解　説

オブジェクト指向とほかのプログラミングパラダイムの融合†

竹　内　郁　雄††

1. はじめに

本特集でオブジェクト指向パラダイムの最近の具体的な活用や展開，あるいはほかのパラダイムとの具体的な融合などについては十分に紹介されるはずなので，本稿は至って書きにくい．特に筆者は世の中の情勢に関して不勉強なので標記のテーマについて万遍なく博物学的な分類学を展開する能力はない．しようがないので，ほかの記事と少し視点を変えて「オブジェクト指向パラダイムとほかのプログラミングパラダイムの融合」に関するやや抽象的な議論を展開することにする．「解説」というより単なる「独断と偏見」の羅列に陥る恐れが多々あることを最初にお断わり申しあげたい．

プログラミングパラダイムという言葉を初めて使ったのがだれかには諸説あるようだが，公の場で陽に言明したのは1978年にチューリング賞を受けた R. W. フロイドだろう．彼のチューリング賞受賞記念講演は“The paradigms of programming”であった[1]．しかし，実はそれより遡ること10年ほど前に T. S. クーンという哲学者が科学史論という哲学の分野にパラダイムという言葉をもちこんで，相対主義対合理主義という激しい哲学論争を巻き起こした．これがパラダイムのそもそもの起源である．実際，パラダイムという言葉が計算機の分野以外に，経済学や社会学でもブーム的に使われていることはご承知のとおりである．

古い器に新しい酒が盛られた言葉の常として，パラダイムの訳語には困ってしまうのだが，一番通例となっているのは古い訳語をそのまま使う「規範」だろう．この言葉の大体の意味は，それに従えば（ほかのことを考えなくてもよいという意味で）思考が節約でき，同じ理由でコミュニティの中で円滑に意思・情報の伝達が行なえるという行動および表現のガイドラインといった意味である．これの意味合いを強めるとクーンの指したような，学派や流派のもつ「信念」や「先入観」にまでなる．自然科学の発展をこのような概念で説明しきってしまうことの是非はもちろん本稿の関知するところではないが，経済学のようにもともと絶対的な学派や流派のない学問分野と同じようにプログラミングの世界でパラダイムという言葉がいみじくも使われているところが面白い．プログラミングが元来相対主義的な世界であることをフロイドは陽に示したわけである．

しかし，プログラミングパラダイムの定義自身にそれほど厳密なものがあるとは思えない．非常に低レベルのプログラミングテクニックのようなものから，あるプログラミングコミュニティによって共有されているプログラミング言語文化や暗黙のプログラミングスタイルまたは思想のようなものに至るまで，いろいろな階層がある．だから一つのパラダイムを取り上げても，きわめて単純なものや，それ自身がまた非常に複雑な体系をなしているものがあり，ひとからげにして論じ切ってしまうことはもとより不可能である．いま話題にしているオブジェクト指向パラダイムはどちらかというとかなり複雑なものであり，同じことを繰り返して言うようだが，はっきりと定義あるいは規定された概念ではない．

二つの対立したプログラミングパラダイムがあったとき，それらの比較の基準を何にするかが問題である．比較基準には（1）計算速度，メモリ消費量などの計算機資源の経済性と（2）扱いやすさ，すなわち，書きやすさ，読みやすさ，直しやすさ，同じことを書いたときのプログラムの大きさなどのいわば人的資源の経済性の二つがあろう．「美しい」という判断基準は後者のほうに入れることができる．数学的に厳密であるというのは計算機資源の経済性に結びつくものなのか，人的資源の経済性に結びつくものなのか筆者に

† Fusion of Object-oriented Paradigm and Other Paradigms by Ikuo Takeuchi (NTT Software Laboratories).

†† NTT ソフトウェア研究所

はにわかに分からない．計算機資源の経済性はいわばパラダイムの絶対性能とも呼ぶべきもので，多くの場合，対立において主要な問題点にならないハズである（しかし，依然としてこの種の議論が絶えないが…）．これに対して人的資源の経済性は絶対的な測定法がない以上，水かけ論に陥りやすい．だからパラダイム論になる．

さて，プログラミングの世界におけるパラダイムにはほかの分野にはあまりない特徴が一つあると思う．それはパラダイムが対立するだけでなく，割合簡単に融合することがあることである．プログラミングパラダイムの対立というとすぐに Lisp 対 Prolog 論争を思い出すが，別の見方をすると，対立したプログラミングパラダイムは，並置して「使い分ける」ことが可能である．それだけではなく，うまくやれば対立していたはずのプログラミングパラダイムを統合・融合して，新たな（いわば）複合パラダイムを作り上げることができる．ただし，統合・融合には後で述べるようにいろいろな様相がある．

パラダイムの並置や融合は，おのおののパラダイムの内的な構造と外面的な機能のどこに着目するか，つまりパラダイムのどの相に着目するかによって多様な組み合わせが可能になる．けだし，オブジェクト指向パラダイムはこういう相がきわめて豊かに含まれている．パラダイム論ばやりの昨今，オブジェクト指向パラダイムが最も多くのパラダイムと融合しているのはこれと無関係ではない．

2. オブジェクト指向パラダイムの諸相

オブジェクト指向パラダイムの相は計算メカニズムに関する相とプログラミング方法論に関する相の二つに大きく分類され，おのおのはまたいろいろな細かい相に分類される．オブジェクト指向パラダイムをほかのパラダイムと並置あるいは融合するとき，このような分類を行なっておくと話がうまく整理できると思われる．

2.1 計算メカニズムの相

オブジェクト指向パラダイムにおけるメッセージ伝達（メッセージパッシング）は，ほかの言語における関数（または手続き）呼び出しと著しい対照をなす．関数呼び出しでは，通常次のように関数名を主体とし，それに引数を付随させるという文法形式をとる．

function (argument)

この文法形式は function という関数名があるユニークな手続き本体と一対一に対応していることを意味している．しかし，典型的なメッセージ伝達式では，手続き本体にユニークな名前をつけず，次のように手続き本体を間接的に指すメッセージ名を引数と従来呼ばれていたもの（レシーバと呼ぶ）に「送る」という文法形式をとる．たとえば，

receiver message

のような式である．メッセージに対応する手続き本体（メソッドと呼ぶ）はレシーバに依存して決定される．この文法形式を

message (receiver)

というように通常の関数呼び出しと同じ記法で書くと，両者の意味論的な差が明らかになる．ただし，このとき静的に（コンパイル時に）実際に走行すべきメソッドが決定されるようであれば，(いわゆるメッセージ伝達式の文法形式をとっていても）同じメッセージ名がオーバロードして使えるという文法的な差異だけとなり，通常の関数呼び出しとの意味論的な差，つまり計算メカニズムとしての差はなくなる．

静的にメソッドが同定できない場合，実行時にレシーバがなんであるかをその都度調べ，走行すべきメソッドを決定しなければならない．このことは，実行時にデータに自分がなんであるかという名札（自己記述子）が立っていることと，メソッドの検索が必要であることを意味する．

メッセージ伝達という形式の特徴は，一つのデータ構造（オブジェクトと呼ぶ）に付随した関数あるいは手続きをおのずから集中管理できることである．なぜなら，手続きの本体はオブジェクトとメッセージ名のペアでしか同定できないので，実効的にオブジェクトに付随したものになってしまうからである．こういうデータと手続きの一体化は単に両者を何かでくくってまとめて書くといった表面的なものを本質的に越えている．

メッセージ伝達という形式のもう一つの特徴は，関数呼び出しと異なり，レシーバ（あるいはオブジェクト）が「主体性」をもつという認識が可能になることである．現在の大半の計算機の構造からいってデータ構造が動作主体になるというのは（実際に走っているのはやはりメソッドであるから）おおかたのところ幻想にすぎないが，プログラミング上このような認識モデルをもつことは従来のアルゴリズム主体のプログラミングに対してコペルニクス的転回をもたらす．そして一つのオブジェクトが一つのプロセッサと対応づけ

られるという並列処理への非常に有望な道を開いてくれる.

2.2 プログラミング方法論の相

2.1 で述べたようにオブジェクト指向パラダイムでは,オブジェクトという単位でデータ構造とそれに付随した手続き群が一体化される.これはオブジェクト指向における最も原始的なモジュール化である.ここでさらに,オブジェクトの内部的なデータ構造へのアクセスにも一定の歯止めをかけることにより,オブジェクトを単位とするモジュール化は一層堅固なものとなる.この歯止めは,内部の構造へのアクセスを直接ではなく,かならずメッセージを通じてしか行なわせないことによってなされる.計算メカニズムとしては,レコード型のフィールドのアクセスよりはるかに重くなってしまうが,モジュール化の観点からは小さな犠牲とみるわけである.これはオブジェクトの情報隠蔽と呼ばれ,計算メカニズムの相とプログラミング方法論の相との接点となっている.

プログラミングにおいては,同じようなものを何度も書かないという意味での記述量の節約が方法論的に最も基本になる.オブジェクト指向におけるインスタンス,クラス,インヘリタンスの三つの概念はこのためのものである(メタクラスは言語構造に整合性や拡張性を与えるのが主たる役目なのでここでは省く).ここで,インスタンスは個々のオブジェクト,クラスは個々のインスタンスやクラスに共通の性質を一括して記述するという意味でインスタンスやクラスを抽象化したもの(これは自己再帰的な定義である!),インヘリタンスは共有化された記述をほどいて広げる手段を提供する.これらは「もの」の概念を基本とした抽象化と呼ばれ,従来のアルゴリズムあるいは手順を基本とした抽象化とは異なるベクトルの抽象化である.これらの三つの概念はオブジェクト指向パラダイムにすべて必須というわけではない.たとえばインヘリタンス概念を落としたり,インスタンス概念を落としたりして,オブジェクト指向をほかのパラダイムと融合させることもある.

オブジェクトを基本とした抽象化階層,すなわちクラス階層の設計はオブジェクト指向プログラミングにおいて最も難しい問題である.迷ったときの判断基準として現実世界のモデリングの忠実性に重きを置く流儀と,記述量の削減に重きを置く流儀があるが,どちらがよいかは一概には言えない.オブジェクト指向パラダイムとほかのパラダイムとの融合においてどちらかの流儀が暗に想定されている場合がある.

3. オブジェクト指向パラダイムとほかのパラダイムの融合の諸相

ここでいろいろなレベルでのパラダイムの融合についてみてみよう.

3.1 演算子記法との融合

最初に,式の記法に関するパラダイム(前置記法,中置記法,後置記法)のうちわれわれに最も親しみやすい中置記法がオブジェクト指向のメッセージ伝達の相と融合した例をあげよう.これはよく知られたSmalltalk-80[2]の文法である.

$x > y$

筆者は,これは本来構文解析の問題とされていたものをメッセージ伝達という文法と意味論で解釈し直した画期的な発明であると思う.これが真に意味論的であることは,x の値が数であれば数の大小比較,x の値が文字列であれば文字列の辞書式順序による大小比較,などなど動的に意味が変わり得る(type polymorphism と言うらしい)ところから明らかである.筆者らの開発した TAO という Lisp ベースの言語で,比較を

$(x > y)$

と書けるようにしたのは Smalltalk-80 からのヒントによる.これは,通常関数名を書く第一要素に関数以外のものがくるとメッセージ伝達と解釈するという形で Lisp の関数式にメッセージ伝達を融合させたものである.これが融合と呼べるのはインタプリタの核部分で,関数式とメッセージ伝達式が区別されるからである.(通常の Lisp インタプリタではエラーになる分岐スロットにこの意味論がピッタリとはまる.とはいっても,実際にはコンパイラが困るので,端末からの対話でなくプログラムの中に書く場合は普通のコンスセルとは文法表現だけが違う

$[x > y]$

というS式で書いてもらうようにしている.)

なお,オブジェクト指向パラダイムで文法・意味を徹底的に統一した Smalltalk-80 であるが,代入文(これも代入の概念のない言語の存在を思えば一つのパラダイム)だけはメッセージ伝達の形式をとっていない.代入文とメッセージ伝達式を「融合」するためには「左辺値」の概念を導入する必要がある.これは不可能ではないが,Smalltalk-80 の場合それを採用しなかっただけのことのようである.このようにパラダイ

ムの融合にはいろいろなレベルでの選択肢がある．

3.2 条件式やループ概念との融合

Smalltalk-80 におけるもう一つの面白い融合の事例は，条件式とループの表現であろう．Smalltalk-80 は条件式とかループというきわめて手順的な制御構造をたとえば次のようなメッセージ伝達式で表現する．

```
predicate  ifTrue: block
predicate  whileTrue: block
```

これは単なる文法的な細工以上の意味をもつ．それは，パックされた手順が block という形で一つのオブジェクトと見なされているところである．Lisp ではこのようなものを昔から closure と呼んでいたが，どちらかというとあまり使われないものであった．しかし，このように基本的な制御構造をメッセージ伝達式で表現すると，手順の「オブジェクト化」は必然的なものとなる．一般に既存の言語の拡張としてオブジェクト指向パラダイムを取り入れても，もとからあった「手順」あるいはプログラムセグメントを「オブジェクト化」するというレベルの融合はなかなかなしえない．

3.3 ジェネリック関数との融合

実行されるべき手続き本体（メソッド）が，メッセージ名とレシーバのペアで決定されることから，同じメッセージ名をいろいろなオブジェクトで共通に使う，いわゆる名前のオーバロードが可能になる．これは型宣言を多用して関数名のオーバロードを可能にするジェネリック関数の考え方に合い通ずる．

Common Lisp のオブジェクト指向機能として現在検討されている CLOS[3] はこの視点で関数式とメッセージ伝達式を融合する（CLOS は設計中の言語であり，多くの読者にとってまだ馴染みが薄いが，本特集の中の大里，梅村『Lisp 上のオブジェクト指向プログラミング』に少し詳しく述べられているので参照していただきたい）．CLOS では通常の関数式の第一引数（ほかの多くの言語の場合，これがレシーバになる）のほかに，それに続く第二引数，第三引数などもメソッドの決定に参加できるように拡張されている．この拡張により，CLOS から本来の意味でのメッセージ伝達の概念はなくなっている．なぜなら，引数のうちどれがレシーバであるかはっきり同定できないからである．レシーバの確定しているメソッドであれば，メソッドの中からのインスタンス変数へのアクセスはレシーバの内側からのアクセスと見なせ，文法的にも意味論的にもそのような書き方ができるが，CLOS の場合，メソッドはオブジェクトのインスタンス変数を外側からアクセスするという格好になってしまう．

また，データと手続きが一体化するというオブジェクト指向パラダイムのかなり基本的なところも少し形が崩れてくる．手続きがオブジェクトの集合と対応づけられるようになるからである．もっとも，実際に第二引数以降をメソッドの決定に参加させないプログラミングスタイルを守り通せば，クラス階層なども含めてふつうのオブジェクト指向パラダイムそのものが実現されることに注意しておく必要がある．つまり，足し合わせた結果 1+1=2 以上のものが得られたわけである．だから，CLOS の試みは Lisp とオブジェクト指向の真の「融合」と言えるかもしれない．ただし，car や string-equal などの普通の Lisp 関数名をジェネリック関数の枠から外していることと，関数とメソッドを，使用形式からは区別できないにもかかわらず別の概念としている点で，この融合は画竜点睛を欠く．基本的アイデアがきわめて優れているのに，どことなく木に竹を継いだ印象を受けてしまう．第二引数以降をメソッドの決定に参加させた場合，クラス階層の設計は現実世界の忠実なモデル化に基づいたものとはかなり異なったものになるだろう．そうなると上に述べた記述量の削減がプログラム構造の選択の第一原理となり，もの中心というよりアルゴリズム中心の抽象化に重点が移る．

なお，CLOS はクラス階層を動的に変えるいわゆる「動的オブジェクト」に関してかなり意を払っている．ただし，まだ仕様として安定している感じではない．

少し話がそれるが，関数式とメッセージ伝達式を同一の形式で表現するという融合を行なわない場合は，両者が同一言語の中で並置されることになる．これはまた別の意味での融合であり，両者が実際のプログラムの中で細かい粒度で（たとえば，両者が何重にでも式の中で入れ子になれるように）混合できるのであれば十分大きな意味をもつ．このことは，たとえば Smalltalk-80 で tarai のような関数（**図-1**）をどう定義するか悩んでみると明らかになる．これは，一つの整数が残り二つの整数を引数にもつ tarai: というメッセージにしたものか（見た目にとても不自然），また

```
tarai (x, y, z) == if x>y
                     then tarai (tarai (x−1, y, z),
                                 tarai (y−1, z, x),
                                 tarai (z−1, x, y))
                     else y
```

図-1 Smalltalk で tarai をどう書く？

は tarai を計算することだけを知っている Tarai というオブジェクトを作るか（こちらのほうがましだが Smalltalk の文法ではいかにも重い）少し迷うところであろう．もし，ふつうの関数が定義できるのであればこんな悩みは生じない．つまり，パラダイムを一つの言語の中に並置するだけの「混合」あるいは「複合」パラダイムのアプローチにもそれなりに意味がある．

3.4 型言語との融合

データ型がコンパイル時にすべて決定できるような言語では，データ型あるいはデータ構造に自分がなんであるかを示す自己記述子をもたせる必要はない．このような言語ではメッセージ伝達の意味論はクラス階層を含め完全に文法的なものにスリ替わる．つまり，クラス階層は一種の階層的マクロ定義，メッセージ伝達式はマクロ呼び出しになる．しかし，こうしても動的にオブジェクトを生成したい場合には，言語によってはヒープ領域管理のメカニズムを内蔵しないといけなくなる．

では，Fortran にオブジェクト指向パラダイムを融合させることができるか？ 筆者の想像ではある程度のことはできる．しかし，たとえば配列をオブジェクトとしてみようとしたときに困ってしまう．Fortran の配列には自己記述子がないからである（だから，オブジェクトとして受け取ってもサイズやランクが分からない）．まさか，配列のサイズやランクが異なるごとに異なるクラスにするわけにもいくまい．Pascal でも事情は同じだろう．だから，こういう言語ではユーザが定義したものだけをオブジェクトとする制限を課す必要がある．

ところがこうすればユーザ定義のオブジェクトには自己記述子をつけることができ，メソッドの動的な検索も可能になる．逆に型言語であることの特徴が失われてくる．型言語とオブジェクト指向はもともとベクトルが反対向きであり，水と油を融合しようとした感が強くなる．ただし，まったく無意味ではあるまい．型言語であることの制限を弱め，オブジェクト指向パラダイムを言語の中に並置させ，型宣言をコンパイルコードの最適化の補助とするという現実的な解がないことはないからである．融合として深くはないが，従来的なアルゴリズム指向の型言語にオブジェクト指向の機能を入れることのメリットは小さくない．どんなところに置かれてもオブジェクト指向の機能はそれ自身で一つの小宇宙を作ることができる．

3.5 論理型パラダイムとの融合

論理型パラダイムとオブジェクト指向パラダイムの取り合わせは一見奇妙である．論理にはシンボルといった低レベルのもの以外に「もの」という概念が存在しないからである．特に内部状態をもったオブジェクトというものは論理学の教科書をいくらひもといても見つけることはできないだろう．

さて，Prolog ではホーン節の集合が手続き型言語における手続きと同じ働きをするが，Prolog の仕様のままではこれらの「手続き」の集合を構造化あるいは階層化することが困難である．この事情は Lisp も同じで，Common Lisp ではシンボルパッケージによる名前空間の分離と局所関数の導入で関数群の構造化（階層化）を行なった．Prolog でもこれと同様のやり方は（いいか悪いかは別として）可能であろう．

しかし，ESP[4] を代表とする論理型パラダイムとオブジェクト指向パラダイムの融合では，ホーン節の集合をオブジェクトとみることによって，「手続き」の構造化と階層化を行なう．すなわち，ホーン節の集合（公理系と見なすことができる）を「知識」というある程度具体的なもののイメージで捉える．「知識」はそれに何か聞けば教えてくれる動作主体とみれないことはないからである．ただし，ここまでは手続き型言語のパッケージルーチンと本質的な差はない．ホーン節で書かれた知識は同じ質問に対して同じ答えしか返さない．オブジェクトと呼び得るには，クラスに共通の知識に加えて個々のオブジェクトに固有の内部状態が必要である．

実際，現実世界を計算機で処理する場合，時間の経過につれて変化する状態を扱わなければならない．時間を表わす項を各述語に追加すれば，論理の枠内でこれを表現することは原理的に可能だが実際的にはまったく無意味である．これに対して ESP のとった考え方はだいたい次のようにまとめることができよう．「計算機が時々刻々変化する現実世界を，計算機の内部時間の進行につれて変化する内部状態でモデル化する．このように公理系が時間によって変化する状態をもつ『実時間』プログラミングを採用しても，瞬間瞬間では論理の体系として整合がとれている．」

かくしてふつうのオブジェクト指向パラダイムのオブジェクトと同様，オブジェクトとしてまとめられた公理系に「状態変数」をもたせることができる．これが ESP のオブジェクトスロットである．これは Prolog の assert や retract で変更のできる事実節とほぼ

等価と見なしてよい．「知識」に「事実」が加わって一体化したものを「世界」と呼ぶ．これは手続きと内部状態をもつデータがオブジェクトと呼ばれるのと並行している．同じ知識をもつものがクラスで，異なる事実が異なるインスタンスに対応するのもオブジェクト指向の枠組にうまく合致している．

実はここまでは論理型パラダイムの根幹を曲げてきているわけで少々苦し紛れの感なきにしもあらずだが，インヘリタンスが知識，すなわちホーン節の単なる和集合ですむというところで，オブジェクト指向パラダイムによる抽象化機能を採用したメリットが現われた．これで，ふつうだったら平板なホーン節の並びにしかならないところを，きわめて巧妙に階層化あるいは構造化できる．これは関数の階層を手順の挿入位置に注意しながら重ねていって一番上位の関数を作り上げていくのとは本質的に違う．スーパクラスの知識（と構造，つまりオブジェクトスロット）を寄せ集めるだけでいいからである．ホーン節の集合の平板性がここでかえって幸いしたと言えよう（実際には Prolog のホーン節に実行順序依存性があるから 100 パーセントうまくいくわけではない）．

ここで得られた階層性と構造性により，コード（ホーン節）が大量に共有できるようになり，大きなシステムを書いたときのコード量を著しく減らすことができる．なお，筆者はインヘリタンスによって，論理の非単調性が生ずる（たとえば，スーパクラスで真であるものがサブクラスで偽になる）問題は，どうせ純粋な論理からすでに離れていることと，非単調性がオブジェクト指向パラダイムのクラス階層が生来もっているむしろ優れた機能だと考えているので，ESP 関係者があまり弁明に努める（もはや努めていないかもしれないが）必要はないように思う．

ESP にみられる論理型パラダイムとオブジェクト指向パラダイムの融合は，より正確に言えば，論理型パラダイムという柔かい素材にオブジェクト指向という柱を加えて構造物を作ったというところだろう．手続き型言語にオブジェクト指向を加えたのと違い，ここではオブジェクト指向パラダイムが論理型パラダイムと独立には存在していない．実はこれがオブジェクト指向パラダイムの大きな特徴を示している．つまり，オブジェクト指向それ自身が独立して機能するような融合も可能であれば，ほかのパラダイムに骨格を与える，いわばメタフレームとして作用することも可能なのである．これはオブジェクト指向パラダイムがほかの多くのパラダイムとよく直交していることの一つの現れであろう．

Prolog/KR[5] はやはり論理型とオブジェクト指向（というより多重世界モデル）を融合した知識表現のための言語であり，基本的な考え方の多くが ESP に受け継がれている．ただし，オブジェクト指向パラダイムとしてみた場合，動的にクラス階層が変えられる（指定できる），インスタンスの概念がない（実はインスタンスとクラスの概念がクリアカットされていないオブジェクト指向システムはほかにも多いが…）など，オブジェクト指向の範疇に入れていいものかどうか迷うところがある．しかし，オブジェクト指向の言葉で説明することは確かに可能で多分了解性も高い．このことは曖昧でありながらも，オブジェクト指向パラダイムが計算機の世界で，スタックとか再帰法とかコルーチンとかと同じように一つの共通概念としてのパラダイムにまで成長したことを物語っている．

Concurrent-Prolog[6] やそれに基づいた Vulcan[7] もオブジェクト指向を標榜しているが，これはアクター理論[8] でいうオブジェクトに近い．インヘリタンスは静的な階層構造というより，動的に役割分担を行なうというやや独特のものであるが，Vulcan ではマクロという形で静的なクラス階層も実現している．

筆者らの開発した TAO[9] も，論理型とオブジェクト指向の融合に関しては ESP とよく似た考え方になっている．異なるのはインスタンスが ESP でいうオブジェクトスロット以外に事実節をもてることと，スロットがふつうの（Lisp 的な意味での）インスタンス変数であって ESP のような特別の意味づけがされていないことである（TAO は元来論理型計算をユニフィケーションとバックトラックという操作的な意味でしか捉えていない）．

3.6 プロダクションシステムとの融合

プロダクションシステムに基づくプログラミングパラダイムはルール指向と一般に呼ばれる．これも論理型パラダイムと同様，ルールの平板な集合で知識を表現するため，大規模な知識ベースを作るには構造化・階層化の手段が必要となる．そして，その手段の最善の候補がやはりオブジェクト指向パラダイムとの融合であることは容易に想像がつく（KEE のようにフレームによる方法もあるが，オブジェクト指向と言えないことはない）．標語的に言えば，ルール指向とオブジェクト指向の融合により知識の分散化ができ，結果として処理の効率化が図れる．

融合の基本的な考え方は，論理型パラダイムの場合とそう違わないが，論理型と異なり理論的な整合性うんぬんに頭を悩ます必要はあまりない．ワーキングメモリがインスタンスに固有の内部状態に対応する．ルールのインヘリタンスは，ホーン節のインヘリタンスと同じくらい単純である．しかし，実際にはベースとなった言語の性格などにより，オブジェクト指向らしさの度合いはシステムによって少しずつ異なっている（LOOPS[10]，OPHELIA[11]，PANDRA など）．

3.7 並列計算パラダイムとの融合

並列計算がパラダイムと言えるものかどうかについては自信がないが，オブジェクト指向パラダイムが並列計算の研究を新たに活性化したことは事実だろう．オブジェクトが「動作主体」と見なせることから，オブジェクトにプロセスが付随していると割り切ってしまうことができる．これが並列計算パラダイムとオブジェクト指向の融合の原点である（もっとも，これはアクター理論の基本原理であって，昨今のオブジェクト指向の流行よりずっと昔からあった思想である．）

日本では ABCL[12]，Orient 84/K[13] などがこの方面の研究をリードしている．しかし，解決しないといけない基本問題がまだまだ山積しているというのが現状だろう．たとえば，並列オブジェクト間のメッセージ伝達の同期，伝播，ロールバック，相互排除，割り込み通信など．これらの問題の多くはオブジェクト間の交信を相互のメッセージ伝達だけに制限することによって生ずるものである．この制限が現実世界のモデル化として妥当かどうかは少し議論のあるところだろう．たとえば，現実世界で動いている動作主体には共通の「場」（同期信号も場の一種）という概念があるが，これをメッセージ伝達だけの枠組で表現すること（できないことはないだろうが）の是非が問題になる．

並列計算とオブジェクト指向の融合でいつも問題になるのが，システム全体の中でのおのおののオブジェクトの粒度とオブジェクト間の結合の疎密度の選択である．これは理論的にはほとんど意味をもたないが，プログラミングや実行性能などの実際面ではかなり決定的な意味をもつ．ルール指向との融合で知識の分散化と処理の効率化が得られるのは，一つ一つのルールベースが比較的大きな動作主体だからである．あまり細かな粒度にして結合を密にすると，モジュール化のメリットが「ネットワークコスト」に埋没してしまう．逆にあまり大きな粒度にして結合を疎にすると，並列化で得られるメリットが目立たなくなる．しかし，見方によれば，オブジェクトの中の動作は集中型パラダイム，オブジェクトの集合の動作は並列型パラダイムと言えるわけで，ある程度大きい粒度のオブジェクトを使うことは二つのパラダイムのいい融合形態かもしれない．

4. おわりに

オブジェクト指向ほど広範囲なパラダイムと融合したパラダイムはほかにない．それは上にみてきたように，オブジェクト指向の構造化・階層化能力，ほかの計算メカニズムや抽象化機能との直交性，情報隠蔽能力，ほかのパラダイムに対する同化能力などプログラミングパラダイムとしての概念的な一般性に大きくよっている（筆者はオブジェクト指向の「再利用性」は現実にはまだあまり機能していないと考えている）．1970年代のパラダイムが構造化プログラミングであるとすれば，オブジェクト指向は1980年代のパラダイムである．構造化プログラミングが1980年代に入ってそうなったように，1990年代に入ればオブジェクト指向は突発性ブームのような流行を終えてごくふつうに言語やシステムに取り入れられていく（取り入れられてしまっている）ことになるだろう．

しかし，現時点でオブジェクト指向とほかのパラダイムの融合の可能性が尽きたとはとても思えない．筆者にはまだよく分からないが，関数型パラダイムにもオブジェクト指向の考え方が取り入れられ始めたという．このほかにもいろいろな新しいパラダイム（たとえば reflection）がオブジェクト指向と融合しそうな気配があるらしい．また，すでに確立されたかに思える融合形態にも大きな見直しが起こるかもしれない．要するに，われわれはパラダイムの融合に関するいわばメタパラダイムをまだはっきり認識できる段階に至っていない．今後，こういうメタパラダイムの理論的または概念的な整備が望まれるところである．

本稿では融合にまつわって起こる効率の低下の問題には言及しなかったがこれも実際的には重要な問題である．逆に効率の問題にこだわりすぎて融合の形が歪んでしまっているような言語やシステムが少なからずある．これをこのままよしとするか，さらなる技術的な進歩を目指して実用に待ったをかけるか，これも悩ましい問題である．

最後に浅学の筆者にいろいろご教示いただいた同僚の尾内理紀夫，梅村恭司，増尾剛の三氏に感謝する．

参考文献

1) Floyd, R. W.: The Paradigms of Programming (1978 ACM Turing Award Lecture), Comm. of ACM, Vol. 22, No. 8 (1979).
2) Goldberg, A. and Robson, D.: Smalltalk-80: The Language and Its Implementation, Addison-Wesley (1983).
3) Bobrow, D. G. et al.: Common Lisp Object System Specification (1986).
4) Chikayama, T.: Unique Features of ESP, Proc. of the International Conference on Fifth Genreation Computer Systems (1984).
5) Nakashima, N.: Prolog/KR User's Manual, METR 82-4, Univ. of Tokyo (1982).
6) Shapiro, E. and Takeuchi, A.: Object Oriented Programing in Concurrent Prolog, New Generation Computing, Vol. 1, No. 1 (1983).
7) Kahn, K. et al.: Vulcan: Logical Concurrent Objects (1986).
8) 米澤明憲：Actor 理論について，情報処理，Vol. 20, No. 7 (1979).
9) Takeuchi, I., Okuno, H. G. and Ohsato, N.: A List Processing Language TAO with Multiple Programming Paradigms, New Generation Computing, Vol. 4, No. 4 (1986).
10) Bobrow, D. G. and Stefik, M.: The LOOPS Manual, Xerox PARC (1983).
11) 安西祐一郎，近藤公久：階層構造を持つルールベース型表現を埋めこんだオブジェクト指向言語 OPHELIA とその応用，コンピュータソフトウェア，Vol. 3, No. 3 (1986).
12) Yonezawa, A. et al.: Modelling and Programming in an Object-Oriented Concurrent Language ABCL/I, Object Oriented Concurrent Programming (ed. by Yonezawa and Tokoro), MIT Press (1987).
13) Ishikawa, Y. and Tokoro, M.: Orient 84/K: An Object-Oriented Concurrent Programming Language for Knowledge Systems, Object Oriented Concurrent Programming (ed. by Yonezawa and Tokoro), MIT Press (1987).

(昭和 63 年 1 月 26 日受付)