

TAO における余剰引数, 多値, 倍長数の 計算などの見直し

竹内郁雄 天海良治 山崎憲一 吉田雅治

NTT 基礎研究所, NTT ヒューマン・インタフェース研究所

記号処理カーネル SILENT 上の言語 TAO の Lisp 部分の引数メカニズムについて述べる.

Lisp の余剰引数の取り扱いは, Lisp 方言ごとに少しずつ異なってきた. Common Lisp の rest 変数が現在のところ標準になりつつあるが, それとて問題を抱えている. TAO では余剰引数をリストとして受け取らないという選択をした結果, 意味的に明解で, かつメモリのムダな消費が生じないという余剰変数の仕様を得ることができた.

TAO は Common Lisp の多値を拡張し, 引数受け渡しの中で, 多値を引数の並びに自由にスプライズできるようにした. スプライザと呼ばれるこのメカニズムにより, フルワードデータの計算においてムダなメモリの消費をしないようにすることが簡単にできるようになった.

TAO の余剰引数メカニズムとスプライザは汎用機上の Lisp でも容易に実現可能であり, 汎用性の高い概念である.

Rest variables and splicers in TAO

Ikuro Takeuchi Yoshiji Amagai Kenichi Yamazaki Masaharu Yoshida

NTT Basic Research Laboratories, NTT Human Interface Laboratories

This paper describes new parameter passing mechanisms adopted in the Lisp part of the TAO language on the symbolic process kernel SILENT.

First, it proposes yet another way to handle with rest arguments. Rest arguments in a function call have been handled with in various ways by various Lisp dialects. Though the rest variable of Common Lisp is becoming widely accepted as a standard, it is still problematic in the viewpoint of semantic clarity and memory efficiency. We come up with a simple and efficient rest argument mechanism in TAO by giving up having the rest variable receive the cons'ed list of rest arguments.

Second, it proposes the concept of *splicer* which enables multiple values to be received and spliced in any position of argument list of a Lisp function. Splicer is only a slight extension of the multiple value concept of Common Lisp. However, it provides an easy and elegant means to save memory in so-called full-word data computation such as double-float, rational, and string.

Both of our rest argument mechanism and splicer can be transferrable to any Lisp dialect, since they can also be easily implemented on stock hardware.

まえがき

我々は記号処理カーネル SILENT の上に、実時間マルチパラダイム言語 TAO (Silent TAO) の設計と実現を進めている^{†1} [1-6]。Silent TAO は DAO と同様、Lisp というベースに論理型プログラミングとオブジェクト指向プログラミングを一つの言語に融合したほか、実時間処理を実現するためのコンカレントプログラミングの強力なプリミティブをもっている。しかし、一方で我々は TAO を SILENT の機械語と規定し、言語を可能なかぎりシンプルなものにすることをねらった。TAO の言語仕様は比較的小さいものになっている。

本報告では、TAO を機械語としてリファインするために行なった言語仕様の見直しのうち、Lisp の話題として一般性があると考えられるものを二つ紹介する。いずれも、計算の途中でムダなコンセルを消費したくないという要求の所産である。言語の実時間性を高めるために実時間 GC を実装することは当然であるが、そうした上でもムダなセルの消費を抑えることが重要である。1 章では、不定個の引数の扱い、2 章では多値を拡張したスプライサの概念と応用について述べる。記述は TAO の記法や実装法に基づくが、Lisp としての一般性については必要なつど言及する。

1. 不定個の引数

この章では、Lisp における不定個の引数の取り扱いの歴史を振り返ったあと、TAO がどのような仕様をとったかを述べる。

1.1 不定個の引数に関する議論

Lisp の関数呼び出しにおいて不定個の引数を受け渡す方法がいくつか考えられてきた。古くは Lisp1.5^[7] の FEXPR、次いで MacLisp^[8] の LEXPR、Interlisp^[9] の nlambdas、EuLisp^[10] の multiple-argument-object^{†2}、最近では ZetaLisp^[11] に始まり Common Lisp^[12] に引き継がれた rest 変数などがよく知られている。現代の Lisp では、記法上の小さな違いを除き、必須変数 (とそのあとのオプション変数) の並びの最後に置いた余剰変数で、不定

個の引数 (以下では余剰引数と呼ぶ) をリストにして受け取るという考え方が主流である。

これを使うと、不定個の引数をリストにして返す関数 list は次のように簡単に書ける。

```
(defun list (&rest x) x) ; Common Lisp
(define list (lambda (x) x)) ; Scheme[13]
```

しかし、余剰引数をリストにしなければならないことはむしろまれである。たとえば、不定個の引数をとる代表的な標準関数に足し算があるが、これが

```
(defun + (x &rest y) ...)
```

と文字どおり定義され、文字どおり解釈されていると、

```
(+ a 123 b c)
```

というような計算をするたびに 3 個のセルがムダに消費されることになる。実際の Lisp の処理系は暗黙の最適化によってこういうムダを防いでいる。しかし、ユーザがこれと同様な関数を定義すると最早そのような最適化の網から外れてしまう。

たとえば、「メモコンプログラマ (memory conscious programmer)」であれば、不定個の文字の並びを出力する関数を作るのに、典型的に使われそうな数文字まではオプション変数で受け取ってセルを消費しないようにし、それを越えたらやむなく余剰変数で受け取るというプログラムを書くであろう。これは現実的ではあるが、あまりスマートでない。

余剰変数で受け取ったものを (ユーザには内緒に) リストにせず、たとえばスタックに積んでしまうという実装も可能である。しかし、これは関数が apply などと呼ばれ、最初から引数がリストで与えられた場合との整合をとる必要性があり、複雑な実装になってしまう^{†3}。

これは些細な問題のように見えるが、実は Lisp 屋のあいだで長い議論を呼んできたものであった。この事情は Common Lisp マニュアル第 2 版^[12] の 77 ~ 78p と 232 ~ 236p に詳しく述べられている (訳書では 66 ~ 77p と 201 ~ 205p)。問題は次の二つに整理できよう。

(1) apply で余剰引数のリストを受け取ったとき、余剰変数は新しいリストを作るべきか? すなわち、余剰変数が押えているリストを破壊してもよいか? もし新しいリストを必ず作るなら、余剰引数の使われ方の実態では、ほとんどがムダな cons になる。

(2) 動的存続 (dynamic extent) のセルというものを認め

^{†1} 以前我々が開発した Lisp マシン ELIS 上の TAO と区別するために、ELIS 上の TAO を DAO と呼んだり、新しい TAO を Silent TAO と呼んだりする。Silent TAO は SILENT マシン上の言語であるという意味のほかに、DAO に比べて「静かになった」つまり「静的になった」という意味合いも含んでいる。

^{†2} ただしドキュメントの記述は意味不明である。

^{†3} Lisp Machine Lisp、すなわち ZetaLisp の実装では cdr-coding した cons をスタック上で行っていた。

るべきかどうか? これを認めると、関数の中で局所的にしか使わないセルを動的存続と宣言すれば、関数からの脱出時に自動的にゴミとして回収できる。すなわち、スタック上にセルを cons することが可能である。ただし、間違った宣言をすると致命的なバグとなる。もし、余剰変数の押えているリストが暗黙的に動的存続宣言されたセルからなるとすると、

```
(defun list (&rest x) x)
```

は致命的なエラーを引き起す。

Common Lisp の仕様を決める X3J13 委員会は、(1) についてはリストを新しく作らないことも許すと決め、(2) については、ユーザの注意深さを要求した上で動的存続セルを宣言することを認めた (dynamic-extent 宣言)。しかし、これらの決定が言語の意味論として明解さを欠くことは明らかである。

TAO は当初、&rest に加えて、&more という予約語で、リストを作らない余剰引数の指定を行なうことを考えたが、言語が複雑になるので取り止めた。

1.2 TAO における余剰引数

この節では TAO における余剰引数の取り扱いについて述べる。これは言語仕様を単純化する過程で生まれた。TAO 固有の文法が出てくるが、単純な読み替えを行えば、Lisp の議論としての一般性は失わない。

TAO では、lambda 式 (に相当するもの) を次のように書く^{†1}。

```
(op* var-list form ...)
```

変数リスト *var-list* は次のような形をしている。

```
([ob ...] [:option option ...+] [. rest])
```

と書く。つまり、最初に必須変数 *ob* が並び、オプション変数 *option* は :option の右に並ぶ。最後、ドットの後に余剰変数 *rest* が (高々 1 個) 来る。必須変数、オプション変数、余剰変数はなくてもよい。ただし、変数リストが余剰変数 1 個だけからなる場合は、リストにせず、単に

rest

^{†1} *op** で作られる関数は下向き専用の lambda 式に相当する。以下、*op** を lambda と読み替えても支障はない。

^{†2} TAO では論理プログラミングとの関係もあり、未定義値はファーストクラスデータである。これは単独のアンダースコアで表現される。

^{†3} 「並び」と呼ぶが、これはリストとは限らない。

^{†4} 構文とは Common Lisp の特殊形式に相当するものである。

と書く。これらの変数はキーワード以外のシンボルであり、すべて静的な変数である。

ほかの Lisp と同様、必須変数は対応する引数がなければならないもの、オプション変数は対応する引数がなくてもいいものである。TAO は、オプション変数に対応する引数がなかった場合、オプション変数の初期値を「未定義値^{†2}」という特殊な値にする。なお、キーワード変数はない。

関数の呼び出しは、次のような関数式によって表現される。

```
(fn [arg ...] [. rest-args])
```

Rest-args は余剰引数指示 (*rest argument specifier*) と呼ぶ。この式は、Common Lisp の通常の関数呼び出し式と、*funcall*、*apply* をすべて包含している。以下に例を示す。ここで、*x* = (3 4 5)、*porm* = (#'+ #'-) とする。

```
(+ 1 2 3) ⇒ 6 ; 通常の呼び出し
```

```
(+ 1 2 . x) ⇒ 15 ; (apply #' + 1 2 x) に相当  
; x が余剰引数指示である。
```

```
(+ 1 2 . (cdr x)) ⇒ 12  
; (apply #' + 1 2 (cdr x)) に相当  
; (cdr x) が余剰引数指示である。  
; TAO は陽に書かれたドットを「意図的  
; ドット」として特別扱いする。つまり、  
; これは (+ 1 2 cdr x) とは異なる式である。
```

```
(- (car porm) 1 2) ⇒ 3  
; (funcall (car porm) 1 2) に相当  
; 関数部の頭に - をつけると評価される。
```

```
(- (car porm) 1 . x) ⇒ 13  
; (apply (car porm) 1 x) に相当。
```

意図的ドットやアンダースコアが文法として風変わりであるが、どれもふつうの Lisp の式として書けることに注意しよう。本稿で詳述する必要のあるのは、余剰変数の意味と、余剰引数指示の扱いだけである。

ボディ内で余剰変数は、余剰引数のどれかと、それを含む後ろの引数の並び^{†3} (余剰引数残 (*rest argument remainder*) と呼ぶ) を保持している。しかし、変数としては、つねにその余剰引数残の先頭にある引数に束縛された読み出し専用の変数である。余剰引数が最初から空であれば、余剰変数の初期値は未定義値である。余剰変数を余剰引数残の 2 番目以降の引数に束縛しなおすには *rest-args-pop* 構文^{†4} を使う。この説明でわかるように、余剰引数残はファーストクラスデータではない。

```
(rest-args-pop rest)
```

余剰引数残をポップする。つまり、余剰引数残の先頭の引数を取除き、いままで2番目であった引数を先頭にする(剰余引数残が一つ短くなる)。そして新しい先頭の引数に余剰変数を束縛しなおす。ただし、余剰引数残が空になった(あるいは、空だった)ときは余剰変数を未定義値に束縛しなおす。ポップされた先頭の引数が返される。

(rest-args-null rest)

余剰引数残が空であれば #t (真) を返し、残っていれば #f (偽) を返す。

(rest-args-init rest)

余剰変数と余剰引数残を関数呼び出し時の値に戻す。

通常の Lisp のように、残っている引数の並びをリストとしてファーストクラスデータに昇格させたい場合は、次の rest-args-list 構文を使う。

(rest-args-list rest)

現在の余剰引数残を新しいリストにして返す。ボディの最初でこれを別の補助変数に代入し、以後その変数を使って処理を行えば、従来の Lisp の余剰変数と同じプログラミングができる。なお、rest-args-list を実行しても余剰変数の値と余剰引数残は不変である。

このほか、余剰引数残の長さを返す構文 rest-args-length がある。

[例 1] 不定個の数を受け取り、その自乗和を求める関数。

```
(defun square-sum x
  (let ((sum 0))
    (loop (:until (rest-args-null x))
      ; 余剰引数残がなくなるまで
      (incf sum (* x x))†
      ; sum を increment していく
      (rest-args-pop x))
    sum ))
```

[例 2] 不定個の引数を受け取り、それをリストにして返す関数。

```
(defun list x (rest-args-list x))
```

次に関数式の中に余剰引数指示があった場合について述べる。余剰引数指示がこの関数式を含む関数での余剰変数であった場合と、それ以外の式であった場合とで動

† この式は TAO 独特の自己代入式である。

```
(incf sum (* x x))
```

と読み替えればよい。

作が異なる。余剰引数指示が余剰変数であれば、その時点での余剰引数残があたかも引数の並びであったかのように関数に渡される。余剰引数指示がそれ以外の式であれば、余剰引数指示の評価結果がリストであることが期待され、このリストがあたかも引数の並びであったかのように関数に渡される(このリストは呼び出された関数からは見えない)。リストでなければ空の並びと解釈される。これにより、残っている引数を動的な関数呼び出しのネストで受け渡ししていくとき、暗黙の cons が起こらない。

[例 3] 評価すべき関数が trace 表に出ていれば関数名を印刷してから実行するという debug-funcall。関数の引数はそのままタライ回しされる。

```
(defun debug-funcall (fn . args)
  (if (tracedp fn) (print-fn-name fn))
  (_fn . args))
```

この仕様では、1.1 で述べた (1) の問題が自動的に解決している。すなわち、余剰引数をリストとして扱おうとした場合には、それがたとえ apply でリストとして渡されたものであっても、rest-args-list によって必ず新しいコピーが作られる。

TAO の余剰変数の仕様は、MacLisp の LEXPR 型関数の引数受け渡しにやや似ている。LEXPR では不定個引数に対してランダムアクセスが可能であるが、TAO の余剰引数指示により関数の動的な入れ子で不定個引数を次々と簡単にタライ回ししていくことはできない。このようなタライ回しがどれほど重要であるかは、文献 [6] に述べたオブジェクト指向における委譲やメソッド探索の例題を見れば了解できるであろう。

1.3 実装法

前節で述べた余剰変数の仕様の実装法の概略について述べる(図 1)。TAO の実装はこれとは少し異なる。

(1) 余剰引数指示のない関数式での余剰引数はスタックに順番に積む。ただし、余剰変数のスタック内のスロット(図 1 では z のスロット)は管理用に空けておく。このスロットで、余剰変数が現在どの余剰引数残を指しているかを表す。全部積み終わったら、余剰引数の終わりを示すマークを入れる。

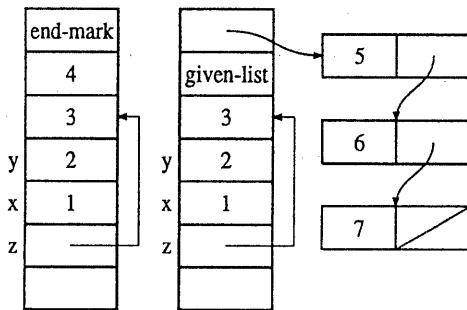
(2) apply で余剰引数がリストで渡された場合は、スタックに given-list というマークを積んでからそのリストを

積む。なお、*given-list* のマークは余剰引数の終わりを示すマークの役割もはたす。

(3) 余剰引数指示が余剰変数であった場合は、単にスタックの内容 (積まれている余剰引数残) をスタックからスタックへコピーするだけである。必要なら、*given-list* の初めほうの部分をスタックに展開する。

上の説明から、*rest-args-pop* や *rest-args-list* がどのような処理を行なうかはほぼ自明であろう。これから推察されるように、この処理は十分に軽い。余剰変数が静的変数に制限されていることもそれに効いていることに注意されたい。

(foo 1 2 3 4) の場合 (foo 1 2 3 . kk) の場合



foo のフレーム foo のフレーム

ここで *foo* は (*defun foo* (*x y . z*) ...) で定義されているとする。また *kk* = (5 6 7) とする。なお、ここで余剰変数 *z* の位置が順番通りになっていないことに注意。

図1 余剰変数の実装概念図

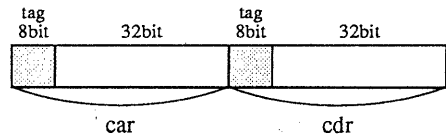
2. スプライサ (Splicer)

スプライサは Common Lisp の多値のごく簡単な拡張である。しかし、この拡張により多くのムダな *cons* が省けるようになった。以下では、なぜスプライサの概念が必要になったかを、TAO の設計過程を振り返るところから説明しよう。

2.1 基本演算におけるムダな *cons*

TAO にはほかの Lisp と同様、いわゆるフルワード型のデータがたくさんある。たとえば、倍長整数 (*double-fixnum*)、倍長浮動小数点数 (*double-float*)、有理数、複素数、文字列、部分文字列などである。これらはちょうど 1 セル相当のメモリを使う (図2)。このうち、TAO に独特

なのは部分文字列というデータ型であろう。以下の説明で必要なので、これについて簡単に述べておく。



double-fixnum, *double-float*

全 64bit で数値を表現。タグ部は *fixnum* になっている。

rational *car* 部, *cdr* 部で分子・分母

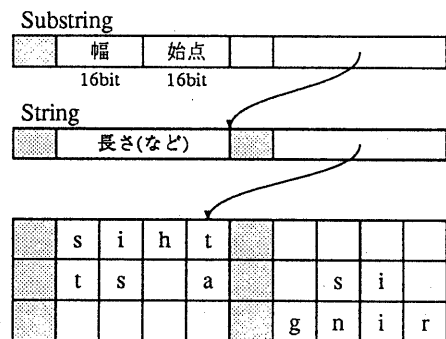
complex *car* 部, *cdr* 部で実部・虚部

string *car* 部, *cdr* 部で文字数・文字列の本体

substring *car* 部, *cdr* 部で部分列範囲情報・親文字列

図2 主なフルワード型データ

TAO では文字列の部分列を取り出す基本演算は、文字列の本体 (つまり、文字の配列) をコピーせずに、親の文字列の (部分) 範囲を示す 1 セルの大きさの記述子 (ヘッダ) を作るだけである。重要なのは、記述子が親の文字列の本体の内部を直接指し示すのではなく、親の文字列ヘッダ経由で、部分列の範囲を指し示すことである (図3)。こうすることにより、文字列本体を指している文字列ヘッダがユニークになり、文字列本体のメモリ管理が容易になる。実際、多くの基本演算で、ゴミになったメモリブロックを陽にシステムに返却することが可能になり、実時間 GC の負担が大いに軽減される。



ここで始点が 5、幅が 2 であれば、部分文字列は "is" を表わす。

図3 部分文字列と文字列

さて、これらフルワード型データに関し、TAO の設計当初から、次のような問題を解くことが目標となった。

(1) 数値演算の中間結果によるムダなセル消費を、少なくともメモコンプログラムのレベルではなくしたい。たとえば、

```
(+ (* a b) (* c d))
```

という倍長整数の計算を行なうと、(* a b) と (* c d) という中間結果で使用されたセルはただちに不要になってしまう。機械語としての TAO では、こういう一過性のセルの消費を軽減できるようなきめ細かい制御を可能にしたい。

(2) 同じく、文字列演算の中間結果によるムダなセル消費を軽減したい。たとえば、文字列の相等比較を行なうのに、

```
(string= str1 str2)
```

という標準関数を用意したとすると、部分列の比較は気持ちよく

```
(string= (substring str1 start end) str2)
```

といった書き方にしたい。ここで、substring は文字列 (あるいは部分文字列) から、新たな部分文字列を作り出す関数である。しかし、これでは、比較するためだけにセルを消費することになってしまう。Common Lisp ではこの問題を回避するために、string= を

```
(string= str1 str2 :start1 :end1 :start2 :end2)
```

というキーワード引数付きの関数にしている。これは効率のためとはいえ、言語設計の「美学」にそぐわないと我々は考える。(機械語だったら許すという考え方があってもいいが、文字列処理プリミティブがこの形だと、その上にユーザが積み上げていく文字列関数群も結局この形になってしまう。「三つ子の魂、百まで」である。)

TAO の設計当初は、「蜻蛉引数」というメカニズムを用意し、Common Lisp の *dynamic extent cell* に相当するものを実現しようとした。しかし、実装が複雑になることと、ユーザにそれを開放するとシステムダウンにつながる致命的なバグが発生する可能性があることから断念した。また、Common Lisp の string= の気持ち悪さを解

† TAO では splicing-unquote を ,@ ではなく、... と書く。

‡ この文法に異和感を覚えるかもしれないが、もともとドットは Lisp の文字の中では昔からかなり特殊な使い方をされてきた。

消するために、キーワード引数の仕様を少し変更し、

```
(string= str1 :begin :end str2 :begin :end)
```

という形式にすることも考えた。すなわち、キーワードの現われる順序を規定してしまう代償として、キーワード引数を必須引数のあいだにも書けるようにするわけである (キーワード引数を省略していいのはもちろんである)。こうすると同名のキーワードが重複して使える。これで :start1 と :start2 があるという気持ち悪さは解消できる。なお、キーワードの順番を覚えやすくするために、アルファベット順にするというコンベンションを使う。しかし、これも言語仕様を単純化するという基本方針のもとでキーワード引数そのものをなくすことになり、取り止めとなった。

2.2 スプライサの定義と応用

スプライサは backquote における splicing-unquote の概念[†]を一般化したものである。これは Common Lisp の多値を拡張した概念である。いわば *latent consing* により、部分文字列や倍長数の計算などによるセルの浪費を防止することができる。

スプライサは

```
.form
```

というシンタクスをもち、関数の引数の位置にのみ書くことが許される。これは

```
(splicer form)
```

という式のシンボルマクロと考えるとよい。なお、ドットと *form* の間に空白を空けてはいけない[‡]。form にはカッコで始まる任意の式が許される (TAO には関数式、述語式、メッセージ式などがある)。

スプライサは、form が多値を返したとき、そのすべての値を順番に関数の引数として展開する。form の値が多値でないとき、スプライサは単に

```
form
```

と書いたのと同じである。

関数から多値を返すには、values あるいは values-fn という関数を使う。

```
(values V1 V2 ...)
```

```
(values-fn fn V1 V2 ...)
```

どちらも、V₁, V₂, ... を多値として返す。values で多値を返したのに、受け側がスプライサでない場合は、その

最初の値 V_1 (主値) のみが渡される。なお、 V_1 もなければ未定義値 $_$ が渡される。values-fn で多値を返したのに、受け側がスブライサでない場合は、fn に $V_1, V_2, \dots$ を引数として与えて実行した結果の値が渡される (fn の中でまた values-fn の値が返ることもあり得る)。

以上の三つ、すなわち、スブライサ .form と関数 values, values-fn がプリミティブである。スブライサの実装はスタックマシンでは多値を単に順番にスタックに積むだけであり、すこぶる効率的である。式がスブライサかどうかの判定は、値を受け取る場所の戻り番地 (継続) に 1 ビットのフラグがあれば十分である。

次にスブライサの使用法や応用について述べる。

(1) Common Lisp の多値の実現

Common Lisp の多値の実現法を示す。これが TAO にそのまま入るという意味ではない。なお、これには values-fn は不要である。

```
(values-list list) = (values . list)
(multiple-value-list form) = (list .form)
(multiple-value-call fn form1 form2 ...)
= (.fn .form1 .form2 ...)
(multiple-value-prog1 form form1 form2 ...)
= (.(op* #x0 form1 form2 ... (values . #x0))
  .form)
; どこでも多値のリストが作られていないことに
; 注意. 1 章を参照のこと.
(multiple-value-bind (var ...) values-form form ...)
= (.(op* (:option var ... #x0) form ...)
  .values-form)
; option と余剰変数 #x0 は数が合わない
; ときのための対策
(multiple-value-setq (var1 var2 ...) form)
= (.(op* #x0
  (!var1 (rest-args-pop #x0))†
  (!var2 (rest-args-pop #x0)) ...)
  .form)
```

(2) string= に関する怨念の解決

我々が怨念的に「美学」を追求していた string= は

```
(string= str1 begin1 width1 str2 begin2 width2)
```

という 6 引数の関数とする。呼び方は、たとえば、

```
(string= .(string "abcd") .(substring s b w))
```

とする。ここで、

```
(string "abcd") = (values "abcd" 0 -)
```

† (!x y) は (setf x y) と読み替えてよい。

; - は最大幅を指定したと解釈される。

である。(substring s b w) はスブライサ以外からの呼び出しでは、部分文字列を返す (セルを一つ使う) が、スブライサからの呼び出しでは、(values s b w) に同じである (cons しない)。概念的には、substring の中からの返り値が

```
(values-fn #'make-substring-header str begin width)
```

になっているわけである。もちろん、上の例を

```
(string= "abcd" 0 - s b w)
```

と書いても構わないが、上の書き方のほうが推奨されよう。

(3) 倍長数の計算

TAO の上に作られる言語 (たとえば、TAO_n) が、型推論により静的なデータ型検査を行なうものであれば、機械語としての TAO はそれに対応した最適化のための基盤をもっている必要がある。

```
(defun discrim-2 (a b c) (- (* b b) (* 4 a c)))
```

は二次方程式の判別式を計算する汎用の関数である。これには実数であればどんな数を与えてもよい。これに、いま double-float を与えたとしよう。すると値も double-float になるが、計算の途中で、(* b b), (* 4 a c) が二つのムダなセルを消費することは前にも述べた通りである。

いま、discrim-2 の引数のすべてと返すべき値が double-float であることが (宣言や型推論で) わかっているとしよう。すると 64 ビットの浮動小数点数を 2 個の 32 ビット整数 (fixnum) の対で表現して、次のような関数にすることが考えられる。実際、これはマシンレベルでの計算をよく反映している。

```
(defun df-discrim-2 (a1 a2 b1 b2 c1 c2)
  (values-fn #'make-double-float-header
    .(df- .(df* b1 b2 b1 b2)
      .(df* .(df 4) a1 a2 c1 c2))) )
```

ここで、df-, df* などとは 2N 個の fixnum を受け取り、それを N 個の double-float と思って引き算、掛け算を行ない、スブライサには結果の double-float を表わす 2 個の fixnum を返し、スブライサ以外にはちゃんと cons した double-float を返すプリミティブである。df は二つの fixnum で表現された double-float への変換プリミティブである。なお、いうまでもないことだが、fixnum の演算では cons が起こらない。df-discrim-2 の呼び方は

.(df-discrim-2 .(df a) .(df b) .(df c))

といった感じであろう (スプライサでなくてもよい)。

このような書き方はメモコンプログラマにしかできないかもしれないが、ユーザに見えないようなある程度の自動化は可能であろう。重要なことはスプライサが自動化の基盤を与えていることである。

以上のようにスプライサは Common Lisp の multiple-value-call を多値の基本にしたようなものだが、関数の任意の引数位置に書けるところと、values-fn によって多値の受け取り方で返す値を変更できるところで、Common Lisp の多値より本質的に機能が高い。そして機能向上のわりに実装の複雑さは上がらない。

3. おわりに

本稿で論じた問題は、一見些細なことのようにであるが、Lisp の関数呼び出しの基本的なセマンティクスに関わっている。

Lisp における不定個の引数の取り扱い、引数の並びをリストというファーストクラスデータとして扱えるという点でふつうの手続き型言語より強力である (関数引数のみならず、「引数並び引数」もあるというべきか)。しかし、その強力さは、ユーザが意図していないムダなセルの消費につながる。本稿の1章で述べた方法はそれを回避するものである。

関数型言語として Lisp を見たとき、データのモジュール性と、「引数のモジュール性」がなるべく一致していることが望まれる。2章で述べたように、部分文字列の記述子はデータとしてのモジュール性はもつが、部分文字列の記述自体を引数として渡そうとしたときにメモリ消費のムダ、あるいは不自然な引数書法を覚悟しなければならなくなる。スプライサはこの気持悪さを効率上および見かけ上かなり改善したといえる。

最初にも述べたが、本稿で述べたアイデアはストックハードウェア上の一般的な Lisp にほとんどそのまま適用可能である。新しい Lisp の設計に際して大いに参考になるであろう。

[文献]

- [1] 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, 56-1, 1990.
- [2] 竹内, 吉田, 天海, 山崎: 新しい TAO の設計, 記号処理研究会, 56-2, 1990.
- [3] 天海, 山崎, 竹内: 新 TAO のメッセージ伝達式, オブジェクト指向計算ワークショップ, 1991.

- [4] 山崎, 天海, 竹内, 吉田: TAO/SILENT の論理型プログラミング, 記号処理研究会, 64-1, 1992.
- [5] 天海, 竹内, 吉田, 山崎: TAO/SILENT のソフトウェアアーキテクチャ, 日本ソフトウェア科学会第8回大会, pp.57-60, 1991.
- [6] 竹内, 天海, 山崎, 吉田: TAO のオブジェクト, 記号処理研究会, 65-3, 1992.
- [7] J.McCarthy, et. al.: Lisp 1.5 Programmer's Manual, second edition, MIT Press, 1965.
- [8] D.Moon: The MacLisp Reference Manual, MIT, 1978.
- [9] W.Teitelman: INTERLISP Reference Manual, Xerox, 1974.
- [10] J.Padget, G.Nuyens, et. al: The EuLisp Definition Version 0.6, AFNOR, 1989.
- [11] D.Weinreb, D.Moon, R.Stallman: Lisp Machine Manual, Fifth Edition, System Version 92, LMI, 1983.
- [12] G.L.Steele Jr: Common Lisp the Language 2nd ed, MIT Press 1990 (邦訳: 井田訳監修, Common Lisp 第2版 (bit 別冊), 共立出版, 1990).
- [13] W.Clinger, J.Ress (ed.): Revised⁴ Report on the Algorithmic Language Scheme, 1991.