平成18年度 修士論文

アスペクト指向言語による
例外処理の記述方法の改善

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
学籍番号 05M37139
熊原 奈津子

指導教員
千葉 滋 助教授

平成19年1月19日

概要

プログラムの実行環境の問題によって例外が発生した場合、その例外に対して適切な処理をすべきである。例外が発生したことを見落として正常時の動作を継続してしまうとより深刻で致命的な異常事態を招いてしまう恐れがある。しかし、プログラムを記述する際には、例外処理に関してより、ロジックを書くのに集中できた方がよい。また、ロジックを記述した後に必要に応じて例外処理を記述したい場合もある。例えば、サーバに負荷をかけて性能を測定するという大規模な実験を行うために複数クライアントを起動させる制御プログラムを作成する場合、必要に応じて例外処理を変更できるとよい。

このような例外処理記述を可能にするために、我々はアスペクト指向システム GluonJ/R を提案する。GluonJ/R がもつ `block` ポイントカット指定子を用いることでプログラム中の範囲を指定することができ、`recover` アドバイスを用いて指定した範囲内で例外が発生した場合の処理を記述できる。これにより、例外処理をプログラムロジックから分離して記述することができ、後から容易に追加・変更・削除できるようになる。また、GluonJ/R は指定した範囲の先頭に戻ってその範囲の処理を再実行することができるという、アドバイスの中で使える特殊なメソッドも提供している。

GluonJ/R はバイトコード変換で例外処理をプログラムロジックに埋め込む。その時、バイトコード上のジョインポイントを選択するのではなくクラスファイルに含まれる行番号を利用して、ソースコード中のジョインポイントを選択する。このようにすることで、`VerifyError` が起こることを回避することが出来るようになる。また、例外処理をアスペクトとして分離して記述してもソースコードに直接 `try-catch` 文を記述した場合とほぼ実行時間は変わらないことが GluonJ/R が生成したコードの実行時間と `try-catch` 文を記述したコードの実行時間を測定し比較することによって分かった。

謝辞

本研究を行うにあたってあらゆる面で支えていただいた方々へ感謝の意を示します。指導教官の東京工業大学 千葉滋 助教授には大学四年生から大学院修士課程までの三年間、研究の方向性や進め方、研究発表についてご指導いただき、大変感謝しております。東京工業大学 光来健一 助手には研究内容や研究発表についてさまざまな意見をいただきました。同研究室所属の先輩方、西澤無我氏、柳澤佳里氏、石川零氏、薄井義行氏、日比野秀章氏には本研究に必要な多くの知識を与えていただき、様々な助言を頂きました。同研究室の同輩、青木康博氏、竹内秀行氏には大学四年生で同研究室に所属して以来、共に研究に励みさまざまな相談を聞いていただきました。その他、同研究室に所属する皆様のお陰で充実した研究を行うことができました。

また、研究に行き詰った際にも様々な方に支えられて本研究を続けてくることができました。テニス等を通してリフレッシュさせてくださった入澤寿平氏をはじめとする東京工業大学硬式庭球部の皆様、どのような相談にも耳を傾け励ましてくださった同専攻所属の立薗真樹氏には大変感謝しております。

最後に、熊原義一氏、久美氏、亮子氏には学生生活を送るにあたって経済的な面で支援していただくと共に、本研究を行うのに十分な環境を与えてくださったことに尚一層の感謝の意を表します。

目 次

図 目 次

表 目 次

第1章　はじめに

1.1　研究の背景と目的

プログラムの実行環境の問題によって例外が発生した場合、その例外に対して適切な処理をすべきである。例外が発生したことを見落として正常時の動作を継続してしまうとより深刻で致命的な異常事態を招いてしまう恐れがある。

しかし、プログラムを記述する際には、例外処理に関してより、ロジックを書くのに集中できた方がよい。また、ロジックを記述した後に必要に応じて例外処理を記述したい場合もある。本稿ではこの問題点の例として、サーバに負荷をかけて性能を測定するという大規模な実験を行うために複数クライアントを起動させる制御プログラムを作成する場合を例にとって考えていく。

1.2　本研究の提案

前章のような例外処理に関する問題点を解決するために、我々は例外処理を行うのに特化したアスペクト指向システム GluonJ/R を提案する。GluonJ/R がもつ `block` ポイントカットを用いてプログラム中の範囲を指定し、その指定した範囲内で例外が発生した場合の処理を `recover` アドバイスを用いて記述する。これにより、例外処理をプログラムロジックから分離して記述することができるようになる。また、指定した範囲の先頭に戻ってその範囲の処理を再実行することができるという、アドバイスの中で使える特殊なメソッドも提供している。

GluonJ/R はバイトコード変換で例外処理をプログラムロジックに埋め込む。その時、バイトコード上のジョインポイントを選択するのではなくクラスファイルに含まれる行番号を利用して、ソースコード中のジョインポイントを選択する。このようにすることで、`VerifyError` が起こることを回避することが出来るようになる。また、例外処理をアスペクトとして分離して記述してもソースコードに直接 `try-catch` 文を記述した場合とほぼ実行時間は変わらないことが GluonJ/R が生成したコードの実行

時間と `try-catch` 文を記述したコードの実行時間を測定し比較することによって分かった。

1.3　本論文の構成

以下、2 章では例外処理を行う際の問題点とその具体例を示し、3 章ではこの問題を解決するために我々が提案する GluonJ/R について述べる。4 章では GluonJ/R の実装について述べ、5 章で GluonJ/R が出力したコードの性能を調べた実験について報告する。6 章では GluonJ/R の関連研究を取り上げ、7 章で本稿をまとめる。

第2章　例外処理の記述方法と問題点

プログラムの実行環境の問題によって例外が発生した場合、その例外に対して適切な処理をすべきである。例外が発生したことを見落として正常時の動作を継続してしまうとより深刻で致命的な異常事態を招いてしまう恐れがある。本章では例外処理の記述の重要性と共に、既存の技術を用いて例外処理を記述する方法とそれらの問題点について述べる。

2.1　例外処理の重要性

プログラムの実行環境の問題によって例外が発生した場合、その例外に対して適切な処理をすべきである。例として、分散環境上で動くサーバマシンの負荷テストをしており、サーバに対して負荷を発生させるプログラムを複数のクライアント上で起動する制御プログラムを作成しているとする。各クライアントに必要なファイルを送信する部分のプログラムは図2.1のようになるだろう。

この場合、サーバとクライアントをつなぐネットワーク障害に障害が発生したときや、クライアントマシンがそもそも起動していないときに、例外が発生する可能性がある。例えば、8行目で `UnknownHostException` や `IOException`、11行目で `IOException`、13行目で `FileNotFoundException` など、起こり得る例外はたくさんある。これらの例外は、もし発生すると、`sendFile()` メソッドの実行を中断し、`sendFile()` メソッドの呼び出し側に投げられる。

このような実験のうち、サーバに負荷をかけるクライアントのマシンの台数を変えて実験をしている場合、マシンの台数が揃っていることを前提とした実験をすることになる。その場合、サーバとクライアントをつなぐネットワークの調子が悪かったりクライアントマシンが起動していないなどの例外が発生した場合、意図した実験が行われていないことになってしまう。

```
 1 class Sender{
 2   public void sendFile(String host,
 3       String fileName) throws Exception {
 4     int n;
 5     int port = 9000;
 6     byte[] buff = new byte[1024];
 7
 8     Socket s = new Socket(host, port);
 9     DataOutputStream out
10         = new DataOutputStream(
11             s.getOutputStream());
12     RandomAccessFile file
13         = new RandomAccessFile(fileName, "r");
14
15     while((n = file.read(buff)) > 0){
16       out.write(buff, 0, n);
17     }
18
19     file.close();
20     out.close();
21     s.close();
22     System.out.println(fileName
23             + " has been sent to " + host);
24   }
25 }
```

図 2.1: ファイルをクライアントに送信するプログラム例

2.2　例外処理の分離の必要性

前章のような実験プログラムのロジックを記述する際には、例外の処理に関しては後で記述したいことが多い。なぜなら、例外処理を書かなくても多くの場合はうまく動くので、最初の段階ではロジックを書く方に集中できた方がよいからである。例外処理は、後で必要になったとき、はじめて実験プログラムに追加できると望ましい。

しかし、プログラムに例外処理を不用意に後から加えると、既に動いているプログラムを変更することになるため、そのロジックを壊してしまう危険性がある。また、追加した例外に修正や変更が加えられた場合には、対象となる例外処理を全て探し出して逐一変更しなければならなくなる。そのようなプログラムの追加は、面倒であるだけでなく、1 つでも変更し忘れるとプログラム全体の整合性がとれなくなる危険性がある。

```
class Sender{
  public void sendFile(String host,
      String fileName) throws Exception {
    int n;
    int port = 9000;
    byte[] buff = new byte[1024];

                    (1)

        Socket s = new Socket(host, port);
          :
        s.close();

                    (2)

    System.out.println(fileName
              + " has been sent to " + host);
  }
}
```

図 2.2: 実験プログラム

追加する例外処理は、実験プログラムの例の場合、図 2.2 の四角で囲んだ部分に書くことになる。図 2.2 は例外処理について記述されていない、プログラムのロジックだけが書かれた実験プログラムの断片である。例えば図 2.2 の (1) に

```
try{
```

を追加し、(2) に

```
}catch(IOException e){
  e.printStackTrace();
}
```

などと追加することになるだろう。

ところが実験プログラムの場合、実験の内容によって発生する例外をどのように処理したいかが変わる可能性がある。例えば、例外処理の内容を、エラーログを画面に出力することから、実験者にメールを送信するように変えるとする。その場合、プログラム中の全ての catch 節を修正しなければならないが、修正もれがあると、全ての例外がメールによって送信されず、例外に気づくのに遅れてしまう恐れがある。

さらには、例外が起こった時点でプログラムの実行を止めずに、処理全体をやり直したいときもある。大きな実験プログラムを実行する場合、全体の処理時間が長くなるので、途中で例外が発生したからといって、実験を途中で止めて最初からやり直すのは好ましいとはいえない。例えば、サーバとクライアント間でネットワーク障害が起こった場合、時間をおいて再試行すれば障害が解決してうまくいく場合がある。また、クライアントマシンからの応答がなかった場合は代替のマシンに換えて再試行することで実験を続行できる場合もある。このように、プログラム全体を止めずに適切にリカバリ処理ができれば、それまでの実験の結果を無駄にせずに済む。

しかし、従来の Java 言語の範囲内ではリカバリ処理を記述するのは困難であった。つまり、プログラムの状態を、例外が起こらないような設定に変えて、もう一度同じ処理を繰り返させるような記述は、必ずしも容易ではなかった。例えば、try-catch 文の try ブロックの部分を catch 節の中から再試行したくても、そのような機能は Java 言語にはない。try ブロックの部分をメソッドにして catch 節の中でそのメソッドを呼ぶようにすれば、目的は達成できるが、catch 節の中にまた try-catch 文を書かなければならず、プログラムが見づらくなる。例えば図 2.2 の場合、2.3 のようになり、見づらい。なお sendFileBody() は元の try ブロックの中身を実行するメソッドであり、getAnotherHost() は代替マシンが存在すればそのホスト名を返すメソッドである。

```
try {
  sendFileBody(host, fileName, port, buff, n);
} catch (IOException e) {
  host = getAnotherHost();
  try {
    sendFileBody(host, fileName, port, buff, n);
  } catch (IOException e) {
    e.printStackTrace();
  }
}
```

図 2.3: リカバリ処理の記述例

try-catch 文を do-while 文で囲むことでリカバリ処理を実現する方法もある。つまり、図 2.2 の (1) に

```
Exception e = null;
do {
  try{
```

を追加し、(2) に

```
  } catch (IOException err) {
    e = err;
    host = getAnotherHost();
  }
} while (e != null);
```

を追加すれば、リカバリ処理を実装できる。しかし、先の方法と同様、プログラムが見づらくなる。

第3章　GluonJ/R

前章の問題点を解決するため、例外処理をアスペクトとして記述できるようにしたアスペクト指向システム *GluonJ/R* (GluonJ with Recovery) を提案する。例外処理をアスペクトとすることで、プログラムロジックから、その中で起こった例外の処理を分離して記述することができるようになる。3.1 問題点を解決するために用いられたアスペクト指向プログラミングについて説明し、3.2 章以降では提案するアスペクト指向システムが持つ機能について説明する。

3.1　アスペクト指向プログラミング

設計時にプログラムの変化や進化を明確に予測できればオブジェクト指向で十分である。どのようなモジュール単位の交換や変更、再利用が必要なのかが設計時にある程度分かっているとデザインパターンを用いるなどして、うまく設計することができる。しかしながら、将来にわたるプログラムの変化を、予め予測するのは多くの場合簡単ではない。2.2 章で述べたように、例外処理についても同様のことが言える。ここで注目されるべきなのが、アスペクト指向プログラミングは将来の変化や進化に強いモジュール化技術だということである。[16] アスペクト指向言語は多数存在しているが、本 3.1 章では代表的なアスペクト指向言語である AspectJ[12][1] の説明することでアスペクト指向言語の概要を述べるとする。

3.1.1　AspectJ

AspectJ では横断的関心事を 1 つのモジュールにまとめるためにアスペクトというコンポーネントを用いる。アスペクトはクラスを拡張したもので、フィールドやメソッドに加えてポイントカットやアドバイスといった要素から構成される。ポイントカットとは実行中のプログラムのある時点を指定するための記述である。プログラムの実行がポイントカットが指定する時点に差し掛かったら、そのポイントカットと結びついたアドバイスが実行される。アドバイスは横断的関心事を実装するための要素で、ポ

イントカットと組み合わせて用いる。この他にアスペクトはインタータイプ宣言等の要素から構成される。これらアスペクトを構成する要素について順に説明する。

ポイントカット

ポイントカットとはプログラムの実行中のある時点を指定するための記述である。ポイントカットを用いて指定できる時点のことをジョインポイントと呼ぶ。ポイントカットを用いて指定できるジョインポイントの種類は、アスペクト指向言語によって異なる。AspectJ では、メソッドを実行する時点やフィールドをアクセスする時点を指定することができる。AspectJ ではポイントカットをポイントカット指定子の組み合わせで表す。表 3.1 は、AspectJ で用いることができるポイントカット指定子の一部を表している。

例えば `Sender` クラスの `sendFile` メソッドを実行する時点を指定するポイントカットは、以下のように書ける。

```
execution(void Sender.sendFile(..))
```

ポイントカット指定子を組み合わせるには論理演算子を用いる。AspectJ は&&、||、！という 3 種類の論理演算子を提供している。

アドバイス

ポイントカットが指定した特定のジョインポイントで実行される処理をアドバイスと呼ぶ。AspectJ のアドバイスには幾つか種類があり、それぞれ処理を実行するタイミングが異なる。代表的なアドバイスとしては before アドバイス、after アドバイス、around アドバイスがある。それぞれのアドバイスは記述されたコードを、ジョインポイントの直前、ジョインポイントの直後、またジョインポイントの代わりに実行する。

インタータイプ宣言

AspectJ のアスペクトは、既存クラスにフィールドやメソッドなどを新たに追加する機能を持つ。これをインタータイプ宣言と呼ぶ。

指定子の名前	指定するジョインポイントの種類
execution (behavior-pattern)	behavior-pattern が表すメソッド・コンストラクタを実行する時点
call (behavior-pattern)	behavior-pattern が表すメソッド・コンストラクタを呼び出す時点
set (field-pattern)	field-pattern が表すフィールドへ値を代入する時点
get (field-pattern)	field-pattern が表すフィールドの値を参照する時点
within (class-pattern)	class-pattern がクラスの内部にあるジョインポイント
cflow (pointcut)	pointcut が表すジョインポイントに含まれるジョインポイント
this (type or id)	実行中のオブジェクトが、type が表すクラスもしくは id の表す変数の型であるジョインポイント
target (type or id)	処理の対象のオブジェクトが、type が表すクラスもしくは id の表す変数の型であるジョインポイント
args (types or ids)	処理の引数のオブジェクトが、type が表すクラスもしくは id の表す変数の型であるジョインポイント

表 3.1: AspectJ の代表的なポイントカット指定子

3.1.2　GluonJ

本研究室で開発した GluonJ はアスペクト指向システムの一つである。AspectJ のアスペクトに相当するものは GluonJ では`@Glue` によって注釈されたクラスによって表現される。`@Glue` で注釈されたクラスを構成する要素はポイントカットとアドバイス、もしくはクラスの改良である。以下ではこれらについて詳しく述べるとする。

ポイントカットフィールド

`@Glue` で注釈されたクラスにポイントカットとアドバイスを表しているフィールドを宣言することができる。これをポイントカット・フィールドと呼ぶ。ポイントカット・フィールドは `Pointcut` 型で、`@Before` 、

`@After` 、`@Around` のいずれかで注釈されなければならない。表 3.2 と表 3.3 はポイントカットフィールドを宣言するのに用いることができるファクトリメソッドとポイントカットフィールドに注釈できるアノテーションについてである。

ファクトリメソッド	指定するジョインポイントの種類
call(String *methodPattern*)	methodPattern によって指定されるメソッド、もしくはコンストラクタが呼ばれるとき
set(String *fieldPattern*)	fieldPattern によって指定されるフィールドの値を書き込むとき
get(String *fieldPattern*)	fieldPattern によって指定されるフィールドの値を読み込むとき
within(String *classPattern*)	classPattern によって指定されるクラス内で、宣言されているメソッドが実行されている間
within(String *classPattern*)	methodPattern によって指定されるメソッドが実行されている間
annotate(String *annotationPattern*)	annotationPattern によって指定されるアノテーションつきのメソッドまたはフィールドがアクセスされている間
when(String *javaExpression*)	javaExpression が `true` であるとき
cflow(String *methodPattern*)	methodPattern によって指定されるメソッドが実行されている間

表 3.2: GluonJ の代表的なポイントカット指定子

また、`.and` `.or` を利用して、より細やかなポイントカットを定義することができる。call、get、および set 以外のポイントカットは、通常、それら 3 つのポイントカットの 1 つと一緒に用いられる。

クラスの改良

AspectJ と同様、既存のクラスにフィールドやメソッドなどを新たに追加することができる。図 3.1 では `Sender` クラスに新たに `getAnotherHost()` という他のホスト名を返すというメソッドを追加している。

アノテーション	アドバイスを実行するタイミング
@Before(*adviceBody*)	注釈したポイントカット・フィールドによって指定されたタイミングの直前
@After(*adviceBody*)	注釈したポイントカット・フィールドによって指定されたタイミングの直後
@Around(*adviceBody*)	注釈したポイントカット・フィールドによって指定された計算の代わり

表 3.3: ポイントカット・フィールドを注釈するアノテーション

```
@Refine static class AnotherHost extends Sender {
    public String getAnotherHost() {
        //他のホスト名を返す
    }
}
```

図 3.1: GluonJ におけるクラスの改良例

3.2　block ポイントカットと recover アドバイス

GluonJ/R は、AspectJ のような一般的なアスペクト指向システムがもつ機能に加えて、block **ポイントカット** と recover **アドバイス**を提供している。block ポイントカットは、2 つのジョインポイントの組を選択するためのポイントカット指定子である。block ポイントカットによって選ばれたジョインポイントの組で囲まれた範囲で例外が発生したときに実行されるコードが recover アドバイスである。なお block ポイントカットで指定される範囲は、同一メソッドの中に含まれていなければならない。

例えば、2 章で示したファイルを送信するプログラムの例外処理を、block ポイントカットと recover アドバイスを使ってアスペクトとして書くと、図 3.2 のようになる。GluonJ/R は 我々の研究室で開発したアスペクト指向システム GluonJ [3] を拡張したものなので、GluonJ の文法に従って書くことになる。GluonJ では @Glue で注釈されたクラスがアスペクトになる。そして、そのアスペクト内で Pointcut 型のフィールド（ポイントカット・フィールドという）を宣言することでポイントカットを指定することができる。具体的には Pcd クラスが持つメソッドを利用してポイントカットを指定する。

block ポイントカットは Pcd クラスに存在するメソッド block を用い

```
import javassist.gluonj.Glue;
import javassist.gluonj.Pcd;
import javassist.gluonj.Pointcut;
import javassist.gluonj.plugin.Block.Recover;

@Glue
class FileSenderRecovery {
  @Recover(etype = "java.io.IOException",
           advice = "{ $1 = getAnotherHost();"
         + " javassist.gluonj.GluonJR.retry(); }")
  Pointcut p = Pcd.block(
      Pcd.call("java.net.Socket#new(..)"),
      Pcd.call("java.io.PrintStream#println(..)"));
}
```

図 3.2: アスペクトの記述例

て指定し、メソッド block の引数には 2 つのポイントカットのペアを渡す。このポイントカットのペアで例外処理を追加したい範囲の始点と終点を宣言する。このとき、例外処理を追加したい範囲の始点と終点は同一メソッド内に存在していないとポイントカットされない。図 3.2 では、範囲の始点と終点を call メソッドを用いて宣言しているが、この call メソッドの引数となる文字列は

（クラス名）#（メソッド名（引数））

で記述する。このとき、new というメソッドはコンストラクタ（オブジェクト生成時）をポイントカットし、メソッド名の引数は .. で省略することができる。call の他にも set や get といったメソッドも存在し、

（クラス名）#（フィールド名）

と記述することでフィールドの値を読み込むときや書き込むときを指定することができる。このとき選択される範囲は 1 つ目のジョインポイントを含むソースコードが実行される直前から 2 つ目のジョインポイントを含むソースコードが実行される直前までである。

recover アドバイスを追加したい場合は、宣言されたポイントカット・フィールドを @Recover アノテーションで注釈する。そして図 3.2 の様に etype に処理したい例外の型を、advice にアドバイスとして例外処理の内容を記述する。このアドバイスの中では、変数 $1 を block ポイントカット指定子で指定した範囲を含むメソッド sendFile() の第一引数を

あらわす変数として利用している。同様に、$e は投げられる例外自体を指す変数としてアドバイス内で利用可能である。また、recover アドバイスの中で GluonJR.retry() が呼ばれているが、これは block ポイントカットで指定された範囲の先頭に戻って、その範囲の処理を再実行するためのメソッドで、リカバリ処理の記述のために、GluonJ/R が提供する特殊なメソッドである。

図 3.2 のアスペクトを、図 2.2 のプログラムに織り込む (weave する) と、図 3.3 の try-catch 文を使ったプログラムと同等の振る舞いをするプログラムが得られる。

```
class Sender{
  public void sendFile(String host,
      String fileName) throws Exception {
    int n;
    int port = 9000;
    byte[] buff = new byte[1024];

    again:
    try{

      Socket s = new Socket(host, port);
        :
      s.close();

    }catch(IOException e){
      host = getAnotherHost();
      goto again;
    }

    System.out.println(fileName
              + " has been sent to " + host);
  }
}
```

図 3.3: 図 3.2 のアスペクトを図 2.2 のプログラムに織り込んだ場合に得られるバイトコードと同等の振る舞いをするプログラム

ただし、catch 節の中で呼ばれている goto は Java の文法ではない。

3.3　行アノテーション

現実には指定したい範囲の直前や直後に適当なジョインポイントがない場合がある。例えば、図 3.4 にあるように if 文の直後に for 文が書かれてある場合、if 文の直後すなわち for 文の直前には適当なジョインポイントは存在しないので、for 文の前後を範囲とする block ポイントカットはそのままでは定義できない。

```
if(){
    :
}else {
    :
}
for(){
    :
}
    :
```

図 3.4: ポイントカットできない例

このような場合は、GluonJ/R が提供する**行アノテーション**を用いて指定する。行アノテーションとはユーザ定義のジョインポイントであり、メソッド中の特定の行に対してつけられるアノテーションのことを指す。

将来例外を捕まえる範囲として指定されそうな箇所にあらかじめユーザが目印として行アノテーションを記述しておくことで後でジョインポイントとして利用できる。図 3.5 に、上の文の for 文の前後に行アノテーションを付加した例を示す。この場合、block ポイントカットを

```
Pcd.block(Pcd.line("begin"), Pcd.line("end"));
```

のように記述することでこの行アノテーションで囲まれた範囲をポイントカットすることができる。

```
if(){
    :
}else {
    :
}
@Line(begin)
for(){
    :
}
@Line(end)
    :
```

図 3.5: 行アノテーションの記述例

3.4　GluonJ/R の制限

本章では GluonJ/R で例外処理を追加する方法のうち大まかな部分については述べてきた。しかし、細かい部分では疑問が残る部分があるだろう。よって以下では、疑問が残るであろうと思われる点について詳しく説明する。

3.4.1　`block` ポイントカットが選択する範囲

3.2 章で GluonJ/R が提供する範囲をポイントカットするポイントカット指定子 `block` ポイントカットについて述べたが、このポイントカット指定子で指定できる範囲について異なるメソッドに存在するジョインポイントをペアとして範囲を指定することができるか、や、同一メソッド内でも同一ブロック内にないジョインポイントをペアとして指定することができるのかなど不明瞭な点がある。以下ではそれらの点に対して簡単な例を用いて説明する。

ブロックの境界をまたいだ場合

範囲の始点と終点として選択されるジョインポイントが同一メソッド内であればどのような範囲であろうと選択される。図 2.2 に対して図 3.2 の

ように範囲を選択して例外処理を追加することはもちろんのこと、図 3.6 のようなプログラムに対して図 3.7 を織り込むこともできる。織り込んだ際の状態をソースコードで考えると図 3.8 のようなおかしなセマンティクスになっているが、バイトコード上では問題がない。図 3.6 のようなプログラムに対して図 3.7 を織り込んだ際のバイトコードが図 3.9 であるが、これを分かりやすく図にしたものが図 3.10 である。つまり、図 3.8 と同じこのバイトコードを実行するとバイトコードの 2 バイト目を実行する直前から 13 バイト目を実行する直前までの間に例外が起こらなければ `while` ループにあたる 9 バイト目から 24 バイト目を実行する。2 バイト目の直前から 13 バイト目の直前までの間に例外が生じた場合は `catch` 節にあたる 26 バイト目以降を実行し、37 バイト目で 13 バイト目に戻って実行するという動きになる。

```
public class Sample{
    public void m() throws Exception{
        int i = 0;
        a();
        while(i<3){
            c();
            b();
            i++;
        }
    }
}
```

図 3.6: 選択される側のプログラム

同一メソッド内ではどのような範囲でも選択されると述べたが、逆に言うと、同一メソッド内でないと範囲として選択されないということである。

3.4.2　`finally` 節が含まれる場合

バイトコード上の `finally` 節

図 3.11 のような `finally` 節を用いたソースコードをコンパイルすると図 3.12 のようなバイトコードが得られる。このバイトコードを見ると、何も例外が発生しない場合（図 3.12 の 0〜7 バイト目）と `IOException`

```
@Glue
class InsertTryCatch {
@Recover(etype="java.lang.Exception",
                  advice="")
Pointcut p = Pcd.block(
            Pcd.call("Sample#a(..)"),
            Pcd.call("Sample#b(..)"));
}
```

図 3.7: 図 3.6 に織り込むアスペクト

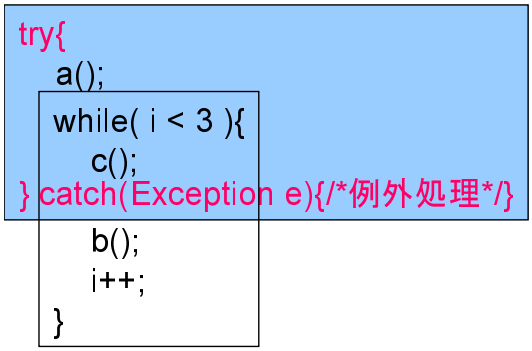

図 3.8: 3.6 の a() から b() に GluonJ/R を用いて例外処理を追加した場合の状態

が発生する場合（11〜17 バイト目）、Exception が発生する場合（23〜31 バイト目）、それら以外の例外が発生する場合（35〜41 バイト目）と例外について場合分けがされていることが分かる。それらの各場合において逐一メソッド b() のバイトコードが展開されているのが分かる。メソッド b() のバイトコード、つまり、finally 節のバイトコードは必ず実行されるためこのようなバイトコードが出来上がるようになっている。要するに、finally 節はそれより上の処理で例外が起ころうが起こらなかろうが、return しようが、メソッドから外に出る前に必ず実行されるコードである。

```
0:   iconst_0
1:   istore_1
2:   aload_0
3:   invokevirtual   #24; //Method a:()V
6:   goto    20
9:   aload_0
10:  invokevirtual   #27; //Method c:()V
13:  aload_0
14:  invokevirtual   #30; //Method b:()V
17:  iinc    1, 1
20:  iload_1
21:  iconst_3
22:  if_icmplt       9
25:  return
26:  astore_2
27:  getstatic       #96;
   //Field java/lang/System.out:Ljava/io/PrintStream;
30:  aload_2
31:  invokevirtual   #101;
   //Method java/lang/Throwable.getMessage:()Ljava/lang/String;
34:  invokevirtual   #103;
   //Method java/io/PrintStream.println:(Ljava/lang/String;)V
37:  goto    13
Exception table:
 from   to  target type
   2    13    26   Class java/lang/Exception
```

図 3.9: 図 3.6 の a() から b() に GluonJ/R を用いて例外処理を追加した場合のバイトコード

```
public void m() throws Exception{
    try{
        a();
    }catch(IOException e){
        c();
    }catch(Exception e){
        d();
    }finally{
        b();
    }
}
```

図 3.11: finally 節を使った例

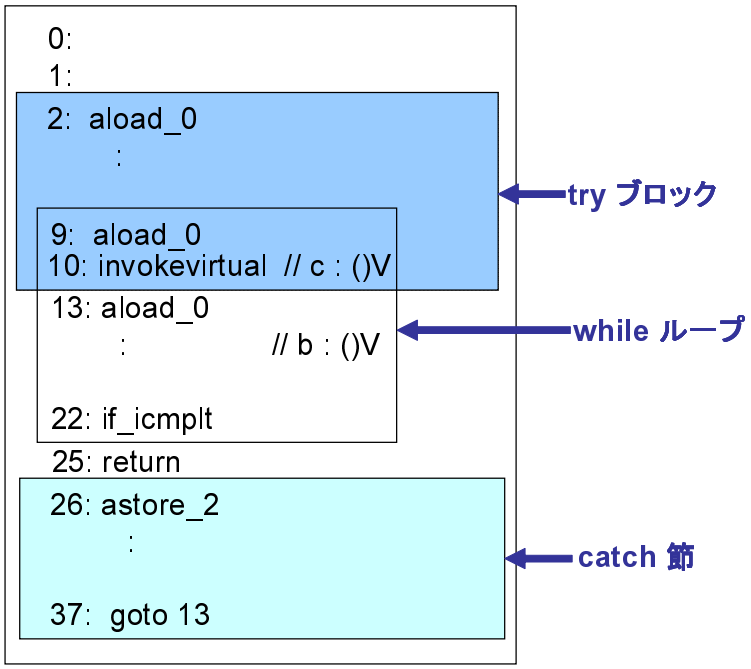

図 3.10: 3.6 の a() から b() に GluonJ/R を用いて例外処理を追加した場合のバイトコードの状態

しかし、GluonJ/R で例外処理を追加するときに、同一メソッド内に finally 節が存在するかについてはチェックしない。図 3.12 を見ても分かるように展開されてしまい、展開されてしまったコードが finally 節であったということは分からなくなってしまっている。よってこのようなソースコードに GluonJ/R を用いて例外処理を追加してしまった場合、本来、必ず実行されると約束されている finally 節が実行されなくなってしまう恐れがある。

例外処理が選択される優先順位

```
try{
    try{
        :
    } catch(IOException e){
        :
    } finally{
        :
    }
} catch(Exception e){
    :
}
```

図 3.13: GluonJ/R で例外処理を追加した範囲に `finally` 節が含まれていた場合

図 3.13 の四角で囲った外側の `try-catch` 文に対応する例外処理を GluonJ/R を用いて追加した場合を考える。このとき内側の `try` ブロックで例外が生じるとソースコードのセマンティクスで考えると発生した例外が `IOException` であろうが、すぐ下の `finally` 節は実行される。しかし、GluonJ/R で実際は追加した外側の例外処理が適用される。つまり、`Exception` 型の例外をハンドリングする `catch` 節を実行する。それにより、本来例外が生じる生じないに限らず実行されるはずの `finally` 節が実行されなくなってしまう。

選択していない範囲をポイントカット

図 3.14 のように `try` ブロックの中で `return` 文等が含まれる場合、それらの文の直前で `finally` 節が展開されているため、選択していないはずの `finally` 節を含んでしまう可能性もある。

アドバイス内で例外を再度投げなおした場合

図 3.15 のように追加したアドバイス内で例外を再度投げなおした場合、`finally` 節が実行されなくなってしまう可能性がある。

3.4.3　同一箇所に複数の例外処理を追加した場合

3.4.2 章で述べたように、GluonJ/R を用いて例外処理を追加した場合、追加した例外処理が検査される優先順位が最も高くなるように実装されている。そのため、`try-catch` 文が直接書かれてあるソースコードに後から GluonJ/R で例外処理を追加すると GluonJ/R で追加した例外処理の方が適用される優先順位は高くなる。同様に、複数の Glue クラスを用いて同一箇所に例外処理を追加すると新しく追加された順に検査される優先順位が高くなるようになっている。

```
public void m()   throws java.lang.Exception;
  Code:
     0:   aload_0
     1:   invokevirtual   #3; //Method a:()V
     4:   aload_0
     5:   invokevirtual   #4; //Method b:()V

   8:   goto    42

     11:  astore_1
     12:  aload_0
     13:  invokevirtual   #6; //Method c:()V
     16:  aload_0
     17:  invokevirtual   #4; //Method b:()V

   20:  goto    42

     23:  astore_1
     24:  aload_0
     25:  invokevirtual   #8; //Method d:()V
     28:  aload_0
     29:  invokevirtual   #4; //Method b:()V

   32:  goto    42

     35:  astore_2
     36:  aload_0
     37:  invokevirtual   #4; //Method b:()V
     40:  aload_2
     41:  athrow

   42:  return
  Exception table:
   from   to  target type
     0     4    11   Class java/io/IOException
     0     4    23   Class java/lang/Exception
     0     4    35   any
    11    16    35   any
    23    28    35   any
    35    36    35   any
```

図 3.12: 図 3.11 をコンパイルして得られるバイトコード

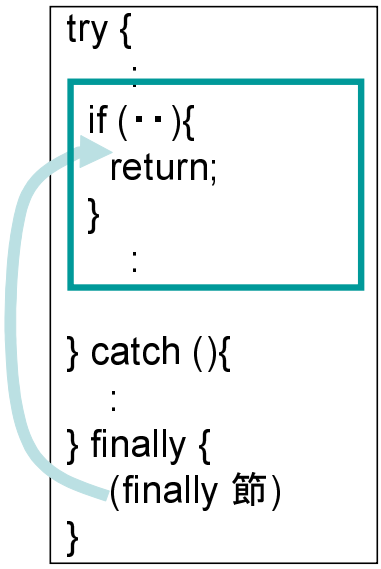

図 3.14: `try` ブロックに展開される `finally` 節

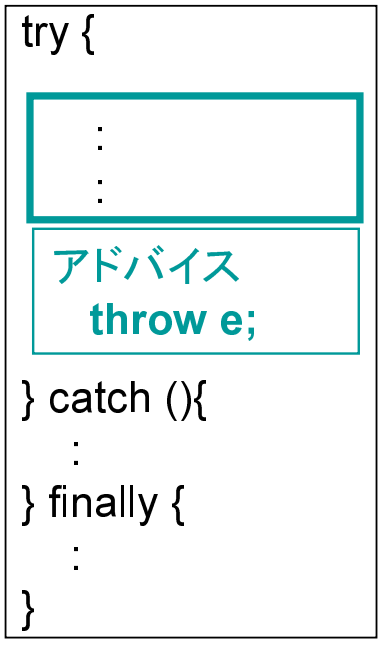

図 3.15: アドバイス内で例外をリスローした場合

第4章　実装

我々は我々の研究室で開発したアスペクト指向システム GluonJ を拡張して GluonJ/R を実装した。`block` ポイントカットで指定された２つのジョインポイントで囲まれた範囲を `try` ブロックとし、アドバイスとして書かれたコードを `catch` 節の中身とした `try-catch` 文を元のプログラムにバイトコード変換で埋め込む。バイトコード変換には Javassist[8] を利用した。

4.1　`block` ポイントカット

GluonJ では、ユーザによって宣言されたポイントカットを見つけるとその宣言されたポイントカットを表現する抽象構文木をポイントカットノードを組み合わせて生成する。ポイントカットノードの種類には、クラス名やメソッド名を表現する文字列をフィールドに持ち、その文字列に一致するメソッド呼び出しをポイントカットする `call` ポイントカット等がある。そして、ソースコード内のジョインポイントにぶつかる度に生成された構文木を巡回し、ポイントカットすべきジョインポイントなのかを判定するという仕組みになっている。抽象構文木を構成するポイントカットノードの種類に、２つのポイントカットのペアをフィールドとして持つ Block ポイントカットノードを追加して `block` ポイントカットを実装している。この２つのフィールドとなるジョインポイントは GluonJ に既にあるポイントカット（`call` ポイントカット等）で表現される。

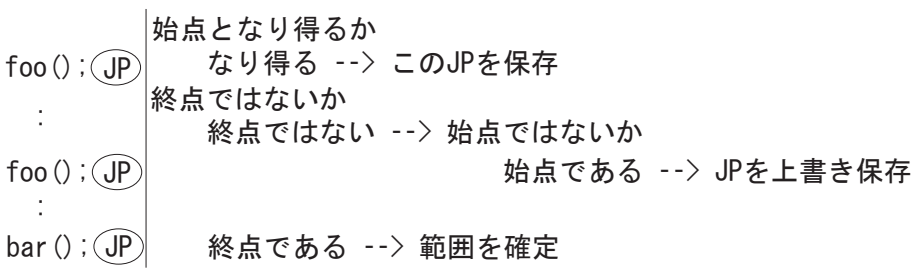

図 4.1: `foo()` から `bar()` までの範囲が選択されるアルゴリズム

次に、block ポイントカットによって選択されるべき範囲を見つけるアルゴリズムについて述べる。このアルゴリズムは始点と終点となり得るジョインポイントが複数存在する場合はそれらのうちで一番近いペア同士が範囲として選択されるように設計されている。図 4.1 に示す様に、まず、ジョインポイントが範囲の始点となり得るかをチェックしていく。始点の候補となるジョインポイント（１番目の foo() の呼び出し）が現れた場合、そのジョインポイントを一時的に保存しておく。始点の候補となるジョインポイントが現れた後は、ジョインポイントを見つけると、保存してある始点と対となる終点のジョインポイントにならないかをチェックする。終点ではないと判定されると、始点のジョインポイントとならないかをチェックする。そこでもし始点となるジョインポイントだと判定された場合は保存しておいたジョインポイントを上書きして保存する。（２番目の foo() の呼び出し）始点の候補が存在している間に、終点のジョインポイント（bar() の呼び出し）が見つかるとその時点で範囲を確定し、始点と終点を対にして保存する。そして、他にもポイントカットすべき範囲がないかをチェックするために、始点となり得るかをチェックする段階から新たな範囲を探していく。つまり、図 4.1 だと二番目の foo() から bar() までが範囲として選択される。

4.2　recover アドバイス

クラスファイルの中には各メソッドの情報が含まれており、そのメソッドの情報の中にはメソッドの属性が含まれている。メソッドの属性の中にはそのメソッドを実装しているバイトコードのほかに Exception Table という表が書かれている。この表に含まれる情報としては、

- 例外ハンドラがアクティブとなるバイトコードの始点と終点
- 例外が生じた場合に実行するバイトコードの先頭
- 例外ハンドラがキャッチする例外のクラス

がある。

まず、recover アドバイスで指定された例外が生じた場合に実行したいコードを、ポイントカットされた範囲の始点と終点が存在するメソッドのバイトコードの末尾に追加する。そして、追加したバイトコードの先頭のインデックスを Exception Table に追加する。次に、block ポイントカットで指定された例外処理を追加したいソースコードの始点と終点の情報をもとに例外ハンドラがアクティブとなるバイトコードの始点と終点を求める。そして、etype で指定された処理したい例外の型と共に Exception

Table に追加する。例外が発生した場合、当てはまるかどうかは表の順番通りに検査されるため、アスペクトで追加された例外処理の優先順位が高くなるように表の最初に追加している。

4.3　特殊メソッド GluonJR.retry()

3.2 章でも述べたように、GluonJR.retry() はリカバリ処理が容易に記述できるアドバイス内で利用可能な特殊メソッドである。この static メソッドをコンパイルすると invokestatic という命令長が 3 バイトのバイトコードに変換される。この命令を同じ 3 バイトの命令長の goto 命令に置換することで block ポイントカットで指定した範囲の先頭に戻って再試行することを実現している。

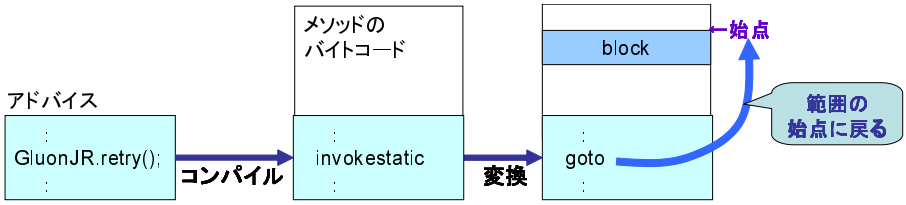

図 4.2: GluonJR.retry() の実現方法

4.4　アドバイスの最後で呼ばれるメソッド

アドバイスの最後には自動的に GluonJR.doNext() メソッドを追加している。このメソッドは GluonJR.retry() と同じく、static メソッドを goto 命令に変換することで実現している。このメソッドはソースコードに try-catch 文を追加したときと同様、catch 節を実行し終わった後でその下のコードを実行するようにするために必要となる。つまり、goto の飛び先は catch 節のすぐ下の、block ポイントカットで指定した終点に相当するコードである。このように goto で飛ぶ先を明確に示さないとバイトコード検査器によって VerifyError[1]が投げられてしまう。

4.5　バイトコード変換

GluonJ/R はバイトコード変換によって try-catch 文を挿入することで recover アドバイスを実現するが、単純な変換ではうまくいかない。3

[1]バイトコードが不正であると検証された場合に投げられるエラー

章で示した図 3.2 の FileSenderRecovery アスペクトでは、call ポイントカットによって、Socket オブジェクトの生成時を表すジョインポイントを選択していた。これはプログラムの次の行に該当する。

```
Socket s = new Socket(host, port);
```

プログラムを Java コンパイラでコンパイルして得られるバイトコードのうち、上の行に対応するバイトコードは以下のようになる。

```
new Socket
dup
aload_1
iload 4
invokespecial Socket()
astore 5
   :
```

aload_1 と iload 4 は、コンストラクタの引数をスタックに積むための命令である。コンストラクタの呼び出しは invokespecial 命令である。このバイトコードを実行している間の JVM のスタックの状態は図 4.3 のように遷移する。

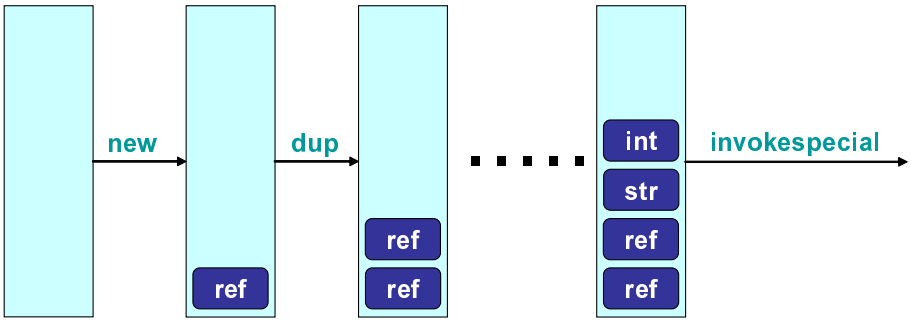

図 4.3: JVM のスタック状態遷移図

AspectJ に代表される通常のアスペクト指向システムでは、一般に、ジョインポイントを invokespecial 命令の実行時と解釈する。ところが、この命令を try ブロックの始点と考えると、retry() 命令の実現が困難になる。単純に goto 命令などで、invokespecial 命令から実行を再開しようとすると、コンストラクタの引数がスタックに積まれていない状態で、コンストラクタを呼ぼうとしてしまう。これは不正な実行なので、バイトコードを Java 仮想機械にロードする段階で、バイトコード検査器が不正なコードとしてロードを拒否し、VerifyError が投げられてしまう。

この問題を回避するため、GluonJ/R では、ジョインポイントに該当するソースプログラムの行を構成するバイトコード列の先頭を、`block` ポイントカットによって選択される範囲の境界として用いる。先の例では、先頭の `new` 命令を境界として用いる。これによって、Java 仮想機械のスタックの状態が常に整合性に保った状態であることを保障する。一般的な Java コンパイラが生成するバイトコードでは、行の境界では必ずスタックが空になるので、整合性が保障できる。なお、実現にあたっては、Java クラスファイル内に記録されているソースプログラムの行と各バイトコード命令との対応関係を表す情報を利用している。

4.6　行アノテーション

行アノテーションに関しては、ソースプログラムを読み込み、行アノテーションを空の static メソッド呼び出しに置換するプリプロセッサによって実現している。呼ばれる static メソッドとして、引数も返り値もないメソッドを選べば、メソッドが呼ばれた場所（`call` ポイントカット）で自由にジョインポイントを指定することができる。また、メソッドの中身は空のためプログラムの実行には影響を与えない。

プリプロセッサは、ソースコード中の

```
@Line(begin)
```

という文字列を

```
LineAnnotation.begin();
```

という static メソッド呼び出しに置換する。次に、`LineAnnotation` クラスに

```
public static void begin(){}
```

というメソッドを追加した後、`LineAnnotation` クラスを再コンパイルするという実装方法である。

第5章　実験

GluonJ バージョン 1.3 を拡張して開発した GluonJ/R で例外処理をアスペクトとして追加した場合のオーバーヘッドを測定するのを目的として以下のような実験を行った。実験環境は、CPU は Intel® Pentium® 4 CPU 2.8GHz、メモリは 1 GB、OS は Microsoft Windows XP Professional Service Pack 2、JVM のバージョンは 1.5.0_06 である。

5.1　try-catch 文との実行速度の比較

図 5.1 のメソッド m() を 10 億回呼んで時間を測定するというマイクロベンチマークを走らせて、ソースコードに直接 try-catch 文を書いたものと GluonJ/R で例外処理を追加したものの実行時間を比較してみた。

```
public class Test {
  public void m() throws Exception{
    a();
    b();
  }

  public void a() throws Exception{}
  public void b() {}

}
```

図 5.1: マイクロベンチマーク

このベンチマークを走らせたものと、メソッド m() の中身を

```
try{
  a();
} catch(Exception e){}
b();
```

のように try-catch 文を追加して書き換えたもの、GluonJ/R で書いた図 5.2 のようなアスペクトを織り込んだ場合の実行時間を比較してみた。これにより得られた結果は表 5.1 の通りである。

```
@Glue
class InsertTryCatch {
  @Recover(etype = "java.lang.Exception",
           advice = "")
  Pointcut p = Pcd.block(
      Pcd.call("test.Test#a(..)"),
      Pcd.call("test.Test#b(..)"));
}
```

図 5.2: 織り込んだアスペクト

	実行時間 (秒)
元のプログラム（例外処理なし）	2.638
try-catch 文を追加	17.064
GluonJ/R で追加	17.046

表 5.1: try-catch 文と GluonJ/R の実行時間の比較

5.1.1　インタプリタのみで実行

Xint オプションを付けて Java VM を実行すると、動的コンパイルを行わずにインタプリタのみでプログラムを実行させることができる。これを用いてもう一度、図 5.1 のメソッド m() を 10 億回呼んで時間を測定してみると表 5.2 のような結果が得られた。

	実行時間 (秒)
元のプログラム（例外処理なし）	229.839
try-catch 文を追加	237.515
GluonJ/R で追加	242.324

表 5.2: インタプリタのみで実行した場合の実行時間の比較

これをみて分かるように、例外処理の記述がない元のプログラムと try-catch を直接ソースコードに追加したプログラムと GluonJ/R で例外処理を追加したプログラムの実行時間にさほど差がない。つまり、5.1 章の性能評価のための実験において、例外処理を追加したプログラムが例外処理の記述がないプログラムに比べてはるかに遅かったのは動的コンパイルによるインライン展開が成されたせいだというとこが分かる。

5.1.2　ポリモルフィズムを用いた例

5.1.1 の原因として挙げられたインライン展開を避けるため、継承を用いた図 5.3、5.4、5.5 のようなコードを動かしてみると表 5.3 のような結果が得られた。比較したものは、図 5.3 に何も追加せずそのまま実行したものと、図 5.3 のメソッド m() を

```
public void m(A a) throws Exception{
  try{
    a.a();
  } catch(Exception e){}
    a.b();
}
```

のように try-catch 文を直接ソースコードに書き加えて追加したものと図 5.3 に対して図 5.6 を実行前に織り込んだものとロード時に織り込んだものである。

	実行時間 (秒)
元のプログラム（例外処理なし）	24.578
try-catch 文を追加	55.758
GluonJ/R で追加（実行前織り込み）	53.312
GluonJ/R で追加（ロード時織り込み）	51.543

表 5.3: try-catch 文と GluonJ/R（実行前織り込み・ロード時織り込み）の実行時間の比較

これの結果を見てわかるように、ソースコードに直接 try-catch 文を記述するより GluonJ/R を用いて例外処理を追加した方が実行時間は若干短い。

その原因を探るために、try-catch 文を直接ソースコードに追加したものと GluonJ/R を用いて例外処理を追加したもののメソッド m() のバ

イトコードを比較してみた。それぞれのバイトコードは図 5.7 と 5.8 の通りである。

```
   0: aload_0
   1: invokevirtual #20; //Method a:()V
   4: goto 8
   7: astore_1
   8: aload_0
   9: invokevirtual #23; //Method b:()V
   12: return
  Exception table:
   from   to  target type
     0     4     7   Class java/lang/Exception
```

図 5.7: try-catch 文を直接ソースコードに追加した場合のメソッド m() を実装するバイトコード

```
   0: aload_0
   1: invokevirtual #20; //Method a:()V
   4: aload_0
   5: invokevirtual #23; //Method b:()V
   8: return
   9: astore_1
   10: goto 4
  Exception table:
   from   to  target type
     0     4     9   Class java/lang/Exception
```

図 5.8: GluonJ/R を用いて例外処理を追加した場合のメソッド m() を実装するバイトコード

これらを見て分かるように、例外が生じない場合、GluonJ/R で例外処理を追加した場合では 0 から 8 番目のバイトコードを実行するのに対してソースコードに直接 try-catch 文を追加した場合では 0,1 番目のバイト

コードを実行した後、goto 命令で 8 に飛び、12 番目まで実行する。つまり GluonJ/R で追加した方が goto 命令一つ分実行せずに済んでいることが分かる。これらが GluonJ/R を用いたときの方が実行時間が短くなった原因と考えられる。

5.2　行アノテーションの性能実験

4.6 章で行アノテーションはプログラムの実行には影響を与えないと述べたが、実行速度にはどのくらい影響するかについて調べてみた。

図 5.1 のプログラムに対して、メソッド `m()` の中身を

```
@Line(begin)
a();
@Line(end)
b();
```

のように行アノテーションを追加したものと図 5.9 のようなアスペクトを織り込んで、行アノテーションを始点・終点とする範囲に例外処理を追加したものの実行速度を測定してみた。得られた結果は表 5.4 の通りである。

	実行時間 (秒)
行アノテーション追加（例外処理なし）	2.652
行アノテーション追加（例外処理追加）	17.121

表 5.4: 行アノテーション利用時の実行時間の比較

表 5.1 と表 5.4 の結果を比較すると、例外処理を追加しない場合の行アノテーションのオーバーヘッドは 0.02 秒以内で、例外処理を追加した場合で 0.08 秒程度遅くなっただけであることが分かる。これは JIT により最適化が行われたためと考えられる。行アノテーション、つまり変換後の static メソッドがプログラム全体において占める割合がこのように高い場合でもこの程度のオーバーヘッドしかないので、実際のプログラムでは行アノテーションのオーバーヘッドはほぼないと考えられる。

```
package test;

public class TestA {
    public void m(A a) throws Exception{
        a.a();
        a.b();
    }

    public static void main(String[] args)
                                throws Exception{
        int n = 1000000000;
        TestA t = new TestA();
        A a = new SubA();
        SubA sa = new SubA();

        long start = System.nanoTime();
        for (int i = 0; i < n; i++){
            t.m(a);
            t.m(sa);
        }
        long end = System.nanoTime();
        System.out.println(end - start + "ns.");
    }
}
```

図 5.3: TestA.java

```
package test;

public class A {
    public void a() throws Exception{
        System.out.println("execute : A.a()");
    }
    public void b(){}
}
```

図 5.4: A.java

```
package test;

public class SubA extends A{
    public void a() throws Exception{}
}
```

図 5.5: SubA.java

```
@Glue
class InsertTryCatch {
@Recover(etype="java.lang.Exception",
                 advice="")
Pointcut p = Pcd.block(
            Pcd.call("test.A#a(..)"),
            Pcd.call("test.A#b(..)"));
}
```

図 5.6: 図 5.3 に織り込んだアスペクト

```
@Glue
class InsertTryCatch {
  @Recover(etype = "java.lang.Exception",
           advice = "")
  Pointcut p = Pcd.block(Pcd.line("begin"),
                         Pcd.line("end"));
}
```

図 5.9: 行アノテーションを範囲の始点・終点として例外処理を追加するアスペクト

第6章　関連研究

6.1　AspectJ を用いた例外処理の分離

AspectJ を利用して、プログラム中の例外処理の分離を試みた研究がいくつか提案されている [13][10]。特に Lippert らは、既存のソフトウェア JWAM[4] 中に記述されている例外処理を AspectJ 0.4 で分離することを試みた。JWAM 内に記述されている例外処理はソフトウェア全体に散らばっており、それらの例外処理のコードは互いに似ている。Lippert らは散らばっているこれらの例外処理を、AspectJ を利用して一箇所にまとめることにより、ソフトウェア全体のコードサイズを減らすことができたと報告した。しかし、既存の AspectJ では、`block` ポイントカットのように任意の範囲をポイントカットし、その範囲の例外処理を追加することはできない。また、例外処理を分離して記述することは可能でも、GluonJ/R のようなリカバリー処理 (retry) を実現することは困難である。このため、本論文の 2 章で取り上げた実験プログラムの例外処理を AspectJ で記述するのは適切ではない。

6.1.1　例外処理に関連する言語機構

AspectJ は、バージョン 0.6 以降から、例外処理に関連する言語機構が 2 つ提供された。それらは `handler` ポイントカット指定子と `after throwing` アドバイスである。それらについて説明すると共に、例外処理を記述するのに不十分な点について述べる。

`handler` ポイントカット

例外ハンドラの実行時、つまり `catch` 節の実行時のみをジョインポイントとして選択する。このため、今回の実験プログラムのように予め `try-catch` 文が使われていない場合、このポイントカット指定子は有効ではない。

after throwing アドバイス

選択されたジョインポイントが例外を投げて異常終了したときに実行される。ただし、このアドバイスは catch 節のように例外を捕まえるわけではなく、暗黙のうちに実行され、例外によるメソッドの異常終了の連鎖を止めるわけではない。それゆえ、after throwing を利用してリカバリ処理を実装することは困難である。

6.2　Eiffel[15][2][7][6] や Ruby[5][9] の retry 機構

オブジェクト指向言語である Eiffel や Ruby には GluonJ/R が提供する特殊メソッド GluonJR.retry() と同様の機構が存在する。

Ruby では基本的な例外処理の記述が他の言語とかなり似ていて、例外処理を含む begin と end で囲まれたブロックでは Python の except、Java や C++の catch の代わりに rescue というキーワードが使うことができる。

例外自体を使って何か処理をしたい場合、==> を用いることで局所変数として例外を抽出することができ、例外を投げるには Python と同じように raise というキーワードを使用する。そして、ensure というキーワードを使う”finally”と同等のメカニズムも存在する。

しかし、Python や Java にある assert のようなものはない。その代わりに、シンタックスシュガーとして、 再試行 (retry) がある。retry を使用すると、ブロック内で例外が発生した場合、実行ポイントがブロックの最初に戻るので、while-not-succeeded ループを用いるときと基本的に同じである。図 6.1 を実行した結果は図 6.2 このより、再試行すると実行ポイントがブロックの最初に戻っていることが分かる。

このように Eiffel や Ruby には再試行を実現する機構がもともと組み込まれているが Java には存在しない。GluonJ/R ではバイトコード変換により Java で再試行を実現する機構を実現している。

6.3　ループのためのジョインポイント

3.3 章で行アノテーションを用いてソースコードの任意の範囲をポイントカットする方法について述べたが、ループをポイントカットする研究は存在している。[11] この研究では、バイトコードからループを見つけ出し、ジョインポイントを提供する。しかしこのアプローチでは、ポイントカットできないループも数多く存在している。それゆえ、ループについても、block ポイントカットと行アノテーションは有用であると考えられる。

```
count = 0
begin
  puts "at beginning"
  if count == 0
    count = count + 1
    raise "First time through"
  else
    puts "second time through"
  end
rescue Exception
  puts "In handler"
  retry
end
```

図 6.1: Ruby の retry 機構の使用例

```
$ Ruby Retry.rb
at beginning
In handler
at beginning
second time through
```

図 6.2: 図 6.1 の実行結果

6.4　例外処理記述の悪い例

例外処理として悪い例がいくつか挙げられている。[14] ログだけを出力したあとすぐに発生した例外をそのまま再度投げ返したり、null を返すといった例外処理は例外を処理してはいないためそれ自体意味のないことになってしまう。独自の例外クラスを定義して発生した例外をラップして投げたりして発生した例外に対して適切な処理をすることはとても重要なことである。GluonJ/R の block ポイントカット指定子と recover アドバイス、更には retry() メソッドを用いることで適切できめ細やかな例外処理を記述することが可能になる。

第7章　まとめ

本稿において我々は例外処理をアドバイスとして扱えるようにしたアスペクト指向システム GluonJ/R を提案した。プログラムロジックから例外処理を分離して記述でき、例外処理の中で再試行できるように書ける事を示した。

GluonJ/R はバイトコード変換で例外処理をプログラムロジックに埋め込む。その時、バイトコード上のジョインポイントを選択するのではなくクラスファイルに含まれる行番号を利用して、ソースコード中のジョインポイントを選択する。このようにすることで、`VerifyError` が起こるのを回避することが出来ることを示した。また、例外処理をアスペクトとして分離して記述してもソースコードに直接 `try-catch` 文を記述した場合とほぼ実行時間は変わらないことが分かった。

今後は、既存のアプリケーションを GluonJ/R を用いて例外処理を記述した場合、うまく記述できるか、コードサイズをどのくらい減らすことができるかについて検証したいと考えている。

発表リスト

発表論文

- 熊原奈津子・光来健一・千葉滋：”例外処理のためのアスペクト指向言語”, 第 62 回情報処理学会・プログラミング研究会 (PRO), 沖縄県 那覇市, 2006 年 1 月 19 日.
- 熊原奈津子・石川零・西澤無我・光来健一・千葉滋：”Recovery アドバイスをもつアスペクト指向システム”, 第 9 回 プログラミングおよび応用のシステムに関するワークショップ (SPA 2006), 栃木県 那須塩原市 塩原温泉, 2006 年 3 月 7 日.

その他の発表

- 第 10 回プログラミングおよび応用のシステムに関するワークショップ (SPA X), 新潟県 南魚沼郡 越後湯沢温泉, 2006 年 8 月 28〜30 日.(ポスター発表)
- 第 4 回 SPA サマーワークショップ, 山梨県 笛吹市 石和温泉, 2005 年 8 月 23 日.(ポスター発表)
- International Conference on Aspect-Oriented Software Development (AOSD 2006), Bonn Germany, March 20-24, 2006.(ポスター発表)
- 第 8 回プログラミングおよび応用システムに関するワークショップ (SPA2005), 群馬県 渋川市 伊香保温泉, 2005 年 3 月 7〜9 日.(ポスター発表)

参考文献

[1] : AspectJ Web Site, http://www.eclipse.org/aspectj/.

[2] : Eiffel Software Web Site, http://www.eiffel.com/.

[3] : GluonJ Web Site, http://www.csg.is.titech.ac.jp/projects/gluonj/.

[4] : JWAM framework Web Site, http://www.jwam.de/.

[5] : Ruby Web Site, http://www.ruby-lang.org/.

[6] : Standard ECMA-367 Web Site, http://www.ecma-international.org/publications/standards/Ecma-367.htm.

[7] : the NICE (the Nonprofit International Consortium for Eiffel) Web Site, http://www.eiffel-nice.org/.

[8] Chiba, S.: Load-time Structural Reflection in Java, *Proceedings of the European Conference on Object-Oriented Programming*, pp. 313–336 (2000).

[9] Eckel, B.: Computing Thoughts Ruby, PHP and a Conference, http://www.artima.com/weblogs/viewpost.jsp?thread=146091.

[10] F. Filho, C. Rubira, A. G.: A Quantitative Study on the Aspectization of Exception Handling, *Workshop on Exception Handling in OO Systems Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 418–427 (2005).

[11] Harbulot, B. and Gurd, J. R.: A join point for loops in AspectJ, *Proceedings of the 5th international conference on Aspect-oriented software development (AOSD '06)*, pp. 63–74 (2006).

[12] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. G.: An Overview of AspectJ, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pp. 327–353 (2001).

[13] Lippert, M. and Lopes, C. V.: A study on exception detection and handling using aspect-oriented programming, *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 418–427 (2000).

[14] McCune, T.: Exception-Handling Antipatterns, `http://today.java.net/pub/a/today/2006/04/06/exception-handling-antipat%terns.html`.

[15] Meyer, B.: *Eiffel: the Language*, Object-Oriented, Prentice-Hall, Upper Saddle River, NJ, USA (1992).

[16] 千葉滋: アスペクト指向入門, 技術評論社 (2005).