6G−2

遅延ナローイング抽象機械のシミュレータ

中川康二、　鈴木太朗、　中村敦司、　井田哲雄
筑波大学

1　はじめに

我々は、遅延ナローイング計算系に基づいた関数・論理型言語Evを設計し、その処理系を開発した[1]。この処理系は、コンパイラの生成した命令列を抽象機械のシミュレータで実行することにより実現している。本稿では、処理系の構成について述べ、遅延ナローイング抽象機械の命令を形式的に定義し、その形式的定義に沿った形でシミュレータを構築する方法について述べる。

2　遅延ナローイング言語Evの処理系

Ev処理系の構成を図1に示す。Ev処理系は、対話的なシステムであり、次の3つの要素からなる。

(1) シミュレータ部
翻訳された命令列を実行する抽象機械をシミュレートする。以後、この抽象機械のことをLNM(Lazy Narrowing Machine)と呼ぶことにする。このEv処理系では、LNMをCommon Lispによりシミュレータとして実現している。3節でその概要を説明する。

(2) コンパイラ部
Evのプログラムを LNMの命令列へ翻訳する[2]。得られた命令列はファイルに出力され、インタフェース部を介してLNMのシミュレータに渡される。コンパイラは、SICStus Prologにより実現されている。

(3) インタフェース部
LNMのシミュレータとコンパイラのプロセスを起動し、プロセスの制御およびデータの受渡しを管理する。具体的には、コンパイルの要求、命令列ファイル名の受渡し、結果の出力を行なう。インタフェース部は、Emacs Lispにより実現されている。

この処理系は、遅延ナローイングに基づく言語処理系を研究するために設計されたものである。そのため既存の言語処理系の機能を活用し、効率よりも仕様の変更に対して柔軟であることと、処理系を正確に短期間で実現することを重視している。

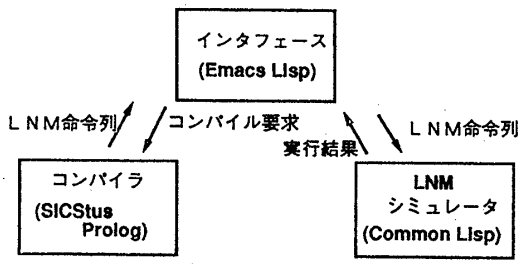

図1.　Ev処理系の構成

3　遅延ナローイング抽象機械 LNM

LNMは、Prologの抽象機械として知られるWAM (Warren Abstruct Machine)を基盤にしたものである[3]。LNMは、次の3つの要素からなる。

(1) プログラム空間
(2) 状態 *State*
(3) 状態遷移写像 $\mathcal{M}$

プログラム空間とは、LNMの命令列の格納領域である。以下、状態と状態遷移写像について説明する。

3.1　状態 *State*

LNMの状態 *State* は、次の9つ組で表現される。

$$(p, heap, stack, trail, a, e, b, cp, s)$$

ここで、p はプログラムポインタ、*heap* はヒープ領域、*stack* はスタック領域、*trail* はトレイル領域、a は一時的に用いられるレジスタ、e は環境ポインタ、b はバックトラックポインタ、cp は継続ポインタ、s は構造ポインタを表す(図2参照)。

LNMの項は、タグ付けされたデータ構造で表現される。例えば、図3(a)は定数3を表わしている。WAMと同様に、未束縛変数は、図3(b)のように自分自身へのポインタで表現される。リスト[a,b]は図3(c)のように表現される。*stack* や *heap* は、このタグ付けされたデータ構造により構成されている。

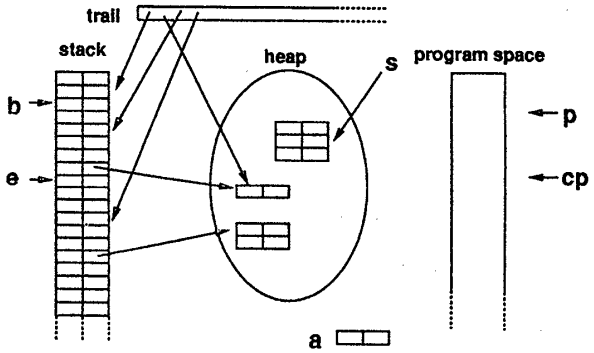

図2.　抽象機械の構造

Simulator of Abstract Lazy Narrowing Machine
Koji Nakagawa, Taro Suzuki, Atsushi Nakamura, Tetsuo Ida
University of Tsukuba

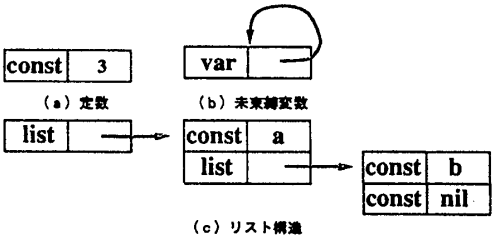

図3. LNMのデータ構造

3.2 状態遷移写像 $\mathcal{M}$

状態遷移写像 $\mathcal{M}$ を次のように定義する。*Instr* を命令の集合とし、*STATE* をLNMの状態の集合とすると、$\mathcal{M}$ は

$$Instr \times STATE \rightarrow STATE$$

の写像である。$state_k$ と $state_{k+1}$ をそれぞれ命令実行前と命令実行後の状態とすると、状態遷移写像 $\mathcal{M}$ は、次のような命令の形式的な記述の集合として定義される。

$$\mathcal{M}[\![\ \langle 命令 \rangle\ ]\!]\ state_k \quad \Longrightarrow \quad state_{k+1}$$

例えば、命令 get_variable は次のように形式的に記述される。

$\mathcal{M}$[get_variable i] $(p, heap, stack, trail, a, e, b, cp, \bot)$

$\Longrightarrow (p+1, heap, stack[i \leftarrow a], trail, a, e, b, cp, \bot)$

$stack[i \leftarrow a]$ は、*stack* の i 番地で示される場所に、a レジスタの内容が書き込まれることを意味している。

4 シミュレータの実現

LNMのシミュレータはLispにより記述されている。タグ付けされたデータ構造をconsセルにより図4(a)のように実現する。スタックはベクタによって図4(b)のように実現する。

LNMの状態を次のように大域変数で実現する。

```
(defvar P ...)
(defvar STACK (make-array ...))
    ...
```

状態遷移写像 $\mathcal{M}$ の定義に従って、各命令は、状態を変化させる処理を行うLispの関数として実現される。状態を変化させる処理を複数のより基本的な処理からなると考え、これらを $op_1 \cdots op_n$ として一つの命令を次のように実現する。

(defun ⟨命令の名前⟩(⟨命令の引数⟩)

op_1

$\vdots$

op_n)

例えば、get_variable 命令は、次のように実現される。

```
(defun GET_VARIABLE(i)
  (set-stack i A)
  (countup P))
```

これは、get_variable 命令が、スタックのi番地に a レジスタ (A) の内容を書き込むという操作とプログラムポインタ (P) をカウントアップするという操作を状態に作用することで実現されていることを意味している。

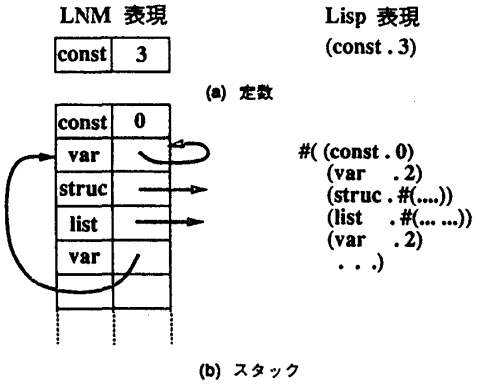

図4. Lispによる表現

5 おわりに

本稿で説明したシステムでは、効率よりも仕様の変更に対して柔軟であることと処理系を正確に短期間で実現することを重視している。

ここでは、既存の言語の機能を活用してEvの処理系の核の部分だけを記述している。LNMのシミュレータにおいては、メモリの管理等にLispの機能を用いており、コンパイラにおいてはEvの構文解析にPrologの構文解析を用いている。このため、短期間でEv処理系を記述でき、仕様の変更に柔軟に対応できる。実際に、短期間でこのEv処理系を構築することができた。

また、処理系を正確に実現するために、形式的な定義をもとに命令を状態変化の複数の操作とみてLispの関数を記述し、LNMを構築した。形式的な定義により、抽象機械の仕様を厳密に定義できる。さらに、形式的な定義に沿ってプログラムを実現しているので、プログラムの可読性が向上する。

一般に、プロトタイプの処理系を構築する際には、これらのことを重視して、処理系の実現技法を洗練していくことが重要である。こうして、処理系の検討が十分になされた後に、C等でシステムを記述することで、効率化することが可能であろう。

今後の課題として、このシミュレータを用いてLNMの実行効率を測定し、さらに最適化の技法を研究していく予定である。

参考文献

[1] 鈴木太朗他. 遅延ナローイング計算系に基づく言語Evとその処理系. 情報処理学会第44回全国大会予稿, 1992.

[2] 鈴木太朗他. 遅延ナローイングに基づく言語Evの等式翻訳方法. 情報処理学会第44回全国大会予稿, 1992.

[3] 中村敦司他. 遅延ナローイング抽象機械のアーキテクチャ. 情報処理学会第44回全国大会予稿, 1992.