記 号 処 理 67－3
（ 1 9 9 3. 1. 2 2 ）

futureベースの並列Schemeにおける継続の拡張

小宮常康　湯淺太一
豊橋技術科学大学 情報工学系

継続とはある時点以降の残りの計算を表したものである．Lispの一方言であるSchemeでは，継続を生成する関数が用意されており，継続をデータとして扱うことができる．これによって，非局所的脱出，コルーチンなどの様々な制御構造を実現することができる．

一方，Scheme言語を並列化するためによく使用されるfuture構文は，プロセスの生成とそれらの間の同期を取るメカニズムを提供する．しかし，future構文に基づく並列環境において継続を使用すると問題が生じることが知られている．

本報告書では，まず継続とfuture構文によって生じる問題を示す．そしてfuture構文に基づく並列環境に適合するように，Schemeの継続機能を拡張することを提案する．

Extended continuations for future-base parallel Scheme languages

Tsuneyasu KOMIYA and Taiichi YUASA
Department of Information and Computer Sciences
Toyohashi University of Technology, Toyohashi 441, Japan

A continuation represents the rest of a computation from a given point in the computation. Scheme, a dialect of Lisp, provides a function to generate a continuation as a first class object. Using Scheme continuations, we can describe various control structures such as non-local exits and coroutines.

The `future` construct, which provides the mechanisms of process creation and synchronization, is supported in many parallel Scheme languages. However, the use of continuations in parallel environment with the `future` construct causes problems.

This paper presents these problems, and proposes an extension to Scheme continuations to make them adaptable to parallel environment with the `future` construct.

1 序論

継続とはある時点以降の残りの計算を表したものである．Lispの一方言であるScheme[2]では，継続を生成する関数が用意されており，継続をデータとして扱うことができる．これによって，非局所的脱出，コルーチンなどの様々な制御構造を実現することができる．

一方，future構文は，プロセスの生成とそれらの間の同期を取るメカニズムを提供する．future構文は，逐次型のプログラムの意味を変えずに容易に並列実行可能なプログラムにすることができるという特徴がある．future構文をベースにした並列Schemeには，MultiScheme[4]，Multilisp[5]，PaiLisp[6]などがある．

しかし継続とfutureを併用した場合，MultiSchemeとMultilispでは逐次型Schemeの継続とは異なる意味となり，複雑な動作をする．またPaiLispでは逐次型Schemeの継続とは異なる意味を継続に与えている．

本報告書では，まず継続とfuture構文によって生じる問題を示す．そして逐次型Schemeの継続と同じ意味になる継続を実現するためにSchemeの継続機能を拡張することを提案する．

2 継続

継続とはある時点以降の残りの計算を表したものである．例えば，

```
(+ 1 (* 2 3))
```

で(* 2 3)の評価時には，評価が終われば結果に1を足すという継続が存在する．このように継続はプログラムの実行を制御するのに欠かせない概念であり，どのプログラミング言語にもこの概念は存在する．Schemeでは，継続を生成するための関数call-with-current-continuation (以後，省略形のcall/ccを用いる)が用意されており，継続をデータとして扱うことができる．これによって，非局所的脱出，コルーチンなどの様々な制御構造を実現することができる．

継続の生成は

```
(call/cc f)
```

という形で行う．この式を評価すると，この式を評価した後の継続を生成し，その継続を引数として1引数の関数fを呼び出す．そして関数fからの返値がcall/cc式の値となる．Schemeにおける継続は1引数の関数として実現されている．これを呼び出すと継続に与えられた引数の値を，その継続を作ったcall/cc式の値としてcall/cc式以降の評価を続ける．例えば，上式の関数fの評価中に継続が呼び出されるとfの評価を直ちに中断し，継続への引数をcall/cc式の値としてcall/cc式以降の評価を続ける．

次に例を示す．

```
(call/cc
  (lambda (exit)
    (for-each (lambda (x)
                (if (negative? x)
                    (exit x)))
              '(54 0 37 -3 245 19))
    #t))
⇒ -3
```

この式は与えられたリストの先頭から値を見ていって，負の値がないかを探すものである．そしてもし見つかればリストの探索を中断して見つかった値を返す．この式はまず始めに継続を生成する．そしてリストを探索し，負の値が見つかるとそれを引数にして継続を呼び出す．すると，リストの探索は中断され，見つかった値をcall/cc式の値として返す．

```
(define x 1)
(set! x (+ (call/cc (lambda (c)
                      (set! cc c)
                      2))
           x))
x ⇒ 3
```

この例では，call/cc式によって生成される継続は変数ccに代入される．この継続は，結果の値をxに加えるというものである．例えば，上の式を評価した後に(cc 5)を評価するとxの値は8となる．

3 Future

future構文は，プロセスの生成とそれらの間の同期を取るメカニズムを提供する．future構文の形式は

(future *exp*)

で，*exp*は任意の式である．この式を評価すると，(future *exp*) はpromiseと呼ばれるオブジェクトを直ちに返し，*exp*を評価するためのプロセスを生成する．このとき(future *exp*)を評価するプロセスを親プロセス，生成されるプロセスを子プロセスと呼ぶことにする．*exp*の値が得られると，その値はpromiseと置き換わり，子プロセスの実行は終了する．この動作を“promiseの値を決定する”と呼ぶことにする．(future *exp*) を評価した親プロセスは，*exp*の値が必要となるまで子プロセスの終了を待たずに実行を続ける．親プロセスが*exp*の値を必要としたときは，子プロセスが*exp*の評価を終了するまで親プロセスはサスペンドされる．promiseの値が決定するのを待つ操作は処理系によって自動的に行われる．ユーザによって明示的にpromiseの値を決定しその値を得たい場合はtouchを使用する．

(touch *prom*)

において*prom*が未決定のpromiseならば，値が決定されるまで待ち，その値を返す．そうでなければ，*prom*の決定した値を返す．

次に (future *exp*) の意味について考える．*exp*が副作用を含まなければ (future *exp*) の意味は，*exp*と同じでありどちらも同じ結果を返す．従って，副作用のない逐次型のプログラム (futureを使用しないプログラム) 中の任意の式*exp*を (future *exp*) で置き換えることにより，意味を変えずに並列実行可能なプログラムにすることができる．副作用がある場合は，副作用の起こる順序が一定でないため逐次型のプログラムと同じ結果が得られるとは限らない．しかし副作用のないプログラムでも，継続を使用したプログラムでは次章で述べるような問題が生じる．

4 並列Schemeの継続

継続とfuture構文を併用した場合，継続を生成したプロセスとは異なるプロセスで継続を呼び出した場合に問題が生じる．まず逐次型Schemeの継続について考える (図1)．図1の実線は制御の流れ，破線は継続の呼び出しを表している．図1の動作は次の通りである：

1. まず (call/cc f) を呼び出すと生成された継続を引数にしてfを呼び出す．
2. そしてfの評価を終えると制御はcall/cc式の直後Aに移る．
3. 次にBにおいて継続を呼び出すと制御はcall/cc式の直後Aに移る．

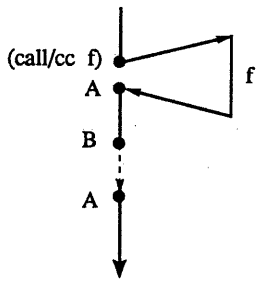

図 1: 逐次型Schemeの継続

次に図1の継続を生成する部分をfutureによって並列に行うことを考える．このとき逐次型Schemeの継続と同じ意味を持つためには図2のように動作しなければならない (ただし継続の評価は，継続を呼び出したプロセスによって行われるとする)：

1. futureによって子プロセスを生成し，その中で (call/cc f) を呼び出す．
2. fの評価が終わると制御はcall/cc式の直後Aに移り，その後future式の直後Cに移る．
3. 次にBにおいて継続を呼び出すと制御はcall/cc式の直後Aに移り，その後future式の直後Cに移る．

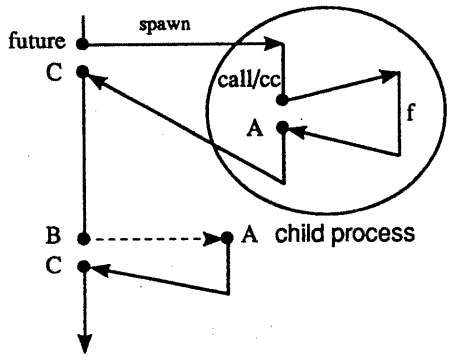

図 2: 逐次型の継続と同じ意味を持つ並列 Scheme の継続

しかし，MultiScheme，Multilisp，PaiLisp などの代表的な並列 Scheme では図 2のように動作せずに図 3のように動作する (ただし PaiLisp では，継続の評価は継続を生成したプロセスによって行なわれる)．継続の評価で図 2との違いは，A に制御が

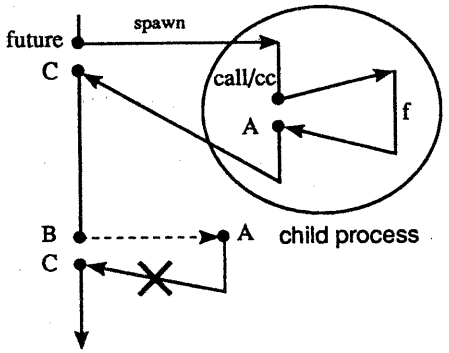

図 3: 逐次型の継続とは異なる動作をする並列 Scheme の継続

移った後，C に制御が移らない点である．その理由は，どの処理系も継続を生成したプロセスの情報しか promise に保存しないためである．

5 継続の拡張

前章で述べた問題は，図 2の C に移るための情報が継続に含まれていないために生じる．そこで，制御を C に移せるように future 構文を次のように拡張する方法が考えられる:

1. まず，future 構文の評価時にその時点における継続を生成し (これを親プロセスの継続と呼ぶことにする)，子プロセスに渡す．

2. そして子プロセス側で継続を生成するときは，子プロセスが生成した継続に親プロセスの継続を結合する．

しかし，この方法ではいつでも親プロセスの継続を生成する．継続の生成は重い処理なので，この方法を用いると future 構文の処理が重くなるという欠点がある．また，従来の future 構文と上記の future 構文を状況に応じて使い分けることも考えられる．しかし，いつどこで継続が生成されるのかを正確に知ることができない限り，これら 2 つの future 構文が混在したプログラムを書くのは困難である．そこで，この問題に対処するために継続の呼び出し方法

(*cont arg*)

を次のように拡張する．

(*cont1 arg* [*cont2*])

ここで *cont1* は継続，*arg* は継続に渡す引数であり，通常の継続の呼び出しの *cont*，*arg* にそれぞれ対応する．*cont2* はオプショナル・パラメータで 1 引数の任意の関数である．この拡張された継続の呼び出しの意味は，(*cont1 arg*) を評価した後，その結果の値を *cont2* の引数として *cont2* を呼び出すというものである．この継続を使うことによって future に負担をかけずに逐次型の場合と同じ意味となる継続を実現することができる．そうするには図 4のようにする：

1. まず，子プロセスを生成する直前 D において継続を生成する．

2. future によって子プロセスを生成し，その中で (call/cc f) を呼び出す．

3. f の評価が終わると制御は call/cc 式の直後 A に移り，その後 future 式の直後 C に移る．

4. 次に B において継続を
(A に移る継続 <値> C に移る継続) という形で呼び出す．すると制御は call/cc 式の直後 A に移った後，D で生成した継続にしたがって future 式の直後 C に移る．

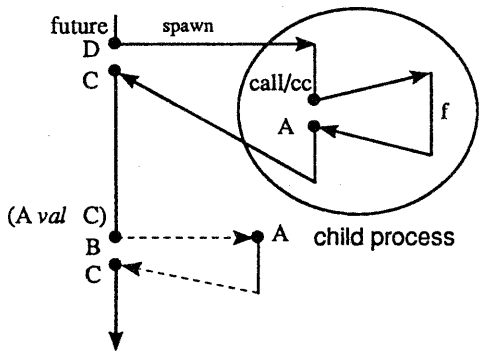

図 4: 拡張された継続

ここで拡張された継続の簡単な使用例を示す．次の式

```
(let ((v (future
           (call/cc (lambda (k) k)))))
  (if v (v #f) v))
```

は，もし future がなければ #f を返す．しかし MultiScheme, Multilisp, PaiLisp では逐次型のときと異なる動作をする [3]．この式を拡張された継続を使って逐次型と同じ意味とするには次のようにする．

```
(let ((v (call/cc (lambda (kk)
           (future
             (call/cc (lambda (k)
               (lambda (x) (k x kk)))))))))
  (if v (v #f) v))
```

6 例

この章では，拡張された継続の使用例としてミニマックス法によってゲームの勝率を並列に計算する例を示す．

ゲーム木とは 2 人のプレーヤーが交互に手を指すときの可能な手を表したものである．ゲーム木のノードは可能な局面を表し，枝は次の手を表している (図 5)．

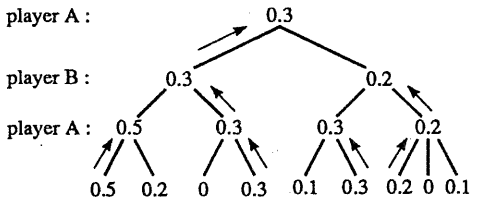

図 5: ゲーム木

ミニマックス法によってゲームの勝率を求めには，まず，ある局面に対してその局面の手番のプレーヤーが勝つ確率を見積もる評価関数を用意する．次にこの評価関数を使ってゲーム木の末端にあたる局面での勝率を求める．そして，A の手番である局面では最も勝率の高いものを選び，逆に B の手番である局面では最も勝率の低いものを選ぶことによって A の手番で始めた場合の A の勝率を求めることができる．

この計算の並列化は，次の局面の勝率を求めるときにプロセスを生成し，ゲーム木の各ノードごとに並列に勝率を求めることで行なう (図 6)．

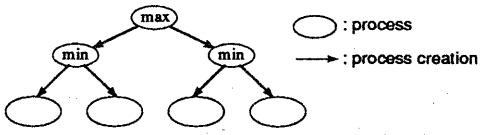

図 6: 勝率計算の並列化

勝率は次のような手順で求められる：

1. 持ち時間がなくなるまで先読みを行なって勝率を求める (図7の黒丸).

2. 制限時間であれば求めた勝率を返す．そうでなければ持ち時間を与えて先読みを再開し，勝率を求める (図7の白丸).

3. 制限時間が来るまで上のことを繰り返す.

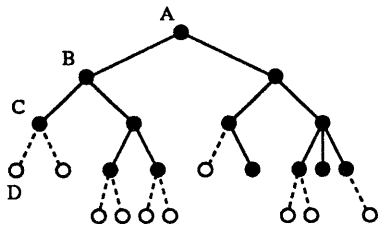

図 7: 先読みの再開

先読みの再開は，ゲーム木の末端で生成した継続を呼び出すことで行なう．もしここで，従来の継続を用いるとそれぞれのノードは別のプロセスで計算されるので求めた勝率を上のノードへ返すことができない．そこでプロセス生成時に継続を生成し，末端で生成した継続に結合して，求めた勝率を上のノードへ返せるようにする．例えば，図7のDのところで生成された継続には，A, B, Cのノードで生成した継続を

```
(D x (lambda (x)
       (C x (lambda (x)
              (B x (lambda (x)
                     (A x)))))))
```

のように結合する.

図8は，ミニマックス法によってゲームの勝率を並列に求めるプログラムである.

参考文献

[1] 湯淺太一：Scheme入門, 岩波書店, 1991.

[2] IEEE Standard for the Scheme Programming Language, the Institute of Electrical and Electronics Engineering, Inc., 1991.

[3] Halstead, R. : New Ideas in Parallel Lisp: Language Design, Implementation, and Programming Tools, *Lecture Notes in Computer Science 441*, Springer-Verlag, pp.2–57, 1990.

[4] Miller, J. : MultiScheme: A Parallel Processing System Based on MIT Scheme, TR-402, Laboratory for Computer Science, MIT, 1987.

[5] Halstead, R. : Multilisp: A Language for Concurrent Symbolic Computation, *ACM Transaction on Programming Languages and Systems*, Vol. 7, No. 4, pp.501–538, 1985.

[6] Ito, T. and Matsui, M. : A Parallel Lisp Language PaiLisp and Its Kernel Specification, *Lecture Notes in Computer Science 441*, Springer-Verlag, pp.58–100, 1990.

```
(define (estimate-my-turn-for-game f g-tree fuel some-fuel)
  (let ((p (estimate-my-turn f g-tree fuel (lambda (x) 'dummy))))
    (if *timeout*
        (car p)
        (begin (for-each (lambda (c) (future (c some-fuel))) (cddr p))
               ((cadr p) some-fuel)))))

(define (maxcar . lst)
  (do ((l (cdr lst) (cdr l))
       (maxnum (caar lst))
       (clist (cdar lst) (append clist (cdar l))))
      ((null? l) (cons maxnum clist))
    (if (> (caar l) maxnum) (set! maxnum (caar l)))))

(define (estimate-my-turn f g-tree fuel cont)
  (define retry '())
  (define args '())
  (set! fuel (call/cc (lambda (k) (set! retry (lambda (x) (k x cont))) fuel)))
  (cond ((empty? fuel) (list (f (car g-tree)) retry))
        ((null? (cdr g-tree)) (list (f (car g-tree)) retry))
        (else
          (call/cc (lambda (k)
            (set! args
                  (cons (future
                          (estimate-his-turn f (cadr g-tree) (1- fuel)
                                             (lambda (x) (k x cont))))
                        (map (lambda (gt)
                               (future
                                 (estimate-his-turn f gt (1- fuel) (lambda (x) x))))
                             (cddr g-tree))))))
          (apply maxcar args))))

(define (estimate-his-turn f g-tree fuel cont)
  (define retry '())
  (define args '())
  (set! fuel (call/cc (lambda (k) (set! retry (lambda (x) (k x cont))) fuel)))
  (cond ((empty? fuel) (list (- 1 (f (car g-tree))) retry))
        ((null? (cdr g-tree)) (list (- 1 (f (car g-tree))) retry))
        (else
          (call/cc (lambda (k)
            (set! args
                  (cons (future
                          (estimate-my-turn f (cadr g-tree) (1- fuel)
                                            (lambda (x) (k x cont))))
                        (map (lambda (gt)
                               (future
                                 (estimate-my-turn f gt (1- fuel) (lambda (x) x))))
                             (cddr g-tree))))))
          (apply mincar args))))
```

図 8: ミニマックス法による評価値の並列計算