、これらの新たな要求が発生した場合には、そのスーパークラスに対する変更が必要である。また、スーパークラスの変更に付随して、そのスーパークラスを継承する全てのクラスに対して同種の変更を行わなければならないかもしれない。さらに、各クラスに対する変更も、

- 切り分けたメソッドの引数の型、個数の変更
- 返り値として用いるクラスの変更

```
1  public class C {
2    public void method1() {
3      int localVar1 = 0;
4      int localVar2 = 0;
5                  int localVar3 = 0;
6                  String localVar4 = "";
7      for (int i = 0; i < 100; i++) {
8        for (int j = 0; j < 100; j++) {
9          method2();
10       }
11     }
12   }
13   public void method2() {
14     localVar1 = i + j;// i, j が参照できず、
15     localVar2 = i * j;// localVar1, 2への代入も不可能
16   }
17 }
```

図 2.2: メソッドの一部を別のメソッドとして切り分ける例

- 元のメソッド内でのローカル変数への代入作業の変更

等複数必要になる。

以上より、Java 言語において、対象のコードをメソッドとして、ブロック単位でのコードの再利用を行おうとすると、

- ローカル変数の引き渡し、代入のためにコードが複雑になる
- サブクラスにおける変更に対し、スーパークラスを含めた多くの部分に変更を加える必要が出てくる

のような問題点が現れる。

2.1.2　Beta

この説では、オブジェクト指向言語である Beta [2] について述べる。Beta においては、メソッドのネスト化が可能である。以下に Beta で記述された、ネスト化されたメソッドを示す。

```
V:< (# x: @Integer;
       VV:< (# do l1; inner; l2 #);
    #)
```

```
1  public class C {
2    public void method1() {
3      int localVar1 = 0;
4      int localVar2 = 0;
5      int localVar3 = 0;
6      String localVar4 = "";
7      for (int i = 0; i < 100; i++) {
8        for (int j = 0; j < 100; j++) {
9          ReturnValue returnValue = method2(i, j);
10         localVar1 = returnValue.v1;
11         localVar2 = returnValue.v2;
12       }
13     }
14   }
15   public ReturnValue method2(int i, int j) {
16     int localVar1 = i + j;
17     int localVar2 = i * j;
18     return new ReturnValue(localVar1, localVar2);
19   }
20   class ReturnValue {
21     int v1;
22     int v2;
23     public ReturnValue(int v1, int v2) {
24       this.v1 = v1;
25       this.v2 = v2;
26     }
27   }
28 }
```

図 2.3: 図 2.2 に変更を加えたもの

この例においては、メソッド V の内部にメソッド VV が定義されている。メソッド VV からは V に定義された Integer 型の変数 x が参照可能である。また、VV 内の `do l1; inner; l2` は、l1 と l2 を実行することを表しており、inner はメソッドがオーバーライドされる際の各文の実行順序を定めるためのものである。

このように、Beta ではメソッド内にメソッドを定義することは可能であるが、メソッドをオーバーライドする際に、スーパークラスにおける振る舞いが取り除かれない。以下にこの例を示す。先の例のメソッド VV をサブクラスにおいて、以下のように記述してオーバーライドする場合を考える。

```
VV::< (# do l3; inner; l4 #)
```

このとき、最終的なメソッド VV の実装は以下のようになる。

```
VV: (# do l1; l3; inner; l4; l2 #)
```

このように、一度スーパークラスのメソッド内に現れた振る舞い（この例では l1, l2）が、サブクラスにおいてオーバーライドされた後にも必ず現れるため、オーバーライド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。

2.2　関連研究

2.2.1　Closure Joinpoints

Closure Joinpoints [1] 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。この技術を用いることで、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。Closure Joinpoints で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むことで、ブロック単位のコードの変更を実現することができる。図 2.4 に、Closure Joinpoints を用いたコードを示す。

図 2.4 の例では、9 行目の exhibit から 12 行目にかけてのコードが、明示的に指定されたジョインポイントであり、この部分に対してアドバイスを織り込むことで、変更を加えることができる。20, 21 行目がジョインポイントに対するアドバイスである。

このように Closure Joinpoints では、アドバイスから参照できるのは引数として渡された値であり、アドバイスのコードからの元のメソッドのローカル変数への代入は不可能である点が本システムとの大きな違いである。また、これは領域を指定するための機構であるため、指定した領域を複数回別の場所で呼び出すなどのことはできない。

2.2.2　Regioncut

regioncut [4] は、Closure Joinpoints と同じく、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。regioncut は、コード領域を二つのポイントカット指定子の間の領域として指定することで、コード領域を選択可能にしている。例えば、

```
region[call(* *.a()), call(* *.b())]
```

のようにすることで、この regioncut は

```
1  //base code
2  class ShoppingSession {
3      int totalAmount = 0;
4      ShoppingCart sc = new ShoppingCart();
5
6      void buy(final Item item, int amount) {
7          Category category = Database.categoryOf(item);
8          totalAmount =
9          exhibit Buying(int amount, Category c) {
10             sc.add(item, amount);
11             return totalAmount + amount;
12         }(amount, category);
13     }
14 }
15
16 //advice
17 aspect BonusProgram {
18     joinpoint int Buying(int amount, Category cat);
19     int around Buying(int amt, Category cat) {
20         if (cat == Item.BOOK) amt += amt / 2;
21         return proceed(amt, cat);
22     }
23 }
```

図 2.4: Closure Joinpoints を用いたコード

```
a();
int x = 42;
y();
b();
```

のような、メソッド呼び出し a() で始まり、b() で終わるようなコード領域を指定する。

regioncut では、指定されたコード領域内の変数の値を取得し、アドバイスへの引数として渡すことで、アドバイス内でその値を使用できるる。しかし、元のメソッドのローカル変数への代入をアドバイス内から行うことは不可能である。

第3章　細かい粒度でコードの再利用を可能とするメソッド内メソッド

本研究では、ローカル変数のスコープの制約を緩め一部のローカル変数をメソッド内のメソッドからも参照可能とし、このメソッド内メソッドをオーバーライドすることで、既存の方法では不可能であった種類の、より細かい粒度でのコードの再利用が可能となる機構を Java を拡張することで実現した。

以下、本章では、本システムの仕様を示す。

3.1　メソッド内メソッドの定義

本システムにおいては、メソッドボディの内部にメソッドの定義を記述することが可能である。メソッド内メソッドの定義は以下の method2 ように宣言される。

```
M1 T1 method1(args1) {
    //method1 のメソッドボディ
    M2 T2 method2(args2) {
        //method2 のメソッドボディ
    }
    //method1 のメソッドボディ
}
```

ここで、M1, M2 はそれぞれ method1, method2 の修飾子、T1, T2 は返り値、args1, args2 は引数である。修飾子、返り値等については、通常の Java のメソッド定義時のそれらと本システムのそれらになんら何ら違いはない。一方で、引数およびメソッドボディ内のローカル変数に対しては、通常の Java とは異なり、後述のアクセス修飾子を宣言することが可能である。

本システムにおいては、

```
M1 T1 method1(args1) {
    M2 T2 method2(args2) {}
    M3 T3 method3(args3) {}
}
```

のように一つのメソッド定義の内部に二つ以上のメソッド内メソッドを定義することが可能である。また、

```
M1 T1 method1(args1) {
    M2 T2 method2(args2) {
        M4 T4 method4(args4) {}
    }
}
```

のようにメソッド内メソッドの内部にさらにメソッド内メソッドを定義することも可能である。

3.1.1　引数、ローカル変数のスコープの制限

本システムでは、メソッドの引数およびメソッドボディ内のローカル変数を、そのメソッド内に定義されたメソッド内メソッドの内部から参照することが可能である。また、三つ以上のメソッドが階層的に定義されている場合、つまりメソッドの内部にメソッドが定義され、その内側のメソッドがさらにその内部にメソッド定義を含んでいるような場合には、任意のメソッド内メソッドの内部からアクセス可能な引数およびローカル変数は、そのメソッド内メソッドの直接の外側のメソッドにおいて宣言されたものだけでなく、自分より外側で定義された全てのメソッドのローカル変数である。

本システムでは、メソッド内メソッドの定義を含むメソッドの引数、ローカル変数にアクセス修飾子 public を付けて宣言することができる。public が用いられた変数は、インナーメソッドがオーバーライドされる際に、オーバーライド後のインナーメソッドからも参照可能となる。この点については、メソッド内メソッドのオーバーライドの項で説明する。

本節で述べた引数、ローカル変数のスコープの仕組みを図 3.1、図 3.2 に示す。

3.2　メソッド内メソッドのオーバーライド

本システムでは、メソッド内に定義されたメソッドを、そのクラスのサブクラスにおいてオーバーライドすることが可能である。

```
1   public void m1(int arg1) {
2     int localVar1;
3     public void m2(int arg2) {
4       int localVar2;
5       public void m3() {
6
7       }
8     }
9   }
```

図 3.1: メソッド内メソッドにおける引数、ローカル変数のスコープ：コード例

本節ではメソッド内メソッドのオーバーライドの方法、およびオーバーライドが行われた場合のメソッドの動きについて述べる。

3.2.1　メソッド内メソッドをオーバーライドするメソッド

メソッド内メソッドをオーバーライドするメソッドの記述法について説明する。

クラス C のメソッド method1 内に定義されたメソッドである method2

```
class C {
    M1 T1 method1(args1) {
        M2 T2 method2(args2) {
            //C における method2 の実装
        }
    }
}
```

を、C のサブクラスである Child クラスでオーバーライドする場合を考える。このとき、Child クラスにおける method2 の定義は以下のように記述される。

```
class Child {
    M1 T1 method1(args1).M2 T2 method2(args2) {
        //Child における method2 の実装
    }
}
```

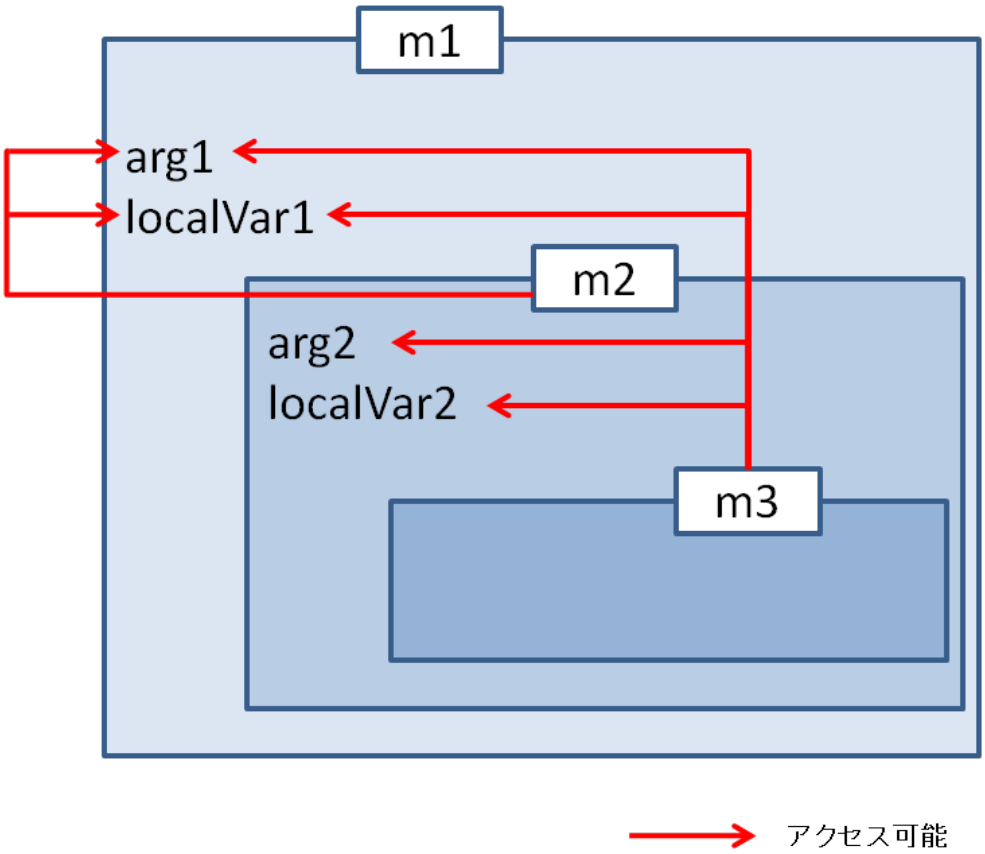

図 3.2: メソッド内メソッドにおける引数、ローカル変数のスコープ：イメージ

なお、コード内における M1, T1, args1, M2, T2, args2 はそれぞれメソッド method1, method2 の修飾子、返り値の型、引数である。この例のように、メソッド内メソッドをオーバーライドする際には、修飾子、返り値の型および引数の型と名前を含めた形でメソッド名をドットで区切って列挙し、オーバーライドするメソッド（この例では method2）の新たな実装のみを記述することになる。

図 3.3 にスーパークラスのメソッド内メソッドをオーバーライドするコードの例を示す。

```
1  //スーパークラス
2  public class C {
3    public void method1(int arg1) {
4      public void method2(int arg2) {
5        //オーバーライドされるメソッド
6        System.out.println("method2 in superclass");
7      }
8    }
9  }
10
11 //サブクラス
12 public class Child extends C {
13   public void
14         method1(int arg1).public void method2(int arg2) {
15     //オーバーライドするメソッド
16     System.out.println("method2 in subclass");
17   }
18 }
```

図 3.3: スーパークラスのメソッド内メソッドをオーバーライドするコードの例

3.2.2　外側のメソッドで宣言されたローカル変数への参照

メソッド内メソッドがオーバーライドされない場合、即ち同一クラスに存在するメソッドとメソッド内メソッドの間においては、メソッド内のローカル変数および引数は、全てがその内部に定義されたメソッドから参照可能である。

一方で、メソッド内メソッドがサブクラスでオーバーライドされた場合、即ちサブクラスにおけるメソッド内メソッドからは、参照できるローカル変数、引数に制限がある。サブクラスから参照できるローカル変数は、ローカル変数の宣言時にアクセス修飾子 public を付けて宣言されたもののみであり、サブクラスから参照可能であるメソッドの引数は、同じく修飾子 public の付けられたもののみである。public を用いてローカル変数、引数を宣言するコードの例を図 3.4 に、このコードにおけるローカル変数、引数のスコープの様子を図 3.5 に示す。

本システムにおいてこの制限が存在しない場合を仮定すると、スーパークラスのメソッド内のあらゆるローカル変数が、メソッド内メソッドを通じてサブクラスからアクセス可能となり、オブジェクト指向におけるカプセル化が実現できなくなる。この問題を回避するために、本システムではこの様な制限を設けている。

```
1  //スーパークラス
2  public class C {
3    public void method1(public int arg1) {
4      int localVar1;
5      public void method2(int arg2) {
6        public int localVar2;
7        public void mehtod3() {
8          //クラスCにおけるmethod3の実装
9        }
10     }
11   }
12 }
13
14 //サブクラス
15 public class Child extends C {
16   public void method1(int arg1).
17          public void method2(int arg2).
18          public void method3() {
19     //クラスChildにおけるmethod3の実装
20   }
21 }
```

図 3.4: ローカル変数、メソッドの引数に public 修飾子を用いたコード例

3.3　メソッド内メソッドの呼び出しの制限

本システムにおいては、メソッド内メソッドに対し、そのメソッドが定義されたメソッドからの呼び出し、またはそのメソッド自身の再帰呼び出しのみが可能である。

したがって、図 3.6 で示すように、method1 のメソッドボディ内における method3 の呼び出しのような直接定義されたメソッド以外の呼び出しや、method3 のメソッドボディにおける method2 の呼び出しのような内部のメソッドからの外部のメソッドの呼び出しは不可能となっている。

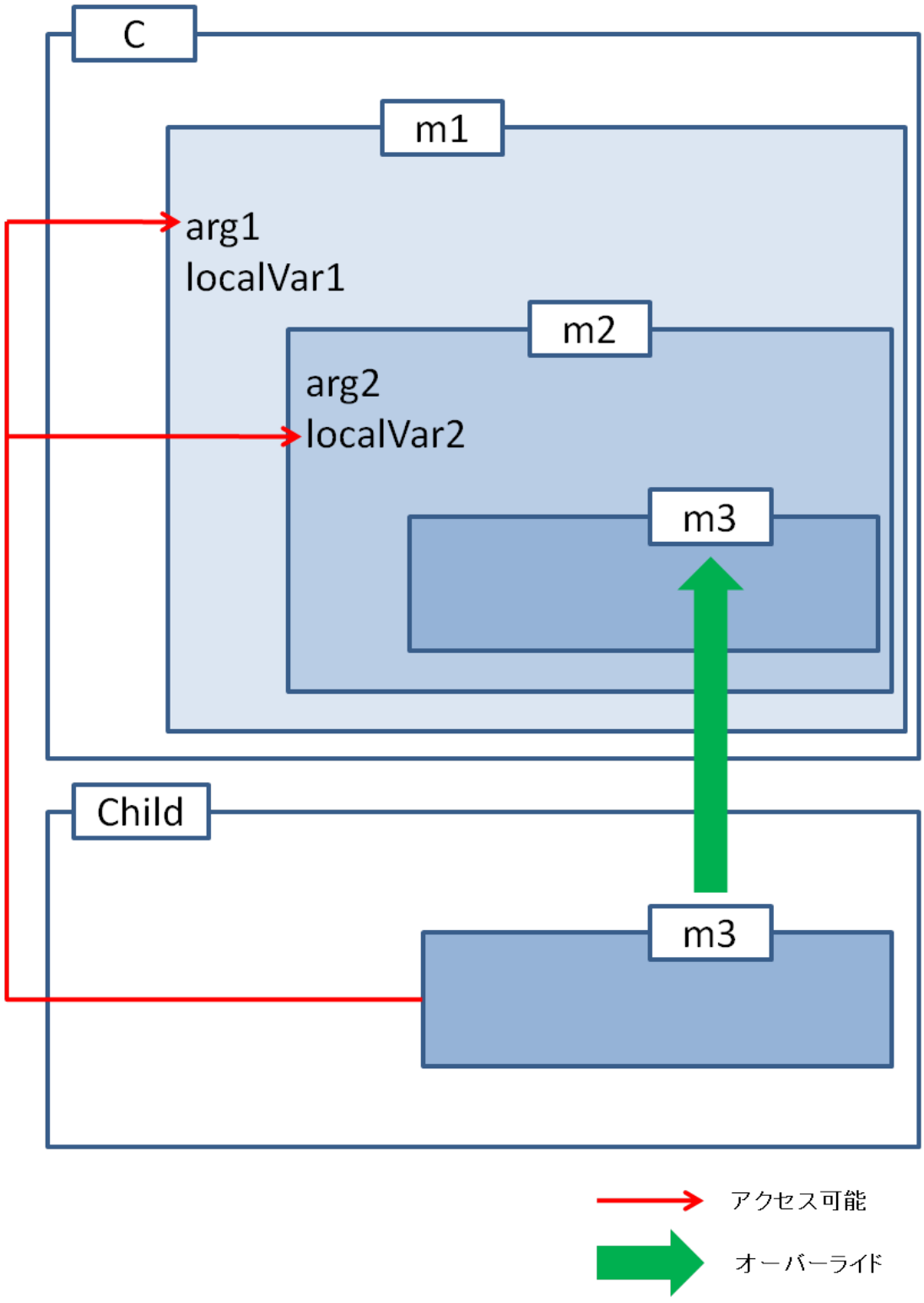

図 3.5:　ローカル変数、メソッドの引数に public 修飾子を用いた際のスコープの様子

```
1  public void method1() {
2    public void mehtod2() {
3      public void method3() {
4        method2();//外側のメソッドの呼び出しは不可能
5      }
6    }
7    method1();//再帰呼び出しは可能
8    method2();//直接の内側のメソッドは呼び出し可能
9    method3();//直接の内側のメソッドでないため、呼び出し不可能
10 }
```

図 3.6: メソッド内メソッドの呼び出しの制限：コード例

第4章　実装

本システムは、コンパイラ実装フレームワークである JastAdd によって生成される Java 言語のコンパイラである JastAddJ を拡張することで実装を行った。

本章では、本システムの実装の概要と、それを述べるにあたって必要となる JastAdd に関連する知識について述べる。

4.1　JastAdd

JastAdd [3] は Java-based なコンパイラを生成するシステムである。JastAdd を用いることで、非常に高い拡張性をもつコンパイラの実装が可能となる。

以下、本システムの実装を説明するにあたって必要な、JastAdd に関連する知識について解説を行う。

4.1.1　抽象構文木

抽象構文木 (abstract syntax tree, AST) とは、プログラムの構造を木構造で表現したものである。JastAdd で用いられる抽象構文木の各ノードは Java のクラスで表現されており、AST ノードと呼ばれる。全ての AST ノードは ASTNode クラスおよびそのサブクラスである。if 文を表す IfStmt クラスが文を表す Stmt クラスのサブクラスであるように、AST ノードクラスでは、より抽象的な表現がスーパークラス、具象的な表現がサブクラスであるような型の階層が作られている。

4.1.2　コンパイラの生成

JastAdd を用いての拡張されたコンパイラの生成には、

- ast ファイル
- parser ファイル

```
1 UnlessStmt : IfStmt;
```

図 4.1: ast ファイルの記述例

- jrag ファイル

が必要となる。それぞれのファイルの概要について、以下で説明する。

ast ファイル

ast ファイルは、AST ノードを表すクラスの継承関係を定義するファイルである。

このファイル内で

```
ClassB : ClassA ;
```

のように記述することで、ClassB を ClassA のサブクラスとして定義できる。

例えば、図 4.1 のように ast ファイルに記述することで、UnlessStmt クラスは JastAdd で既に提供されている IfStmt クラスのサブクラスとなり、IfStmt クラスの実装を引き継ぐことが可能となる。

parser ファイル

parser ファイルは、構文解析のアルゴリズムを定義するファイルである。このファイルの内容に従って、プログラムのソースコードの構文解析を行い、AST ノードを用いた抽象構文木を生成する。

図 4.2 は parser ファイルの記述例である。この例の 1 行目から 4 行目の記述は、`statement` は `unless_then_statement` または `unless_then_else_statement` から還元されることを表している。また、6 行目から 9 行目の記述は、

`UNLESS LPAREN expression.e RPAREN statement.s`

つまり

`unless (expression) statement`

は `unless_then_statement` に還元可能であり、新たに `UnlessStmt` オブジェクトを作成し、抽象構文木を作成することを表している。ここで先頭の `UnlessStmt` はこの文の返り値の型に当たり、中括弧およびコロン（`{: :}`）で囲まれた部分は、この構文規則が適用される際に実行される処理を Java で記述したものである。

```
1  Stmt statement =
2    unless_then_statement.u {: return u; :}
3    |unless_then_else_statement.u {: return u; :}
4    ;
5
6  UnlessStmt unless_then_statement =
7    UNLESS LPAREN expression.e RPAREN statement.s
8    {: return new UnlessStmt(e, s, new Opt()); :}
9    ;
10
11 UnlessStmt unless_then_else_statement =
12   UNLESS LPAREN expression.e RPAREN
13         statement_no_short_if.s ELSE statement.t
14   {: return new UnlessStmt(e, s, new Opt(t)); :}
15   ;
```

図 4.2: parser ファイルの記述例

jrag ファイル

jrag ファイルは、各 AST クラスへ追加されるメソッドやフィールドの定義を行うファイルである。このファイルではメソッドやフィールドの定義はアスペクトを用いたインタータイプ宣言で行われる。インタータイプ宣言とは、他のクラスやインタフェースのメンバ（メソッドやフィールド）を宣言することである。

例えば図 4.3 は、`UnlessStmt` というクラスに、返し値が void である `transformation` メソッドを追加する宣言である。

インタータイプ宣言は、JastAdd が提供する AST クラスと、ast ファイルで定義した新たなクラスの両方に対応しており、JastAdd が提供するクラスの拡張および ast ファイルで定義したクラスの実装はこの jrag ファイルで定義する。

4.2　本システムの実装

本システムの実装は、主に既存の Java への文法の追加と、コード変換の二つの部分からなる。以下では、本システムの実装における Java への文法の追加、コード変換および全体の処理の順序について述べる。

```
1 aspect Unless {
2   public void UnlessStmt.transformation() {
3   Expr co = getCondition();
4   LogNotExpr not = new LogNotExpr(co);
5   setCondition(not);
6   flushCache();
7   }
8 }
```

図 4.3: jrag ファイルの記述例

4.2.1　文法の追加

本システムでは、メソッド内にメソッドの定義を記述するため、およびメソッド内のメソッドをオーバーライドするための文法を追加した。

図 4.4 はメソッド内にメソッドの定義を記述することを可能にするために記述した parser ファイルの一部である。この記述は、メソッド内にメソッドの定義が現れた場合に、それを InnerMethodDeclStmt ノードとして AST に追加することを表している。

図 4.5 はメソッド内のメソッドをオーバーライドするための parser ファイルの一部である。１０行目から１５行目の部分では、メソッドのヘッダをドットで繋げて記述できることを表しており、5行目から8行目の記述は、ヘッダが連続しているメソッド定義に対しては makeORIMethod メソッドを呼び出し、適切なメソッドに変換して AST に追加することを表している。この適切なメソッドへの変換については後述する。

4.2.2　コード変換

本システムでは、メソッド内に定義されたメソッドから外側のメソッドのローカル変数にアクセスすることを可能とするため、ローカル変数をフィールドとして持つようなクラスを生成し、生成したクラスを入れ子にし、ローカル変数へのアクセスはフィールドへのアクセスに変換するという方法を採った。

以下の節では、本システムにおいて行われるコード変換についての概要を述べる。

```
Stmt block_statement_inner =
  inner_method_declaration.m
  {:
    InnerMethodDeclStmt s = new InnerMethodDeclStmt();
          s.setMethodDecl(m);
          return s;
  :}
;

MethodDecl inner_method_declaration =
  inner_method_header.m inner_method_body.b
  {:
    m.setBlockOpt(b);
    return m;
  :}
;
```

図 4.4: メソッド内にメソッドの定義を可能とするための parser ファイルの一部

メソッド内メソッドのコード変換

クラス C のメソッド method1 が内部にメソッド定義 method2 を持つ場合を考える。この場合には、C のメンバに C$method1 クラスおよび C$method2 クラスを追加する。C$method1 クラスは method1 のメソッドボディで宣言されるするローカル変数、引数および method1 メソッドを呼び出すオブジェクトを格納する C 型の$this フィールドを持ち、run$$メソッドを持つ。run$$メソッドは method1 メソッドに後述する変換を行ったものとなる。このとき `public` 修飾子の付いたローカル変数および引数に対しては、対応するフィールドは protected メンバとなり、それ以外のローカル変数、引数に対しては対応するフィールドは private メンバとなる。

C$method2 クラスは、C$method1 と同様に、method2 のメソッドボディで宣言されるするローカル変数、引数および method2 メソッドを呼び出すメソッドを表す C$method1 型のフィールド$outer を持ち、run$$メソッドを持つ。

run$$メソッドの作成に要するメソッドの変換について説明する。run$$メソッドの引数、返り値は変換前の引数と等しい。run$$メソッドのメソッドボディは、変換前のメソッドのメソッドボディにおけるローカル変数の宣言を、それが宣言と同時に初期化を行う場合には先に述べたフィールドへの代入へと変換し、初期化を伴わない宣言の場合にはその宣言文を除去

```
1  MethodDecl method_declaration =
2    ori_method_declaration.o {: return o; :}
3  ;
4
5  MethodDecl ori_method_declaration =
6    method_headers.l method_body.b
7    {: return new MethodDecl().makeORIMethod(l, b); :}
8  ;
9
10 List method_headers =
11   method_header.h
12   {: return new List().add(h); :}
13   | method_headers.l DOT method_header.h
14   {: return l.add(h); :}
15 ;
```

図 4.5: メソッド内メソッドをオーバーライドするための parser ファイルの一部

するよう変換する。また run$$メソッドの先頭で引数のそれぞれに対応するフィールドへの代入を行う。

C$method1 および C$method2 クラスのコンストラクタは、それぞれ C 型, C$method1 型の引数をとり、それをそれぞれ$this, $outer フィールドへ代入する。

メソッド method1 および method2 はクラス C のメンバであり、その返り値は変換前の method1, method2 と等しく、引数は変換前の method1, method2 の引数に加えて、method1, method2 を呼び出すオブジェクトを加えたものである。メソッドボディでは、method1 では C$method1 の、method2 では C$method2 のインスタンスを作成し、作成したインスタンスの run$$メソッドを呼び出し、その結果を return している。ただし、変換前のメソッドの返り値が void であった場合には、return を行わずに run$$メソッドを呼び出してメソッドの実行は終わる。

図 4.6 に変換を行う前のコードの、図 4.7 に変換後のコードの例を示す。

配列定数を用いたローカル変数の変換

先に述べたローカル変数の変換では、配列定数を用いたローカル変数の変換を正しく行うことができない。何故なら、配列定数を用いることが可能なのは配列の初期化時においてのみであり、あらかじめ存在するフィールドへの配列定数の代入は禁じられているからである。したがって、

```
1 public class C {
2   public void method1(public int p1, String p2) {
3     public int lv1 = 0;
4     int lv2;
5     public int method2(int p3, String p4) {
6       return lv1;
7     }
8   }
9 }
```

図 4.6: メソッド内メソッドのコード変換の例：変換前

```
void method() {
int[] a = {1, 2, 3};
}
```

のようなコードに対して先に述べた方法で変換を行うと、

```
class C$method {
private int [] a;
void run$$() {
a = {1, 2, 3};//error
}
}
```

のようになりエラーが発生する。

この問題を解決するため、配列定数を用いたローカル変数の変換時には、新たな配列**配列名$ArrayInit** を宣言し、この配列を配列定数を用いて初期化し、初期化後にこの配列をフィールドへ代入するという変換を行う。配列定数を用いたローカル変数の変換の例を図 4.8, 図 4.9 に示す。

this アクセスの変換

メソッド内に this アクセスが存在する場合には、これに適切な変換を行わなければ this の指すオブジェクトは本来の意図であるそのメソッドが呼び出されたオブジェクトではなく、本システムのコード変換によって生成されたクラスのオブジェクトとなる。

このため、変換前のメソッド内に存在する this アクセスは、コード変換の際に$this フィールドへのアクセスへと変換される。

インナーメソッドの呼び出しの変換

メソッド内にインナーメソッドの呼び出しがある場合には、メソッド呼び出しの引数を、変換前の引数に this を加えたものに変換する。また、インナーメソッドの呼び出し

```
method(args);
```

は全て変換前の this に対するメソッド呼び出し

```
this.method(args);
```

であると解釈する。
以上の変換を行うと、変換前のインナーメソッド呼び出し

```
method(args);
```

は、

```
$this.method(args, this);
```

というコードに変換される。

インナーメソッドの再帰呼び出しの変換

メソッド内メソッドの再帰呼び出しが存在する場合には、先に述べたインナーメソッドの呼び出しの変換の際に引数に渡すオブジェクトを this ではなく、呼び出し側がメソッド内メソッドであれば$outer, メソッド内メソッドでない（通常の）メソッドであれば$this とする。
インナーメソッドの呼び出し、および再帰呼び出しの変換の例を図 4.10、図 4.11 に示す。

インナーメソッドをオーバーライドするメソッドのコード変換

この節では、スーパークラスの持つインナーメソッドをオーバーライドするサブクラスのメソッドの変換について述べる。
本システムにおいては、インナーメソッドのオーバーライドは、該当するメソッドを run$$メソッドをオーバーライドした無名クラスのインスタンスを作成するメソッドでオーバーライドすることで実現する。以下にメソッド定義

```
T1 method1(args1).T2 method2(args2) {
//method body
}
```

を例としてこのコード変換について述べる。ここで、T1, T2 はそれぞれ method1, method2 の返り値であり、args1, args2 は method1, method2 の引数である。

本システムでこのコードの変換を行うと、

```
T2 method2(args2, C$method1 $outer) {
return new C$method2($outer) {
         //method body
        }.run$$(args2);
}
```

のように変換されるなお、このコード内の C はオーバーライドされるインナーメソッドが初めに定義されたクラスであるとする。ここで、method2 によって作成されるクラスを、run$$メソッドがオーバーライドされた無名クラスにすることで、インナーメソッドのオーバーライドを実現している。

スーパークラスのメソッド内メソッドをオーバーライドするサブクラスのメソッドボディ内に this アクセスが存在する場合には、コード変換の際に$outer フィールドおよび$this フィールドを用いて本来意図するオブジェクトを指すように変換を行う。

以下に具体的に this アクセスの変換の方法を説明する。例えば、k 個のメソッド定義 $method_1, method_2, ..., method_k$ があり、$method_i$ が $method_{i-1} (2 \leq i \leq k)$ 内に定義されていたとする。このとき、任意の $method_j$ 内における this アクセスは、j-1 個の$outer と一つの$this アクセスを用いて、

```
$outer.$outer. ... .$outer.$this
```

のように変換される。

図 4.12, 図 4.13 にインナーメソッドをオーバーライドするメソッドのコード変換の例を示す。

4.2.3　処理の順序

JastAddJ におけるコンパイルの手順は、一般に

1. 入力の各ファイルの構文解析、抽象構文木の作成

2. 作成された抽象構文木を元にエラーチェック

3. バイトコードの生成

のように進行する。

本システムにおけるコード変換処理はプログラムの構文解析時および、すべてのファイルの構文解析終了後に行われる。

大部分の変換は構文解析時に抽象構文木を作成する際に行われるが、一部の変換は構文解析時に行うことが困難であるため、全てのファイルの構文解析終了後に行われる。

オーバーライドするメソッドの変換における一部の変換は全てのファイルの構文解析および抽象構文木の作成が終了してからエラーチェックが行われる前、つまり先に示した JastAddJ におけるコンパイルの手順の 1. と 2. の間で行われる。これは、サブクラスにおいてインナーメソッドをオーバーライドするメソッドの変換を行うためには、そのインナーメソッドが定義されたクラスの情報が必要となるが、サブクラスのファイルの構文解析時にその必要な情報を得ることが困難であることからくる。したがって、インナーメソッドをオーバーライドするメソッド定義の抽象構文木作成時には、最終的にそのインナーメソッドが定義されたクラスの名前が必要となる部分を、仮の名前`$TCN$`として抽象構文木を作成する。そしてその後全てのファイルの構文解析が終了したのちに、抽象構文木内の`$TCN$`を適切なクラス名で置き換えることによって、コード変換を完成させる。図 4.14 にこの変換を行う直前の、インナーメソッドをオーバーライドするメソッドのコードの例を示す。

以下に、適切なクラス名を特定し、`$TCN$`をそれで置き換える際の処理の流れを示す。

作成された各抽象構文木に対し、

1. その抽象構文木が`$TCN$`を含んだ名前のノードを持つかどうかを判定する。

2. 抽象構文木が該当するノードを持つ場合には、その抽象構文木に対して、3. 以降の手順を行う。該当するノードを持たない場合には、その抽象構文木内にはインナーメソッドをオーバーライドするメソッドが存在しないことが分かるため、その抽象構文木にはなにも変換を行わずに、次の抽象構文木に対して 1. からの手順を行う。

3. 抽象構文木内に存在するメソッドのうち、`$TCN$`を含むもののリストを作成し、その各メソッド m に対し以下の手順を行う。

   (a) 全ての抽象構文木の中から、名前の末尾が`$(`m のメソッド名`)`であるようなクラス定義 c を検索する。

(b) c を含む抽象構文木において、抽象構文木を根本に向かって探索することで、c を含むクラス定義を得る。

(c) クラス名が$を含まなくなるまで (b) を繰り返し、同時にクラスのフィールドを参照して m 内の変数へのアクセス, メソッド呼び出しを`$outer, $this` を用いて変換する。

(d) クラス名が$を含まなければ、そのクラスは本システムによって生成されたクラスではなく、オーバーライドされるメソッドが定義されたクラスであることが分かるしたがってこのクラスの名前を用いて、メソッド m 内の`$TCN$`を置き換える。

```
1  public class C {
2    public void method1(int p1, String p2) {
3      new C$method1(this).run$$(p1, p2);
4    }
5    public int method2(int p3, String p4, C$method1 $outer) {
6      return $outer.new C$method2($outer).run$$(p3, p4);
7    }
8
9    class C$method1 {
10     public C $this;
11     protected int p1;
12     private String p2;
13     protected int lv1;
14     private int lv2;
15     public C$method1(C $outerArg) {
16       super();
17       $this = $outerArg;
18     }
19     public void run$$(int arg$p1, String arg$p2) {
20       p1 = arg$p1;
21       p2 = arg$p2;
22       lv1 = 0;
23     }
24
25     class C$method2 {
26       public C$method1 $outer;
27       private int p3;
28       private String p4;
29       public C$method2(C$method1 $outerArg) {
30         super();
31         $outer = $outerArg;
32       }
33       public int run$$(int arg$p3, String arg$p4) {
34         p3 = arg$p3;
35         p4 = arg$p4;
36         return lv1;
37       }
38     }
39   }
40   public C() {
41     super();
42   }
43 }
```

図 4.7: メソッド内メソッドのコード変換の例：変換後

```
1 public void method() {
2   int[] a = {1, 2, 3};
3 }
```

図 4.8: 配列定数を用いて初期化されるローカル変数の変換例：変換前

```
1 class C$method {
2       private int[] a;
3       public int run$$() {
4         int[] a$ArrayInit = { 1, 2, 3 } ;
5         a = a$ArrayInit;
6       }
7     }
```

図 4.9: 配列定数を用いて初期化されるローカル変数の変換例：変換後

```
1 public class C {
2   public void method1() {
3     public void method2() {
4       method2();//インナーメソッドの再帰呼び出し
5     }
6     method2();//インナーメソッドの呼び出し
7   }
8 }
```

図 4.10: インナーメソッドの呼び出しの変換例：変換前

```
1  public class C {
2    public void method1() {
3      new C$method1(this).run$$();
4    }
5    public void method2(C$method1 $outer) {
6      $outer.new C$method2($outer).run$$();
7    }
8
9    class C$method1 {
10     public C $this;
11     public C$method1(C $outerArg) {
12       super();
13       $this = $outerArg;
14     }
15     public void run$$() {
16       $this.method2(this);//インナーメソッドの呼び出し
17     }
18
19     class C$method2 {
20       public C$method1 $outer;
21       public C$method2(C$method1 $outerArg) {
22         super();
23         $outer = $outerArg;
24       }
25       public void run$$() {
26         $this.method2($outer);//インナーメソッドの再帰呼び出し
27       }
28     }
29   }
30   public C() {
31     super();
32   }
33 }
```

図 4.11: インナーメソッドの呼び出しの変換例：変換後

```
1  //スーパークラス
2  public class C {
3    public void method1() {
4      public void method2(int i) {
5                  //オーバーライドされるメソッド
6        System.out.println("method2 in superclass");
7      }
8    }
9  }
10
11 //サブクラス
12 public class Child extends C{
13   public void method1().public void method2(int i) {
14         //method1内のmethod2をオーバーライド
15     System.out.println("method2 in subclass");
16   }
17 }
```

図 4.12: インナーメソッドをオーバーライドするメソッドのコード変換例：変換前

```
1  //スーパークラス
2  public class C {
3    public void method1() {
4      new C$method1(this).run$$();
5    }
6    public void method2(int i, C$method1 $outer) {
7    //サブクラスにおいてオーバーライドされるメソッド
8      $outer.new C$method2($outer).run$$(i);
9    }
10
11   class C$method1 {
12     public C $this;
13     public C$method1(C $outerArg) {
14       super();
15       $this = $outerArg;
16     }
17     public void run$$() {
18     }
19
20     class C$method2 {
21       public C$method1 $outer;
22       private int i;
23       public C$method2(C$method1 $outerArg) {
24         super();
25         $outer = $outerArg;
26       }
27       public void run$$(int arg$i) {
28       //サブクラスのmethod2で生成される無名クラスに
29       //オーバーライドされるメソッド
30         i = arg$i;
31         System.out.println("method2 in superclass");
32       }
33     }
34   }
35   public C() {
36     super();
37   }
38 }
39
40 //サブクラス
41 public class Child extends C {
42   public void method2(int i, C$method1 $outer) {
43   //method2をオーバーライド
44     $outer.new C$method2($outer) {
45         public void run$$(int i) {
46         //オーバーライド後のmethod2の振る舞いで
47         //C$method2のrun$$メソッドをオーバーライド
48           System.out.println("method2 in subclass");
49         }
50     }.run$$(i);
51   }
52   public Child() {
53     super();
54   }
55 }
```

図 4.13: インナーメソッドをオーバーライドするメソッドのコード変換例：変換後

```
1  public void method2(int i, $TCN$$method1 $outer) {
2    $outer.new $TCN$$method2($outer) {
3        public void run$$(int i) {
4          System.out.println("method2 in subclass");
5        }
6    }.run$$(i);
7  }
```

図 4.14: TCNを含んだ状態のコードの例

第5章　実験

本システムを用いてコンパイルしたプログラムの性能を判定するため、実験を行った。

5.1　概要

本実験では、図 5.1，5.2，5.3，5.4 に示したコードを本システムのコンパイラを用いてコンパイルし、その実行時間を計測し、比較を行った。

図 5.1 のプログラムでは、method1 内に定義された method2 が、method1 のローカル変数である i, j を参照し、同じく method1 のローカル変数である localVar1, localVar2, localVar3, localVar4 への代入を行う。method1 内で、この method2 を 10000 × 10000 回呼び出しており、この method1 の実行にかかった時間を計測した。

図 5.2 に示したプログラムは、図 5.1 に示したプログラムを通常の Java で記述したものである。通常の Java では当然ながらメソッド内にメソッドを定義することは不可能であり、したがって異なるメソッドのローカル変数の参照も不可能である。そのため、図 5.1 におけるメソッド内メソッドに相当する図 5.2 のメソッド method2 は通常のメソッドとして定義されており、メソッド method1 で宣言されたローカル変数への method2 からのアクセスは、method2 呼び出し時に引数として必要なローカル変数を渡すことで実現している。さらに、method2 の内部からの method1 のローカル変数への代入は不可能であるため、図 5.2 では必要な値をメソッドの返り値として返し、method1 内で、method2 の返り値を元にローカル変数への代入を行う（図 5.2, 12-15 行）。この際、Java におけるメソッドの返り値は一つであるので、method2 では複数の値を返すために ReturnValue オブジェクトを作成し、それを return することで複数の値を返している。それに伴い、 5.1 では void であった method2 の返り値の型は 5.2 では ReturnValue に変更した。

図 5.3 のプログラムでは、method1 内に定義された method2 が method1 のローカル変数である i, j を参照し、計算結果を返す。method1 内で、この method2 を 10000 × 10000 回呼び出しており、この method1 の実行にかかった時間を計測した。

```
1  public class C {
2    public void method1() {
3      int localVar1 = 0;
4      int localVar2 = 0;
5      int localVar3 = 0;
6      int localVar4 = 0;
7      public void method2() {
8        localVar1 = i + j;
9        localVar2 = i - j;
10       localVar3 = i * j;
11       localVar4 = i / j;
12     }
13     for (int i = 1; i <= 10000; i++) {
14       for (int j = 1; j <= 10000; j++) {
15         method2();
16       }
17     }
18   }
19
20   public static void main(String[] args) {
21     C c = new C();
22     long start = System.currentTimeMillis();
23     c.method1();
24     long stop = System.currentTimeMillis();
25     System.out.println(stop - start);
26   }
27 }
```

図 5.1: プログラム 1

図 5.4 のプログラムは、図 5.3 のプログラムを本システムを用いずに記述したものである。図 5.3, 5.4 のメソッド内メソッド method2 は、図 5.1, 5.2 とは異なり、ローカル変数への代入に当たる動作を行っていない。このため、図 5.4 の method2 では、図 5.2 における ReturnValue の様なクラスは必要とせず、単純に計算結果をメソッドの返り値として返している。

本実験における実験環境を以下に示す。

- OS: Windows 7
- CPU: Intel Core i5
- メモリ: 4.00GB

表 5.1: 実行時間（ミリ秒）

	時間
プログラム 1	1261
プログラム 2	1515
プログラム 3	1694
プログラム 4	352

5.2　結果

図 5.1 および 5.3 のコードを本システムを用いて変換したものの一部を、図 5.5, 5.6 に示す。

実験の結果を表 5.1 に示す。

5.3　考察

プログラム 1 は、プログラム 2 と比較して、約 17 %実行時間が短くなった。プログラム 1 における method2 では、一回の実行につき、`C$method2` クラスのオブジェクトの作成と、`run$$`メソッドにおける算術計算、ローカル変数への代入が行われる。一方でプログラム 2 における method2 では、一回の実行につき、method2 内部での算術計算、ReturnValue クラスのオブジェクトの作成、method2 から返された ReturnValue オブジェクト method1 におけるローカル変数への代入が行われる。両プログラムでともにオブジェクトの作成、算術計算を行っているため、この実行時間の差は、主にメソッド内メソッドからの返り値を元にローカル変数への代入を行うことからくると思われる。

また、プログラム 3 とプログラム 4 を比較すると、プログラム 4 の実行時間が大幅に短く、プログラム 3 よりおよそ 79 %少ない実行時間であった。プログラム 4 における method2 は、一回の実行毎に一つの `C$method2` クラスのオブジェクトを作成し、算術計算を行っているのに対し、プログラム 3 の method2 は、一回の実行につき算術計算を行うだけであることが、この実行時間の差の原因であると思われる。

以上より、ローカル変数への代入が頻繁に行われるようなプログラムに対しては、本システムを用いて記述したプログラムは、既存の Java で書かれたプログラムと比べてコストの高いオブジェクトの作成を行う回数が増えなく、またメソッド内メソッドを呼び出す側でローカル変数への代入を行う必要がないため、実行速度の上で優れていることが分かる。一方で、ローカル変数への代入を頻繁に行わないようなプログラムに対して

は、本システムを用いた場合には、用いない場合と比べ、オブジェクトの作成回数が増えるため、パフォーマンスは悪化すると考えられる。

```
1  public class C {
2    public void method1() {
3      int localVar1 = 0;
4      int localVar2 = 0;
5      int localVar3 = 0;
6      int localVar4 = 0;
7      for (int i = 1; i <= 10000; i++) {
8        for (int j = 1; j <= 10000; j++) {
9          ReturnValue result = method2(i, j);
10         localVar1 = result.r1;
11         localVar2 = result.r2;
12         localVar3 = result.r3;
13         localVar4 = result.r4;
14       }
15     }
16   }
17   public ReturnValue method2(int i, int j) {
18     int r1 = i + j;
19     int r2 = i - j;
20     int r3 = i * j;
21     int r4 = i / j;
22     return new ReturnValue(r1, r2, r3, r4);
23   }
24
25   class ReturnValue {
26     int r1;
27     int r2;
28     int r3;
29     int r4;
30     public ReturnValue(int p1, int p2, int p3, int p4) {
31       r1 = p1;
32       r2 = p2;
33       r3 = p3;
34       r4 = p4;
35     }
36   }
37
38   public static void main(String[] args) {
39     C c = new C();
40     long start = System.currentTimeMillis();
41     c.method1();
42     long stop = System.currentTimeMillis();
43     System.out.println(stop - start);
44   }
45 }
```

図 5.2: プログラム 2

```
1  public class C {
2    public void method1() {
3      int result = 0;
4      public int method2() {
5        int r1 = i + j;
6        int r2 = i - j;
7        int r3 = i * j;
8        int r4 = i / j;
9        return r1 + r2 + r3 + r4;
10     }
11     for (int i = 1; i <= 10000; i++) {
12       for (int j = 1; j <= 10000; j++) {
13         result = method2();
14       }
15     }
16   }
17
18   public static void main(String[] args) {
19     C c = new C();
20     long start = System.currentTimeMillis();
21     c.method1();
22     long stop = System.currentTimeMillis();
23     System.out.println(stop - start);
24   }
25 }
```

図 5.3: プログラム 3

```
1  public class C {
2    public void method1() {
3      int result = 0;
4      for (int i = 1; i <= 10000; i++) {
5        for (int j = 1; j <= 10000; j++) {
6          result = method2(i, j);
7        }
8      }
9    }
10
11   public int method2(int i, int j) {
12     int r1 = i + j;
13     int r2 = i - j;
14     int r3 = i * j;
15     int r4 = i / j;
16     return r1 + r2 + r3 + r4;
17   }
18
19   public static void main(String[] args) {
20     C c = new C();
21     long start = System.currentTimeMillis();
22     c.method1();
23     long stop = System.currentTimeMillis();
24     System.out.println(stop - start);
25   }
26 }
```

図 5.4: プログラム 4

```
1  public class C {
2    public void method1() {
3      new C$method1(this).run$$();
4    }
5    public void method2(C$method1 $outer) {
6      $outer.new C$method2($outer).run$$();
7    }
8
9    class C$method1 {
10     public C $this;
11     private int localVar1;
12     private int localVar2;
13     private int localVar3;
14     private int localVar4;
15     private int i;
16     private int j;
17     public C$method1(C $outerArg) {
18       super();
19       $this = $outerArg;
20     }
21     public void run$$() {
22       localVar1 = 0;
23       localVar2 = 0;
24       localVar3 = 0;
25       localVar4 = 0;
26       for(i = 1; i <= 10000; i++) {
27         for(j = 1; j <= 10000; j++) {
28           $this.method2(this);
29         }
30       }
31     }
32
33     class C$method2 {
34       public C$method1 $outer;
35       public C$method2(C$method1 $outerArg) {
36         super();
37         $outer = $outerArg;
38       }
39       public void run$$() {
40         localVar1 = i + j;
41         localVar2 = i - j;
42         localVar3 = i * j;
43         localVar4 = i / j;
44       }
45     }
46   }
47 }
```

図 5.5: 5.1 の変換後（一部）

```
1  public class C {
2    public void method1() {
3      new C$method1(this).run$$();
4    }
5    public int method2(C$method1 $outer) {
6      return $outer.new C$method2($outer).run$$();
7    }
8
9    class C$method1 {
10     public C $this;
11     private int result;
12     private int i;
13     private int j;
14     public C$method1(C $outerArg) {
15       super();
16       $this = $outerArg;
17     }
18     public void run$$() {
19       result = 0;
20       for(i = 1; i <= 10000; i++) {
21         for(j = 1; j <= 10000; j++) {
22           result = $this.method2(this);
23         }
24       }
25     }
26
27     class C$method2 {
28       public C$method1 $outer;
29       private int r1;
30       private int r2;
31       private int r3;
32       private int r4;
33       public C$method2(C$method1 $outerArg) {
34         super();
35         $outer = $outerArg;
36       }
37       public int run$$() {
38         r1 = i + j;
39         r2 = i - j;
40         r3 = i * j;
41         r4 = i / j;
42         return r1 + r2 + r3 + r4;
43       }
44     }
45   }
46 }
```

図 5.6:　5.3 の変換後（一部）

第6章　まとめと今後の課題

6.1　まとめ

本研究では、Java 言語にローカル変数を参照できるメソッド内メソッドを導入することを提案し、実装した。本システムにおけるメソッド内メソッドは、単にメソッドの一部を別のメソッドとするだけでなく、その外側のメソッド内で宣言されたローカル変数を参照できるため、より細かい粒度での、柔軟なコードの再利用が可能となる。

Java 言語において、メソッドより細かい粒度でのコードの再利用は困難であった。そのため本システムでは、メソッド内にメソッドを定義することを許し、それをオーバーライドすることでブロック単位でのコードの再利用を可能にした。そして、コードを変換してクラスを作成し、メソッド内のローカル変数をそのクラスのフィールドとして変換することで、メソッド内メソッドからのローカル変数へのアクセスを実現した。また、カプセル化を破壊しないために、ローカル変数に対するアクセス修飾子 public の有無で、サブクラスからのローカル変数へのアクセスの可否を制限した。

本システムは、コンパイラ実装フレームワークである JastAdd によって実装された Java コンパイラである JastAddJ を拡張することで実装した。文法の追加を行い、構文解析時および構文解析終了後にコードの変換を行うことによって、本システムのメソッド内メソッドを実現した。

実験を行い、ローカル変数への代入が頻繁に行われるようなプログラムに対しては、本システムはより高い実行速度が出せることが分かった。

6.2　今後の課題

本システムについては、まだ改善すべき点が存在する。以下にそれらについて述べる。

6.2.1　本システムの仕様

メソッド内メソッドの呼び出しの制限

現在の仕様では、メソッド内メソッドは、それが定義されたメソッドからの呼び出し、そのメソッド自身からの再帰呼び出しのみが可能であるという制限がある。この制限を緩め、階層的に隔たりのあるメソッドからのメソッド内メソッドの呼び出しを可能とすることについて、検討を要する。

メソッド内メソッドのオーバーライド

メソッド内メソッドをオーバーライドする記述において、冗長であるため、これは改善の必要がある。具体的には、オーバーライドするメソッドの外側のメソッドの修飾子、引数の名前などは省略できて然るべきである。

6.2.2　実装

現在の実装では、メソッド内メソッドおよびメソッド内メソッドを含むメソッド全てに対して対応するクラスを作成している。しかし、内部にメソッド定義を含まないメソッドに対しては、この対応するクラスを作成する必要がない場合がある。クラス作成の必要がない場合を判定し、その場合にはクラスの作成を行わないようにすることで、メソッド呼び出し時のオブジェクト作成の回数が減少し、オーバーヘッドが減少すると考えられる。

参考文献

[1] Bodden, E.: Closure joinpoints: Block Joinpoints without Surprises (2011).

[2] Knudsen, J. L., Lofgren, M. and Magnusson, O. L. M. B.: *Object-Oriented Environments - The Mjolner Approach*, Prentice Hall (1994).

[3] Team, T.: JastAdd, `http://jastadd.org`.

[4] 赤井駿平, 千葉滋: コード領域を対象とする関心事を扱うためのアスペクト指向プログラミング言語の拡張, 情報処理学会プログラミング研究会発表資料 (2010).