プログラミング-言語・基礎・実践- 8－21
(1992. 8. 21)

Schemeにおけるプロセス管理

戸川 敦之† 大久保 英嗣† 大野 豊† 白川 洋充††

† 立命館大学 理工学部 情報工学科
†† 近畿大学 理工学部 経営工学科

プログラミング言語が第一級の継続をサポートすることによって, 言語の仕様を拡張することなく, 様々な制御構造を実現できることが知られている. プロセス管理も継続によって簡潔に表現できる制御構造の一つである. 従来はプロセス管理を実現するためにはオペレーティングシステムの支援が必要であった.

本論文では, 継続をプロセスの状態の表現に用いることができること, そして, 第一級の継続を持つ言語でプロセス管理プログラムが簡潔に記述できることを示す. また, 閉包を用いて情報の隠蔽を行うことによって, 特別な保護機構を用いずに, プロセス管理プログラム内の情報を保護できることを示す.

Process Management in Scheme

Atsushi Togawa† Eiji Okubo† Yutaka Ohno† Hiromitsu Shirakawa††

†Department of Computer Science and Systems Engineering,
Faculty of Science and Engineering,
Ritsumeikan University
56-1 Tojiin, Kita-ku, Kyoto 603, Japan

††Department of Industrial Engineering,
Faculty of Science and Engineering,
Kinki University
3-4-1 Kowakae, Higashi-Osaka 577, Japan

Various control facilities can be built in the languages which support first-class continuations. Process management is one of such control facilities. Kernel support is not required to provide process management in the languages.

This paper shows how the state of a process is respresented by first-class continuations, and the process managers can be written concisely in terms of first-class continuations. And we propose protection mechanisms using closure to protect the process manager from applications programs.

1 はじめに

第一級の継続によって様々な制御構造を実現できることが知られている．プロセス管理もそのような制御構造の一つである．従来はプロセス管理を実現するためにはOSの支援が必要であった．しかし，第一級の継続を持つ言語ではその必要はなく，言語が持つ機能だけを用いてプロセス管理を実現することができる．このような手法で実現されたプロセス管理機構はOSが提供するプロセス管理機構と比較して次のような点が優れている．

性能．OSが提供するプロセス管理機構を利用する場合はシステムコールのオーバヘッドが問題となる．システムコールの度にカーネルモードとユーザモードの間を往復しなくてはならないが，この処理は手続き呼び出しに比べるとはるかに重い処理である．また，OSはシステムの保護のために様々な検査を行わなくてはならない．それに対して，任意のポインタ操作を許さない言語で実現されたプロセス管理プログラムではコンパイル時にある程度の検査を行うことができる．

柔軟性．プロセス管理プログラムとアプリケーションプログラムを同一の言語で記述できる．従って，プロセス管理プログラムは，より親密にアプリケーションプログラムと対話することが可能となる．これによって，個々のアプリケーションの要求に応えられるプロセス管理をより簡単に実現することができる．

SchemeはLispの方言の一つであり，第一級の継続を備えた言語である[1]．本論文では，この言語を用いてプロセス管理を実現する手法を述べる．実現したプロセス管理プログラムは，制御の流れであるスレッドと，関連する複数のスレッドの集団であるタスクの概念を有している．従来のOSのタスクはスレッド群の実行を制御する機能だけでなく，アドレス空間や保護情報を持つ重いものであった．従って，スレッド群の実行を制御する目的に使用することが難しい場合があったが，本プロセス管理プログラムでは，そのような場合にもタスクを用いて制御を行うことができる．

本論文では，まず2章でScheme言語の特徴を述べた後に，3章で実現するプロセス管理プログラムの概要を示す．4章では実現手法として，継続を用いたスレッドの状態の表現，タスクとスレッドの表現，非同期的事象によるプリエンプションの3つの点を中心に述べる．

2 Schemeの概要

SchemeはLispの方言の一つであり，Lispと同様の構文を採用している．また，Common Lispと同じように静的なスコープを持つ．また，処理系は末尾再帰を適切に扱わなくてはならないことが言語仕様の中で明記されている．Schemeの最大の特徴は，第一級の継続を有していることである．第一級の継続を用いることによって様々な制御構造を実現することができる．

2.1 継続

Schemeで式が評価される時は常に，評価結果を受け取り，残りの計算を行う**継続**(*continuation*)が存在している．つまり，継続は，ある時点以降の計算を表現している．通常はプログラマが継続の存在を意識することはないが，様々な制御構造を実現する場合に，陽に継続を操作する必要が生じることがある．基本的な手続き`call-with-current-continuation`(以後，`call/cc`と省略する)を呼び出すことによって，この継続と全く同じ振舞いをする手続きであるエスケープ手続きが得られる．この手続きを呼び出すと，呼び出した時点の状況とは無関係に，このエスケープ手続きを生成した`call/cc`から後に制御が移行する．例えば，次の式は1ではなく0を返す．

```
(call/cc (lambda (k) (k 0) 1))
```

この式では, call/cc によって生成されたエスケープ手続きが下線部の手続きに引数 k として与えられる. (k 0) を評価することによって, call/cc が値 0 を返す. call/cc とエスケープ手続きの呼び出しが Common Lisp の catch 及び throw と異なっている点は, 大域変数などにエスケープ手続きを格納しておくことによって, call/cc の実行を終えた後でもエスケープ手続きを起動することができることである. つまり, 手続きの内から外へのジャンプだけでなく, 外から内へのジャンプも可能となっている.

2.2 手続きと閉包

手続きの操作が容易であるという特長と静的なスコープを持つという特徴のために, Scheme では内部状態を持つデータ構造を閉包を用いて表現することがしばしば行われる. 例えば, 次に示すのは put! と get という二つのメッセージを受け付け, 内部に一つの値を保持するオブジェクトを生成する手続きと, その実行例である.

```
(define (make-cell cell)
  (lambda (msg)
    (case msg
      ((get) cell)
      ((put!) (lambda (x)
                (set! cell x))) )))
(define a-cell (make-cell #t))
; 最初に make-cell に与えた値が返る
(a-cell 'get) ⟹ #t
; (a-cell put!) は手続きを返す.
; これに引数 "foo" を与えて呼び出す
((a-cell 'put!) "foo")
; すると cell の値が変更される
(a-cell 'get) ⟹ "foo"
```

変数 cell は手続き make-cell 中でしか参照することができず, 他のプログラムが不用意にこの変数を参照することが不可能となっている.

3 Scheme におけるプロセス管理

本論文で提案しているプロセス管理プログラムは次の機能をアプリケーションプログラムに提供している.

- 複数の制御の流れ (スレッド)
  スレッドの生成・中断・強制終了, スレッド固有データの操作
- 関連するスレッド群 (タスク) に対する様々な操作
  タスクの生成・中断・強制終了
- モニタ機構に基づく排他制御と同期

このように, タスクに関する機能を除けば Mach の C-threads[2], あるいは ML-threads[3] とほぼ同じ機能を提供している. タスクとスレッドを提供している多くの OS では, タスクはスレッドの入れ物であると同時にアドレス空間を含む様々な資源の集合体である. これに対して, 本プロセス管理プログラムではタスクは前者の役割しか持っていない. このため, 本プロセス管理プログラムのタスクは OS のタスクに比べて軽くなっており, より多くの場面でスレッド群の実行を制御するために利用することができる.

タスクの生成は以下の make-task 手続きによって行われる.

(make-task *mother*)

この構文によって*mother*を親とするタスクが生成され, 結果としてタスクオブジェクトが返される. タスクオブジェクトに様々な引数を与えて呼び出すことによってタスク内のスレッドの実行を制御することができる. 例えば, 次の式を評価すると,

((task 'exit) *value*)

タスク中の全スレッドと全ての子孫タスクが終了させられ, タスクの結果として*value*が設定される. この機能を用いることによって, 並列 or を記述することができる. 例えば, 次のプログラムは二つのスレッドを生成し, どちらが早く exit を実行したかに応じて 1 あるいは 2 を返す.

```
(let* ((task (make-task #f))
       (make-thread (task 'make-thread)))
  (make-thread
   (lambda () ···((task 'exit) 1)···))
  (make-thread
   (lambda () ···((task 'exit) 2)···))
  (task 'result))
```

このプログラムの中の (task 'make-thread) は, スレッド生成手続きを返す式である. この手続きを呼び出すことによってスレッドが生成される. また, (task 'suspend) を評価することによって, タスク中の全スレッドと全子孫タスクの実行が中断される.

4 実現手法

一般に, スレッドを管理するプログラムはスレッドの状態を保持するデータ構造を管理しているが, この状態とはスレッドが中断された時点の継続に他ならない. 従って, 継続を操作する機能を持った言語では, この機能をプロセスの状態を保存するために用いることができる. Wand は, 継続と, 継続に与える引数の対によってプロセスを表現し, ディスパッチャとセマファをこの手法で実現している [8]. 本論文のプロセス管理プログラムは, スレッドの管理に必要となる様々な情報をスレッドオブジェクトと呼ぶ閉包の中に保持している. さらに, ディスパッチャや排他制御・同期手続きが継続を管理している. このような方式を採用したのは, 継続を参照するのは, ディスパッチャや排他制御・同期手続きだけであり, 他のモジュールから参照されるべきではないことによる.

4.1 スレッドとタスクの実現

以下にタスク生成手続き make-task を示す.

```
(define (make-task mother)
  (let (; このタスク中のタスク
        (daughters '())
        ; このタスクを親とするスレッド
        (threads '())
        ; 外側のタスクに要求を行うための手続き
        (to-mother
          (if mother (to-mother 'make-daughter)
              (lambda ())))
        ··· ))
    ···
    (define (make-thread proc)
      (let (···) ···
        (define (thread msg) (case msg ···)
        (make-ready (cons thread proc))
        thread ))
    (define (make-daughter daughter)
      (let ((cell (cons daughter daughters)))
        (lambda (msg)
          (case msg ···
            ((detach) (set-car! cell #f))
            ··· ))))
    ···
    (define (task msg)
      (case msg ···
            ((make-thread) make-thread)
            ((make-daughter) make-daughter)
            ··· ))
    task )
```

この手続きの中にスレッド生成手続きも含まれている. 第 2.2節で示したプログラム例のように, タスクオブジェクトは閉包によって表現されている. 新しいタスクが生成されると, 親タスクに対して様々な要求を行うための手続き to-mother を親タスクから得る. アプリケーションと子タスクに対して必要な情報だけに参照を限定するためにこのような実現手法を採用している (図 1).

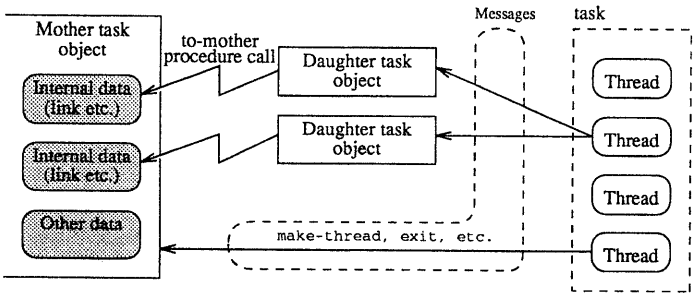

図 1

4.2 ディスパッチャの実現

ディスパッチャが持つべき機能は, 実行可能なスレッドの記録と, ディスパッチである. そのためには実行可能なスレッドを記録するキューを保持しなくてはならない. キューは他のプログラムから参照されるべきではないので, 閉包を用いてキューを他のプログラムから隠蔽している.

```
(define (make-dispatcher)
  (let (; レディキュー
        (q (make-queue))
        ; 現在実行中のスレッド
        (cur #f)
        ...)
    ...
    (define (make-ready th-and-k)
      ((q 'enq) th-and-k))
    (define (dispatch)
      (let ((th-and-k (q 'deq)))
        (set! cur (car th-and-k))
        ; スレッドの状態を復元・実行再開
        ((cdr th-and-k) #f) ))
    (lambda (msg)
      (case msg ...
        ((make-ready) make-ready)
        ((current-thread) cur)
        ((dispatch) (dispatch))
        ... ))))
```

```
(define the-dispatcher (make-dispatcher))
```

mutex にたいして行われる操作は, lock と unlock の二つである. すでに lock されている mutex を lock しようとすると, スレッドはブロックされ, mutex 中のキューに連結された後にディスパッチャが起動される. mutex もディスパッチャ同様, 閉包で表現している. mutex の実現の概略を以下に示す.

```
(define (make-mutex)
  (let (; mutex の状態
        (mutex #f)
        ; unlock を待っているスレッドのキュー
        (q (make-queue))
        ... )
    (define (lock)
      (cond (mutex (set! mutex #t))
            (else
             (call/cc
              (lambda (k)
                ((q 'enq)
                 (cons (the-dispatcher
                        'current-thread) k))
                (the-dispatcher 'dispatch)))
             (lock))))
    (define (unlock)
      (let ((th-and-k (q 'deq)))
        (if 待っているスレッドがあった
```

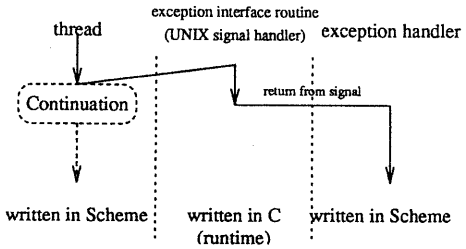

図 2　例外処理

```
        (make-ready th-and-k)
        (set! mutex #f) )))
(lambda (msg)
  (case msg ···
    ((lock) lock)
    ((unlock) unlock)
    ··· ))))
```

4.3　非同期的な事象によるプリエンプション

Scheme は非同期的に発生する例外事象を処理する機構を持たないため, プリエンプションを実現するためには処理系を拡張しなくてはならない. そこで, 例外発生時にその時点の継続を call/cc によって得た後に例外処理手続きに制御を移す機能を処理系に加えた. 継続を例外処理手続きに渡すことによって, 例外処理を終えた後に実行を再開するか, あるいは他の処理を行うかを例外処理手続きが自由に選択することができる.

例外処理の概要を図 2に示す. 例外が発生すると, 制御は例外インタフェースルーチンに移行する. 現在, 処理系は UNIX 上で実現されており, このルーチンは UNIX のシグナルハンドラとして実現されている. このルーチンに制御が移行すると, 例外が発生した時点の継続を表現するエスケープ手続きを構築し, Scheme で記述された例外処理手続きに制御を移す. この時, エスケープ手続きが引数として渡される. 例外処理手続きは, 必要な処理を行った後にこの継続を起動することによって元の処理に復帰してもよいし, 次のプログラムのように他の処理を行ってもよい.

```
(define (preempt-by-timer k)
  (make-ready k)
  (dispatch) )
```

この手続きをタイマ割り込みが発生する度に実行することによって, タイマによるプリエンプションを実現することができる.

処理が他の部分に移行してはいけない処理がある. 例えば, ディスパッチャの実行中やメモリを割り当て中にプリエンプションが行われては, 正しく実行を続けることができない. そこで, 特定の処理を行っている間はプリエンプションを禁止する機構が必要となる. システムコールによってシグナルを禁止することによって実現することも可能であるが, メモリ割り当てのように頻繁に実行される部分でシステムコールを行うことは非現実的である. そのため, 次に示す方法で実現している.

まず, 例外インタフェースルーチンは次に示す構造となる.

```
int pending_signals = 0;
int critical = 最小の整数;
exception_interface_routine()
{
  pending_signals |= シグナルの種類に応じた値;
  if (critical > 最小の整数) {
    --critical; return;
  }
  引数に例外発生時点の継続を設定
  例外処理手続きへジャンプする
}
```

プリエンプション禁止区域では, コンパイラは次に示すコードを生成する.

```
++critical;
プリエンプション禁止区域内のコード
--critical;
```

```
if (オーバーフロー発生) {
  現時点の継続を引数として，例外処理手続きへジャンプする;
}
```

これと同様の機能がReppyによってMLで実現されている[7]. Reppyの手法はメモリ割り当てが行われるまで, 割り込み処理が延期されるという問題があったのに対して, 本論文の手法にはこのような問題はない. また, SPARC上での実現では, プリエンプション禁止区域の前後にそれぞれ機械語1命令を加えるだけであり, オーバヘッドが小さい.

5 まとめ

本論文では, 継続をプロセスの状態の表現に用いることができること, そして, 継続を陽に操作する機能を持つ言語でプロセス管理プログラムが簡潔に記述できることを示した. また, 閉包を用いて情報の隠蔽を行うことによって, 特別な保護機構を用いずに, プロセス管理プログラム内の情報を保護できることを示した.

実現したプロセス管理プログラムは, 制御の流れであるスレッドと, 関連のあるスレッド群の実行を制御するタスクを持っている. 従来のOSのタスクはスレッド群の実行を制御する機能と保護情報を持つ重いものであった. 従って, スレッド群の実行を制御する目的に使用することが難しい場合があった. これに比べて本プロセス管理プログラムでは, そのような場合にもタスクを用いて制御を行うことが可能となっている.

今後の課題としては共有メモリ型並列計算機上での実現と, 分散環境での実現手法の検討がある.

参考文献

[1] William Clinger and Jonathan Rees (Editors), Revised^4 Report on the Algorithmic Language Scheme, LISP Pointers, Volume IV, Number 3, July-September 1991.

[2] Eric C. Cooper and Richard P. Draves, C Threads, Technical Report CMU-CS-88-154, School of Computer Science, Carnegie Mellon University, May 1990.

[3] Eric C. Cooper and J. Gregory Morrisett, Adding Threads to Standard ML, Technical Report CMU-CS-90-186, School of Computer Science, Carnegie Mellon University, December 1990.

[4] Christopher T. Haynes and Daniel P. Friedman, Embedding Continuations in Procedual Objects, ACM Transactions on Programming Languages and Systems, Vol.9, No.4, October 1987.

[5] Norman Ramsey, Concurrent Programming in ML, Technical Report CS-TR-262-90, Department of Computer Science, Princeton University, April 1990.

[6] John H. Reppy, First-Class Synchronous Operations in Standard ML, Technical Report TR-89-1068, Department of Computer Science, Cornell University, December 1989.

[7] John H. Reppy, Asynchronous Signals in Standard ML, Technical Report TR-90-1144, Department of Computer Science, Cornell University, August 1990.

[8] Mitchell Wand, Continuation-based Multiprocessing, *Proceedings of the 1980 LISP Conference*, pages 19–28, 1980.