記　号　処　理　60－6
プログラミング—言語・基礎・実践—　2－6
（1991. 6. 21）

オブジェクト指向のための手続的計算モデルと型

大竹 和雄
日本電気(株) C&C システムインタフェース技術本部

オブジェクト指向言語のために手続的な計算モデルと型の体系が必要であることを明らかにし、そのような計算モデルの備えなければならない条件について分析する。さらにこれらの条件に合致する計算の枠組みと型を構築する試みを紹介する。この計算の枠組みは、変数と代入や計算の順序など手続的な要素からなるなど従来の関数的なモデルとは基本的に異なるものである。またこの枠組みはひとつのオブジェクト指向言語のためのものというよりも、種々のオブジェクト指向の核となるべき枠組みである。この枠組みの上でオブジェクト指向の継承と抽象データ型、動的束縛などの概念をそれぞれ組み合わせて検討することができる。

An Imperative Computation Model and its Types for Object Oriented Programming

Kazuo Otake
C&C Systems Interface Engineering Laboratory, NEC Corporation
11-5, Shibaura 2-chome, Minato-ku Tokyo 108, JAPAN

We make it clear that object oriented language needs imperative computation model. Then, the requirement breaks down into conditions that the imperative computation model should hold. After these discussion, we present an on going work for creating computation framework and its types. This framework is completely different from functional model because the model include imperative computation elements such as variables, assignment and sequencing. This framework is intend to be used as a kernel of several object oriented languages. Object orientation related notions such as inheritance, abstract data types and dynamic binding can be combined on the kernel and analyzed.

1 はじめに

オブジェクト指向言語は情報隠ぺいや継承や実行メソッドの動的束縛(dynamic binding)などの様々な概念を持ち、それぞれに多くの定式化が提唱されている。それらの個々のモデルの選択はお互いに関係し合い、全体に影響するので他の言語に比べてもはるかに設計が複雑で難しい。このような決定を行なうためには考察の土台となる基礎理論があることが望ましいが、残念ながらオブジェクト指向には欠けているとされていることである。

その意味で型理論のオブジェクト指向への適用が有望視されている[4]。抽象データ型のパラメータ化と多相型や継承に対するサブタイプの応用などである。しかしながら、オブジェクト指向に土台となる基礎理論が無いために型理論の枠組みの上での議論が多い。そのために、オブジェクト指向のさまざまなモデルを共通の土台の上で議論できるような一般性の高いオブジェクト指向のための計算モデルの構築が必要であると考える。同時にそのようなモデルに整合する型の体系も考えなければならない。

第2章ではオブジェクト指向のための計算モデルの持つべき条件を議論する。第3章から第6章でこの条件に合う計算の枠組みの構築をめざした試みを紹介し、第7章でこの枠組みの上でオブジェクト指向の諸概念がどのように表現されるかを見る。

2 手続型計算モデルの要件

オブジェクト指向のための計算モデルの備えるべき要件について検討していく。

2.1 式ベースと手続型

例えば関数型言語の基盤であるλ計算などはすべてが式であり、プログラム全体も大きな式である。式は環境のもとで評価によって値をひとつ与える。この環境から値を作ることが式の意味のすべてであって他に隠れた作用は無い。また式はシンボリックな表現を持ち、式の意味はその表現だけによって決定され、実行の履歴すなわち式を評価する前に行なわれたことなどから影響を受けることはない。このような式を基本的な構成要素とする計算の枠組みを式ベース[1]と呼ぶことにする。

このような式ベースの定式化は多くの良い性質を持つので型に基づくオブジェクト指向の定式化にも用いられている[3]。現在の型理論はほとんど型付きのλ計算という式を主体とするモデルの上で研究されており、型理論をオブジェクト指向へ応用する場合も式を主体とすることが多い。

しかし、式はオブジェクト指向の定式化の土台としては不十分である。物体があってこれを移動するmoveという操作が定義されているとしよう。これを式ベースで解釈しようとするとmoveの結果を式の値として表現しなければならない。従って、moveは移動した後の新しい位置にある物体を値として返すことになる。例えばプログラムの中で物体をx方向に10、y方向に-3移動する記述は以下のようになるであろう。

```
obj := move(obj, 10, -3);
```

そしてmoveは次の型を持つ。

$$\mathrm{OBJ} \times \mathrm{int} \times \mathrm{int} \rightarrow \mathrm{OBJ}$$

しかしこれは厳密には物体objを移動したのではなく、新しい位置に物体を作り出したのであってもとの物体とは別物である。この別物を変数objに代入してはじめて移動が行なわれる。もし、代入が行なわれなければこの操作をmoveと呼ぶことは適切ではないであろう。

一般にオブジェクトの状態を変える操作を式ベースで表現する場合にはこのような方法が取られるが、そもそも代入は式ベースからはみ出した概念である。

このような例においては従来、(少なくとも見かけ上は)オブジェクトのコピーを作るので効率

[1]ここで定義した式ベースということばは普通「関数型」や「関数的」という言葉で表されることが多い。しかし、関数型という言葉が意味する範囲についてはコンセンサスが得られておらず他にも多くの意味で使われることがあるので、議論を明確にするために式ベースという言葉を用いる。

を問題とすることがあったが、それよりも根本的な問題はこの中途半端なスタイルではオブジェクトのアイデンティティがあいまいになっていることである。すなわち、新しいオブジェクトが作られても古いオブジェクトは残っており、代入が行なわれる段階でアイデンティティの移動が起こるからである。

オブジェクト指向プログラミングにおいてはオブジェクトのアイデンティティは非常に重要な概念である。オブジェクトを内部状態を持つ自立した存在として捉え、問題の中からオブジェクトを切り出すことがオブジェクト指向プログラミングの基本的な考え方である。オブジェクトをこのようなものだとすると、状態が変化しても自己の同一性、すなわちアイデンティティを保持することは必要条件である。

上の例で物体がアイデンティティを保ったまま移動することを代入によって間接的に表現するのではなく直接的に表現するためには次のように移動操作を隠れた作用をともなう手続として表現する。

```
true_move(obj, 10, -3);
```

true_move は値を返さない。したがって、その型は move とは異なる。とりあえず次のように表現しておく。

$$\mathrm{OBJ} \times \mathrm{int} \times \mathrm{int} \rightarrow (\text{隠れた作用})$$

さらに、作用によってオブジェクトに変化が生じるので処理の順序が重要な意味を持つ。同一のオブジェクトに対する同一の処理もオブジェクトに変化が生じる前と後ではその結果が異なるからである。式ベースの枠組みでは計算の順序の概念は本質的には無い。計算の順序によって結果が異なることは式ベースでは無い。

以上の議論から、変化しながら自己同一性を保つオブジェクトや計算順序の概念を持つ計算の体系、すなわちきわめて手続的な計算の枠組みとこの枠組みに適合する型の体系がオブジェクト指向に必要であると主張する。続いて、議論をさらに進めて具体的に計算の枠組みに要求される条件を追っていこう。

2.2 アイデンティティ

まず、アイデンティティとオブジェクトの関係をはっきりしておこう。

1. すべてのオブジェクトはそれぞれが固有のアイデンティティを持ち、オブジェクトはこのアイデンティティによって一意に識別することができる。

2. アイデンティティはオブジェクトに対して不変で、オブジェクトの一部または全部が変化を受けてもアイデンティティがあいまいになることは瞬間的にもない。

このようなアイデンティティの概念を組み込むために次のような条件が要請される。

2.3 同一性判定

第2.2 章の 1 より二つのオブジェクトの同一性判定ができなければならない。これは型の体系を考える場合の前提条件となる。

2.4 変化と共有

変化はアイデンティティと表裏一体をなす概念と言って良いほど深い関わりがある。第2.2 章の 2 でオブジェクトのアイデンティティは変化によっても不変であるとしたが、逆の見方をすれば変化が起きてもアイデンティティが保たれるから変化なのであって、アイデンティティが保持されなければ別物であって変化ではないとも言える。式ベースの考え方はまさにこれである。

変化がおきる場合にはまた共有という概念も生じる。複数の物から参照されているオブジェクトは、ひとつの参照を通してそのオブジェクトに起こされた変化が他の参照からも観測される。逆に、もし変化の起こる可能性がまったくなければ共有していても個別に持っていても意味上の差異はない。従って、式ベースでは式の共有という概念は意味上は存在しない。

2.5 局所的なオブジェクトとアイデンティティ

プログラミングにおいて、あるオブジェクトの中に局所的に存在して外側から観測することのできないようなオブジェクトの概念を導入することが多い。局所的なオブジェクトに対する作用は外側に影響を及ぼすことがないなど、局所性の概念は有用で、例えば抽象データ型の実現に重要な役割を果たす。

このときアイデンティティはどのように考えたらよいのであろうか。ひとつには局所的なオブジェクトは局所的なアイデンティティを持つとすることが考えられる (図1)。局所的なアイデンティティは有効範囲があり、その外では意味を持たない。こうすると、局所的なオブジェクトを外に見せることは局所的なアイデンティティを外へ持ち出すことになるので許されないということになる。

一方、すべてのアイデンティティは大域的なものと考えることもできる。この場合、局所的なオブジェクトはオブジェクトに含まれるように所有されているというイメージではなく、オブジェクトの外側にあって参照されているというイメージになる (図2)。局所的なオブジェクトはただひとつのオブジェクトからだけ参照されているオブジェクトであると言い替えることができる。そのようなオブジェクトが他のオブジェクトから参照されるようになることは基本的には許される。逆に局所性を保持しようとするならば、そのようなただ

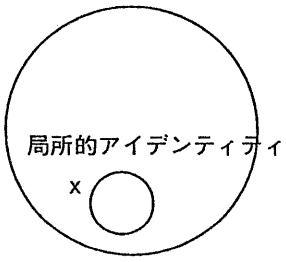

図 1: 局所的オブジェクト

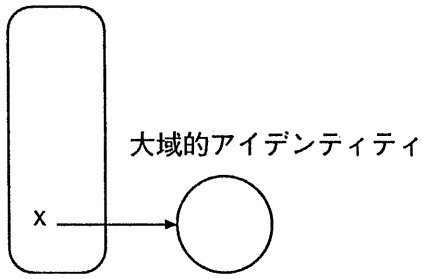

図 2: 大域的オブジェクト

ひとつのオブジェクトから参照されているアイデンティティはそのオブジェクトが積極的にそのアイデンティティを外に見せようとしなければ他から参照される状態にはなることはない。

どちらの考え方にも一長一短があるが、ここでは単純な方をとってすべてのアイデンティティは大域的なものであるとする。

2.6 スタックモデルでは十分ではない

スタックを用いて手続のコール・リターンを実現することは実装上の技術であるが、それは言語の意味にも影響を与える。手続内の局所変数はその手続の実行が終わると消滅する。多くの言語ではこれは局所変数の性質であるとしている。この性質は実はスタックモデルと深く結びついている。この解釈の問題点はあやまって局所変数を手続の外に見せるようにしてしまうと、手続の終了後存在しない変数を参照してしまうことである (不定参照)。一方、我々のアイデンティティの議論から局所的なアイデンティティを認めないので、手続内で参照できるオブジェクトはいつでも外に見せることができる。したがって、不定参照の問題は起こらない。

この要請を局所変数にも拡張すると、一般的には局所変数をスタックの上に取る実現を許さない。実際、後にすべての変数をオブジェクトとしてとらえることについて述べる。局所変数の領域は手続からリターンするときに解放されるのではなく、もはや使われる可能性の無くなったときに

回収される。

2.7 アイデンティティの消滅

オブジェクトを実世界の物体になぞらえて捉えるとオブジェクトが作られたり、また無くなったりすることがあっても良いと考えられる。実際、多くのオブジェクト指向言語はオブジェクトを消滅させる操作を持っていたり、スコープを離れるときに自動的に消滅するオブジェクトを持っていたりする (C++ など)。

スコープの外には存在しない局所的なオブジェクトの存在は不定参照の可能性をもたらし、アイデンティティをあいまいにするのでこれを排除した。しかし、ある参照を通してオブジェクトを破壊することができればやはり不定参照が生じる。

オブジェクトを積極的に消滅させずに参照をはずすことだけを許すならばアイデンティティからみて安全である。

一方で、あるオブジェクトが存在することが不適当になるような場合も考えられる。しかし、このような場合に取るべき処理は状況によるので言語に組み込むことはできない。ある場合にはすべての参照をはずしてまわることができるように設計するであろうし、またある場合にはもはやそのオブジェクトが存在しないことを示す印をオブジェクトの替わりに置くようにするかもしれない。

いずれにしてもアイデンティティはそれへの参照が存在する限り破壊されるべきではない。

2.8 手続的モデルの要件のリスト

以上の議論をまとめると、手続的モデルの持つべき要件として以下のリストを得る。

- アイデンティティが常に明確である
- アイデンティティの同一性判定を持つ
- 変化と共有の概念を持つ
- オブジェクトはすべて大域的な存在とする
- 手続の呼び出しの実現はスタック方式としない
- オブジェクトの消滅を組込みとしない

続いてこれらの要件に合致するモデルの検討を紹介する。

3 モデルの前提

できる限り一般的なモデルを目指すとしてもある程度限られた範囲を対象とせざるを得ない。ここで構築を目指すモデルの前提をここでまとめておく。

1. **クラスに基づく**
   大規模ソフトウェアの開発には型の安全性や型に基づく静的な解析能力が有効であると考えている。オブジェクト指向においては型はクラスの概念と結びつくのでクラスに基づくモデルとする。

   しかし、クラスモデルとプロトタイプモデルはもちろんどちらが望ましいということではない。プロトタイピングによってプログラムを作成する局面ではまずクラスを設計していく方法は不便である。望ましいのは明かにクラスモデルとプロトタイプモデルの両方を使い分けることができるようなモデルである。それは今後の課題であるが、とにかくすべてのオブジェクトは必ず何かのクラスのインスタンスであるとはせずに、クラスとは独立にオブジェクトが存在できる余地を残しておきたい。

2. **抽象データ型を組み込まない**
   抽象データ型の概念はオブジェクト指向において非常に重要であることは言うまでもない。しかし、抽象データ型の情報隠ぺいと継承による情報公開の二面性の整合はこれから吟味されなければならないテーマであると考えている。従って、それを検討する土台としては抽象データ型を根本的に組み込んだモデルは望ましくない。

3. **逐次的モデル**
   オブジェクト指向には並列的なモデルと逐次的なモデルがある。並列計算モデルはそ

れ自身で解決されなければならない課題を持っていてオブジェクト指向がその解決に役立つと期待されてもいる。しかし、本稿では一度に考えなければならない項目を制限するために逐次型モデルで考えていく。

並列的なモデルは今後の課題である。

4 手続的言語の構成要素

オブジェクト指向の計算は手続的に行なわれる。しかし手続的言語の構成要素は多く、それぞれが複雑に関係している。それらの構成要素としてはつぎのようなものが考えられるであろう。式は外から観察される作用を持たないような構成要素のひとつとして含まれる。

- 式
- 手続と手続呼び出し
- 隠れた作用 (副作用)
- 順序的な実行と飛び越し
- 変数と代入
- ポインタ

手続的言語の種々の構成要素をなるべく少ないプリミティブで表現したい。本稿ではこのプリミティブとして

- 手続
- その呼び出し (コール)
- 作用

を選び、できるだけ多くのものごとを説明していくことを目指す。

4.1 計算の順序

計算の順序の概念を手続のコールに結びつける。2.1 章の true_move の例で考えよう。

```
true_move(obj, 10, -3);
```

この呼び出しは隠れた作用を引き起こすが、値を返すことはない。しかし全く何も返さないのではなく、実行が終わって次の実行を始めることのできるようなコントロールを返していると考えらる。1 という型を導入して true_move の型を

$$\mathrm{OBJ} \times \mathrm{int} \times \mathrm{int} \to 1 \text{ (と隠れた作用)}$$

とする。1 はただひとつの値から成るような型である。この型を用いて、実行の順序をコールによって表すことができる。例えば、

$$p; q;$$

のような順序実行は p の返す 1 型の値を受け取って q が実行をはじめると説明することができる。一方、

- goto l
- return(9)

のようにコントロールを返さないようなものもある。このような文の結果の型を 0 と書くことにする。上の return の型は

$$\mathrm{int} \to 0$$

である。0 は値をひとつも持たないような型である。ただし、return にしても goto にしてもこの場所にコントロールが返って来ないだけで、どこかに実行が続いていることに注意。逐次実行の言語ではこのコントロールは実行中常にただひとつだけ存在していると考えることができる。並列実行ではコントロールが一度に複数存在するであろう。

goto l はラベル l 以下を手続と見なして、その手続を (実パラメータ無しで) コールすることとみなすことができる。goto をコールで実現するとスタックが無限に深くなる不都合を感じるが、ここではスタックによる実現を用いないのでその心配はない。

同様に return は自動的にスタックを短くすることはしない。return してもすべての局所変数は存在していて、どこからも参照されていない場合にのみ後で回収される。従って、return もこの手

続を呼び出した時点で作られた接続(コンティニュエーション)をコールしていると見なすことができる。すなわち、手続のコールもreturnも一種のgotoである。ただし、手続のコールは接続を作るが一般のgotoとreturnは作らない。

4.2 作用

moveとtrue_moveの型を比べてみるとmoveの型の持つ情報量がはるかに多い。

$$\mathrm{OBJ} \times \mathrm{int} \times \mathrm{int} \to \mathrm{OBJ}$$

式は結果の値以外を持たないのでこの型は式moveの行なうことの可能性をかなり厳密に限定している。一方、true_moveの表面に現われる型は

$$\mathrm{OBJ} \times \mathrm{int} \times \mathrm{int} \to 1$$

であり、ここに現われない作用はまったく限定を受けないので全体としてはるかに弱い限定しか持たない。

しかしプログラマはtrue_moveの詳細な実現を知らないとしても抽象的にはtrue_moveの起こす作用を理解して使っている。手続の起こしうる作用をある程度限定するような型を導入して手続の型の記述能力を高めることが要求されるが現在のところそれは将来の課題である。

以上の議論から、型の持つ次の二つの意味が浮かび上がって来る。

1. 整合条件
2. 手続の行なう処理をある程度限定するもの

true_moveの型はまさに整合条件である。実パラメタの型を規定し、またtrue_moveを例えばint型を要求する式の一部として用いることが間違いであることがわかる。しかし、起こりうる処理の限定としては作用に関するものが抜けている。式ベースでは整合条件と起こりうる処理の限定が一体となっていたが、手続的な枠組みではこのふたつが分離してきたわけである。

ここで二つの方向が浮かぶ、整合条件とは別の物として独立した作用の型はより自由度が高いのでより適切に作用の意味をとらえられるのではないかということ。一方は、作用には本当に制限がないことが適切なのかという疑問である。作用にも整合条件があるとすることの利用価値はないのかという疑問である。

以下では作用の型についてはこれ以上深入りしないで整合のレベルの型だけを考えていく。

4.3 変数と代入、ポインタ

変数と代入は手続的な計算に中心的な役割を果たす。変数をよりプリミティブな要素から構築する方法はReynoldsのアイデア[6, 5]から導かれたものである。Reynoldsは手続の組合せという概念を導入し、変数を現在保持している値を返す手続と与えられた値を新たに保持するようにする手続の組み合わせであると考えた。Reynoldsはオブジェクト指向を導入していなかったが、オブジェクト指向の立場からは自然な見方である。すなわち、T型の変数は次の型を持つとした。

$$T\&T \to 1$$

$S\&T$は型Sとしても型Tとしても働くようなものの型である。こう捉えることによって変数は作用を持つ手続によって表現され、代入は単なる手続のコールとなる。

$$x := 5; \overset{\text{マクロ}}{\simeq} x(5);$$

同様に配列は整数値を与えられて変数を返すような(作用の無い)手続である。

$$\mathrm{int} \to (T\&T \to 1)$$

さらにポインタは変数の変数として定義できる。すなわち現在保持している変数を返す手続(dereferencing)と新しい変数を保持するようにする手続(ポインタ代入)の組合せである。T型の変数へのポインタの型は次のようになる。

$$(T\&T \to 1)\&((T\&T \to 1) \to 1)$$

ポインタpに配列aの3番目の要素を指させることは次の代入(実は手続の呼び出し)によって達成される。

$$p := a(3);$$

5 オブジェクトとクラス

細かな差異を無視すればほとんどのオブジェクト指向言語のオブジェクトは同じ物である。すなわち、オブジェクトとは状態変数(スロット変数、インスタンス変数)と手続き(メソッド、)がそれぞれいくつか組み合わされたものである。

型の概念と関連付けたオブジェクト指向のためのよく使われるモデルであるレコードモデルはこの手続を副作用の無い純粋な関数に限定したものである。

ここで採用するオブジェクトとメソッドの関係のモデルはレコードモデルを検討することによって得られた。

5.1 レコードモデルの解剖

型の観点からオブジェクト指向を見るモデルとしてレコードモデル [2, 3, 1] がひろく使われている。

レコードモデルは単純な構造にメソッド選択や継承の意味づけを含んでいるが、逆にそのかたくまとまっているところが欠点でもある。オブジェクト指向に提案されているさまざまな計算モデルを取り替えて考えるような土台としては向いていない。

レコードモデルはメソッドをオブジェクトの中に持つが、この意味を検討すると次の 3 つがあると考える。

1. **情報隠べい** オブジェクトは抽象データ型であることを前提とすると、メソッドだけがは抽象データの内部表現にアクセスできる。すなわちメソッドがオブジェクトの中にあり、他の手続はオブジェクトの外側にあるとすることは自然な発想である。

2. **動的束縛** 指定されたメソッドの名前に対して実行する手続を選択するメカニズムはオブジェクト指向に必須である。このとき、対象のオブジェクトが異なれば、同じメソッド名に対しても実際に実行されるオブジェクトを変えるダイナミックバインディングの機能が必要であることが認識されている。

   メッセージパッシングモデルとして知られる選択方式はひとつのオブジェクトがメソッドの名前 (とパラメータ) を受取り、このオブジェクトが実行する手続を決定する。このオブジェクトが異なればじっこうされる手続も自然に異なるのでダイナミックバインディングが自然に実現される。手続を持つ構造体のモデルは構造体の要素の選択として実行手続の選択が表現される。

3. **環境** メソッドの手続本体はさまざまな名前を含む。その名前の大部分はオブジェクト変数を指す。すなわち、そのような名前の束縛を使うために手続がオブジェクトの中にあると考える。

1 に関しては抽象データ型の概念はオブジェクト指向において非常に重要であることは言うまでもない。しかし、抽象データ型の情報隠べいと継承による情報公開の二面性の整合はこれから吟味されなければならないテーマである。従って、それを検討する土台としては抽象データ型を根本的に組み込んだモデルは望ましくない。

2 に関してはより一般的に複数のオブジェクトをメソッドの実行手続の選択に参加させる CLOS のような言語のモデルを表現できない。この場合実行手続がどれか特定のオブジェクトの中に含まれているとするのは適当ではない。動的束縛を構造体の要素選択で表現するモデルは融通がきかない。動的束縛をオブジェクトのモデルに組み込んでしまうのではなくて別に定義できることが望ましい。

したがって残るのはレコードの環境としての機能である。すなわち、オブジェクトを環境としメソッドをこの環境で実行される手続とする。

6 モデルの定式化

以上で述べたモデルをもう少し厳密に定式化しよう。環境は名前とオブジェクトの組の集合である。その型は例えば次のようなものである。

(x: int, y: int var, z: char array)

手続的モデルの構成要素は式ではなく句(phrase)である。環境eのもとで句pを実行することを

$$e.p$$

と書くことにする。例えばpが式$x+x$で環境eにおいてxが整数5を指すならば、実行結果は10という値である。このときpの型は次のようになる。ただし、環境eの型をEとする。

$$p \in E \vdash \mathrm{int}$$

この型はある意味では$E \to \mathrm{int}$に近い。しかし、Eは一般の型ではなく環境でなければならない。またpが$y := 0;$でeにおいてyが整数変数を参照するならば実行結果はこの変数に0を代入することである。このとき型は次のようになる。

$$p \in E \vdash 1$$

ここでpという名前は環境eの外側の名前であることに注意。一方pの指す句に現われるxやyなどの名前はeの中の名前である。

このシンタックスの選び方は意図的で、メッセージパッシングにもとづく実行を行なうプログラム言語の多くがメソッドの実行としてこのシンタックスを用いている。次の例はさらにこの見かけの一致をきわだたせるであろう。句pはパラメータを受け取った手続であるかもしれない。

$$o.move(1,2)$$

は手続moveにパラメータ1と2を与えたものを環境oのもとで実行する。

さらにこの句全体も一般にはなんらかの環境のもとで実行される。すなわち、oで見つからなかった名前はこの句全体の置かれている環境に探しに行く。このことから環境のネストが導かれる。例えば

$$e'.e.p$$

はふたつの環境e'とeを明示的に与えている。名前はe、e'の順に探される。

環境も手続のパラメータの一種である。普通のパラメータとの違いは環境として渡されると環境の中の名前を使うことができるということだけである。環境を普通のパラメータとして渡された場合は中の名前を使うことはできない。

7 オブジェクト指向の種々のモデルの分類

以下でオブジェクト指向の中心となる概念である動的束縛と抽象データ型、継承をどのようにとらえるかの基本的な考え方をみる。

7.1 動的束縛

$$o.p(x,y)$$

というコールに対して、実行される手続を選択する方法が必要である。メソッドの選択は特定のオブジェクトの中で行なわれるのではなくコールの位置で行なわれる。メッセージパッシングのモデルのようにレシーバのような特別なものはなく、メソッドとパラメータがあるだけである。実行メソッドの選択は具体的には次のように行なわれる。

- 実パラメータのクラスから実行メソッドを決定する
- 複数のパラメータが関与できる
- さらに型パラメータを指定することもできる

このとき、

(1) オブジェクトoのクラスだけで実行手続を決める場合がメッセージパッシングモデルに対応する。

(2) オブジェクトo、x、yすべてのクラス(型)によって決定する方法がCLOSのモデルである。

(3) さらに型パラメータを指定することができるのでクラスへのメッセージパッシングをモデル化できる。すなわち、クラスCへnewというメッセージ送ることは型パラメータCを指定することに相当する。

```
new[C]();
```

なお、オブジェクトのクラスだけで決定するのではなくオブジェクトごとにメソッドを決定するようなモデルもクラスの概念を弱くとらえる委譲モデルでは考えられる。しかし、すでに述べた

ようにここではクラスに基づくモデルだけを考える。

7.2 抽象データ型

抽象データ型は機能と実現を分離しようとするものである。ここで提案したモデルはオブジェクトとこの内部表現を操作する手続をもともと一体のものとはしなかった。しかし、一般の環境からクラスのインスタンスとしてのオブジェクトを区別して、オブジェクトの内部表現を操作できる手続群をクラスごとに管理して、登録されていない手続 f を

$$o.f$$

の形式で使うことを許さないようにすれば抽象データ型と同等の情報隠ぺいが達成できる。実現の内部表現を知ることができたとしても実現に依存したプログラミングを無制限には許さないからである。

ここで複数のオブジェクトの内部表現を一度に操作するような手続は複数のクラスに登録されるであろう。

7.3 継承

レコードモデルにおいては型の大小関係であるサブタイプの関係によって継承を説明しようとする。

ここで採用する方法は最大限の一般性を得るためにメソッド継承の方式をサブタイプと結びつけることはしない。既存のクラスを継承して新しいクラスを定義する場合は既存のクラスに対して管理されているメソッド群が継承の対象になる。どれを継承の対象とするかどうかはモデルでは決めない。また、継承したメソッドはもとのメソッドを手本とするだけで概念上は異なるメソッドを新しくつくるものとする。このようにすると継承せずに新たに定義することも自由にできる。

8 まとめ

オブジェクト指向のための手続的な計算モデルが必要なことを主張した。そして、そのようなモデルとして検討しているモデルを紹介した。このモデルは未だ完成したものではないがオブジェクト指向のさまざまな性質を検討するための核となるモデルとして構築することをめざしている。

明かに、多くの項目が今後の課題として残っている。まず、作用をどのようにとらえるか、作用を型としてどのように表現したらよいのかという大きな課題がある。また、ここでは逐次的な計算モデルのみを考察したが並列計算モデルへの拡張も重要であろう。

参考文献

[1] Peter Canning, William Cook, Walter Hill, and Walter Olthoff. F-bounded polymorphism for object-oriented programming. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pp. 273–289. ACM, 1989.

[2] Luca Cardelli. A semantics of multiple inheritance. In G. Kahn, D. MacQueen, and G.Plotkin, editors, *Semantics of Data Types*, New York, 1984. SpringerVerlag.

[3] Luca Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, Vol. 17, No. 4, pp. 471–522, December 1985.

[4] Scott Danforth and Chris Tomlinson. Type theories and object-oriented programming. *ACM Computing Surveys*, Vol. 20, No. 1, pp. 29–72, March 1988.

[5] John C. Reynolds. *The Essence of ALGOL*, pp. 543–573. North-Holland, 1981.

[6] John C. Reynolds. Preliminary design of the programming language forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, June 21 1988.