

PaiLisp と PaiObject の処理系

田村 清朗 伊藤 貴康

東北大学大学院 情報科学研究科

PaiLisp は共有メモリ型並列計算機に基づいて設計された, Scheme ベースの並列 Lisp 言語である. 豊富な並列構文と P-continuation という強力な制御機構を持つ. そのインタプリタが Alliant FX/80 上で試作され, 評価やアプリケーションの作成が行われている.

PaiObject は PaiLisp の上にオブジェクト指向機能を導入した言語である. クラス階層を持ち, プロセスを含めたすべてのデータをオブジェクトとして扱う, などの特徴を持つ. オブジェクト指向機能を効率良くサポートできるように PaiLisp インタプリタを拡張した上で PaiObject のインタプリタを実現し, その評価を行った.

Interpreter of PaiLisp and PaiObject

Seiro TAMURA Takayasu ITO

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

PaiLisp is a Scheme based parallel lisp language, for shared memory architecture. PaiLisp has many parallelized Lisp constructs together with a new powerful control structure supported by P-continuation.

PaiObject is an object-oriented extension of PaiLisp. PaiObject has class hierarchy and treats all data as the object including processes. In order to implement on efficient PaiObject interpreter, some new constructs are added to PaiLisp interpreter. This paper reports the details of PaiObject and its interpreter implemented on Alliant FX/80.

1. はじめに

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として、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が行なわれている。PaiLisp は Lisp の方言である Scheme をベースにして、共有メモリ型マルチプロセッサに基づいて設計された並列言語である。関数引数の並列実行、Lisp 基本構文の並列化に加え、Multilisp[1] の future, Qlisp[2] の排他クロージャを取り入れるなどして、多数の並列構文を有している。また、Scheme の continuation を並列に拡張した P-continuation とそれを生成する拡張した call/cc という、強力な制御機構を持っている。さらに、この PaiLisp のコンパクトな核言語 PaiLisp-Kernel により並列構文の意味記述が与えられている。

この PaiLisp のインタプリタが共有メモリ型並列計算機 Alliant FX/80 上で試作され、ベンチマークプログラムによる評価や研究室でのさまざまな応用に供されている [7][8]。また、PaiLisp のための対話型デバッグも試作されつつある [9]。

この PaiLisp にオブジェクト指向機能を導入した共有メモリ型並列オブジェクト指向言語 PaiObject が提案されている [4]。PaiObject はクラス階層を持ち、プロセスを含むすべてのデータをオブジェクトとして扱うことができる言語である。また、プロセスやコンティニュエーションなどが属するクラスや、オブジェクト単位の排他制御などのコンストラクトが導入されている。

PaiObject におけるオブジェクトは 1) 所属するクラス 2) インスタンス変数環境 3) 排他機構を持つものとして定義される。PaiObject インタプリタではオブジェクトを、環境(シンボルと値との束縛の集合)を用いて実現している。

PaiObject インタプリタは 1) 環境及びプロセスをファーストクラスとして扱えるように、またプロセス間通信ができるように PaiLisp インタプリタを拡張し

2) 拡張されたインタプリタ上でオブジェクト指向機能をサポートする関数、マクロ等を定義することで実現されている。この拡張された PaiLisp インタプリタを用いれば、オブジェクトやプロセスを効率良く実現することができる。

本稿では、2 節で PaiLisp とそのインタプリタについて紹介したのち、3 節でオブジェクト指向機能の効率化のために拡張されたインタプリタの機能について述べる。4 節で PaiObject の概要を述べ、5 節で PaiObject インタプリタの実現、6 節で試作した PaiObject インタプリタの予備的な評価結果について述べる。

2. PaiLisp とそのインタプリタの概要

PaiLisp 言語とそのインタプリタの実現及び評価については [3][4][5][6] に報告されている。本節では PaiLisp の概要を紹介する。

2.1 PaiLisp-Kernel と PaiLisp

PaiLisp の核言語 PaiLisp-Kernel は、Scheme に 4 つの並列構文を加えて定義されるコンパクトな並列 Lisp 言語である。PaiLisp-Kernel により PaiLisp の全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]。PaiLisp-Kernel と PaiLisp は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, eager, future, par-and, par-or, pcond, pmap, signal, wait}.

PaiLisp-Kernel の 4 つの並列構文は次のような意味を持っている。

(spawn *e*) syntax

式 *e* を評価する子プロセスを生成し、親プロセスはそれと並列に実行を続ける。*e* の評価が終了すると子プロセスは終了する。spawn が返す値は不定で *e* の値は捨てられる。

(suspend) procedure

引数をとらない関数であり、適用を行なったプロセスの実行を停止する。

(call/cc *proc*) procedure

PaiLisp の call/cc は PaiLisp で導入された P-continuation を生成し、それを引数にして一引数関数 *proc* の適用を行う。P-continuation については後述する。

(exlambda (*x*₁ ... *x*_{*n*}) *e*₁ ... *e*_{*m*}) syntax

排他的関数クロージャを作る。これは Scheme の (lambda (*x*₁ ... *x*_{*n*}) *e*₁ ... *e*_{*m*}) で作られる関数クロージャにキューを付け加えたものであり、同時に 2 つ以上のプロセスにより実行されないよう排他制御を行なう。なお、この構文は排他的プロセスクロージャを生成する Qlisp の Qlambda を改良したものである。

PaiLisp は PaiLisp-Kernel の 4 つの並列構文に加えて、多数の並列構文を持っている。par は構造化プロセスの生成を行う。pcall, eager は関数引数を並列に評価する。future は Multilisp から導入されたもので、引数評価を行うプロセスを生成して仮の値を返す。par-and,

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

図 1: P-continuation の使用例

par-or はそれぞれ and, or の引数の評価を並列に行うものである。pcond は cond の条件節を並列に評価するものである。pmap は map における関数適用を並列に行う。signal/wait は適用中の排他的関数クロージャの同期に用いる。

2.2 P-continuation と PaiLisp の call/cc

continuation は元来、逐次計算プロセスのある時点の「残りの計算」を表わし、大域脱出なども記述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。

PaiLisp の call/cc は Scheme の call/cc を並列に拡張したもので、1) call/cc を実行したプロセスの「残りの計算」と 2) そのプロセスのプロセス ID とから成る P-continuation を作る。P-continuation の詳細は [3] に述べられている。

図 1 に call/cc と P-continuation の適用によって他のプロセスを制御する例を示す。この例ではプロセスの resume とプロセスの kill が行われている。spawn で子プロセスを生成した親プロセスは suspend により実行を停止する。その後、子プロセスによる (resume k) によって親プロセスは実行を再開し、kill を k の値すなわち「spawn 式の引数の評価が終了した後の子プロセスの残りの計算」= 「子プロセスの実行の終了」に束縛する。従って親プロセスが (kill 'dummy) を実行することにより子プロセスは実行の終了を行なう。

2.3 P-continuation に基づく PaiLisp インタプリタ

逐次関数型言語の実現方式として CPS(continuation-passing style) という方式があるが、PaiLisp インタプリタは新たに導入された並列コンティニュエーション P-continuation に基づいて実現されている。すなわち P-continuation を用いて並列言語の実現が行われている。その詳細は [3] に述べられている。

PaiLisp インタプリタは共有メモリ型並列計算機 Alliant FX/80 上で実現されており、ベンチマークプログ

ラムによる評価とともにいくつかの応用例による評価が行われている。ベンチマークのプログラムとしてはフィボナッチ関数、たらい回し関数、数え上げソート、TPU, N-Queen, 巡回セールスマン問題, Pure-Prolog などについての評価が行われて好ましい結果が得られている。また応用例としてはベトリネット操作システム [7] や、研究室内における CYK パーサー, ATMS, 等式求解システム [8] などの並列の実現に利用され同様に好ましい結果が得られている。

Multilisp の場合、並列性は future 構文のみによって達成されるといっても過言ではないが、PaiLisp の場合には [3] に報告されているように、問題に応じて豊富な並列構文を有効利用することで、future 構文のみによる並列実行よりも簡明で効率良い並列プログラミングができる。PaiLisp の豊富な並列構文を有効利用した、効率の良いプログラミング方式を系統的に確立することは今後の課題である。

3. オブジェクト指向機能実現のための PaiLisp インタプリタの拡張

PaiObject は PaiLisp に並列オブジェクト指向を導入した共有メモリ型並列オブジェクト指向言語である。PaiObject は PaiLisp を完全に含み、さらに次節で述べるようなオブジェクト指向機能、プロセス間通信機能などが加えられている。

PaiObject におけるオブジェクトは

- (1) 自分が所属するクラス
- (2) インスタンス変数の束縛環境
- (3) 自分自身に対する排他機構

を持つものとして定義される。オブジェクトが PaiLisp における「環境」すなわち「シンボルと値の束縛の集合」で実現できれば都合が良い。これは環境が「インスタンス変数の束縛環境をそのまま表現でき、その探索の効率が良い」という理由からである。

PaiLisp インタプリタをベースとして PaiObject インタプリタを実現することを考えると次のような問題が生じる。第一に、PaiLisp は環境をファーストクラスとして扱う機能を持っていないので、前述のように環境を用いてオブジェクトを実現するのに都合が悪い。第二に、PaiObject ではプロセスをファーストクラスとして扱うが、PaiLisp にはその機能がない。第三に、PaiLisp におけるプロセス (PaiLisp プロセスと呼ばれる) は通信機能を持っていない。PaiLisp 上でシミュレートすることでこれらの機能を実現することもできるが、効率が非常に悪くなってしまう。

このため 1) 環境とプロセスのファーストクラスな扱いと 2) プロセス間通信ができるように PaiLisp を拡張する。(以降、拡張された PaiLisp を「拡張 PaiLisp」と呼ぶ)。以下では拡張 PaiLisp の構文及び機能について説明する。なお、本節で拡張する構文をすべて C 言語で実現して PaiLisp インタプリタに組み込んだ、「拡張 PaiLisp インタプリタ」が作成されている。

3.1 オブジェクトの実現のための構文

環境によってオブジェクトを実現するために、次に示す環境操作の構文を用意した。

(eval *e env*) procedure
環境 *env* のもとで式 *e* を評価して、その結果を返す。

(current-environment) procedure
現在の環境を取り出して返す。

(append-environment *env₁ env₂*) procedure
環境 *env₁* と環境 *env₂* の和をとった環境を作って返す。

(subset-env *env var-list*) procedure
環境 *env* のうち変数リスト *var-list* で指定された変数の束縛のみを取り出した環境を生成して返す。

3.2 プロセスを扱うための構文

プロセスをファーストクラスとして扱うために、プロセスが自分自身を参照するための構文とプロセス生成時にそのプロセスを返す構文を用意した。また、プロセスの kill を明示的に行うための構文も用意した。

(spawn *e*) syntax
e を評価するプロセスを生成して、そのプロセスを返す。(PaiLisp の仕様では spawn の返す値は不定。)

(p-process-kill *p*) procedure
PaiLisp プロセス *p* を強制的に終了させる。

(p-process-kill *p*) の評価は、*p* に対して kill continuation を呼び出すことで実現している。kill continuation はシステム内部で使われている P-continuation で、任意の PaiLisp プロセスを終了させるという「残りの計算」を持つものである。

(current-process) procedure
(current-process) を実行したプロセスを返す。

3.3 プロセス間通信のための構文

PaiLisp プロセス間通信のために次のような構文を用意した。

(p-send *p d*) procedure
プロセス *p* にデータ *d* を送り、*p* によって *d* が受け取られるのを待つ (同期通信)。

(p-as-send *p d*) procedure
プロセス *p* にデータ *d* を送る。*p* によって *d* が受け取られるのを待つことなく次の処理に進む (非同期通信)。

(p-receive *p₁ ... p_n*) procedure
この式を評価したプロセスは、send, as-send によって送られてきたデータを 1 つ受け取る。データが送られてくるまでは次の処理を行わない。送信側の PaiLisp プロセス *p₁ ... p_n* を指定すると、それらから送られてきたデータのみを受け取る。

これらの構文の実現のために、PaiLisp インタプリタ内で PaiLisp プロセスを表している構造体に次の 2 つを加えた。

メッセージキュー 他のプロセスから届いたメッセージを入れておくための FIFO 方式のキュー
コンティニュエーションバッファ データの送信側プロセスと受信側プロセスの間の同期をとるためのバッファ。メッセージキューにデータが入っていない状態で receive を実行したプロセスは、再開のための P-continuation をこのバッファに入れ、停止する。このプロセスのメッセージキューにデータが入ると、バッファから P-continuation が取り出されて適用され、このプロセスは再開される。

[各構文の実現法]

(p-send *p d*) の評価は次のように行われる。

- 1) *p* によって receive されたときに再開するための P-continuation を生成する。
- 2) *p* のメッセージキューに *d* とその P-continuation の対を追加する。
- 3) *p* のコンティニュエーションバッファに P-continuation があれば、取り出して再開させる。
- 4) 停止する。

(p-as-send *p d*) の評価は次のように行われる。

- 1) *p* のメッセージキューに *d* を追加する。
- 2) *p* のコンティニュエーションバッファに P-continuation があれば、取り出して再開させる。

(p-receive *p₁ ... p_n*) の評価は次のように行われる。

- 1) p-receive を実行したプロセス (*p* とする) のメッセージキューを調べる。空のときは *p* の再開の P-continuation を生成してコンティニュエーションバッファに入れてから停止する。
- 2a) 送信側のプロセスが指定されていなければ、キューから一つデータ (*d* とする) を取り出す。*d* が p-send によって送られてきたものならば、p-send 側

のプロセスを P-continuation の呼び出しによって再開させる。→ 2b) へ

- 2b) 2a) → p-receive の評価結果として *d* を返す。
- 3) 送信側のプロセスが指定されているときは、キューの中に指定されたプロセスから送られてきたデータがあるかどうか調べる。該当するデータがあれば 2a) と同様の処理を行う。なければ、再び 3) から始めるための P-continuation を生成しコンティニュエーションバッファに入れてから停止する。

4. PaiObject の概要

PaiObject は次のような特徴を持つ並列オブジェクト指向言語である。

- (1) クラスとクラス階層を持つオブジェクト指向言語として設計されている。下位のクラス (サブクラス) は上位のクラス (スーパークラス) の性質を継承する。1 つのクラスは複数のスーパークラスを持つことができる。
- (2) すべてのデータはオブジェクトとして統一的に扱われる。クラスもオブジェクトとして実現されており、クラスの生成、インスタンスオブジェクトの生成はクラスオブジェクトへのメッセージ送信で行われる。
- (3) プロセスをファーストクラスとして扱うことができる。これによりプロセスの強制終了、プロセス間通信などが簡単に行える。
- (4) 複数のプロセス間でオブジェクトを共有できる。
- (5) プロセスや P-continuation が属するクラスや、オブジェクト単位の排他制御などのコンストラクトが導入されている。

4.1 PaiObject の言語仕様

PaiObject の言語仕様は、拡張 PaiLisp にオブジェクト指向のコンストラクトを加えた、次のようなものであると考えることができる。これらのコンストラクトは拡張 PaiLisp のプログラムとして実現されている。

PaiObject = 拡張 PaiLisp + DefMethod + Class + Method.

DefMethod = {add-method, add-exmethod}.

Class = {object, class, process, virtual, (UserDefineClass), (SchemeDataClass)}.

(SchemeDataClass) = {number, symbol, pair, boolean, string, continuation}.

Method = {make, initialize, get-class, is-a?, subclass?, process-init, process-kill, send, as-send, receive, (UserDefinedMethod)}.

[DefMethod]

(add-method (*method-name* (*c . ivars*) . *args*) *body*)
syntax

クラス *c* に、メソッド名 *method-name* で呼び出されるメソッドを定義する。*ivars* に使用されるインスタンス変数、*args* に仮引数、*body* に実行される本体を指定する。(add-exmethod は排他メソッドを定義する。)

[Class]

object クラス サブクラス-スーパークラス関係のルートになるクラス。

class クラス クラス-インスタンス関係のルートになるクラス。クラスオブジェクトは class クラスに属する。class クラスは自分自身に属する。

process クラス プロセス (プログラムを逐次実行する制御の流れ) が属するクラス。

virtual クラス (future *e*), (delay *e*) を評価したときに返される仮の値が属するクラス。

(SchemeDataClass) Scheme のプリミティブなデータが属するクラス。例えば、1,2,... は number クラスに属するオブジェクトである。

[Method]

make メソッド クラスオブジェクトに適用することで、そのクラスのインスタンスオブジェクトが生成される。特に、class クラスに適用することでクラスオブジェクトが生成される。

initialize メソッド オブジェクトの初期化を行う。

send, as-send, receive メソッド プロセス間通信のためのメソッドで、process クラスに定義されている。

4.2 PaiObject プログラミングの例

PaiObject によるプログラミング例を示す。図 2 はオブジェクト指向機能を使用した例である。座標軸における点を point クラス、円を circle クラスとして定義し、点の移動のための move メソッドを定義している。point クラスは座標を表す *x*, *y* をインスタンス変数として持つ。circle クラスは point クラスをスーパークラスとして持つ。move メソッドの定義では point オブジェクトのインスタンス変数 *x*, *y* を使用することを宣言している。実際に実行されるのは (set! *x* (+ *x* *dx*)) 以降の部分である。c1 に circle オブジェクトを生成して束縛する。c1 に move メソッドを適用すると circle クラスのスーパークラスである point クラスに定義されているものが呼び出され、move の仮引数 self, *x*, *y* にそれぞれ c1, 10, 12 が束縛されて本体が実行される。

```

(define point                               ;point クラス生成
  (make class '(x y) '()))

(define circle                               ;circle クラス生成
  (make class '(x y r) (list point)))

(add-method                                 ;initialize メソッド定義
  (initialize (circle x y) self ix iy)
  (set! x ix)
  (set! y iy))

(add-method                                 ;move メソッド定義
  (move (point x y) self dx dy)
  (set! x (+ x dx))
  (set! y (+ y dy))
  self)

(define c1                                   ;circle オブジェクト生成
  (make circle 3 4 2))

(move c1 10 12)                             ;circle オブジェクトに move
                                           ;メソッドを適用 (メソッド継承)

```

図 2: オブジェクト指向機能の使用例

```

(define top-level-process (current-process))

(define p1
  (spawn
    (begin
      (print 'start')
      (send top-level-process 'data')
      (print 'end'))))

(receive p1)

```

図 3: プロセス通信の例

図 3 はプロセス通信の例である。はじめに、top-level-process にトップレベルのプロセスを束縛する。次に spawn によってプロセスを生成して p1 に束縛する。p1 は "start" を表示した後 "data" を top-level-process に送る。top-level-process が receive によって p1 からのデータを受け取った後で、p1 は "end" を表示する。

5. PaiObject インタプリタの実現

オブジェクトは環境として実現されている。あるオブジェクトのインスタンス変数の値は、そのオブジェクトのもとでインスタンス変数名を評価することで得られる。また、クラスもオブジェクトであるから環境として実現されている。あるクラスに定義されているメソッド本体は、そのクラスのもとでメソッド名を評価することで得られる。

拡張 PaiLisp によって直接扱うことができるようになった PaiLisp プロセスは、そのまま PaiObject におけるプロセスオブジェクトとして扱われている。一方、(make process) という形などでプロセスオブジェクト

```

environment
[(my-class <所属するクラス>)
 (ivar1 . <インスタンス変数 ivar1 の値>)
 :
 (ivarn . <インスタンス変数 ivarn の値>)
 (lock .
  (exlambda (method-exec emethod-env args)
    (apply (eval method-exec method-env)
      args)))]

```

図 4: 環境によるオブジェクトの実現

が生成される場合もあり、この場合は環境として実現されている。このように、PaiObject のプロセスオブジェクトは 2 タイプ存在する。process クラスのメソッドは受け取ったプロセスオブジェクトのタイプに応じて異なる動作をするように記述されている。

5.1 オブジェクトの実現

前節でも述べたように、PaiObject でのオブジェクトは 1) 所属するクラス 2) インスタンス変数環境 3) 自分に対する排他メソッドの適用機構から構成される。オブジェクトはこの 3 つで構成される環境として図 4 のように実現される。なお、lock は後述する排他メソッド適用機構を持つ排他クロージャである。

5.2 メソッドの実現

メソッドは、呼び出すときに用いられるメソッド名と実際の操作を記述したメソッド本体から成る。メソッド本体は次のような構造をしている。

```

method-exec メソッド実行部 (実際に実行されるプログラム)
ivars メソッド実行部で使われるインスタンス変数
method-env メソッド定義時の環境
ex-flag 排他メソッドかどうかのフラグ

```

5.2.1 メソッド定義

```

(add-method
  (method-name (c . ivars) . args) body)

メソッド定義は次のようにして実現される。はじめにメソッド実行に必要な情報を持つメソッド本体を生成して method-body に束縛する。

(set! method-body
  (list '(lambda (. args) body)
    ivars
    method-env ;メソッド定義時の環境
    ex-flag)) ;排他メソッドフラグ

```

次に、メソッド定義を行うクラス *c* にメソッド名とメソッド本体の束縛を追加する。

```
(eval '(define ,method-name ,method-body) c)
```

5.2.2 メソッド適用

オブジェクトに対するメソッド適用は次の形で行われる。

```
(method-name obj . args)
```

ここで *method-name* はメソッド名、*obj* はオブジェクト、*args* は引数である。*method-name* に対応するメソッド本体の探索は次のようにして行われる。

始めに、*obj* のもとで "my-class" を評価して自分が属するクラスを得る (これを *c* とする)。*c* において *method-name* が束縛されているかどうかを関数 *method-search* で調べる。

```
(method-search method-name c)
```

method-search はクラス *c* に *method-name* が束縛されているかどうか調べ、束縛されていれば対応するメソッド本体を返す。束縛されていないときは *c* のもとで "super-class" を評価して *c* のスーパークラスを求め、再帰的に *method-search* を適用する。

メソッド本体が得られたら、メソッド実行部を取り出して適用する環境のもとで評価する。このようにして得られたクロージャを (*cons obj args*) に適用することでメソッド適用が行われる。

```
(apply
  (eval
    (method-exec
      (append-environment
        (subset-environment obj ivars)
        method-env))
    (cons obj args)))
```

5.2.3 排他メソッド適用

前述のように、各オブジェクトには *lock* という排他クロージャが定義されている。排他メソッドの適用は *lock* の内部で行われているので、複数のプロセスによって同時に一つのオブジェクトに対する排他メソッドの同時適用は起こらない。

```
lock = (exlambda (method-exec env args)
  (apply (eval method-exec env) args))
```

5.3 クラスの実現

PaiObject のクラスは *class* クラスに属するオブジェクトとして扱われる。クラスオブジェクトは次の2つのインスタンス変数を持つオブジェクトとして実現さ

れる。

super-class 自分のスーパークラスのリスト

instance-ivar-list 自分のインスタンスオブジェクトが持つべきインスタンス変数のリスト

5.3.1 class クラス

class クラスのインスタンスオブジェクトはクラスオブジェクトであるから、*instance-ivar-list* はクラスオブジェクトが持つ2つのインスタンス変数で構成される。*class* クラスは次のようなクラスオブジェクトとして実現される。

super-class object

instance-ivar-list ('super-class 'instance-ivar-list)

[make メソッド]

オブジェクトを生成するメソッド。make メソッドは適用されたクラスオブジェクトを *my-class* に、排他機構のための排他クロージャを *lock* に、*Nil* を各インスタンス変数に束縛した環境をオブジェクトとして生成する。次にそのオブジェクトに *initialize* メソッドを適用する。

[initialize メソッド]

新しく生成されたクラスオブジェクトを初期化するメソッド。新しく生成されたクラスオブジェクトのもとで、*instance-ivar-list*、*super-class* を与えられた初期値に設定する。

5.3.2 process クラス

次の2種類のオブジェクトがプロセスオブジェクトとして扱われる。

(a) *PaiLisp* プロセスそのもの

(b) make メソッドを *process* クラスに適用することで「環境」として生成されたオブジェクト

(b) タイプのプロセスオブジェクトの場合、実際の *PaiLisp* プロセスは *p-process* というインスタンス変数に束縛されている。*process* クラスは次のようなクラスオブジェクトとして実現される。

super-class object

instance-ivar-list ('p-process)

[process-init メソッド]

プロセスオブジェクトの初期化を行うメソッド。(b) タイプのプロセスオブジェクトは *process-init* が適用されるまでは実際の *PaiLisp* プロセスの生成を行わない。*process-init* の適用によってはじめて *PaiLisp* プロセスが *spawn* で生成され、*p-process* というインスタンス変数に束縛される。なお、(a) タイプのプロセス

オブジェクトに process-init を適用しても何も行われない。

[process-kill メソッド]

プロセスオブジェクトを kill するメソッド。プロセスオブジェクトが (a) タイプのときは p-process-kill を適用し、(b) タイプのときは p-process に束縛されている PaiLisp プロセスに p-process-kill を適用することで実現される。

[send, as-send, receive メソッド]

プロセスオブジェクトへのデータ送信 (同期的, 非同期的), プロセスオブジェクトからデータ受信を行うメソッド。それぞれ p-send, p-as-send, p-receive を用いて process-kill メソッドと同様にして実現される。

6. PaiObject インタプリタの評価

6.1 PaiLisp インタプリタとの比較

PaiObject インタプリタを PaiLisp インタプリタとして用いた場合のオーバーヘッドを、フィボナッチ数の並列計算によって調べた。fib($n-1$) と fib($n-2$) を並列に計算するため future を用いてプロセスを生成している。計測の結果、PaiLisp インタプリタと比較して 1.03 倍の時間がかかることがわかった。これは PaiLisp プロセスの構造を変更してサイズが大きくなったことで、プロセス生成のコストが若干増加したためと考えられる。

6.2 メソッド適用のコスト

メソッド呼び出しのコストはシステムの実行効率を大きく左右する。PaiObject インタプリタでのメソッド呼び出しの実行効率を、PaiLisp インタプリタで同等の機能をシミュレートしたものと比較した。その結果、PaiLisp の約 2 倍の時間がかかった。

6.3 共通の問題による PaiObject と PaiLisp の比較

次の問題をシミュレートするプログラムを PaiLisp と PaiObject の両方で記述し、実行効率を比較した。

「複数のノードが直線上に接続されている。各ノードはそれぞれ固有の関数を持っている。最初のノードにデータを送ると、ノードはデータに自分が持つ関数を適用して隣のノードにそれを渡す。こうしてデータは次々に関数に適用されながら移動して、最後のノードから結果が出力される。」

最初のノードにデータを数個同時に送ってデータの移動を並列に行った。また、同じ問題をノードをプロセスとして実現した場合の比較も行った。実験の結果、PaiObject インタプリタでの実行時間はいずれも Pai-

Lisp インタプリタの約 3 倍であった。

7. おわりに

PaiLisp と PaiObject の言語仕様及びインタプリタについて、PaiObject インタプリタの実現を中心に述べてきた。PaiObject インタプリタの実用処理系としての性能はまだ不十分である。今後、オブジェクト指向機能を実現するために拡張 PaiLisp 上で書かれている部分を C 言語でインプリメントすることで、十分に実行効率を上げることができると考えている。

参考文献

- [1] R.H.Halstead, Jr., Implementation of Multilisp: Lisp on a multiprocessor, Conference Record of 1984 ACM Symposium on Lisp and Functional Programming, 9-17(1984)
- [2] R.Gabriel, J.MaCarthy, Queue-based multiprocessing Lisp, Conference Record of 1984 ACM Symposium on Lisp and Functional Programming, 25-44(1984)
- [3] Takayasu Ito, Tomohiro Seino, P-Continuation Based Implementation of PaiLisp Interpreter, to be published in Springer LNCS (1993) [Proc. of US/Japan Workshop on Parallel Symbolic Computing at MIT]
- [4] Takayasu Ito, LISP and Parallelism, Artificial Intelligence and Mathematical Theory of Computation, Papers in Honor of John McCarthy, (ed V.Lifshitz), 187-206, Academic Press(1991)
- [5] Takayasu Ito, Manabu Matsui, A parallel Lisp language PaiLisp and its kernel specification, Springer LNCS, 441, 58-100(1990)
- [6] 清野 智弘, 伊藤 貴康, PaiLisp による並列処理の実現と評価, 情報処理学会講演論文集, 9F3(1993)
- [7] 川本 真一, 伊藤 貴康, PaiLisp を用いたベトリネット操作システムの試作, 電気関係学会東北支部連合大会, 1C8(1993)
- [8] 神林 義明, 伊藤 貴康, メタ推論を用いた方程式求解システムの試作, 電気関係学会東北支部連合大会, 1C10(1993)
- [9] 小柳 朗, 伊藤 貴康, 並列 Lisp のデバッガの試作, 電気関係学会東北支部連合大会, 1C6(1993)