Vol. 29 No. 8　　情　報　処　理　　Aug. 1988

解　説

関数型言語の基本概念†

横内　寛文††

1. はじめに

関数型言語の特長は，代入などの副作用がなく，プログラムの各部分が計算過程によらず単独で意味をもつ点である．この性質を参照透明性 (referential transparency) と呼ぶ．従来の手続き型言語では，変数に割り当てられている値，制御の位置などの内部状態が存在し，計算は内部状態の推移で実現される．プログラムを構成している各手続きの意味は，それまでの計算結果に依存して決まる．したがって，プログラムの意味を理解するためには，頭の中でプログラムを実行し，内部状態がどのように推移していくかを考える必要がある．一方，関数型言語では，参照透明性により，より静的にプログラムの意味を捉えることができる．

本解説では，関数の概念がいかに計算の概念と結び付き，いかにプログラミング言語としてまとめられるかを考える．また，関数型言語の基礎になるいくつかの概念ないし機能を解説して，これらの基本機能を実現した関数型言語の具体例を紹介する．

2. 計算規則としての関数

関数の概念は二つの側面をもつ．一つは，値の対応である．fを集合AからBへの関数とすると，fはAの各要素xとBの要素$f(x)$の対応を表す．もう少し厳密にいうと，AからBへの関数とは，次のような集合 f と A および B の三つの組 (f, A, B) を表す．

(1) $f \subset \{\langle x, y\rangle \mid x \in A, y \in B\}$,

(2) 任意の $x \in A$ について，$y \in B$ が唯一存在して，$\langle x, y\rangle \in f$.

この意味で，関数は写像とも呼ばれる．一方，関数 f は，与えられた入力 x から出力 $f(x)$ を生成する計算規則とも考えられる．たとえば，自然数上の足し算＋は，適当な公理系を設定して，その下で定義されるが，もっと直接的に，1組の自然数 m と n から $m+n$ を求めるアルゴリズムすなわち計算規則を与えることでも定義できる．ただし，値の対応としての関数の概念は，計算規則としての関数の概念よりも広い．事実，自然数上の関数で，計算不可能なものが存在する．

関数型言語は，計算規則としての関数を記述する言語と考えられる．関数を計算規則とみる考え方は，ラムダ計算の体系の中にも見い出せる．ラムダ計算の目標の一つは，計算規則としての関数の最も基本的な性質を調べることである．ラムダ計算における諸結果は，関数型言語の理論的な基礎付けに応用されている．

計算規則の観点からみた関数の定義方法について考える．関数fを定義するためには，入力xから出力$f(x)$を求めるプログラムを記述すればよい．プログラムは，四則演算などの基本関数を組み合わせて記述される．この際，すでに定義されている関数を組み合わせて，新しい関数を作るための基本操作の選び方が重要である．関数型言語において，最も基本的な組み合わせ操作はなにかを考える．

関数型言語にとっては，まず，関数作用 (function application) が基本である．たとえば，関数fを値aに作用させることを$f(a)$ないしは$f\ a$と表す．すなわち，関数作用とは関数呼び出しのことである．また，新しい関数を定義する際には，引数の機能が必要である．たとえば，足す1を表す関数 add 1 は，add 1 $(x)=x+1$ と定義できる．あるいは，add 1 という新しい関数名を使わずに，ラムダ記法を用いて，$\lambda x.\ x+1$ で足す1を表せる．一般に，式 M から式 $\lambda x.\ M$ を作ることをラムダ束縛 (λ-binding, λ-abstraction) と呼ぶ．関数型言語にとって，関数作用とラムダ束縛が最も基本的な操作である．

以上の枠組みの中では，基本演算以外の新しく定義した関数は，ラムダ束縛 $\lambda x.M$ の形をしている．ラ

† Basic Concepts in Functional Languages by Hirofumi YOKOUCHI (IBM Research, Tokyo Research Laboratory).

†† 日本アイ・ビー・エム(株)　東京基礎研究所

ムダ束縛の意味を考える．式 M は変数 x を含むので，x に対する値が決まって初めて，M の値が決められる．変数 x の値が a のときの M の値 $[\![M]\!]_{x.a}$ がすでに定義されていると仮定すると，式 $\lambda x.M$ の意味は，各値 a に値 $[\![M]\!]_{x.a}$ を対応させる関数と定義できる．一方，関数の第2の側面，すなわち計算規則としての側面から考えると別の見方もできる．ラムダ束縛 $\lambda x.M$ は，式 N に作用させたとき，M 中の変数 x を N で置き換えた式 $M[x:=N]$ を生成する計算規則と考えることもできる．

関数型言語における計算は，与えられた式を順次変形させる簡約 (reduction) で定義できる．簡約において最も基本になる規則が，ラムダ束縛と関数作用の関連を示す β 規則 (β-rule)

$$(\lambda x.M)N \rightarrow M[x:=N]$$

である．プログラミング言語の観点からすると，β 規則は，仮引数と実引数の結合を意味する．また，四則演算などの基本演算による計算も簡約と考えられる．たとえば，足し算＋は，$1+1\rightarrow 2, 2+3\rightarrow 5$ などの簡約規則の集まりと定義できる．簡約による計算は，与えられた式に簡約規則を順次適用して実行され，もはやどの簡約規則も適用できなくなった時点で止まる．適用できる簡約規則がない式を正規形 (normal form) と呼ぶ．式の値を求めることは，その正規形を求めることに相当する．

関数型言語における関数 f は，式 N に作用させたときに，式 (fN) から始まる簡約を引き起こす計算規則と位置付けられる．関数 f の定義は，ラムダ束縛，関数作用，基本関数などから構成されており，(fN) の簡約による計算のアルゴリズムを含んでいる．ただし，個々の簡約規則を与えただけでは，関数 f の意味を完全に規定することはできない．式を簡約する際の簡約規則の適用の仕方で，答えが異なる可能性があるからである．

式を簡約するには，その部分式に簡約規則を適用することになる．簡約可能な部分式をリデックス (redex) と呼ぶ．一般には，同一のリデックスに対して適用できる簡約規則は複数あるし，与えられた式は複数のリデックスを含む．通常の関数型言語では，同一のリデックスに適用できる簡約規則はただ一つであるが，リデックスの選び方は問題になる．簡約を実行する際のリデックスの選び方を簡約戦略 (reduction strategy) と呼ぶ．関数型言語を考える上では，次の2種類の簡約戦略が基本的である．

(1) 作用順序 (applicative order)，あるいは最左最内規則 (leftmost-innermost rule)，値呼び出し (call by value)．

(2) 正規順序 (normal order)，あるいは最左規則 (leftmost rule)，名前呼び出し (call by name)．

以上の簡約戦略の厳密な定義は，対象とする言語を正確に定義してからでないと決められないし，文脈に応じて少しずつ違った意味にも使われる．ただし，正規順序による簡約では，もし答が存在すれば，常に簡約が停止し，その正規形が答になる点が重要である．作用順序による簡約は，一般にはこの性質をもたない．

以上，関数の概念がいかに計算と結び付くかを述べてきた．まとめると次のようになる．

(1) 関数の概念は，値の対応と計算規則の二つの側面をもつ．

(2) 計算規則としての関数を記述する基本操作は関数作用とラムダ束縛である．

(3) 計算は式の簡約により実現され，最も基本的な簡約規則が仮引数と実引数の結合を意味する β 規則である．

(4) 計算規則としての関数の意味を規定するためには，簡約戦略を決めなくてはならない．

3. 関数型言語の例

関数型言語の代表例として，Miranda, ML, FP を取り上げる．

Miranda は，SASL[25]，KRC[27] を経て，Turner により提案された言語[28),29)] である．SASL, KRC, Miranda の特長は，正規順序による計算に基づいた意味論をもつ点である．正規順序による意味論を採用する理由は二つある．一つは，関数型言語の利点とされている参照透明性を真に実現する意味論であること，もう一つは，より現実的に正規順序の計算法により高階関数，無限リストなどが統一的に扱え，言語の表現能力が高まる点である[28]．

次に，ML は LCF と呼ばれる定理検証システムの一部として考案された言語である[9]．LCF (Logic for Computable Functions) は，PPLAMBDA (Polymorphic Predicate λ-calculus) と呼ばれる理論体系を基に，プログラムの諸性質を検証するシステムである．ML は，LCF における証明を操作するための言語として設計され，その後，独立したプログラミング言語として，Cardelli ML[4]，Standard ML[21] と発展してきた．ML の最も重要な特長は，ポリモルフィ

ズムと型推論と呼ばれる機能である.

最後に FP は Backus が 1977 年のチューリング賞を受賞した際に行った講演[1] の中で提案された言語である. Backus のチューリング講演によると, それまでの計算機のアーキテクチャであるフォンノイマン式計算機の反省から関数的プログラムの概念を提案し, FP が生まれた. FP は, 関数型言語の先駆的な言語で, 以降の研究に大きな影響を与えている.

FP のプログラムは, 基本関数と関数を合成する演算から成り立ち, 変数を必要としない. たとえば, 内積を求める関数 IP は, Trans を次元が同じ二つのベクトル $(a_1, \cdots, a_n)$ と $(b_1, \cdots, b_n)$ からリスト $((a_1, b_1), \cdots, (a_n, b_n))$ を作る関数として,

IP=(Insert +)∘(ApplyToAll ×)∘Trans

と定義できる. あるいは, 省略として,

IP=(/ +)∘(α ×)∘Trans

と記述される. 詳しい説明は省略するが, 関数合成∘ Insert, ApplyToAll は, 関数を結合して新しい関数を作る演算で関数形式 (functional form) と呼ばれる. いくつかの関数形式を与えることで, 変数の概念なしに十分多くの関数が定義できることが示されている. 最近では, FP の拡張として, FL という言語が提案されている[2].

4. 関数型言語の諸機構

この章では, 関数型言語を調べる上で重要になるいくつかの基本的な考え方, 機能を概観する. より詳しい説明は, 本特集号の各解説にゆずる.

4.1 高階関数

関数型言語における高階関数の取り扱いについて考える. 高階関数とは, 関数を引数に取る関数, 関数を値とする関数, およびそれらの混合を意味する. 手続き型言語でも関数を引数にもつ関数を定義できるが, 一般には制限が強い. 高階関数の機能は, 関数型言語の分野でより積極的に利用されている.

3. で取り上げた SASL, KRC, Miranda では, 2 引数以上の関数は, 高階関数として定義される. たとえば, 2 引数関数 $f(x,y):A\times B\to C$ は, $\lambda x.(\lambda y.f(x,y)):A\to(B\to C)$ のように 1 引数の高階関数と解釈できる. この操作をカリー化 (Currying) と呼ぶ. カリー化を行うためには, 関数を値とする関数が定義できなくてはならない.

高階関数の例として, Miranda によるプログラムをあげる. 3 引数関数 foldr を定義する.

```
foldr op k [ ]=k
foldr op k (a:x)=op a (foldr op k x)
```

ただし, [ ] は空リスト, (a:x) を要素 a をリスト x の先頭に付け加えたリストを表す. また, 定義しようとする関数の仮引数部には, パターンが書け, 異ったパターンの引数をもつ複数個の等式を与えることで関数を定義できる. foldr の第 1 引数 op に適当な関数を与えると, さまざまなリスト処理用の関数が定義できる. たとえば, 数値のリストからその合計を求める関数 sum は,

```
sum=foldr (+)0
```

と定義できるし, リストの反転 reverse は,

```
reverse=foldr postfix [ ]
        where postfix a x=x++[a]
```

と定義できる. ただし, **where** は, postfix を局所的な関数として定義することを示す. 演算++は, リストの結合, [a] は a を要素とするリストを表す.

高階関数の利点の一つは, 機能をより細分化して, モジュール化を高める点にある. 上の例では, foldr は, sum, reverse などのリスト処理用関数に共通な制御構造を与えている. しかし, 高階関数を取り扱うための理論は, 一般に複雑になるし, 高階関数を利用したプログラムも複雑になる[8].

4.2 遅延評価

2. で述べた正規順序による簡約を利用すると, 新しいスタイルのプログラミングが可能になる. 次の prime n は, n 番目の素数を求める Miranda のプログラムである (説明のため構文を少し変えてある).

```
prime n=pick n (sieve (from 2))
where pick n (a : l)
    =if n=1 then a else pick (n-1) l
    from n=n:from (n+1)
    sieve (p:x)=p:sieve (filter x)
    where filter (n:x)=
        if n rem p=0 then filter x
        else n: (filter x)
```

この prime を通常の評価方法である作用順序で評価すると無限ループに陥る. (from 2) を評価しようとすると [2, 3, ⋯] の無限リストが生成されていくからである. Miranda では, 正規順序による評価法を採用して, 正しく prime が実行される.

正規順序による計算では, 与えられた式の部分式の先頭に着目して, 最も左にあるリデックスを優先して簡約を行う. その意味で最左規則とも呼ばれる (ただ

し，正規順序という言葉は，与えられた式が正規形をもてば常に正規形にたどり着ける簡約戦略一般に対して用いられる場合もある．最左規則はその一例である)．上の例で，prime 1 を最左規則で簡約すると，

```
prime 1
→pick 1 (sieve (from 2))
→pick 1 (sieve (2 : (from (2+1))))
→pick 1 (2 : (sieve (filter (from (2+1)))))
→if 1=1 then 2
  else pick(1−1) (sieve (filter (from (2+1))))
→2
```

となり，1番目の素数2が求まる．

最左規則は，関数を呼び出す際の引数の渡し方に着目して別の見方もできる．上の例の (pick 1 (sieve (from 2))) で関数 pick を呼び出す際，通常の評価方式では，実引数である1と (sieve (from 2)) を評価してからその値を pick に渡す (値呼び出し)．しかし，(sieve (from 2)) の引数 (from 2) を評価すると無限リストが生成されてしまう．一方，上の簡約では，(sieve (from 2)) は評価されずに pick に渡され，pick の第2引数がパターン $(a : l)$ なので，途中の値 (2 : sieve (filter (from (2+1)))) まで評価されている．最左規則による計算は，このように引数の渡し方に注目して，名前呼び出しと呼ぶこともある．

名前呼び出しの評価法には，注意が必要である．たとえば，$f\ x=x+x$ と f を定義して，(f (sqrt 4)) を名前呼び出しで評価すると，

```
f (sqrt 4)
→(sqrt 4)+(sqrt 4)
→2+(sqrt 4)
→2+2
→4
```

のように，(sqrt 4) が2度評価されてしまう．sqrt は平方根を求める関数で，＋に比べて実行実間が長いと仮定する．上の (sqrt 4)＋(sqrt 4) の二つの (sqrt 4) は同一の仮引数 x によって生成されたものだから，最初の (sqrt 4) を評価した時点で，2番目の (sqrt 4) もその値2に置き換える工夫ができる．この評価方式を必要呼び出し (call by need) と呼ぶ．必要呼び出しは，値が必要でない実引数は評価されないという名前呼び出しの利点と，値が必要な実引数はただ1度だけ評価される値呼び出しの利点とをもっている．必要呼び出しによる評価法を遅延評価 (lazy evaluation)[10] と呼ぶ．遅延評価の考え方は，さらに一般化すると本特集号 1.4 のストリーム計算へと続く．

必要呼び出しの機能はさらに拡張できる．たとえば

```
g x y=y+(sqrt x)
f z=(z 1)+(z 2)
```

と定義して，f (g 4) を必要呼び出しで評価すると次の計算が実行される．

```
f (g 4)
→(•1)+(•2)
   |_____|→(g 4)
→(1+(sqrt 4))+(•2)
                |→(g 4)
→3+(•2)
     |→(g 4)
→3+(2+(sqrt 4))
→3+4
→7
```

上の例では，(g 4) は確かに共有されているにもかかわらず，(sqrt 4) は2度評価されている．効率のためには，すべての式は，その中に含まれる変数に実引数が割り当てられた後には，高々1回だけ評価されることが望ましい．厳密な定義を与えることは困難だが，この評価法を完全遅延評価 (fully lazy evaluation) と呼ぶ．上のプログラムを

```
g' s y=y+s
g x=g' (sqrt x)
f z=(z 1)+(z 2)
```

のように書き換えると完全遅延が実現できる．たとえば，f (g 4) を必要呼び出しで評価すると次のようになる．

```
f (g 4)
→(•1)+(•2)
   |_____|→(g 4)
→(•1)+(•2)
   |_____|→(g'(sqrt 4))
→(1+•)+(•2)
   |_____|→(g'(sqrt 4))
→(1+•)+(•2)
   |_____|→(g' 2)
→3+(•2)
     |→(g' 2)
→3+(2+2)
→7
```

上の簡約では，(sqrt 4) が共有化され，1度だけ評価されている．

4.3 実現方法

高階関数および遅延評価の実現方法について考える．まず高階関数の機能を実現するためには，値としての関数をいかに実現するかが問題となる．たとえば，add＝$\lambda x\lambda y.\ x+y$ として，関数 add1＝add 1 を表現する方法を考える．一つは，仮引数xを1に置き換えた $\lambda y.\ 1+y$ を動的に作り出す方法が考えられる．あるいは，$\lambda x\lambda y.\ x+y$ と x の値1の組で add 1 を表す方法がある．遅延評価を実現する場合も同様である．たとえば，(add 2 (3+4)) を名前呼び出し，あるいは必要呼び出しで評価すると，仮引数 x と y にはそれぞれ2と (3+4) の式が割り当てられる．add を呼び出した直後の式をいかに実現するかが問題である．一つは，仮引数 x と y をそれぞれ2と (3+4) に置き換えた式 2+(3+4) を動的に生成させる方法がある．あるいは，式 $x+y$ と x の値2および y の値 (3+4) の組で，add 2 (3+4) を表す方法がある．

上のそれぞれの場合で，1番目の方法を用いるためには，仮引数を実引数に置き換える操作をいかに実現するかが問題である．単純に式を記号列で表現して，記号列上で置き換え操作を実行したのでは効率が悪い．解決策として，式をグラフで表し，式の置き換えはポインタの付け換えで実現する方法が考えられる．グラフを利用する方法は，さらに後で述べるコンビネータあるいはスーパ・コンビネータを用いる方法に拡張される．

2番目の方法は，SECD マシン[11),17)]と呼ばれるラムダ式の評価のための計算方式として知られている．この方法では，変数の置き換えを実際に行うのではなく，変数を含んだ式と，実引数の対応との組を計算の途中結果として生成する．評価しようとする式そのものは変数の置き換えによって変形されることはない．したがって，式はマシン・コードにまで落ちていてもかまわない．仮引数と実引数の対応を環境 (environment) と呼ぶ．変数を含んだ式は，ある環境の下で評価される．値としての関数も，ラムダ束縛されていない変数を含んだ関数の定義式（たとえば，$\lambda y.\ 1+y$）と環境との対で表される．関数閉包 (function closure) と呼ばれる．この考え方は，Lisp における関数引数の処理でも使われている．

以上の2種類の実現法は，いずれも仮引数と実引数の結合，すなわち β 規則の処理が中心である．しかし，ラムダ計算の理論によると，二つのコンビネータ $K=\lambda x\lambda y.\ x$ と $S=\lambda x\lambda y\lambda z.\ xz\ (yz)$ を用いると，ラムダ式からラムダ束縛を除去できることが知られている．Turner は，このコンビネータの原理を SASL の処理系に利用している[26)]．原理的には，コンビネータ K と S だけを使って，すべてのラムダ式からラムダ束縛を除去できるが，K と S を用いて変換した後の式はもとの式より複雑になる．また，β 規則による簡約に比べて，K と S による簡約のステップ数がふえる．解決法として，スーパ・コンビネータと呼ばれる考え方が提案されている[12)]．Turner の方法だと，コンビネータの種類を固定していたが，スーパ・コンビネータの方法では，与えられたプログラムから動的に新しいコンビネータを作る．スーパ・コンビネータ (supercombinator) とは，$\lambda x_1\lambda x_2\cdots\lambda x_n.\ M$ の形をしたラムダ式である．ただし，M はラムダ束縛を含まず，M 中のすべての変数は $x_1,\cdots,x_n$ に含まれているものとする．たとえば，$((\lambda x.\ (\lambda y.\ x+y)x)\ 4)$ は，

$$Y=\lambda x\lambda y.\ x+y$$

$$X=\lambda x.\ Yxx$$

の二つのスーパ・コンビネータ X と Y を定義すると $(X\ 4)$ と表現できる．式 $(X\ 4)$ は，X と Y の定義に従って簡約できる．X と Y の定義式は，$(X\ 4)$ の簡約中には現れず，簡約規則としてマシン・コードまで落とすことができる．与えられたプログラムをスーパ・コンビネータを用いた式に変換するアルゴリズムとして，lambda-lifting[14)] と呼ばれる方法が知られている．スーパ・コンビネータの解説としては，文献[23)]がすぐれている．

正規順序による評価方式の実現法について述べてきたが，いずれの方法も作用順序による評価方式に比べて効率が悪い．しかし，正規順序による計算でも，すべての式を一律に名前呼び出しで評価する必要はない．関数によっては，名前呼び出しによる評価結果と値呼び出しによる評価結果が同じ場合があり得る．プログラムを解析して，名前呼び出しと値呼び出しで結果が同じことが前もって分かれば，値呼び出しによる評価を実行してもよい．この解析をストリクトネス解析 (strictness analysis) と呼ぶ．一般に，未定義の値$\perp$に対する関数 f の値 $f(\perp)$ が再び未定義の値になる場合，f はストリクト (strict) であるという．未定義の値とは無限ループを意味する．M を評価すると無限ループになる式とすれば，M は未定義の値をもつと考える．f をストリクトな関数，M を未定義の値をもつ式とすれば，$f(M)$ を正規順序で評価しても無限ループになる．一般に，ストリクトな関数は，正

規順序で評価しても作用順序で評価しても結果は同じになる．与えられた関数がストリクトかどうかの解析がストリクト解析である．詳しくは，本特集号の **2.2** に解説がある．

4.4 ポリモルフィズムと型推論

関数型言語における型について考える．関数を定義するためには，定義域と値域を明示しなくてはならない．しかし，プログラム言語の観点からは，定義域と値域を固定すると使いにくい関数がある．たとえば，リストの第1要素を求める関数 hd は，複数の型をもつと考えられる．型 t のリストを表す型を (list t) とすると，hd を int 型のリストに作用させた場合は，hd の型は (list int)→int となるし，hd を char 型のリストに作用した場合は，(list char)→char になる．このように，複数の型をもつ関数をポリモルフィック関数 (polymorphic function) と呼ぶ．hd の型自身は，型の上を動く変数＊を用いて，(list ＊)→＊と表せる．ポリモルフィック関数の機能がもし無ければ，int 用の hd, char 用の hd を別々な関数として定義しなくてはならない．もう少し複雑な例として，**4.1** で定義した foldr もポリモルフィック関数で，

foldr : (＊→＊＊→＊＊)→＊＊→(list ＊)→＊＊

の型をもつ．ただし，＊と＊＊の二つの型変数が用いられている．ポリモルフィック関数の機能は，ML, Miranda らの多くの関数型言語で実現されている．

強い型付けを採用している言語では，すべての式はコンパイル時に決定する型をもち，型の矛盾はコンパイル時にチェックされる．一般に，ポリモルフィック関数の機能をもつ言語では，型チェックが複雑になる．たとえば，**4.1** の例で postfix は，

postfix : ＊＊＊→(list ＊＊＊)→(list ＊＊＊)

の型をもつから，reverse＝foldr postfix [ ] の型は，foldr の型の＊を＊＊＊に，＊＊ を (list ＊＊＊) に置き換えて，

reverse : (list ＊＊＊)→(list ＊＊＊)

となる．このように，式の型を自動的に導く機能が必要となる．式の型を推論する手法として，Milner のアルゴリズムW[20]が有名である．

型推論の機能は実用的な見地からも重要である．関数型言語では，プログラムは多数の小さな関数から段階的に組み立てられる．各関数を定義するたびに型を宣言するのはめんどうである．関数の定義式からその型を自動的に決定する機能が望まれる．したがって，ポリモルフィック関数と型推論の機能は，強い型付けをもつ言語に，型をもたない言語の良い所を取り入れる働きをしている．

型の理論は，現在，活発に議論されている分野である．本特集号 **1.5** で詳しく解説されている．

4.5 FP

FP は，ML, Miranda などとは違った発想で言語設計がなされている．FP は，基本関数と関数を合成して新しい関数を作るための演算から成り立っている．ML, Miranda などの言語との最大の違いは，変数をもたず，したがって，ラムダ束縛の機構をもたない点である．Backus は，FP のプログラムの性質を代数と捉え，その代数上でプログラムの諸性質を証明することを試みている．たとえば，

$[f_1, \cdots, f_n] \circ g = [f_1 \circ g, \cdots, f_n \circ g]$

$\alpha f \circ [g_1, \cdots, g_n] = [f \circ g_1, \cdots, f \circ g_n]$

$/f \circ [g_1, \cdots, g_n] = f \circ [g_1, /f \circ [g_2, \cdots, g_n]]$ $(n \geq 2)$

$/f \circ [g] \equiv g$

などが，FP のプログラム代数上で成り立つ．FP のプログラムに関する代数を構築できることは，FP ではラムダ束縛がないことと本質的に関連する．事実，ラムダ計算のモデルを代数系として定義すると複雑になることが知られている[3]．

一方，ラムダ計算のモデル理論の研究[15),16),24)]から導かれたカテゴリカル・コンビネータ (categorical combinator) と呼ばれる理論 CCL[6),7)] を応用すると FP と同様な言語が作れることが知られている．CCL における式を次のように定義する．

〈式〉::=〈定数〉|〈式〉∘〈式〉|
‘〈’〈式〉,〈式〉‘〉’|Λ (〈式〉)

ただし，〈定数〉には，特別な定数記号として，Id, Fst, Snd, App が常に含まれているものとする．CCL の式はすべて関数を表し，$F \circ G$, Id, $\langle F, G \rangle$, Fst, Snd, $\Lambda(F)$, App は，それぞれ F と G の合成，恒等関数，関数の組，1番目の要素を取る関数，2番目の要素を取る関数，F のカリー化，カリー化した関数の関数作用を表す．直観的には，次のような関数を表す．

$(F \circ G)(x) = F(G(x))$,

$\mathrm{Id}(x) = x$,

$\langle F, G \rangle(x) = \langle F(x), G(x) \rangle$,

$\mathrm{Fst}(\langle x, y \rangle) = x$,

$\mathrm{Snd}(\langle x, y \rangle) = y$,

$\Lambda(F)(x) = \lambda y.\, F(\langle x, y \rangle)$,

$\mathrm{App}(\langle f, x \rangle) = f(x)$.

CCL の式の間には，次のような等式が成り立つ．

$\mathrm{Id} \circ F = F$,

$\mathrm{Fst} \circ \langle F, G \rangle = F$,

$\mathrm{App} \circ \langle \Lambda(F), G \rangle = F \circ \langle \mathrm{Id}, G \rangle$.

これらの等式のいくつかを選び，左辺を右辺へ変換する簡約規則を定義すると，ラムダ計算の簡約が CCL の簡約でシミュレートできることが知られている．詳しくいうと，ラムダ式を CCL の式に変換する翻訳規則 Trans が存在して，

$M \to N \Leftrightarrow \mathrm{Trans}(M) \to \mathrm{Trans}(N)$

が成り立つ[30]．ここで，M, N はラムダ式，Trans(M), Trans(N) は M, N に対応する CCL の式である．CCL の特徴は，ラムダ計算と異なり変数をもたない点である．この性質は，コンビネータ理論にもみられる．CCL はコンビネータ理論の一種ともみなせる．CCL については，ラムダ計算，コンビネータ理論とともに本特集号 **1.2** で詳しく解説される．

CCL は，実際の関数型言語に応用されつつある．Cousineau らは，CCL に基づいた抽象機械 CAM (categorical abstract machine) を提案し，ML の処理系に利用している[5),22)]．井田は，Common Lisp の処理系に CCL を応用している[13)]．また，Lins は CCL を拡張し，SASL の処理系に応用している[18),19)]．

5. おわりに

関数型言語の基本的な考え方を述べてきた．プログラミング言語としての実際の関数型言語は，ここで述べなかった多くの機能を備えている．各文献を参照されたい．

関数型言語は，実際にソフトウェアの開発に使われてきた言語を段階的に拡張，修正して発展してきたというよりも，むしろプログラミング言語そのものの研究，ないしは理論的な計算機構の研究を基礎に提案されてきた言語のように思われる．関数型言語を理解するためには，基礎になる計算モデルの理解が重要になる．また，関数型言語を実際の大規模ソフトウェアの開発に利用するためには，今後さらに実験を積み重ねる必要があるだろうし，関数型言語に向く応用分野を開拓する必要があるだろう．

参考文献

1) Backus, J.: Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs, Comm. ACM, Vol. 21, No. 8, pp. 613-641 (1978).
2) Backus, J., Williams, J. H. and Wimmers. E. L.: FL Language Manual, Research Report IBM Almaden Research Center (1986).
3) Barendregt, H. P.: The Lambda Calculus—its Syntax and Semantics, revised edition, North-Holland, Amsterdam (1984).
4) Cardelli, L.: ML under Unix, Bell Laboratories, Murrary Hill, New Jersey (1982).
5) Cousineau, G., Curien, P.-L. and Mauny, M.: The Categorical Abstract Machine, LNCS 201, pp. 50-64, Springer-Verlag (1985).
6) Curien, P.-L.: Categorical Combinators, Sequential Algorithms and Functional Programming, Pitman, London (1986).
7) Curier, P.-L.: Categorical Combinators, Int. Control, Vol. 69, pp. 188-254 (1986).
8) Goguen, J.: Higher Order Functions Considered Unnecessary for Higher Order Programming, SRI-CSL-88-1, SRI International (1988).
9) Gorden, M. J., Milner, A. and Wadsworth, C. P.: Edinburgh LCF, LNCS 78, Springer-Verlag (1979).
10) Henderson, P. and Morris, J. M.: A Lazy Evaluator, Proc. 3rd Symp. Priciples of Programming Languages, pp. 95-103 (1976).
11) Henderson, P.: Functional Programming, Prentice Hall, London (1980).
12) Hughes, R. J. M.: Super-combinators, A New Implementation Method for Applicative Languages, Proc. Symp. Lisp and Functional Programming, pp. 1-10 (1982).
13) 井田哲雄：ラムダ計算とそのモデル「カルテシアン閉カテゴリ」による COMMON LISP の解釈と新しい処理系，コンピュータソフトウェア，Vol. 4, No. 4, pp. 33-44 (1987).
14) Johnsson, T.: Lambda-Lifting: Transforming Programs to Recursive Equations, LNCS 201, pp. 190-203, Springer-Verlag (1985).
15) Lambek, J.: From Lambda-Calculus to Cartesian Closed Categories, in H. B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism, Eds. Seldin and J. R, Hindley, pp. 375-402, Academic Press, New York/London (1980).
16) Lambek, J. and Scott, P. J.: Introduction to Higher Order Categorical Logic, Cambridge University Press (1986).
17) Landin, P. J.: The Next 700 Programming Languages, Comm. ACM, Vol. 9, No. 3, pp. 157-164 (1966).
18) Lins, R. D.: Categorical Multi-Combinators LNCS 274, pp. 60-79, Springer-Verlag (1987).
19) Lins. R. D.: On the Efficiency of Categorical Combinators as a Rewriting System, Software

—Practice & Experience, Vol. 17, No. 8, pp. 547-559 (1987).
20) Milner, R.: A Theory of Type Polymorphism in Programming, J. CSS, Vol. 17, No. 3, pp. 348-375 (1978).
21) Milner, R.: A Proposal for Standard ML, Proc. Symp. LISP and Functional Programming, pp. 184-197 (1984).
22) Mauny, M. and Suárez, A.: Implementing Functional Languages in the Categorical Abstract Machine, Proc. Symp. Lisp and Functional Programming, pp. 266-278 (1986).
23) Peyton Jones, S. L.: An Introduction to Fully Lazy Supercombinators, LNCS 242, pp. 176-206, Springer-Verlag (1986).
24) Scott, D. S.: Relating Theories of the λ-calculus, in H. B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism, Eds. J. P. Seldin and J. R. Hindley, pp. 403-450, Academic Press, New York/London (1980).
25) Turner, D. A.: SASL Language Manual, St. Andrews University (1976).
26) Turner, D. A.: A New Implementation Technique for Applicative Languages, Software—Practice and Experience, Vol. 9, pp, 31-49 (1979).
27) Turner, D. A.: Recursion Equations as Programming Language, in Functional Programming and its Applications, Eds. J. Darlington P. Henderson and D. A. Turner, pp. 1-28 (1982).
28) Turner, D. A.: Miranda: A Non-strict Functional Language with Polymorphic Types, LNCS 201, pp. 1-16, Springer-Verlag (1985).
29) Turner, D. A.: An Overview of Miranda. ACM SIGPLAN Notices, Vol. 21, No. 12, pp. 158-166 (1986).
30) Yokouchi, H.: Relationship between Lambda Calculus and Rewriting Systems for Categorical Combinators, Preprint, IBM Tokyo Research Laborotory (1988).

(昭和63年5月24日受付)