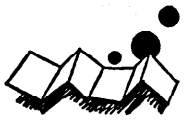
解　説



型階層をもつオブジェクト指向言語†

久　野　　靖††

1. はじめに

プログラム言語における型の概念は，もともとはFortranなどにみられるように，整数，実数など性質の異なるデータを使い分けることから始まっている．その後 Pascal などの普及を通じて，型宣言によってプログラムに一見冗長に思える情報を含めておくことでプログラムの読みやすさを増し，またデータ使用の不整合などの誤りをコンパイル時に検出することの利点が広く理解されるようになった．最近ではさらに進んで，データの外部から利用できる性質とその具体的な表現とを分離するデータ抽象の考え方が一般化している．

一方，オブジェクト指向言語の側からみると，クラスはそれに属する実体の性質を規定するという点では型と同じであるし，外部からの不当なアクセスを防止するという点で抽象データ型の機能も備えているといえる．しかし，オブジェクト指向言語の初期における代表である Smalltalk-80 や Flavors などは変数や引数に対する型宣言機構をもたず，翻訳時の型検査も行っていない．これは，プログラムをメソッド単位で対話的に開発し，また継承機構によって一つのコードを複数のクラスで共有するという前提のもとでは妥当な選択であったと考えられる．

しかしながら，オブジェクト指向パラダイムの有効性が広く認識されるにつれ，従来の手続き型言語を拡張してオブジェクト指向機能を導入し，大規模なソフトウェア開発に使用することが試みられるようになってきた．そして，そのような状況化では型検査機構による翻訳時の誤り発見を行いたいというのは自然な要求である．

本解説では上記のような視点のもとに，以下 **2.** では従来の手続き型言語の流れをくみ，抽象データ機能をもつような言語におけるオブジェクト指向パラダイム適用の可能性と限界について解説し，**3.** で強い型をもつ言語の発展であり，代数的仕様記述に関する研究の成果を取り込んだ，型階層をもつ言語について説明する．続いて **4.** でこれらの流れとオブジェクト指向言語の流れをともに引き継いだ，型階層をもつオブジェクト指向言語について解説し，最後に **5.** でまとめを行う．紙面の制約から，各章を通じて具体的な言語の例としては適切と思われる2，3をあげるにとどめ，あとは参考文献に記した．

2. データ抽象機能とオブジェクト指向

一般にデータ抽象機能とは，型定義とその型に対する操作を一つのモジュール内にまとめ，モジュール外からその型のデータに対してできることは操作を呼び出すことのみに限定するような機構をいう．ここでデータ抽象機能によって外部から保護された型を抽象型，モジュール内で定義されている実際の構造をその内部表現と呼ぶ．データ抽象機能を使用することの利点としては，次のものがあげられる．

a．抽象型の内部表現を扱うのはモジュール内の操作に限られるため，内部表現の整合性を保証しやすい．

b．内部表現を変更してもその影響がモジュール内に限定されるため，機能の変更・拡張が容易である．

c．モジュールと外部との界面は操作呼び出しのみであるため，モジュール間の独立性が高く，部品としての再利用も容易である．

データ抽象機能における「抽象型」を「クラス」，「操作」を「(クラスまたは実体の) メソッド」と呼び換えてみると，オブジェクト指向言語との類似性があることが分かる．以下本章ではデータ抽象機能をもつ言語の具体例として Modula-2, Ada, CLU を取り上げてこれらの言語とオブジェクト指向パラダイムの関連について解説し，最後にこれらの言語の限界についてまとめる．

† Object-Oriented Programming Languages with Type Hierarchy by Yasushi KUNO (Dept. of Information Science, Tokyo Institute Technology).

†† 東京工業大学理学部

2.1 Modula-2

Modula-2 は N. Wirth により，Pascal の後継言語として開発された．Modula-2 ではモジュールは定義部と実現部に分けて記述し，外部からモジュールを使用する場合には，定義部に記述された情報だけが利用できる．例として次のような定義部を考えてみる．

```
DEFINITION MODULE M;
  TYPE T;
  PROCEDURE New ( ): T;
  PROCEDURE DoIt (x : T);
END M;
```

ここで型Tは隠蔽型と呼ばれるもので，その内部構造は定義部には現れず，実現部で定義される．Mの外部からこの型を使う場合には

```
VAR x: M. T;
...
x := M. New ( );
M. DoIt (x);
```

のように記す．このように Modula-2 でもオブジェクト指向ふうプログラミングは可能であるが，ただし型名・操作名ともにモジュール名を前置する必要がある（名前の衝突がない範囲で前置を省略できる機構もある）．

2.2 Ada

Ada は米国防総省が組み込みシステム記述用に開発した言語であり，モジュール化や並行処理など多くの機能をもつ．Ada においても Modula-2 の場合と同様，原則としては型名や操作名にモジュール名を前置する．ただし Ada は同じ名前の操作が複数あって曖昧さが生じても，引数の型から正しい操作を見つけ出す(オーバローディング)機能をもつので，これを利用してモジュール名の前置を省略することもできる．

Ada でクラスに相当するモジュールを作るもう一つの方法はタスク型を使用することである．同じ例をタスク型を使用して書き直したものを次に示す．

```
task type T is
  entry New;
  entry DoIt;
end T;
task body T is
  変数定義
begin
  loop
    select
      accept New do…end New;
    or
      accept DoIt do…end DoIt;
    end select;
  end loop;
end T;
```

すなわち，タスクの場合には操作はエントリに対応することになる．これを利用する側では

```
x: T;
...
x. New;
x. DoSomething;
```

のように自然な形で書くことができる．タスク型は型であるのでタスク型の要素を代入したり引数として渡すことに問題はない．また，各タスクは並行に動作するので，タスクを利用してオブジェクトを実現した場合には，各オブジェクトを並行して動作させることも可能である．一方，並行動作を必要としない場合にはこの方法だとタスク切り替えにともなうオーバヘッドが問題になる可能性もある．

2.3 CLU

CLU[8] は 1970 年代に MIT で開発された言語であり，データ抽象機能を素直な形で取り入れ，しかもコンパクトな言語仕様となっていることが特徴である．また，CLU のモジュール（クラスタと呼ぶ）はそれぞれが一つの型を定義する．したがって前記の例を CLU で記述した場合，モジュールを利用する側では

```
x: T := T $ New( )
T $ DoSomething(x)
```

のように一つの名前だけを扱えばよい．

このように CLU ではモジュール名と型名が一致するところが好ましい点であるが，Modula-2 などと同様呼び出しの際に常にこの名前を前置する必要がある．筆者の手元ではこの点を改良するため構文を拡張して

```
TypeOf (x) $ OpName (x, …)
```

の形の呼び出しを

```
x ! Opname (…)
```

のように書くことを許す処理系[10]が作成され使用されている．この記法によれば上の例は

```
x ! DoSomething ( )
```

のように書くことができる．

2.4 抽象データ型言語とオブジェクト指向

これまでにみてきたように，抽象データ型言語にお

いても「データとその操作を一体化して記述する」という意味でオブジェクト指向の考えを自然な形で取り入れることは十分可能である．ただし，その場合に問題となる事柄として，これまでにみてきたものを含めて次のようなものがあげられる．

a．記法的な問題．これらの言語では通常，複数のモジュールで同じ名前の操作を作ることは許しても，それらの前にモジュール名などを前置する必要があり，わずらわしい．ただし Ada のオーバローディングや CLU の構文拡張のようにコンパイラで型情報を利用してこの問題をなくすことはできる．

b．型の階層化．これらの言語では型どうしにはなんの関連もなく，一つの型を修正して別の型を定義するようなことは考えられていない．

c．動的な対象の切り替え．これらの言語では変数が厳密な型をもつため，一つの変数に場合によってさまざまなクラスの実体を入れて扱うようなことは原理的に不可能である．

これらのうち，a．については単に言語設計上の問題であるが，b．および c．は強い型の思想そのものと関わる根本的な問題であるといえる．次章ではこれらのうち特に b．に着目して，型階層をもつ言語について解説する．

3. 型階層をもつプログラム言語

前章で述べたように，Ada や CLU などの抽象データ型言語では型同士は基本的に独立であり，ある型を拡張して別の型を作るようなことはできない．しかし（オブジェクト指向言語のクラスのように）ある型をもとにして別の型を作ることができれば，プログラムの構造化や再利用の面で利点が大きいことは明らかである．

さらに，型の継承の概念はプログラムの検証の面からも重要である．たとえば検証においては，型Tの実体がもつ基本的性質（公理）からさまざまな関係（定理）を導いていく．ここでTと類似した公理をもつ型T′について，重複した公理を保存し，重複した定理を毎回導いていたのでは記憶容量・計算時間の両面から無駄が大きい．型を階層化し，複数の型に共通する公理を1か所にまとめることによりこのような無駄をなくすことができる[1]．

実際には本章で述べるような型階層はプログラム言語そのものとしてよりは検証のための仕様記述言語に多くみられるが[5]，ここではそれらについては立ち入らない．以下本章では型階層を取り入れた言語の例として IOTA および OBJ2 について解説し，これらの言語の提供する機構とオブジェクト指向の関連について論じる．

3.1 IOTA

IOTA[9] はモジュールによる抽象化とそれに基づくプログラム検証をサポートするプログラム言語およびプログラミングシステムである．ここでは IOTA 言語の型階層化機能について主に解説する．

IOTA 言語は TYPE, SYPE, PROCEDURE の3通りのモジュールをもつ．これらはそれぞれ具体的な型，抽象的な型 (Smalltalk-80 でいえば抽象クラス)，独立の手続きを定義する．各モジュールはさらに操作の界面（インタフェース）を定める界面部，その公理を定める仕様部，実現を規定する実現部からなる．ただし SYPE や組み込みの TYPE には実現部はない．例として自然数を表す型 nn の記述を示す．

```
INTERFACE TYPE nn
  FN zero : -> @ AS 0
    suc : @ -> @
    less : (@, @) -> bool AS @ =< @
END INTERFACE
SPECIFICATION TYPE nn
  VAR x, y, z : @
AXIOM 1 : suc(x) = suc(y) => x = y
      2 : suc(x) =< x
      3 : x =< y => suc(x) =< suc(y)
      4 : x =< y V y =< x
      5 : x =< y & y =< x => x = y
      6 : x =< y & y =< z => x =< z
END SPECIFICATION
```

すなわち界面部では型 nn に対する操作として0，suc，=< があることを宣言し，仕様部ではこれらに関する公理を与えている（この中には等号＝に関するものがないが，＝に関する界面と公理は自動的に含められる）．

上記の型 nn の記述には全順序に関するものが含まれているが，次のように「全順序をもつようなもの」という sype を別に用意しておいてこれを継承するようにもできる．

```
INTERFACE SYPE order
  FN less : (@, @) -> bool AS @ =< @
END INTERFACE
SPECIFICATION SYPE order
```

```
  VAR x, y, z : @
  AXIOM 1 : x =< y V y =< x
        2 : x =< y & y =< z => x =< z
        3 : x =< y & y =< x => x = y
END SPECIFACATION
```

これを継承する場合，nn の記述は

```
INTERFACE TYPE nn
  IS order
  FN zero : -> @ AS 0
     suc : @ -> @
END INTERFACE
SPECIFICATION TYPE nn
  VAR x, y : @
  AXIOM 1 : suc(x) = suc(y) -> x = y
        2 : suc(x) =< x
        3 : x =< y => suc(x) =< suc(y)
END SPECIFICATION
```

のように書ける．また，nn 以外にも順序をもつ型は多数考えられるが，これらも order を継承すればよい．このようにすることでモジュールの記述量を減らすことができるだけでなく，順序に関する定理の証明を順序をもつ型ごとに毎回行う代わりに1回で済ませることができる．

これらに加えて IOTA ではモジュールのパラメータ化も sype の重要な役割である．たとえば型 t の配列 array (t) を扱うモジュール arraysort を考えてみる．

```
INTERFACE PROCEDURE arraysort (t : order)
  FN sort : array(t) -> array(t)
    sorted : array(t) -> bool
    locate : (array(t), t) -> (bool, nn)
END INTERFACE
```

この界面部はモジュール arraysort において配列を整列する sort，整列済みかどうかを調べる sorted，整列済みの配列上で探索を行う locate の三つの操作をもつことおよびこれらの操作の引数，戻り値の型を記述している（実現部については略した）．ところで，整列するためには要素型 t が全順序をもたなければならないが，このため1行目で t が order の子孫（サブソート）であることを宣言している．これによって正しくない型について arraysort を適用することを防ぐとともに，arraysort の実現が正しいことを検証する際に要素型 t について order の公理を使用することができる．

3.2 OBJ 2

OBJ 2[11] は一連の代数的意味記述の試みの流れをくむプログラム言語であり，項書き換えをその実行原理としている．ここでは OBJ 2 の型階層化機能に焦点を絞って解説する．

OBJ 2 ではモジュールには object と theory の2種類があり，前者は実行可能なコードを含み，後者はアサーションのみを含む．ただしモジュールそのものが型に対応するのではなく，一つのモジュールで複数の型（sort）を定義することも許す．また object モジュールが操作定義のみを含む場合もある．たとえば「半順序をもつもの」に対応する theory は次のように書ける．

```
th POSET is
  protecting BOOL.
  sort Elt.
  op_<_ : Elt Elt -> Bool.
  vars E E' E" : ELt.
  eq : E<E=false.
  ceq : E<E'=true
        if (E<E" and E"<E').
endth
```

POSET は theory であるので，実際のオブジェクトは作り出さないが，他のモジュールがそれを引用している場合に，そこにオブジェクトを生成するような型，たとえば Nat（自然数）や Rat（有理数）などをはめ込んで具体化することができる．Smalltalk-80 の言葉でいえば Elt は抽象クラスに，Nat と Rat はその下の具体的な実体を表すクラスに対応する．

上の例とは逆に，オブジェクトの下にその操作を拡張した theory を作ることもできる．たとえば次の例では自然数から「上限をもつ自然数」を作り出している．

```
th NAT*is
  protecting NAT.
  op bound : -> Nat.
endth
```

これまでの例はすべて公理の上の包含関係に関するものであったが，OBJ 2 ではオブジェクト間に直接 subsort 関係を規定し，操作を継承させることもできる．たとえば有理数は整数を含み，整数は自然数を含むので

Nat<Int<Rat

である（<は subsort 関係を表す）．また

horse＜vehicle
horse＜animal

のように多重継承とさせることもできる．

3.3 型階層をもつ言語とオブジェクト指向

本章で述べた型階層化機能において重要なことは，これらにおいて継承とは型の仕様（IOTA の用語でいえば界面），つまり型がもつ操作の集合および各操作の引数や戻り値の型の継承を意味し，型の内部実現についてはそれぞれ独立なものとして扱う点である．これは内部実現（実体変数と操作のコード）を継承し，その結果外部仕様も類似したものとなる Smalltalk-80 や Flavors とは一線を画しているが，抽象データ型の思想からみればきわめて当然のことであるといえる（ただし，型階層をもつ言語でも必要な場合には実現などを共有する機構をもつことは自然であり，実際 IOTA はそのような機構をもつ）．

一方，これらの言語においても **2.4** で述べた問題点 c．については解消されていない（モジュールのパラメータ化によって一つの字面のコードが複数の型に対して使用されることはあるが，これらはあくまでも静的なものであり，動的に型が切り替わるわけではない）．

しかし，動的に対象の型が変化できることはオブジェクト指向パラダイムを導入する上で非常に重要である．次章で述べる型をもつオブジェクト指向言語群はこの点で本章で述べた言語からさらに一歩踏み出したものとなっている．一方，仕様と実現の分離についてはこれらの言語は本章で述べた言語と基本的に一致している．

4. 型階層をもつオブジェクト指向言語

前章までにみてきたように，従来の型階層をもつ言語の主要な目的は型の仕様や公理の継承にあり，したがって一つの変数が動的に複数のクラスの実体をもつような多態はサポートされていなかった．しかし，Smalltalk-80 をはじめとするオブジェクト指向言語の成功につれて，型のないオブジェクト指向言語の主要な特徴は残しながら強い型と型階層の概念も取り込んだ言語を作ることにより，双方の利点を取り入れようとする試みが多くみられるようになった．本章ではそのような言語について解説する．

4.1 Trellis/Owl

Trellis/Owl[11] は DigitalEquipment 社で開発されているオブジェクト指向プログラミング環境 Trellis のために開発された．Trellis/Owl では基本的に子供の型の実体は親の型としても振る舞える，という立場を取る．たとえば型 Text_Window（文字を読み書きする窓）が Window と IO_Stream という二つの親をもつ場合，Text_Window の実体は Window や IO_Stream としても扱うことができる．すなわち次のようなことが可能である．

```
var tw : Text_Window ;
var io : IO_Stream ;
…
io := tw ;
read(io)  ! Text_Window の read を呼ぶ
```

このように，「親子関係にあれば同じ型でなくとも代入できる」というのは型をもつオブジェクト指向言語の多くにみられる方式であるが，強い型の精神からみれば大きな変革であるといえる．

一方，子供の型の実体は親の型としても振る舞えることから操作の入出力関係に制約が課せられる．たとえば型 s，t で s＜t（＜は「子供である」の意）かつ同じ名前の操作Fをもつとき，たとえばその仕様が

```
T -- F(me, Tin1) returns (Tout1)
S -- F(me, Tin2) returns (Tout2)
```

だったとすると，Tin1 ⟨= Tin2 および Tout2 ⟨=Tout1 である必要がある．これは

```
var vt : t ;
var vs : s ;
…
vt := vs ;
…F(t, x)… ! S のFを呼ぶ
```

のような場合，SのFは最低限TのFと同じものを受け取れる必要があり，また返すものはせいぜいTのFと同じものまでであると考えれば納得できる．

また，Trellis/Owl では親子関係にあることは外部から見える振る舞いが似ていることは意味するが，その内部実現は全く異なっていてよい．このように階層化が実現ではなく型の仕様による点は型をもつオブジェクト指向言語に多くみられ，Smalltalk-80 などにみられる「親子関係に置くことで望まないのに同じ実現を強いられる」という問題に対する解答となっている．ただし，実際上は親と同じ内部構造・操作実現を使用する場合も多いので，これらの定義を親から継承してくることが標準となっている．

4.2 Emerald

Emerald[2] はワシントン大学で開発された，分散シ

ステム向けオブジェクト指向言語である．ここではその分散機能などについては触れず，型階層機能のみについて解説する．Emerald の特徴は型同士の親子関係を（他の多くの言語のように）明示的に宣言することによって決めるのではなく，互換性 (conformity) があるかどうかによってのみ定まるとしたところにある．

具体的には，S<= (conforms to, の意) Tとは

- Sは少なくともTのもつ操作すべてをもつ．
- 各操作の引数/戻り値の個数はそれぞれ等しい．
- Sの操作の各戻り値の型<=対応するTの操作の戻り値の型である．
- Tの操作の各引数の型<=対応するSの操作の引数の型である．

のように定義される．したがって，たまたま二つの型の操作に互換性があれば，プログラマが意図しなくともそれらの型は親子関係をもつことになる．

一方，親子関係にあれば子供の実体を親の実体の代わりに用いることになんら問題はない．そこで Emerald では Trellis/Owl と同様，

```
var x : T
```

のように宣言した変数に対して S<= T であるような任意の型Sの実体を代入して使用できる．操作呼び出しの実現は，「S型の実体をT型の実体として使う場合の操作呼び出しベクトル」を [S, T] の組ごとに on demand で（初めてそのような代入が起きた時点で）動的に作成することにより効率よく行える．

4.3 Misty

Misty[7] は筆者が提案した，CLU をもとに拡張した型階層をもつオブジェクト指向言語である．ここまでに述べてきた言語の多くと同様，Misty においてもモジュールは仕様部と実現部に分けて記述する．たとえば型 bool の仕様部が

```
bool=class spec
    defines and, or, not
  and=proc(x, y : self) returns(self)
  or=proc(x, y : self) returns(self)
  not=proc(x : self) returns(self)
end bool
```

のようなものだったとして，これを継承してさらに操作 xor を追加した型 xbool の仕様部は次のように書ける．

```
xbool=class spec
    super bool
    defines xor
  xor=proc(x, y : self) returns(self)
end xbool
```

この場合，xbool は操作として and, or, not, xor の4つをもつことになる．このように Misty においても継承は型の外部仕様の継承を意味し，内部実現はこれとは独立に定められる（内部実現を親から借りてくることも可能である）．

Misty では Trellis/Owl, Emerald などの言語と異なり，親の型の変数に子供の型の実体を代入することを許していない．このためより厳密な型検査を行うことができるが，一方で動的な対象の切り替えを行うために別の機構を用意する必要があった．Misty ではこれを型生成子 any を通じて行う．any の外部仕様を次に示す．

```
any=class spec[t : object]
    super t
    defines relax, force, is_type, isa_type
  relax=proc[t1 : t](x : t1) returns(self)
  force=proc[t1 : t] (x : self) returns(t1)
  is_type=proc[t1 : t] (x : self) returns(bool)
  isa_type=proc[t1 : t](x : self) returns(bool)
end any
```

すなわち，型 any [t] (ただし型 t は object の子孫) は型 t のすべての操作に加えて relax, force などの操作生成子をもつ．any [t] の役割は t の子孫の型の実体であればどれでもまとめて扱い，それらを動的に切り替えることである．

たとえば horse, car がともに型 vehicle の子孫のとき，

```
x : any[vehicle]
x := any[vehicle]!relax[horse](horseの実体)
...
x := any[vehicle]!relax[car](carの実体)
```

のようにして car あるいは horse の実体を変数xに入れることができる．また，

```
x ! turn_right( )
```

のように操作を呼び出した場合には現在xに入っている実体が何であるかによって car または horse の操作が呼び出される．

このように，Misty では動的に対象の型を切り替える必要がある場合には any を使用し，それ以外の場所では呼び出される操作は厳密に定まる．このように動的な切り替えを明示させることによって意図しない型の誤りがより多く検出でき，また動的な切り替えを

必要としない呼び出しを高速化することができる.

4.4 型をもつオブジェクト指向言語の比較

ここで述べたほかにも型をもつオブジェクト指向言語はいくつか提案されているが[6),13),17)], これらすべてに共通する点は先にも述べたように

●仕様の継承と内部実現の継承を区別して扱っているという点にある. この区別の必要性については文献[15)]などに論じられている. また, 研究者によっては継承 (inheritance) という言葉を後者の意味のみに使用し, 前者の意味では副型 (subtyping) という言葉を当てているものもある.

一方, 言語間で差異のある点としては

●仕様の継承を明示する/しない

●子孫の型を親の型と互換とする/しない

の2点があげられる. これらの点についてはいずれも「する」方が多数派であるが, 「しない」側にもそれなりの言い分がある. たとえば Emerald は仕様の継承を明示せず, 操作の互換性に基づいて自動的に親子関係を定めるがこれにより「子で削除したために親にあって子にない操作がある」といった不整合を回避している. また, Misty では原則として同じ型以外の代入を認めず, 動的に型が切り替わる必要があるところはプログラマに明示させることによってより厳密な型検査を行っている.

これらの点のほかに本文では紙面の制約から省いたが, 一つの仕様に対して複数の実現を提供する機能をもつ言語も多く, さらに実現の選択を自動化するといった試みもある[13)]. また, オブジェクト指向言語に関連する話題として分散/並列実行があるが, 型階層機構とこれらの関連についての研究も盛んである[2),6)]. これらの点については直接文献を参照されたい.

5. ま と め

ここまででみてきたように, 型をもつオブジェクト指向言語は文字どおり単にオブジェクト指向言語に型宣言を追加したもの, という見方をすることもできるが, 一方で強い型検査, 抽象データ型, 型階層の導入という一連の流れの延長としてオブジェクト指向の柔軟性を取り込んだものと考えることもできる.

元来強い型検査の厳格さとオブジェクト指向の柔軟性はある程度相反する面をもち, そのため型をもつオブジェクト指向言語についてはどの程度どちらの側面を重視するかによってさまざまなものが考えられる. また, この分野はまだ活発な研究が進められている段階にあり, 今後とも新しいメカニズムの提案や新言語の実施使用に基づく評価が進められていくと予想される.

参 考 文 献

1) 稲垣, 坂部: 多ソート代数と等式論理, 情報処理, Vol. 25, No. 1, pp. 47-53 (1984).
2) Black A. et al.: Object Structure in the Emerald System, Proc. OOPSLA'86, pp. 78-85.
3) Futatsugi, K.: Simple Examples of Parameterized Programming in OBJ 2, プログラムの合成/変換/再利用シンポジウム報告集, pp. 55-61 (1985).
4) Futatsugi, K. et al.: Principles of OBJ 2, Proc. POPL '85, pp. 52-66 (1985).
5) Guttag, J. et al.: The Larch Famirly of Specification Languages, IEEE Software, Vol. 4, No. 5, pp. 24-36 (1985).
6) Kristensen, B. et al.: Abstraction Mechanisms in the Beta Programming Language, Proc. POPL '83, pp. 285-298 (1983).
7) 久野: 多重継承と強い型付けを持つ言語 Misty, ソフトウェア科学会第1回大会論文集, pp. 73-76 (1984).
8) Liskov, B. et al.: CLU Reference Manual, Lecture Notes in Compt. Sci. 114, 190 p., Springer (1981).
9) Nakajima, R. and Yuasa, T. eds.: The IOTA Programming System, Lecture Notes in Compt. Sci. 160, 217 p., Springer (1983).
10) 歴本: CLU 言語の構文の拡張とその処理系, 個人書簡 (1984).
11) Schaffert, C. et al.: An Introduction to Trellis/Owl, Proc. OOPSLA '86, pp. 9-16 (1986).
12) Seidewitz, E.: Object-Oriented Programming in Smalltalk and Ada, Proc. OOPSLA '87, pp. 202-213 (1987).
13) Sherman, M.: Paragon: Novel Uses of Type Hierarchies for Data Abstraction, Proc. POPL '83, pp. 208-217 (1983).
14) Sherman, M.: Paragon: A Language Using Type Hierarchies for the Specification, Implementation and Selection of Abstract Data Types, Springer, 355 p. (1982).
15) Snyder, A.: Encapsulation and Inheritance in Object-Oriented Programming Languages, OOPSLA '86, pp. 38-45.
16) Touati, H.: Is Ada an Object Oriented Programming Language ?, SIGPLAN Vol. 22, No. 5, pp. 23-26 (1987).
17) Thorelli, L-A.: Modules and Type Checking in PL/LL, Proc. OOPSLA '87, pp. 268-276 (1987).

(昭和62年11月12日受付)