Javaと相互呼び出し可能なScheme処理系「ぶぶ」における継続機能と例外処理機能の実装

鵜川始陽[†] 湯淺太一[†]
小宮常康[†] 八杉昌宏[†]

「ぶぶ」は，Java言語とのシームレスなインタフェースを備えたオブジェクト指向型のScheme処理系である．Javaの機能を最大限に利用しつつSchemeを使って対話的にプログラミングができることを目標としており，Schemeの継続機能，Javaの例外処理機能の両方に対応していることが望まれる．ぶぶではSchemeとJavaで記述したメソッドを相互に呼び出し合うことができる．Scheme部分の実行には専用に確保した制御スタックを使っているが，Java部分の実行にはJava VMの制御スタックを使っている．したがって，完全なファーストクラスの継続を実現するためには，Javaの継続も得なければならない．しかし，Java言語はこのような手段を提供していない．そこで，継続オブジェクトを生成するときは実行中のScheme部分だけの継続をヒープに保存しておき，Java部分の継続はJava VMの制御スタック上に残しておくことにした．継続を呼び出すときにJava部分の継続が制御スタック上に残っているかどうかを調べ，残っていれば完全な継続呼び出しとして動作する．残っていないときはScheme部分だけの部分継続として呼び出す．また，Javaの例外処理機能をSchemeで記述したプログラムでも直接利用できるようにした．この例外処理機能は継続機能と同時に使うことができる．

Implementation of Continuations and Exceptions for a Scheme System with Java Interface

Tomoharu Ugawa,[†] Taiichi Yuasa,[†] Tsuneyasu Komiya[†]
and Masahiro Yasugi[†]

"Bubu" is an object-oriented Scheme system with seamless interface to Java. Because the goal of Bubu is to provide an interactive environment of Scheme and draw out maximum functionality of Java, we expect it to support both first-class continuations of Scheme and the exception system of Java. In Bubu, methods written in Java and Scheme can call each other. The control stack for methods in Scheme is implemented by using an array object of Java. On the other hand, methods in Java uses the control stack of Java VM. Therefore, when a first-class continuation is captured, a continuation of Java must be also saved to heap. However, Java does not support this facility. In our proposal, when a continuation is captured, only a continuation of Scheme part is saved to heap and the continuation of Java part is left on the control stack of Java VM. When the continuation is called, whether the continuation of Java part is left on the stack of Java VM or not is checked, and if left, this call works as a traditional continuation call. If not, this works as a partial continuation call which has only the Scheme part. In addition, we developed a seamless interface to the Java exception system which can cooperate with the continuation facility.

1. はじめに

「ぶぶ[1]」はJava言語で記述したScheme[2),3)]処理系である．Schemeの持つ対話的なプログラミングインタフェース，ファーストクラスの継続のような利点を残したままJavaの機能を最大限に利用できる処理系にすることが，ぶぶの目標である．そのために，Schemeに対する拡張機能としてぶぶオブジェクトシステム[4)]という機構を備えており，Javaとの言語透過性を実現している．これを用いることで，プログラマはクラスライブラリ等のクラスをロードして利用できるほか，Scheme言語レベルのサブクラスを定義することができる．Scheme言語レベルのサブクラスでは，メソッドをSchemeを使って記述する．

このオブジェクトシステムを使うと，Scheme言語

† 京都大学大学院情報学研究科
Graduate School of Informatics, Kyoto University

で記述された関数と，Java 言語で記述されたメソッドが相互に呼び出し合うプログラムを記述することができる．しかし，Java 言語はメソッドが実行時に生成する関数フレームを操作する手段を提供していない．そのため，Scheme と Java を相互に運用する状況下では，Scheme のファーストクラスの継続を実現することが困難になる．

例外処理機能と継続をファーストクラスとして扱う機能とは，類似点が多く，あわせて設計するのが望ましい．さらに，オブジェクトシステムに適応した例外処理機能にすることで，Java のメソッド実行中に発生した例外を Scheme で処理したり，Scheme の関数の実行中に例外が発生したことを呼び出し元の Java のメソッドに伝えたりするといったことが可能になる．

本論文では，Java と Scheme が相互に呼び出し合う状況での継続の扱いと例外処理機能をあわせて設計し実装を行った．2 章では，実装の対象となるぶぶの内部の動作を述べる．3 章では問題点を明かにしたうえで，解決法を検討し，4 章で実装の詳細を示す．5 章では 4 章で実装した機能の性能評価を行う．6 章では他の Java 上の Scheme の処理系との比較や，他の実装方法との比較を行う．

2. ぶぶ処理系の内部構造

ぶぶは入力された Scheme の式を，その場でぶぶバイトコードと呼ぶ中間コードにコンパイルし，ぶぶバイトコードを解釈実行するインタプリタである．本論文では，この一連の作業を単に Scheme の式を解釈実行すると書く．インタプリタは Java のオブジェクトであり，ぶぶのシステムがマルチスレッドで動作している場合は，各スレッドごとに 1 つ作られる．Scheme の式の解釈実行は，インタプリタのメソッドが行う．インタプリタのメソッドは Java で記述されており，Java VM のスタックを使って動作する．

ぶぶでは，オブジェクトシステムを使うことにより，Scheme と Java を相互に運用することができる．プログラム例を以下に示す．

C1.java:

```
public class C1 {
  public Object m1() { return m2(); }
  public Object m2() { return null; }
}
```

Scheme:

```
(defclass C2 (C1) (:method m2() (f3)))
(define (f1) (f2) (f4))
(define (f2) (send (new C2) m1))
(define (f3) '())
(define (f4) '())
```

この例では，Java 言語で定義したクラス `C1` の Scheme 言語レベルのサブクラス `C2` を定義する．Scheme の関数 `f2` はクラス `C2` のインスタンスを生成し，そのメソッド `m1` を呼び出す．メソッド `m1` はクラス `C2` の定義時にオーバライドされたメソッド `m2` を呼び出すことになり，その中で Scheme の関数 `f3` が呼び出される．

この例のプログラムで `f1` を呼び出し，`f3` まで実行が進んだとき，それぞれの関数フレームは概念的には**図 1** のようにスタックに積まれている．このうち `m1` は Java で記述されたメソッドである．図 1 で示したような概念的なスタックを**ぶぶスタック**と呼ぶことにする．

これらのフレームは，実際には**図 2** のように実現されている．ぶぶ処理系は Scheme の解釈実行用に 1 本のスタック（図 2 の右）を持つ．これを Scheme スタックと呼ぶ．Scheme の関数が呼び出されると，Scheme スタックの上に関数フレームを作り，実行を行う．Scheme スタックは Java の配列を用いて実現されており，自由に参照することができる．

一方，Java のメソッドは Java VM のスタック（図 2 の左）を使って実行される．また，処理系自身も Java で記述されているため，同じ Java VM のスタックを

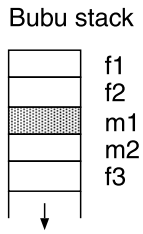

図 1 ぶぶスタック
Fig. 1 Bubu stack.

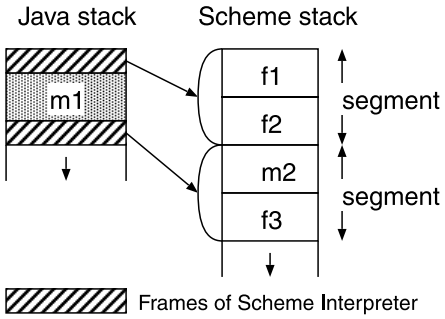

図 2 Java と Scheme の相互運用時のスタック
Fig. 2 Java stack and Scheme stack.

使って実行されてる．このスタックを Java スタックと呼ぶ．Java スタックは Java VM が Java のプログラムを解釈実行するためのスタックであり，Java 言語にはそのスタックを操作する方法がない．したがって，Java スタックは自由に操作することができない．

このように，Scheme の関数は Scheme スタックを使って，Java のメソッドは Java スタックを使って実行される．Scheme で記述されたメソッド `m2` はリターンするとき，Scheme スタック上の直前の関数 `f2` ではなく，呼び出した `m1` にリターンする．このように，Scheme と Java を相互運用すると，実行は Scheme スタックと Java スタックを行き来することになる．

Scheme スタック上の Java のメソッドから呼び出されてから次に Java のメソッドを呼び出すまでの間をセグメントと呼ぶことにする．また，現在実行中のセグメントはカレントセグメントと呼ぶことにする．先の例では，`f1` と `f2` は同一のセグメントで実行されるが，`m2` は `m1` から呼び出されるため，`m2` とさらに `m2` から呼び出される `f3` は，`f1` とは別のセグメントで実行される．さらに `m1` からリターンした後に呼び出される `f4` は `f1` と同じセグメントで実行される．

3. 問題点と設計

3.1 問 題 点

継続とは，プログラムのある時点での残りの実行のことである．Scheme では，`call-with-current-continuation` または `call/cc` と略した関数を利用することにより，実行中の任意の時点で，その時点の継続をキャプチャし，保存することができる．保存された継続は何回でも呼び出される可能性がある．

ファーストクラスの継続は，単純には次のようにして実現することができる．

- 継続のキャプチャでは，スタックの状態をヒープへコピーする（**図 3**）．
- キャプチャした継続が呼び出されたときは，ヒープに保存しておいたスタックの状態をスタックにコピーする（**図 4**）．

Java と Scheme を相互運用する場合，2 章で述べたように制御が Scheme スタックと Java スタックを行き来することになる．したがって，ぶぶで継続をキャプチャするためには，Scheme スタックと Java スタックの両方をヒープにコピーしなければならない．しかし，Java スタックは Java VM が管理している．Java 言語は，Java VM の上で動作するプログラムから Java VM の管理するスタックを操作する手段は提供していない．そのため，一般には，ぶぶでは従来の意味での完全な継続を利用することができない．

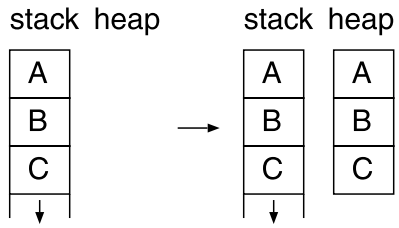

図 3 継続のキャプチャ
Fig. 3 Capturing current continuation.

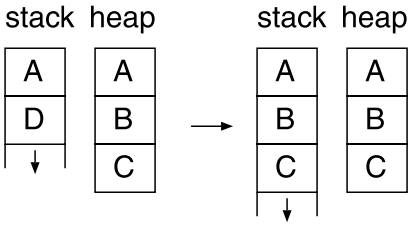

図 4 継続の呼び出し
Fig. 4 Calling a continuation.

なお，特殊な Java VM を利用する等すれば，Java VM のスタックを操作することができるかもしれないが，これでは Java の持つポータビリティを犠牲にしてしまう．ぶぶでは，Java の持つポータビリティを活かすことも目標にしているため，別の方法を考える必要がある．

3.2 検 討

`call/cc` によりキャプチャされた継続が呼び出される場面を考える．

同一セグメントから呼び出される場合 キャプチャしたのと同じセグメントを実行中に呼び出される場合である．

別セグメントから呼び出される場合 キャプチャしたセグメントからリターンした後に呼び出される場合や，さらに Java のメソッドを呼び出した先で呼び出される場合である．

ぶぶの拡張機能であるオブジェクトシステムを使わない場合，セグメントは 1 つしか存在しないため，別セグメントの場合は起こりえない．別セグメントから呼び出されるのは，ぶぶの拡張機能を使った場合に限定される．したがって，同一セグメントから呼び出される場合に継続が Scheme と同様に扱われるようにすれば，他の Scheme 処理系との互換性は保たれる．

仮にキャプチャした継続が必ず同一セグメントからの呼び出しにしか使われないということが分かっていたとする．この場合，継続が呼び出されたときに，呼び出した継続をキャプチャしたセグメント以前のスタッ

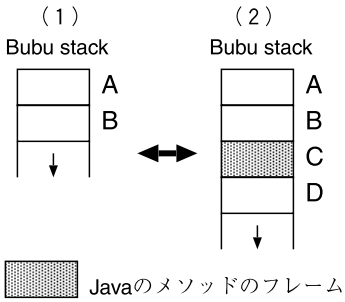

図 5 セグメントを越えた継続の利用
Fig. 5 Calling continuations at different segment.

クの領域は残っている．したがって，この仮定をすれば，継続をキャプチャするとき，スタック上のカレントセグメントの領域だけをヒープに退避すればよい．つまり，Java スタックを操作する必要はない．

次に別のセグメントから呼び出される場合を考える．別セグメントから呼び出される場合，完全な継続を提供しようとすると Java スタックを操作する必要がある．このため，完全な継続を提供するのは一般には不可能である．そこで，可能な限り使い勝手の良い機能を提供することにする．

別のセグメントから呼び出される場合，次の場合が考えられる．

下向きの継続の場合 継続をキャプチャした `call/cc` の本体の実行中に，キャプチャした継続が呼び出される場合である．この場合を下向きの継続（downward continuation）と呼ぶ．この典型的な例として非局所的脱出がある．ぶぶスタックが**図 5** の（1）の状態になっているときに継続がキャプチャされたとする．その後，Java で記述されたメソッド `C` が呼び出され，さらに Scheme で記述された関数 `D` が呼び出されぶぶスタックが図 5 の（2）の状態になったときに，さきほどキャプチャした継続が呼び出される場合である．ユーザはメソッド `C` も脱出し，図 5 の（1）の状態から再開することを期待している．

上向きの継続の場合 キャプチャしたセグメントからリターンした後に呼び出される場合である．この場合を上向きの継続（upward continuation）と呼ぶ．たとえば，スタックが図 5 の（2）の状態でキャプチャされた継続が，`B` までリターンし図 5 の（1）の状態になったときに呼び出される場合である．この場合，図 5 の（2）の状態から再開しなければならないが，Java スタックを操作することはできないので，一般には実現できない．

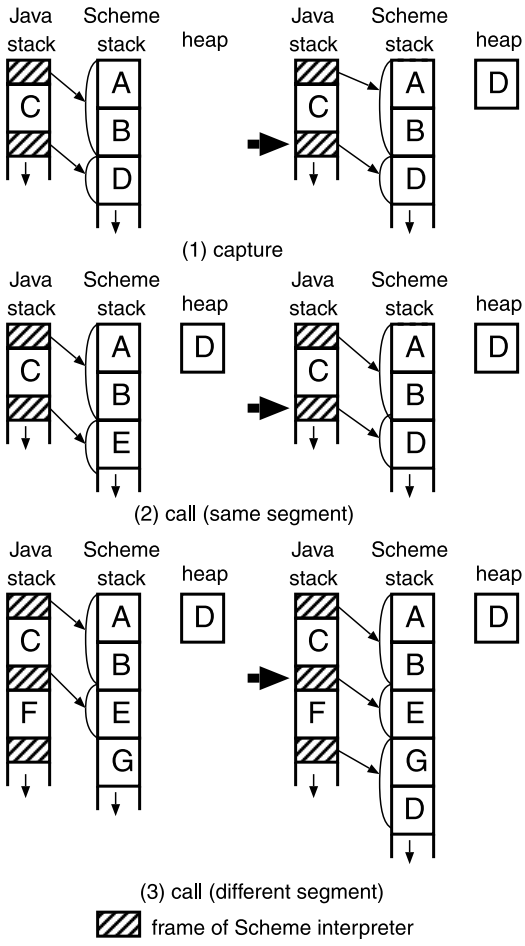

図 6 ぶぶにおける継続の扱い
Fig. 6 Continuation handling in Bubu.

3.3 解 決 策

これらのことをふまえて，可能な限り完全なファーストクラスの継続を提供することを考える．`call/cc` により継続をキャプチャするときは，**図 6** の（1）ようにスタック上のカレントセグメントの領域だけをヒープに退避することにする．

キャプチャした継続を同一セグメント内で呼び出した場合，図 6 の（2）のように，スタックのカレントセグメントの領域だけ捨てて，ヒープに退避しておいたスタックの状態を復元する．キャプチャしたセグメントからまだリターンしていないので，キャプチャしたセグメント呼び出し以前のスタックの状態はまだ残っている．したがって，スタックのカレントセグメントの領域だけを置き換えることで，完全な継続を提供することができる．同一のセグメント内で呼び出した場合は完全な継続として動作するので，Scheme のみで

記述されたプログラムは，完全な Scheme として実行できる．

次に，キャプチャしたのとは別のセグメントを実行中に呼び出した場合は図 6 の（3）のように呼び出す継続を，ヒープに退避した部分だけの部分継続として扱う．このようにすると，部分継続の最後まで実行が終わった後の動作を定義する必要がある．これには次の方法が考えられる．

(1) 部分継続の最後まで実行が終わる前に別の継続を呼び出すことを前提とし，もし最後まで実行してしまった場合はエラーとする．

(2) 部分継続の実行中，最後に評価した値を返り値として，部分継続の呼び出し元にリターンする．

これらは，どちらの方法を採用した処理系を使っても，簡単に他方の動作をするプログラムを書くことができる．(1) の処理系で (2) の動作をシミュレートする場合は，継続の呼び出し前に脱出用の継続を保存しておき，部分継続の実行の最後で保存しておいた継続を呼び出せばよい．逆に (2) の処理系で (1) の動作をシミュレートする場合は，継続の呼び出し以降の文が実行されるのはエラーであると見なして，エラーを発生するようにプログラムを書けばよい．ところで，図 6 の (3) のように，Java のメソッドから呼び出されたセグメントでは，別セグメントでキャプチャされた継続を呼び出しても，いつかは Java のメソッドにリターンするために，呼び出し元のセグメントにリターンする必要がある．そこで本論文では，(2) の方法を採用した．

図 7 は別セグメントでキャプチャされた継続を呼び出す例である．この例では，Java の入力ストリームクラスを拡張してクラス `LeafInputStream` を定義している．`LeafInputStream` は，コンストラクタで与えられた木（木の節は cons セルで表現し，葉はすべて正の整数とする）をたどって，見つかった葉の値を順に返すストリームである．`read` メソッドが呼び出されると，木の探索を開始し，葉が見つかるとその時点の継続を保存した後，葉の値を返す．次に `read` メソッドが呼び出されたときは，保存しておいた継続を呼び出して，探索を再開する．`read` メソッドは実行が終わると Java のメソッドにリターンし，`read` メソッドの呼び出しによってつくられた Scheme のセグメントの実行が終了する．したがって，`read` メソッドは，呼び出されるたびに別のセグメントで動作することになり，探索を再開するために呼び出す継続は部分継続として扱われる．

次に，部分継続を用いて完全な継続をシミュレートする例を示す．**図 8** はオブジェクトシステムを利用したプログラムの例である．`f1` から実行を開始し `f3` の実行中にキャプチャした継続は，`f1` の実行中に呼び出すと `m2` からリターンするところまでの部分継続として呼び出される．しかしメソッド `m1` における `m2` 呼び出し以降の処理を別の関数に切り出し，後からその処理だけを行うことができれば，**図 9** のようにして，完全な継続としての呼び出しをシミュレートすることができる．**図 10** は図 9 を `f3` まで実行したときのぶぶスタックの状態である．このうち図 10 の a の部分の継続は `f2` が `m1` を呼び出す前にキャプチャし `cont2` に保存してある．図 10 の b の部分の継続は Java のプログラムを書き換え，`m1cont` というメソッドで表現する．図 10 の c の部分の継続は `f3` の中でキャプチャされ `cont` に保存される．呼び出すときは，`cont`，`m1cont` を順に部分継続として呼び出した後，`cont2` を完全な継続として呼び出す．`m1cont` が `m1` の環境を利用する場合は，利用される環境を `m2` 呼び出し前に `C1` のインスタンス変数等に保存する必要がある．

```
(import ‘‘java.io.*’’)

(defclass LeafInputStream (InputStream)
  (next Object))

(defmethod LeafInputStream
    LeafInputStream ((tree Object))
  (define (find-next tree)
    (if (pair? tree)
        (or (find-next (car tree))
            (find-next (cdr tree)))
        (call/cc
         (lambda (c)
           (set! next c) tree))))
  (or (call/cc
       (lambda (c)
         (set! next c) #t))
      (find-next tree)))

(defmethod LeafInputStream read ()
  (or (next #f) 0))
```

図 7 入力ストリームクラス
Fig. 7 An input stream class.

3.4 例外処理

別セグメントでキャプチャした継続を呼び出すと，それは必ず部分継続としての呼び出しとなる．そのた

C1.java:
```
public class C1 {
  public Object m1() {
    ...
    m2();
    ...
  }
  public Object m2() { return null; }
}
```
Scheme:
```
(defclass C2 (C1) (:method m2() (f3)))
(define cont '())
(define obj (new C2))
(define (f1) (f2) (cont '()))
(define (f2) (send obj m1))
(define (f3)
  (call/cc (lambda (cc) (set! cont cc))))
```
図 8 部分継続としての呼び出しの例
Fig. 8 Continuation call as partial continuation.

C1.java:
```
public class C1 {
  public Object m1() {
    ...
    return m1cont(m2());
  }
  public Object m1cont(Object x) {...}
  public Object m2() { return null; }
}
```
Scheme:
```
(defclass C2 (C1) (:method m2() (f3)))
(define cont '())
(define cont2 '())
(define obj (new C2))
(define (f1)
  (f2)
  (cont2 (send obj m1cont (cont '()))))
(define (f2)
  (call/cc
    (lambda (cc)
      (set! cont2 cc)
      (send (new C2) m1))))
(define (f3)
  (call/cc (lambda (cc) (set! cont cc))))
```
図 9 部分継続を使って完全な継続を表現する例
Fig. 9 Expressing full continuation by using partial continuation.

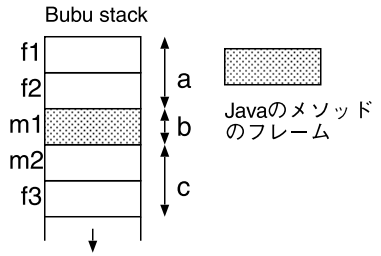

図 10 図 9 において f3 実行中のぶぶスタック
Fig. 10 Bubu stack when executing f3 of Fig. 9.

め，継続を使ったセグメントをまたがる非局所的脱出は行えない．そこで，Java 言語にある例外処理機能[5]を Scheme から扱う機能を提供し，これを用いてセグメントを越えた非局所的脱出ができるようにする．例外処理機能を提供することで，非局所的脱出以外にも，Java のメソッドに例外が発生したことを伝えたり，Java のメソッド内で投げられた例外を受け取ったりするといった処理が可能になり，オブジェクトシステムの利用範囲が広がる．

ぶぶでは，オブジェクトシステムを利用することにより，Java のオブジェクトを Scheme のファーストクラスオブジェクトとして扱うことができる．そこで，例外の種類の識別には，Java の例外クラス（ `Throwable` クラスのサブクラス）を利用する．例外のキャッチャは Java の try-catch と同様の能力を持ち，処理する例外の種類にはあらゆる例外クラスを指定することができる．例外のキャッチャは Java 言語単体での利用と同様にダイナミックスコープに従って検索される．**図 11** の状態で例外が発生し，例外オブジェクトが投げられたとすると，E，D，C，B，A の順に，そのハンドラが処理すべき例外かどうか調べられ，処理すべき例外であればキャッチされる．

例外ハンドラは次のようにして定義する．

`(with-handler` *handler-list body*`)`

*handler-list*は，例外クラス と例外ハンドラのリストを指定する．*body*はその例外ハンドラを設定した環境で実行する式の列を指定する．また，次のようにして例外を発生させる．

`(throw` *exception*`)`

ここで *exception*は `Throwable` クラスのサブクラスのインスタンスである．たとえば，
```
(with-handler
  ((Exception (lambda (e) (write e))))
  (throw (new Exception)))
```
では，例外 `Exception` が発生したときには，その内容を表示するという例外ハンドラを設定し，本体を実

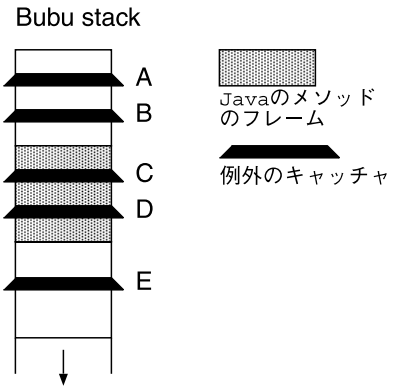

図 11 例外ハンドラの設定されたぶぶスタック
Fig. 11 Bubu stack with exception handlers.

行する．本体では `Exception` のインスタンスを生成し，それを使い例外を発生させる．発生した例外は，設定した例外ハンドラが受け取る例外 `Exception` に一致するため，キャッチされて例外ハンドラが例外オブジェクトに適用される．

4. 実　　装

4.1 インクリメンタルスタック/ヒープ法

継続の実装にはインクリメンタルスタック/ヒープ法[6]を用いる．まず，インクリメンタルスタック/ヒープ法を簡単に説明する．`call/cc` により継続をキャプチャするときは，その時点でのスタックの内容を関数フレームに分け，フレームを表現するオブジェクトのリストとしてヒープに退避する（**図 12** の（2））．このとき，スタック上ではスタックを表現する配列のインデックスで表されているリンクやスタック上での相対的な位置で表現してあるリンクは，ヒープ上ではリンク先のフレームを表現するオブジェクトへの参照として表現する．退避したフレームのリストはキャプチャした継続から指される．また，スタックに積まれたフレームはすべて破棄し，スタックの続きを表すレジスタ（more レジスタと呼ぶ）からも退避したスタックのリストを指すようにする．空になったスタックで `call/cc` の引数に与えられた本体を実行する．本体の実行を終え `call/cc` からリターンしようとすると，スタックアンダフローが発生する．これをトラップして，more レジスタから 1 フレーム分だけスタックに戻す（図 12 の（3））．

ぶぶでは，インクリメンタルスタック/ヒープ法をベースにして，Java と相互呼び出しを行った際の，セグメントの境界までの継続のキャプチャ，部分継続呼び出し等の拡張を行う．

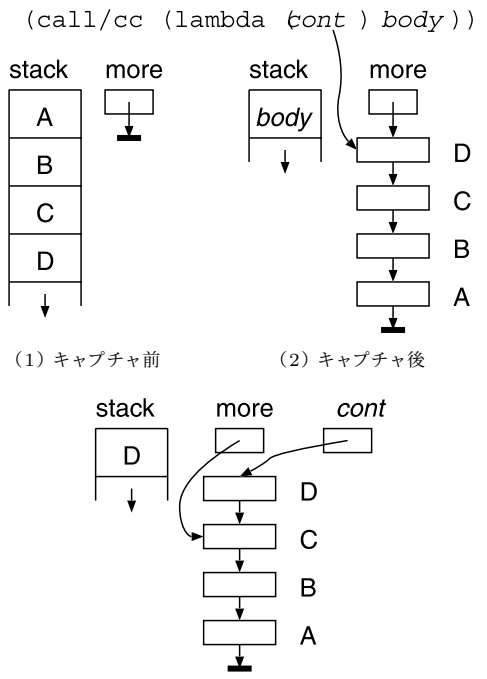

図 12 インクリメンタルスタック/ヒープ法
Fig. 12 Incremental stack/heap strategy.

4.2 セグメント開始時の処理

新しいセグメントの実行が開始されると，more レジスタ等のセグメントごとに持つ情報はスタック上に保存され，そのときのスタックトップを新たなスタックの底として実行を始める．これにより，継続のキャプチャの際にキャプチャする範囲を，カレントセグメントのみにすることができる．また，スタックアンダフローも，スタック上のカレントセグメントの領域が空になった時点で検出できる．

4.3 継続呼び出し時の判定

継続を呼び出すときには，呼び出そうとしている継続がカレントセグメントでキャプチャされたものかどうかを調べる．これには，各セグメントに一意な番号を付けるとよい．この一意な番号をタイムスタンプと呼ぶことにする．タイムスタンプを発行するために，インタプリタクラスのクラス変数としてタイムスタンプ発行用のレジスタを用意する．新たなセグメントが生成されると，タイムスタンプ発行用のレジスタからタイムスタンプを発行し，レジスタはインクリメントされる．タイムスタンプ発行用のレジスタをクラス変数とするのは，インタプリタが各スレッドごとに 1 つ用意されるので，マルチスレッドで動作している場合の別スレッドのセグメントを区別するためである．継

続をキャプチャすると，キャプチャしたセグメントのタイムスタンプを継続の属性として保存する．継続がカレントセグメントでキャプチャされたかどうかは，カレントセグメントのタイムスタンプと継続オブジェクトの持つタイムスタンプを比較すればよい．別の方法として，タイムスタンプレジスタの代わりに，直接継続オブジェクトへの参照をセグメントごとに保持する方法が考えられる．しかし，この方法では，Scheme 言語レベルで継続への参照がなくなり，継続オブジェクトが不要になったとしても，Java 言語レベルでは継続オブジェクトへの参照が残ってしまい，Java のごみ集めが利用できない．

このようにして，呼び出そうとしている継続がキャプチャされたセグメントを調べた結果，カレントセグメントでキャプチャされた継続であれば，スタックのカレントセグメントの領域を捨てて，呼び出した継続の持つスタックで置き換える．別のセグメントでキャプチャされた継続であれば，カレントセグメントのスタックは残したままで，呼び出そうとした継続を部分継続として呼び出す．

4.4 部分継続の実装

継続が部分継続として呼び出されたとき，部分継続の実行は呼び出したセグメントと同じセグメントで行う．したがって，部分継続の実行中に継続をキャプチャすると，キャプチャした継続には部分継続呼び出し以前のフレームも含まれる．これを実現するために，文献 7) にある拡張された継続の実現と同様の方法を用いる．文献 7) では，継続を呼び出すとき，その継続の実行が終わった後に実行する継続を，継続呼び出し時のオプショナル引数として明示的に指定する拡張を行っている．本論文の部分継続呼び出しでは，呼び出された継続の実行が終わった後に実行する継続として，暗に呼び出し前の継続が与えられていると考えることができる．拡張された継続の実装には，more レジスタをスタック状にした継続スタックと呼ばれるスタックを用意する．実装では同じスレッドで動作するすべてのセグメントが同じ継続スタックを使っているが，ここでは便宜上，セグメントごとに別の継続スタックを持つことにする．この継続スタックをコピーしたものに，先に述べたタイムスタンプ等のいくつかの属性を加えたものを継続オブジェクトとする．

継続をキャプチャするときは，スタックからフレーム単位でポップし，フレームオブジェクトのリストにする．このリストを継続スタックに積み，スタックと more レジスタを空にする．`call/cc` 式からリターンするときは，スタックアンダフローが発生し，more

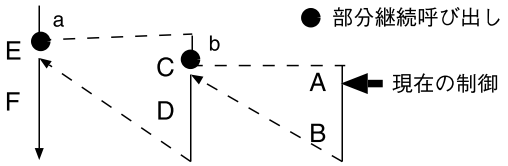

図 13 部分継続をともなう実行の様子
Fig. 13 Control flow when using partial continuation.

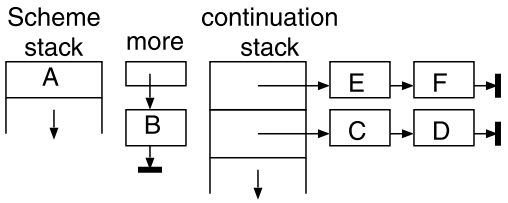

図 14 部分継続呼び出し後のインタプリタ
Fig. 14 Call of partial continuation.

レジスタからスタック上にフレームを復元しようとする．しかし，more レジスタにもフレームがないので，まず継続スタックからフレームのリストをポップし，more レジスタに設定する．さらに more レジスタから 1 フレームだけスタック上に復元し実行を再開する．

継続が完全な継続として呼び出されるときは，継続オブジェクトの持つ継続スタックで現在の継続スタックを置き換え，スタック，more レジスタを空にすればよい．

継続が部分継続として呼び出されるときは，まず，呼び出した時点の継続を継続スタックに退避する．これは継続のキャプチャと同様に行う．そのうえで，呼び出した継続オブジェクトの持つ継続スタックの内容をすべて現在の継続スタックの上に追加し，スタックと more レジスタを空にする．

部分継続をともなう実行の様子を**図 13** に示す．また，図 13 の矢印の位置に制御があるときのインタプリタの状態を**図 14** に示す．まず a の時点で継続が部分継続として呼び出されると，a の時点でスタックに積まれてある E，F のフレームをフレームオブジェクトのリストとしてヒープに退避し，継続スタックに積む．さらに，呼び出された継続の持つ継続スタックを継続スタックに追加し，継続スタックのトップは C と D のフレームを含むリストになる．このリストは，すぐに more レジスタに取り出され，実行される．さらに，C の実行中 b の時点で，more レジスタに D のフレームをを残した状態で，別の継続が部分継続として呼び出されたとする．すると，スタック上にある C のフレームと more レジスタ上にある D のフレームは

あわせて継続スタックに積まれ，先ほどの呼び出しと同様に呼び出された継続を実行する．図 14 では，スタック上で A が実行中であり，more レジスタに B の実行のためのフレームが残っている．

4.5 部分継続の実装の最適化

この実装には最適化の余地がある．部分継続の呼び出し時点に継続を継続スタックに積む作業を本当に行う必要があるのは，部分継続呼び出し中に継続がキャプチャされる場合だけである．また，部分継続呼び出し中，継続がキャプチャされるまでは部分継続呼び出し時のスタックは壊されることはない．したがって，部分継続呼び出し時のスタックをヒープに退避する作業は実際に継続がキャプチャされるまで遅延することができる．このためには，部分継続呼び出しのときに，ヒープに退避されたフレームのリストを継続スタックに積む代わりに，スタック上の退避されるべき範囲を指すオブジェクトを継続スタックに積めばよい．more レジスタにもフレームが残っている可能性があるが，これは部分継続呼び出し時に more レジスタをスタック上に退避することで解決できる．また，スタックアンダフローの検出，more レジスタからのフレームの復元を正しく行うためには，仮のスタックの底を指すレジスタを用意し，部分継続の呼び出しを行った直後のスタックトップが仮のスタックの底となるようにすればよい．仮のスタックの底を指すレジスタは，部分継続の呼び出しのときは，more レジスタとあわせてスタック上に保存しておく必要がある．

最適化した図 13 の矢印の位置に制御がある状態が**図 15** である．図 14 と違い，E，F，C のフレームはスタックに残したままである．

この最適化により，プログラム全体の実行時間が短縮される可能性があるほか，継続の呼び出しが部分継続としての呼び出しとなった場合にかかる時間を抑え，継続の呼び出しにかかる時間の見積りを正確にすることができる．

4.6 例外の実装

インタプリタは例外ハンドラを保持するために例外ハンドラレジスタを持つ．`with-handler` により例外ハンドラが設定されると，処理する例外と例外ハンドラの対応を表すオブジェクトを作る．これを単に例外ハンドラと呼ぶ．例外ハンドラは設定された逆順でリストにして例外ハンドラレジスタから指しておく．例外が発生すると，基本的には，例外ハンドラレジスタからリストを順に見て該当するハンドラがあるか検索する．

例外ハンドラレジスタは more レジスタと同様，新

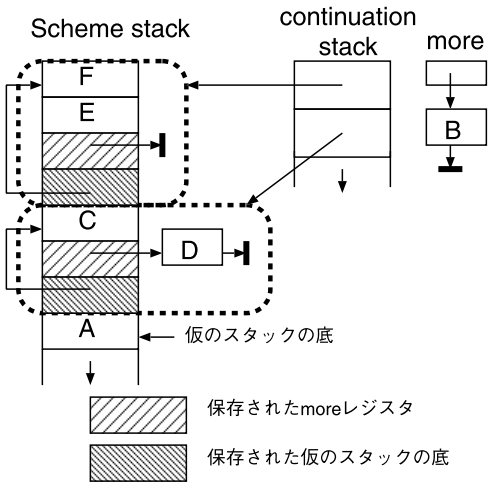

図 15 最適化された部分継続呼び出し
Fig. 15 Optimized call of partial continuation.

しいセグメントの実行が始まるときにスタック上に保存し，レジスタをクリアする．これにより，Java のメソッド中で設定された例外ハンドラと Scheme の関数中で設定した例外ハンドラを正しい順序で検索することができる．インタプリタはあらゆる例外をいったんキャッチし，カレントセグメントで例外ハンドラが設定されていれば呼び出し，そうでなければ，単にその例外を呼び出し元の Java のメソッドに投げればよい．

継続をキャプチャするときは，例外ハンドラのリストもあわせてキャプチャする．また，例外ハンドラのリストもフレームのリストと同様，継続としてキャプチャされなければならない．例外フレームのリストもフレームのリストとあわせて継続スタックに保存する．例外が発生したときに行う例外の検索もこれを考慮して行う．例外ハンドラレジスタの最後まで検索しても，継続スタックに継続が積まれていれば，継続をポップし，保存されていた例外ハンドラのリストを例外ハンドラのレジスタに戻したうえで再度検索を行う．

例外を処理すべき例外ハンドラが見つかり，例外の処理が終わったとする．すると，制御を例外を処理した例外ハンドラを設定した `with-handler` からリターンするところに戻さなければならない．これを高速に行うため，例外ハンドラには `with-handler` により設定されたときに，リターン先のフレームを保存する．リターン先のフレームは，そのフレームがスタック上にあるときはスタックのインデックスにしておき，フレームがヒープに退避されると，フレームオブジェクトへの参照に切り替える．例外ハンドラの実行が終わると，例外ハンドラから参照されているフレームが

表 1 ベンチマークプログラムの実行時間（秒）
Table 1 Evaluation time of benchmark (sec).

	tak	（比）	takl	（比）	ctak	ctak (ep)
ぶぶ	2.487	0.68	15.621	0.44	13.633	19.427
Kawa	3.663	1.00	35.287	1.00		9.865

more レジスタから指されるようにし，スタックを空にして，リターンすればよい．継続スタックは例外ハンドラの検索のために，必要なところまでポップされているため，リターンのときには何もする必要がない．

Java 言語では，すべてのメソッドは，コンパイル時にそのメソッドが投げる可能性のある例外の種類を指定しておく必要がある．しかも，メソッドをオーバライドする場合，オーバライドされるメソッドと同じ種類の例外しか投げることができない．ぶぶで Java のメソッドをオーバライドする場合も同様の問題が発生する．Java 言語には例外的に `RuntimeException` という例外のクラスがあり，この型の例外はあらゆるメソッドが，コンパイル時の宣言なしに投げることができる．そこで，例外を Java のメソッドに投げるとき，そのまま投げることのできない（コンパイル時に宣言されていない）例外は `RuntimeException` を継承するクラスの例外でカプセル化して投げる．するとその例外は再び Scheme インタプリタにキャッチされ，1 つ前のセグメントで設定された例外ハンドラを検索することになる．

5. 性能評価

Gabriel ベンチマーク[8]の中から，tak，takl，ctak のベンチマークについて，ぶぶと Java 上の Scheme 処理系である Kawa[9]で実行し，比較を行った．ベンチマークは，以下の環境で行った．

- Sun Enterprise 3000
- JDK 1.1.7
- Kawa 1.6.1

Kawa は Scheme プログラムを Java のバイトコードにコンパイルして，Java VM 上で動作させる．継続の実現には，Java の例外を用いており，下向きの継続に限定して利用することができる．一方，ぶぶは Scheme スタックを用いているため，Scheme の関数呼び出しのたびに，Scheme スタックを実現している Java の配列への数回のアクセスを必要とする．

ベンチマークで用いた tak，takl は関数呼び出しとリターンを大量に行うベンチマークである．これらのプログラムは継続のキャプチャを行わないため，ぶぶが Scheme スタックを用いていることのオーバヘッドをおおまかに調べることができる．ctak は，tak のリターンを下向きの継続の呼び出しに置き換えたベンチマークであり，継続を使った場合の性能を調べることができる．

結果は**表 1** のようになった．なお，ctak のベンチマークについては，ぶぶでは継続を用いた実装と例外を用いた実装が可能なため，継続を用いたオリジナルの ctak に加え，例外を用いた ctak の結果も示した．表 1 では前者を “ctak”，後者を “ctak（ep）” として示す．また，Kawa の継続は脱出の用途にしか利用できないため，表 1 では ctak（ep）に分類した．ぶぶの ctak（ep）では，Java のオブジェクトをアクセスするオーバヘッドを削るため，返り値は例外オブジェクトに格納せずに，大域変数に置いた．

tak，takl では Scheme のプログラムを Java バイトコードにコンパイルした Kawa よりも Scheme スタックを使って解釈実行しているぶぶの方が高速に実行できている．これは，Scheme スタックを利用することによるオーバヘッド等，継続を利用しないプログラムにかかる余分なオーバヘッドは，処理系の他の部分の実装方式の違いによる実行速度の差に比べてそれほど大きくないことを示している．

一方 ctak の実行には，Kawa の ctak（ep）よりも時間がかかっている．これは，Kawa が Java の例外を用いているのに対してぶぶでは Scheme スタックをコピーしているためである．しかし，処理系全体の実行速度がぶぶの方が速いことを考慮しても，極端に実行効率が落ちているわけではなく，完全な継続を利用できるようにするためには許容できる範囲と考えられる．

次にぶぶの ctak と ctak（ep）とを比較すると，ctak（ep）では ctak に比べさらに時間がかかっている．この原因としては，

- 例外ハンドラの設定を動的に行うため，例外ハンドラ設定時にオブジェクトの生成や Scheme スタックへのアクセスが発生する，
- 継続の呼び出しは値を返すだけであるのに対して，例外を投げると例外ハンドラが起動されるため，関数呼び出しのオーバヘッドが発生する，

が考えられる．

6. 議　　論

Java 上の Scheme としては，Kawa のほかに，

Skij [10] と SILK [11] がよく知られている．これらの処理系は Scheme の関数呼び出しを Java の関数呼び出しで表現しており，Kawa 同様，`call/cc` の実装には Java の例外を用いている．そのため用途は下向きの継続に限定されている．本論文の対象としたぶぶの場合は，Scheme で記述されたプログラムはインタプリタとして実行し，さらに Scheme 用のスタックを用意しているため，部分的に継続をキャプチャし，ある程度完全な継続と同様に扱うことができるようになった．

完全なファーストクラスの継続を実現するためには，本論文の方法以外に，Java のメソッドを実行する Java VM に変更を加えるという方法[12]も考えられる．しかし，ぶぶは Web ページに埋め込んだアプレットとしての利用も考えており，特殊な Java VM なしに実現できる本論文の方法を選んだ．

7. ま と め

本論文では，Java のメソッドと Scheme の関数が相互に呼び出すことができる Scheme 処理系ぶぶにおいて，可能な限り完全なファーストクラスの継続の実装法を設計し，実際に実装した．これにより，ぶぶはフルセットの IEEE Scheme [3] に準拠した Scheme とオブジェクトシステムが両立した処理系となった．本論文で用いた方法では，Java のメソッドを実行する Java VM のスタックを直接操作することなく，ほとんど完全なファーストクラスの継続を実現した．

まず，Scheme の関数の実行のためのスタックを Java のメソッドを呼び出したところで区切って考え，この区切られたスタックをセグメントと呼ぶことにした．継続をキャプチャするときは，カレントセグメントのみをヒープに退避する．継続が呼び出されると，キャプチャしたときと同じセグメントなら現在の継続を捨てて呼び出された継続を実行する．別のセグメントなら，現在の継続を残して，呼び出された継続を実行する．こうすることで，同じセグメントでの呼び出しは完全な継続として，別のセグメントでの呼び出しは部分継続として呼び出される．部分継続として呼び出される場合でも，ある程度の場合においては，完全な継続と同様の能力を持つ．さらに，部分継続として呼び出されることでできなくなった，セグメントを越えた非局所的脱出を可能にするために，Java の例外を扱うインタフェースを用意した．

本論文で用いた方法はぶぶの場合にとどまらず，継続の一部をキャプチャすることができないような実装のインタプリタで，ファーストクラスの継続を実装する場合にも応用できると考えられる．たとえば，インタプリタ言語 X が C 言語とのインタフェースを持ち，C 言語から X がコールバックされる可能性のある場合，C 言語のスタックはプラットフォームに依存するため，一般には直接操作できない．このような場合でも，本論文の方法を適用することができる．

参 考 文 献

1) Yuasa, T.: An Object-oriented Scheme System Bubu with Seamless Interface to Java, *Parallel and Distributed Computing for Symbolic and Irregular Applications*, Ito, T. and Yuasa, T.(Eds.), pp.101–121, World Scientific (2000)
2) Kelsey, R., Clinger, W. and Rees, J.(Eds.): *Revised^5 Report on the Algorithmic Language Scheme*, Vol.11, No.1, Kluwer Academic Publishers (1998).
3) IEEE: *IEEE Standard for the Scheme Programming Language* (*IEEE P1178*) (1991).
4) 窪田貴志，湯淺太一，倉林則之，八杉昌宏，小宮常康：Java 上の Scheme 処理系「ぶぶ」における単一のクラスローダを用いたオブジェクトシステムの実装，情報処理学会論文誌：プログラミング，Vol.42, No.SIG7(PRO11), pp.57–69 (2001)
5) Gosling, J., Joy, B., Steele, G. and Bracha, G.: *The Java Language Specification*, Second Edition, Addison-Wesley.
6) Clinger, W.D.: Implementation Strategies for First-Class Continuations, *Higher-Order and Symbolic Computation*, Vol.12, No.1, pp.7–45, Kluwer Academic Publishers (1999).
7) 小宮常康，湯淺太一：Future ベースの並列 Scheme における継続の拡張，情報処理学会論文誌，Vol.35, No.11, pp.2382–2391 (1994).
8) Gabriel, R.: *Performance and Evaluation of Lisp System,* MIT Press (1985).
9) Lisp Users Conference: *Kawa: Compiling Scheme to Java* (1998). http://www.gnu.org/software/kawa/
10) Travers, M.: Skij. http://www.alphaworks.ibm.com/tech/skij/
11) Anderson, K., Hickey, T. and Norvig, P.: SILK. http://www.norvig.com/SILK.html
12) 山本晃成，湯淺太一：末尾再帰の最適化と一級継続を実現するための JVM の機能拡張，第 33 回 PRO 研究会 (2001)

(平成 13 年 3 月 12 日受付)
(平成 13 年 6 月 21 日採録)

鵜川　始陽

1978 年生．2000 年京都大学工学部情報学科卒業．同年より同大学大学院情報学研究科修士課程に在学中．言語処理系に興味を持つ．

湯淺　太一（正会員）

1952 年神戸生．1977 年京都大学理学部卒業．1982 年同大学大学院理学研究科博士課程修了．同年京都大学数理解析研究所助手．1987 年豊橋技術科学大学講師．1988 年同大学助教授，1995 年同大学教授，1996 年京都大学大学院工学研究科情報工学専攻教授．1998 年同大学院情報学研究科通信情報システム専攻教授となり現在に至る．理学博士．記号処理，プログラミング言語処理系，超並列計算に興味を持っている．著書「Common Lisp 入門」（共著),「C 言語によるプログラミング入門」,「コンパイラ」ほか．日本ソフトウェア科学会，電子情報通信学会，IEEE，ACM 各会員．

小宮　常康（正会員）

1969 年生．1991 年豊橋技術科学大学工学部情報工学課程卒業．1993 年同大学大学院工学研究科情報工学専攻修士課程修了．1996 年同大学院工学研究科システム情報工学専攻博士課程修了．同年京都大学大学院工学研究科情報工学専攻助手．1998 年より同大学院情報学研究科通信情報システム専攻助手．博士（工学）．記号処理言語と並列プログラミング言語に興味を持つ．平成 8 年度情報処理学会論文賞受賞．

八杉　昌宏（正会員）

1967 年生．1989 年東京大学工学部電子工学科卒業．1991 年同大学大学院電気工学専攻修士課程修了．1994 年同大学院理学系研究科情報科学専攻博士課程修了．1993〜1995 年日本学術振興会特別研究員（東京大学，マンチェスター大学）．1995 年神戸大学工学部助手．1998 年より京都大学大学院情報学研究科通信情報システム専攻講師．博士（理学）．1998 年より科学技術振興事業団さきがけ研究 21 研究員．並列処理，言語処理系などに興味を持つ．日本ソフトウェア科学会，ACM 会員．