Lisp における並列動作の記述と実現†

岩崎 英哉††

Lisp に並列性（マルチプロセス）を導入することは，マルチプロセッサ計算機への対応，Lisp を用いた応用プログラムの開発等の意味において重要である．本論文では，陽な（ユーザの指定可能な）並列性を Lisp に導入する立場にたち，Lisp における並列動作の記述および実現の一方法を示す．この方法は，Utilisp の拡張版 mUtilisp で採り入れている．提案する方法では，並列動作指定のプリミティブは fork があるだけである．fork によってプロセス間に親子関係ができ，全プロセスは木構造を形成する．各プロセスでは，既存の環境（組み込み関数等）に加えて独自の環境を設定するが，これを，プロセスごとにシンボル空間を分割することを基本とし，さらに，親プロセスから子プロセスへ環境を継承させることによって実現する．このような実現法により，shallow binding の利点を生かすと同時に，プロセス間の予期せぬ干渉を避けている．プロセス間通信には，通信相手の指定が可能な，メッセージ伝達による方法を提案するが，その利点についても触れている．

1. はじめに

Lisp は，各分野のプログラム，システム開発に用いられている．しかし並列性（マルチプロセス）をサポートしている Lisp は数少なく，また，プロセス関係の記述法なども，確立しているとは言い難い．

Lisp にマルチプロセスを導入することは，以下の意味において重要である．

1) マルチプロセッサ計算機に対して，Lisp が対応する必要がある．
2) Lisp の S 式を基本としたデータ構造が，様々なアプリケーションソフトウェアの開発に適している．しかも，それらのソフトウェアは，役割分担の明確なプロセスの集合体としてとらえると，素直に記述できる．したがって，Lisp としても，プロセスを扱えるのが望ましい．

Lisp に実際に並列性を導入するには，以下のふたつの方法を考えることができる．

1) 陰並列性
   プログラムを実行する際，並列実行の可能な部分を検出し，その部分について並列実行を行う．
2) 陽並列性
   並列実行することを，プログラム中で陽に指定する．

多くの実用的な Lisp のように，副作用がある場合は，陰並列性の検出はむずかしい．一方，陽並列性を導入すると，プログラマが並列実行を指定できるため，プログラマの自由度が増すと同時に，プログラム言語としての記述力も上がる．

本論文では，陽並列性を Lisp の言語仕様に入れるという立場をとり，Lisp における並列動作について議論し，並列動作の記述および実現の一方法を示す．まず第2章では，Lisp でのプロセスの構成要素，特に環境について述べる．第3章では，Utilisp[1),2)]にマルチプロセスを導入した mUtilisp における並行動作の記述およびその意味，評価方法，プロセス間通信などについて述べる．第4章では，mUtilisp のプログラム例をいくつか示す．また，第5章では，mUtilisp の環境の実現法について論じる．

まず，本論文中で使用する言葉を三つ定義しておく．

“プロセス”とは，並列に行われるプログラムの実行のおのおのをいう．ひとつのプロセスでは，プログラムを逐次的に実行する．

“関数”とは，普通の関数（car，cons など），マクロ，特殊形式（special form）をあわせていう．

“value cell”とは，シンボルの値を保持する場所のことをいう．

2. プロセスと環境

Lisp におけるプロセスの構成要素としては，

1) そのプロセスで実行すべき S 式
2) 環境

をあげることができる．ここで環境とは，

変数と値との対応
変数と関数定義との対応
変数と属性リスト（property list）との対応

† Programming and Implementation of a Multi-processing Lisp by HIDEYA IWASAKI (Department of Mathematical Engineering and Instrumentation Physics, Faculty of Engineering, University of Tokyo).

†† 東京大学工学部計数工学科

の三つを指す．各プロセスは，独自の環境の中で，プログラム（S 式）を実行する．結局，プロセスの数だけ環境があることになる．（このことを“多重環境”とよぶ．）

Lisp の実現法は，value cell をとる場所により，deep binding と shallow binding とにわけられる．マルチプロセスになると，プロセスは他のプロセスとは独立に環境を設定できる．したがって，シンボルのデータ構造の中に value cell をとる shallow binding を単純に用いては，マルチプロセスの実現はむずかしい．また，従来の Lisp では，関数定義や属性リストを，シンボルの中にとることが多かった．（言語仕様でそのように定められている場合もある．）これらも，shallow binding における value cell と同様の問題をひきおこす．マルチプロセスの Lisp では，value cell，関数定義および属性リストを全く同等に扱う必要がある．mUtilisp でとった方法は，次節で説明する．

ひとつのプロセスに対応する環境には上であげた三つがあるが，これとは全く独立に，次のような分類をすることもできる．

1) 先天的環境

そのプロセスが生成された時点で，既に与えられている環境，組み込みシンボルとそれに付随する値（nil など），関数定義（car，cond など）などがこれにあたる．

2) 後天的環境

そのプロセスにおいて，setq（代入），let（ラムダ束縛），defun，defmacro，putprop などを用いて確立した環境．

組み込みシンボルに関する先天的環境は，基本的には，すべてのプロセスが享受するものである．ここで問題となるのは，新たに生成したプロセスに先天的環境を与える手段である．プロセスを生成すると同時に，先天的環境を整備（たとえば，すべての組み込みシンボルの value cell，関数定義をコピーするなどして）するのは，メモリ効率や速度の面で問題がある．先天的環境を実現するには，何らかの特別な評価メカニズムが必要である．次章で，mUtilisp でとった方法を示す．

3. mUtilisp におけるマルチプロセスの実現

mUtilisp は，Utilisp の拡張版であり，マルチプロセスを導入している．ここでは，前章までの議論をふまえた上で，mUtilisp におけるマルチプロセスについて述べ，マルチプロセスの記述，その意味，および実現法などの提案を行う．

Utilisp は，shallow binding を用いた動的スコープを持つ Lisp[1] であり，近山によってメインフレーム上に開発され，その後，マイクロプロセッサ 68000 に移植された[3]．mUtilisp は，68000 版の Utilisp を拡張して開発した．言語として複雑になるのを避け，なるべく簡単な形でプロセスを導入することを基本方針としている．

mUtilisp は，プロセスごとにシンボル空間をわけることにより多重環境を実現し，プロセスどうしの干渉を避けている．また，mUtilisp は Utilisp を発展させたものであるため，動的スコープをもち，副作用もある．Friedman らの Lisp[4]（副作用がない），Multilisp[5),6)]（静的スコープをもち単一の名前空間の Lisp）とは，これらの点で異なっている．

現在，Lisp object としてのプロセスは，ベクタを用いて実現している．以後，そのベクタをプロセスベクタと呼ぶことにする．プロセスベクタの要素は，プロセス名（ストリング），シンボル名とシンボルの対応表（ベクタ），プログラムカウンタなど，プロセスに関する情報を含んでいる．プログラム中からは，プロセス名を表すストリング，そのストリングを印字名にもつシンボル，あるいはプロセスベクタそのものによって，プロセスを特定することができる．

3.1 プロセスの生成

プロセスは，特殊形式 fork を以下のように呼び出して生成する．

(fork *process-name* . *body*)

評価は，以下のように行う．*process-name* を評価した結果をプロセス名とするプロセスを，fork を実行したプロセスの子プロセスとして生成し，親プロセスに対しては子プロセスのプロセスベクタを即座に返す．（子プロセスの終了は待たない．）子プロセスでは，*body* を順次（progn で）評価する．

fork を実行することによって，プロセスを動的に生成することができ，プロセスは全体として木構造を構成する（**図 1**）．このプロセスの木構造を“プロセス木”とよぶ．

子プロセスは，親プロセスの環境を先天的環境として持つこととする．例えば，

```
(let ((foo 'foo))
  (fork "proc" (print foo)))
```

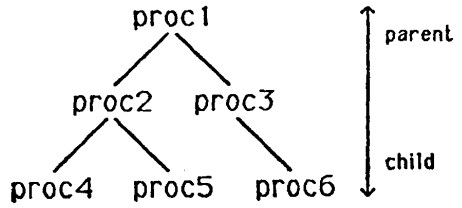

図 1 プロセスの木構造 (プロセス木)
Fig. 1 Tree structure of processes (process tree).

というプログラムで，プロセス proc における print の引数 foo を評価すると，foo (親プロセスの環境) が得られる．このようにすることにより，組み込みのシンボルと，そうでないシンボルを統一的に扱うことができる (次節参照)．

3.2 多重環境の実現

Utilisp では，shallow binding を行っている．したがって，mUtilisp でも，環境を shallow binding を基本として実現する．ここでは，変数と値の対応を例にとって説明するが，関数定義，属性リストも全く同様である．

実現方法のひとつとして，各プロセスでは同じ名前のインターンされたシンボルの実体は同じにし (すなわち，シンボル空間は全プロセスで共通にし)，その value cell を連想リスト (alist) 風に，

(プロセスベクタ . そのプロセスにおける値)

のリストにする方法が考えられる (図 2)．この方法は，値をとるときに alist の探索を必要とするため，本質的な作業は deep binding の場合と等しく，mUtilisp では採用しなかった (第 4 章参照)．

mUtilisp では，プロセスごとにシンボル空間をわけることによって，多重環境を実現する．すなわち，名前 (印字名) が同じインターンされたシンボルでも，プロセスごとに，実体は全く異なるものにする (図 3)．実際には，名前とシンボルの対応表をオブベクタ (obvector) のようなベクタで保持している．Common Lisp 風にいえば，プロセスごとに異なるパッケージを使用することになる．(以後，“プロセス p のシンボル空間に属すシンボル” を，単に “プロセス p に属すシンボル” ということにする.)

このような実現法をとると，特殊形式 fork の評価の際，*body* 部 (当然，fork を実行したプロセスすなわち親プロセスに属すシンボルによって構成されている) を，新たに生成するプロセス (すなわち子プロセス) に属すシンボルから構成されるように再構築 (このことを，*body* の子プロセスのシンボル空間への

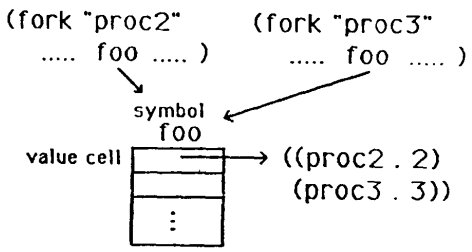

図 2 連想リストによる環境の実現
Fig. 2 Environment implemented as association list.

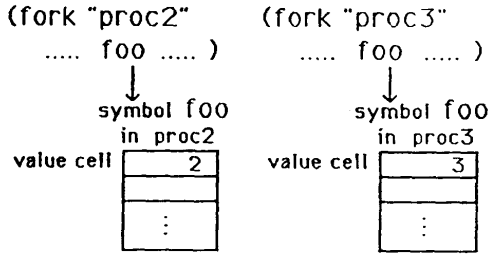

図 3 シンボル空間の分割による環境の実現
Fig. 3 Environment implemented by dividing symbol space.

“翻訳” という) しなければならない．インタプリタではその分オーバヘッドが増すが，コンパイルすれば，*body* の翻訳を実行時にする必要がなくなる．

シンボル空間をわけたため，car，cond，nil のような組み込みシンボル (およびそれらに付随する関数定義，値) は共有されず，このままでは新たに生成されたプロセスは，先天的環境を享受できない．そこで，プロセス木の根のプロセス (ROOT プロセスと呼ぶ) を常駐させ，ROOT プロセスのシンボル空間に，これらの組み込みシンボル (および値，関数定義) を保持させる．そして，子プロセスは，親プロセスの環境を継承することとする．環境継承の鍵は，印字名である．すなわち，子プロセスに属すシンボルの値は，もし value cell に値がなければ (未定義ならば)，印字名が同じ，親プロセスに属すシンボルの値を受け継ぐ．環境の継承は，親子間だけではない．印字名が同じシンボルが親プロセスにない場合，および，あっても未定義の場合は，さらに，上 (親の親) の環境をみにいく．すなわち，あるプロセスは，プロセス木において，ROOT プロセスにいたるパス上のすべてのプロセスの環境を継承する．この際，そのパス上，自分に近いプロセスの環境を優先する．ただし，シンボル空間をわけるという方針に従い，環境の継承によって得られた値は，必ず，自分のシンボル空間に翻訳して

返すこととする．以上をまとめると，以下のようになる．シンボル x をプロセス p で評価した結果を eval (x,p) とすると，

eval $(x,p)=$
  if val $(x)\neq\perp$ then val (x)
  lese ev1 (pname (x), parent (p), p);
ev1 $(name,p,q)=$
  if $p=\perp$ then $\perp$
  else if sym $(name,p)\neq\perp$
      and val (sym $(name,p))\neq\perp$
    then trans (val (sym$(name,p)$), q)
  else ev1 $(name$, parent (p), $q)$;

ただし，$\perp$ は未定義，val (x) は x の value cell の値，pname (x) は x の印字名，parent (p) は p の親プロセス (parent (ROOT)$=\perp$ とする)，sym $(name,p)$ は p に属し *name* という印字名をもつシンボル（そのようなシンボルが存在しないとき，および，$p=\perp$ のときは，sym $(name,p)=\perp$ とする)，trans (s,p) は s を p のシンボル空間に翻訳した結果を表すものとする．

たとえば，図1のようなプロセス木があり，proc1 が ROOT とする．

proc1 では，
  val (x)=100, val (y)=200, val (z)=300
proc2 では，
  val (x)=10, val (y)=20, val (z)=$\perp$
proc4 では，
  val (x)=1, val (y)=$\perp$, val (z)=$\perp$

の場合を考える．proc4 は，proc2，proc1 の環境をこの順の優先順位をもって継承する．proc4 において，x, y, z を評価すると，それぞれ，1, 20, 300 となる（**図 4**）．このようなシンボルの評価は，プロセス内では，shallow binding を行い後天的環境を得ており，プロセス木を親にたどる部分では，deep binding を行い先天的環境を得ていることに相当する．先天的環境を得る際のオーバヘッドを軽減するために mUtilisp でとった方法は 5.1 節で説明する．

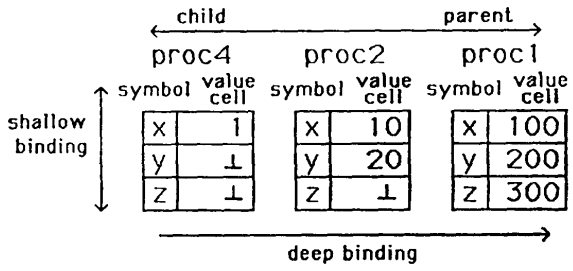

図 4 シンボル評価の例
Fig. 4 Example of symbol evaluation using process tree.

シンボルへ値を代入あるいは束縛するときは，プロセス木を親にたどることはせず，必ずそのシンボルの value cell に値を設定する．これは，後天的環境を設定していることになる．したがって，あるプロセスに属すシンボルが，他のプロセスによって値を変更されることはない．

3.3 通信および同期

プロセス間の通信には，シンボル空間をわけた方針に沿う，メッセージ伝達による方法を採用する．そのために，関数 send と receive を用意した．

メッセージの送信には，send を用いる．
  (send *process message*)
は，*process* というプロセス（名前でもプロセスベクタでもよい）に *message* を送信し，*message* を値として返す．送信は，非同期式に行う．

メッセージの受信には，receive を用い，
  (receive)
    または
  (receive *process*)
の形で呼び出す．前者は通信相手を指定せず，多対1のプロセス間通信に用い，後者は *process* というプロセス（send と同様，名前でもプロセスベクタでもよい）と1対1の通信を行う場合に用いる．いずれも，
  (送信プロセスベクタ . メッセージ)
という形のリストを返す．いずれにしても，メッセージの部分は，受信プロセスのシンボル空間に翻訳して返す．受信すべきメッセージがない場合は，メッセージが到着するまで，receive を実行したプロセスは待たされる．すなわち，封鎖型 receive (blocking receive) であり，これにより同期が実現できる．

このプロセス間通信は，S 式という構造を持ったメッセージを与えることができるため，Unix のパイプのような文字ストリームによる通信よりさらに強力なものとなる．

send と receive は，プロセスベクタにメッセージ受信用の場所（メイルボックス）を設け，メッセージはそこにリストの形で保持することによって実現している．

4. プログラム例

この章では，mUtilisp のプログラムの例をふたつ

```
(defun prime (to)
  (do ((next (fork (genprocname) (sift)))
       (i 2 (1+ i)))
      ((<= to i) (send next nil))
      (send next i)))
(defun sift nil
  (let ((p (cdr (receive))))
   (cond
     ((not (null p))
      (print p)
      (let ((next (fork (genprocname) (sift)))
            (n (+ p p)))
       (loop
        (let ((x (cdr (receive))))
         (cond ((null x) (send next nil) (exit)))
         (do nil
             ((<= x n)
              (and (< x n) (send next x)))
             (setq n (+ n p)))))))))
```

図 5 素数を生成するプログラム
Fig. 5 Program which generates prime numbers.

示す．プログラム中，genprocname は新しいプロセス名（ストリング）を作って返す関数とする．また，変数 current-process の値は，その変数が属すプロセスに対応するプロセスベクタとする．

4.1 素数の生成

図 5 に与えられた引数未満の素数を生成する関数 prime を示す．このプログラムでは整数列を，“ふるい” にかけるプロセス（fork によって動的につくられる）につぎつぎに渡していく．“ふるい” にかけるプロセスは関数 sift (prime を実行したプロセスの環境から定義が得られる) の実行により，最初に nil が送られなければ以下のことを行う．

最初に受け取った数（p とする）を，素数として出力する；

次の “ふるい” プロセス (next とする) をつくる；

次々と送られてくる整数列から，p の倍数を取り除いた整数列を，next に送る；

nil が送られてきたら，next にも nil を送った後，実行を終了する；

このプロセスが受け取る整数列からは，前の “ふるい” プロセス（の並び）により，p より小さい素数の倍数はすべて除かれているので，最初の数を素数と判定できるわけである．

4.2 引数の並列評価

図 6 に，与えられた三つの引数を並列に評価し，第 1 引数を，残りの引数に関数として適用するマクロ pfuncall2 を示す．(これは，Multilisp における pcall の簡単な版である．) pfuncall2 では，各引数の評価のためにプロセスを三つ作り，結果は receive で受け取っている．

```
(defmacro pfuncall2 (f x y)
  (lets ((p1 (genprocname))
         (p2 (genprocname))
         (p3 (genprocname)))
    `(progn
       (fork ',p1 (send ,current-process ,f))
       (fork ',p2 (send ,current-process ,x))
       (fork ',p3 (send ,current-process ,y))
       (funcall (cdr (receive ',p1))
                (cdr (receive ',p2))
                (cdr (receive ',p3))))))
```

図 6 引数を並列評価するプログラム
Fig. 6 Program which evaluates its arguments in parallel.

5. mUtilisp 実現法の検討

ここでは，mUtilisp の得失を，

1) shallow binding を基本とした環境
2) プロセスごとのシンボル空間の分割
3) プロセスの記述と通信

の面から論じる．

5.1 shallow binding

shallow binding の利点が，本方法の最大の利点となっている．すなわち，プロセス木を親にたどる必要がない限り，環境（すなわち後天的環境）の参照が高速な点である．シンボルの値への参照の多くは，後天的環境の参照であると考えられるので，この方法は非常に有利である．deep binding 風の実現 (3.2 節参照) でも，コンパイルすれば値の参照は高速化できるが，スペシャル変数に関しては，やはり連想リストの長さ分の時間がかかる．先天的環境を得るためにプロセス木を親にたどるのは，かなりのオーバヘッドになるが (deep binding でも同じ問題がある)，以下のようにして，それを軽減することができる．第一に，コンパイルして，car, cond などの呼び出しをインライン展開することにより，組み込み関数の定義を ROOT プロセスまでとりにいく必要がなくなる．第二に，先天的環境を得た場合には，それを後天的環境に設定するように，インタプリタを実現することである．(現在動いている mUtilisp インタプリタはこのように実現している．) 例えば図 4 の例では，proc4 において z を評価するときに，proc4 に属す z の value cell に値 (300) を設定する．このようにすると，ひとつの先天的環境への参照において，プロセス木の探索と，得られた環境の翻訳は，最初の一回だけ

行えばよい．その後，その環境は後天的環境として得られるために，時間的（プロセス木の探索の必要がない）および空間的（翻訳によるヒープの消費がない）に，インタプリタの性能が向上する．この実現法では，先天的環境を demand driven に取り込んでいることになる．

5.2 シンボル空間の分割

シンボル空間の分割は，プロセスどうしの予期しない干渉を未然に避けていることを意味する．環境の継承による評価メカニズムにより，ユーザは，シンボル空間がわかれていることを意識せずに，しかも，変数への相互排除的なアクセスというようなことも気にせずにプログラムが書ける．このことは，プログラムの書きやすさへの大きな助けとなる．

その反面，プロセスの生成時，先天的環境から値などを得るとき，および，プロセス間通信のときに，翻訳というオーバヘッドが生じる．はじめのふたつの場合は，5.1 節後半で述べた方法によってかなり軽減できる．

5.3 プロセスの記述と通信

Multilisp には，与えられた引数を並列に評価する pcall という関数があり，これが並列動作を指定するひとつの方法になっている．fork は pcall と比べて，

1) 引数の並列評価ということに言及してないので陰並列性を導入する可能性を残している．
2) プロセスに名前がついているので，特定のプロセスを名指しして通信できる（3.3 節参照）．

などの利点がある．

send, receive による，通信相手の指定が可能なプロセス間の通信は，共有変数を用いての通信と比べ，プロセス間のインタフェースが明確になっており，ネットワークを通しての通信にも拡張可能である．また，Lisp オブジェクトのストリームをプロセスにすることにより，send, receive を print, read と統合することができる．

6. むすび

本論文では，Lisp のプロセスおよび環境について考察し，mUtilisp を通して，プロセスの記述，実現法などについての提案を行った．

現在，mUtilisp を使った応用プログラムとしては，画面エディタがある．これは，エディタを四つのプロセス（コマンド入力処理部，エディタの内部バッファ更新部，画面表示バッファ更新部，実際の画面更新部）の集まりとして記述している．そのため，コマンド入力処理プロセスを，ユーザ自身が記述することにより，カスタマイズが容易な，柔軟なシステムとなっている．

mUtilisp のインプリメントに関しては，プロセスのベクタによる実現，プロセス切り替えのタイミング（現在は，関数呼び出しを一定の回数行うとプロセス切り替えをおこすようにしている）など，まだ洗練されていない部分がある．

今後，mUtilisp の処理系を充実させ，様々な応用プログラムの開発を通して，mUtilisp の有効性を立証していく必要がある．

謝辞 本研究に際し，有益なご助言をいただいた，東京大学工学部計数工学科の和田英一教授，寺田実助手，電気通信大学電子情報学科の多田好克助手に感謝いたします．

参考文献

1) 近山 隆：Utilisp システムの開発，情報処理学会論文誌，Vol. 24, No. 5, pp. 599-604 (1983).
2) Chikayama, T.: Utilisp Manual, Technical Report, METR 81-6, Dept. of Mathematical Engineering and Instrumentation Physics, Faculty of Engineering, Univ. of Tokyo (1981).
3) 和田英一，富岡 豊：UtiLisp の 68000 への移植，情報処理学会記号処理研究会資料，84-29 (1984).
4) Friedman, D. and Wise, D.: Aspects of Applicative Programming for Pararell Processing, *IEEE Trans. Comput.*, Vol. C-27, No. 4, pp. 289-296 (1978).
5) Halstead, R. H.: Implementation of Multilisp: Lisp on a Multiprocessor, *Proc. ACM Symp. Lisp and Functional Programming*, pp. 9-17 (1984).
6) Halstead, R. H.: Multilisp: A Language for Concurrent Symbolic Computation, *ACM Trans. Prog. Lang. Syst.*, Vol. 7, No. 4, pp. 501-538 (1985).

(昭和 61 年 10 月 1 日受付)
(昭和 62 年 2 月 12 日採録)

岩崎 英哉（正会員）

1960 年生．1983 年東京大学工学部計数工学科卒業．1985 年同大学院工学系研究科情報工学専門課程修士課程修了．現在同大学院博士課程在学中．プログラミング言語，オペレーティング・システム，文書処理などに興味をもつ．