1E−5

Multilispの操作的意味及び実現

浅井 健一　松岡 聡　米澤 明憲
(東京大学理学部情報科学科)

1 はじめに

近年、並列 Lisp が関数型言語の潜在的な並列性を大きく引き出せるものとして注目されている。実際に Multilisp[3] をはじめとして Multischeme[6], Mul-T[5], Qlisp[2] などたくさんの並列 Lisp が開発され並列計算機上で高い性能が報告されている。しかし、現在のところ並列 Lisp は並列計算機上での性能を向上させることを目的としているのでもっぱら性能に関しての議論がなされ、言語の意味に関する考察はほとんどなされていない。そのため言語仕様があいまいになるし、言語仕様の変更も難しくなっている。このことはスケジューリング方式の固定化を引き起こし、ひいては自己反映計算 [8] の実現を難しくしている。そこで Multilisp の操作的意味記述 [1] を与え、これを用いて逐次型計算機上に Scheme[7] によるインタプリタを作成した。さらにこれをもとに表示的意味記述を与える。またその記述から導かれる future と call/cc との相互干渉について述べる。

2 Multilispの並列機構

Multilisp の特徴的な機能は future という命令にある。(future <exp>) という命令を実行すると <exp> を評価する子プロセスが作られる。親プロセスにはプロミスが (future <exp>) の値として即座に返される。プロミスというのは将来いつの日かに値が確定するという手形のようなものである。親プロセスの計算は子プロセスの終了を待たずに進められるので並列実行が実現される。子プロセスが評価を終了して値を返すとプロミスの値はその返り値に確定する。car のように引数の値が確定しないと実行が進まない命令は値が確定していないプロミスに出会うとブロックしてしまうが、cons のようにすぐに引数の値を必要としない命令中で使うと大きな並列性を引き出すことができる。

3 プロセス

並列性を限られた数のプロセッサで実現するには並列に動くプロセス間でコンテキストスイッチを行なう必要がある。そのためにはプロセスの情報を保存しておかなくてはならない。ここではプロセスを中間結果と継続 (continuation) のペアと考える。すなわち現在までに行なわれた計算の結果と今後その結果に対して行なわれる操作とのペアと考える。例えば中間結果が (cons 1 2) で継続が (-eval- -driver-loop-) だったならそのプロセスは今後 (cons 1 2) を評価してトップレベルに戻るという動作を行なう。この継続を詳しく記述してやることでプロセスが1ステップ計算を進めるごとにコンテキストスイッチを行なうという非常に細かなプロセス制御を行なうことが可能となる。

継続の表現方法は上の例でもわかるように明示的にリストを作ることとした。Scheme の call/cc を使うことも考えられるが、これだと call/cc 自体の意味が明確に記述されないという欠点がある。また明示的にリストを作るとより表示的意味記述に近くなるのでこの方法を採用した。

継続に関してひとつ注意しなくてはならないのは、この継続はそのプロセスが今後何をするかを示すもので、他のプロセスに関する情報は一切持たないということである。これはプロセスがそれぞれ独立していると考えることに相当しており素直な解釈である。

4 プロミス

プロミスという新しいデータ構造が導入されたので、これを従来のデータと区別するためにタグが必要である。(表示的意味を与えるときには injection が使えるためタグを導入する必要はない。) そして各 primitive procedure 処理ルーチンでは引数がプロミスであった場合の処理を書いてやらなくてはならない。しかし、これは関数の性質を考察することで一か所にまとめることができる。すなわち関数が cons だったならそのプロミスを引数としてそのまま渡し、それ以外だったならプロミスの値が確定するまでブロックすればよい。具体的には次のようになる。

```
(define (apply-primitive-procedure
                    p args processes promises)
  (let ((name (primitive-id p)))
    (cond ((eq? name 'cons)
           (eval-cons args processes promises))
          (else
           (touch-list (cons p args)
                       (add-one-cont '(-basic-apply-)
                                     processes)
                       promises)))))
```

else 節中では関数 p と引数 args の値を touch-list に渡して確定させた後、関数適用 -basic-apply- を行なうよう指示している。関数適用は自分の継続に -basic-apply- を加えることで touch-list の実行が終了すると行なわれるようになっている。

プロミスは普通の変数と同じく多くのところから参照される可能性がある。そのため実現のためには location を使う必要がある。location というのは記憶領域を抽象化したもので、上の例では promises の部分にあたる。普通の変数に関しても同じく location を使って実現するべきであるが、これについてはすでに多くの研究がなされているので一般的な副作用を持った関数を使うことにした。

5 スケジューリング方式

スケジューリング方式には次のようにもっとも簡単なラウンドロビンを用いた。

```
(define (schedule-process processes)
  (append (cdr processes)
          (list (car processes))))
```

しかし、独立に書き換えることによってどのようなスケジューリング方式にすることも可能である。今回の実現はこの部分に非決定性が含まれていないため、システム全体としても非決定性が排除されている。

Operational Semantics of Multilisp and its Implementation
Kenichi Asai, Satoshi Matsuoka, and Akinori Yonezawa
The University of Tokyo

6 future と call/cc の相互干渉

実際にインタプリタを作成し使ってみると種々の面白い結果が得られた。そのうちのひとつに call/cc と future の相互干渉があげられる。次の例は実行するとトップレベルが複製されてしまうものである。

```
==> (call/cc (lambda (f) (future f 1) 2))
2                   ;親プロセスの結果
==> (car '(a b))    ;親プロセスのプロンプト
1                   ;子プロセスの結果
==> 3               ;子プロセスのプロンプト
a                   ;親プロセスの結果
==> 4               ;親プロセスのプロンプト
3                   ;子プロセスの結果
==> 5               ;子プロセスのプロンプト
4                   ;親プロセスの結果
...                 ;...
```

トップレベルで call/cc を使っているので、ここで保存される継続は「結果を返しトップレベルに戻る」というものである。この継続を子プロセスが呼び出すとトップレベルが2つになってしまう。2行目ではたまたま実行時間が短かった親プロセスが結果を返しプロンプトを出したが違う式を入力してやると4行目で今度は子プロセスが実行を終了して結果を返しプロンプトを出している。結果を返すのはラウンドロビンで先に実行を終了した方である。また逆に

```
==> (future call/cc (labmda (f) (set! f1 f) 1))
```

として大域変数 f1 に子プロセスの継続をセットした後

```
==> (f1 2)
```

のようにトップレベルで継続を呼び出してみよう。保存される継続は「結果をプロミスの値として確定する」である。注意しなくてはならないのはここに「トップレベルに戻る」が入っていないことである。そのために (f1 2) を実行するとトップレベルがなくなってしまいシステムエラーとなる。これらの相互干渉は call/cc の命令の問題点を指摘している。このことは我々が表示的意味記述として次節で与えたような言語の仕様が確定してはじめて理解されることである。

7 表示的意味記述

ここで作成したインタプリタはほとんど副作用を使っておらず、また継続に基づいて作られているため簡単に表示的意味記述に書き換えることができる。この記述の一部を図1に示す。ここで P はプロセスを Ps はプロセス空間を表している。expression continuation にプロセス空間を渡すことでプロセス全体の情報を伝えている。記述の詳細は [4] を参照されたい。この表示的意味記述は今回作成したインタプリタに対して非常に忠実であり例えば前節での call/cc の異常な振舞いもこの記述から導き出せるものである。

8 おわりに

Multilisp を逐次型計算機上に実現することでその操作的意味を与えた。またそれに基づいていろいろな面白い結果が得られた。さらにこの実現に基づいて表示的意味記述も与えた。この研究が今後、意味の形式的な議論の1ステップになれば幸いである。

謝辞

渡部卓雄、脇田建両氏には多くの助言をいただいた。この場をかりて感謝の気持ちを表したい。

領域方程式 (一部)

$\alpha \in \mathtt{L}$ locations
$\alpha_w \in \mathtt{W}$ promise locations
$\epsilon \in \mathtt{E} = \mathtt{Q} + \mathtt{H} + \mathtt{R} + \mathtt{E_p} + \mathtt{E_v} + \mathtt{E_s} + \underline{\mathtt{W}} + \mathtt{M} + \mathtt{F}$ expressed values
$\sigma \in \mathtt{S} = \mathtt{L} \to (\mathtt{E} \times \mathtt{T})$ stores
$\sigma_w \in \mathtt{S_w} = \mathtt{W} \to (\mathtt{E} + \mathtt{P_s}) \times \mathtt{T}$ promise stores
$\rho \in \mathtt{U} = \mathrm{Ide} \to \mathtt{L}$ environments
$\theta \in \mathtt{C} = \mathtt{S} \to \mathtt{S_w} \to \mathtt{A}$ command continuations
$\kappa \in \mathtt{K} = \mathtt{E}^* \to \underline{\mathtt{P_s}} \to \mathtt{C}$ expression continuations
$\beta \in \mathtt{P} = \mathtt{E} \times \mathtt{K}$ processes
$\omega \in \mathtt{P_s} = \mathrm{Set\ of\ P}$ process spaces

意味関数 (future 部)

$\mathcal{E} : \mathrm{Exp} \to \mathtt{U} \to \mathtt{K} \to \underline{\mathtt{P_s}} \to \mathtt{C}$

$\mathcal{E}[\![(\mathbf{future}\ \mathrm{E})]\!] =$
$\lambda\rho\kappa\omega\,.\,\lambda\sigma\sigma_w\,.\,\mathit{new\text{-}promise}\,\sigma_w \in \mathtt{W} \to$
$\mathit{send}\,(\mathit{new\text{-}promise}\,\sigma_w\,|\,\mathtt{W})$
κ
$(\mathit{add\text{-}process}$
$(\mathit{mk\text{-}process}$
E
$(\lambda\epsilon\omega\,.\,\mathcal{E}[\![\epsilon]\!]\,\rho$
$(\lambda\epsilon\omega\sigma'\sigma'_w\,.\,\mathit{assign\text{-}promise}$
$(\mathit{new\text{-}promise}\,\sigma_w\,|\,\mathtt{W})$
ϵ
$(\mathit{run}$
$(\mathit{add\text{-}processes}$
$(\sigma'_w\,(\mathit{new\text{-}promise}\,\sigma_w\,|\,\mathtt{W})\downarrow 1)\,|\,\mathtt{P_s}$
$\omega))$
σ
$\sigma'_w)$
$\omega)$
$\omega)$
σ
$(\mathit{update\text{-}promise}\,(\mathit{new\text{-}promise}\,\sigma_w\,|\,\mathtt{W})\,\langle\rangle\,\sigma_w),$
wrong "out of memory" $\sigma\,\sigma_w$

図 1: Multilisp の表示的意味記述の一部

参考文献

[1] Allison, L. *A practical introduction to denotational semantics*, Cambridge: Cambridge University Press (1986).

[2] Goldman, R., R. P. Gabriel, and C Sexton "Qlisp: Parallel Processing in Lisp," US/Japan Workshop on Parallel Lisp, June (1989).

[3] Halstead, R. H., Jr. "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Trans. Prog. Lang. Syst.*, Vol. 7, No. 4, pp. 501–517 (1985).

[4] Kenichi, A. "Multilisp Implementation," 東京大学理学部情報科学科卒業論文 (1990).

[5] Kranz, D. A., R. H. Halstead, Jr., and E. Mohr "Mul-T: A High-Performance Parallel Lisp", US/Japan Workshop on Parallel Lisp, June (1989).

[6] Miller, J. *Multischeme: A Parallel Processing System Based on MIT Scheme*, Ph.D. thesis, M.I.T. E.E.C.S. Dept., Cambridge, Mass., August (1987).

[7] Rees, J., and W. Clinger *Revised*^{3} *Report on the Algorithmnic Language Scheme*, SIGPLAN NOTICE, Vol. 21, No. 12, December (1986).

[8] Smith, B. C. "Reflection and Semantics in Lisp," *Proceedings of ACM Symposium on Principles of Programming Languages (POPL)*, pp. 23–35, (1984).