Vol. 34 No. 12 情報処理学会論文誌 Dec. 1993

PaiLispの並列構文の実現法と評価

清野智弘[†] 伊藤貴康[†]

PaiLispはSchemeを基にした共有メモリ型マルチプロセッサ向きの並列LISP言語である．1986年にその初版が設計されて以来改良が加えられ，並列実行のための構文としてpcall, par-or, par-and, pcondなどの基本構文の並列版と，MultilispおよびQlispに由来するfuture, exlambdaを有している．Schemeのcall/ccによって取り出されるcontinuation（残りの計算）はエラー処理やコルーチンの実現に利用されているが，PaiLispではそれが拡張されており他のプロセスの実行を制御することができる．PaiLisp-Kernelは拡張されたcall/ccも含めて四つの基本的な並列構文をSchemeに加えたコンパクトな並列LISP言語であり，これによってPaiLispの他の並列構文の意味が与えられている．本論文ではPaiLisp-KernelをベースとしてPaiLispの並列構文をPaiLispの拡張されたcontinuationを用いて実現する方法を与え，この考えを基にマルチプロセッサAlliant FX/80上で試作したインタプリタによる評価結果について報告する．

On Implementations of Parallel Constructs of PaiLisp and Their Evaluations

TOMOHIRO SEINO[†] and TAKAYASU ITO[†]

PaiLisp is a Scheme-based parallel Lisp language for a shared memory multiprocessor. PaiLisp has many parallelized Lisp constructs such as pcall, par-and, par-or, pcond, etc, in addition to some basic concurrency constructs such as spawn, suspend, par, future and exlambda, where future is a construct introduced in Multilisp. One of the most important and interesting constructs of PaiLisp is the PaiLisp's call/cc which is an extension of Scheme's continuation into concurrency. After discussing how to implement PaiLisp call/cc we explain on implementations of PaiLisp constructs and PaiLisp interpreter. We show the performance of a PaiLisp interpreter implemented on a shared memory multiprocessor Alliant FX/80 with 8 processors. And using this parallel interpreter we evaluate the effects of PaiLisp constructs including future, par-and an PaiLisp call/cc.

1. はじめに

VLSI技術の進歩によりマルチプロセッサ・システムが開発・実用化され，その上での各種並列分散オペレーティング・システムも試作され，さまざまな用途に使われるようになってきている．また，人工知能や記号処理における，並列記号計算に関する研究も盛んに行われるようになり，代表的な記号処理言語であるLISPの並列処理に関する研究も行われている．

LISPプログラムを並列に実行させる方法として，並列性を自動的に抽出する方法[1),2)]と，ユーザが明示的に並列処理を指定する並列LISP言語を用いる方法がある．後者の試みとして，Multilisp[3)]，Qlisp[4)]および筆者らの研究室で設計されたPaiLisp[5)]などの並列LISP言語がある．

PaiLispは，LISPの一つの方言であるScheme[6),7)]に並列構文を導入し，プログラムの並列性を明示的に表せるようにした並列LISPであり，共有メモリ型マルチプロセッサに基づいて設計されている．関数引数の並列実行（pcall, eager），基本構文の並列実行（par, par-or, par-and, pcond, pcond#, pmapcar）を行う構文に加え，Multilispのfuture, Qlispの排他クロージャを取り入れた非常に強力な言語となっている．また，Schemeのcontinuationを並列に拡張したPaiLisp continuationと呼ばれている[8),9)]強力な制御構文を持つ．さらに，PaiLispの核言語PaiLisp-Kernelにより，並列構文の意味記述が与えられている[5)]という特徴を持つ．

本論文では，PaiLispの並列インタプリタを共有メモリ型マルチプロセッサ上で実現する場合の並列構文の実現法，特にcontinuationを生成するcall/cc, call/epそしてcontinuationを用いた他の構文の実現法を与える．これに基づいて，PaiLispのインタプリタを筆者らの研究室に設置されている共有メモリ型マルチプロセッサAlliant FX/80のCONCENTRIX

† 東北大学情報科学研究科
Graduate School of Information Sciences, Tohoku University

OS で試作・評価した結果について報告する．

2. PaiLisp と PaiLisp-Kernel

2.1 PaiLisp の概要

PaiLisp は LISP プログラムの並列実行に関する理論的考察[10),11)]および実験[2)]を基に筆者らの研究室で設計され，Multilisp および Qlisp の並列構文を参考として改良を加えられた共有メモリ型マルチプロセッサ向きの並列 LISP 言語である．PaiLisp の核言語 PaiLisp-Kernel は，Scheme に四つの並列構文を加えたコンパクトな並列 LISP 言語である．それらの詳細は文献5),12)に与えられているが，PaiLisp-Kernel と PaiLisp はおよそ次のようなものであると考えることができる．

PaiLisp-Kernel＝Scheme
　　＋{spawn, suspend, call/cc, exlambda}．
PaiLisp＝PaiLisp-Kernel
　　＋{par, pcall, future, eager, par-or, par-and,
　　　pcond, pcond#, pmapcar, signal, wait}．

PaiLisp-Kernel により PaiLisp のすべての並列構文の意味が記述できることが示されている[5)]．これらの構文のうち，call/cc が生成する continuation は並列に拡張されており PaiLisp continuation と呼ばれている[8)]．exlambda は Qlisp のプロセスクロージャを生成する qlambda を参考に設定されたもので，排他的関数クロージャを生成する．future は Multilisp に初めて導入された並列構文である．PaiLisp 全体のシンタックスを付録Aに与えた（構文の詳細については文献5)を参照）．

2.2 PaiLisp-Kernel

PaiLisp-Kernelの4つの基本的な並列構文(spawn, suspend, call/cc, exlambda) は次のような意味を持っている．

(spawn *e*)　　syntax☆

返す値は不定である．これ以降のプログラムの実行と並列に式 *e* の評価を行う．これを，式 *e* を評価するプロセスの生成と呼ぶ．式 *e* の評価の終了をプロセスの終了と呼ぶ．*e* の評価結果はどこにも残らない．

(suspend)　　procedure☆

引数をとらない関数である．この適用を行ったプロセスは停止する．そのプロセスの再開は continuation の呼び出しにより行われる．

(call/cc *proc*)　　procedure☆

PaiLisp continuation を生成し，それを引数にして *proc* の適用を行う．詳細は次章で述べる．PaiLisp continuation は PaiLisp-Kernel で PaiLisp の並列構文を記述するときに重要な役割を果たす．

(exlambda($x_1, \cdots, x_n$)$e_1 \cdots e_m$)　　syntax☆

排他的関数クロージャを作る．これは Scheme の (lambda($x_1 \cdots x_n$)$e_1 \cdots e_m$) で作られる関数クロージャに排他制御を行うためのキューを付け加えたものである．このクロージャは同時に2つ以上のプロセスの中で適用されることはない．

PaiLisp の並列構文の意味の詳細は文献5)に与えたが，付録Bで概要を説明した．

☆ PaiLisp の並列構文は "syntax" 的に解釈されるかまたは procedure (関数オブジェクト) として解釈されるかのどちらかである，suspend と call/cc は procedure として定義されている．

3. PaiLisp continuation

そもそも continuation は逐次計算プロセスのある時点の「残りの計算」を表し，大域脱出などを記述することができる強力な制御機構である．Scheme ではそれをファーストクラスのオブジェクトとして扱うことができ，エラー処理やコルーチンの実現に応用されている[13)]．PaiLisp continuation は，Scheme における continuation を並列に拡張したもので，自分以外のプロセスに影響を与えることができるという特徴を持つ．PaiLisp continuation を以降 P-continuation と呼ぶことにする．

3.1 Scheme continuation

スタックとレジスタを用いた計算に基づく Scheme のインタプリタは，例えば関数適用の実行では，まず関数の評価，次に引数の評価，最後に関数適用を行う．式の評価は，評価する式と評価するときの環境，さらに評価した後の行き先を適当なレジスタに入れて evaluator を呼ぶことにより行う．インタプリタでは式の評価によってどのレジスタが破壊されるかをあらかじめ知ることができないので，後で必要になるすべてのレジスタをスタックに退避しておかなければならない．したがって関数の評価の前には，関数適用後の行き先と，引数の評価に必要な環境と引数のリストを退避する．引数の評価の前には，評価された引数の値を蓄積するリストを退避し，評価すべき引数が複数個ある場合は，残りの引数の評価に必要な環境と残りの引数のリストも退避する．つまり，ある式の評価にその部分式の評価が含まれる場合，部分式を評価した後の「残りの計算」に必要なレジスタの内容をスタック

に退避してから部分式を評価する．したがって，スタックにはその時点の「残りの計算」に必要な情報がすべて入っていることになる．

Scheme の call/cc は一引数の関数を引数にとる関数である．適用時の「残りの計算」を表すスタックの内容を取り出し，これを基に一引数の関数である continuation を作る．そしてその continuation を call/cc の引数である一引数の関数に引数として渡す．つまり，次のように call/cc に一引数関数 f を与えると，

(call/cc $f(k)$)

k に continuation が代入されて関数 f が実行される．関数 f として (lambda (k)…) の形の lambda 式を書くことが多い．continuation の呼び出しは一引数の関数呼び出しと同様に次のように行う．

(k arg)

この呼び出しにより目下の「残りの計算」は捨てられ，continuation k が持つ「残りの計算」が行われる．このとき，arg の評価結果がその「残りの計算」へ返る．

図1の式をインタプリタへ入力した場合，call/cc を適用する直前のスタックの状態は**図2**のようになる．sp はスタックポインタである．作られる continuation はこのスタックの内容をすべてコピーしたものを持つ．その continuation は lambda で作られる関数の適用により変数 k に代入され，set! により大域変数 x に代入される．次に continuation の呼び出し (k 4) により，そのときのスタックの内容が捨てられて図2のスタックの状態が復元される．ここで k に与えた引数の値 4 が返されるので，あたかも call/cc が 4 を返したかのように (* 6 9 4 (* 4 5) 8) が計算される．一方大域変数 x に保存された「残りの計算」はその後何回でも実行することができ，例えば x を 100 に適用すると式 (* 6 9 100 (* 4 5) 8) が計算される．

```
(+ (* 2 3) 9
   (call/cc (lambda (k)
              (set! x k)
              (* 2 (k 4))))
   (* 4 5) 8)
```

図 1 call/cc を用いた式
Fig. 1 Use of call/cc.

	スタック
sp→	→ 引数リスト蓄積ルーチン
	→ 評価する残りの引数 ((* 4 5) 8)
	→ 評価する所の環境
	→ 評価された引数の値のリスト (9 6)
	→ +の評価結果
	→ トップレベルの print ルーチン

図 2 call/cc 実行時のスタックの内容
Fig. 2 Stack contents at the call/cc of Fig. 1.

3.2 PaiLisp の call/cc と P-continuation

P-continuation は，Scheme の continuation の並列計算プロセスへの拡張である．PaiLisp の call/cc は P-continuation を生成し，P-continuation の呼び出しには次のような効果がある．

- continuation を呼び出したプロセスは実行を続ける．呼び出しの値は不定である．
- continuation を生成したプロセスは実行を中断し，Scheme の場合と同様に「残りの計算」を行う．

PaiLisp の call/cc は，Scheme と同様にスタックの内容をコピーして continuation を生成する．さらに，その continuation に call/cc を実行したプロセスの名前を付け加えてどのプロセスの continuation であるかを識別できるようにしておく．これが P-continuation である．P-continuation を呼び出すと，その continuation を生成したプロセスがまだ終了していないならば，そのプロセスの現在の「残りの計算」は呼び出された continuation が表す「残りの計算」で置き換えられる．すでに終了していた場合は何も起こらず呼び出しは無効となる．P-continuation を呼び出したプロセスがその P-continuation を生成したプロセスそのものであった場合は Scheme の continuation 呼び出しの場合と同じ結果になる．

この P-continuation を用いることにより，他のプロセスの実行の流れを制御することができる．**図3**では P-continuation の二つの用法，停止したプロセスの再開とプロセスの終了を示している．spawn でプロセスを生成した後 (suspend) を実行して停止したプロセスは，(resume k) によって再開し，kill を k の値つまり P-continuation に束縛する．(spawn e) の e を評価した後の「残りの計算」は，プロセスを終了させることである．したがって (kill 'dummy) により，spawn で生成されたプロセスはその「残りの計算」

```
(let ((kill
        (call/cc (lambda (resume)
                   (spawn
                    (call/cc (lambda (k)
                               (resume k)
                               .....)))
                   (suspend)))))
  .....
  (kill 'dummy))
```

図 3 P-continuation の使用例
Fig. 3 Usage of P-continuation.

を行う．つまりそのプロセスは終了する．

3.3 call/ep

call/cc では「残りの計算」を表すスタックの内容をすべて別の場所へコピーし，continuation の呼び出しによってそれをスタックに戻していた．しかし，continuation が呼び出されたときの「残りの計算」に call/cc から値が返るような動作が含まれている場合，つまり大域脱出を行う場合は，call/cc の実行と continuation の呼び出しの処理を軽くすることができる．実際に大域脱出のためだけに call/cc を用いている応用例は多いので，そのための特別な関数 call/ep を考える．

図1の (k 4) を行うときのスタックの状態は**図4**のようになる．tag は call/cc を実行したときのスタックポインタの位置であり，そのときの「残りの計算」がそれより下に存在する．この「残りの計算」を始めるときにはスタックポインタを tag へ移動させるだけでよい．

call/ep はこの tag の位置が記録されている escape procedure というものを作る．escape procedure の呼び出しによりその tag の位置へスタックポインタが移動する．call/ep はスタックの内容をコピーしないので call/cc より軽く，escape procedure の呼び出しも continuation の呼び出しより軽くなる．call/ep が値を返した後は「残りの計算」を表すスタックの内容が保存されているとは限らないので，その call/ep が作った escape procedure を呼び出すとエラーになる．

call/ep で作られる escape procedure についても P-continuation と同様な拡張を考えることができる．そして P-continuation についても同様に，call/cc の代わりに call/ep を用いることによりコストを削減できる．これは，後に述べる実験で実証されている．また，PaiLisp-Kenrnel による PaiLisp の並列構文の記述の中で使われている call/cc はすべて call/ep で置き換えることができることがわかっている．

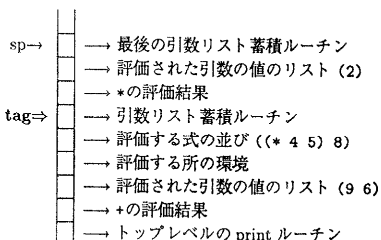

図 4 (k4) の実行時のスタックの内容
Fig. 4 Stack contents at the (k 4) of Fig. 1.

4. PaiLisp インタプリタの実現

インタプリタは，式を読み込む reader と式を出力する printer および式を評価する evaluator から成る．evaluator は，レジスタ計算機の命令を模擬するC言語のプログラムで実現する．レジスタ計算機は，汎用レジスタ5つ (exp, val, fun, argl, unev) と，環境レジスタ (env)，継続レジスタ (cont) および制御スタックを用いて計算を行う．レジスタ計算機を動かすプログラムの例として，evaluator の一部を**図5**に示す．評価する式を exp，評価する所の環境を env，評価した後に制御を移すアドレスを cont に入れて evaluator を呼び出すことにより式を評価する．評価結果は val に入る．

4.1 OS のプロセスと PaiLisp のプロセス

Alliant の CONCENTRIX OS は，プロセスを単位として並列実行を行う．OS のプロセスは fork システムコールで生成されるが，次のような欠点がある．

- データ領域を複写するため生成の処理が重い．
- コンテクストスイッチングの処理が重い．

PaiLisp のプロセスは，spawn や pcall などの構文により再帰的なプログラムの中で生成されることが多く，一般にプロセッサの数より多く生成される．また，その寿命も短いものが多い．したがって，OS のプロセスを PaiLisp のプロセスに対応させて並列処理を実現する方法はオーバヘッドが大きい．

そこで，OS のプロセスの生成とそのコンテクストスイッチングを避けるため，一つの OS のプロセスに一つのプロセッサを割り当てることにする．したがって，OS のプロセスは多くてもプロセッサの数しか存在しない．つまり，OS のプロセスとマシンのプロセ

```
int eval_dispatch(void){
    if self_evaluatingp(exp)
        go(ev_self_eval);
    if variablep(exp)
        go(ev_variable);
    if (length(exp) <= 0){
        val = exp;
        go(unknown_exp); }
    fun = car(exp);
    if syntaxp(fun)
        go(sym_syntax(fun));
    if macrop(fun)
        go(expand_macro);
    if (cdr(exp) == Nil)
        go(ev_no_args);
    go(ev_application); }
```

図 5 evaluator のプログラムの一部
Fig. 5 A part of program of the evaluator.

ッサを同一視できる状態になる．以降では，プロセスという言葉は PaiLisp のプロセスを意味するものとする．

PaiLisp のプロセスについては，生成するプロセスの数に制限を付けたくはない．ユーザがマシンのプロセッサ数を意識しないでプログラムを書けるように，ある程度自由にプロセスを生成できるようにしたい．そうすると，プロセッサがどのようにそれらを実行するかが問題となる．

一つには，複数のプロセスをラウンドロビン方式で処理する方法がある．プロセスを切り換える時期を，タイムスライスで制御するのではなく，evaluator から帰るときを選べば都合がよい．このとき必要なレジスタは val と cont だけであるからである．この方法は並列をシミュレーションするという意味ではふさわしい．また，P-continuation との相性がよく，切換えのときに continuation の呼び出しを調べて「残りの計算」を切り換えることができる．しかし，この方法はオーバヘッドが大きいと考えられる．したがって，プロセッサが一度割り当てられたら，プロセスが終了するか停止するまで実行するという方法を用いて PaiLisp のプロセスを処理することにする．

4.2 スタック領域

continuation を明示的に扱うため，マシンが使うスタックの他にスタック領域を用意する．ここでは用意するスタックの数が問題となる．

単一のスタックで処理する場合は，プロセスが終了した後を除いて，プロセスの切り換え時にスタックの内容を他の領域へ退避し，再開するプロセスのスタックの内容を入れる必要がある．スタックオーバフローは，書き込みが禁止されている領域へ書き込もうとしたときに自動的に検出される．このときはスタック領域を新たに追加することによって計算を続けることができる．

複数のスタックで処理する場合は，プロセスの切り換え時はスタックポインタを別のスタック領域へ移すだけでよい．ただし，プロセスの生成数は用意するスタックの数に制限される．また，複数のスタックが隣り合っているので push するたびにスタックオーバフローが起こったかどうかを検査する必要がある．オーバフローが起こったらそれ以上計算を続けることができないので実行を中止しなければならない．

call/ep の実行を軽くするため，後者の方法に従う．前者の方法を用いると，call/ep では call/cc と同様にスタックの内容をコピーしなければならないので，call/ep の利点が失われる．

4.3 PaiLisp のプロセス

PaiLisp においてプロセスとは共有された環境の下で共有データを処理する並列実行可能な一続きの計算のことをいう．PaiLisp プロセスは次のような属性を持つものとして定義されている[5]．

- プロセスの現在の値
- プロセスの状態
- プロセスの現在の「残りの計算」
- プロセスが現在使用している排他資源の情報
- プロセスの名前

PaiLisp-Kernel の並列構文はこのプロセスの生成・停止・再開・終了という基本的な操作およびプロセス間の排他制御を行うために用意されている．プロセスが持つ状態の遷移は**図 6** のようになる．

spawn で生成されたプロセスは suspend で停止し，continuation の呼び出しによって再開する (resume)．排他的関数クロージャが他のプロセスの中で使われていた場合は実行順番が来るのを待って (wait) 適用を行う (enter)．spawn の引数の評価が終了したならばプロセスは終了する (terminate) が，continuation の呼び出しによって強制的に終了させる (kill) こともできる．

4.3.1 PaiLisp プロセスの実現

本システムでは PaiLisp プロセスを**図 7** の構造を持つオブジェクトとして実現する．プロセスの現在の値は evaluator が返した値であり，これは val レジ

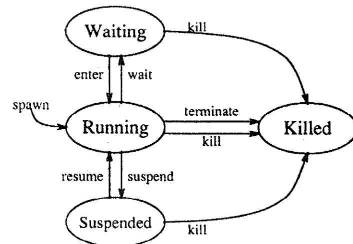

図 6 PaiLisp プロセスの状態遷移
Fig. 6 The state transitions of PaiLisp processes.

ポインタ	⟶ 使用している排他資源の情報
ポインタ	⟶ 割り当てられたスタック
ポインタ	⟶ continuation キュー
状態	
ロックバイト	

図 7 プロセスオブジェクト
Fig. 7 A process object.

スタに入っている．プロセスの現在の「残りの計算」はプロセスが一つずつ持つスタックの内容が表している．continuation キューは continuation の呼び出しを一つも漏らさないために用意する．プロセスの名前としてはプロセスオブジェクトへのポインタを用いる．排他資源の情報として現在使用している排他的関数クロージャの情報を持つ．これは signal と wait を実行するときに必要となる．また，排他的関数クロージャを適用しようとして waiting 状態になった場合に，どのクロージャを待っているのかを保持する必要がある．continuation によって阻止状態から解放されるときにそれを排他的関数クロージャへ知らせなければならないからである．また，プロセスオブジェクトに関する読み出しや書き込みを安全に行うため，ロックバイトを付加する．

状態は次の四つのフラグで実現する．

S (suspended) プロセスが停止していることを示すフラグ．

I (invoked) プロセスが生成した P-continuation の呼び出しがあることを示すフラグ．

W (waiting) 排他的関数クロージャの順番待ちを示すフラグ．

K (killed) プロセスが終了したことを示すフラグ．

continuation の呼び出しの処理は，呼び出されたプロセスの状態フラグによって変わる．waiting 状態ならば排他的関数クロージャ待ちから解放され，killed 状態ならばその呼び出しは無効になる．running, killed 状態でないならばプロセスを実行可能にする必要がある．通常，プロセッサはプロセスを停止するか終了するまで実行し続けるので，continuation が呼び出されたときに実行を中断させるため，Ｉフラグを設ける必要がある．これは continuation の呼び出しによって他のプロセスから立てられ，ＳフラグやＷフラグと両立する場合もある．プロセッサは evaluator から帰るときにこのフラグを調べ，continuation が呼び出されていたならば実行を中断しスタックを復元する．

4.3.2 プロセッサのプロセスへの割り当て

プロセッサの数が限られているために，実行可能な PaiLisp プロセスをすべて同時に走らせることはできない．そこで，プロセスがプロセッサの割り当てを待つための待ち行列を２種類用意している．新プロセスキューは，新たに生成されたプロセスが待つキューであり，プロセス間で共有されている．このキューは各プロセッサにできるだけ均等に計算を割り付けるため FIFO 方式にする．再開プロセスキューは停止しているプロセスが，continuation の呼び出しによって再開するときに待つ LIFO 方式のキューである．このキューはキュー操作に伴うオーバヘッドを小さくするためプロセッサごとに用意する．

プロセッサが現在実行しているプロセスが停止または終了したときは，次の順序でキューを調べ別のプロセスを実行する．

（1） プロセッサが持つ再開プロセスキューからプロセスオブジェクトを取り出してプロセスを再開する．このとき，呼び出されている continuation に従って制御スタックの内容を復元してから実行する．キューが空なら(2)へ行く．

（2） 新プロセスキューからプロセスオブジェクトを取り出し，スタック領域を割り当てて実行を開始する．キューが空なら(1)へ行く．

プロセスを過剰に生成しないようにするため，最初に再開プロセスキューを見に行き，再開するプロセスに優先的にプロセッサを割り当てる．

4.3.3 実行中の制御の切り替え

プロセスの実行の流れは，そのプロセスが生成した P-continuation が関数適用の形で呼び出されたときに変わる．つまり，現在の「残りの計算」が呼び出された P-continuation が持つ「残りの計算」に置き換えられる．特に「残りの計算」がプロセスの実行を終了させるようなものであったときは，その P-continuation を生成したプロセスはただちに終了する．したがって，P-continuation を用いて他のプロセスの流れを変えることができる．

P-continuation の呼び出しをそれを生成したプロセスに知らせるため，プロセスオブジェクトのＩフラグが立てられる．また，複数の呼び出しがあっても漏れがないように，プロセスオブジェクトには continuation キューが付いている．このキューは FIFO 方式であり先に呼ばれたものを先に処理する．プロセッサは evaluator から制御が帰るときにＩフラグを検査し，P-continuation の呼び出しを検出したならばプロセスを中断し次の処理を行う．

（1） プロセスオブジェクトの continuation キューの先頭から continuation または escape procedure を一つ取り出す．

（2） それに従いプロセスに割り当てられている制御スタックの内容を復元する．

（3） continuation キューが空になったら，Ｉフラグ

を下ろす．

この処理の後，復元された制御スタックを用いて実行を開始する．キューの中のどれかがプロセスを終了させるような continuation であった場合は，それ以降に並んでいる continuation は無効になる．continuation キューにまだ continuation がある場合は，Ｉフラグが立ったままであるので，evaluator からの次の復帰時に再度この処理が行われる．Ｉフラグが下ろされるのは，最後に呼び出された continuation が取り出されたときである．

制御スタックの内容の復元は次のように行う．

continuation の場合 コピーしておいた制御スタックの内容とスタックポインタの位置を戻す．

escape procedure の場合 現在のスタックポインタの位置から制御スタックの底までの間に，呼び出された escape procedure に記録されているスタックの位置があれば，そこにスタックポインタを移動する．見付からなければスタックポインタは移動しない．

escape procedure の場合，現在の「残りの計算」にその eacape procedure を生成した call/ep から値が返るという動作が含まれていない場合は，スタックポインタは動かないのでプロセスは元の実行を続ける．escape procedure にスタック上の位置がどのように記録されているかについては，後の call/ep の実現法で述べる．

4.4 PaiLisp-Kernel の実現

4.4.1 spawn の実現

(spawn *e*) を評価するときはそのときの環境で次の式を評価するプロセスを作る．つまり，式と環境を持った新しいプロセスオブジェクトを作り新プロセスキューに入れる．

(*term* *e*)

term☆ は引数を評価した後，プロセスを終了する構文である．

4.4.2 call/cc と continuation 呼び出しの実現

call/cc は一引数の関数であり，適用時の制御スタックの内容のコピーと call/cc を呼び出したプロセスの名前を持つ**図 8** の continuation オブジェクトを作る．制御スタックのコピーはリストに変換してヒープ領域に保存する．そしてこの continuation を引数として call/cc の引数の適用を行う．continuation が呼び出されなかった場合は，その適用の結果が call/cc の値として返る．

☆ “*” で囲まれたものはシステム内で使われる内部構文または内部変数である．

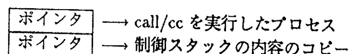

図 8 continuation オブジェクト
Fig. 8 A continuation object.

continuation の呼び出しは次の (1) から (4) までを行うことにより実現する．

(1) call/cc を実行したプロセスが終了状態ならば呼び出し処理を終了する．この呼び出しは無効になる．

(2) 生成したプロセスオブジェクトの continuation キューの末尾に continuation オブジェクトと引数の値のペアを追加する．

(3) Ｉフラグが立っていたらすでに実行可能状態であるので呼び出し処理を終了する．立っていなければＩフラグを立てる．

(4) Ｓフラグが立っていたらこのプロセスは suspended 状態である．実行可能状態にするため，再開プロセスキューにプロセスオブジェクトを入れプロセッサの割り当てを待たせる．

P-continuation を呼び出したプロセスとそれを生成したプロセスが同一である場合は，Scheme の場合の continuation 呼び出しの実現法と同様に直接制御スタックの復元を行うこともできる．この場合ただちに「残りの計算」を行うことができるが，continuation の呼び出しの度にプロセスが同一であるかどうかの判定を行う必要がある．

4.4.3 call/ep の実現

call/ep の適用ではタグオブジェクトというものを作り，制御スタックに push する．次にそのタグオブジェクトと call/ep を実行したプロセスの名前を持つ**図 9** の escape procedure オブジェクトを作る．制御スタック上の位置だけを記録しておく方法では，escape procedure の有効性の判定が煩雑になるのでタグオブジェクトを push する方法を採る．

escape procedure の呼び出しは，continuation の呼び出しと同様に行う．escape procedure が呼び出されずにそれを生成した call/ep から値が返るときはタグオブジェクトが pop される．そしてこれ以降の

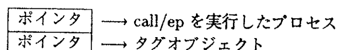

図 9 escape procedure オブジェクト
Fig. 9 An escape procedure object.

その escape procedure の呼び出しは continuation によって再度有効になる場合を除いて無効となる．

4.4.4 suspend の実現

suspend の適用では，プロセスオブジェクトの I フラグを検査し，continuation の呼び出しがなければ S フラグを立ててプロセスを停止する．continuation の呼び出しがあれば停止せずにそれが示す「残りの計算」を行う．

4.4.5 exlambda の実現

exlambda は (lambda($x_1 \cdots x_n$)$e_1 \cdots e_m$) で作られるクロージャと，FIFO 方式のキューおよび現在適用しているプロセスの名前を持つ排他的関数クロージャを作る．FIFO 方式のキューには排他的関数クロージャを適用しようとして阻止されたプロセスを再開させる escape procedure が入っている．Qlisp のプロセスクロージャはそれ固有のプロセスにより実行されるが，排他的関数クロージャは呼び出したプロセスにより適用される．

排他的関数クロージャが適用可能なのは，どのプロセスも適用していない場合であり，それ以外はプロセスを再開する escape procedure を排他的関数クロージャのキューの末尾へ追加した後 waiting 状態へ移る．プロセスの実行が排他的関数クロージャの適用の途中で continuation の呼び出しによって飛び出してしまうのを防ぐため，実際の適用は新たなプロセスオブジェクトを作って行う．

排他的関数クロージャが持つ escape procedure のキューはリスト構造になっており，waiting 状態のプロセスは自分を再開する escape procedure が指されているセルへのポインタを持つ．waiting 状態のときに continuation の呼び出しがあった場合は，そのセルのポインタを NULL にして自分を再開する escape procedure をキューから削除する．こうしてキューの排他制御を行わないで削除する．

関数適用が終了したならば，キューが空のときは現在適用しているプロセスの名前を取り消して排他的関数クロージャを解放する．空でないときは待っているプロセスのうち一つを，キューに入っている escape procedure を呼び出すことにより再開させる．

4.5 PaiLisp の並列構文の実現

PaiLisp の並列構文は PaiLisp-Kernel ですべて記述できるが，直接に実現する方法も与える．ここでは興味深い四つの並列構文について説明する．

4.5.1 future の実現

future は Multilisp に初めて導入されたものであり，Multilisp を特徴付ける並列構文である．(future *e*) は式 *e* を評価するプロセスを生成し，future 値と呼ばれる *e* の仮の値を返す．*e* の値の代わりに future 値を用いて計算ができる間は，その計算は *e* の評価と並列に実行される．例えば cons は引数が future 値であってもその値へのポインタを用いて計算できる．しかし car，＋などは引数が future 値のときは計算ができない．このときは実行を停止し *e* の値が求められたときに実行を再開する．この future 値の本当の値を求めることを force または touch という．暗黙的に force するのは以下の場合である．

- ＋や car などの引数の型が決まっている関数の適用．
- pair?，number? などの型を調べる関数の適用．
- 関数適用の関数の部分にある場合．
- eq? などの同値関係を調べる関数の適用．
- if などの条件文の述部を評価する場合．

また，force 関数により (force *e*) のように明示的に式 *e* の値を force することもできる．

(future *e*) の評価は次のように実現する．まず future 値として**図 10** の構造を持つ future オブジェクトを作る．これは future 値に対する force 操作により停止したプロセスを再開させる escape procedure のリストを持ち，さらに *e* の評価が終了した後はその *e* の値を持つ．

そして次の式を評価するプロセスを生成する．

(*det-future* *future_object e*)

det-future は *e* を評価し，その値を force したものを *future_object* に記録する．そして，この future 値を force して停止したプロセスを，escape procedure の呼び出しによりすべて再開する．

force は次のように行う．すでに本当の値が求められている場合は future オブジェクトに記録されているその値を返す．そうでないときはプロセスを再開する escape procedure を作り，それを eacape procedure リストに追加してプロセスを停止する．

4.5.2 pcall の実現

(pcall *operator operand*$_1 \cdots$*operand*$_n$)　　syntax

ポインタ	⟶ escape procedure のリスト
ポインタ	⟶ 引数の本当の値
ロックバイト	

図 10 future オブジェクト
Fig. 10 A future object.

pcall は関数適用において引数の評価を並列に行う構文である．まずプロセス間で共有される以下の環境を作る．

count オペランドの数を初期値として持つ

ep pcall を実行したプロセスを再開する escape procedure

この環境で次の式 ($i=1, \cdots, n$) を評価するプロセスを生成する．

(*det-pcall* *operand*$_i$)

これらのプロセスオブジェクトのリストを作った後，pcall を実行したプロセスを停止させる．

det-pcall は *operand*$_i$ を評価し，値をプロセスオブジェクトに記録する．そして *count* を一つ減らし 0 になったならば *ep* を呼び出す．これによって pcall を実行したプロセスは再開し，プロセスオブジェクトのリストを基に引数の値のリストを作り *operator* の適用を行う．

4.5.3 par-or の実現

(par-or *test*$_1$…*test*$_n$) syntax

par-or は or の実行において引数を並列に評価する構文で，最も早く計算された #f でない値を返す．まずプロセス間で共有される以下の環境を作る．

count 引数の数を初期値として持つ

pl プロセスオブジェクトのリスト

ep par-or を実行したプロセスを再開する escape procedure

この環境で次の式 ($i=1, \cdots, n$) を評価するプロセスを生成する．

(*det-par-or* *test*$_i$)

pl にこれらのプロセスオブジェクトのリストを作った後，par-or を実行したプロセスを停止させる．

det-par-or は *test*$_i$ を評価し *count* を一つ減らす．*test*$_i$ の値が #f でなければ，その値を引数にして *ep* を呼び出し，par-or の値としてその値を返す．このときはさらに *pl* 中の各プロセスに対し kill continuation を呼び出す．これはプロセスを終了させる「残りの計算」を持ち，システム内部でのみ使うことができる continuation で，これにより残りのプロセスを強制的に終了させる．*count* が 0 になったときは #f を引数にして *ep* を呼び出し，par-or の値として #f を返す．

4.5.4 pcond の実現

(pcond (*test*$_1$…)…(*test*$_n$…)) syntax

pcond は cond の実行において述語を並列に評価する構文であり，副作用がなければ cond と同じ結果になる．まずプロセス間で共有される以下の環境を作る．

pl プロセスオブジェクトのリスト

ep pcond を実行したプロセスを再開する escape procedure

この環境の下で次の式 ($i=1, \cdots, n$) を評価するプロセスを生成する．

(*det-pcond* *test*$_i$)

pl にこれらのプロセスオブジェクトのリストを作った後，pcond を実行したプロセスを停止させる．

det-pcond は *test*$_i$ を評価した値をプロセスオブジェクトに記録する．それから，*pl* 中のプロセスオブジェクトを先頭から順番に調べていき，まだ終了していないプロセスに出会う前に #f でない値で終了したプロセスに出会ったならば，*ep* を呼び出して pcond を実行したプロセスを再開する．そして *pl* 中でそれより後に並んでいるプロセスを，kill continuation を呼び出すことにより終了させる．再開したプロセスは，真の値になった *test* に対応する本体を実行する．

これ以外の構文については，par-and は par-or と同様に，delay は future と同様に，pcond# は pcond の方法を応用して実現できる．

5. PaiLisp インタプリタの評価

前章で述べた方法に従って，PaiLisp のインタプリタを共有メモリ型マルチプロセッサ Alliant FX/80 の CONCENTRIX OS 上にC言語を用いて実現した．FX/80 は 8 台の計算用プロセッサを有しており，インタプリタの実行に用いるプロセッサの数を指定して台数効果を見ることができる．

以下では PaiLisp の主要な並列構文のうち，並列処理効果を見る点で興味深い構文 pcall, future, par-or, par-and を用いた評価結果および call/cc と call/ep の比較を行った結果について報告する．

5.1 並列処理の効果

並列化したプログラムの実行時間を，使用するプロセッサ数を変えて計り並列処理の効果を調べる．用いたプログラムは次の五つである．

fib フィボナッチ数列の計算．

fib$(n-1)$ と *fib*$(n-2)$ を並列に計算するため，**図 11** に示すように future を一つ挿入する．

tarai タライ回し関数[14]．

tarai の引数を並列に計算するため，**図 12** に示す

```
(define (fib n)
  (if (< n 2)
      n
      (+ (future (fib (- n 1)))
                 (fib (- n 2)))))
```

図 11 フィボナッチ数列を計算するプログラム [I]
Fig. 11 Computing the Fibonacci sequence using future [I].

```
(define (tarai x y z)
  (cond ((> x y)
         (pcall tarai (tarai (- x 1) y z))
                      (tarai (- y 1) z x))
                      (tarai (- z 1) x y)))
        (else y)))
```

図 12 タライ回し関数
Fig. 12 The "tarai" function.

ように pcall を一つ挿入する．

sort 数え上げソート[15]．

各要素が何番目になるかを数える処理を並列に行うため，付録Cの図21に示すように要素数のプロセスを spawn で生成する．

tpu 定理証明プログラム[14] (Theorem Proving with Unit binary resolution)．

単節の導出を future によって並列に行う．

tsp☆ 巡回セールスマン問題 (Traveling Salesman Problem)[16]．

複数の都市とその都市間の距離が与えられているとき，ある都市から出発してすべての都市を1回ずつ周り出発した都市へ帰る行程のうち，最も道のりが短い行程を求める問題が巡回セールスマン問題である．

分枝限定法を用いて最適解を求める．部分解の展開を並列に行うため，par-or によりプロセスをプロセッサの個数だけ生成し展開順序を PaiLisp で書いたキューによって制御する．

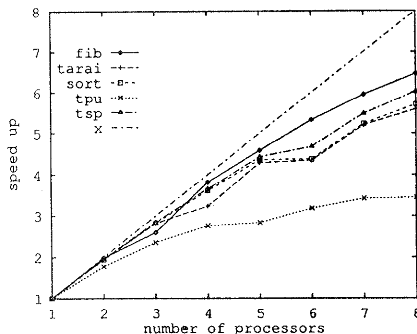

図 13 処理速度の向上
Fig. 13 Speedup.

☆ これは文献16)の qtrav に相当するプログラムである．

図 13 が並列化による処理速度の向上を示す．fib ではプロセッサが8個のときに 6.5 倍の処理速度が得られたが，tarai, sort では並列化効果が少し落ちる．これはプロセスの過剰生成が原因と思われる．tpu は，導出した節がすでに存在するかどうかを調べる包含検査を逐次的に行う必要があり，そのため十分に並列化されていないので台数効果が上がっていない．

5.2 future と pcall の比較

future と pcall で並列化したプログラムの実行時間の比較を，生成する総プロセス数が等しい，という条件の下で行う．次のように，プログラム中の主要な関数適用

($operator$ $operand_1$ $operand_2$…)

を future を用いて

($operator$ (future $operand_1$)(future $operand_2$)…)

のように並列化する．また pcall を用いて

(pcall $operator$ $operand_1$ $operand_2$…)

のように並列化する．

前に述べた fib, tarai のプログラムについてこの並列化を行う．並列化する関数適用として fib では＋を選び，tarai では外側の tarai を選ぶ．プロセッサ数を変えて実行時間を計ったものを**図 14** に示す．

fib では $operator$ が＋であり引数を暗黙に force するので，最悪の場合引数の数だけ実行の停止と再開を繰り返す．したがって実行の停止が1回だけである pcall を用いた場合の方がオーバヘッドが小さい．

tarai 関数は $operator$ が tarai である．tarai に渡された引数のうち変数 z が束縛される第3引数は (> x y) が偽のときは全く force されない．したがって，future で生成されたすべてのプロセスが終了していなくても tarai は値を返すことができ，pcall を用いる

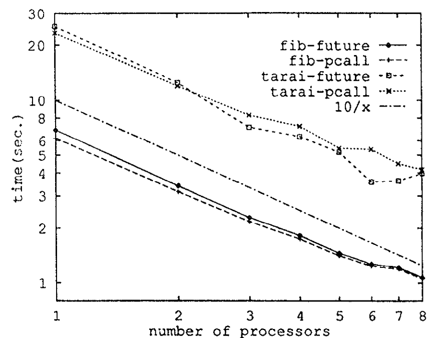

図 14 future, pcall を用いたプログラムの比較
Fig. 14 Comparison of programs using future or pcall.

よりも応答が速くなる．

5.3 par-and の評価

木構造を持った二つのリストが等しいかどうかを判定することを考える．ここでは，リスト中に現れるリスト以外の要素はシンボルのみとし，二つのリストが等しいとは，リスト構造中の同じ位置に同じシンボルがあることであると定義する．**図 15** の二つのリストが等しいことを判定する Scheme のプログラムを par-and で並列化したものを**図 16** に示す．また，生成するプロセスの数を制限するプログラムを**図 17** に示す．

```
(define (equals l m)
  (cond ((and (pair? l) (pair? m))
         (and (equals (car l) (car m))
              (equals (cdr l) (cdr m))))
        (else (eq? l m))))
```

図 15 Scheme のプログラム equals
Fig. 15 Scheme program of equals.

```
(define (equalp l m)
  (cond ((and (pair? l) (pair? m))
         (par-and (equalp (car l) (car m))
                  (equalp (cdr l) (cdr m))))
        (else (eq? l m))))
```

図 16 並列化したプログラム equalp
Fig. 16 Parallel version of equal.

```
(define equalc
 (letrec
  ((ec
    (lambda (l m d)
     (if (zero? d)
         (equals l m)
         (if (and (pair? l) (pair? m))
             (par-and
               (ec (car l) (car m) (- d 1))
               (ec (cdr l) (cdr m) (- d 1)))
             (eq? l m))))))
  (lambda (l m d) (ec l m d))))
```

図 17 プロセス生成を制限するプログラム equalc
Fig. 17 Restricted parallel version of equal.

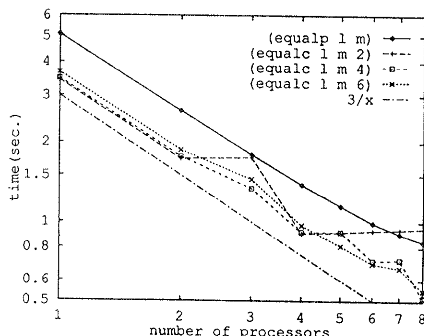

図 18 同じリストを比較する時間
Fig. 18 Execution time of equal.

表 1 異なるリストを比較する時間 (sec)
Table 1 Execution time of equal on erroneous list.

errors	sequential	equap	equalc (d=4)
1	2.843	0.020 (0.325)	0.267 (0.392)
2	0.005	0.008 (0.158)	0.012 (0.207)
2	0.663	0.379 (0.732)	0.377 (0.505)
3	0.015	0.081 (0.641)	0.510 (0.503)
4	0.015	0.077 (0.513)	0.342 (0.381)

これらのプログラムについて，全く同じリストを比べた場合と数箇所が異なるリストを比べた場合について実行時間を計る．木構造の深さはいずれも 10 である．

図 18 は同じリストの比較の結果である．プロセスの生成を制限しない (equalp lm) は，プロセス生成のオーバヘッドのため，プロセスが 2^4 個だけ生成される (equalc lm 4) の約 1.5 倍の時間がかかる．

表 1 は数箇所異なるリストの比較の結果である．異なる要素が少ない場合はプロセス生成を制限しない方が早く誤りを見付けることができる．異なる要素が多い場合は，並列化した方が equals に比べて評価する引数の数が多くなるので実行時間が逆に長くなる．

一つの par-and 構文で生成されたプロセスは，その par-and の値が決まるときに終了させられるが，その下の代のプロセスまでは終了させることができない．したがって，値が返った後も，計算を続けているプロセスが残る場合がある．( ) 内は同じ例を 5 回続けて実行し，その実行時間の平均をとったものである．残っているプロセスの影響で，連続実行させた方の平均実行時間が長くなっている．

5.4 call/ep と call/cc の比較

call/cc を用いたプログラムとその中の call/cc を call/ep に置き換えたプログラムの実行時間を比較し call/ep の効果を調べる．

図 19 は continuation の生成と呼び出しを頻繁に行うプログラムである．lcp を再帰的に呼ぶことにより call/cc を繰り返し実行する．n が 0 になったならばプロセスを繰り返し生成する．そのプロセスは停止した親プロセスを再開する continuation を呼び出す．

```
(define (lcp n)
  (lambda (k)
    (if (zero? n)
        (k 0)
        (call/cc (lcp (- n 1))))
    (spawn (k 0))
    (suspend)))
```

図 19 continuation 生成・呼び出しプログラム
Fig. 19 Creating and invoking continuations.

```
(define (ffib n)
  (if (< n 2)
      1
      (+ (future (ffib (- n 2)))
         (future (ffib (- n 1))))))
```

図 20 フィボナッチ数列を計算するプログラム［II］
Fig. 20 Computing the Fibonacci sequence using future［II］.

表 2 call/cc と call/ep を用いたプログラムの実行時間 (sec)
Table 2 Execution time of programs using call/cc or call/eq.

	lcp 20	lcp 40	future	par-or
call/cc	0.730	0.1845	2.184	0.2807
call/ep	0.523	0.1028	2.135	0.2790

これに加えて PaiLisp-Kernel を用いた記述[5]を基にして future, par-or をマクロで定義したプログラムについても実験する．実際には，(call/cc (lcp 20)), (call/cc (lcp 40)), **図 20** のプログラムを用いた (ffib 10) および (par-or (fib 20) (fib 15) (fib 10)) を実行する．

その結果を**表 2** に示す．continuation に関する処理が大きな割合を占めているプログラム lcp の実行結果から，call/ep を用いることによりコストを下げられることがわかる．制御スタックに退避されているデータの量が多いほど call/ep の効果が大きく現れる．future, par-or の場合はそれほど効果が現れていないが，これは全体の処理に対する continuation の生成とその呼び出しの処理の比重が軽いためである．

6. おわりに

PaiLisp-Kernel の実現法と PaiLisp の並列構文の continuation および escape procedure を用いた実現法を与えた．

PaiLisp のインタプリタを試作し，並列構文の評価を行った．PaiLisp の並列処理の効率を上げるためには，プロセスの過剰生成を防ぎ，必要がなくなったプロセスを自動的に終了させ，プロセスの実行に優先順位を付けることができるような仕組みによって，限られたプロセッサ資源を有効に使うことが必要である．

call/cc の代わりに call/ep を用いることにより，制御スタックの復元処理を軽減することができることが実験でも確かめられた．

今後の課題として，PaiLisp で効率のよいプログラムを書くために，future を用いた効果的な並列化の方法，future と pcall の使い分け，call/cc を call/ep で自動的に置き換える方法などを考える必要がある．

まだ実現していない構文に signal, wait があるが，これによりモニタを実現することができ，排他的関数クロージャの応用範囲が広がるだろう．

試作した PaiLisp インタプリタの GC は自由セルを使い尽くしたときにすべてのプロセッサの実行を止めてから逐次的に行っている．システム全体の処理効率を上げるためには，GC を並列に行う必要がある．

参考文献

1) Harrison III, W. L. and Ammarguellat, Z.: The Design of Automatic Parallelizers for Symbolic and Numeric Programs, Parallel Lisp: Languages and Systems, *LNCS*, No. 441, pp. 235-253 (1990).
2) Ito, T., Matsuyama, T., Kurokawa, H., Kon, A. and Ohtomo, M.: An MC68000-Based Multi-Microprocessor System with Shared Memory and its Application to Parallel Lisp Intepreter, *Symposium on Computer System*, pp. 140-150, IPS Japan (1987).
3) Halstead, R. H., Jr.: Implementation of Multilisp: Lisp on a Multiprocessor, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pp. 9-17 (1984).
4) Gabriel, R. P. and McCarthy, J.: Queue-based Multi-processing Lisp, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pp. 25-44 (1984).
5) Ito, T. and Matsui, M.: A Parallel Lisp Language PaiLisp and Its Kernel Specification, Parallel Lisp: Languages and Systems, *LNCS*, No. 441, pp. 58-100 (1990).
6) Rees, J. and Clinger W. (eds.): The Revised^3 Report on the Algorithmic Language Scheme, *ACM SIGPLAN Notices 21*, pp. 37-79 (1986).
7) 湯浅太一：Scheme 入門，岩波コンピュータサイエンス，岩波書店 (1991).
8) Halstead, R. H., Jr.: New Ideas in Parallel Lisp: Language, Implementation, and Programming Tools, Parallel Lisp: Languages and Systems, *LNCS*, No. 441, pp. 2-57 (1990).
9) Ito, T. and Seino, T.: On PaiLisp Continuation and its Implementation, *ACM SIGPLAN Workshop on Continuations*, pp. 73-90 (1992).
10) Ito, T., Tamura, T. and Wada, S.: Theoretical Comparisons of Interpreted/Compiled Executions of Lisp on Sequential and Parallel Machine Models, *Information Processing 86* (*Proc. IFIP Congress* 86), pp. 349-354 (1986).
11) 伊藤貴康，和田慎一：LISP 関数の並列実行モデルとその評価，情報処理学会記号処理研究会資料

26-5 (1983).

12) Ito, T.: Lisp and Parallelism, *Artificial Intelligence and Mathematical Theory of Computation*, Lifschitz, V. (ed.), Academic Press (1991).

13) Springer, G. and Friedman, D.: *Scheme and the Art of Programming*, chapter 16, The MIT Press (1989).

14) 竹内郁雄：LISP 処理系コンテストの結果，情報処理学会記号処理研究会資料，5-3 (1978).

15) Akl, S.G.: *Parallel Sorting Algorithms*, Academic Press (1985).

16) Osborne, R.B.: Speculative Computation in Multilisp, *Parallel Lisp: Languages and Systems*, *LNCS*, No. 441, pp. 103-137, Springer-Verlag (1990).

付　　録

A. PaiLisp のシンタックス

A.1 PaiLisp-Kernel のシンタックス

E_k (PaiLisp-Kernel の式) ⟶ K (定数) | I (識別子)

| (E^+)

| $(\text{lambda}(I^*)E^+)$ | $(\text{lambda}(I^*.\ I)E^+)$

| $(\text{lambda}\ I\ E^+)$

| (if E E E) | (if E E)

| (set! I E)

| (spawn E)

| (suspend)

| $(\text{exlambda}(I^*)E^+)$

| (call/cc E)

$E \longrightarrow E_k$

A.2 PaiLisp のシンタックス

E_p (PaiLisp の式) ⟶ E_k

| $(\text{par}\ E^+)$

| $(\text{pcall}\ E^+)$

| $(\text{eager}\ E^+)$

| (future E)

| $(\text{par-and}\ E^*)$ | $(\text{par-or}\ E^*)$

| $(\text{pcond}(E^+)^*(\text{else}\ E^+))$

| $(\text{pcond\#}(E^+)^*(\text{else}\ E^+))$

| $(\text{pcond}(E^+)^+)$

| $(\text{pcond\#}(E^+)^+)$

| (signal E E) | (wait E)

$E \longrightarrow E_p$

B. PaiLisp の並列構文

(pcall *operator* $operand_1$…) syntax

$operand_1$, …を並列に評価し，関数適用を行う．

(eager *operator* $operand_1$…) syntax

operator と $operand_1$, …を並列に評価し，適用を行う．

(future *e*) syntax

式 *e* を評価するプロセスを生成し，その値が入る場所を返す．これを future 値と呼ぶ．

(par-and $test_1$…) syntax

$test_1$, … を並列に評価し，どれかの値が #f となったならば，#f を値として返し他の *test* の評価を終了させる．すべて #f でないならば最後の値を返す．

(par-or $test_1$…) syntax

$test_1$, …を並列に評価し，最も早く #f でない値に評価された値を返し，他の *test* の評価を終了させる．

(pcond ($test_1$ e_{11}…) ($test_2$ e_{21}…)…) syntax

$test_1$, … を並列に評価し，最初に真になった所の本体を実行する．他の述語の評価はそのときに終了させる．

(pcond# ($test_1$ e_{11}…) ($test_2$ e_{21}…)…) syntax

e_{11}, … も $test_1$, … と並列に評価し，最初に真になった所を除いてすべての実行を終了させる．

(par e_1…) syntax

e_1, …を並列に評価する．

(pmapcar *proc list*) procedure

list の各要素への *proc* の適用を並列に行う．

```
(define (null-copy l)
  (if (null? l)
      '()
      (cons '() (null-copy (cdr l)))))
(define sort
  (let ((l '()) (m '()) (c 0) (k 'dummy))
  (let (
(term (exlambda ()
 (if (= c 1) (k m) (set! c (- c 1))))))
  (letrec(
(count (lambda (l r n a i)
 (cond ((null? l) r)
       ((> a (car l))
        (count (cdr l) (cdr r) (+ n 1) a i))
       ((= a (car l))
        (if (> n i)
            (count (cdr l) (cdr r) (+ n 1) a i)
            (count (cdr l) r (+ n 1) a i)))
       (else (count (cdr l) r (+ n 1) a i)))))
(copy-rec (lambda (a n)
 (cond ((null? a) (suspend))
       (else (spawn (begin
              (set-car! (count l m 0 (car a) n)
                        (car a))
              (term)))
             (copy-rec (cdr a) (+ n 1)))))))
(lambda (x)
  (set! l x)
  (set! m (null-copy x))
  (set! c (length x))
  (call/cc (lambda (r) (set! k r)
                       (copy-rec l 0))))))))
```

図 21 数え上げソートのプログラム
Fig. 21 Counting sort program.

(signal *cvar e*), (wait *cvar*) syntax

排他的関数クロージャの中だけで使える．wait は，実行を停止して最も内側の排他クロージャを解放し，*cvar* で示されるキューで待つ．signal は，その実行を再開して *e* の値を排他クロージャ適用の値として返す．

C．数え上げソートのプログラム

図 21 が数え上げソートのプログラムである．このソートでは各要素が全要素の中で何番目になるかを，それより小さい要素を数えることにより決定する．count はそれを数える関数である．copy-rec では各要素ごとに count を行うプロセスを生成し停止する．term はすべてのプロセスの count が終了したかどうかを検査する排他的関数クロージャであり，すべて終了したときは停止しているプロセスの実行を再開する．

(平成 4 年 8 月 18 日受付)
(平成 5 年 9 月 8 日採録)

清野 智弘（正会員）

1968 年生．1991 年東北大学情報工学科卒業．1993年同大学院工学研究科情報科学専攻修士課程修了．現在，(株)日立製作所ソフトウェア開発本部勤務．大型汎用コンピュータ用 OS の開発に従事．

伊藤 貴康（正会員）

昭和 15年生．昭和 37 年京都大学工学部電気卒業．昭和 42 年スタンフォード大学コンピュータサイエンス修士．工学博士（京大）．スタンフォード大学 AI プロジェクト助手，三菱電機中研，東北大学工学部助教授を経て，昭和45年より，東北大学教授．現在，東北大学工学部情報工学科および情報科学研究科に所属．研究分野は，プログラム理論，並列理論，並列記号計算などのソフトウェア基礎科学と人工知能・認識科学の基礎研究．Int'l Journal of Foundations of Computer Science の Managing Editor, Information and Computation の Editor 等．ACM, AAAI, 電子情報通信学会，ソフトウェア科学会各会員．