記　号　処　理 56－2
計算機アーキテクチャ 84－2
（1990. 9. 28）

新しい TAO の設計

竹内郁雄 天海良治 山崎憲一
NTT 基礎研究所, NTT ソフトウェア研究所

新しい記号処理カーネル SILENT の上に設計を開始した新しい Lisp 言語 TAO について述べる. TAO は, ELIS 上に開発された TAO/ELIS をリファインし発展させたコンカレント・マルチパラダイム言語である. 本報告では, まず, Lisp パラダイム, 論理型パラダイム, オブジェクト指向パラダイムのよりエレガントな融合法を, 主にシンタクスの側面から述べる. 次に, TAO の基本データ型に関するいくつかのトピックス, システム内部制御に使うハッシュ表 (シムタブ), 文字列, 配列などについて述べる. 最後に, TAO で最も重点を置くコンカレンシーの基本部分について, 簡単にスケッチする.

SILENT 上の TAO は TAO/ELIS に比べて, 言語仕様を整理・リファインし, 簡潔なものにすると同時に, 性能を1桁以上上げることを目指す. 現在のところ, 性能向上比の試算値あるいは目標値は, リスト処理で 10, 数値処理で 20～30である.

Design of new TAO

Ikuo Takeuchi　Yoshiji Amagai　Kenichi Yamazaki
NTT Basic Research Laboratories, NTT Software Laboratories

This paper introduces a new Lisp language TAO which is now being designed on a new symbolic processing kernel machine SILENT. TAO is a concurrent multi-paradigm Lisp which comes out as a refined successor of the TAO, or TAO/ELIS designed and implemented on the ELIS machine.

This paper first describes an elegant way of fusion of Lisp programming, logic programming and object-oriented programming paradigms, mainly from the syntactical viewpoint. Then it describes some topics on fundamental data types of TAO, such as hash tables which will be extensively used in the system internal controls, strings of characters and numbers, and fast accessible arrays, the design of which is deeply related with the design of the SILENT machine. Finally, it briefly sketches a few topics on the concurrency which is one of the most emphasized features of TAO.

The TAO on the SILENT machine would have a simpler and more elegant language specification compared with the TAO/ELIS, and moreover it would achieve more than ten times as good running performance as that. At present, the estimation of the performance improvement will be a factor of 10 in the list processing and a factor of 20 to 30 in the numerical processing.

1. はじめに

Lisp は Common Lisp[1] によって標準化されるというのが趨勢である．しかし，標準化は相反する2つの作用をもつ．1つは，言語の普及が促進されるという正の作用であり，もう1つは，言語のダイナミックな成長が阻害されるという負の作用である．言語屋としての筆者らは後者に対して多少の憂いをもつ．もっとも，ほかの言語と異なり，Lisp の場合は，Common Lisp にオブジェクト指向をどう取り入れるといった議論などがかまびすしく，Lisp の潜在的な生命力と成長力が依然として示されている．Scheme や EuLisp がピンピンしているのもそのあたりの事情を物語っている．

さて，政治的な理由はともかく，Lisp というブロードな土台の上でプログラミングの本質をあれこれ考えることが効率的で，思考の経済につながっていることは事実である．また Lisp が単なる理論的言語にとどまらず，現実的実用性を生ずることも忘れてはならない．

我々はすでに TAO という名の Lisp を2種類作ってきた (TAO/60[2], TAO/ELIS[3])．ここで第3作目の TAO† を設計しようというのは，いつの時代にも新しい言語を設計することが計算機科学や工学の発展にとって不可欠だと考えるからである．

TAO の設計を新しく始めた根源的な理由は上記の通りであるが，より直接的で具体的な動機は次の通りである．

(1) SILENT という新しいアーキテクチャの高速記号処理マシンの開発によって新しい Lisp の設計がサポートされたこと．(SILENT の開発の動機は [4] を参照のこと.)

(2) TAO/ELIS で行なった Lisp パラダイム (あえてパラダイムと呼ぶ)，オブジェクト指向パラダイム，論理型パラダイムの融合をもっとエレガント，かつ効率よく再構成したかったこと．

(3) TAO/ELIS で有効性を確認した Lisp 上でのコンカレント・プログラミング [5] をリファインし，ヘテロ結合マルチプロセッサ系の中で Lisp マシンがあたかも脳全体の中での大脳皮質のような役割を果す記号処理カーネルとして TAO/SILENT を発展させたかったこと．

このような動機あるいは目的のため，TAO の設計方針は次のようなものになった．

(a) 基本的にストック (汎用アーキテクチャ) マシンを対象とした Common Lisp にはとらわれない．SILENT という専用マシンの特性を生かした独自の言語とする．むしろ記号処理専用マシンの可能性をとことんまで追求する．ただし，Common Lisp からのプログラム変換の容易性は考慮する．

(b) コンカレンシーの機能と性能の開拓に最も重点をおく．具体的には1つの SILENT の上で 1,000 個の小プロセス‡ がそれなりの速度で走ることを目指す．

(c) 上述したようにパラダイム融合のエレガンスと効率を向上させる．また，雪ダルマ式に大きくなった TAO/ELIS の言語仕様を整理し，単純でエレガントな言語にする．しかし，実用性は犠牲にしない．また，TAO/ELIS で成功したと考えられる特徴は受け継ぐ

(d) TAO/ELIS では設計上インタプリタを優先したが，TAO ではコンパイラを優先する．ただし，インタプリタの速度も重視する．AI の記号処理では S 式の解釈実行が本質的と考えられることが多いからである．

本報告は，SILENT チップの設計 [4] と並行して進められた TAO の設計から，いくつかのトピックスを選んで述べた，いわば中間報告である．いくつかの点で今後設計変更があるかもしれないが，性能に関係するボトムアップ的な部分は，すでにマイクロコードの試作・評価が完了しており，大きな変更はないであろう．

2. 式

2.1 式の種類

TAO/ELIS では，Lisp 関数式と論理述語式は外見上の区別がなく，

(fn arg1 arg2 …)

と書かれていた．Lisp 関数も論理述語も applobj という大きな関数概念のもとで統一されていたからである．これはこれでエレガントな仕様であるが，実装上いくつか困難な問題をもたらした．たとえば，fn が Lisp 関数か論理述語か判定できず，やむなくほとんどインタプリタに近いコンパイルコードを出す場合があった．

また，メッセージ式は

[receiver selector arg …]

と書くが，便宜上，角カッコの代わりに丸カッコを書いてもよかった．これはトップレベルの対話では便利だったが，ソースプログラムに丸カッコのメッセージ式が書かれると，コンパイラが悩んでしまうことになる．

TAO ではこれらの反省から，Lisp 関数の呼び出し，論理述語の呼び出し，メッセージの伝達を完全に分離した文法 (S 式) で書くようにする．

Lisp 関数 (およびマクロ) の呼び出しは従来通り，丸カッコを使い

(fn arg …)

† 本来は new TAO あるいは νTAO と呼ぶのが正しいのであろうが，ここでは単に TAO と呼び，ELIS 上の TAO を TAO/ELIS と呼ぶ．

‡ TAO ではプロセスとスレッドの概念を分けるので，これはスレッドというほうが正確．

と書く．これに対し，論理述語の呼び出しは波カッコを使い，

{pred arg … }

と書く．(ここで，一般に引数が Lisp の意味では評価されないことに注意．また波カッコや角カッコは内部的なタグだけが異なるコンスセルを表わしている．) こうすることによって，Lisp の関数式と論理述語式がどんなに入り交じっても混乱しなくなる．さらに，1つのシンボルに対して，Lisp 関数の空間と論理述語の空間を分離する．だから，たとえば append のように基本的な名前でありながら，両方の意味をもっているものも，名前変更の必要がなくなった†．たとえば，

(append '(1 2 3) (cdr x))
{append (1 2 3) _x _y}

は同じシンボルを使いながら異なった意味になる (_x, _y は論理変数と解釈される — 2.3 節参照)．

メッセージ式は TAO/ELIS と同様，角カッコを使うが，より一般的に次のような形式にする．

[obj (msg arg …)]
[obj {msg arg … }]

前者は Lisp とオブジェクト指向の融合，後者は論理とオブジェクト指向の融合である．前者を Lisp メッセージ式，後者を論理メッセージ式と呼ぼう．レシーバである obj は評価されるが，msg は評価されない．引数は論理メッセージの場合，評価されず，ユニファイの対象となる．これらのメッセージ式は次のように解釈される．

レシーバである obj が msg という Lisp 関数 (または論理述語) を知っていたら，それを自分の環境 (つまり自分のインスタンス変数が見える状態) で実行する．これはふつうのメソッド呼び出しにほかならない．知らなければ，obj の指定された委譲先，あるいは現在の計算環境へこのメッセージを委譲 (delegate) する．この場合，自分のインスタンス変数は通常見えなくなる．委譲先のない Lisp メッセージ式は obj を無視した，ふつうの関数呼び出しになってしまうわけである．これは一見無意味な委譲のようであるが，論理メッセージ式では実際的な意味がある．指定した委譲先ではなく，現在の計算環境へ委譲することを，ユニバース前提と呼ぶ．簡単な例を示そう．

[yin (+ 3 4)] (1)
[yang {friend Tom _x}] (2)

もし，yin が 5 であれば，(1) の値は 12 である．しかし，yin が + を知らないオブジェクトであれば，7 が返る．また，yang の世界で Tom と Jerry の仲がよければ (2) は t を返し，x に Jerry がユニファイされる．しかし，yang の世界でそれが assert されていなければ，ユニバース前提が呼ばれて，たとえば，nil が返る．

このメッセージ式は obj という環境で Lisp 関数式や論理述語式を評価するという意味に解釈されるが，これはメッセージ式の新しい解釈を示すシンタクスといえよう (図 1)．

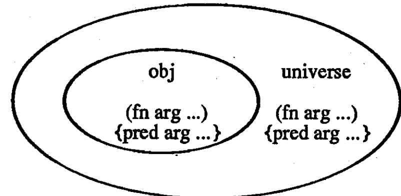


図 1 メッセージ式の解釈とユニバース前提

上記の一般的なメッセージ式だけではなく，Lisp メッセージについては

[obj msg arg …]

という省略記法が許され，TAO/ELIS で愛用された

[x + y]

といった中置記法が TAO でも使える．

TAO/ELIS と異なり，TAO では，この 3 種類のカッコが，選ばれたパラダイムを表わすという方針を徹底する (マクロは除く)．たとえば，TAO/ELIS で許されていた丸カッコのメッセージ式は廃止し，また

(apply 123 (list '+ 456))

という形でメッセージ伝達が起こることも廃止する．この方針を徹底するために，各パラダイムにおける apply の表現や意味を以下のように統一する．ここで，Lisp 式の中のコンマや論理述語式の中のアンダースコアは通常は評価しないものを強制的に評価させる eval 演算子だと考えてよい (2.3 節参照)．

(,fn arg1 arg2 … . arglist) (3)
(,fn arg1 arg2 …) (4)
{_pred arg1 arg2 … . arglist} (5)
{_pred arg1 arg2 … } (6)
[obj ,expr] (7)
[obj (,fn arg1 arg2 …)] (8)
[obj {_pred arg1 arg2 … . arglist}] (9)
[obj ,fn arg1 arg2 …] (10)

(3) と (4) はそれぞれ

(apply fn arg1 arg2 … arglist) (3')
(funcall fn arg1 arg2 …) (4')

† TAO/ELIS では論理の append が lappend という名前になっていた．

と等価である (だから, TAO ではむしろ apply, funcall が派生的な関数となる). ここで, (3) にあるドット記法に注意されたい. リスト記法に変換できるドット記法は一般には無意味であるが, TAO では

(goo . (choki pah))
(goo choki pah)

という 2 つの式は, TAO/ELIS で区別されていた nil と () と同様, 内部表現が (微妙に) 区別される. コンパイラ, インタプリタ, I/O, いくつかのコピー関数のみ, この区別を検知する. だから,

(,fn x . (cdr y))

という式が

(,fn x cdr y)

と混同されることはない. なお, 関数にコンマがついていない

(+ 1 2 3 . (list 4 5 6))

も apply の意味になる. (5) と (6) も同様で, それぞれ

{apply pred arg1 arg2 ⋯ arglist} (5')
{predcall pred arg1 arg2 ⋯ } (6')

と等価である. (5') の apply と (3') の apply は同じシンボルであり, Lisp 関数と論理述語の両方の意味をもつ (丸カッコ式は Lisp 関数定義だけを見, 波カッコ式は論理述語定義だけを見る). 論理述語の apply は引数を評価しないから必要な場合は, 引数の前にアンダースコアをつけなければならない. また, 論理述語の apply は論理におけるバックトラックスコープを区切らない†.

メッセージ式 (7), (8), (9), (10) なども同様で, メッセージ全体, メッセージセレクタなどに, ある式を評価した結果を使うことができる.

以上のように, シンタクスレベルで見たときのパラダイム融合は TAO/ELIS よりもエレガントになったといえよう.

2.2 Lisp 式

本節では Lisp 関数式に固有で, TAO 独特のもの 2 点について述べる. なお, TAO の lambda は Common Lisp とは異なり, いわゆる downward funarg のみである. それ以外, upward と sideward funarg (プロセスを渡る関数閉包) は陽に closure を作らなければならない.

第 1 点はキーワード引数である. TAO のキーワード引数は少々重すぎる Common Lisp のキーワード引数とオプション変数の中間を狙う. 次の list-match の例が TAO のキーワード引数を最もよく例示している.

(list-match x :start s1 :end e1 y :start s2 :end e2)

これは 2 つのリスト (の部分列) の相等を調べる関数としよう. 部分列を指定したいときは, :start や :end で範囲を指定するわけである. さて, TAO のキーワードは順序が固定されている. そして, 義務変数の間にでも書ける. そのかわり, 誤解がないのであれば, 同じキーワードを上記の例のように二度以上使える. 意味は簡明直截でインタプリタにとってもコンパイラにとっても効率がよい. すなわち, lambda リストを左から順に見ながら実引数とマッチをとっていき, lambda 引数中にキーワードが現われたら, 実引数がちょうどマッチしたときに限り, キーワード引数が陽に指定されたと考えるのである (それ以外はデフォールト値).

第 2 点は, 無駄な中間値を作らない蜻蛉引数というメカニズムである. これは S 式で定義された関数には使えないが, マイクロコードで書かれた基本関数では多用される. たとえば, ストリング (の部分列) の相等を調べる s= という関数は, キーワード引数をもつ Common Lisp の string= と異なり, 単純な 2 引数の関数である. 部分列を引数にしたいのであれば, 陽に substring という関数を使って,

(s= (substring x 3 10) y)

と書く. 通常の Lisp では, ここで (substring x 3 10) で示される部分列を陽に作ってしまう (TAO ではヘッダーだけである). これは比較が終わればただちにゴミとなるデータである. しかし, s= の 2 つの引数は蜻蛉引数となっており, 実引数にすぐゴミとなるような substring の式があった場合は, substring に本格的データではなく, s= の終了後ただちに蜻蛉のように消滅する特殊なデータを作らせてしまう (関数の入れ子に連動する push-down stack の中にヘッダーを作らせる). このほか, 64 ビット倍長整数, 倍長浮動小数点数の出てくる四則演算の引数は, 無駄な中間値を作らないようにするため, すべて蜻蛉引数になっている.

TAO は実時間ゴミ集めを最初から実装するので, これはあまり必要のないものかもしれないが, 少しでも実時間性を高めようという設計思想の発現である.

2.3 ユニフィケーション

TAO/ELIS では, インタプリタにおけるユニフィケーションの高速化をはかるため, Lisp 変数 (通常のシンボル) と論理変数が, たとえば, p と _p のように異なる型のシンボルとして区別されていた. これに対して, TAO で

† 論理式の中で Lisp 式が評価され, その Lisp 式の中でさらに入れ子になった論理式が評価された場合, もとの論理式とはバックトラックのスコープが分離される.

は論理変数という型を設けない．ただし，アンダースコアの見かけの意味は変えないようにした．

すなわち，変数宣言のときに，アンダースコアが前置されたシンボルは，初期値がundefという値になる．そして，ユニフィケーションのときにアンダースコアのついた式(項)は評価されてからユニファイされる．このとき，値がundefであれば，ユニファイの結果，そこにundef以外の値が代入される．

このようにアンダースコアの意味は内部的に改変されたが，プログラムの見かけはほとんど同じである．たとえば，appendの定義は

```
(assert {append () _x _x})
(assert {append (_a . _x) y (_a . _z)}
        {append _x _y _z} )
```

とほぼ従来通りである．また，

```
(let (x _y) … {foo _x _y} …)
```

とすると，従来の意味でのLisp変数をユニフィケーションの前に評価するもの†が，見かけ上は従来の論理変数のようになる．アンダースコアのついてない項はすべてパターンと解釈される．さらに，新しいアンダースコアの意味では，

```
{foo _p _(aref a i)}
```

のように，(もとの値がundefであった)「場所」への代入をユニフィケーションで書けるようになる．なお，Lisp式の中でアンダースコアを使うとエラーである．

TAO/ELISでは，ホーン節のリテラルを事前に一度評価するのにもコンマを使っていたが，TAOでは論理述語としてのevalを使う．たとえば

```
(assert {if _p _q _r} {eval _p} ! {eval _q})
(assert {if _p _q _r} {eval _r})
```

このようにTAOは，Lisp変数と論理変数の区別をなくし，言語仕様がより簡素なものになった．

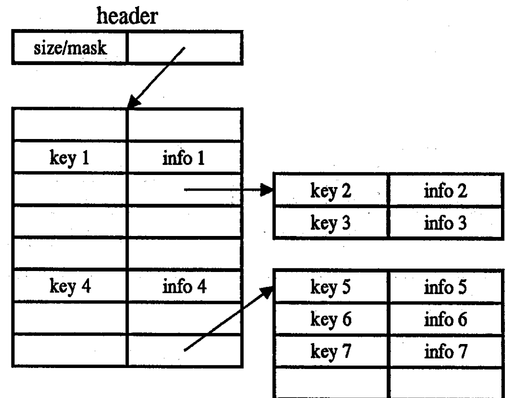
3. データ型

Lispの設計ではつねにデータ型の設計が最初の重要な課題となる．TAOの設計はSILENTの設計と同時並行的に行なわれたため，筆者らはSILENTの特徴を生かしたTAOのデータ構造，また逆にTAOに必要なデータ構造に望まれるSILENTアーキテクチャが得られたと信じている．言語設計という観点からするとややボトムアップ的にすぎるかもしれないが，ここでTAOのいくつかの特徴的なデータ型について述べよう．

3.1 シムタブ (symtab 仮称)

FLATS[6]は，ハッシュに基づいてシステム全体を構成しようとした野心的なマシンである．TAO/SILENTもシステム制御の基本部分にハッシュを取り込む．しかし，FLATSのような大々的なハッシュ・ハードウェアを仮定せず，SILENTのALUにビットをシャフルするmashという簡易ハッシュ命令(30ns!)を設けただけである．

シムタブ(symtab)はその名から推察できるように，シンボルからそれに付随した情報を素早く引き出すためのデータ構造である．基本的にはチェイン法を使ったハッシュ表であるが，チェインはコンスセルによるリストではなく，後述するバディを使った線形探索表である(といっても通常は大きさが最大4まででであり，この範囲だと，水平型マイクロと，キャッシュメモリの効果で性能に悪影響はない)．図2に典型的なシムタブを示す．シムタブはエントリ数に応じて極めて柔軟に再ハッシュされる．シミュレーションの結果では，エントリ数Nに対して総メモリ使用量は$2N$セル程度に押えることができ，エントリ数3から10,000程度までは，(キャシュがすべてヒットしているとして)平均340nsでシンボルに対応した情報が得られる．つまり，シムタブは通常の連想リストあるいは属性リストと同程度のメモリを使うが，約340nsで必要な情報がアクセスできるわけである．



(ヘッダ以外はすべて2のベキ乗バディブロック)
図2 シムタブの形

前述のようにTAOはシムタブを言語の骨格に据える．たとえば，属性リスト，連想リストは(ユーザが自分で作るのならいざしらず)シムタブに置き換えられる．スレッドごとに必要な非局所スレッド変数(Common Lispのスペシャル変数に近い)の検索表もシムタブである．これは，走行時ほとんどキャッシュに乗っていると考えられるので，非局所変数のアクセスには平均300～400nsかかるということを意味する‡．なお，後述するように，

† TAO/ELISではコンマが使われていた．

TAOではシンボルを共有するマルチプロセスが大前提なので，非局所変数の値をいつもシンボルに直接ぶら下げるわけにはいかない．

シムタブはメッセージ伝達におけるメソッド探索にも使われる．現在の見積もりでは，メソッドの探索時間はメソッドの数によらず平均400ns程度である． TAO/ELISでは二分探索を行なっていたが，これに比べて典型的な例で一桁の速度向上となる．

シムタブにより，シンボルと情報のリンクを間接的にしても高い性能が得られるので，言語設計に関して高い自由度が得られることは特筆しておくべきであろう．

シムタブのみならず，TAOでは2のベキ乗のメモリブロックを多用する．これはよく知られたバディシステムで管理する．バディはビグナムの表現にも使われる．最悪の場合でもコンスセルと同等のメモリ効率であり，なおかつキャッシュの速度効果が期待されるからである．

3.2 オブジェクト

TAO/ELISではユーザ定義のオブジェクト（数や文字列といったシステム定義オブジェクトと区別し，user defined object，あるいは単に udo と呼ぶ）を図3-aのように，インスタンス変数名と値を対にしたベクタで表現していた．これはシステムのデバグには非常に便利であったが，必要なメモリが2倍になる．TAOではこれを常識通り，値だけからなるベクタで表現する（図3-b）．

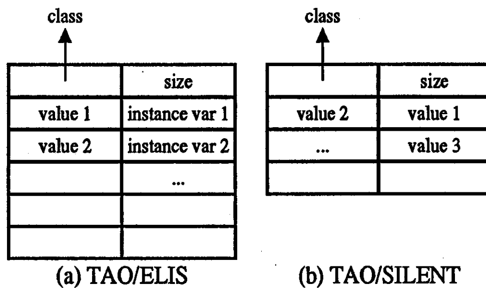


図3　udoの構造

オブジェクトからは親のクラスを指すポインタが出ているが，メソッド探索を親ベクタ経由にするか，オブジェクトから直接にするかは，まだ検討中である．

3.3 文字列

TAO/ELISにおいては文字列処理に重点がおかれ，その文字列処理は本来のリスト処理よりも速いとまでいわれた． 1バイトのASCIIコードと2バイトのJISコードをパックしてしまう「無無駄表現」は，その世界で完備性をもった文字列処理関数の体系をマイクロコードで書きまくった（まさにこの表現通り！）からこそ可能だったものである．しかし，C言語などで書き溜められた文字列処理のアルゴリズムを流用したいときなどはまったくお手上げであった（たとえば，n番目の文字へのアクセスが定数時間にならない）．

TAOの文字列は，内部的には1バイト系と，2バイト系の2種類に分かれる．しかし，ユーザがそれを意識する必要はない． 1バイト系の文字列に2バイト系のコードが入ろうとしたところで自動的に2バイト系の文字列に変換（coerce）される（逆の自動変換はなし）．これによって任意位置の文字へのアクセスが定数時間になる．

文字列（charstring）のほか，これとほとんど同じ構造と，同じ処理関数をもつ数値列（numstring）という概念も導入された． これら全体をTAOではストリングと呼ぶ．数値列を文字列のように伸縮自在に扱う応用は意外と多そうである． たとえば，TAO/ELISで文字列の文字個々につけられていた付加情報（フォントと呼ぶ）はTAOではこの数値列で表現される．

上述のような1バイト系から2バイト系への自動変換が起こったとき，元の文字列とその部分列を指しているポインタがすべて同時に新しい2バイト系の本体を指すようにしなければならない． これを可能にするのが部分列（substring）というデータ型である．以下，親ストリングを文字列として説明するが，数値列に対しても同様である．

部分（文字）列は親文字列の部分列を表現する一種の記述子である．TAOでは陽にコピーを指定しないかぎり部分文字列を取る操作では親文字列の本体をコピーしない．

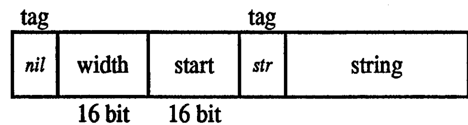


図4　部分列

図4のように，親文字列のどの始点から何文字の幅という情報をもった部分列記述子が作られるだけである． 始点は文字と文字の間の位置を表わす整数である． 左からだと0から始まる正の数で，右からだと −0 から始まる負の数である． ここで注目すべきは，−0 という数があることである（実装では16ビットの負の最小数 #x8000 がこれに充てられる）．幅にも0と −0 があり，それぞれ右0文字，左0文字ということを表わす．このような正と負の0が有用であることは次の関数を見れば了解されるであろう．

```
(smodify str new-str)
```

‡（前ページ注）局所変数へのアクセスはすべて60nsである．

これは，str で示された文字列を new-str で示された文字列に破壊的に置き換える関数である．もし，str が部分列であれば，親文字列のその部分を new-str で置き換える．このとき str (部分列記述子) の内容は，新しく埋め込まれた new-str を示すように書き換わるが，部分列の方向 (右向き，左向き) は変えない．str として幅 ±0 の部分列を与えると smodify は挿入関数として働き，例外的に str は始点だけが移動する．つまり，幅 ±0 の部分列はここでは向きのついたカーソルのように使えるのである．

親文字列が破壊的に書き換えられたとき，文字列が伸びて新しい場所に移動したり，1 バイト系 2 バイト系変換が起こったりすることがあるが，部分列はすべて親文字列のヘッダー経由であるから問題は起こらない．部分列が親文字列の範囲をはみ出している場合は，はみ出さない範囲の自然な解釈がなされる．なお，この方法では親文字列を直接指すポインタは 1 個しかない．だから，破壊操作で不要な文字列本体が出たときはただちにゴミとして回収できる．また，シンボルの印字名やプログラム中の文字列などは書き換え不能というマークがつけられるので事故による破壊に対して安全である．

このように TAO の文字列あるいは数値列は，比較的少数の概念でいろいろなことが説明あるいは記述できるようになっている．多少，実験的ではあるが，高い実用性が期待できよう．

3.4 配列

TAO は SILENT が TARAI プロセッサと結合することもあって，数値処理は TAO/ELIS 以上に重点をおく．特に，配列については，ほとんど C 言語なみの最適化が可能なようにする．実際，[4] でも述べられているように，ほかのプロセッサと配列を共有してデータのやりとりを行なうことが簡単なので，これは必須であろう．

配列は 1, 2, 3 次元まではタグを分けた独立のデータ型 (vector, matrix, cube) とし，チェックを行なう場合でもそれが高速になるようにした．SILENT に，ALU と並行動作可能な乗算器が備わることもあって，配列のアクセスは TAO/ELIS の 10 ～ 30 倍になる見込みである．

さらにどのメモリもアドレス (C のポインタに相当) で参照できる機能を TAO に設ける．まったくチェックなしだから非常に高速である．これは Lisp マシンとしては無謀といえる発想であるが，アドレスを使ってデータを書き込むときは，そのワードのタグ部分が (たとえばバイト配列用の) 特殊な値になっていないと書き込めないようにするので，一応フェールセーフではある．

4. コンカレンシー

TAO/ELIS におけるコンカレンシー (並行プログラミング) はプリミティブな「機構」の集合であり，その上で「言語」としてのコンカレンシーをいろいろ実験しようという設計思想であった．プリミティブな機構の性能が悪くなかったので，実験的とはいえ Lisp マシンとしては十分な実用性が発揮された．

TAO ではその経験を生かし，プリミティブをさらにリファインし，性能を上げる方向で設計を進める．すなわち，高レベル・コンカレンシーを先に設定するのではなく，低レベル・コンカレンシーを先に設定する．高レベル・コンカレンシーはそれさえあればいかようにでも設計できるからである．

4.1 プロセスとスレッド

TAO では，プロセスとスレッドの概念を分離する．スレッドは実行環境スタックと一対一に対応した概念である．プロセスは (複数の) スレッドの共通の根となる環境である†．TAO/ELIS では両者が一体化していたが，Unix のプロセスのように重くなく，それ自身がかなり軽いいわゆる light weight process であった．TAO ではそれをさらに軽くしようというわけである (スレッドの生成時間は数 10μs 以下の予定)．なお，プロセスもスレッドも udo (3.2 節) である．

1 つのスレッドはさらにそこから複数の子スレッドを生むことができる (図 5)．

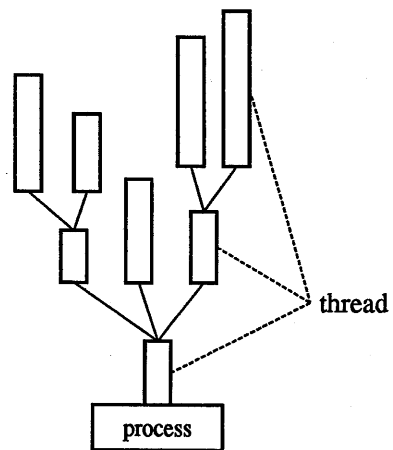


図 5 プロセスとスレッド

つまり，木構造のスタックを作ることができる．ただし，木構造を保つため，親スレッドにコントロールが戻るとき，すべての子スレッドは消滅しなければならない．だから，子スレッドの生成時に，1 つの子スレッドから戻りがあったときに親へ戻る OR 分岐であるか，すべての子

† TAO のプロセスが Mach でいうスレッドに相当すると思ってよい．

スレッドからコントロールが戻ったときに初めて親へ戻るAND分岐であるかといった指定を行なうようにする．

SILENTはスタック・オーバーフロー割込みのためのスタックリミットの設定を1語単位でソフト的に行なえるので，スタック領域の管理をELISに比べてよりきめ細かく行なえる．その結果，スレッドを最大1,024個まで同時に走らせることが可能になる．なお，1つのプロセス上のスレッドの最大深さ（木構造になったスタックの底から先端までの最大深さ）は128K語である．スタック領域の管理についてここでは詳しく述べないが，スレッドとスレッド（別のプロセスのスレッドであってもよい）の間のスイッチは数μs程度になる予定である．

4.2 変数

コンカレンシーのところで変数を論ずるのは奇異に見えるかもしれないが，基本的にすべてのデータ空間（および，Lispなので，プログラム空間）を共有することを前提にしたコンカレンシーを実現する場合，プロセスやスレッドが同一のシンボルを変数として共有するための方法論を確立しておかないと，実際的なプログラムを作るときに大きな困難が生ずる[5].

TAO/ELISでの変数分類を少し整理して，TAOでは変数を局所的なものから，大域的なものまで，次のように分類する．

局所変数
インスタンス変数
非局所変数
　スレッド変数 / プロセス変数 /
　ログイン変数 / パッケージ変数

スレッド変数は，関数やletなどの変数宣言のところで動的に作られる非局所変数であり，作ったプログラム構造が終了すると消滅する，いわばスタック上の変数である．プロセス変数はそのプロセスから生えたすべてのスレッドに共通の変数であり，プロセスudoに付随したプロセス固有の作業記憶である．ログイン変数は同一ログインの下にあるプロセス† に共通の変数であり，ログインを表わすudoに付随する．パッケージ変数は，シンボルに値が直接結びついた変数である．

非局所変数はすべて値がバインディングと呼ばれるセルに格納される（図6).

value	lock

図6 バインディング

このようにすると，バインディングそのものが共有可能になる．こうした場合，変数への書き込みにロックが必要になることがある．このためにセマフォアを使うのはやや重いので，バインディング自身にロック機能を入れる．図6のcdr部分に，スレッドが入るとそのスレッド以外はcar部分に値を書き込めない．書き込もうとしたスレッドはビジーウェイトに入る．

† データ空間を共有したマルチユーザというのは，名前空間の問題を明確に浮き彫りにするので，TAO/SILENTも一応マルチユーザ対応にしておく．

5. おわりに

本報告は，TAOの設計構想，あるいは中間報告といった性格のものであり，系統だった言語仕様体系について述べたものではない．新しい記号処理プロセッサの設計と同時進行している専用言語の設計の相互作用について詳しく述べることはできなかったが，一番重要で難しいところのマイクロコード，すなわちハードウェアとソフトウェアの接点が設計の原点であったことは，少しだけ窺えたのではないかと思う．

SILENT上のTAOは，記号処理でTAO/ELISの約10倍，数値処理で20～30倍の性能を狙っている．実際に達成された性能については機会を改めて報告する．

謝辞

SILENTのアーキテクチャを設計し，そのチップを開発している吉田雅治氏，およびこのプロジェクトを支援していただいている関係各位に感謝する．

参考文献

[1] Steel, G.L.: COMMON LISP THE LANGUAGE 2nd ed, Digital Press, 1990.
[2] 竹内，大里:「会話型Lisp TAO」電気通信研究所所内資料，1982
[3] Takeuchi, I., Okuno, H.G., Ohsato, N.: A List Processing Language TAO with Multiple Programming Paradigms, New Generation Computing, vol.4, No.4, Ohmsha and Springer, 1986.
[4] 吉田，竹内，山崎，天海:「新しい記号処理カーネルSILENTの設計」，記号処理研究会，56-2, 1990.
[5] Takeuchi, I.: Concurrent programming in TAO — Practice and Experience, Parallel Lisp, Language and Systems, LNCS No.441, Springer, 1990.
[6] Goto, E. et al.: Design of a Lisp Machine — FLATS, Proceedings of Lisp and Functional Programming, pp. 208 – 215, 1982.