New Unified Environment (NUE) の基本構想

竹内郁雄・奥乃博・大里延康・渡邊和文・日比野靖
(日本電信電話公社・武蔵野電気通信研究所)

1. はじめに

NUE (New Unified Environment) は、① Lisp machine ELIS [1] 上で、②主として玄人 programmer(s) が、③ display 端末を介して、④高度に対話的に、⑤大規模な programming を行うための総合的 programming 環境として構想されたものである。現在は、基盤となる hardware ELIS が完成し、その上のいわば「核」言語 (我々は TAO Lisp と呼ぶ) の仕様が具体化しつつある段階である。

試作した ELIS は、LSI 技術の進展を見越した単純で大柄の architecture を持つ。主な spec は次の通りである。

- 主記憶実装 512 Kcell (64 bit/cell, 8 bit の tag 2個/cell を含む。記憶空間は 16 Mcell。16 Kbit の memory chip 使用) ポートレベルでのアクセス 420ns
- スタック 32 Kword (32 bit/word) アクセスタイム 60ns.
- WCS 16 Kword (64 bit/word) 80ns. 1μ命令当たり 180ns が標準.
- 速度 試験的な coding によれば、(CAR something) の interpreter での実行時間は約 3μ秒. (something の評価時間は除く) 0.3 MLips? (Lisp instruction/sec!)

NUE (鵺に通ず(*)) の image を一言で表わすと; その名の通り、頭は Lisp, 体は TOPS-20 + Unix 少々, 手足は emacs, 尾は Prolog, 声は Teitelman に似ている。一見正体不明で、PL/I 病に罹っているようにも見えるが、我々は Lisp の枠組のもとに、従来的 OS 核や command processor, file system, editor 等を、概念的にも実質的にも融合一体化した高性能の system として仕上げる計画である。なお、設計目標に玄人向けと謳ったのは、こうしておけば素人向けへの modify が容易だからである。(逆は困難)

本論文では、NUE の一番核心的な問題である人間機械系における対話機能の話題は一切省略し、NUE の基盤を Lisp にした理由と、仕様の固まりつつある「核」言語についての "state of the art" について報告する。

2. 何故 Lisp が枠組なのか?

対話型言語として Lisp が優れている理由の多くは、Sandewall の論文[2]に見ることができるが、我々の場合むしろ Lisp というより、S式という「前言語的言語」を基本枠組として選んだと言えよう (図1)。S式は、1次元の文字列を構造化するための必要最小限の文法 (カッコと atom) を持つ。その上の基本演算 car, cdr, cons, atom, eq で S式について我々は語ることができる。Lisp はこの S式の上に λ算法をモデルにした計算意味論を与えたものであり、現実に過去 20 年間先進的 programmer 達によって使われてきた。programming 言語も1つの言語文化であるとしたとき、「使われてきた」ということの意味は非常に大きいと我々

---

*) 鵺 (ねえ) (または鵼) 名. ①[動] とらつぐみの別名. ②想像上の動物. 頭は猿, 体は狸, 手足は虎, 尾は蛇, 声は鵺 (ぬえどり) (再帰的定義!) に似ている. 妖怪年表によれば, 1933 年京都に出没し; 人心を不安に陥れて以来出現の記録なし. ③(転じて) 正体不明の人物.

は考える。

しかし、実はS式の上の計算意味論は入算法に固定しているものではない。S式自身は、記号の列に必要最小限の構造を持たせた無色のものである。だから、Lispの枠組内でも、S式の持つ意味をいくらでも拡張し深化させることが可能であった。つまり、S式という「前言語」は如何なる計算意味論をも吸収し得る器なのであって、我々はS式に新しい意味を付与する自由度を常に持っている。現状でも、Lispは関数型にも手続き型にも使える適応性を持っている。これほど多くのしかも相当根本的なところで異なる方言が異和感なく共存し得るのもその現れだろう。今後起こり得る新しい計算意味論への要求に一番速く対処し、成長し続けるのはS式という器に(のみ)基づいているLispだと我々は考えている。

```
TAO produced the NIL. The NIL produced
the atom. The atom produced the
S-expression. And the S-expression
produced the ten thousand things. The
ten thousand things carry the tag and
embrace the bug, and through the
blending of the unknown force they
achieve harmony.

TAO Lisp                          Lao-Shu.
```

図1 「前言語的言語」S式

3章で述べるように、TAO LispはPrologに代表される簡易形resolution(後述)を従来のLispの計算意味論に融合させた。述語論理型programmingは使ってみると便利なこともあるからである。TAO Lispを意味論的にFortranと両立させるような言語仕様も取り入れる。なんといってもFortranは便利なことが多い。(FLATSは何年も前からそう主張していた[3])。NUEの名の通り、TAO Lispの発想は便利なものは取り込むことであり、見方を変えれば、Lisp, Prolog, Fortran等,如何様にでも見えるのである。(PrologやFortranがLispを取り込むことの不自然さや困難さを考えよ。)

S式についてもう1つ注意しておかなければならないのは、人間と機械の間の「共通言語(深層言語)」としてのS式の優位性,自然さである。S式は入出力が常に整形されていれば人間にとって決して扱いにくいものではない。もちろん機械にとってS式は機械内部表現と1対1に対応している。我々はS式が,人間と機械の双方向通信にとって,少なくとも現時点の我々の技術レベルにおいて,最も自然で直接的であると考える。(表層言語は情報の伝播が一方向のときに,より有効である。)

3. Logic Programming in TAO Lisp

3.1 Prologの何をTAO Lispに取り込むのか?

述語論理型言語として有名なPrologの長所としては、① pattern matchingによるリストの構成,分解が可能,②複数出力が可能,③部分的にinstantiateされた構造が扱える,④ゴール指向programmingが可能,といった点を挙げることができる。一方,短所としては,⑤エラー処理が劣る,⑥デバッグ機能が弱い,⑦制御構造が貧弱,といった点がある。Prologの短所は貧弱な会話環境に起因している。programming環境として充実しているLispの中にPrologを埋め込むことで,これらの短所を補おうという試み — LOGLISP[4], PROLOG/KR[5], DURAL[6]など — が行われている。しかし,これらの試みはあくまでLispの表層で行われており,処理効率は当然Lispより劣っていた。ただ,LOGLISPでは,Lisp単純化と呼ぶreductionにより,導出(resolution)と評価(evaluation)を統合しようと試みているが,Lisp単純化だけでは導出の評価への喰い込みが足りない感があり,

logic と Lisp が完全に融合しているとは言い難い。

TAO Lisp では、logic（述語論理型）programming の essence として pattern matching（from-left-to-right unification）の機能と short range の backtracking 機能とを Lisp の中に取り込んだ。これらの機能は TAO Lisp と不可分のものである。しかし、programmer は、その機能を使わず、Lisp として TAO Lisp を用いることもでき、かつ、Lisp program の実行に当っては、これらの機能は何らの overhead ももたらさない。また、TAO Lisp は述語論理型言語としても用いることができ、その実行は、Lisp 上の同種の言語よりも格段に速い。「便利なものは処理速度を落さず何でも取り込む」という NUE の精神はここでも生かされている。TAO Lisp では、logic programming の取り込みにおいて、logic としての意味解釈の整合性、厳密性は Prolog と同様、余り問題にしない。TAO Lisp で新たに導入した概念は relation と pattern である。

Logic Programming の機能を Lisp の中に取り込む上で問題となるのは次の点である。

ⓐ Lisp レベルにおける relation (logical expression) の意味
ⓑ Logic レベルにおける Lisp form の意味
ⓒ 論理変数 (logical variable) のスコープ、Lisp 変数との共存
ⓓ backtracking control, Lisp form との共存

TAO Lisp ではこれらの問題点をどのように解決しているかを以下で述べる。

3.2 Lisp と Logic との融合

Logic programming というリテラルにあたるものを TAO Lisp では relation と呼ぶ。TAO Lisp の実行モードには評価モード（evaluation mode）と導出モード（resolution mode）の2つがある。評価モードは原則として form を対象としているのに対し、導出モードは原則として relation を対象としている。しかし、2つのモードは厳密に2分割されている訳ではなく、実行すべき対象をどう見るかの順位を示していると考えればよい。つまり、評価モードでは対象をまず form と見、もし form でなければ relation とする。表1に、2つの実行モードと、その処理をまとめておく。

モード＼順位	第1順位	第2順位
評価モード	form 評価する	relation　導出を実行（一時的に導出モードに入る） 成功の場合は instance を 失敗の場合には NIL を返す。 統一化は破壊し、unifier は残さない［3.4参照］
導出モード	relation 導出を実行する	form　評価する（一時的に評価モードに入る） 値が non-NIL ならば成功；NIL ならば失敗。 Hinge 関数の場合、統一化は保持し、unifier を残す［3.4参照］

表1. 実行モードの処理内容

両モードとも macro は展開してから、もう一度同じモードで実行する。

もし、同じ atom に対して Lisp 関数と relation の両方が定義されていて、かつ、導出モードで Lisp 関数の方を実行したい場合には、form の前に ` (backquote, ASCII コード 140_8) を付ける。一般に同一 atom に対して Lisp 関数と relation の両方が定義されることは少ない。特に、TAO Lisp では atom 名が oblist グラフ［7］で管理され、複数名前空間（同一名の atom reader が複数個存在し得る。）が使用できるので、このような状況は極めて稀である。従って、各モードにおける default の考え方は極めて自然なものである。

Pattern は relation 中の引数（項 term）として現われる。Pattern には、論理変数か構造データ、あるいは form が許される。論理変数の表現法は read handle によって

自由に定義できる。例えば、DEC-10 PROLOG のように大文字で始まるatomを論理変数にすることも自在である。ここでは便宜上論理変数は＊で始まるatomとする。Formと構造データとは両者ともS式であり区別が付かないので、formの前に`(backquote)を付けて区別する。`はS式の任意の場所で使うことができる。例の(POSITION `lat `lon) はpatternであり、その中にformが現われている。formが現われた場合は、formの値でその部分が置き換えられる。もちろん論理変数が構造データの中に現われてもよい。同じ例の(PLACE *x)を見よ。

論理変数が評価モードで実行された場合には、そのinstanceが値として返される。例えば、*x←*u, *y←(*u. *v), *z←(*u. *v) [←はinstanceを表わす]が与えられているとしよう。このとき、次のような結果が得られる。

(equal *x *u)	値Tが返される
(equal *x *y)	値NILが返される
(equal *y *z)	値Tが返される
(greaterp *x *u)	エラーが生ずる

記号'は'(QUOTE)同様read-symbol macroの1種である。'は導出モードだけでなく、評価モードでも使うことができる。例えば、変数xのもつ値に代入したい場合、通常は (apply 'setq (list 'x value)) と書く。しかし、'を用いて

(setq 'x value)

と書くこともでき、この方がapplyという余計な関数を使用しなくてもよいので簡潔な表現となっている。

3.3 Hinge関数

Hinge関数は、実行モードを導出モードに変更し、論理変数のスコープの管理やbacktrackingのためのスタックの管理を行うものである。Hinge関数を次に示す。

(& relation . . .)	: and に対応
(! relation . . .)	: or に対応
(~ relation)	: not に対応

関数&はlogic programmingにおける問合せ(←B1…←Bn [Biはリテラル]という形のホーン節)に対応している。&の中のrelationは左から順に実行される。実行の方法は変数の出現チェックを行わない順序付統一化(from-left-to right unification without variable occur check)による導出であり、通常のPrologと同じ方法である。もし統一化が失敗に終れば&はNILを返す。成功した場合には最後に実行したrelationのinstanceを&の値として返す。(&)=T である。

関数!はORを表わし、引数で与えられるrelationは各々がホーン節に対応している。従って、各relationに現われる論理変数はそのrelation内で局所的(local)である。!の実行は&と同様に左から順にrelationを実行し、一つでも成功すれば!は成功したrelationのinstanceを!の値として返す。もし、すべてのrelationが失敗すれば!の値はNILとなる。(!)=NILである。!を用いると条件文を簡単に書くことができる。(& A (if p then B else C) D) と書きたいときには、

(& A (! (& p B) C) D)

と書けばよい。!は決定性ORであり、非決定性ORはlogicで定義できる。

関数~はNOTを表わし、relationの実行が成功すればNILを、失敗すればTを返す。

&, !, ~はLisp関数AND, OR, NOTと少し違う。これは次節で述べる。

3.4 Backtracking の control と論理変数のスコープ

Hinge 関数 &, !, ~ は short range の backtracking を control するのに用いることができる。一方、通常の Lisp 関数（AND, OR, NOT も含む）は導出モードで実行されても backtracking の対象とはならない。つまり、backtracking によって再びその関数を実行しようとしたときには、仮想的に実行し失敗するようになっている。Hinge 関数も一旦値を返せば、他の Lisp 関数と同様に backtracking の対象とはならない。

& の中に現われる論理変数は、& の中の全 relation に共通である。relation としてはもちろん Lisp 関数も許されるので（表1を見よ）、& の中に現われる Lisp 関数中の論理変数も共通なものとして扱われる。それに対して、! や ~ の中に現われる論理変数は、その論理変数を含む relation の中だけで局所的に使われる。Lisp 関数は実行が終了すると（値を返すと）、その中で行った統一化を破棄してしまう。それに対して、Hinge 関数は値を返しても、その中で行った統一化は保持する。

まず、Hinge 関数による backtracking の control を説明しよう。今、次の relation が定義されているとする。なお、print, list は Lisp 関数である。

```
(defrelation belong ((belong *x (*x . *z)))
                    ((belong *x (*y . *z)) (belong *x *z)))
(defrelation = ((= *x *x)))
```

このとき、次の2つの form を考えよう。

```
(1) (& (belong *x (1 2)) (print *x)
       (belong *y (2 3))
       (print (list *x *y))
       (= *x *y)
       '(Answer ~*x))
```

```
(2) (& (belong *x (1 2)) (print *x)
       (& (belong *y (2 3))
          (print (list *x *y)))
       (= *x *y)
       '(Answer ~*x))
```

各々の form は実行され、次のような出力を行う。最後の出力は top level eval による。

(1): 1 (1 2) (1 3) 2 (2 2) (Answer 2)

(2): 1 (1 2) 2 (2 2) (Answer 2)

上述したように、(2) の form では内側の & については backtracking が行われない。従って、(2 3) を (3 2) で置き換えると (1) の form では成功するが、(2) の form では失敗することに注意。このように & で backtracking の control が行える。

次に、Hinge 関数（&, !, ~）と Lisp 関数（AND, OR, NOT）の違いを例示しよう。

```
(3) (& (belong *x (1 2)) (print *x)
       (AND (belong *y (2 3))
            (print (list *x *y)))
       (= *x *y)
       '(Answer ~*x))
```

```
(4) (& (belong *x (1 2)) (print *x)
       (AND (& (belong *y (2 3))
               (print (list *x *y))))
       (= *x *y)
       '(Answer ~*x))
```

(3) と (4) の form の実行で、次のような出力が行われる。

(3): 1 (1 *y) (Answer 1)

(4): 1 (1 2) (Answer 1)

(3) と (4) の form とも AND が実行し終ると、AND の中で *y に対して行われた統一化が破棄されるので、= という relation によって *y が 1 と統一化されることになる。また、(4) における & 内での統一化の効果にも注意。

TAO Lisp の logic programming では depth-first search と breadth-first search を混在させることが容易にできる。例だけを示す。（注意、all は Hinge 関数ではない。）

```
(& (= *list '(all *x (Human *x)))
   (& (belong *x *list) ... ) (& ... ) ... )
```

3. 5 Logic programming 用の諸関数

主な関数を示す。(以下の関数名，仕様は最終的なものではない.)

(assert [relation]...) relation が一つのときは事実の表明. relation が複数のときは先頭の relation 名を持つ手続きの定義.

(defrelation atom [(relation1 [relation]...)]...) header名が atom である同一 relation 1に対する一括した事実の表明 or 手続きの定義. defrelation は assert を呼んでいる.

(groundp 論理変数) 論理変数が ground instance を持てばその ground instance を返す. さもなくば, NIL を返す.

(all skeleton [relation]...) relation (&が陰に仮定) を満足する統一化を施した skeleton をすべて求め, それを値として返す.

(the skeleton [relation]...) all と同じであるが, the は1つだけ求める.

ただし，より高級な backtracking の control を行う関数については現在検討中である

3. 6 Logic program の例

例1. 与えられた経緯度から最も近い n 個の都市を求める Lisp 関数 ([4]の修正版)

```
(defun nearest (n lat lon)
    (first n (quicksort
              (all (*x *y) (distance (PLACE *x) (POSITION `lat `lon) *y)
              (increasing) 2))))

(defrelation distance
     ((distance *x *y *d) (getposition *x *la1 *lo1) (getposition *y *la2 *lo2)
                          (= *d `(SPHDST *la1 *lo1 *la2 *lo2))))

(defrelation getposition
     ((getposition (POSITION *la *lo) *la *lo))
     ((getposition (PLACE *x) *la *lo) (latitude *x *la) (longitude *x *lo)))
```

例2. リストの要素を奇数番の要素と偶数番の要素に分割する relation.

```
(defrelation partition
     ((partition () () ()))
     ((partition (*x) (*x) ()))
     ((partition (*x *y . *rest) (*x . *odd) (*y . *even))
      (partition (*rest *odd *even))))
```

4. おわりに

TAO Lisp の設計哲学は Taoism に通じている。「TAO道可道，非常TAO道」，「TAO道常無為而無不為」は，いわば自然体の system を設計するべきことを示唆している。成長と進化のカギがそこにある。基礎言語は可能な限り広い器たるべきである。(言語仕様は言語使用上の discipline と混同してはならない。たとえば，goto の使用を許さないのは一向に構わないが，goto の仕様を許さないのはおかしい.) 3章で述べた logic の導入も，様相の導入などさらに拡張の余地があるように思える。いずれにせよ，次の会話が我々の "state of the art" を表現している。

"How to lisp?" "Lisp lisps 'Lithp.'" (解題: How to は著者の頭文字の組: lisp の s を th のように発音すること. Lithp の th は theorem に通ず!)

最後に，山下私一室長，外山芳人，後藤滋樹，J.A. Robinson の4氏に感謝する。

[文献] (1)日比野他: ELIS, 記号処理 12-15, '80. (2) Sandewall: The Lisp Experience, Comp. Surv, 10-1; '7
(3)後藤他: FLATS, 記号処理 1-1, '78. (4) Robinson: LOGLISP, Syracuse 大, '81. (5) Nakashima: PROLOG/
東大, '81. (6)後藤: DURAL, 数解研講究録 396, '80. (7) 竹内他: NUE, 23回情処全大, '81.