

TAO のオブジェクト

竹内郁雄 天海良治 山崎憲一 吉田雅治

NTT 基礎研究所, NTT ソフトウェア研究所, 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 のオブジェクト指向メカニズムについて述べる. TAO は, TAO 言語族ともいふべき多数の言語の (拡張関係を順序とする) 半順序集合のベースとなる「機械語」であるため, TAO のオブジェクト指向は, 多様なタイプのオブジェクト指向計算パラダイムに対応できる基本メカニズムを与えるという設計が行なわれた.

TAO のオブジェクトは Scheme などの let によって作られる無限存続の静的環境を, メソッド表, 不明メソッドハンドラなどが付随したファーストクラスのデータとして再構成したものである.

本稿は, 最近再設計された TAO の骨格を紹介し, 次にオブジェクト指向のための 20 種類を下回る小さなプリミティブについて述べる. 最後に, これらのプリミティブをベースにして, クラス, 継承, 委譲などのオブジェクト指向概念が実現できることを例示する.

Objects in TAO

Ikuo Takeuchi Yoshiji Amagai Kenichi Yamazaki Masaharu Yoshida

NTT Basic Research Laboratories, NTT Software Laboratories,
NTT Human Interface Laboratories

This paper describes the object-oriented computing mechanism in the TAO language on the symbolic processing kernel SILENT. The object-oriented computing mechanism of TAO is designed to be capable of coping with widely diverse types of object-oriented computing paradigms, since TAO can be deemed as a basis of TAO language family which is partially ordered by the relation of language extensions. Recently, TAO itself was totally so re-designed to be truly a *machine language* for the SILENT machine.

Objects in TAO are essentially a lexical environment of indefinite extent created by let in such lexical scoping languages as Scheme, but they are augmented with method table, missing method handler, etc. and are qualified as first class data.

This paper at first describes the core part of TAO which was re-designed recently and then describes a small set of less than twenty primitives for object-oriented computation. Finally, it illustrates several examples of the usage of these primitives to realize various object-oriented concepts such as class, inheritance, and delegation.

まえがき

我々は、記号処理カーネル SILENT⁽¹⁾ 上の「機械語」TAO を設計している^(2,3,4)。TAO は、Lisp 風の関数呼び出しや実行制御をベースに、Prolog 風のユニフィケーションやバックトラック、Smalltalk 風のメッセージ送信を融合した ELIS 上の旧 TAO (以下 DAO と呼ぶ) の流れをくむ。TAO の設計は概略の設計を終えていたが、機械語としてのリファインを徹底するために、最近ゼロから再設計を行なった。本報告は TAO のオブジェクト指向部分について述べるが、Lisp や論理パラダイム部分にも大きな改良があったので、必要なところでは言及する。

1. TAO 再設計の基本方針

TAO は、TAO 言語族ともいうべき多数の言語の半順序集合のベースとなるものである⁽⁵⁾。TAO はこの言語族の中で最も SILENT に近い機械語という意味で TAO₀ とも呼ぶ。より高機能あるいは高級な言語 TAO_n は、TAO₀ を底とする半順序集合のどこかに存在するものである。

TAO (TAO₀) には次のことが宿命づけられる。

- (1) 仕様は可能なかぎりコンパクトで明解・厳密であること。コンパクトさの目標は Scheme の仕様書⁽⁶⁾であるが、TAO は Scheme より多機能なので、これはあくまでも精神論。
- (2) すべての仕様は SILENT 上でのファームウェアによる実装において性能値が保証されていること。
- (3) それ自身でマルチパラダイム言語機能の原料をすべて押えており、TAO_n ($n > 0$) への拡張に困難をもたらさないこと。また、派生的な機能で実行性能を決める主要因にならないものは可能なかぎり省く。

TAO のオブジェクト指向パラダイムの設計にあたっては、最小のプリミティブで最大の機能と (種々のタイプの TAO_n への) 言語仕様適応性の実現を目指した。

2. TAO の基本骨格

TAO のオブジェクトを述べる前準備として、TAO の基本骨格で必要なものについて最小限の紹介をしておく。

2.1 式 (expression) とフォーム (form)

Lisp では S 式で書かれたデータをそのままプログラムとして実行できた。しかし、TAO は一般にデータをそのままではプログラムとして認めない。データ (ここでは式と呼ぶ) をプログラムとして解釈するにはフォーム化 (formulation) を行なってフォームにする必要がある。

フォーム化は (構文要素を評価する) 構文 form で行なう。式 *expr* のフォーム化は次のように書く。

† フォームは first class data (以下、甲種データあるいは単にデータと呼ぶ) であるが、フォーム化した環境のスケルトン (環境そのものではなく環境の骨組み — 構文テキストの構造のみに依存する) と同一の環境スケルトンの環境でなければ評価できない。なお、システム内部データでユーザに公開されていないものを乙種データと呼ぶ。

(form *expr*)

厳密に言えば、この式をフォーム化したフォームの評価によって、*expr* がフォームになる。しかし、このような厳密な記述をいちいちするのは面倒なので、区別が必要な場合以外は、式とフォームは同一視し、単に式と呼ぶ。

フォーム化は部分評価あるいはセミコンパイルにほぼ等しい概念である[†]。

2.2 式の分類

式は図 1 のように分類される。関数式は (append *x y z*) のように丸カッコ、述語式は {append *x y z*} のように波カッコ、メッセージ式は [obj (sel *arg ...*)] のように角カッコで書く (アンダースコアは関数、述語、述語の引数のように本来評価しないものを強制評価するために使う)。本来は関数式をフォーム化したものは関数フォームと呼ぶのが厳密であるが、ここでは両者をともに単に関数式と呼ぶ。マクロ式はフォーム化のときに展開されるので、「マクロフォーム」という概念は存在しない。構文はいわゆる特殊形式のことである。

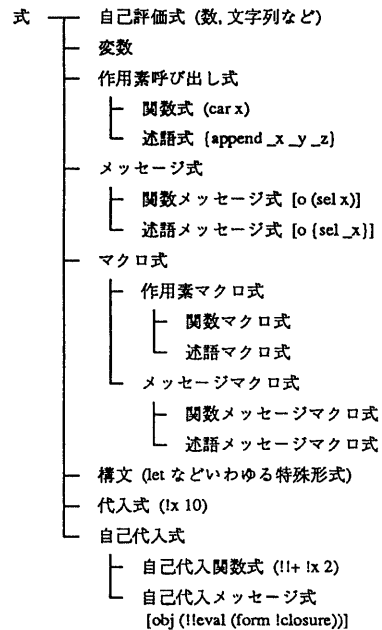


図 1 TAO の式

2.3 作用素 (operator)

TAO には Lisp での関数のほか、Prolog での述語がある。作用素はこれらを統合した概念である。さらに、一つの作用素が関数と述語の両面を備えている場合がある。これを両生作用素 (amphibious operator) と呼ぶ。

作用素は甲種データであり、次のような作用素生成構

(op 仮引数部 ● ボディ)	単純関数
(op* 仮引数部 ● ボディ)	動的関数
(op@ 仮引数部 ● ボディ)	関数クロージャ
{op (仮引数部 ● ボディ) ...}	単純述語
{op* (仮引数部 ● ボディ) ...}	動的述語
{op@ (仮引数部 ● ボディ) ...}	述語クロージャ

(amphibious *fn pred*)

```
(op (x y) (/ (+ x y) 2))
```

作用素からもとの作用素生成構文式を得るには、関数 `express` を使う (いつも可能とはかぎらない).

Lisp の上でオブジェクト指向プログラミングの実験を行なうのに、let などで作られた静的環境をオブジェクトの内部状態と見なし、そこで作られた無限存続 (*indefinite extent*) の lambda がこの環境を保持していることを利用するテクニックがよく使われる⁽⁷⁾。TAO もオブジェクト指向と従来のパラダイムとの融合にこのアイデアを使う。しかし、オブジェクト指向固有の機能が必要なため、ふつうの let と lambda そのものではなく、独自の構文 (obj-let と op@) を使う。逆に、Lisp の意味でフル機能の関数閉包が欲しいときは、obj-let と op@ のもつ機能のサブセットを利用するわけである (実際、Common Lisp に近いある TAO_k ($k \geq 29$) では、obj-let が let、op@ が lambda と呼ばれるであろう)。

- (1) 内部状態 (オブジェクトスロットの集合)
- (2) 内部状態を操作することのできる作用素 (メソッド) の集合
- (3) そのほかの管理情報

The diagram illustrates the internal structure of an object core. It consists of several components and their interconnections:

- Method Table (メソッド表):** A central table containing:
 - メソッド (Method):** Points to a list of methods (メソッド).
 - 不明メソッドハンドラ (Unknown Method Handler):** A handler for methods not found in the table.
 - 補助情報 (Auxiliary Information):** Labeled as **管理情報 (Management Information)**, it contains additional data.
- Slot Table (スロット表):** A table containing slot names: **x**, **y**, and **n**. It is labeled as **スロット名の集合 (Collection of Slot Names)**.
- Object Slots (オブジェクトスロット):** A table containing slot values: **aho** and **5**. It is labeled as **スロット値の集合 (Collection of Slot Values)**.
- Object Core (オブジェクトコア):** The entire structure is part of the object core.
- Shared Slots (共有スロット):** Indicated by a dashed arrow pointing to the **aho** slot, labeled **共有スロット (見えないポインタ) (Shared Slot (invisible pointer))**.

Arrows indicate the flow of information and pointers:

- A solid arrow points from the **Method Table** to the **Slot Table**.
- A solid arrow points from the **Slot Table** to the **Object Slots**.
- A dashed arrow points from the **Object Slots** to the **Shared Slots**.
- A solid arrow points from the **Method Table** to the **Shared Slots**.

オブジェクトを作るには `obj-let` 構文を使う。 `obj-let` は次のように `let` 構文と同じシンタクスをもつ。(<> でくくられた部分は評価されないか、特別な評価を受ける。)

```
(obj-let ((<slot-name> slot-value) ...) • <body>)
```

次にこの束縛を内部状態とするオブジェクトを生成する。この時点では、メソッドの集合は空である。次に *body* を実行する。オブジェクトリージョン内では、そのオブジェクトを読み出し専用の静的変数 @ で参照できる。

- 21 -

オブジェクトには図2のように、スロット値の集合(厳密にはスロット値を格納する場所の集合)、スロット名を保持するスロット表、メソッド表のほか、不明メソッドハンドラ、補助情報が含まれている。このうち、スロット値の集合を除いた部分をオブジェクトコアと呼ぶ。

以上の説明でわかるように、TAOのオブジェクトの生成は「静的」であり、動的なスロットの改名・削除・追加はできない。しかし、動的オブジェクトの実現も可能である(本稿では紙数の都合で述べない)。

4. メソッド

オブジェクトリジョンでは、単純作用素、動的作用素のほかにクロージャを作ることができる。このうち単純作用素とクロージャは無限存続であり、オブジェクトにセレクトタの名前で登録できる。これをメソッドという。メソッドの登録は関数 `attach-method` で行ない、登録抹消は `delete-method` で行なう。単純作用素は、ボディからオブジェクトスロットを参照できないので、メソッドとして登録しても無意味のように思われるかもしれないが、オブジェクトに固有の知識を述語で表現したい場合や、オブジェクト毎に関数の定義を別にしたい場合には有効である。しかし、以下では説明を簡略化するために、メソッドにはクロージャしかないものとする。

クロージャを作る `op@` 構文は次のように書く。

```
(op@ 仮引数部 • ボディ)
{op@ (仮引数部 • ボディ) ...}
```

クロージャのボディから見えるのは、構文的に外側にある静的束縛のうち、`obj-let`の変数束縛と変数`Q`、`labels`と`op-let`(Common Lispの`flet`の作用素への拡張版)の作用素束縛、`macro-let`と`message-macro-let`による静的なマクロ束縛のみである。それ以外の`let`や作用素の仮引数での静的束縛、`block`などによる静的継続束縛は一切見えない。これは本当に閉じ込めたい静的環境の制御が自由にできることにつながる。

オブジェクトリジョンで作られたクロージャは、オブジェクトを閉じ込めた作用素である。クロージャに束縛されている変数を `closure` とすれば、`(_closure arg ...)` や `{_closure arg ...}` という作用素呼び出し式によって、そのクロージャを実行できる[†]。

クロージャは概念的には、`op@` 構文の仮引数部情報と `op@` 構文が書かれた構文位置での静的環境のスケルトン(たとえば、スロットアクセスのための「オブジェクト内オフセット」のようなもの)を含んだプログラムコードと、クロージャを作ったオブジェクト自身の対と考えられる。前者は後述する相似オブジェクトの間で共有可能

[†] アンダースコアによって作用素のところを強制評価している。関数式のほうは `(funcall closure arg ...)` に相当する。

なものなので、クロージャスケルトンともいう。

クロージャはメソッドとして登録し、メッセージによって呼び出すのがオブジェクト指向での原則である。TAOにおけるメッセージ送信は次のように書く。

```
[recv (sel • args)]
[recv {sel • args}]
```

前者は関数クロージャ(関数メソッド)、後者は述語クロージャ(述語メソッド)の呼び出しである。これらの式は見かけ上レシーバオブジェクト `recv` の「環境」の下で作用素を呼び出すという体裁をしているが、実際そういう解釈が可能である。たとえば、オブジェクトを「世界」とすれば、オブジェクト固有の述語をメソッドとすることで、述語による多重世界モデルが容易に実現できる。

上の理由でキーワードはセレクトタになり得ない(キーワードは作用素を表わし得ない)。しかし、これは多重パッケージ環境では不便なので、セレクトタパッケージという大局的なパッケージを用意した。“:”で終わるシンボルは自動的にセレクトタパッケージに入れられる。たとえば、`make:`、`describe:` など。

メッセージ、すなわちセレクトタ `sel` と引数 `args` の組のうち、セレクトタは評価されない(通常の作用素呼び出し式と同じである)が、アンダースコアをつければ強制評価が可能である。なお、関数メソッドを呼び出す関数メッセージ式は次のように略記できる。

```
[recv sel • args]
```

本稿では無引数の関数メッセージ式をこの記法で書く。

オブジェクトにメッセージが送られたときのメソッド探索は、オブジェクトコアに含まれているメソッド表で行なわれる。メソッド表はTAOの基本データであるシムタブ(`symtab`)で実現されている。シムタブはシンボルに対応したエンティティを、項目数に関係なく、メモリアクセス最大3回、キャッシュがすべてヒットしていれば平均350ナノ秒、すべてミスヒットの場合でも2マイクロ秒で探索可能な簡易ハッシュ表である。

メソッド表にクロージャを登録するには、関数 `attach-method` を使う。

```
(attach-method obj sel closure)
```

ここで注意したいのは、登録されるクロージャ `closure` に閉じ込められたオブジェクトと、登録先のオブジェクト `obj` が一致しなくてもよいことである。このようなクロージャは駐在クロージャ(`alien closure`)といい、一致しているクロージャは自家製クロージャ(`home closure`)という。駐在クロージャは最も高速な委譲(`delegation`)のメカニズムである。自家製クロージャをメソッドとして登録する場合は、クロージャそのものでなく、クロージャスケルトン(これは乙種データである)のみがメソッド表に登録される。これにより相似オブジェクト間でメソッド

表が共有可能になる。

5. 相似オブジェクト

一般のオブジェクト指向言語では同類のオブジェクトをクラスとしてまとめる機能がある。TAO ではさまざまなクラス概念の実装に対応できるようにするため、くくりつけのクラス概念を導入しない。

クラス概念の基礎は関数 `obj-copy` と `obj-new` によって与えられる。どちらもオブジェクトのコピーを行なう。`obj-copy` はスロット値の集合のみをコピーし、オブジェクトコアは共有させる。`obj-new` はオブジェクトコアもコピーするが、スロット表は共有させる。

スロット表を共有している2つのオブジェクトを相似 (*akin*) であるという。さらにオブジェクトコアを共有している2つのオブジェクトは強相似 (*strongly akin*) と呼び、スロット表だけを共有しているものは弱相似 (*weakly akin*) と呼ぶ。Smalltalk のクラスは強相似なオブジェクトの集合と考えることができる。

強相似オブジェクトはオブジェクトコアを共有しているので、`attach-method`、`delete-method`、`attach-missing-method-handler` (後述) などは実行した瞬間にすべての強相似オブジェクトに影響が及ぶ。同一の `obj-let` 構文 (より厳密にはその `obj-let` 構文式の同一のフォーム) が2回以上走った場合は弱相似のオブジェクトが作られる。これから明らかにように強相似のオブジェクトを作る方法は `obj-copy` だけである。

強相似なオブジェクトは常に同一 (*eq*) のメソッドをもつ。弱相似オブジェクトは `obj-new` された当初は同一のメソッドをもつが、`attach-method` や `delete-method` を繰り返せば、もとのオブジェクトと異なるメソッドやメッセージインタフェースをもてる。

6. 不明メソッドハンドラとメソッド発見フック

TAO はくくりつけの継承・委譲の枠組みをもたないもので、これらの設定はメタプログラミングで行なう。TAO はこのメタプログラミングを可能にするいくつかのプリミティブをもっている。

不明メソッドハンドラは、メッセージ式の評価時にメソッドが見つからなかったとき、自動的に呼び出される両生作用素で、通常次のようなインタフェースをもつ。

```
(amphibious
  (op@ (hook-fn selector . args) ~)
  {op@ ((-hook-fn _selector . _args) ~) ...} )
```

この両生作用素は、もとのメッセージ式が関数メッセージ式であれば関数部が呼び出され、述語メッセージ式であれば述語部が呼び出される。`selector` と `args` にはもとのメッセージのセレクトアと実引数の並びが渡される。ふつうのメッセージ送信では、`hook-fn` に渡される実引数は `#f` である。しかし、`method-found-hook` 構文の中に書か

れたメッセージ式の場合には、そこに書かれたフック関数を実引数として渡される。不明メソッドハンドラの返す値が、もとのメッセージ式の値となる。

オブジェクトに不明メソッドハンドラを付随させるには関数 `attach-missing-method-handler` を使い、それを得るには関数 `missing-method-handler` を使う。

メッセージ式の評価において、メソッド探索とその実行は一続きであり、最終的に実行されるメソッドがなんであったかを知ることとはできない。メソッド発見フックはそれを可能にするものである。`method-found-hook` 構文は次のように書く。

```
(method-found-hook <message-form> hook-fn)
```

フック関数 `hook-fn` は1引数の関数でなければならない。`method-found-hook` は、`message-form` のレシーバを評価する。次にセレクトアでメソッド探索を行なう。メソッドが見つければ、それを引数として `hook-fn` を呼ぶ。`hook-fn` から戻ってきたら、`message-form` の評価を続行する。メソッドが見つからなかった場合は、レシーバの不明メソッドハンドラの第1引数に `hook-fn` を渡す。

9節で、これらを使ってメソッドの継承が実現できることを示す。

8. その他のプリミティブ

TAO はオブジェクトの内部構造を甲種データとして観測するために、これまでに述べたもののほか以下のような最小限の関数と構文を用意している。

(slot-list <i>obj</i>)	関数
(shared-slot-list <i>obj</i>)	関数

これらは *obj* のスロット名とスロット値のリストのリストを返す。`slot-list` は、*obj* 固有のスロット、すなわち *obj* を作った `obj-let` で陽に定義されていたスロットのリストで、`shared-slot-list` は外側の `obj-let` で定義されたスロットのリストを返す。

(obj-core-aux <i>obj</i>)	関数
----------------------------	----

obj のオブジェクトコアの補助情報を返し、それとともにその場所を左辺値として返す (代入が可能になる)。

(closure-obj <i>closure</i>)	関数
-------------------------------	----

closure に閉じ込められているオブジェクトを返す。

(selectors <i>obj</i>)	関数
-------------------------	----

obj に登録されているセレクトアすべてのリストを返す。

(method <i>obj sel</i>)	関数
--------------------------	----

obj に登録されているメソッドで、*sel* がセレクトアのもの返す。なければ、`-` (`undef` と読む。未定義を表わす甲種データ) が返る。

(slot-value <slot-name>)	構文
--------------------------	----

slot-nameで表わされるスロットの値を返すが、同時にその場所を左辺値として返す。これを使うとオブジェクトのスロットへの代入を「外側」から行なえる。たとえば、pointのx座標を見るメソッドが

```
(attach-method @ 'x (op@ () x))
```

ではなく

```
(attach-method @ 'x (op@ () (slot-value x)))
```

として登録されていると、

```
(![point x] 100)
```

でpointのx座標を外からの代入によって100にすることができる。これはオブジェクト指向の精神からは外れているが、defstructで作られた構造体をオブジェクトで代用してしまう使い方には非常に便利である。(蛇足であるが、上のように単にスロットの値を返すメソッドは内部的には特別なコードに変換され、メソッド探索のあと3〜4ステップのマイクロコードが走るだけである。)

9. 例

ここでは、いままでに述べた十数種類の「機械語的」プリミティブだけで、いろいろなオブジェクト指向計算の枠組みが可能になることを例で示す。ここに書かれたもの以外も機械語的プログラミングの労を厭わなければそれほど困難ではない。TAO_nでは、obj-letなどが裸で出てくることはなく、マクロなどによる高い抽象化が行なわれるはずである。

[1] 簡単なオブジェクトの生成

```
(!aSpaceShip
; TAOのシンボルはcase sensitive
(obj-let ((x 0) (y 0))
(attach-method @ 'distant:
(op@ () (sqrt (+ (* x x) (* y y)))))
(attach-method @ 'move:
(op@ (dx dy)
(!!+ !x dx)
; 自己代入式で、(!x (+ x dx))と等価
(!!+ !y dy)
@ ))
@ ))
```

この例では、x, yというスロットを宣言したのち、distant:とmove:という2つのメソッドを登録している。obj-letはボディの最後の式の値を返すので、オブジェクトを返すにはこのように最後に@を書く。

[2] メソッドの動的追加

一旦生成されたオブジェクトに、後から自家製クロージャを登録するには注意が必要である。そのオブジェクトを閉じ込めたクロージャを作るために、そのオブジェ

クトのリージョン内でフォーム化されたop@構文などによって作用素を生成しなければならないからである。このために次のようなメソッドevalが各オブジェクトに定義されるのが普通である(実際にはgensymを使って名前の衝突を回避する必要がある)。

```
(!aSpaceShip
(obj-let ((x 0) (y 0))
...
(attach-method @ 'eval
(op@ (e) (eval (form e))))
@ ))
```

直観的な表現をすると、メソッドevalにより、オブジェクトのリージョンに後から入りこんで仕事をすることができる。実際にメソッドを追加登録するには、以下のようにする。(ここで、[x - xpos]は[x (- xpos)]の略記。)

; ある点が宇宙船より(上下45°の範囲内で)
; 左側にあるかどうかを判定する真理値メソッド
(attach-method aSpaceShip 'leftp:

```
[aSpaceShip
(eval '(op@ (xpos ypos)
[[x - xpos] >= (abs [ypos - y])]) )])
```

aSpaceShipに送信したメソッドevalにより、aSpaceShipを閉じ込めた自家製クロージャが作られ、これをaSpaceShipに登録する。

このように、メソッドの動的追加や動的継承にはformやevalが本質的である[†]。

[3] メソッドの直接呼び出し

labelsやop-letを使って、クロージャに静的な名前をつけられ、オブジェクト内部ではメソッドの相互呼び出しをC++のように、メッセージ送信ではなく行なえる。実際、DAOで作ったEmacs ZENの中では約30%がselfに対するメッセージ送信であったので、直接の静的メソッド呼び出しの効果は大きいであろう。

また、非常に頻繁にメッセージ送信を行なう場合は、

```
(loop ~ [obj freq-sel] ~)
```

と書かずに、次のようにクロージャを取り出してしまおうとよい。

```
(let
((aa (block (:name kk)
(method-found-hook [obj freq-sel]
(op* (c) (exit kk c)) ))))
(loop ~ (-aa ~) ~))
```

[4] 不明メソッドハンドラ

上記のevalはすべてのオブジェクトで共通に使われるメソッドである。このようなメソッドは他にもあり、それらを陽にすべてのオブジェクトで定義することは煩雑であり、必要なときにオンデマンドで定義したい。これには不明メソッドハンドラを用いればよい。

[†] evalとformの分離は、defunや作用素生成構文に重要であり、また実行効率の向上にも有効である。

```
(!aSpaceShip
  (obj-let ((x 0) (y 0))
    ...
    (attach-missing-method-handler @
      (op@ (hook-fn selector . args)
        (cond ((system-defined-selector-p selector)
          (attach-method @ selector
            (eval (form (get-op@-code selector))))))
        (return [@ (_selector . args)]))))
    ...))
```

関数 system-defined-selector-p は、すべてのオブジェクトに共通なシステム定義メソッド (たとえばスロットの値を印刷するメソッド describe: など) を判定する関数である。また関数 get-op@-code は、その定義式を取り出す関数で、たとえば、eval に対して次のような式を返す。

```
(op@ (#:gen000) (eval (form #:gen000)))
```

eval というメソッドがないと、不明メソッドハンドラが起動され、自動的にメソッド eval を定義し、それを呼び出す。(return は作用素からの脱出を意味する。)

[5] obj-let のネストと super

従来のオブジェクト指向言語におけるクラス、インスタンスなどの概念の一つの実現法を示そう。インスタンス→クラスの階層は obj-let のネストで表現できる[‡]。クラス変数をもつクラスは次のように定義する。

```
(!クラス
  (obj-let クラス変数
    (obj-let (template
      (obj-let インスタンス変数
        メソッド定義など ...
        @ ))
      クラスメソッド定義など ...
      @ )))
```

obj-let のスロットの初期値の評価は外側の環境で行なわれる。このため最も内側の obj-let から template は見えず、クラス変数のみが見えることに注意。

以下は aSpaceShip のクラス SpaceShip の例である。

```
(!SpaceShip
  (obj-let ((number-of-spaceships 0))
    ; クラス変数の宣言
    (obj-let
      ((template ; インスタンスの型紙
        (obj-let ((x 0) (y 0))
          (attach-method @ 'distant: ~)
          (attach-method @ 'move: ~)
          ...
          @ )))
      ; 以下はクラスメソッドの定義
      (attach-method @ 'make:
```

```
(op@ (x y)
  (!!1+ !number-of-spaceships)
  [(obj-copy template) (move: x y)]))
(attach-method @ 'defmethod:
  (op@ (name closure)
    [template (eval '(attach-method @
      ',name ,closure) ))])
(attach-method @ 'get-os-list:
  (op@ () (get-os-list template)))
...
@ )))
```

関数 get-os-list は、関数 slot-list を使って、obj-let の変数宣言を合成する。たとえば、slot-list が ((x 3) (y aho)) を返してきたときは get-os-list は ((x '3) (y 'aho)) を返す。これは 2 行程度で書ける簡単な関数である。

一番外側の obj-let はクラス変数の宣言であり、返されるのは 1 つ内側の obj-let のオブジェクトである。make:, defmethod:, get-os-list: はクラスメソッドに相当する。

次の式は SpaceShip のインスタンスを一つ作る。

```
(!aSpaceship [SpaceShip (make: 3 4)])
```

メソッド make: を受け取ると、obj-copy で template をコピーする。obj-copy はオブジェクトコアを共有するインスタンス (強相似オブジェクト) を生成するから、同一のメソッド表、同一の不明メソッドハンドラをもつ。よって

```
[SpaceShip (defmethod: 'leftp: '(op@ ~))]
```

とすれば、SpaceShip から過去に作られたすべてのインスタンスにもメソッド leftp: が登録される。もちろん、今後生成されるすべてのインスタンスにも登録される。

[6] スロットの継承

継承にはスロットの継承とメソッドの継承がある。まず、スロットの継承について考える。SpaceShip を継承した DisplayedSpaceShip を作りたいときは、たとえば次のようにする。

```
(!DisplayedSpaceShip
  (eval (form
    'obj-let
      ,(nconc '(((super 'SpaceShip))
        [SpaceShip get-os-list:]
        '(((color :red)))
        (attach-method @ 'move:
          (op@ (dx dy)
            (display-erase x y)
            (!!+ !x dx)
            (!!+ !y dy)
            (display-show x y color)
            @ ))
          (attach-missing-method-handler ~)
          @ )))))
```

ここではクラス SpaceShip から生成されるインスタンスのスロットのリストが必要なので、SpaceShip にメッセー

[‡] Smalltalk の意味でのメタクラスは、クラスに自由に attach-method できるので最初から実現されている。

ジ get-os-list: を送っている。

上記の書き方は煩雑で、わかりにくい。たとえばマクロ obj-let-inherit を定義すれば、次のように書くことができる。obj-let-inherit を定義するのは容易である。

```
(obj-let-inherit SpaceShip ((color :red))
  (attach-method @ 'move: ~)
  ...
  (attach-missing-method-handler ~)
  @)
```

[7] メソッドの継承

メソッドの継承は、上位クラスのメソッドを下位にコピーしてくるのが最も単純であるが、ここでは実際に呼び出されたメソッドだけを、不明メソッドハンドラによりオンデマンドでコピーしてくる例を示す。(このプログラムは、ユーザが使った method-found-hook には対処していないが、対処させるのは簡単である。しかし、煩雑になるのでここでは記述しない。また、関数メソッドだけについて述べる。述語メソッドについてもほぼ同様の構造になる。)

```
(attach-missing-method-handler†
  (op@ (hook-fn selector . args)
    (block (:name kk)
      (method-found-hook [super (.selector . args)]
        (or hook-fn
          (op* (closure)
            (attach-method @ selector
              [Q (!!eval (express !closure))]))
            ; これは自己代入メッセージ式
            (exit kk (.closure . args)) )))))
```

不明メソッドハンドラが起動されると、method-found-hook が呼ばれる。その第2引数がまず評価されるが、最初はフック関数 hook-fn は #f なので、op* 構文が評価されて、その関数が method-found-hook に渡される。次に super へ同じメッセージを送信する。そこでメソッドが見つかると、それを引数にしてフック関数を呼び出す。フック関数では、見つかったクロージャを定義する式を関数 express によって得、自分(もとのレシーバ)のオブジェクトリージョンで評価して新たなクロージャを生成し、それを自分自身に attach-method する。以上の操作により super のメソッドが、自分自身へ登録された。最後に、今生成した closure を直接呼び出すが、終了後に super へ本当に送信がなされないように method-found-hook を exit で抜ける。

すぐ上位のクラスにメソッドがなかった場合の動きはプログラムを追っていただきたい。フック関数は最初に

† block の名前の定義はキーワードで始まるリストで (:name kk) のように書く。これはトレインと呼ばれる記法で、それ自身は評価される式でないことを意味している。(closure . args) は、従来 (apply closure args) と書いていた式とほぼ同じ意味である。

一つ作られるだけということと、そこから exit で抜けるときは途中の不明メソッドハンドラのブロックを全部飛び越していくことに注意されたい。

ここでは親のメソッドを S 式の世界に移す express、それを自分の環境でフォーム化する form、そしてそれを実際に評価する eval が有効に使われている。

[8] そのほか

多重継承、多重継承におけるクラス変数、Flavors 風のメソッド結合、動的オブジェクトなども、これだけのプリミティブで実現できるが、紙数が尽きたので省略させていただく。TAO_n ユーザはこれら(見かけは)面倒な内部メカニズムを気にせず、構文シュガーで甘くなった言語を楽しめるであろう。

おわりに

本稿は、TAO のオブジェクトの機械語プリミティブについて述べ、一般的なオブジェクト指向計算の枠組みの実現例について述べた。TAO はオブジェクト指向計算の動的な側面のうち、メッセージ送信の性能を最も重視し、次にオブジェクト生成の性能を重視した。あとは基本的に解釈実行程度の速度と割り切った。とはいっても、SILENT 上に実装される TAO の解釈実行(特にフォーム化)はインタプリタ重視の DAO の伝統にならない、可能な限り高速のものにする予定である。メッセージ送信とオブジェクトの動的生成・変更の頻度が 1 対 1 といった極端な使用形態でないかぎり、TAO は静的な型付きオブジェクト指向から、委譲を中心とするような動的なオブジェクト指向まで幅広く対処できる機械語になった。

本稿ではコンカレンシ、メッセージマクロ、永続性などについては触れなかった。これらについては、機会を改めて報告したい。

[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] W.Clinger, J.Ress (ed.): Revised⁴ Report on the Algorithmic Language Scheme, 1991.
- [7] 高田、柴山、米澤: Common Lisp による並列オブジェクト指向言語の処理系の設計と実現、日本ソフトウェア科学会第 3 回大会, 1986.
- [8] G.L.Steele Jr: Common Lisp the Language 2nd ed, MIT Press 1990.
- [9] B.C.Smith: Reflection and semantics in a procedural language, MIT/LCS/TR-272, 1982.