Java への限定された継続の実装と、その応用[1]

Sun Microsystems, Inc. Web Technologies and Standards
川口耕介[2] kk@kohsuke.org

概要

「継続」は、関数型言語で良く見られる一風変わった言語機能である。継続には色々と面白い用途があるが、残念なことに Java などの手続き型汎用言語では使うことができなかった。本稿では、まず、Java 言語上のライブラリとして継続をエミュレートし、アプリケーションから利用できるようにする javaflow プロジェクトを紹介する。次に、このライブラリのテストベッドとして開発している dalma プロジェクトを紹介する。

継続と手続き型言語

多くの関数型言語には、「継続」という機能が備わっている。端的には、これは現在のコールスタックの内容(変数の値、関数の戻り先)を値として捕捉し、そこから後で実行を再開するための機能である。継続を使うと、バックトラッキングを含む探索問題を綺麗に記述したり、実行を仮想的に並列化したりといった、特有の様々な実装技法が可能になる面白い言語プリミティブである。多くの関数型言語で利用可能なこの機能は、残念なことに、Java や C#といったほとんどの手続き型言語には存在していない。

ところが、最近、上で見たような継続の自然な応用の幾つかが、その有用性を手続き型言語において見直されてきている。例えば、C# 2.0 では、ジェネレータを記述する為の「yield 値」というキーワードが追加された。例えば、次のようなプログラムは次のような結果を出力する。ここで行われているのは、yield 毎にコールスタックを束縛してそれを再実行している事に他ならない[3]。

```
public class Foo : Ienumerable<int> {
  public IEnumerator<int> GetEnumerator() {
      for( int i=0; i<3; i++ )
          yield i*i;
  }
  public static void Main(string[] args) {
```

[1] Implementing a limited form of continuation in Java and its applications
[2] Kohsuke Kawaguchi
[3] 但し、C#では実装方法の規定はなく、Microsoft のコンパイラはこれを別な方法で実装している

```
        foreach(int x in new Foo())
            Console.WriteLine(x);
    }
```

```
1
4
9
```

もう一つの例としては、Service Oriented Architecture (SOA)に象徴される、メッセージの非同期通信を前提としたソフトウェアアーキテクチャの台頭がある。SOA では、企業システム間の連携といったような粒度の粗いシステム結合技術で、こういった環境では、例えば注文書を送り、見積もりをもらうまでに2営業日かかる、などといったようなメッセージ交換が頻繁に起こると考えられている。このような環境化では、注文書を送ってから見積もりをもらうまでの間に、注文に関する状態情報をどこかへ記録しなくてはならない。この時、継続を使ってコールスタックを捕捉してそれを永続化すれば、イベント駆動式にプログラムを記述するより状態の保存が透過的に実行できるようになる。

更にもう一つの例としては、Amazon.com など、オンラインでの商品購入でよく見られる複数ウェブページをまたがった情報入力があげられる。この例では、HTTP は本質的に状態を持たないプロトコルなので、ユーザーがあるページをリクエストしてから次のページをリクエストするまで、そのユーザーに関する状態情報がどこかに保存されねばならない。また、HTTP では往々にして非常に多くの同時利用ユーザーをサポートせねばならない。継続を使うと、計算機資源を効率よく利用して、これを非常に簡単に実装することができる。

遺跡発掘：エージェント技術

現在はさほどの注目を浴びていないものの、もう一つ継続が有用な例として、エージェント技術が挙げられる。エージェント技術では、あるマシンで実行中のプログラム(エージェント)の実行状態を別なマシンへ転送する、という事が行われる。この「実行状態の転送」の実現に、継続をライブラリとして用いるという実装方法が見られた。エージェント技術自体はもはやそれほど注目を浴びていないが、この時の研究のひとつに BRAKES project[4]がある。

C++など旧来の手続き型言語では、継続をライブラリとして綺麗に実装する、というのは大変に難しかった。これは、実行時にコールスタックの型の情報やローカル変数の型の情報が存在しない為、これらを利用するのが難しい為である。また、C++ではメモリ管理にも問題があった。ところが、Java や C#などの中間言語を利用し、実行時に型情報がプログラムからアクセス可能な環境においては、継続として捕捉しなくてはならないローカル変数やコールスタックの情報にアクセスするこ

[4] http://www.cs.kuleuven.ac.be/~eddy/BRAKES/brakes.html

とが比較的容易になり、ライブラリとしての実装が現実的になってきた。BRAKES project のようなアプローチが可能になった背景には、このようなプログラミング言語の進化が挙げられる。

BRAKES project はその後開発が放棄された為、Apache Jakarta Commons で同等の実装を Apache Software License で行おうという「javaflow プロジェクト」が立ち上がった。筆者はこのプロジェクトのコミッターの一人である。

継続の Java での実装

Java においては、継続は以下の2つの情報からなる。1つは、各スタックフレーム上のローカル変数及びオペランドスタックの内容であり、もう1つは関数呼び出しの戻り先である。javaflow は、この内容を捕捉するためにアプリケーションのバイトコードを後処理する。

関数が使用するローカル変数及びその型は、バイトコードを検査することで静的に決定できる。同様の手法で、任意の時点でのオペランドスタックの型も静的に決定できる[5]。Java では、関数は自分自身のローカル変数及びオペランドスタックにしかアクセスできず、呼び出し元のコールスタックには触ることができない。このような制約の元でコールスタック履歴を捕捉するために、関数は、自身のコールスタックを記録した後、呼び出し元関数に戻る。呼び出し元関数では、関数呼び出し(invokeXXX バイトコード)にそれぞれ通し番号を振っておき、実行が関数から復帰した時点で、その関数呼び出し位置に対応する通し番号を記憶する。これに加えて、通常の関数呼び出しからの復帰と、スタック捕捉中の復帰を区別する為に、スレッド毎に現在スタック捕捉中か否かを記憶するフラグが必要である。

これを繰り返すことによって、コールスタックを破壊しながら(すなわち関数呼び出しから戻りながら)全てのコールスタックの内容及び戻り先を記憶していくことができる。この繰り返しを停止する為に、継続の「根元」にも、javaflow が必要である。

例えば、以下のようなコードがあったとする。

```
class Foo implements Runnable {
    static void main() {
      Continuation c = Continuation.start(this);
      Continuation.resume(c);
  }

  void run() {
      for( int i=0; i<10; i++ ) {
          m2(i);
      }
```

[5] これらの計算は基本的にバイトコード型安全性検証を行うのと同様の計算である

```
    }

    void m2(int i) {
        System.out.println(i);
        Continuation.suspend();
    }
}
```

Continuation.start メソッドは継続捕捉の基点となる「根元」であり、そういった処理を除けば、単に run メソッドを呼び出す。

Continuation.suspend()も javaflow で実装され、この内部では単に「スタック捕捉中フラグ」が on になる。suspend()が復帰すると、バイトコード後処理によって挿入されたコードがこのフラグをチェックし、on になっているので、関数呼び出し位置（Continuation.suspend()の直後）、ローカル変数（i 及び this）とオペランドスタック（なし）を記録してから復帰する。呼び出し元関数 run では、同様に「m2(i);」の後に挿入されたコードがフラグをチェックし、この関数呼び出し位置（m2(i)の直後）を記録した後、ローカル変数（i 及び this）とオペラウンドスタック（なし）を記録してから復帰する。

この繰り返しは、呼び出しが javaflow の根元関数 Continuation.start に到達するまで続き、その時点で捕捉された継続が Continuation オブジェクトとして Continuation.start メソッドから得られる。

Continuation.resume()が呼び出されると、「スタック復帰中」フラグが on になり、先ほどとは逆順にローカル変数・オペランドスタックの復元が行われる。バイトコード後処理が施された関数の先頭では、まずスタック復帰中フラグをチェックし、on であれば、まず、関数呼び出し位置をチェックする。これによって実行位置は m2(i)であったとわかるので、これから復元すべきローカル変数の型・サイズ、オペランドスタックの型・サイズがわかる。これらの情報を復元した後、関数呼び出し m2(i)に必要なオペランドスタックを準備する。m2 の呼び出し後、m2 内部で引数などは改めて復元されるので、この関数を呼びだしの為に正確な i の値を復元する必要はなく、単に 0 や null をセットすればよい。

このようにして実行が m2()に入ると、同様にして m2 に相当するスタックフレームの復元が行われる。この繰り返しは実行が Continuation.resume()に到達するまで行われ、この内部で「スタック復帰中」フラグが off になり、改めて呼び出しが復帰する。この時点から実行は通常通り行われるようになる。

結果として、アプリケーションプログラマにとっては、Continuation.start()によって実行が開始され、Continuation.suspend()で一旦実行が停止して Continuation.start()が復帰し、次に

Continuation.resume()呼び出しが起こった後、実行が Continuaion.suspend()から再開されるかのように見える。このプログラミングパラダイムは、協調的マルチタスキングとスケジューラにも似ている。

バイトコード後処理はバイトコードレベルで行われるが、説明のために以下に擬似 Java ソースコードで表示する。

```
class Foo {
  void run() {
      Stack s = Stack.currentStack();
      if(s.isRestoring) {
          switch(s.popRetAdrs()) {
          case 1:
              i = s.popInt();      // ローカル変数とスタックを復元
              // m2(i)呼び出しの為に暗黙の「this」オブジェクトをオペランドスタックにつむ
              push s.popTarget();
              // m2(i)のために 0 をオペランドスタックにつむ
              push 0;
              goto ADR1;
          // もし他に関数呼び出しがあれば類似の case が続く
          }
      }

      for( int i=0; i<a; i++ ) {
          push this;
          push i;
      ADR1:
          invoke m2;        // m2(i);呼び出し
          if(s.isCapturing) {
              s.pushInt(i); // ローカル変数とスタックをキャプチャ
              s.pushRetAdrs(1);
              return;
          }
      }
  }
```

```
    ...
}
```

備考としては、Java 言語ではオブジェクトを初期化する際に、一時的にオペランドスタックに「未初期化オブジェクト」という特殊なオブジェクトが詰まれる事がある。例えば、以下のようなメソッド呼び出しを javac でコンパイルすると、file.getPath()の呼び出し中には未初期化の String オブジェクトがオペランドスタックに残るバイトコードが生成される。

```
new String(file.getPath());
```

未初期化オブジェクトの扱いには強い制限があるため、file.getPath()の中でスタックの捕捉が始まった場合、オペランドスタックを保存することができない。この問題を避ける為に、未初期化オブジェクトができるだけオペランドスタックに残らないように、一時変数を導入して、これを以下のように変換する処理が必要になる。

```
String tmp = file.getPath();
new String(tmp);
```

javaflow の実装方式の問題点（API 形態の違い）

javaflow を実際に実装して利用してみた結果、このようにライブラリとして実装された継続は、関数型言語に組み込みのそれに比べて幾つかの制約があることがわかった。

一つ目は、スタックの捕捉と復元を正しく処理する為に、コールスタックの根元に javaflow がいなくてはならないということである。すなわち、アプリケーションコードは例えば、次のように、明示的に「継続を使う世界への入り口」が書かれなくてはならない。Scheme のように元々言語に継続が備わっている言語にはこのような制約はない。

```
static void main(String[] args) {
  … // アプリケーションの処理
  Continuation.start(new Runnable() {
      public void run() {
          … // 継続を利用するプログラム
      }
  });
}
```

実際に javaflow を実装して、またそれを利用しての感触は、現在 Java において継続の利用が試みられているような分野（ワークフローエンジン、エージェント、ウェブサーバ）においては、これはそれ程大きな問題ではないようである。1つには、これらの用途では、ワークフローエンジンなどのコンテナ側が、継続を利用するアプリケーションコードをホストする、という形態で行われる。このような形態では、いずれにせよアプリケーションコードはコンテナから呼び出されるという形になるので、Continuation.start()呼び出しはコンテナ側に隠すことができ、実際に継続を利用したいアプ

リケーションには複雑さが漏洩しないからである。また、スキームの call/cc とは異なり、これらの環境では継続を実行するタイミングはアプリケーションではなくコンテナが制御する（例えば、エージェントシステムでは、コンテナがコードの移動後に再実行を開始する）ので、いずれにせよコンテナ側に継続オブジェクトは処理するコードが必要だからである。

一方、バックトラックを使った検索アルゴリズムの記述などの用途においては、このような制約は無視できないであろう。

なお、上の例では、継続をオブジェクトとして扱う呼び出し側と、継続が利用可能な環境下で実行されるプログラム側が分かれており、後者では Continuation オブジェクトに触っていない。こうではなく、Scheme の call/cc のように（もしくは setjmp/longjmp のように）捕捉した継続に即座にアクセスしたり、そこへプログラム中から復帰したりするためには、この API を少しラップすればよい。

javaflow の実装方式の問題点（バイトコード後処理）

もう1つの問題は、バイトコードの後処理に関する問題である。まず、後処理をいつ行うかという問題がある。javaflow では、開発時に（コンパイルの直後に）実施する方式と、実行時にクラスがロードされる時点で動的に後処理を行う2つの方法をサポートしている。前者は、ビルド手順の複雑化を招き、後者はアプリケーションの実行環境を複雑にする。

コンテナ環境であれば、コンテナがアプリケーションのコードをロードするので、後者の方法を使って複雑性をアプリケーションから覆い隠すことができる。だが、探索問題を綺麗に記述したい、というような場合には、この複雑さは覆い隠すことができない。

バイトコードの後処理による実装には他の問題もある。javaflow の現在の方式では、コールスタック中の全てのコードが後処理されているかどうかを正しく検出することができない。また、後処理されたコードとされていないコードが入り混じっている場合には、プログラムが奇妙な動作を起こしてしまい、ライブラリからはユーザーのエラーを指摘できない。筆者も開発中に何度もこの問題に遭遇し、経験的にも、これは生産性を損なう大きな問題である。

この制約は、Java からはコールスタック各フレーム上の「this」オブジェクト（またはそのクラス）にアクセスできないことによる。ただ、この機能は JVM には備わっていて公開されていないだけなので、今後の Java 言語の発展によっては利用可能になる可能性もある。

javaflow の実装方式の問題点（コンストラクタ）

JVM にはコンストラクタメソッドの呼び出しに対して強い制限がある。具体的には、未初期化オブジェクトに対してしか呼び出しをすることができない。ところが、実装方法の説明で触れたことにも関連するが、未初期化オブジェクトは継続として保存することができない。従って、コンストラクタ呼び出しを含むようなコールスタックは、継続として保存することができない。

この制約は、JVM が型安全を保障する為に必要条件より強い制約を課しているのが原因である。

現在のところ、この問題に対する対処方法は見つかっていない。

javaflow の実装方式の問題点（モニタ）

JVM にはモニタの利用に関しても強い制限がある。具体的には、ある関数の実行中に獲得したモニタはその関数が復帰する前に解放されなくてはならない。従って、継続を捕捉する際に関数呼び出しが戻っていく過程でモニタは解放されてしまう。

他方、仮にモニタが保持できたとしても、モニタはスレッドに関連付けられているので、1つのスレッドが複数の継続を取替え引き換え実行する場合に、モニタによる排他制御がアプリケーション開発者の期待するような形になるとは限らない。従って、いずれにせよモニタを利用する場合には注意深いプログラミングが欠かせない。

最適化技法

javaflow のバイトコード後処理方式では、全てのメソッド呼び出しの手前と後にローカル変数及びオペランドスタックを捕捉・復元するコードを挿入しなくてはならない。これはコードサイズの増大及び実行速度の低下を招く。ところが、これらのコードが実際に使われるのは、呼び出されたメソッドの中でスタックの捕捉が行われる場合だけである。従って、呼び出されるメソッドの中では捕捉が起こらないとわかっている場合は（例えば、String.length()など）、コードを挿入する必要がない。コールバックを伴わない事が保証されているライブラリ関数呼び出しなどがこれに該当する。

実際に、javaflow の利用例を見てみると、メソッド呼び出しのほとんどは JDK のライブラリ呼び出しで、コールバックを持たずスタック捕捉が起こらないと保証できるものが多い。従って、このような最適化には大きな効果が期待できる。

単純な実装としては、安全なメソッドのリスト（例えば JDK のメソッドリスト）を用いる、という事が考えられる。より高度な手法としては、あるメソッドがどのメソッドを呼び出すかという閉包を計算したりすることで、安全が保証できるメソッドの数を増やす、という事も考えられる。

しかしながら、実行時に高度な計算をするには時間を要する点と、逆に開発時にクラス間をまたぐ計算をする場合には、実行時には同じクラス群が実行されるとは限らないという点に、留意が必要である。

ワークフローエンジンへの応用

この javaflow プロジェクトを利用して筆者はワークフローエンジン dalma を実装した。

ある種のプログラムは、他のプログラムと対話的に処理を行う必要がある。例えば、メーリングリストに参加する為の電子メールによるユーザー認証などがこの例である。手順としては、このようなプログラムは次のように振舞う。

1. ユーザーからメーリングリスト参加のメールを受け取る

2. 参加確認の本人確認メールを返送
3. ユーザーが本人確認メールに返信
4. メーリングリストへ参加させる

ところが、今までは、このようなプログラムをこのまま手続き的に書き下すことはできなかった。ステップ2からステップ3の間には大きな時間差が生じる可能性があるので、通常、このようなプログラムはイベント駆動的に記述されていた。しかし、イベント駆動的プログラムでは、この進行中の対話に関する状態情報(例えば、どのユーザーか、どのメーリングリストか、など)は、アプリケーション開発者が責任を持って保存・復元しなくてはならず、煩雑である。

dalma では、この煩雑さを避ける為に「メールを送信して返事を待つ」といったようなオペレーションをアプリケーション開発者に提供する。このメソッドの中では、まずメールを送信し、現在の継続を永続化し、そして会話に関する状態情報をメモリから取り除いてしまう。返信が受信されると、永続化された継続は復元され、実行される。これにより、数多くの会話が同時に進行していても、あるいは会話が長期間ブロックしても、計算機資源を無駄にすることなく処理を行える。また、会話の途中で JVM が一度シャットダウンしてもかまわない。

これにより、大きな時間差、または複数回の JVM の起動・終了をまたいで手続き的に徐々に進行するプログラムを容易に記述することができる。

エンドポイント

このように、dalma はある種の非同期 I/O をあたかも同期的に操作できるように見せる。このような仕組みは、電子メールに限らず、様々な I/O で有用である。例えば、SOAP ウェブサービスや、Java Messaging Service(JMS)などのメッセージキューが挙げられる。従って、様々な I/O 接続性を第三者が追加できると便利である。

dalma には、このような I/O 接続レイヤを「エンドポイント」と呼び、これらはプラグインとして追加が可能である。これを用いて、電子メールエンドポイント、JMS エンドポイント、IRC エンドポイントなどを実装した。

永続化とその問題

dalma では、会話が中断するたび(すなわちメッセージの受信を待機する場合)に、実行状態が Java serialization を使って永続化される。このため、会話が中断する場合には、その時コールスタックに乗っているオブジェクトは全て永続化可能でなくてはならない。

実際に dalma を利用してみた結果、幾つかの場合でこれが障害になることがわかった。例えば、ArrayList の iterator オブジェクトは永続化可能でない為に、次のようなコードはうまく動作しない。

```
List<String> data = new ArrayList<String>();
...
```

```
for( String s : data ) {
  ...
  // このメソッドの中で永続化が発生する
  sendMessageAndWaitForReploy(...);
}
```

かわりに、次のようにして配列を使う必要がある。これは、致命的ではないものの、煩雑である。

```
for( String s : data.toArray(new String[0]) ) {
```

今のところ、これが致命的な問題になって dalma が使えなくなる、というケースは見つかっていないが、このような問題が発生したことをアプリケーション開発者に分かりやすい形で伝えるよい方法を見つかっていない。

終わりに

継続の概念そのものにはなじみの薄い Java 開発者も、ここで触れたようなその応用については面白いと思ってもらえるのではないだろうか。

最近は、例えば Ruby や C# 3.0 で、closure や lambda 関数など、様々な関数型言語の機能が手続き型言語に組み込まれるという潮流がある。継続も、今後の進展に期待している。