■6 群（基礎理論とハードウェア）- 3 編（アルゴリズムとデータ構造）

4 章 抽象データ型とオブジェクト指向

（執筆者：久野　靖）[2012 年 7 月 受領]

■概要■

近年におけるソフトウェアの巨大化・複雑化の結果，今日のソフトウェアは直接アルゴリズムとデータ構造に基づいて構築されるというよりは，多数の関連して動作し合うモジュールの集合体として構築されるようになった．そしてそれぞれのモジュールは，自身が外部に対して公開するインタフェースの機能を実装することを役割とし，その実装のための手段として他のモジュールが公開している機能を利用する，というかたちで互いにかかわるようになっている．

このようなモジュール間のインタフェースを定式化する考え方が抽象データ型であり，抽象データ型が提供する（外部に公開する）機能は代数的仕様記述のかたちで厳密に記述することが可能である．抽象データ型に基づく言語は今日では主流ではないが，今日主流であるオブジェクト指向言語においても，抽象データ型を実装することが，その主たる利用形態の一つであり続けている．

オブジェクト指向は，抽象データ型に動的分配やクラス間の継承などのツールを追加したものと考えることができる．これによって，多くの場面においてよりコンパクトにコードが記述できたり，人間にとってより理解しやすいコードが書けるようになっている．

また，C++のテンプレートに代表されるジェネリック（汎用的）プログラミングのための機構も，従来的なオブジェクト指向とはまた別の切り口でコードを構造化し，ライブラリなどのかたちでコードを再利用するための有力なサポートを提供するようになっている．

【本章の構成】

4-1 節では，データ構造を構造そのものとしてとらえる代わりに，それがもつ演算に基づいて定義する抽象データ型の考え方と具体例を紹介する．続いて 4-2 節ではオブジェクト指向の考え方を紹介し，今日一般的になっているオブジェクト指向言語の機能がデータ型やデータ構造に対してどのような機能を提供しているかを解説する．更に 4-3 節ではジェネリックプログラミングの概念と，それが実現するデータ型やデータ構造の機能とについて解説する．

■6 群 - 3 編 - 4 章

4-1 抽象データ型と代数的仕様記述

（執筆者：神林　靖）[2012 年 7 月 受領]

4-1-1 抽象データ型とプログラミング言語

抽象データ型（abstract data type）とは，一つ以上のデータオブジェクトの集合とその集合上に定義された一つ以上の演算，そして演算を記述する若干の公理から構成される代数である．

抽象データ型によってデータオブジェクトと演算がどのように実装されているかを気にせずにプログラミングすることができる．例えば多項式を処理するための抽象データ型を生成しなければならないとしよう．二つの多項式 p と q の和を表す式 add(p, q) を使用するのに異論はないであろう．この抽象データ型を実装するために，多項式を係数の配列で表現したうえで，加算は対応する配列の要素を加えることで実装することができる．もちろんこのほかにも多項式とその加算を表現する有効で有用な方法はある．しかしどのような実装方法を用いるにしても，文 add(p, q) は同じことを意味している．

1970 年代になると，このような実装の詳細を抽象化する考え方が広まり，抽象データ型をプログラミング言語に取り入れることに対して活発な研究が行われるようになった．CLU[3] や Alphard[4]などの言語の提案は，その成果である．とりわけ MIT で Liskov らが開発した CLU はほかの大学などでも教育に使用された実績がある．[2]

```
boolean = cluster is zero, one, add, not, mul, equal, b2i
  rep = int
    zero = proc() returns(cvt) return 0 end zero
    one = proc() returns(cvt) reutrn 1 end one
    add = proc(x:cvt, y:cvt) returns(cvt) return int$min(1, x+y) end add
    mul = proc(x:cvt, y:cvt) returns(cvt) return x*y end mul
    not = proc(x:cvt) returns(cvt) return 1-x end not
    equal = proc(x:cvt, y:cvt) returns(bool) return x = y end equal
    b2i = proc(x:cvt) returns(int) return x end b2i
end boolean
```

図 4･1　CLU **における抽象データ型**

図 4･1 に CLU でブール代数の抽象データ型を記述した例を示す．CLU では `cluster` が抽象データ型を定義するモジュールであり，その内部で定義される `rep` という型が，その抽象データ型の内部表現型となる．ここでは整数の 0 と 1 を内部表現に使うので，`rep = int` としている．

`cluster` が提供する操作のシグネチャで `cvt` と指定された型は，`cluster` の内部では `rep` 型，外部からはこの `cluster` が定義している抽象データ型（この場合は `boolean`）を表すことを意味する（`cvt` は convert つまり変換を意味するが，実際には型が読み替えられるだけであり，値はそのまま渡される）．

ここでは cluster の手続きとして，0 と 1 の値をつくり出す，加算，乗算，否定，相等，そして整数への変換（例えば出力などのため），の各操作を定義している．boolean 型の内部表現にアクセスできるのは cluster の手続きだけであり，内部表現の値が 0 と 1 しかないことは保証できるので，それを前提として操作を実装することができる．

4-1-2 代数的仕様と具体例

前述のように，抽象データ型では，その「外部から観察される振る舞い」と「内部での実現方法」が独立している．そこでこれを利用して，外部から観察される振る舞いを代数的に記述し，(a) 内部実現がそれを正しく実装していることの証明を与え，(b) このデータ型を使用している箇所の正しさを代数的仕様に基づいて証明することで，プログラムの正しさを分割して検証することが可能になる．このような代数的仕様をサポートする言語としては Clear, Obj などが挙げられる（Clear/Obj → 7 群 1 編 1 章）．

以下では，まず自然数を例にとって代数的仕様の考え方を説明し，次節以降で代表的なデータ型に対する代数的仕様を紹介する．

自然数は，次のように帰納的に定義するのが自然である．

1. $0 \in \mathbb{N}$
2. 「後者（successor）」と呼ばれる，次の性質をもつ関数 $s : \mathbb{N} \to \mathbb{N}$ が存在する．

   $x \in \mathbb{N}$ であれば，$s(x) \in \mathbb{N}$
3. すべての $x \in \mathbb{N}$ について，$s(x) \neq 0$
4. $s(x) = s(y)$ であれば，$x = y$

これをもとにすると，自然数の代数的記述は，次のように定められる．

台: $\mathbb{N}$

演算: $0 \in \mathbb{N}$
$s : \mathbb{N} \to \mathbb{N}$

公理: $s(x) \neq 0$
$s(x) = s(y)$ であれば $x = y$

型上の有用な演算を基本的な演算によって定義できるとき，代数は抽象データ型として有用である．例えば自然数の加算演算 plus を s のみに基づいて次のように定義できる．

$$\mathrm{plus}(0, y) = y,$$
$$\mathrm{plus}(s(x), y) = s(\mathrm{plus}(x, y))$$

加算演算ができたので，これに基づいて次のように乗算演算を定義できる．

$$\mathrm{mult}(0, y) = 0,$$

$$\mathrm{mult}(s(x), y) = \mathrm{plus}(\mathrm{mult}(x, y), y)$$

「前者（predecessor）」演算 p を，s に基づいて「$p(s(x)) = x$」により定義できる．自然数だけを扱っているので，$p(0)$ を未定義としてもよいし，問題を起さないように適切な定義を与えてもよいが，通常は，$p(0) = 0$ と定義する．これらの概念を含む代数を記述するために，零判定が返す真偽の結果を含む別の台も必要である．そこで Boolean = {True, False} とし，s と p をより記述的な名称 succ，pred で置き換え，自然数の抽象データ型を表す次の代数を得る．

台: $\mathbb{N}$, Boolean

演算: $0 \in \mathbb{N}$

$\mathrm{isZero} : \mathbb{N} \to \mathrm{Boolean}$

$\mathrm{succ} : \mathbb{N} \to \mathbb{N}$

$\mathrm{pred} : \mathbb{N} \to \mathbb{N}$

公理: $\mathrm{isZero}(0) = \mathrm{True}$

$\mathrm{isZero}(\mathrm{succ}(x)) = \mathrm{False}$

$\mathrm{pred}(0) = 0$

$\mathrm{pred}(\mathrm{succ}(x)) = x$

先の公理 $\mathrm{succ}(x) \neq 0$ は，同じアイディアを表す新しい公理 $\mathrm{isZero}(\mathrm{succ}(x)) = \mathrm{False}$ で置き換えている．また先の公理「$\mathrm{succ}(x) = \mathrm{succ}(y)$ であれば $x = y$」は，新しい公理「$\mathrm{pred}(\mathrm{succ}(x)) = x$」で置き換えている．なぜなら，$\mathrm{succ}(x) = \mathrm{succ}(y)$ であれば $\mathrm{pred}(\mathrm{succ}(x)) = \mathrm{pred}(\mathrm{succ}(y))$，そして $\mathrm{pred}(\mathrm{succ}(x)) = x$，$\mathrm{pred}(\mathrm{succ}(y)) = y$ より $x = y$ と結論づけられるからである．

これらの基本演算によって plus を次のように書き改めることができる．

$$\mathrm{plus}(x, y) = \text{if } \mathrm{isZero}(x) \text{ then } y \text{ else } \mathrm{succ}(\mathrm{plus}(\mathrm{pred}(x), y))$$

mult も先の基本演算と plus に基づいて次のように書くことができる．

$$\mathrm{mult}(x, y) = \text{if } \mathrm{isZero}(x) \text{ then } 0 \text{ else } \mathrm{plus}(\mathrm{mult}(\mathrm{pred}(x), y), y)$$

以下では，コンピュータ科学における代表的な抽象データ型の代数的仕様について述べる．代数的仕様に基づいて，対応するデータ構造が正しく実装されていることを確認できる．

4-1-3 代表的な抽象データ型の代数的仕様

プログラミングに際しては，データ構造を設計し実装しなければならない．そのとき実装したデータ構造が正しく動作することを保証するのが代数的仕様記述である[1)]．

つまり抽象データ型の公理を用いることで，プログラマは実装が正しいことを確かめることができる（つまり実装された演算が公理を満たしているかどうかを検証できる）．以下で

は，代表的なデータ構造であるリスト，スタック，キュー，そして優先度つきキューを用いて，抽象データ型がどのように実践的に使用されるか見る．

（1）リスト

集合 A 上の**リスト（list）**とは，構成子としての空リスト $\langle\,\rangle$ と cons 演算（中置形式 ::）を用いて帰納的に定義できる．ここでは A 上のすべてのリストの集合を lists(A) と記す．

基底: $\langle\,\rangle \in \mathrm{lists}(A)$

帰納: $x \in A$ かつ $L \in \mathrm{lists}(A)$ であれば $\mathrm{cons}(x, L) \in \mathrm{lists}(A)$

構成子 $\langle\,\rangle$ と cons 及び基本演算 isEmptyL, head, tail を用いて，リストの代数的仕様が定義できる．

台: lists(A), A, Boolean

演算: $\langle\,\rangle \in \mathrm{lists}(A)$

$\mathrm{isEmptyL} : \mathrm{lists}(A) \rightarrow \mathrm{Boolean}$

$\mathrm{cons} : A \times \mathrm{lists}(A) \rightarrow \mathrm{lists}(A)$

$\mathrm{head} : \mathrm{lists}(A) \rightarrow A$

$\mathrm{tail} : \mathrm{lists}(A) \rightarrow \mathrm{lists}(A)$

公理: $\mathrm{isEmptyL}(\langle\,\rangle) = \mathrm{True}$

$\mathrm{isEmptyL}(\mathrm{cons}(x, L)) = \mathrm{False}$

$\mathrm{head}(\mathrm{cons}(x, L)) = x$

$\mathrm{tail}(\mathrm{cons}(x, L)) = L$

上の基本演算を用いて，必要な任意のリスト関数を記述できる．次の関数をリスト代数の演算として実装したい．

length:	$\mathrm{lists}(A) \rightarrow \mathbb{N}$	リスト長さを求める．
member:	$A \times \mathrm{lists}(A) \rightarrow \mathrm{Boolean}$	リストのメンバを判定する．
last:	$\mathrm{lists}(A) \rightarrow A$	リストの最後の要素を求める．
concatenate:	$\mathrm{lists}(A) \times \mathrm{lists}(A) \rightarrow \mathrm{lists}(A)$	リストを連結する．
putLast:	$A \times \mathrm{lists}(A) \rightarrow \mathrm{lists}(A)$	要素をリストの右端に配置する．

いくつか例を挙げる．リスト代数のシグネチャ中のすべての演算が実装されていると仮定したとき，length の定義は次のように書くことができる．

$$\mathrm{length}(L) = \mathrm{if}\ \mathrm{isEmptyL}(L)\ \mathrm{then}\ 0$$
$$\mathrm{else}\ 1 + \mathrm{length}(\mathrm{tail}(L))$$

この場合 length 関数が適切に動作するためには，代数 $\langle \mathbb{N}; +, 0 \rangle$ が実装されていなければな

らない．同様に「member」を次のように定義したとする．

$$\begin{aligned}\mathrm{member}(a, L) = {} & \text{if } \mathrm{isEmptyL}(L) \text{ then False} \\ & \text{else if } a = \mathrm{head}(L) \text{ then True} \\ & \text{else } \mathrm{member}(a, \mathrm{tail}(L))\end{aligned}$$

この場合述語「$a = \mathrm{head}(L)$」を計算する必要がある．つまり台 A 上で等価関係を実装しなければならない．

これらの例が示すように，リストの代数によってリスト関数は定義できるものの，加えて $\langle \mathbb{N}; +, 0 \rangle$ のようなほかの代数や A 上の等価関係のようなほかの関係が必要となることも多い．

(2) スタック

スタック（stack） とは，後入れ先出し（LIFO，last in first out）の性質を満たすデータ構造である．つまり，最後に入力された要素が最初に出力される．スタック演算の主なものは，スタックに新しい要素を押し込む push，スタックから最上位の要素を取り除く pop，そしてスタックの最上位の要素を調べる top である．更にスタックが空であるときに，それを知る方法も必要である．

スタックの仕様を代数として記述することにしよう．任意の集合 A について，A の要素を要素とするスタックの集合を $\mathrm{Stks}[A]$ で記す．演算が定義されない場合のために，エラーメッセージを記述中に含めることにする．

台: A, $\mathrm{Stks}[A]$, Boolean, Errors

演算: $\mathrm{emptyStk} \in \mathrm{Stks}[A]$

$\mathrm{isEmptyStk} : \mathrm{Stks}[A] \to \mathrm{Boolean}$

$\mathrm{push} : A \times \mathrm{Stks}[A] \to \mathrm{Stks}[A]$

$\mathrm{pop} : \mathrm{Stks}[A] \to \mathrm{Stks}[A] \cup \mathrm{Errors}$

$\mathrm{top} : \mathrm{Stks}[A] \to A \cup \mathrm{Errors}$

公理: $\mathrm{isEmptyStk}(\mathrm{emptyStk}) = \mathrm{True}$

$\mathrm{isEmptyStk}(\mathrm{push}(a, s)) = \mathrm{False}$

$\mathrm{pop}(\mathrm{push}(a, s)) = s$

$\mathrm{pop}(\mathrm{emptyStk}) = \mathrm{stackError}$

$\mathrm{top}(\mathrm{push}(a, s)) = a$

$\mathrm{top}(\mathrm{emptyStk}) = \mathrm{valueError}$

スタック代数とリスト代数の類似に注目されたい．実際のところスタック代数は，スタック記号に次のような意味を割り当てることでリスト代数として実装することができる．

$$
\begin{aligned}
\text{Stks}\,[A] &= \text{lists}\,(A)\\
\text{emptyStk} &= \langle\,\rangle\\
\text{isEmptyStk} &= \text{isEmptyL}\\
\text{push} &= \text{cons}\\
\text{pop} &= \text{tail}\\
\text{top} &= \text{head}
\end{aligned}
$$

この実装が正しいことを証明するためには，上記の割当てについてスタックの公理が真であることを示させばよい．これらは次のように，それぞれ一行で容易に書くことができる．

$$
\begin{aligned}
&\text{isEmptyStk}(\text{emptyStk}) = \text{isEmptyL}(\langle\,\rangle) = \text{True}\\
&\text{isEmptyStk}(\text{push}(a, s)) = \text{isEmptyL}(\text{cons}(a, s)) = \text{False}\\
&\text{pop}(\text{push}(a, s)) = \text{tail}(\text{cons}(a, s)) = s\\
&\text{top}(\text{push}(a, s)) = \text{head}(\text{cons}(a, s)) = a
\end{aligned}
$$

(3) キュー

キュー（queue）とは，先入れ先出し（FIFO，first in first out）の性質を満たすデータ構造である．つまり，最初に入力された要素が最初に取り出される．キュー上の演算の主なものは，新しい要素を加える，先頭の要素を調べる，先頭の要素を取り除く，などである．A を集合とし，A 上のキューの集合を Q[A] と記す．代数的仕様記述は，次のとおり．

台: A, $Q[A]$, Boolean

演算: $\text{emptyQ} \in Q[A]$

$\text{isEmptyQ} : Q[A] \to \text{Boolean}$

$\text{addQ} : A \times Q[A] \to Q[A]$

$\text{frontQ} : Q[A] \to A$

$\text{delQ} : Q[A] \to Q[A]$

公理: $\text{isEmptyQ}(\text{emptyQ}) = \text{True}$

$\text{isEmptyQ}(\text{addQ}(a, q)) = \text{False}$

$\text{frontQ}(\text{emptyQ}) = \text{queueError}$

$\text{delQ}(\text{emptyQ}) = \text{queueError}$

$$
\begin{aligned}
\text{frontQ}(\text{addQ}(a, q)) &= \text{if isEmptyQ}(q) \text{ then } a\\
&\quad \text{else frontQ}(q)\\
\text{delQ}(\text{addQ}(a, q)) &= \text{if isEmptyQ}(q) \text{ then } q\\
&\quad \text{else addQ}(a, \text{delQ}(q))
\end{aligned}
$$

キューをリストして表現することにしよう．例えばリスト $\langle a, b\rangle$ は，先頭が a で後方が b であるキューを表している．このキューに新しい項目 c を加えるとキュー $\langle a, b, c\rangle$ を得る．したがって addQ(c, $\langle a, b\rangle$) = $\langle a, b, c\rangle$ である．つまり addQ は，リストの最後に要素を付け加える putLast 関数として実装できる．リスト代数としてのキュー代数の実装は，次のように与えることができる．

$$
\begin{aligned}
Q\,[A] &= \mathrm{lists}\,(A)\\
\mathrm{emptyQ} &= \langle\,\rangle\\
\mathrm{isEmptyQ} &= \mathrm{isEmptyL}\\
\mathrm{frontQ} &= \mathrm{head}\\
\mathrm{delQ} &= \mathrm{tail}\\
\mathrm{addQ} &= \mathrm{putLast}
\end{aligned}
$$

この実装が正しいことを証明するため，上記の割当てについてスタックの公理が真であることを示す．キューの公理の二つは条件式を含み，かつ putLast は，リストの基本演算を用いて書かれているので少し複雑である．

$$\mathrm{isEmptyQ}(\mathrm{emptyQ}) = \mathrm{isEmptyL}(\mathrm{emptyL}) = \mathrm{True}$$

$$
\begin{aligned}
\mathrm{isEmptyQ}(\mathrm{add}(a, q)) &= \mathrm{isEmpty}(\mathrm{putLast}(a, q)\\
&= \mathrm{isEmptyL}(\mathrm{if\ isEmptyL}(q)\ \mathrm{then\ consL}(a, q)\\
&\quad\ \mathrm{else\ consL}(\mathrm{headL}(q), \mathrm{putLast}(a, \mathrm{tailL}(q)))\\
&= \mathrm{if\ isEmptyL}(q)\ \mathrm{then\ isEmpty}(\mathrm{consL}(a, q))\\
&\quad\ \mathrm{else\ isEmptyL}(\mathrm{consL}(\mathrm{headL}(q), \mathrm{putLast}(a, \mathrm{tailL}(q))))\\
&= \mathrm{False}
\end{aligned}
$$

3 番目の公理は if-then-else 文なので，二つの場合に分けて考える．場合 1 は，q = emptyQ のときである．

$$
\begin{aligned}
\mathrm{frontQ}(\mathrm{addQ}(a, \mathrm{emptyQ})) &= \mathrm{head}(\mathrm{putLast}(a, \mathrm{emptyQ}))\\
&= \mathrm{head}(a :: \mathrm{emptyQ})\\
&= a
\end{aligned}
$$

そして場合 2 は，$q \neq$ emptyQ のときである．

$$
\begin{aligned}
\mathrm{frontQ}(\mathrm{addQ}(a, q)) &= \mathrm{head}\,(\mathrm{putLast}\,(a, q))\\
&= \mathrm{head}\,(\mathrm{head}\,(q) :: \mathrm{putLast}\,(a, \mathrm{tail}\,(q)))\\
&= \mathrm{head}\,(q)\\
&= \mathrm{frontQ}\,(q)
\end{aligned}
$$

4 番目の公理も if-then-else 文ではあるが，まとめて書いても，それほど複雑にならない．

delQ(addQ(*a, q*)) = tailL(putLast(*a, q*))
= tailL(if isEmptyL(*q*) then consL(*a, q*)
else consL(headL(*q*), putLast(*a*,tailL(*q*))))
= if isEmptyL(*q*) then tailL(consL(*a, q*))
else tailL(consL(headL(*q*), putLast(*a*,tailL(*q*))))
= if isEmptyL(*q*) then *q* else putLast(*a*, tailL(*q*))
= if isEmptyQ(*q*) then *q* else addQ(*a*, deleQ(*q*))

（4）優先度つきキュー

優先度つきキュー（priority queue）とは，最良先出し（BIFO，best in first out）の性質を満たすデータ構造である．例えば Best = Last とすればスタックは優先度つきキューである．同様に Best = First とすればキューも優先度つきキューである．優先度つきキューの主な演算には，新しい要素の追加，最良要素へのアクセス，そして最良要素の削除が含まれる．

P[*A*] を，*A* 上の優先度つきキューの集合を記すものとする．$a \in A$ かつ $p \in P[A]$ とすると insert(*a, p*) は，*a* を *p* に加えて得られた優先度つきキューである．優先度つきキューの抽象データ型は，次の代数として記述できる．

台: *A*, *P*[*A*], Boolean

演算: emptyP ∈ *P*[*A*]

isEmptyP : *P*[*A*] → Boolean

better : $A \times A \rightarrow$ Boolean

best : *P*[*A*] → *A*

insert : $A \times P[A] \rightarrow P[A]$

delBest : *P*[*A*] → *P*[*A*]

ここで関数「better」を，*A* 上の二項関係と仮定する．

公理: isEmptyP(emptyP) = True

isEmptyP(insert(*a, p*)) = False

best(emptyP) = pQError

delBest(emptyP) = PQError

best(insert(*a, p*)) = if isEmptyP(*p*) then *a*
else if better(*a*, best(*p*)) then *a*
else best(*p*)

delBest(insert(*a, p*)) = if isEmptyP(*p*) then emptyP

else if better(a, best(p)) then p
else insert(a, delBest(p))

演算 best と delBest は，空でない優先度つきキュー上でだけ定義されることに注目されたい．優先度つきキューは，集合 A 上で「better」や「best」をどのように定義するかによって様々に異った方法で実装することができる．

優先度つきキューの有効性を示すために，優先度つきキューの要素を整列して整列リストに収める整列関数を書くことにしよう．

$$\begin{aligned}\mathrm{sort}\,(p, L) = {} & \text{if isEmptyP}\,(p)\ \text{then}\ L \\ & \text{else sort}\,(\mathrm{delBest}\,(p)\,, \mathrm{best}\,(p) :: L)\end{aligned}$$

優先度つきキュー p を整列させる初期呼出しは，sort(p, ⟨ ⟩) である．

■参考文献

1) James L. Hein 著, 神林　靖 訳, “独習 コンピュータ科学基礎 II 論理構造,” 翔泳社, 2011.
2) 久野　靖, “CLU とその仲間たち（連載）,” bit, vol.21, no.6-14, 1989.
3) Barbara Liskov, John Guttag, “Abstraction and Specification in Program Development,” MIT Press/McGraw-Hill, 1986.
4) Mary Shaw, “ALPHARD: Form and Content,” Springer-Verlag, New York, 1981.

■6 群 - 3 編 - 4 章

4-2 オブジェクト指向

（執筆者：久野　靖）[2012 年 7 月 受領]

4-2-1 オブジェクト指向の基本的な考え方

オブジェクト指向とは，データを自律的なオブジェクトとしてとらえ，計算をオブジェクト間のメッセージ交換によって進むものとしてとらえる考え方である．プログラミング言語におけるオブジェクト指向は，プログラミング言語 Simula 67[2)]に端を発しており，Smalltalk-80[3)]言語によって広く知られるようになった．今日では C++[4)]，Java[1)]をはじめメインストリームの手続き型言語は多くがオブジェクト指向の機能を備える．

オブジェクト指向のモデルでは，それぞれのオブジェクトはそれが理解できる（取り扱える）メッセージの集合（プロトコル）をもち，オブジェクト間のやりとりはこのメッセージの交換に限られる（図 4･2 左）．具体的には，メッセージに対応して操作（メソッドとも呼ばれる）が起動され，計算が行われる（図 4･2 右）．オブジェクトの内部状態はほかのオブジェクトからは隠蔽されアクセスできない．

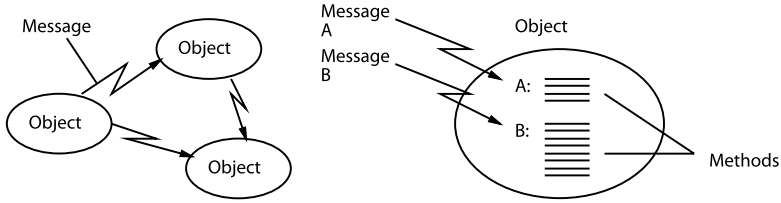

図 4･2　オブジェクトとメッセージ

これは，抽象データ型がその内部表現を隠して操作のみによって定義されるのと同様であり，実際，オブジェクト指向言語におけるオブジェクトの最も基本的な用途はオブジェクトを抽象データ型として扱い，抽象化が提供する利点である，内部実現と外部インタフェースを分けて扱えることを活用するところにある．

抽象データ型に基づく言語とオブジェクト指向言語の最大の違いは，オブジェクト（ないし抽象データ型）の内部表現の記述方法にある．CLU をはじめとする抽象データ型言語では，抽象型 *A* に対して，その内部実現となる実装型 *R* を対応させることとし，*R* としては基本型，レコード，配列など任意の型を用いることができる．

これに対し，オブジェクト指向言語では，オブジェクトの内部実現は「複数の変数（インスタンス変数）の集まり」であるとするのが通例である．それぞれの変数は固有の名前と型をもつので，その集まりはレコード型に相当する．すなわち，オブジェクトの内部実現は暗黙的なレコードだといえる（図 4･3）．C++ 言語ではこの考え方を推し進め，struct によって定義されるレコード型はメソッドをもたないようなオブジェクト型と位置づけられている．

メソッドは基本的には手続きであるが，各オブジェクトのメソッド内ではそのオブジェクトのインスタンス変数は単に名前だけで参照できる．これによって「オブジェクトは変数の集ま

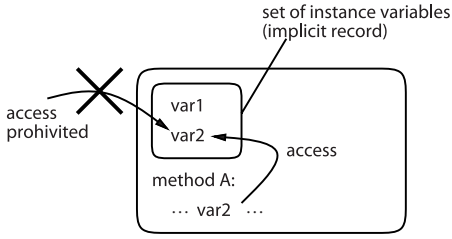

図 4･3　オブジェクトとインスタンス変数

り」という見え方を提供している．場合によっては暗黙的なレコード全体（オブジェクトそのもの）を参照したい場合もあり，そのような場合は self，this などの**擬変数（pseudovariable）**を用いて参照できる．

図 4･4 は Ruby によって有理数を表現するクラス（抽象データ型）Rational を定義した例である（簡単のため，分子は非負整数であるものとした）．インスタンス変数@dividend，@divisor に分子と分母の整数値を保持し，加算に際しては通分を行う．メソッド initailze はインスタンスを初期化するための特別なメソッドであり，Rational.new によってオブジェクトが生成されるときに自動的に呼び出される．このなかで分母と分子を両者の最大公約数で割ることで既約分数としている．実行の様子は次のようになる．

```
irb> x = Rational.new(3, 7)
=> #<Rational:0x810c0f4 @divisor=7, @dividend=3>
irb> y = Rational.new(4, 5)
=> #<Rational:0x80ff700 @divisor=5, @dividend=4>
irb(main):018:0> puts x + y
43/35
=> nil
```

ここで「x + y」の「+」は Rational で定義されたメソッドが呼び出されていることに注意されたい．また，puts は渡されたオブジェクトのメソッド to_s を呼び出して文字列表現を得て出力するので，分数表現が出力されている．

Ruby をはじめとして，C++や Java など多くの言語ではオブジェクトを図 4･4 のような**クラス（class）**と呼ばれる構文単位によって定義し，同じクラスから生成されたオブジェクトは，同じ定義を共有する．これを**クラス方式（class based）**のオブジェクト指向言語と呼ぶ．Simula 67 や Smalltalk-80 もクラス方式の言語であった．

これと異る方式として，**プロトタイプ方式（prototype based）**の言語がある．プロトタイプ方式では，ひな型となるオブジェクトを構築し，（言語によっては論理的に）コピーすることで同種のオブジェクトを生成していく．プロトタイプ方式を採用した言語としては，Self，JavaScript などが挙げられる．

```
class Rational
  def initialize(a, b = 1)
    if a == 0
      @dividend = 0; @divisor = 1
    else
      x = gcd(a.abs, b.abs); @dividend = a/x; @divisor = b/x
    end
  end
  def divisor
    @divisor
  end
  def dividend
    @dividend
  end
  def +(x)
    Rational.new(@divisor * x.dividend + x.divisor * @dividend,
                 @divisor * x.divisor)
  end
  def *(x)
    Rational.new(@dividend * x.dividend, @divisor * x.divisor)
  end
  def to_s
    "#{@dividend}/#{@divisor}"
  end
  private
  def gcd(a, b)
    while true
      if a % b == 0 then return b else a = a % b end
      if b % a == 0 then return a else b = b % a end
    end
  end
end
```

図 4･4　Ruby による「有理数」クラスの定義

4-2-2 動的分配（多相性）

動的分配（dynamic dispatch）とは一般的にはプログラム実行時に複数のコードを切り替えることをいう．オブジェクト指向の文脈でいえば，プログラムの字面上で $o.M(\cdots)$ というメソッド呼出しがあったとき，実際に呼び出されるメソッド M が「実行時に」o が表すオブジェクトに応じて決まる（そのオブジェクトに対応して定義されているメソッドが呼び出される）ことをいう．CLOS のように，言語によっては単一のオブジェクトではなく，メソッド呼出しの引数となっている複数のオブジェクト群の種別に応じてメソッドが選択されるものもある．これを**マルチメソッド（multimethods）**と呼ぶ．なお，一つの字面が複数の意味をもつことを一般に**多相性（polymorphism）**と呼ぶことから，オブジェクト指向における動的分配も多相性と呼ばれることもある．動的分配はコードの記述を簡潔で見通しよくするうえで，大きな効果をもつ．

弱い型の（型検査を行わない）言語では，プログラムの字面上の式や変数は実行時に様々な種別のものであり得るため（本来的に多相性をもつ），その種別に応じたメソッドが呼び出されることは自然に実現できる．これは，クラス方式の言語であればオブジェクトが実行時に自分のクラスの情報を参照でき，そこを経由して動かすメソッドの情報を取得することで実現される．プロトタイプ方式では，オブジェクト自身のプロパティ（ないし論理的なコピー元の対応するプロパティ）からメソッドを取り出すことで同様のことが行える．更に言語によっては，個別のオブジェクトに対してメソッドを定義することで，個々のオブジェクトごとに固有のメソッドが動くようにもできる．

強い型の（型検査を行う）オブジェクト指向言語では，複数のオブジェクト種別を包含するような型を何らかの方法で導入し，その型をもつ式に対しては，その型に定義されているメソッド呼出しが記述され，それに対して動的分配を行う．一般にはこれを，継承（下記）によるクラスの親子関係を定義し，親クラスの変数には任意の子クラスのオブジェクトを保持できるようにすることで行う言語が多い（C++など）．このほか，インタフェースとして一群のメソッド名と引数・返値の型を定義し，インタフェース型の変数にはそのインタフェースに従う様々なオブジェクトを格納できるようにする流儀などがある（Java はこの方式と前述のクラス継承による方式を併用している）．これらの場合，個々のクラスやインタフェースが一つの型に対応している必要がある．

動的分配の実現方法は基本的には，オブジェクトごとにそのオブジェクトのメソッドコードへのポインタをもたせることによる．ただし，全オブジェクトに全メソッドへのポインタをもたせることはメモリ消費の点で無駄が大きい．クラス方式のオブジェクト指向言語では，同じクラスのインスタンスは同じメソッド群をもつ．このため，メソッドポインタを集めたメソッド表を用意し，各オブジェクトはこの**メソッド表（method table）**へのポインタをもたせることが一般的である（図 4･5）．メソッド表はクラスに対応しているので，この表を実行時に利用可能なクラスに関する情報を格納するデータ構造としても使うことができる．

4-2-3 継承と委譲

継承（inheritance）とは，クラス方式の言語において，あるクラスを土台として新しいクラスを定義することをいう．このとき，土台となるクラスを**親クラス（superclass）**，新しく定義するクラスを**子クラス（subclass）**と呼ぶ．多くの言語では，子クラスは親クラスを一つ

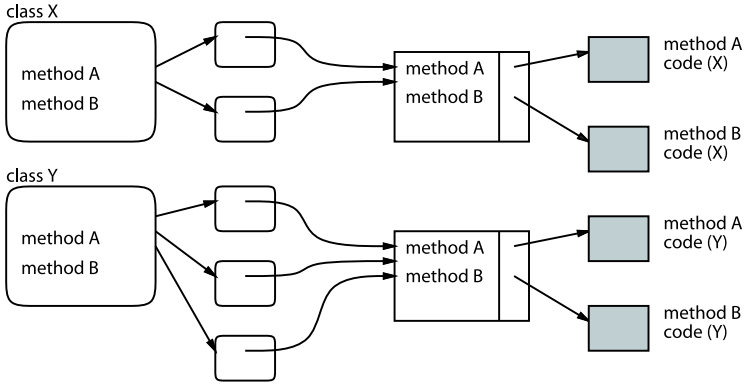

図 4･5　メソッド表

だけもつことができる．これを**単一継承（single inheritance）**と呼ぶ．子クラスは（論理的には）親クラスに定義されているインスタンス変数群とメソッド群を引き継ぎ，それに自分固有の変数群とメソッド群を追加したものである．更に，親クラスに定義されているメソッドを差し替え（上書き定義）することもできる．これを**オーバライド（override）**と呼ぶ．

これらの機能により，子クラスのオブジェクトは親クラスのオブジェクトと同じメソッド群をもちながら，その振る舞いは違ったものとなり得るため，前述のようにこれを基にして動的分配を行うことは言語設計上の自然な選択肢の一つである．

ただし，継承では親クラスと子クラスはインスタンス変数定義を共通にするため，オブジェクトの内部構造も類似したものとなる．これは，本来，動的分配はオブジェクトやメソッドの内部実現とは独立した概念であるのに，継承を用いることで両者に依存関係ができてしまうという副作用をもっているといえる．インタフェースのみに基づく多相性ではこのような問題はない．

一方，継承では親クラスから実装を引き継ぐため，少ないコード量で新しいクラスを定義することができる．これにより少量の「差分」を記述することで次々にクラス群を定義することを**差分プログラミング（differential programming）**と呼ぶ．差分プログラミングは Smalltalk-80 のライブラリで多用されたが，継承によるクラス間の依存関係の多さのため，保守性に問題があるとする意見もある．

継承において，一つの子クラスが複数の親クラスをもつものを**多重継承（multiple inheritance）**と呼ぶ．多重継承の場合，複数の親から同名のメソッドを継承したり，それらの親が更に共通の親から同一の変数やメソッドを継承することもあるため，意味づけが面倒である．例えば C++の場合，重複継承した変数を子クラスの内部で複数保持する場合と一つに統合する場合とをプログラマが選択可能になっている．

クラス方式の言語で単一継承を採用している場合，そのオブジェクトやメソッドテーブルに対して効率のよい実装が可能である．例えば図 4･6 では，クラス Y，Z はクラス X の子クラスであり，親クラスの変数 p，q を継承している．そこで，これらのクラスのインスタン

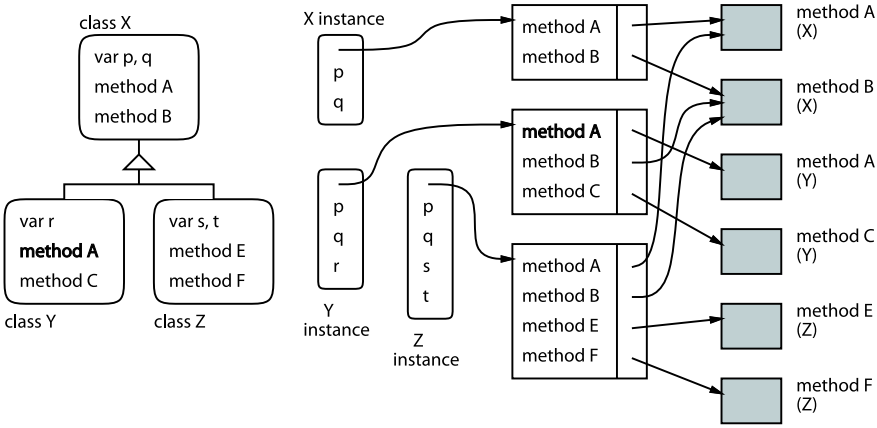

図 4･6　単一継承の効率よい実現

スは親クラスのインスタンスの末尾に自クラスで追加したインスタンス変数を加えるように配置する．これにより，親クラスで定義されている変数の位置は子クラスでも変わらないので，オブジェクト先頭からのオフセットでインスタンス変数をアクセスするように実装した親クラスのメソッドをそのまま動かすことができる．また，メソッド表も同様に，子クラスのメソッド表は親クラスのメソッド表の後ろに追加したメソッドのエントリを置くようにすることで，インスタンス変数と同様，オフセットにより効率よいアクセスができる．メソッド表に格納するメソッドコードのポインタは，親クラスからメソッドを継承している場合は親クラスのメソッド表と同一であり，オーバライドしている場合はそのコードを格納する．

インタフェースによる動的分配や多重継承を含む場合の動的分配はこのようなかたちでオフセットを同一に維持できないので，言語上で動的分配に対する制約を設ける（C++の場合），実行時にメソッド名でメソッド表内を検索する（Java のインタフェースの場合），などの方法が取られる．実行時に検索する場合は，よく使われるクラスに対して専用のコードを生成する，検索結果をキャッシュするなどの工夫によりオーバヘッドを軽減することが多い．

プロトタイプ方式の言語では，オブジェクトの機能を拡張するのに**委譲（delegetion）**と呼ばれる機構が用いられる．もともと，プロトタイプ方式具では複数の類似したオブジェクトを定義するのに，**ひな型（prototype）**を論理的にコピーして作成する．具体的には，新規オブジェクトにプロトタイプへのポインタをもたせ，新規オブジェクトにないプロパティ（メソッドなど）を参照する際にはこのポインタをたどってプロトタイプのプロパティを使用する（図 4･7）．これを，自分が直接保持していないメソッドをプロトタイプに委譲する，というふうに呼ぶ．プロトタイプ関係は一般に多段になっていて，それらを順次指すポインタの連鎖をたどっていくことになる．例えば図 4･7 では，method B についてはすべてのオブジェクトは object X がもっているものを使用するが，method A については object T，object Y は object Y がもっているものを使用する．

継承がクラス生成時に変数定義やメソッド定義を「継承してくる」という点で静的であり，

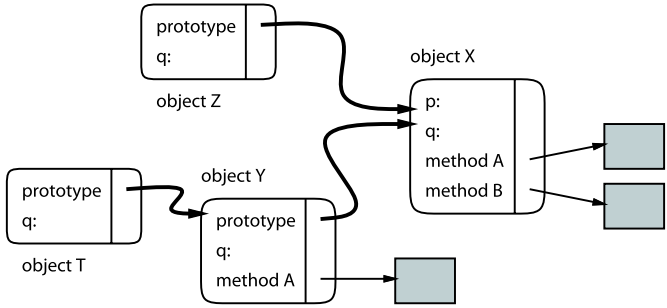

図 4･7　プロトタイプ方式と委譲

実行時には変化しないのに対し，委譲は実行時にプロトタイプの連鎖をたどって変数やメソッドを探すため動的であり，（言語にその機構があれば）実行時に委譲先を変更することも可能である．

委譲においても多重継承と同様，プロトタイプを複数もたせることは論理的には可能だが，主流となっている言語でそのようなモデルを採用しているものはない．これは実装上，コンパイル時に親子関係をすべて把握してどの優先順位を調整できる継承と異なり，実行時に連鎖をたどっていく委譲の場合，実行時に枝分かれのある連鎖をすべてたどることは実用的でないからだと思われる．

なお，クラス方式の言語であっても，部分的にコード規約として委譲を実現することはあり得る（委譲先となるオブジェクトを決まった変数に保持し，そのメソッドを呼び出すかたちとなる）．**デザインパターン（design patterns）**のなかでも Policy パターンや Strategy パターンなどはそのようなものの定式化だと考えることができる．

■参考文献

1) Ken Arnold, James Gosling, David Holmes, “The Java Programming Language, Fourth Edition,” Addison-Wesley, 2006.
2) G.M. Birtwistle, Simula Begin, “Van Nostrand Reinhold,” 1979.
3) Adele Goldberg, David Robson, “Smalltalk-80 The Language,” Addison-Wesley, 1989.
4) Bjane Stroustrup, “The C++ Programming Language Special Edition,” Addison-Wesley, 2000.

■6 群 - 3 編 - 4 章

4-3 ジェネリックプログラミング

（執筆者：久野　靖）[2012 年 7 月 受領]

4-3-1 ジェネリック性の定義と由来

プログラミング言語において**ジェネリック（generic）**ないし汎用的とは，一つのコードの字面が複数の型に対して適用できることをいう．型が違えばそのコードの振る舞いも違ってくるので，ジェネリック性は多相性の下位概念と考えられる．ジェネリック性は必ずしも新しい概念ではなく，例えば多くのプログラミング言語において「+」という演算子は整数の加算にも実数の加算にも使われる．したがって，これらの言語において「+」はジェネリックな演算子ということになる．

古典的な言語ではジェネリックな要素は言語に組込みの演算子や関数としてしか存在しなかったが，近年ではこのようなものをユーザが定義できる言語が増えている．例えば同じ演算子や同名の関数を**多重定義（overload）**することを許し，使用場面ごとに最も適合する定義を選択することで，一つの字面を複数の型に適用することができる．このような機構は Ada にはじまり，Java や C++など，多くの言語に引き継がれている．ただし，この方法では異る型ごとに別の定義を用意するので，コード量の節約にはならない．

更に，クラスやモジュールなどの構文単位に**型パラメタ（type parameter）**をもたせ，このパラメタに様々な型を指定することで，一つの構文単位を複数の型に対して適用可能にできる．このような方式は，CLU 言語において，抽象データ型を定義するモジュール（CLU の用語では cluster）にパラメタをもたせたのが最初である．

型パラメタそのものは，組込みの型については古くから存在していたと考えることができる．例えば配列型は，整数の配列，実数の配列，文字の配列など任意の型について定義できる．そこでこれを *array*[*T*] と表記し，型パラメタ *T* に具体的な型（整数，実数，…）を与えることで，その型の要素が並んだ配列型を定義しているものと考える．この場合，*array* は「パラメタを与えることで具体的な型をつくり出す」ことから**型生成子（type generator）**と呼ばれる．

4-3-2 C++と Java のパラメタつきクラス

C++や Java では，クラスに型パラメタをもたせることができ，これによってクラスを型生成子として機能させられる．例えば図 4･8 は，型 *T* の要素を昇順に並べて保持し，検索によって「小さい方から何番目か」を返すことができるような集合型を定義する C++のコードである（定数 `size` はこのコードの外のどこかで定義されているものとする）．

型パラメタ機構の設計や実装において難しい点は，パラメタつきクラスの内部で型 *T* の操作を呼び出す場合があるため，(1)*T* がもつ操作をどのようにして宣言させ型検査させるか，(2) 実際に *T* の操作呼出し（当然，*T* が異なれば呼び出すべき操作の実体は別のものになる）をどのように実現するか，である．

C++では，パラメタつきのクラスはパラメタ型 *T* に具体的な型が当てはめられるごとに，*T* をその具体的な型で置き換えて実体化（ひらたく言えばコンパイル）する，という方針を取っている．これを**非均質な実装（heterogenous implementation）**と呼ぶ（図 4･9 左）．非

均質な実装では適用されるパラメタ型ごとにそのクラスのコードが別につくられるため，コンパイル後のコード量の増大を招きやすいが，個別の型ごとの最適化が行いやすく実行効率は一般に良い．

```
template<class T> class ordset {
  T buf[size];
  int top;
public:
  ordset() { top = -1; }
  void insert(T x) {
    int i = 0;
    while(i <= top && buf[i] < x) { ++i; }
    if(i <= top && buf[i] == x || top+1 >= size) { return; }
    for(int j = top++; j >= i; --j) { buf[j+1] = buf[j]; }
    buf[i] = x;
  }
  int find(T x) {
    for(int i = 0; i <= top; ++i) { if(buf[i] == x) { return i; } }
    return -1;
  }
};
```

図 4･8　C++による「T 型の順序つき集合」クラスの定義

また，型検査についても置き換えて実体化する際になされるので，とりたてて宣言をしなくても済む．ただし，宣言をしないことはパラメタ型に課される要件が明確にならないという問題があり，また実体化の際に不整合があった場合のエラーメッセージがクラス内部の実装に対応したものとなり理解しづらいという問題がある．このため，C++コミュニティでは concept と呼ばれる型パラメタの要件を指定する言語機構を追加することで検討を行ってきている（現時点では正式な言語標準には組み込まれていない）．

Java では，パラメタつきクラスのパラメタ型として指定できるものをオブジェクト型（クラス型）のみに制限し，なおかつ「どのクラスのサブクラスか」という指定を書くことができる（指定しなければ Object 型）ようにすることで，通常のオブジェクト型と同様の型検査を行い，また T 型の操作呼出しも通常の動的分配によることで，実際に使われるパラメタ型の種類や数にかかわらず一つの実装だけで済ませるようになっている．これを**均質な実装（homogeneous implementation）**と呼ぶ（図 4･9 右）．均質な実装はコンパイル後のコード量は小さくなるが，パラメタ型 T の種類に応じた最適化が行えず，また T の操作を呼び出すごとに動的分配のコストが必要になることから，実行効率面では不利となる。

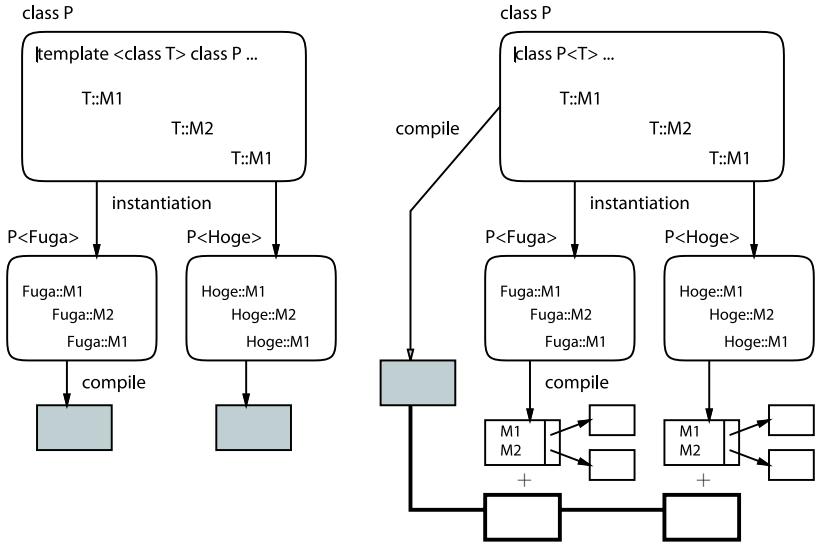

図 4･9　テンプレートの非均質な実装（左）と均質な実装（右）

■参考文献

1) Brett McLaughlin, David Flanagan, “Java 1.5 Tiger — A Developer’s Notebook,” O’Reilly, 2004.
2) David Vandevoorde, Nicolai M. Josttis, “C++ Templates — The Complete Guide,” Addison-Wesley, 2003.