Vol. 35 No. 11 情報処理学会論文誌 Nov. 1994

# 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†

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，非局所的脱出や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 構文に基づく従来の並列 Scheme の継続機能では，並列プログラムを記述するのに不十分であった．そこで本論文では，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-Based Parallel Scheme Languages

TSUNEYASU KOMIYA† and TAIICHI YUASA†

A continuation represents the rest of computation from a given point. Scheme, a dialect of Lisp, provides a function to generate a continuation as a first-class object. Using Scheme continuations, we can express 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, continuations of these languages are not powerful enough to describe parallel programs. This paper proposes an extension to Scheme continuations to make them adaptable to parallel environment based on the *future* construct.

## 1. はじめに

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[1),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，非局所的脱出や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とるメカニズムを提供する[3)]．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．この future 構文をベースにした代表的な並列 Scheme には，Multilisp[4)]，MultiScheme[5)]，PaiLisp[6)] などがある．

future 構文に基づく並列環境においては，継続の動作は定義されておらず，処理系によって異なる意味を継続に与えている．しかし Multilisp などの代表的な並列 Scheme の継続機能は，並列プログラムを記述するのに不十分であった．そこで本論文では並列プログラムに適した新しい継続機能を提案する．

本論文では，まず並列環境における継続の問題点について述べ，次に本論文で提案する継続について述べる．そして提案する継続の使用例をいくつかあげ，提案する継続が並列プログラムの記述に適していることを示す．最後に，従来の代表的な並列 Scheme の継続では並列プログラムを記述するのに十分な能力を持っていないことを示す．

## 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，

† 豊橋技術科学大学情報工学系
Department of Information and Computer Sciences, Toyohashi University of Technology

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 〈関数〉)

という形で行う．なお，本論文において〈A〉はその部分を評価した結果がAであることを意味し，《A》はAそのものが式に現れることを意味するものとする．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〈関数〉を呼び出す．そして〈関数〉からの返り値が 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〉の評価中に継続が呼び出されると〈関数〉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．

```
(let ((count 0)
      (x '(1 9 6 9 3 2)))
  (call/cc
    (lambda (return)
      (do () (#f)
        (if (null? x) (return count))
        (set! count (+ count 1))
        (set! x (cdr x))))))
⇒ 6
```

この式は与えられたリストの長さを求めるものである．この式はまず初めに継続を生成する．そして do 式の繰り返しによってリストの長さが得られるとその値を引数にして継続を呼び出す．すると，do 式の繰り返しは中断され，得られた値を 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《式》)

で，《式》は任意の式である．この式を評価すると，(future《式》) は promise と呼ばれるオブジェクトを直ちに返し，《式》を評価するためのプロセスを生成する．このとき (future《式》) 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．《式》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《式》) を評価した親プロセスは，《式》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《式》の値を必要としたときは，子プロセスが《式》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 〈promise〉)

において〈promise〉が決定的の promise ならば，値が決定されるまで待ち，値が決定されたときにその値を返す．そうでなければ，〈promise〉の決定した値をただちに返す．

次に (future《式》) の意味について考える．《式》が副作用を含まなければ (future《式》) の意味は，《式》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》を (future《式》) 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．

---

* ここでは，関数＋の呼出時の引数の評価は左から右へ行われるものとする．右から左へ評価する場合は，x の評価後に継続を生成するので，結果の値に1を足した値を x に代入するという継続になる．

## 4. 継続の拡張

### 4.1 並列環境における継続

各プロセスが継続を持つ並列環境においては，他のプロセスが生成した継続を実行し，その結果を受け取って以降の処理を続けることができるとよい．しかし，future 構文に基づく並列環境においては，継続を生成したプロセスとその継続を呼び出すプロセスが異なる場合の継続呼び出しの動作は定義されていない．それは，1)どのプロセスが継続を実行するか，2)継続が表す残りの計算とは何かが決まっていないためである．しかし，上で述べる並列環境に適合した継続を実現するには，これらを明確にしただけでは不十分である．ここでは代表的な並列 Scheme である Multilisp と MultiScheme を例としてあげ，どのような問題が生じるかについて述べる．

Multilisp[4] での継続の生成と実行の方法は逐次型 Scheme と同じである．すなわち，継続の実行はそれを呼び出したプロセスによって行われ，継続が表す残りの計算は，継続を生成したプロセスが保持する残りの計算（ある時点以降から現在のプロセスを終了するまでの計算）である．一方，future 構文 (future《式》) は

```
(let ((p (make-promise)))
  (fork (begin
          (determine! p《式》)
          (quit)))
  p)
```

という形で実現されている．ここで (make-promise) は未決定の promise を生成して返し，(quit) は現在のプロセスの実行を終了する．(fork $X$) は $X$ を評価するための子プロセスを生成する．このとき親プロセスは子プロセスの終了を待たずに実行を続ける．(determine! p《式》) は promise p を《式》の値に決定する．

Multilisp においてプロセスＢが生成した継続をプロセスＡが呼び出した場合について考える．プロセスＢは

```
(future《プロセスＢの処理》)
```

のように future によって生成されるプロセスである．プロセスＢが生成する継続は上に示す式が

```
(let ((p1 (make-promise)))
  (fork
    (begin
      (determine! p1《プロセスＢの処理》)
      (quit)))
  p1)
```

と等価であることから，プロセスＢの残りの計算を行った後，promise p1 をその結果の値で決定するという継続であることがわかる．この継続を実行するプロセスＡもプロセスＢと同様に future によって生成されるプロセスである．つまりプロセスＢが生成した継続は，次の式

```
(future《プロセスＡの処理》)
  ≡
(let ((p2 (make-promise)))
  (fork
    (begin
      (determine! p2《プロセスＡの処理》)
      (quit)))
  p2)
```

の《プロセスＡの処理》の中で呼び出される．この式の意味は，プロセスＡの処理を行い，promise p2 の値をその実行結果に決定するというものである．しかし《プロセスＡの処理》の中で呼び出される継続は，プロセスＢの残りの計算を行った後，promise p2 ではなく promise p1 を結果の値に決定した後，プロセスＡは終了してしまう．このため promise p2 は決定されることがなく，p2 の決定を待っているプロセスはいつまでも待たされる．このように Multilisp では，継続の実行結果を受け取って以降の処理を続けることができない．

一方，MultiScheme[5] では，future 構文 (future《式》) を

```
(let ((p (make-promise)))
  (fork (begin
          (set-process-waiting-for! p)
          (let ((value《式》))
            (determine! (process-goal) value))
          (quit)))
  p)
```

という形で実現している．ここで set-process-waiting-for! は現在のプロセスが決定する promise を設定する．(process-goal) は set-process-waiting-for! によって設定された現在のプロセスが決定すべき promise を返す．

MultiScheme では promise の決定の仕方が Multilisp とは異なる．Multilisp の継続では，プロセスＢ

の残りの計算を行った後，promise p1 を結果の値に決定したが，MultiScheme では，現在のプロセスが持つ promise，すなわち promise p2 を決定する．しかし，5.2 節で述べるように MultiScheme の継続でも継続の実行結果を受け取ることや以降の処理を続けることができない場合がある．

以上の問題はいずれの場合も，他のプロセスが生成した継続を実行すると，プロセスの終了の処理まで行ってしまうために起こる．そこで本論文では，継続の実行終了時の動作をユーザが指定できるように Scheme の継続機能を拡張することを提案する．

### 4.2 継続の拡張

前節の問題点を解決し，並列環境に継続を適合させるために本論文では，future の実現は Multilisp と同じものとし，継続の呼び出し方法

(〈継続〉〈データ〉)

を次のように拡張することを提案する．

(〈継続₁〉〈データ〉[〈継続₂〉])

ここで〈継続₁〉は継続，〈データ〉は継続に渡す引数であり，通常の継続の呼び出しの〈継続〉，〈データ〉にそれぞれ対応する．〈継続₂〉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．〈継続₂〉を与えないときは Multilisp の継続と同じ動作をする．この拡張された継続の呼び出しの意味は，(〈継続₁〉〈データ〉) を評価した後，その結果の値を引数として〈継続₂〉を呼び出すというものである．つまり〈継続₁〉の実行終了時の動作は，〈継続₂〉によって決まる．

この継続の簡単な使用例を示す．継続呼び出しは，通常，値を返さない．しかし，拡張された継続を使うことによって，あるプロセスが生成した継続を呼び出し，その結果を呼び出し側が受け取ることができる．例えば，

```
(define cont '())
(future
  (let ((x (call/cc
             (lambda (k)
               (set! cont k)
               1))))
    (* x 2)))
```

として与えられた値の 2 倍の数を返す継続を cont に代入した後，それを

```
(call/cc
  (lambda (k) (cont 10 k)))
```

として呼び出すと，この式は 20 を返す．この動作を図 1 に示す．図中の黒丸は継続を生成する点，白丸は継続を生成した call/cc 式の直後の点を表す．継続 cont の呼び出し直前に生成される継続 k は，結果の値を（k を生成した）call/cc 式の値としてそれ以降の処理を続けるというものである．この継続を cont の呼び出しのオプショナル・パラメータに与えることによって，cont の実行終了後，その結果を call/cc 式の値として返すことができる．

### 4.3 応用例 1：OR 並列

ここでは，並列アルゴリズムによくみられる OR 並列を拡張された継続によって実現する例を示す．

並列 Prolog[7] などで利用される OR 並列処理とは，ゴールを，単一化可能なすべての節（候補節）と並列に単一化を試みるものである．Prolog の処理の過程はユーザが入力したゴールを根とした OR 木で表現することができる．図 2 は Prolog プログラム

```
s :-t.    s :-u.
t :-v.    t :-w.
u :-x.    u :-y.
```

が与えられたときのゴール s に対する OR 木である．図 2 の各ノードは Prolog の節を表し，各ノードから出ている下向きの枝は，そのノードのゴールと単一化

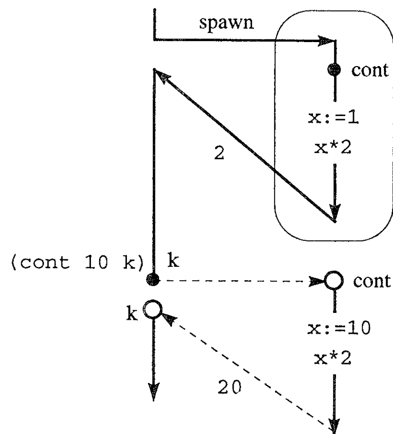



図 1 拡張された継続
Fig. 1 The extended continuation

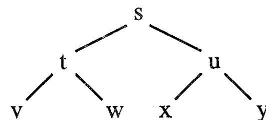



図 2 OR 木
Fig. 2 An OR tree.

可能な候補節を表している．候補となる節はOR関係にあることから並列に実行することができる．図2のOR木をプロセスに割り当てたプロセス木を**図3**に示す．図3の各プロセスは，現ノードのゴールと単一化可能な候補節があれば各候補節の実行を行うプロセスを生成する．そしてノードのゴールが空となったとき，その節の呼び出しが成功する．このときプロセスは現在の継続を生成して親プロセスに返すようにする．親プロセスはこの継続を並列に呼び出すことによって別解を並列に求めることができる．

継続によるOR並列処理はコルーチンの形で表現することができる．いま，いくつかのプロセスが定義されているとする．コルーチンでは**図4**のように，ある時点にはそのうちの1つのプロセスのみが実行中で，他のプロセスは中断している（図中の実線は制御の流れ，破線は制御の移行，同一のアルファベットは同じ継続点を表している）．制御の移行（図中の破線）は継続を呼び出すことによって行うことができる．例えば，プロセスAのa点で生成した継続をプロセスBが呼び出すことによってa点以降の処理を再開することができる．コルーチンの考え方は並列プログラムにおいても有効であるが，図4に示したコルーチンは並列性を持たない．そこでコルーチンに並列性を持たせるために**図5**のように複数のプロセスからなるプロセス・グループを1つのプロセスとみなしたコルーチンを考える．図5において，プロセスAからプロセス・グループBに制御を移すと複数の制御の流れ（プロセスB′とプロセスB″）が生じる．プロセス・グループBからプロセスAに制御を戻すと，プロセスAは中断していた計算を再開する．そして再びプロセスAからプロセス・グループBに制御を移すとプロセスB′とプロセスB″は中断していた計算を再開する．

この動作はメッセージ通信として考えることができる．プロセスAはfutureによってプロセス・グループBの継続を呼び出すことでメッセージ送信（プロセス・グループBからみればメッセージ受信）を行い，プロセス・グループBが返したpromiseの値が決定することでプロセス・グループBからのメッセージ受信（プロセスBからみればメッセージ送信）を行っている．

このようなコルーチンを実現するには，プロセスAからプロセス・グループBに制御を移した後，プロセスB′とプロセスB″の継続をプロセスAに返すことができればよい．しかし，一般にfutureベースの並列Schemeはメッセージ通信プリミティブを持たず，また従来の継続では，継続実行後にプロセスを終了してしまうため，プロセスB′とプロセスB″の継続をプロセスAに返すことができず，上で述べたメッセージ通信を実現することができない．そこで拡張された継続を用いてこれを実現する．以下では，このOR並列の具体例として8クイーンのすべての解を並列に求める例を示す．

チェスのクイーンは**図6**のように前後，左右，斜めのいずれかの方向に進むことができる．8クイーンとは，8つのクイーンを8×8のチェス盤上にどのクイーンも他のクイーンの進行を妨げないように配置す

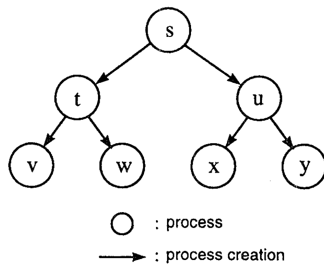



**図3** 図2のプロセス木
Fig. 3 The process tree for Fig. 2.

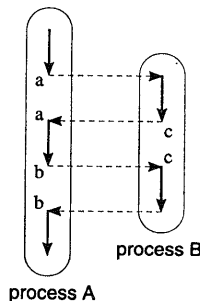



**図4** コルーチン
Fig. 4 Coroutines.

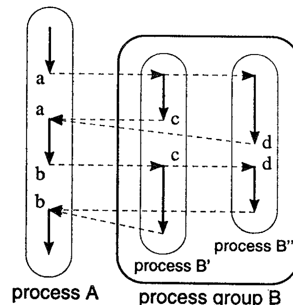



**図5** 並列性を持ったコルーチン
Fig. 5 Coroutines with parallelism.

る問題である．この問題には複数の解が存在する．**図7** はそのうちの1つである．

この問題の並列化は以下のように行う（**図8**参照）．

1. まず，8個のプロセスを生成する．各子プロセス $j$ では0行 $j$ 列にクイーンを置いた場合に残りの行に7個のクイーンを置くという処理を行う．
2. そして1つの解が見つかると，その解を親プロセスに渡し，子プロセスは終了する．

すべての解を求めるには以下のようにする．

1. 1つの解が得られた時点で子プロセスは継続を生成し，それを親プロセスに渡す．ここで生成される継続は次の解を求めるものである．
2. 親プロセスは，子プロセスから継続を受け取って並列に実行するということを繰り返す．

このとき次の解を求める継続を実行した後，新たに求められた解を親プロセスに渡す必要がある．そこで拡

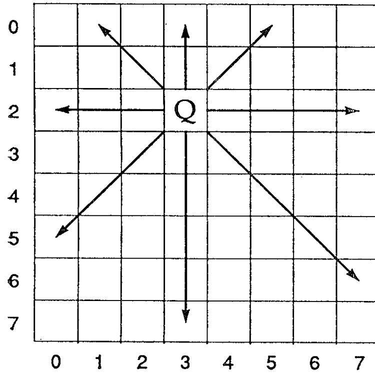



**図 6** クイーンの動き
Fig. 6 Movement of a queen.

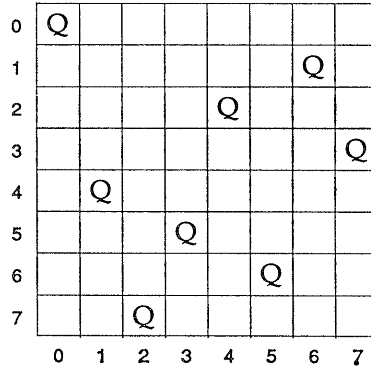



**図 7** 8クイーンの解の1つ
Fig. 7 A sample solution for the 8 queens problem.

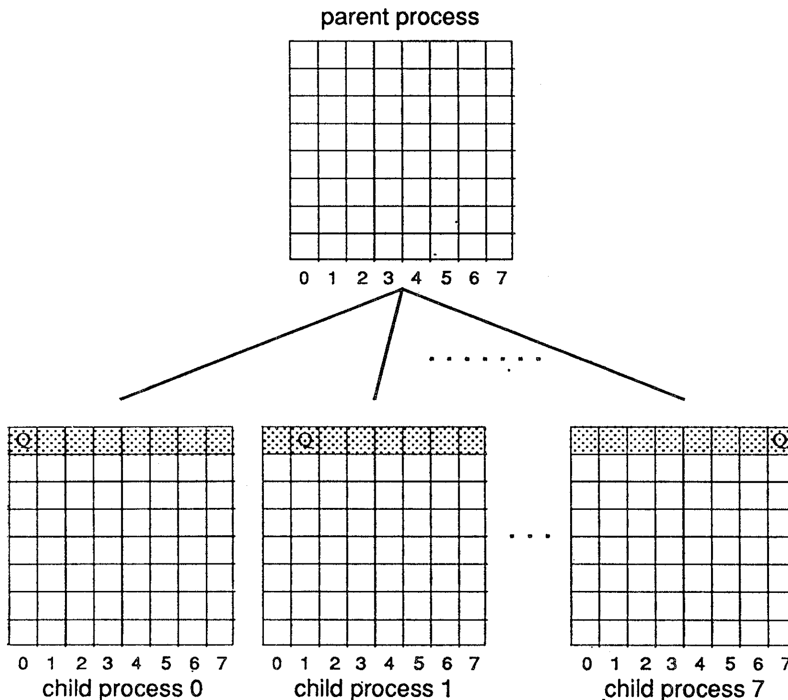



**図 8** 8クイーンの並列化
Fig. 8 Parallelization of the 8 queens problem.

張された継続を使って，親プロセスに戻ることにより，求めた解を親プロセスに渡すようにしている（**図 9** 参照）．

プログラムを付録に示す（ただし，ここで示すプログラムは単純さを優先し，効率については追求しない）．ここで１つの解と次の解を求めるための継続を返す関数 8queen は逐次型のものをそのまま使用している．

### 4.4 応用例 2：逐次型の継続と同じ意味を持つ継続

future をかぶせるだけで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のは future 構文の重要な特徴である．従って，逐次型の継続と同じ意味を持つ継続を実現できると便利である．しかし Multilisp や MultiScheme の継続は逐次型のときと異なる動作をする．ここでは Multilisp における future と継続との相互作用を説明するために次の例について考える[8]：

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

future を取り除くと，この式はまず call/cc によって生成された継続を v に束縛する．この継続は，継続の引数の値を v に束縛し，let 式の本体を評価する継続である．従って継続が呼び出されると v は #f に束縛され，#f を let 式の値として返す．一方，future が存在する場合の call/cc によって生成される継続は，determine! によって promise を継続への引数の値で決定し，quit によって現在のプロセスの実行を終了するという動作を行う．従って逐次型のときと異なる動作をする．

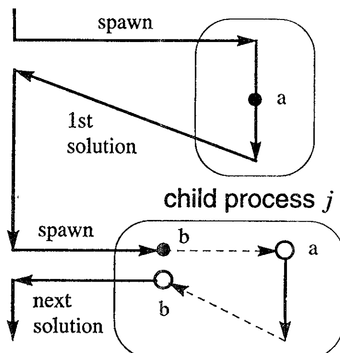



**図 9**　次の解を並列に求める方法
Fig. 9　Finding next solutions in parallel.

Katz と Weise は，逐次型 Scheme の継続と同じ意味になるような継続を提案している[9]．Katz らの継続では（future《式》）を

```
(call/cc
  (lambda (cont)
    (let ((p (make-promise))
          (first-return? #t))
      (fork (cont p))
      (let ((result 《式》))
        (if first-return?
            (begin (set! first-return? #f)
                   (determine! p result)
                   (quit))
            (cont result))))))
```

と実現している．この式が評価されると，future 式以降の評価を続ける継続を cont に束縛する．first-return? は１回目の《式》からのリターン時には #t，２回目以降の《式》からのリターン時には #f を値として持つ．１回目の《式》からのリターン時には Multilisp のときと同様に promise の値を決定し，現在のプロセスを終了する．しかし，継続を呼び出したことによって２回目の《式》からのリターンが生じたときは，《式》の値を引数として継続 cont が呼び出される．また次の式

```
(call/cc
  (lambda (abort)
    (let ((dummy (future (abort 0))))
      1)))
```

のように，複数の値（let 式の本体１と（abort 0）による０）を返す場合にも逐次型のときと同じ動作となるように Katz らは正当性（legitimacy）という属性を各プロセスに持たせている．プロセスは逐次型のプログラムによって実行される制御の流れと一致するときにのみ正当となり（つまり正当なプロセスは同時に複数存在することはない），正当なプロセスだけが値を返すことができる．従って，上の例では（abort 0）を評価するプロセスが正当となり，逐次型のときと同じ動作をする．しかし，Katz らの継続では継続を使用するしないに関わらず，future によってプロセスを生成するときには必ず継続を生成するため，future 構文の処理が重くなるという欠点がある．

本論文で提案した継続を使うことによって不必要な継続の生成をせ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．そうするには子プロセス

を生成する直前に親プロセスが継続を生成しておき，子プロセスが生成した継続を呼び出すときに

```
(〈子プロセスの継続〉
  〈値〉
  〈親プロセスの継続〉)
```

という形で呼び出せばよい．例えば上にあげた例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))
```

### 4.5 拡張された継続の実現

拡張された継続は以下のように実現されている．まずプロセスごとに継続を格納するためのスタックを用意する（以後，継続スタックと呼ぶことにする）．そして継続の実行の際，オプショナル・パラメータである継続が与えられていれば現在のプロセスの継続スタックにオプショナル・パラメータである継続を積む．future 式の本体の実行が終わり promise の値を決定した後，継続スタックが空でなければ継続スタックから継続を取り出して実行する．継続スタックが空の場合は子プロセスを終了する．継続スタックは，オプショナル・パラメータとして与えられた継続の実行に関する制御情報を保存する．この制御情報は制御スタックに保存することができない．なぜなら，ネストした拡張継続の呼び出しにおいて継続が呼び出されると，制御スタックの内容が継続生成時のスタックの内容に置き換わり，オプショナル・パラメータの継続の実行に関する制御情報が失われてしまうからである．

継続の生成は，制御スタックの内容と継続スタックの内容を保存する．そして，オプショナル・パラメータのない継続呼び出しでは，保存しておいた制御スタックの内容と継続スタックの内容の両方をそれぞれスタックに戻す．一方，オプショナル・パラメータが与えられた場合の継続呼び出し

(〈継続$_1$〉〈データ〉〈継続$_2$〉)

では，〈継続$_1$〉が保持する制御スタックの内容を戻し，〈継続$_2$〉を継続スタックに積む．

実行の様子を**図 10** に示す．ここで k1, k2, c2 は (k1 x c1) の評価を行うプロセスとは別のプロセスで生成された継続とする．まず，(k1 x c1) を評価すると c1 を継続スタックに積む．続いて (k2 x c2) を評価すると c2 を継続スタックに積む．継続 k2 の実行が終わると，継続スタックから c2 を取り出してそれを実行する．そして継続 c2 の実行が終わると，継続スタックから c1 を取り出してそれを実行する．

この拡張された継続は TUTScheme[10] をもとに開発された future ベースの並列 Scheme 上で実現されており，現在，Mach OS[11] 上のマルチプロセッサ・ワークステーションで稼働している．

## 5. 他の並列 Scheme との比較

4.1 節で述べたように複数の制御の流れが存在する並列環境においては，継続を生成したプロセスと，その継続を呼び出すプロセスが異なることがある．その場合の継続呼び出しの動作は処理系によって異なるが，従来の並列 Scheme 処理系の継続は，並列プログラムを記述するのに十分な能力を持っているとは言い難い．ここでは，代表的な並列 Scheme 処理系の継続および Katz と Weise が提案する継続[9]では図 5 に示す並列性を持ったコルーチンを記述することができないことを示す．

### 5.1 Multilisp

図 5 におけるプロセス A は，まず

```
(future《プロセス B′ の処理》)
(future《プロセス B″ の処理》)
```

を実行し，プロセス B′ とプロセス B″ を生成する．プロセス・グループが制御をプロセス A に戻すときには，プロセス B′ とプロセス B″ は継続を生成してそれを返り値として返す．このとき生成される継続は上に示す式が

```
(let ((p1 (make-promise)))
  (fork
    (begin
      (determine! p1《プロセス B′ の処理》)
      (quit)))
  p1)
```

と等価であることから，プロセス B′ の残りの計算を

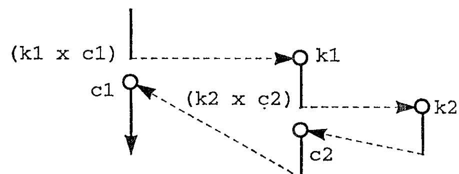



**図 10** 拡張された継続の実行の様子
Fig. 10 Control flow of extended continuation.

行った後，promise p1 をその結果の値で決定するという継続であることがわかる．次にプロセス・グループBの処理を再開するために，プロセスAはプロセス B′ とプロセス B″ の継続を

```
(future (〈プロセス B′ の継続〉〈値〉))
(future (〈プロセス B″ の継続〉〈値〉))
```

という形で呼び出す．この式は

```
(let ((p2 (make-promise)))
  (fork (begin
          (determine! p2
            (〈プロセス B′ の継続〉〈値〉))
          (quit)))
  p2)
```

と等価である．この式の意味は，プロセス B′ の継続を実行し，promise p2 の値を継続の実行結果に決定するというものである．しかしプロセス B′ の継続は，プロセス B′ の残りの計算を行った後，promise p2 ではなく promise p1 を結果の値に決定し，プロセスを終了する．このため promise p2 は決定されることがなく，p2 の決定を待っているプロセスAはいつまでも待たされる．従って，Multilisp では 4.3 節に示した問題を記述することができない．

### 5.2 MultiScheme

MultiScheme[5] では，プロセス B′ が生成する継続は，プロセス B′ の残りの計算を行った後，現在のプロセスが持つ promise をその結果の値で決定するというものである．プロセスAがこの継続を

```
(future (〈プロセス B′ の継続〉〈値〉))
```

として呼び出すと，生成されたプロセスはプロセス B′ の残りの計算を行った後，現在のプロセスが待つ promise，すなわち上の future 式が返した promise を決定する．従って，プロセスAはプロセス B′ の継続の実行結果を受け取ることができる．しかし，プロセスを生成せずにプロセスAが継続を

```
(〈プロセス B′ の継続〉〈値〉)
```

と呼び出すと，プロセスAが持つ promise が決定された後，プロセスAは quit によって終了するため，継続の実行結果を受け取ることができない．また，プロセス B′ の処理を並列化するために，プロセス B′ が子プロセスを生成してその中で継続を生成した場合，この継続は子プロセスとプロセス B′ の残りの計算を含んでいる必要がある．しかし MultiScheme の継続は，Multilisp と同様に継続を生成したプロセスの残りの計算のみであるため，このような並列化を行うことができない．

### 5.3 PaiLisp

PaiLisp[6] の継続はプロセスの制御を行うためのものである．PaiLisp では継続を生成したプロセスとは異なるプロセスによって継続が呼び出されると，継続を生成したプロセスはその実行を中断し，継続を実行する．ただし，継続を生成したプロセスがすでに終了している場合は何も行われない．いずれの場合も継続を呼び出したプロセスはその実行を続ける．

PaiLisp の継続では，プロセスからプロセス・グループへ制御を移すことはできるが，プロセス・グループから呼び出し元のプロセスへ複数の継続を返すことができない．このため，4.3 節に示した並列性を持ったコルーチンを実現することができない．

### 5.4 Katz と Weise の継続

副作用のない逐次型のプログラム中の任意の《式》を (future《式》) 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という future 構文の特徴を維持するという点では Katz らのアプローチは望ましい．しかし Katz らの継続が表す残りの計算は，継続を生成したプロセスの残りの計算だけではなく，その親プロセスの残りの計算までも含めたすべての残りの計算である．このため，図5において，プロセス B′ はプロセス B′ の残りの計算を終えた後，プロセスを終了せずにプロセスAの残りの計算を実行し続ける．従って 4.3 節の問題を記述することができない．

## 6. おわりに

本論文では，future ベースの並列 Scheme に適合するように，Scheme の継続機能を拡張することを提案した．この拡張によって，並列プログラムの記述能力を高めると同時に，future に負担をかけずに逐次型のときと同じ意味を持つ継続を実現することができる．

**謝辞** 有益なコメントをいただいた査読者に感謝いたします．

## 参考文献

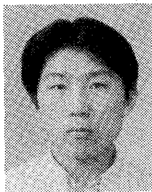
1) 湯淺太一：Scheme 入門，岩波書店 (1991).
2) IEEE Standard for the Scheme Programming Language, IEEE (1991).
3) Baker, H.: Actor Systems for Real-Time Computation, TR-197, Laboratory for Computer Science, MIT (1978).
4) Halstead, R.: Multilisp: A Language for Con-

current Symbolic Computation, *ACM Trans. Prog. Lang. Syst.*, Vol. 7, No. 4, pp. 501-538 (1985).
5) Miller, J.: MultiScheme: A Parallel Processing System Based on MIT Scheme, TR-402, Laboratory for Computer Science, MIT (1987).
6) Ito, T. and Matsui, M.: A Parallel Lisp Language PaiLisp and Its Kernel Specification, *Lecture Notes in Computer Science 441*, pp. 58-100, Springer-Verlag (1990).
7) Shapiro, E.: Concurrent Prolog: Collected Papers, Vol. 1-2, MIT Press (1987).
8) Halstead, R.: New Ideas in Parallel Lisp: Language Design, Implementation, and Programming Tools, *Lecture Notes in Computer Science 441*, pp. 2-57, Springer-Verlag (1990).
9) Katz, M. and Weise, D.: Continuing into the Future: On the Interaction of Futures and First-Class Continuations, *Proc. 1990 ACM Conference on Lisp and Functional Programming*, pp. 176-184 (1990).
10) 湯淺太一ほか：TUTScheme のマニュアル，豊橋技術科学大学湯淺研究室 (1992).
11) Walmer, L. and Thompson, M.: A Programmer's Guide to the Mach System Calls, CMU (1989).

## 付録　8クイーンのすべての解を並列に求めるプログラム

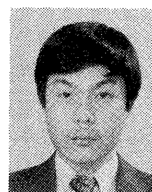
```
(define (all-queens)
  (define sols
    (map (lambda (j)
           (future (8queen 1 0
                     (cons j '())
                     (cons j '())
                     (cons (- j) '()))))
         '(0 1 2 3 4 5 6 7)))
  (call/cc
    (lambda (exit)
      (do ()
          ((not (solution-exists? sols)) (exit #f))
        (dolist (sol sols)
          (if sol
              (print-solution (cdr sol))))
        (set! sols
          (map (lambda (sol)
                 (if sol
                     (future
                       (call/cc
                         (lambda (k) ((car sol) #f k))))
                     #f))
               sols))))))

(define (8queen i j column left right)
  (cond ((= i 8)
         (call/cc (lambda (k) (cons k column))))
        ((= j 8) #f)
        ((or (member j column)
             (member (+ i j) left)
             (member (- i j) right))
         (8queen i (1+ j) column left right))
        ((8queen (1+ i) 0
           (cons j column)
           (cons (+ i j) left)
           (cons (- i j) right)))
        ((8queen i (1+ j) column left right))))
```

(平成6年1月13日受付)
(平成6年7月14日採録)



**小宮　常康**

1969年生．1989年育英工業高等専門学校電気工学科卒業．1991年豊橋技術科学大学工学部情報工学課程卒業．1993年同大学院工学研究科情報工学専攻修士課程修了．現在，同大学院工学研究科システム情報工学専攻博士課程に在学中．記号処理言語と並列プログラミング言語に興味を持ち，現在は並列記号処理言語に関する研究に従事．



**湯淺　太一**（正会員）

1952年神戸生．1977年京都大学理学部卒業．1979年同大学理学部修士課程修了．1982年同大学博士課程修了．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年同大学助教授となり現在に至る．理学博士．記号処理システムと並列処理に興味を持っている．著書「Common Lisp 入門(共著)」ほか．訳書「プログラミング言語 Turing(共著)」ほか．