プログラミング 13－3
（1997．5．21）

スティール評価法を備えたPaiLispシステムの実現とその評価

川本 真一　　伊藤 貴康

東北大学 大学院 情報科学研究科

PaiLispは**pcall**, **pbegin**, **plet**, **par-and**, **pif**, **future**などの並列構文を備えた並列Lisp言語である．従来のPaiLispの処理系は，並列構文の評価法としてETC(Eager Task Creation)を用いていたため，プロセスの過剰生成によるオーバヘッドが大きく効率的な並列実行ができないという問題点があった．スティール評価法というETCのプロセス過剰生成問題を克服できる汎用的で効率の良い並列構文の評価法が提案されている[4]．本稿では，マルチスレッド機構を用いて実現されたPaiLispのインタプリタPaiLisp/MTにおいて実現されているスティール評価法による並列構文の実現法を説明する．また，DEC7000上での試作システムによるスティール評価法の評価結果について報告し，スティール評価法が効率の良い評価法であることを実験的に示す．

Implementation of the Steal-based Evaluation Strategy in PaiLisp System

Shin-ichi Kawamoto　　Takayasu Ito

Department of Computer and Mathematical Sciences,
Graduate School of Information Sciences,
Tohoku University

PaiLisp is a parallel Lisp language based on Scheme, and it has a rich set of parallel constructs like **pcall**, **pbegin**, **plet**, **par-and**, **pif**, and **future**. An implementation of PaiLisp using the Eager Task Creation (ETC) strategy is reported in [7]. ETC incurs considerable overhead caused by excessive creation of processes, since ETC creates new PaiLisp processes whenever a parallel construct is evaluated. The Steal-based Evaluation strategy [4] is an efficient evaluation strategy, which suppresses excessive process creation. In this paper, after explaining an implementation of the steal-based evaluation strategy in a PaiLisp system called PaiLisp/MT, we report its experimental results using a set of benchmark programs.

1 はじめに

PaiLisp は Lisp の一方言である Scheme[8] に様々な並列構文を導入して拡張した共有メモリ型並列計算機のための並列 Lisp 言語である.

PaiLisp の処理系は, まず並列構文の評価法として ETC (Eager Task Creation) を用いた PaiLisp/FX が設計・試作された [7]. ETC は並列構文が現れる度に必ずプロセスを生成するため, プロセスが過剰に生成され実行効果が良くないという問題がある.

文献 [4] で提案されているスティール評価法は, プロセスの過剰生成を抑制できる効率の良い評価法である. 本稿では, DEC7000 上で実現した PaiLisp/MT システムにおけるスティール評価法の実現法と評価実験について述べ, ETC との比較も行う.

2 PaiLisp とマルチスレッド機構による処理系

本章では並列 Lisp 言語 PaiLisp の構文, ETC による評価法, および, PaiLisp/MT の構成について説明する.

2.1 並列 Lisp 言語 PaiLisp

PaiLisp は, Scheme にいくつかの並列構文を導入して拡張した共有メモリ型並列計算機のための並列 Lisp 言語である. PaiLisp の構文は図 1のように定義される. 例えば, **pcall**, **pbegin**, **plet**, **pif** は次のような意味を持つ.

(pcall f $e_1 \cdots e_n$**)** 式 $e_1, \cdots, e_n$ を並列に評価し, すべての評価が終了した後, 式 f を評価し得られた関数を $e_1, \cdots, e_n$ の値に適用する.

(pbegin $e_1 \cdots e_n$**)** 式 $e_1, \cdots, e_n$ を並列に評価し, 全ての評価が終了した後 e_n の値を返す.

(plet $((x_1\ e_1) \cdots (x_n\ e_n))$ $E_1 \cdots E_m$**)** 式 $e_1, \cdots, e_n$ を並列に評価し得られた各値を変数 $x_1, \cdots, x_n$ に逐次的に束縛した後, $E_1, \cdots, E_m$ を並列に評価し, 全ての評価が終了した後 E_m の値を返す.

(pif e_1 e_2 e_3**)** 式 e_1, e_2, e_3 を並列に評価し e_1 の値が偽でなければ e_2 の値を返し, e_3 の評価を強制終了させる. e_1 の値が偽であれば, e_3 の値を返し, e_2 の評価を強制終了させる.

その他の並列構文の意味については文献 [4][5] を参照されたい.

2.2 ETC (Eager Task Creation)

PaiLisp の最初のインタプリタである PaiLisp/FX[7] では, 並列構文の実現方法として ETC を用いた.

```
D  ::= (define x E)

E  ::= V
     | (E0 E1 ··· En)
     | (begin E1 ··· En)
     | (if E0 E1 E2)
     | (cond ((E11 E12)···(En1 En2)))
     | (let ((x1 E1) ··· (xm Em)) Em+1 ··· En)
     | (let* ((x1 E1) ··· (xm Em)) Em+1 ··· En)
     | (letrec ((x1 E1) ··· (xm Em)) Em+1 ··· En)
     | (and E1 ··· En) | (or E1 ··· En)
     | (set! x E)
     | (call/cc E)
     | (delay E)
     | (pcall E0 E1 ··· En)
     | (pbegin E1 ··· En)
     | (plet ((x1 E1) ··· (xm Em)) Em+1 ··· En)
     | (pletrec ((x1 E1) ··· (xm Em)) Em+1 ··· En)
     | (pif E0 E1 E2)
     | (pcond ((E11 E12)···(En1 En2)))
     | (pcond# ((E11 E12)···(En1 En2)))
     | (par-and E1 ··· En) | (par-or E1 ··· En)
     | (future E)
     | (call/pcc E)
     | (spawn E)
     | (suspend)

V  ::= c | x
     | (lambda (x1 ··· xm) E1 ··· En)
     | (exlambda (x1 ··· xm) E1 ··· En)

c  ∈ Constant
x  ∈ Variable
```

図 1: PaiLisp の構文

ETC は並列構文を実現する際に通常用いられる方法で, 並列構文が出現する度にプロセスを生成する. すなわち, 式 **(pcall** f $e_1 \cdots e_n$**)** を ETC で評価すると, $e_1, \cdots, e_n$ を評価する n 個のプロセスが必ず生成される. 頻繁に並列構文を呼び出す並列プログラムを ETC で実行すると, プロセスの過剰生成に伴うオーバヘッドのために実行効率が悪くなる. 例として, 図 2のプログラムを考える.

```
(define (pfib n)
   (if (< n 2) n
       (pcall + (pfib (- n 1))
                (pfib (- n 2)))))
```

図 2: フィボナッチ数を求める並列プログラム

引数に 20 を与え pfib を ETC で評価すると, 21890 個のプロセスが生成される. DEC7000 上の PaiLisp/MT システムにおける 6 つのプロセッサを用いた実験から, (pfib 20) の ETC による評価時間は 0.82 [sec] である. 一方, pfib プログラムの **pcall** 構文を取り除いた逐次プログラムの評価時間は 0.69 [sec] であり, 並列プログラムのほうが逐次プログラムより遅い. これは, ETC によるプロセスの過剰な生成のオーバヘッドが原因となっている.

2.3 PaiLisp/MT の構成

PaiLisp/MT は文献 [1] の Register Machine (RM) と呼ばれる仮想計算機をベースとした並列インタプリタである.

PaiLisp/MT は，図3に示すように，6台のプロセッサを有する共有メモリ型並列計算機 DEC7000 上で，OSF/1 OS の P-thread ライブラリを用いて実現された．なお，SST_0，…，SST_5 は3章で説明するスティール評価法の実現に用いられるスタックである．

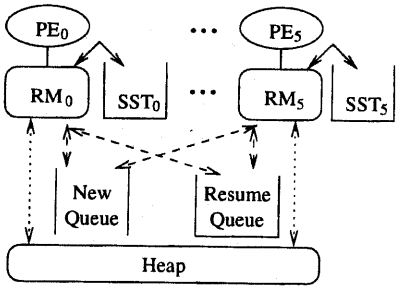

図 3: PaiLisp/MT のシステム構成

RM は，レジスタとスタックと式の評価ルーチンから構成される仮想計算機で，並列計算機の各プロセッサに RM を一つ割り付け，複数台の RM を並列に動作させる．各 RM はスレッドとして実現されており，RM は並列プログラムを評価すると新しいプロセスを生成する．生成されたプロセスは New Queue に入れられ，停止していたプロセスの実行が再開される場合は，Resume Queue に入られ，それぞれ RM の割当てを待つ．空き状態となった RM は，Resume Queue，New Queue という順でプロセスキューを検査し，PaiLisp プロセスが存在する場合はそれを取って評価する．すべての RM は一つのヒープ領域を共有している．

仮想計算機 RM のうち，RM_0 はマスタプロセッサと呼ばれ，ユーザとの対話機能を備え，図4の構成をしている．その他の RM はスレーブプロセッサと呼ばれ，図4から Reader と Printer を除いたものである．Evaluator の parallel construct には，並列構文を ETC モードとスティール評価モードで評価する評価モジュールが具備されている．

3 スティール評価法とその実現法

スティール評価法は，ETC のプロセス過剰生成を克服する並列評価法として文献 [4] で提案されたもので，次のような考え方に基づいている．

プロセッサが (**pcall** $f\ e_1 \cdots e_n$) のような並列構文を含む式に出くわすと，そのプロセッサはホームプロセッサ (home processor) と呼ばれるプロセッサとなって，式 (**pcall** $f\ e_1 \cdots e_n$) の評価を主に受け持つ．ホームプロセッサは，引数式を $e_1, e_2, \cdots$ の順に逐次的に評価していく．他のプロセッサが空き状態になると，そのプロセッサはホームプロセッサの SST (Stealable STack) に蓄えられた式をスティール (steal) し，それを評価する．この式をスティールするプロセッサをシーフプロセッサ (thief processor) と呼ぶ．シーフプロセッサがホームプロセッサから式をスティールし評価することによって，並列処理が行われる．プロセスの生成はシーフプロセッサが式をスティールする場合にのみ発生するため，ETC に比べてプロセス生成数が少なく効率的な並列実行が可能となる．

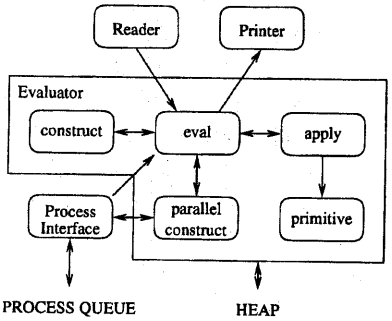

図 4: RM_0 におけるインタプリタの構成

3.1 pcall 構文のスティール評価法による実現法

スティール評価法では SST というスタックを用いて処理が行われる．図3に示すように，i 番目の RM_i には SST_i がある．また，RM_i は SST_i を管理するための2つのポインタ top_i と $bottom_i$ を持つ．SST の個々のスロットは，式へのポインタと，共有オブジェクトへのポインタから構成される．

(1) ホームプロセッサによる逐次評価モード

ホームプロセッサによる，式 (**pcall** $f\ e_1 \cdots e_n$) の評価は，SST の初期化を行ったのち，シーフプロセッサがスティールした式を正しく評価するための共有オブジェクト SO (Shared Object) を生成する．共有オブジェクトは次の5つの要素からなる．

- 環境 (env)
- 未評価の引数の数を数えるカウンタ (counter)
- 親プロセスのプロセス ID (pid)
- 並列構文の識別子 (constructid)
- 排他変数 (mutex)

共有オブジェクトを生成した後，それを式と共に SST に入れ，式 $e_1, \cdots, e_n$ を cons して ($e_1 \cdots e_n$)

というリストを作る．SST_iのtop_iには，各式e_iへのポインタと共有オブジェクトを格納する．argvは生成したリスト$(e_1 \cdots e_n)$を指し，このリストに式$e_1, \cdots, e_n$の評価値が入れられる．

ホームプロセッサとなったRM_iは，まず式e_1を評価し，値をargvリストのe_1の部分に上書きし，共有オブジェクトのcounterの値を1だけ減らす．次に，SST_iのtop_iから式を一つ取り出し，それを評価し，e_1と同様に値を書き込み，counterの値を1だけ減らす．この操作を繰り返す．

共有オブジェクトのcounterの値が0でないが，ホームプロセッサRM_iのSST_iが空となった場合は，式の評価を停止し，子プロセスの実行の終了を待つ．このとき空状態になったプロセッサは，シーフプロセッサとなって他のホームプロセッサから式をスティールして評価する．共有オブジェクトのカウンタの値が0となれば，すべての式の評価が終了し値が求まっているので，式fを評価し，その値を$e_1, \cdots, e_n$の値に適用する．

(2) シーフプロセッサのスティール評価モード

プロセスの実行が終了したり停止して空き状態になったプロセッサは，他のプロセッサのSSTを検査し，式が存在するホームプロセッサRM_kを捜す．RM_kのSST_kの$bottom_k$から，式と共有オブジェクトを一つ取り出し，プロセスオブジェクトを生成する．スティールした式が$(\mathbf{pcall}\ f'\ e'_1 \cdots e'_{n'})$のような並列構文を含む場合，このシーフプロセッサはこの式に対するホームプロセッサとなって，この式を逐次評価モードで実行する．式の評価が終了すると，得られた値をホームプロセッサのargvリストに格納し，共有オブジェクトのカウンタの値を1だけ減らす．カウンタの値が0となれば，親プロセスは停止しているので，その実行を再開させる．そして，再び他のプロセッサのSSTを検査し，式が存在すれば，スティール評価モードで実行を行う．

3.2 future構文のスティール評価法による実現法

futureは文献[3]において提案された並列構文である．**future**構文のスティール評価法による実現法について説明する．ホームプロセッサが(**future** e)を評価すると，まず式eの仮の値であるfuture値を生成し，それをSSTのtopに格納すると共に，それを値として返す．ホームプロセッサはその仮の値を用いて計算を続け，eの真の値が必要となった時に始めて式eの評価を行う．式eの評価がホームプロセッサで開始される前に，他のプロセッサが空き状態になった場合は，そのプロセッサはホームプロセッサのSSTからfuture値を奪う．future値には式eや環境などの情報が格納されているので，シーフプロセッサはそれらの情報を取り出して式eの評価を行う．式$(f\ (\mathbf{future}\ e))$を評価すると，まずfuture値$fv$が生成されてSSTに入れられ，$fv$を用いて残りの計算$(f\ fv)$が続けられる．空き状態のプロセッサは$fv$をスティールして式$e$を評価するので，ホームプロセッサによる式$(f\ fv)$の評価と，シーフプロセッサによる式$e$の評価が並列に行われる．

future構文の効率の良い評価法として，Mul-TにおけるLTC[6] (Lazy Task Creation)もある．LTCによって式$(f\ (\mathbf{future}\ e))$を評価すると，ホームプロセッサはまず式$e$を評価し，次に残りの計算$(f\ \square)$を評価する．空き状態のプロセッサがあれば，残りの計算$(f\ \square)$をスティールして評価するので，シーフプロセッサによる残りの計算$(f\ \square)$を行うプロセスが生成され，ホームプロセッサによる式eと並列に計算が行われる．(**future** e)と書いたとき，式eを並列実行することがプログラマによって指示されていると考えるのが自然であるから，上述のスティール評価法の**future**構文の実現法の方が本来の意味に忠実な方法であるといえる．また，LTCでは残りの計算$(f\ \square)$をスティールする際にスタックのコピーを行う必要があり，プログラムによってスティールのコストが大きくなる可能性があるが，スティール評価法のスティール動作はポインタを取るだけで一定のコストで行える．

なお，$(f\ e_1 \cdots e_n)$の式e_iのみをスティール評価させたいときには，**future**値は不要である．$(f\ e_1 \cdots (\mathbf{stealable}\ e_i) \cdots e_n)$と書きスティール評価法によって効率良く実行させる構文**stealable**が文献[4]に提案されている．

3.3 pbegin構文およびplet構文の実現

式$(\mathbf{pbegin}\ e_1 \cdots e_n)$の$e_1, \cdots, e_n$の評価は，式$(\mathbf{pcall}\ f\ e_1 \cdots e_n)$における式$e_1, \cdots, e_n$のスティール評価と同様に行われるが，すべての式の評価が終了すると，ホームプロセッサは，argvリストの最後の値，すなわち式e_nの値を取り出し，その値を返す．

plet構文$(\mathbf{plet}\ ((x_1\ e_1) \cdots (x_n\ e_n))\ E_1 \cdots E_m)$の束縛部における式$e_1, \cdots, e_n$の並列処理は，**pbegin**

構文のスティール評価と同様の方法によって行われ，ホームプロセッサは変数 $x_1, \cdots, x_n$ と argv リストの個々の要素の対を逐次的に生成して環境を拡張する．そして，拡張された環境の下で，**pbegin** 構文と同様の方法によってボディ部の式 $E_1, \cdots, E_m$ をスティール評価する．

3.4 スティール評価法による pif 構文の実現

ホームプロセッサが式 (**pif** e_1 e_2 e_3) を評価すると，式 e_3, e_2 を共有オブジェクトと共に SST の *top* に入れ，e_1 を評価する．e_1 の評価結果が偽である場合には，もし e_2 がスティールされていれば e_2 を評価するプロセスに kill シグナルを送って，e_3 の評価の終了を待つ．e_1 の値が偽でない場合には，e_3 がスティールされていれば，e_3 を評価するプロセスに kill シグナルを送ってから，e_2 を評価する．kill シグナルの送信は，シグナルを送る相手のプロセスのプロセスオブジェクトの kill フラグを立てることによって実現されている．

シーフプロセッサは，式 e_2 や e_3 をスティールするとそれを評価するが，式の評価の最中にホームプロセッサから kill シグナルが送られてきた場合は，その実行を即終了する．シグナルの処理は RM が，Evaluator の eval を呼び出す度にフラグの検査をしており，kill フラグが立っていた場合には，現在実行中のプロセスを破棄して，処理を終了する．**pif** 構文の場合は，**pcall** 構文で述べた各種のオーバヘッドの他に，kill シグナルの送信コスト (11 [μsec]) や，kill シグナルを受信しプロセスの終了を行うコスト (15 [μsec]) がかかる．

4 スティール評価法の評価

PaiLisp の並列構文をスティール評価法によって実行できる PaiLisp/MT が 6 台の Alpha プロセッサを持つ DEC7000 上に実現されている．PaiLisp/MT を用いてスティール評価法の評価を行う．

スティール評価法によって式を評価すると，(a) 共有オブジェクトの生成コスト (C_{so})，(b) SST に排他的にアクセスするコスト (C_{sst})，(c) スティール動作によってプロセスオブジェクトを生成するコスト (C_{steal}) が要る．PaiLisp/MT におけるこれらのコストの計測結果を表 1に示す．

表 1中の C_{pro} は，PaiLisp/MT の ETC でのプロセスの生成コストを表している．この結果から，逐次評価モードにおけるオーバヘッド C_{so} や C_{sst} は，スティールコスト C_{steal} やプロセス生成コスト C_{pro} に対して，1/6 程度と小さい．また，C_{steal} は C_{pro} より多少大きいが，これは，ETC では New Queue からプロセスオブジェクトを取り出すだけで良いのに対して，スティール評価法では SST から式と共有オブジェクトの 2 つを取り出すのに多少コストがかかるためである．しかし，その差は僅かであり，pfib の例のように並列構文が再帰的に呼出される場合には，スティール評価法では，ETC よりプロセス生成数が大幅に減少するため，スティール評価法の方が ETC より効率が良くなる．

表 1: PaiLisp/MT におけるオーバヘッド [μsec]

C_{so}	C_{sst}	C_{steal}	C_{pro}
9	11	70	60

4.1 ベンチマークプログラムによる実験

5 つのベンチマークプログラム，**fib**, **tarai**, **queen**, **map1**, **map2** を用いて，スティール評価法の評価実験を行った．実験結果を表 2に示す．

表 2: ベンチマークプログラムの実験結果

プログラム	粒度 [msec]	SBE [sec]	ETC [sec]	逐次 [sec]
(fib 20)	0.02	0.20 (50)	0.82 (21890)	0.69 (0)
(tarai 9 4 0)	0.02	0.47 (403)	1.41 (41421)	1.69 (0)
(queen 8)	0.15	0.49 (132)	0.73 (11016)	2.33 (0)
(map1 l1)	0.02	0.06 (1000)	0.16 (2000)	0.04 (0)
(map2 l1)	4.82	0.11 (100)	0.16 (200)	0.52 (0)

粒度は ETC でプログラムを評価した時のプロセスの平均実行時間，SBE と ETC はそれぞれスティール評価法と ETC によるプログラムの実行時間，逐次は並列構文を取り除いたプログラムの実行時間をそれぞれ示している．カッコ内の数字は生成されたプロセス数 (スティール評価法ではスティールの回数) である．プロセッサは 6 台使用した．

(1) fib, tarai, queen の 3 つのプログラムでは，引数式が再帰的定義関数であるような関数適用式 (f e_1 $\cdots$ e_n) において，**pcall** 構文を用いて (**pcall**

$f\ e_1 \cdots e_n$) のように並列化している．これを ETC で評価すると，どの引数 e_i も再帰的にプロセスを生成するため，木構造的にプロセスが多数生成される．これを，スティール評価法で評価すると，スティールによってプロセス生成が行われるのは，空プロセッサが発生したときのみであるから，プロセス生成は大幅に抑制される．

実験結果を見ると，プログラム `fib`, `tarai`, `queen` では，スティール評価法によるスティールの回数は，ETC におけるプロセス生成の回数の 1/400 ～ 1/100 程度となっている．また実行時間を見ても，ETC では逐次と同程度かそれより遅いが，スティール評価法ではプロセス生成が大幅に抑制されてオーバヘッドが小さいため，逐次の 3 ～ 4 倍程度の高速化が達成されていることがわかる．

(2) プログラム `map1`, `map2` の場合は，引数式 e_2 のみが再帰的定義関数になっている関数適用式 ($f\ e_1\ e_2$) において，**pcall** 構文を用いて (**pcall** $f\ e_1\ e_2$) のように並列化した．

`map1` の場合は，粒度がプロセス生成コストよりも小さく線形構造型であるから，本質的に逐次実行に適したプログラムである．実験結果を見ると ETC による並列評価の時間は逐次の 4 倍にもなっている．スティール評価による評価では，プロセス生成が抑制されるから，逐次と比べて余り遅くはなっていない．スティール評価法による評価では，1000 回のスティールが発生し，一回のスティールのコストが表 1 より 70 [μsec] であるから，オーバヘッドは 0.07 [sec] となって計算上は実行時間 0.06 [sec] より大きくなる．しかし，1000 回のスティールは 6 台のプロセッサに分散されて行われるから，プロセッサ一台あたりのスティール数が 1000/6 程度と考えると，その値は $0.07/6 = 0.012$[sec] となって，実行時間 0.06[sec] より小さい．

一方，`map2` も線形構造型であるが，粒度が大きいので ETC でも十分並列化の効果が得られているが，スティール評価法では，プロセス生成数が少ない分だけ，ETC より更に効率的に実行されている．

この実験結果では，スティール評価法は ETC に比べて常に効率が良く，特に木構造型並列プログラムに対し，効率的な並列実行が可能である．線形構造型プログラムでプロセス粒度が細く並列化の意味がない場合を除いて，スティール評価法を用いれば，常に効率的な並列実行を行うことが期待できる．

また，文献 [2] の高階単一化の PaiLisp プログラムを PaiLisp/MT で実行した結果，その実行時間はスティール評価法の場合 1.27[sec](23)，ETC の場合 1.31[sec](263)，逐次の場合 3.90[sec] となった．ETC は逐次の 3 倍高速であるが，スティール評価法では ETC より更に若干高速になっている．

5 おわりに

本稿では，スティール評価法について述べ，PaiLisp/MT における各並列構文のスティール評価法による実現法を説明した．また，DEC7000 上の PaiLisp/MT システムを用いてスティール評価法の評価を行い，スティール評価法が ETC に比べて効率的であることを示した．なお，スティール評価法に基づくコンパイラが，東北大学大学院情報科学研究科伊藤研究室の梅原正義 (現，日本電気) によって試作されており，インタプリタの処理速度の数倍の高速化が得られている．また，ETC が様々な並列言語の実現において用いられているが，スティール評価法は PaiLisp 以外の並列言語にも適用できる効率の良い評価法として使用できると考える．

参考文献

[1] H. Abelson, G. Sussman.: Structure and Interpretation of Computer Programs, MIT Press (1985).

[2] M. Hagiya: Running Higher-Order Unification in PaiLisp, LNCS 748, pp.155-160, Springer, (1993).

[3] R. Halstead, Jr.: Multilisp: A language for concurrent symbolic computation, ACM Trans. on Programming Languages and Systems, Vol.4, No.7, pp.501-538 (1985).

[4] T. Ito.: Efficient evaluation strategies for structured concurrency constructs in parallel Scheme systems, LNCS 1068, pp.22-52, Springer, (1995)

[5] T. Ito, M. Matsui.: A parallel Lisp language PaiLisp and its kernel specification, LNCS 441, pp.58-100, Springer (1990).

[6] E. Mohr, D. A. Kranz, R. Halstead, Jr.: Lazy task creation: A technique for increasing the granularity of parallel programs, IEEE Trans. Parallel and Distributed systems, Vol.2, No.3, pp.264-280 (1991).

[7] 清野智弘, 伊藤貴康: PaiLisp の並列構文の実現法と評価, 情報処理学会論文誌, Vol.34, No.12, pp.2578-2591 (1993).

[8] IEEE Standard 1178-1990, IEEE Standard for the Scheme Programming Language, IEEE (1990).

[9] 湯浅太一: Scheme 入門, 岩波書店 (1991).