ICOT Technical Report: TR-162

TR-162

関数型言語の計算モデル

井田哲雄(理化学研究所)
田中二郎(ICOT)

March, 1986

©1986, ICOT

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

チュートリアル

関数型言語の計算モデル

井田 哲雄　田中 二郎

1 はじめに

プログラミング言語の分類の基準を関数型や論理型といった計算モデルに求め，プログラミングという知的活動を考えていこうとする試みは比較的最近のことである．従来から，プログラミング言語について語る場合には，記号処理言語や数値計算用言語といった応用による分類が使われてきた．それはそれなりに便利でもあり，妥当でもある．

しかし，ここでプログラミングを行なう上での思考方法に焦点を当ててみよう．すると，応用による分類では見てとれなかった，種々のプログラミング言語が持ついくつかの共通した像が浮び上がってくるように思われる．この像の一つが計算モデルと言われるものである．言いかえると，このような計算モデルのもつ共通の性質から，プログラミング言語はいくつかの範疇に分けて考えることができるのである．プログラミング・スタイル[†1]に関する議論も，つきつめると，計算のモデルにいきつく．そして，この関数型，論理型といった範疇がプログラミング言語の性質を，従来よりもはるかに豊富な内容で，しかも抽象的にとらえることを可能にすると信じられるのである．

本稿では，この範疇のうち，関数型言語を取り上げる．関数型言語は数学の基本概念である関数を出発点として作られた言語であると一義的には言うことができる．しかし，「写像」としての関数の概念も，構成的数学で用いる「操作」と「証明可能性」を基礎とした関数の概念[4, p. 41][†2]も，関数型言語の「関数」の概念を完全に包含してはいないと思われる．本稿では，計算モデルを通して，関数の概念がいかにプログラミング言語に取り入れられていくかをみようとするのであるが，まずは，次のことに留意することから始めたい．

つまり，上で述べたような「関数」に関する概念の差異が，計算機科学者の間で許容され，関数といった概念が表面上に全く現われてこないプログラミング言語に対しても，「関数型」であるとみなすのは，以下に述べるような動機を共有しているからであると思われる．

関数型言語において，関数を出発点とするそもそもの動機は，プログラミングを数学にみられるような緻密な枠組みの中に置こうとすることである．さらに，これは「放任すればとめどなく増大してしまうプログラムに内在する論理の複雑さ，むずかしさ」にたがをはめようとする試みであると解釈することができる．

以前よりプログラミングにおける論理の複雑さに対処する，規律ある(disciplined)思考のスタイルが求められてきた．1970年代の初めの構造化プログラミング(structured programming)は，この論理の複雑さに対し規律あるプログラミング作法によって対処しようとする試みであった．関数型プログラミングでは，論理の複雑さに対処するには，プログラミング言語のセマンティ

Computation Models of Functional Languages.
Tetsuo Ida，理化学研究所．
Jiro Tanaka，新世代コンピュータ技術開発機構．
コンピュータソフトウェア，Vol. 3, No. 2(1986)，pp. 2-18.
1986年2月28日受付．

†1 プログラミング・スタイルというと日本語の語感では好み，趣味，流行といったものが連想されるが，ここではプログラミングにおける思考の様式あるいは形式のことを意味する．

†2 [4, p. 41]は，文献[4]の41ページを示す．

ックス(semantics, 意味論)にまで，たちいって考える必要があるとする立場をとる．

そして，セマンティックスを与える計算のモデルは次のような性質を満たすことが要求される．

（1） プログラムそのものを数学的な対象として取り扱うことができること，

（2） プログラムという複雑な論理をもつシステムを構築する上で，システムの基本的構成要素と，構築のための道具立てとを提供すること，

（3） (2)の道具立ては，プログラムされるべき問題を分解し，抽象化し，そして階層的に再構成する思考過程を支援するものであること，

（4） (3)の階層的構成を可能にするために，各構成要素は入出力(定義域と値域)が完全に規定され，かつ副作用を及ぼすことのないものとして作られること．

以降では，このような性質をもつ抽象化されたプログラムの単位を関数と同一視する．そして，関数を要素として組み立てられる計算モデル(あるいは計算系という)を関数型計算系とよぶ．同様の考え方により，関数型言語，関数型プログラミング，関数型プログラム・パラダイム等の用語の使用も妥当性をもつものと考える．

2 関数型言語の計算モデルの種類

どのレベルで関数型言語を抽象化するか，あるいは計算対象のどの構造に注目するかによって，いくつかの計算モデルが考えられる．たとえば，

（1） ラムダ計算系(lambda calculus)

（2） コンビネータ論理(combinatory logic)

（3） 項書換え計算系(term rewriting system)

（4） CCC(Cartesian Closed Category)に基づく計算モデル

（5） 属性文法に基づく計算モデル

（6） SECDマシン

などが考えられる．

このうち(6)は現在の計算機に最も近い計算モデルである．抽象機械モデルといってもよい．SECDマシン[27]は関数型言語の実用的インプリメンテーションのために考案された計算のモデルであり，ISWIM[29], SASL[44], Lispkit Lisp[15]等の関数型言語のインプリメンテーションに使われてきた．SECDマシンに関しては，Burgeによる詳細な記述がある．興味ある読者は[5]を参照されたい．ちなみに，SECDマシンのSはstack, Eはenvironment, Cはcontrol, Dはdumpである．

この他に，ハードウェア・アーキテクチャの立場から，データフローモデル，リダクションモデルといった計算のモデルが考えられるが，この見方は本稿での我々の視点とは異なるので取り扱わない．

(1)～(5)の計算モデルは関数型言語とは別個にその理論体系が形成されてきたものである．関数型言語の研究が進展するにつれて，これら計算モデルと関数型言語との結びつきが明らかになってきている．

以下では筆者らが研究に携わっている分野の(1)～(3)について，入門的な解説を試みる．記述の方針としては，厳密な形式化よりもプログラミングとの直感的な結びつきを重視した．予備知識として，Lisp(ここではCommon Lisp[41]を用いる)による簡単なプログラミングの知識を仮定した．

(4)については，ごく最近の研究の動向を受けて，簡単にその可能性についてふれる．

(5)の属性文法計算モデルについては，最近活発な研究が行なわれており(たとえば[38][39][40])，適切な研究の展望が期待される．

3 ラムダ計算系

ラムダ計算系は1930年代にChurch, Curryらによって築かれた論理体系である[7]．Churchらの目的は，(1)関数の概念に考案を加え，(2)あわせて数学の基礎を与えること，であった．(2)の目標は失敗に終わったが，関数を計算の規則としてとらえる(1)の目標はその後独自の発展をとげ，今日ラムダ計算系とよばれる論理体系ができ上がった．1984年のBarendregtの著書The Lambda Calculus: Its Syntax and Semantics[3]はラムダ計算系に関する現在までの仕事の集大成であると考えられる．

一方，計算機科学の側では，1960年代に至って，Landinがラムダ計算系がプログラミング言語の操作的な意味を与えることを示した[28]．McCarthyのLisp[32]にもラムダ計算系の影響をみてとることができる．またScott-Straccheyの表示的意味論の発展はラムダ計算系と計算機科学の結びつきをさらに強いものとした[42]．

3.1 基本事項

ラムダ計算系は形式的には以下に定義される項の集合 Λ と項の間の二項関係 R で定まる論理体系 (Λ, R) と定義される．ラムダ計算系は，二項関係 R から誘導される同値関係を基にした等式論理系と考えることができるが，ここでは二項関係 R によって作られる計算の体系に焦点を当てる．

項はアルファベット A_L からなり，構文規則 S_L によって生成される表現である．

アルファベット A_L

ラムダ抽象子　λ

変数　$\nu_0, \nu_1, \cdots, \nu_n, \cdots$

括弧　(,)

(以下では表記上の便宜から $a, b, \cdots, z$ も変数として用いる．)

構文規則 S_L

$\langle$項$\rangle$ :: = $\nu_i \mid (\lambda\nu_i\ \langle$項$\rangle) \mid (\langle$項$\rangle\ \langle$項$\rangle)$

[例]　$(\lambda x\ (\lambda y\ x))$ は項である．

構文規則 S_L で生成される項は括弧や λ が多く読みにくいので，次の略記規則を導入する．

略記規則

(1)　$(\lambda\nu_{i_0}\ (\lambda\nu_{i_1}\ (\cdots(\lambda\nu_{i_k}\ M))))$ は $(\lambda\nu_{i_0}\nu_{i_1}\cdots\nu_{i_k}.\ M)$ と書く．(2)の規則によって，これは $\lambda\nu_{i_0}\nu_{i_1}\cdots\nu_{i_k}.\ M$ でもある．

(2)　項の最外側の括弧は省略することができる．

(3)　項の並びは左結合(left associative)とし，余分な括弧は省略することができる．つまり，

$$((\cdots(M_1M_2)\cdots)M_j) \equiv M_1M_2\cdots M_j$$

(4)　$\lambda\nu_{i_0}\nu_{i_1}\cdots\nu_{i_k}.$ の . の結合力は最も弱いものとし，

$$\lambda\nu_{i_0}\nu_{i_1}\cdots\nu_{i_k}.\ M_1\cdots M_j \equiv \lambda\nu_{i_0}\nu_{i_1}\cdots\nu_{i_k}.(M_1M_2\cdots M_j)$$

とする．

ここで $\equiv$ は表記上，同一のものを示すときに用いる．

注意事項

(1)　上の規則で生成される項のうち，$\lambda\nu.M$ の形をしたものを λ-項とよぶ．

(2)　任意の項 M の部分項とは，構文規則 S_L を繰り返し適用して，M を構成する過程で作られる項のことをいう．

[例]　$\lambda x\,(\lambda y\,(x\ y))$ の部分項は $\lambda x\ (\lambda y\ (x\ y))$, $(\lambda y\ (x\ y))$, $(x\ y)$, x, y である．ここでは，自分自身も部分項として含めることとする．

(3)　$\lambda\nu.M$ を Lisp の (LAMBDA (ν) M) と対応させ，$\lambda\nu.M$ はフォーマルパラメータ ν をもった(名前なし)関数であると考えると，直感的に理解しやすい．

(4)　$\lambda\nu.M$ の $\lambda\nu$ の部分(これは部分項ではない)を変数束縛部，M を($\lambda\nu.M$ の)本体部(これは部分項である)とよぶ．

(5)　上の直感的理解に従って，λ-項を関数ともよぶこととする．

(6)　自由変数，束縛変数の概念を以下では使うが，これらは Lisp の対応する概念から類推可能であろう．

[例]　$\lambda x.(\lambda y.\underline{x}y)$ の下線をひいた変数 x は，$\lambda y.xy$ において自由であり，$\lambda x.(\lambda y.xy)$ においては束縛されている．

二項関係 R

R としては，ここでは計算過程を記述するのに意味のある二項関係をとる[13]．プログラミング言語の計算モデルとして重要である二項関係は次の β-リダクションおよび δ-リダクションとよばれるものである．

この他に，ラムダ計算系では α-変換，η-変換とよばれる二項関係がある．(歴史的な理由で「リダクション」とはいわず「変換」という．)

α-変換とは，束縛変数名を組織的に異なった変数名に変えてえられる項ともとの項との間に成立する二項関係のことである．たとえば，$\lambda x.(\lambda y.x)$ を α-変換すると $\lambda a.(\lambda b.a)$ になる．プログラミング言語でいうと，変数の名前を組織的に変えて，新しいプログラムを作ることに相当する．このような変換は，言語の処理系では容易に行なえるものであり，計算モデルの領域にまで，もちこむまでもないと考えられる．本稿では，束縛変数名のみが異なる λ-項は同一の項とみなし，以降では α-変換はリダクションとして，陽には扱わない．

η-変換とは $\lambda\nu.M\nu$(ただし M の中で ν は自由変数でない)を M とする変換のことで，ラムダ計算系の能力を考察したり，λ-項を変形し，プログラムの最適化を考える場合に，重要となる．

$\lambda\nu.M\nu$ を M に変換する規則 $\lambda\nu.M\nu \rightarrow M$ を η-規則ともいう．本稿の以下の議論では η-変換はあらわれてこ

---

†3　形式的な取り扱いは[3, pp. 50-51]を参照されたい．

ない.

β-リダクション

β-リダクションは形式的には,

$$\beta = [(P,Q)]P \equiv (\lambda\nu.M)N, Q \equiv M[\nu:=N] \tag{3.1}$$

と定義できる. ここで, $M[\nu:=N]$ は M の中に出現する自由変数 ν をすべて N で置き換えた表現である.

(3.1)式は,

$$(\lambda\nu.M)N \xrightarrow[\beta]{} M[\nu:=N] \tag{3.2}$$

とも書く. 直感的には, 関数 $\lambda\nu.M$ が N に作用(apply)して, $M[\nu:=N]$ の表現へとリダクションされると考える.

[例] $M \equiv if\ (eq\ n\ 0)\ m\ (f\ n\ (mod\ m\ n))$

としたとき,

$$M[n:=0] \equiv if\ (eq\ 0\ 0)\ m\ (f\ 0\ (mod\ m\ 0)) \tag{3.3}$$

であるので,

$$(\lambda n.M)0 \xrightarrow[\beta]{} if\ (eq\ 0\ 0)\ m\ (f\ 0\ (mod\ m\ 0)) \tag{3.4}$$

である.

一般に $P \xrightarrow[\beta]{} Q$ であるとは, P がβ-リダクションという操作によって, Q へと変換される(あるいは計算される)ことを意味している.

[例] 整数の四則演算をラムダ計算系で次のように行なうことができる.

まず, 整数を λ-項によって表現する. (この数をChurch の numeral という.)

$0 \equiv \lambda fx.x$

$n \equiv \lambda fx.f^n x$, $n > 0$ (ただし, $f^n x \equiv f(f^{n-1}x)$ である)

次に, 加算を *succ*(1 を加える関数)から作る. *succ* を次のように定義する.

$$succ \equiv \lambda abc.b(abc)$$

確かに, *succ* が 1 を加える関数になっていることを以下にみてみよう.

$$succ\ n \equiv (\lambda abc.b(abc))(\lambda fx.f^n x)$$
$$\xrightarrow[\beta]{} \lambda bc.b((\lambda fx.f^n x)bc)$$
$$\xrightarrow[\beta]{} \lambda bc.b((\lambda x.b^n x)c)$$
$$\xrightarrow[\beta]{} \lambda bc.b(b^n c) \equiv \lambda fx.f(f^n x) \tag{3.5}$$

最後の ≡ は α-変換による.

つまり $succ\ n \xrightarrow[\beta]{} \cdots \xrightarrow[\beta]{} n+1$ であることが示された.

plus m n は, n に *succ* を m 回作用させたものであるから, $plus \equiv m\ succ\ n$ となる. (計算して確かめよ)

他の四則演算 *sub*, *mult*, *div*(各々減, 乗, 除算)も, 同様の考え方で, λ-項で定義することができる.

[練習問題] 乗算 *mult* を λ-項で定義せよ. (答: $\lambda xy.(\lambda abc.a(bc))xy$)

δ-リダクション

上の例でみたように, 普通の計算で用いるデータや基本処理(たとえば数や四則演算)は, データも演算も λ-項で表現することにより, ラムダ計算系の中で, 実現することができる.

しかし, 実用上は, 基本となるデータや処理は, ラムダ計算系の組込みの λ-項と, リダクション規則として与える. この項を **δ-定数**, 規則を **δ-規則**とよび, δ-規則によって実現される二項関係を **δ-リダクション**とよぶ.

δ-リダクションを β-リダクションの場合と同様の表記を用いて, $\xrightarrow[\delta]{}$ と書く.

[例 1] 0, 1, … を δ-定数とし, *succ* を δ-規則とすることもできる.

[例 2] T, F を真理値(各々)真, 偽を表わす δ-定数とする. δ-規則として, 次のものを導入する.

$$eq\ m\ n \xrightarrow[\delta]{} \begin{cases} T & m \text{ と } n \text{ が数値として等しい時} \\ F & m \text{ と } n \text{ が数値として等しくない時} \end{cases} \tag{3.6}$$

$$if\ p\ q\ r \xrightarrow[\delta]{} \begin{cases} q & p=T \text{ の時} \\ r & p=F \text{ の時} \end{cases} \tag{3.7}$$

[例 1], [例 2]で導入した δ-定数, δ-規則は以下の議論では, ラムダ計算系にあらかじめ備わっていることとする.

δ-リダクションの考えを用いると,

$$\text{式}(3.3) \equiv M[n:=0]$$
$$\equiv if\ (eq\ 0\ 0)\ m\ (f\ 0\ (mod\ m\ 0))$$
$$\xrightarrow[\delta]{} if\ T\ m\ (f\ 0\ (mod\ m\ 0))$$
$$\xrightarrow[\delta]{} m \tag{3.8}$$

となる.

以上の β-リダクションと δ-リダクションを合わせて, $R = \beta \cup \delta$ と書く. また, $\rightarrow_R$ の 0 回以上の繰り返し(反射推移閉包)を $\rightarrow\!\!>_R$ と書く. 等号 $=_R$ を $\rightarrow\!\!>_R$ から誘導される同値関係とする. 文脈から明らかな時は添字を省略して, $\rightarrow_R$, $\rightarrow\!\!>_R$, $=_R$ を各々 $\rightarrow$, $\rightarrow\!\!>$, $=$ と書く. 同様に R-リダクションを省略して, リダクションという.

[例] 式(3.4)と(3.8)をつなげると,

$$(\lambda n.\, M)0 \longrightarrow m \qquad (3.9)$$

となる．これが，関数 $\lambda n.\, M$ を 0 に作用することによって行なわれる(ラムダ計算系における)計算の意味である．

リデックスと正規形

ある項 M がリダクション可能であるとは，項 $N(\equiv M)$ が存在して，$M \longrightarrow N$ となることをいう．リダクション可能な部分項をリデックス(redex)という．リデックスを部分項としてもたない項を正規形(normal form)という．

[例 1] $\lambda fx.\, f^n x$ は正規形である．

$succ\ n \equiv \underline{(\lambda abc.\, b(abc))(\lambda f.\, f^n x)}$ はリデックス(下線の部分)をもつ．$succ\ n$ はリダクションを繰り返すことにより，正規形 $n+1$ となることを式(3.5)でみた．

[例 2] $(\lambda y.(\lambda x.\, y)2)3$ はリデックス $(\lambda x.\, y)2$ と $(\lambda y.(\lambda x.\, y)2)3$ をもつ．これをリダクションすると,

$$(\lambda y.(\lambda x.\, y)2)3 \xrightarrow{}_{\beta} (\lambda y.\, y)\,3 \xrightarrow{}_{\beta} 3 \qquad (3.10)$$

となる．3 はリデックスを含まないので正規形である．

3.2 再帰(recursion)と θ コンビネータ

関数型言語では再帰呼び出しは最も基本的な制御構造である．繰り返し(iteration)も再帰によって実現される．ラムダ計算系では再帰呼び出しは θ コンビネータ[14]を用いて実現される．

θ は $\theta f \longrightarrow f(\theta f)$ というリダクション規則をもつコンビネータである．このような θ として，$\theta \equiv (\lambda xy.\, y(xxy))(\lambda xy.\, y(xxy))$ をとることができる．

なお，自由変数を含まない λ-項をコンビネータとよぶ．

[練習問題] $\theta f \longrightarrow f(\theta f)$ を確かめよ．

[例] ラムダ計算系において gcd(最大公約数)を求める関数を定義する．

まず，Lisp を用いて，最大公約数を求めるアルゴリズムを書いてみよう．

```
(DEFUN GCD (M N) (IF (= N 0)
    M (GCD N (MOD M N))))          (3.11)
```

Lisp の DEFUN は GCD という関数名と関数,

```
(LAMBDA (M N) (IF (= N 0)
    M (GCD N (MOD M N))))          (3.12)
```

を(再帰的に)結びつける役割を果たす．

これをラムダ計算系に翻訳するには，IF, =, MOD を，各々 δ-規則 if, eq, mod に対応させる．ついで式(3.12)をそのまま λ-項に翻訳しようとすると[15],

$$gcd \equiv \lambda mn.\, if\ (eq\ n\ 0)\ m\ (gcd\ n\ (mod\ m\ n)) \qquad (3.13)$$

となるが，これでは右辺の gcd が自由変数として現われてしまう．gcd はこれから定義しようとする関数であるから，これを δ-定数として扱うこともできない．これを避け，gcd が再帰呼び出しとして機能するようにするには，θ を用いて,

$$gcd \equiv \theta(\lambda fmn.\, if\ (eq\ n\ 0)\ m\ (f\ n\ (mod\ m\ n))) \qquad (3.14)$$

とすればよい．

ここで，$gcd\ 12\ 8$ をリダクションしてみよう．

$$gcd\ 12\ 8 \equiv \theta(\lambda fmn.\, if\ (eq\ n\ 0)\ m\ (f\ n\ (mod\ m\ n)))\ 12\ 8 \qquad (3.14.1)$$

$$\xrightarrow{}_{\beta} (\lambda fmn.\, if\ (eq\ n\ 0)\ m\ (f\ n\ (mod\ m\ n)))\ (\theta G)\ 12\ 8 \qquad (3.14.2)$$

ここで,

$$G \equiv \lambda fmn.\, if\ (eq\ n\ 0)\ m\ (f\ n\ (mod\ m\ n)) \qquad (3.14.3)$$

とし，$\theta G \xrightarrow{}_{\beta} G\ (\theta G)$ を用いた．

$$\xrightarrow{}_{\beta} if\ (eq\ 8\ 0)\ 12\ ((\theta G)\ 8\ (mod\ 12\ 8)) \qquad (3.14.4)$$

$$\xrightarrow{}_{\delta} if\ F\ 12\ ((\theta G)\ 8\ (mod\ 12\ 8)) \qquad (3.14.5)$$

$$\xrightarrow{}_{\delta} (\theta G)\ 8\ (mod\ 12\ 8) \qquad (3.14.6)$$

$$\xrightarrow{}_{\delta} (\theta G)\ 8\ 4 \qquad (3.14.7)$$

$$\xrightarrow{}_{\beta} G\ (\theta G)\ 8\ 4 \qquad (3.14.8)$$

$$\xrightarrow{}_{\beta} if\ (eq\ 4\ 0)\ 8\ ((\theta G)\ 4\ (mod\ 8\ 4)) \qquad (3.14.9)$$

$$\xrightarrow{}_{\delta} (\theta G)\ 4\ (mod\ 8\ 4) \qquad (3.14.10)$$

$$\xrightarrow{}_{\delta} (\theta G)\ 4\ 0 \qquad (3.14.11)$$

$$\xrightarrow{}_{\beta} G\ (\theta G)\ 4\ 0 \qquad (3.14.12)$$

---

†4 従来，再帰を実現するためのコンビネータとして $Y \equiv \lambda f.(\lambda x.\, f(xx))(\lambda x.\, f(xx))$ が用いられていたが，本稿ではリダクションを問題としているので，任意の項 f に対して，$\theta f \xrightarrow{}_{\beta} f(\theta f)$ の成立する θ を用いた．Y では $f(Yf) =_\beta Yf$ であるが，一方から他方へのリダクションは行なえない．

†5 Lisp のプログラムの中にある(LAMBDA (X1 … XN) M)を $\lambda x_1 \cdots x_n.\, M$ のように翻訳し，ラムダ計算系でリダクションしても，もし Lisp で答が求まるならば，同一の答がラムダ計算系でも求まる．しかし，逆は正しくない．$\lambda x_1 \cdots x_n.\, M$ のほうが Lisp の(LAMBDA (X1 … XN) M)よりも，能力が高いのである．この点については，3.3 で論じる．

$\xrightarrow[\beta]{}$ *if* (*eq* 0 0) 4 ((*θG*) 0 (*mod* 4 0)) (3.14.13)

$\xrightarrow[\delta]{}$ 4 (3.14.14)

上のリダクションの過程で *θG* を *gcd* と書き直して考えてみると(たとえば式(3.14.6)), *gcd* を求める操作があたかも *gcd* という関数が *gcd* 自身を再帰的によびながら実行されていくようにみることができる.

3.3 高階関数とカリー化

3.1 で $\lambda\nu.M$ は Lisp の関数(LAMBDA (ν) M)と対応づけて考えることができると述べた. 同様の類推から, $\lambda\nu_1\cdots\nu_k.M$ を(LAMBDA (ν_1 ν_2 ⋯ ν_k) M)と対応させることができるが, この2つの表現は次の点で相違がある.

ラムダ計算系では,

$$(\lambda\nu_1\cdots\nu_k.M)N \xrightarrow[\beta]{} \lambda\nu_2\cdots\nu_k.M[\nu_1:=N]$$

であって, このリダクションの結果, 新たな関数 $\lambda\nu_2\cdots\nu_k.M[\nu_1:=N]$ がえられる. $\lambda\nu_1\cdots\nu_k.M$ は1つのパラメータに作用することによって, 新たな関数を作り出すという点で, **高階関数**(higher-order function)である.

これに対し, 対応する Lisp のプログラム((LAMBDA (ν_1 ⋯ ν_k) M) N)は, パラメータ不足のエラーになる.

日常使われる関数(たとえば四則演算)は, 要求されるパラメータの数(これを項数(arity)とよぶ)が決まっている. + を二項関数として導入しよう. そして,

$$plus \equiv \lambda\nu_1\nu_2.+(\nu_1,\nu_2)$$

とする[16]. *plus* は上で述べた意味で高階関数である. というのは, *plus* 1 は *plus* 1⟶$\lambda\nu_2.+(1,\nu_2)$ となり, 1 を加える関数となるからである. *plus* はもともとの二項関数 + に変数束縛部「$\lambda\nu_1\nu_2$」をつけ加えることによって, 項数1の高階関数となったと考えられる. このようにして, 項数 $n>1$ の関数を項数1の高階関数にすることをカリー(Curry)化とよぶ.

高階関数とカリー化(両者は上にみたように表裏一体であるが)によって, 新しい(関数型)プログラミング・スタイルが生まれる.

[例] m から n ($n\geq m$) の数をすべて掛け合わせた数を求める関数 *product* を考える.

$$product \equiv \theta(\lambda fmn.\ if\ (eq\ m\ n)\ n\ (mult\ m\ (f\ (add1\ m)\ n)))$$

$$add1 \equiv (\lambda xy.+(x,y))\ 1$$

product は高階関数である. たとえば, $fac \equiv product\ 1$ とすると, *fac* は階乗を計算する関数となる. つまり, *product* を1に作用することにより, *fac* という関数がえられる.

3.4 リダクションの戦略

ここでは, 直感的な説明を行なう. 形式的な定義は[3, pp. 325-327]を参照されたい. リダクションの**戦略**(strategy)とは, リダクションの仕方のことである. 一般には項の中にはリデックスが複数個存在する. たとえば, 式(3.14.4)に現われる表現,

$$if\ \underbrace{(eq\ 8\ 0)}_{a}\ 12\ (\underbrace{(\theta G)}_{b}\ 8\ \underbrace{(mod\ 12\ 8)}_{c}) \tag{3.15}$$

では, 下線 *a*, *b*, *c* がリデックスになっている.

したがって, どのような順序でもって, リデックスをリダクションしていくかが問題になり, 戦略に関するいくつかの性質がそれに答えてくれる. 問題を正規形の存在の有無と正規性の唯一性に関することに分けて考えよう.

正規戦略

まず, 任意の項に対して, 正規形が存在するとは限らないことに注意する. 任意の項に対して, それが正規形をもつかどうか, つまり正規形へとリダクションされるかどうかを判定する問題は, 一般に「停止問題」として知られ, 解決不能(unsolvable)である. したがって, 戦略としては, もし正規形が存在すれば必ずその正規形がえられるような戦略が望まれる. このような戦略を**正規戦略**(normalizing strategy)という. 正規戦略には, **最左戦略**(leftmost reduction strategy), Gross-Knuth 戦略などがある. 最左戦略とは(部分項の最も左にある文字の位置を基準にして)最も左にあるリデックスを常にリダクションしていく戦略のことである. Gross-Knuth 戦略については[3, p. 330]を参照されたい.

[例1] $(\lambda y.\underbrace{(\lambda x.y)2}_{b})\underbrace{3}_{a}$ は $\longrightarrow_a$ と $\longrightarrow_b$ のリデックスをもつが, $\longrightarrow_a$ が最左リデックスであり, 最左戦略では,

$$(\lambda y.(\lambda x.y)2)3 \xrightarrow[\beta]{} (\lambda x.3)2 \xrightarrow[\beta]{} 3$$

となる. (3.10)のリダクションの仕方とは異なることに注意.

[例2] *gcd* 12 8 を最左戦略でリダクションする.

gcd 12 8

$\xrightarrow[\beta]{}$ *if* (*eq* 8 0) 12 ((*θG*) 8 (*mod* 12 8))

16 $+(\nu_1,\nu_2)$ は今までに述べたラムダ計算系の構文規則にはないが二項関数を示すために本節でのみ便宜的に使用する.

$\xrightarrow[\delta]{}$ *if F* 12 ((*θG*) 8 (*mod* 12 8))

$\xrightarrow[\delta]{}$ (*θG*) 8 (*mod* 12 8)

$\xrightarrow[\beta]{}$> *G* (*θG*) 8 (*mod* 12 8)

$\xrightarrow[\beta]{}$> *if* (*eq* (*mod* 12 8) 0) 8 ((*θG*)(*mod* 12 8)
(*mod* 8 (*mod* 12 8)))

$\xrightarrow[\delta]{}$ *if* (*eq* 4 0) 8 ((*θG*)(*mod* 12 8)
(*mod* 8 (*mod* 12 8)))

$\xrightarrow[\delta]{}$> (*θG*)(*mod* 12 8)(*mod* 8 (*mod* 12 8))

$\xrightarrow[\beta]{}$> *G* (*θG*)(*mod* 12 8)(*mod* 8 (*mod* 12 8))

$\xrightarrow[\beta]{}$> *if* (*eq* (*mod* 8 (*mod* 12 8)) 0)(*mod* 12 8)
((*θG*)(*mod* 8 (*mod* 12 8))(*mod* (*mod* 12 8)
(*mod* 8 (*mod* 12 8))))

$\xrightarrow[\delta]{}$> *if T* (*mod* 12 8)((*θG*)(*mod* 8 (*mod* 12 8))
(*mod* (*mod* 12 8)(*mod* 8 (*mod* 12 8))))

$\xrightarrow[\delta]{}$ *mod* 12 8

$\xrightarrow[\delta]{}$ 4 (3.16)

ここで Lisp におけるリダクション戦略についてふれておこう．Lisp では最内側の表現からリダクションを行なう戦略をとっている．2個以上の引数をもつ関数の評価(たとえば(F M1 Mn)としよう)では，M1, …, Mn の評価が F の作用に先立たねばならない．このため，Lisp では(IF M1 M2 M3)のように，第1引数 M1 の評価結果に依存して，M2 か M3 の評価を行なうような「関数」は，特別形式(special form)として他の関数とは異なった扱いをしなくてはならない．また，Lisp のこのような戦略は正規戦略でないため，正規形が存在してもえられないことが起こりうる．この点に関しては 3.5 で再び議論する．

合流性(confluent property)

正規形の一意性に関しては，関係 R の合流性がいえると，(R に対する)正規形が存在するならばそれは一意に定まることが証明される．

まず合流性を定義する．

関係 R が合流性をもつとは，$\longrightarrow_R$ が次の関係を満たすことをいう[17]．合流性を **Church-Rosser 性**ともいう．

$\forall M, M_1, M_2 \in A$ に対して，

$M \longrightarrow_R M_1$ かつ $M \longrightarrow_R M_2$ ならば，

$\exists M_3$, $M_1 \longrightarrow_R M_3$ かつ $M_2 \longrightarrow_R M_3$

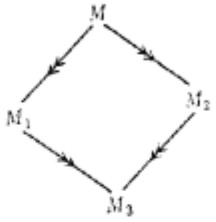

図 1 項の合流性

である．これは，図1によって容易に理解することができる．

つまり，M に含まれる異なったリデックスをリダクションして，相異なる項 M_1, M_2 をえても，さらに M_1 と M_2 のリダクションを行なうことにより，必ず同一の項 M_3 へと合流することを意味している．

ここでは $R = \beta \cup \delta$ であったことに注意をしよう．

R の合流性は β が合流性をもつこと，δ が合流性をもつこと(δ-規則としては，合流性をもつもののみを許す．今までの δ-規則はみな合流性を満たしている)，β と δ が可換であること($\xrightarrow[\beta]{}$ と $\xrightarrow[\delta]{}$ の順序をかえてもよい)を利用して，示すことができる．

[例] *gcd* 12 8 を用いて，合流性を考察する．

(3.15)には3つのリデックス $\underline{eq\ 8\ 0}_a$, $\underline{\theta G}_b$, $\underline{mod\ 12\ 8}_c$ がある．

(3.14)ではリデックス —$_a$ をリダクションして計算をすすめたが，今度はリデックス —$_b$ をリダクションすることによって，この流れから分岐してみよう．すると，

式(3.15)$\xrightarrow[\beta]{}$> *if* (*eq* 8 0) 12 (*if* (*eq* $\underline{(mod\ 12\ 8)}$ 0) 8
((*θG*) $\underline{(mod\ 12\ 8)}$ $\underline{(mod\ 8\ (mod\ 12\ 8))}$)) (3.17)

次に式(3.17)で下線部を次々に δ-リダクションして，次の項をえる．

式(3.17)$\xrightarrow[\delta]{}$> *if* $\underline{(eq\ 8\ 0)}$ 12 (*if* $\underline{(eq\ 4\ 0)}$ 8
((*θG*) 4 0)) (3.18)

さらに(3.18)の下線部を次々に δ-リダクションして，

式(3.18)$\xrightarrow[\delta]{}$> $\underline{if\ F\ 12\ (if\ F\ 8\ ((\theta G)\ 4\ 0))}$ (3.19)

ここで，*if* に関する δ-リダクションを式(3.19)に行ない，

式(3.19)$\xrightarrow[\delta]{}$> (*θG*) 4 0 (3.20)

となり，(3.14.11)の式に合流した．

ここで，次のことに注意したい．合流性は，適切なリダクションを行なえばいったん別々になった項が合流することを意味しているのであって，勝手な戦略をとって

17 ここでは，ラムダ計算系における二項関係 R について考えるが，任意のリダクションの関係について合流性は定義される．

も合流性が保証されるわけではない．たとえば，式(3.15)の $\longrightarrow_\beta$ に相当する部分を繰り返しリダクションしていても，項はどんどん長くなるばかりで，式(3.14.11)には決して合流しない．

3.5 最左戦略と遅延評価

正規戦略は従来の手続き型言語やLispには備わっていない次のような能力をプログラミングに付け加えることができる．すなわち，(1)計算の途中結果が無意味であってもその途中結果を以後の計算に用いないならば，計算全体として意味をもたせることができること，(2)途中結果が無限計算の可能性をもっていても，最終結果をえるのに，そのうちの有限な部分しか用いないならば，答が求まることである．

(1)については，次の(人為的な)例を考えてみよう．

[例] $(\lambda xy.x)\,1\,\underline{((\lambda x.xx)(\lambda x.xx))}$ (3.21)

のリダクション．

上の項には $\longrightarrow_\alpha$ と $\longrightarrow_\beta$ のリデックスがある．最左リデックスは $\longrightarrow_\alpha$ なので，これをリダクションすれば1をえる．一方，$\longrightarrow_\beta$ をリダクションして，この項の正規形を求めようとすると，無限のリダクションを繰り返してしまう．しかし，式(3.21)の正規形は1であり，項(3.21)自体は意味をもつ．

(2)の具体例として，自然数列のリストを表わすプログラムを考えてみよう．Lisp風にまず書いてみると次のようになる．

```
(DEFUN NATSEQ (X) (CONS X
    (NATSEQ (1+ X))))          (3.22)
```

(NATSEQ 1)が1から始まる自然数列のリストを与えると考え，その2番目の要素をとり出すプログラム，

```
(CADR (NATSEQ 1))              (3.23)
```

を実行してみよう．答として，2を期待する．しかし，Lisp処理系は(NATSEQ 1)をまず計算しようとして，無限計算を始め，処理系の資源を使い果たしてしまう．これはLisp処理系が3.4でみた正規戦略を実現していないからである．

最近の関数型言語の多くは最左戦略を採用しており，上のような計算を問題なくこなすことができる．

これをラムダ計算系で考えてみよう．式(3.22)は再帰的であるから，θ を用いて，

$$natseq \equiv \theta(\lambda fx.\,cons\ x\ (f\ (add1\ x))) \quad (3.24)$$

と表わせる．ここで，*cons*, *add1* は式(3.22)のLispの関数，CONS, 1+ に対応する δ-規則である．

ラムダ計算系では，Lispのようなリストという特別なデータ構造をもちこまなくても，関数 *cons* によってリストを表現できる．Lispのリスト(1 2 3)は *cons* を用いて *cons* 1 (*cons* 2 (*cons* 3 *nil*)) という項で表現する．*nil* はLispのNILに対応する空リストを示す δ-定数である．このようにして作られるリストに対して，リストの参照を行なう δ-規則 *hd*, *tl* を導入する．*hd*, *tl* は各々LispのCAR, CDRに対応し，次のようなリダクション規則で定義される．

$$hd\ (cons\ x\ y) \xrightarrow[\delta]{} x$$

$$tl\ (cons\ x\ y) \xrightarrow[\delta]{} y$$

これだけの準備で，最左戦略により自然数列の第2番目の要素を求めてみよう．

(CADR x)は *hd* (*tl* x) に対応するから，式(3.23)は *hd* (*tl* (*natseq* 1)) になる．

natseq 1は，

$$natseq\ 1 \equiv \theta(\lambda fx.(cons\ x\ (f\ (add1\ x))))\ 1$$
$$\longrightarrow (\lambda x.(cons\ x\ (natseq\ (add1\ x))))\ 1$$
$$\longrightarrow cons\ 1\ (natseq\ (add1\ 1))$$

したがって，

$$hd\ (tl\ (natseq\ 1)) \longrightarrow hd\ (natseq\ (add1\ 1))$$
$$\longrightarrow hd\ (cons\ (add1\ 1)(natseq\ (add1\ (add1\ 1))))$$
$$\longrightarrow (add1\ 1)$$
$$\longrightarrow 2$$

となる．

上の例でわかるように，最左戦略では *natseq n* のリダクションは必要なところまでしか行なわれない．(*add1 n*)の評価も，このリデックスが最も左に移るまで行なわれない．

リデックスのリダクションは真にそれが必要になるまで行なわないことを遅延評価(lazy evaluation)というが，普通は上に述べたような最左戦略下におけるリストの処理についての評価(リダクション)法のことを指している．

3.6 最左戦略のインプリメント

3.5にみたように最左戦略は，計算能力の点では，従来の内側から計算していく戦略よりもすぐれているが，インプリメントが従来の方法よりも複雑で，実行速度が

遅いという難点もあった．しかし，最近では最左戦略のインプリメント技術も向上し，従来の方法に劣らない実行能率をえるようになってきている．

最左戦略のインプリメント法としては次の3つが基本的である．

（1） 閉包を用いる方法．

（2） 項をグラフで表現し，リダクションをグラフのリダクションで行なう方法．

（3） 次節で述べるコンビネータを用いる方法．コンビネータで表現された項をグラフで表現するならば，原理的には(2)と同じになる．

方法(1)はβ-リダクションを閉包(closure)というものを用いて実現するものである．$(\lambda x.M)N \xrightarrow{}_{\beta} M[x:=N]$において，$N$を$M$の中の$x$の出現位置に代入することはせず，変数名とその変数に代入されるべき項の対(ここではxとN)を作る．この対の集合を環境(environment)とよんでいる．そして，$M[x:=N]$を関数の本体部Mと環境の対で表わす．この対を閉包とよぶ．

[例] $M \equiv \lambda x.\,add1\ x$とすると，$M[x:=1]$は環境を$env=\{(x\ 1)\}$として，閉包$\{(add1\ x), env\}$で表現する．

方法(2)による*natseq* 1のリダクション途中のグラフ表現は次のようになる(図2)．ここで，θを用いずに，ポインタのループによって*natseq*を表現していることに注意．この手法は関数型言語の処理系ではよく用いられるものである．これは，また，数学上で扱いにくい概念(少なくとも，初心者の読者には3.2のθの導入はわかりにくかったと思う)が計算機科学では容易にかつ直感的にとらえられる一例であると思われる．

方法(3)については疋田の解説[16]が詳しい．

いずれの方法にせよ，リダクションの過程でグラフを作る必要がある．この手間が内側からリダクションを行なう手法(この場合にはスタックでよい)に比べて，余分になるため，インプリメンテーション改良の努力は，この手間を減らすことに注がれてきた．本稿ではこれ以上深く議論しないが，Hughesのスーパーコンビネータ[18]，Johnssonのlambda lifting[23]のような最適化手法も，基本的発想はグラフのリダクションを単純化しようとすることである．また，正規形を求める上で，必ずリダクションされねばならない項は，最左でなくても，直ちにリダクションを行ない，中間のグラフの生成を抑えることも考えられる．答をえるのに必ずリダクションしなくてはならないリデックスをあらかじめ見出しておく技法はstrictness解析(関数の本体部の評価に全部の引数を必要とする関数をstrictであるという)と知られ，abstract interpretationとよぶ一般的な理論も作られている[34]．また，strictness解析の理論的考察を実際の関数型言語処理系へと応用する試みも行なわれている[8][43]．

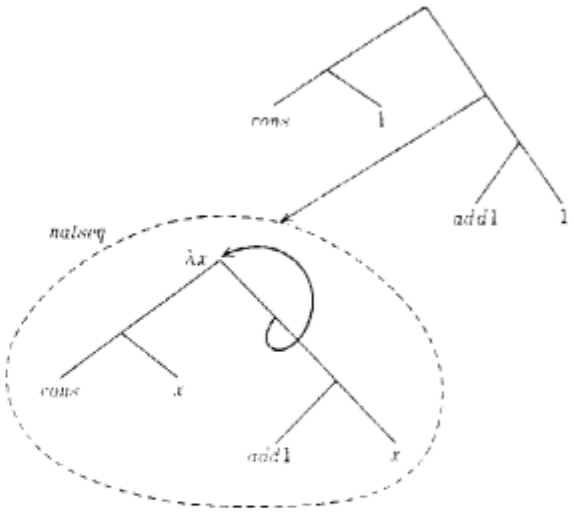

図2 ***cons*** **1 (*natseq* (*add1* 1))** のグラフによる項の表現(*natseq* 1のリダクションの過程で生成されるグラフ)

4 コンビネータ論理

4.1 基本事項

コンビネータ論理は，以下のアルファベットA_cおよび構文規則S_cより生成される項に関する論理体系である．

アルファベットA_c

定数 K, S(コンビネータという)

変数[18] $v_0, v_1, \cdots, v_n, \cdots$

括弧 (,)

構文規則S_c

$\langle$項$\rangle$ :: = $v_i \mid K \mid S \mid (\langle$項$\rangle\ \langle$項$\rangle)$

括弧および項の結合に関する略記規則はラムダ計算系と同じである．

[例] SKK, Kxyは項である．

ここでは，3節と同様に，項のリダクション関係によ

18 以下の議論では，自由変数を含まないλ-項を変換してえられるコンビネータ項を扱うので，コンビネータ論理における変数は現われない．

って生成される計算系に興味がある．本節の主題は，自由変数を含まない項(つまりコンビネータ)を対象としたラムダ計算系のサブセットをコンビネータ論理に翻訳することである．関数型プログラムからラムダ計算系への対応はすでに3節でみているので，ラムダ計算系を介して，関数型プログラムとコンビネータ論理の対応付けができることになる．

まず，項について次の対応付けを行なう．

対応規則 φ_{LC}

ラムダ計算系	コンビネータ論理
変数 v_i	⟶ 変数 v_i
λ-項 $\lambda xy.x$	⟶ 定数 K
λ-項 $\lambda xyz.xz(yz)$	⟶ 定数 S
λ-項どうしの結合	
$(M\ N)$	⟶ $(M'\ N')$

ただし，

$M'=\varphi_{LC}(M), N'=\varphi_{LC}(N)$

である．

3節では自由変数を含まないλ-項をコンビネータとよんだが，上の対応付けでわかるように，S, Kはλ-項のコンビネータに対応したものになっている．この用語を流用して，S, Kもコンビネータとよぶ．

自由変数を含まないすべてのλ-項はφ_{LC}により，コンビネータS, Kのみからなるコンビネータ論理の項に変換されることが知られている．

[例]　$\lambda x.x$はSKKに変換される．

[練習問題]　上の事実を証明せよ．(ヒント：$S\equiv\lambda xyz.xz(yz)$, $K\equiv\lambda xy.x$ とし，$\lambda x.x=SKK$, $\lambda x.MN=S(\lambda x.M)(\lambda x.N)$ であることに着目し，λ-項の項の構造に関する帰納法を用いる．)

コンビネータ論理では，項の代入の概念がないので，β-リダクションはコンビネータ論理には移らない．その代りに，コンビネータ論理に次のリダクションを導入する．

$$KMN \rightarrow M$$

$$SLMN \rightarrow LN(MN)$$

このリダクションをC[†9]とし，コンビネータ論理の項の集合をCとすると$\langle C, \mathrm{C}\rangle$は，コンビネータ論理における計算系となる．

ラムダ計算系ではβ-リダクションが項をリダクションする上で基本となったが，コンビネータ論理では，この操作に直接対応するリダクションはない．その代りにSとKが基本的リダクションとして与えられる．β-リダクションは変数の場所に実引数値をいれることに相当すると直感的に理解することができた．これと同様の操作的解釈をSとKに加えると，$(SLM)N$においてSはNをLとMに分配すること，Kは$(KM)N$においてNを消去することと考えることができる．(括弧は上の解釈を行ないやすくするために付けた．もちろん，書かなくてもよい．)

さらに，ラムダ計算系の場合と同様に，δ-定数およびδ-規則を導入する．以降δ-定数，δ-規則をも含めたコンビネータ論理における計算系を改めて，$\langle C, \mathrm{C}\rangle$と書くこととする．また$\varphi_{LC}$についてもラムダ計算系の$\delta$-定数をそのままコンビネータ論理の$\delta$-定数に変換する規則をつけ加える．

[例]　$(\lambda x.add1\ x)\ 1$をφ_{LC}で変換した$S(K\ add1)(SKK)1$は，次のようにCでリダクション[†10]される．

$$S(K\ add1)(SKK)1 \rightarrow (K\ add1\ 1)(SKK\ 1)$$

$$\rightarrow add1\ (K\ 1\ (K\ 1))$$

$$\rightarrow add1\ 2$$

リデックス，正規形，戦略の概念はRに代えたCについて，ラムダ計算系と同様の定義ができる．

なお，関係Cが合流性をもつことも証明される[3, p. 155]．

4.2 コンビネータの種類

上の例にみるようにSとKのコンビネータのみでは，φ_{LC}で変換されるコンビネータ項が一般には大きくなってしまい，またリダクションの回数も増える．このために，実用的計算モデルとして用いるにはコンビネータの種類を増やす必要がある．

基本的と思われるものに，次のλ-項と対応づけられるコンビネータ，I, B, B', C, C', S'がある．

$\lambda x.x$	⟶ I $(\equiv SKK)$
$\lambda xyz.x(yz)$	⟶ B
$\lambda kxyz.kx(yz)$	⟶ B'

†9　関係Cと後で導入されるコンビネータCとを混同しないよう注意．

†10　括弧を何度か省略していることに注意．たとえば，$(K\ add1)\ 1\equiv K\ add1\ 1$.

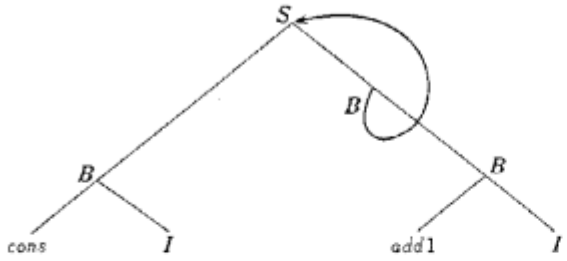

図 3 ラベル付グラフによるコンビネータ項の表現(*natseq* の定義)．ここで，ラベル付グラフではもとの λ-項の骨格が保存されていることに注意．*S* はもとの λ-項のグラフでみて，左右のサブグラフに変数があったときラベルとして置かれる．*B* は右のサブグラフに，*C* は左のサブグラフに変数があったときにラベルとして置かれる．*I* は乗が変数のときに付けられる．

$$\lambda xyz.(xz)y \longrightarrow C$$
$$\lambda kxyz.k(xz)y \longrightarrow C'$$
$$\lambda kxyz.k(xz)(yz) \longrightarrow S' \qquad (4.1)$$

この対応づけを付け加えて，新たに φ_{LC} を定義し直した φ_{LC} で，λ-項をコンビネータ項に変換してみよう．この変換は，手計算では面倒であるが，ラベル付グラフによって直感的な把握ができる．ここでは，詳しいアルゴリズムを省略するが，図3の *natseq* を表現する λ-項のグラフは，図3のようにコンビネータ(の列)をラベルにもったコンビネータ項のグラフで表わすことができる．

式(4.1)において，「′」を付けたコンビネータは Turner が SASL のコンビネータによる処理系の効率の向上を計るために導入したものである．これらの新しいコンビネータは二変数以上の変数束縛をもつ λ-項の変換の時に効力を発揮する．

[例] $\lambda xy.mult(plus\ x\ y)(sub\ y\ x)$ $((x+y)\times(y-x)$ を計算する関数)を S', B', C' をも用いてコンビネータ項に変換する．

図4に変換されたコンビネータ項のラベル付グラフの表現を示す．これを，式に書き直すと，

$$S'S(B'B\ mult(C'B(B\ plus\ I)I))$$
$$(B'C(B\ sub\ I)I)$$

となる．

上の例をみる限りは，λ-項をコンビネータ項に変換することにより，より複雑な(記号の数の多い)項表現をえる．しかし，一般には η-規則の適用によって，項表現をより単純なものとすることができる．

以上に述べたコンビネータの他にも，計算系の効率の向上に寄与するコンビネータが提案されている[36]．

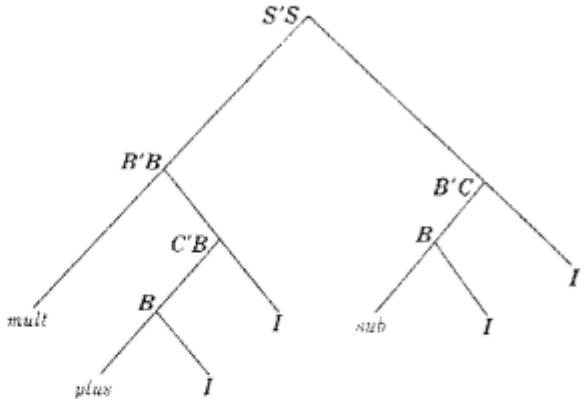

図 4 S', B', C' をも用いたコンビネータ項 $\lambda xy.mult(plus\ x\ y)(sub\ y\ x)$

[練習問題]

$$natseq \equiv S(B\ cons\ I)$$
$$(B\ natseq\ (B\ add1\ I))$$

とし，*hd* (*tl* (*natseq* 1))をリダクションせよ．(*natseq* は図3のようにポインタのループで表現し，再帰的になっていると考える．もちろん，理論的には θ をコンビネータであらわすことができる．)

[練習問題]

$$gcd \equiv S'S(B'C(B\ if\ (C(B\ eq\ I)0))I)$$
$$(B'S(B\ gcd\ I)(C'B(B\ mod\ I)I))$$

とし，*gcd* 12 8 を，本節で定義したリダクション規則および，*mod*, *eq*, *if* の δ-規則を用いてリダクションせよ．(ポインタによるループ表現を仮定する．)

4.3 コンビネータ論理による関数型言語処理系の実現

コンビネータ論理による関数型言語処理系の実現は，Turner の SASL が始めてである．その後この研究に触発されて，コンビネータを用いた関数型言語の処理系が作られている[17][33]．コンビネータのリダクションに基づく計算機も試作されている[9]．

言語処理系をインプリメントするための計算系としてみたときのコンビネータ論理のよい点は，リダクション時に変数を取り扱う必要がないという点である．

自由変数を含まない関数型プログラムは変換規則 φ_{LC} により変数を全く含まないコンビネータ項に変換することができる．さらに，コンビネータ論理におけるリダクションはラムダ計算系の β-リダクションに比べると単純明快である．3.6でもふれたように，ラムダ計算系の

処理の多くの手間が変数の取り扱いにさかれていたことを考え合わせると，コンビネータ論理の利点は明白であろう．

しかしながら，コンビネータ論理による方法がラムダ計算系によるインプリメント法に比べて，確実に有利であると断言することもできない．確かにラムダ計算系の単純なインプリメンテーションに比べると，コンビネータ論理を用いた手法はすぐれているが，手続き型やLispの処理系で伝統的に用いられているコンパイル手法をも活用したラムダ計算系のインプリメント法に比べて能率がよいとは言えないからである．両者の手法ともまだ改善の余地があり，今後の研究の発展が期待される．

5 項書換え計算系

項書換え計算系は，今まで述べてきたリダクションの概念をより一般化した抽象リダクションシステムである．4節で述べたコンビネータ論理も項書換え計算系の一例である．代数における語問題で取り扱う等式の集合や，いわゆるプロダクション・システムとよばれるものも，項書換え計算系の一つである．

5.1 基本事項

A を項の集合とする．A は変数，定数，および，適当な構文規則より生成される項より成る．ここでは，構文規則については議論しない．

R を書換え規則 $l_i \mapsto r_i$, $i=0,1,\cdots$ の集合とする．ここで，$l_i, r_i \in A$ でかつ，r_i の中に現われる変数はすべて l_i に現われるものとする．この時 (A, R) を項書換え計算系とよぶ．

［例］ 項書換え計算系における gcd のプログラムの定義

$$gcd\ m\ 0 \mapsto m \tag{5.1}$$

$$gcd\ m\ n' \mapsto gcd\ n'\ (mod\ m\ n') \tag{5.2}$$

ここで，gcd, mod は定数，m, n は変数とする．A を規定する時に，変数，定数は与えられているものとする．(5.1)，(5.2)の項の構文規則としては，筆者らの項書換え言語 T[20]の構文規則を用いた．式(5.2)は，

$$\begin{aligned} gcd\ m\ 1 &\mapsto gcd\ 1\ (mod\ m\ 1)\\ &\vdots\\ gcd\ m\ n &\mapsto gcd\ n\ (mod\ m\ n) \quad n \geqq 1\\ &\vdots \end{aligned} \tag{5.3}$$

の略記である．

なお，項の構文規則を決めれば部分項の概念も，ラムダ計算系と同様に定めることができる．

リダクション

項書換え計算系におけるリダクションの概念を説明する前に代入 θ の概念を導入する．代入 θ とは変数 ν_i と項 N_i との対応付けの集合 $\{\nu_0 := N_0, \nu_1 := N_1, \cdots, \nu_k := N_k\}$ であり，操作 $\theta(M)$ を次のように定める．

$\theta(M)$: M の中に出現するすべての $\nu_0, \nu_1, \cdots, \nu_k$ を各々 $N_0, N_1, \cdots, N_k$ で置き換える．

リダクション $\rightarrow_R$ を代入 θ と項の書換えで定まる関係として次のように定義する．

すなわち，$M \rightarrow_R N$ とは書換え規則 $l_i \mapsto r_i \in R$ が存在して，M の一つの部分項 $\theta(l_i)$ を $\theta(r_i)$ で置き換えたものが N である．

他の計算モデルと同様，リデックスの概念もこのリダクション $\rightarrow_R$ の関係に基づいて定められる．

［例］ $R=\{(5.1), (5.2)\}$ とすると，

$gcd\ 12\ 8 \rightarrow_R gcd\ 8\ (mod\ 12\ 8)$

ここで，書換え規則 $l_i \equiv gcd\ m\ n$, $r_i \equiv gcd\ n\ (mod\ m\ n)$, $n \neq 0$ とし，$\theta=\{m := 12, n := 8\}$ とした．

この場合は $M \equiv gcd\ 12\ 8$, $N \equiv gcd\ 8\ (mod\ 12\ 8)$ で，$\theta(l_i)$ は M そのもの，$\theta(r_i)$ も N そのものとなっている．

［例］ コンビネータ論理と項書換え計算系との対応．

$R=\{Sxyz \mapsto xz(yz), Kxy \mapsto x\}$, A をアルファベット A_c と構文規則 S_c で生成される項の集合とすると，(A, R) は項書換え計算系となる．つまり，コンビネータ論理の計算系は項書換え計算系の一つとなっていることがわかる．

［例］ ラムダ計算系と項書換え計算系との対応．

λ-項の集合 $\{P_1 \equiv \lambda\nu_1.Q_1, P_2 \equiv \lambda\nu_2.Q_2, \cdots, P_n \equiv \lambda\nu_n.Q_n\}$ に対して，$\nu_1, \cdots, \nu_n$ を変数，$P_1, \cdots, P_n$ を定数とし，上の集合を P のように変換する．

$$P \begin{cases} P_1\ \nu_1 \mapsto \bar{Q}_1,\\ P_2\ \nu_2 \mapsto \bar{Q}_2,\\ \quad\vdots\\ P_n\ \nu_n \mapsto \bar{Q}_n \end{cases} \tag{5.4}$$

$Q_1, \cdots, Q_n$ はそれ自身 λ-項であるかもしれないので，各々について，再帰的に(5.4)で示した λ-項から項書換え規則への変換を行なう．その結果を $\bar{Q}_1, \cdots, \bar{Q}_n$ とする．この時に生成された書換え規則の集合を Q，定数と変数の集合を Q とする．

$$A = \{\nu_1, \cdots, \nu_n, P_1, \cdots, P_n\} \cup Q$$

$$R = P \cup Q$$

として(A, R)は項書換え計算系となる.

5.2 項書換え計算系におけるプログラミング

ラムダ計算系, コンビネータ計算系を簡単なプログラミング言語と考えた時, これらの計算系ではプログラムとは項であった. 与えられた項があらかじめ定められたリダクション規則に従って, リダクションされ, (正規形が存在すれば)正規形に到達することが計算の意味であった. 正規形, 合流性の概念も, このあらかじめ組み込まれたリダクションの概念に基づいて定められた.

これに対し, 項書換え計算系においては, プログラムは項書換え規則の集りである. 正規形, 合流性の概念もこの項書換え規則の集りが定めるリダクションに基づいて定めなくてはならない. 当然のことながら, このような計算系では, 合流性(およびそれに基づく正規形の一意性)は一般には保証されない.

[例] 正規形が複数個存在する簡単な例.

$$f\ 0 \mapsto 1 \tag{5.5}$$

$$f\ x \mapsto 2 \tag{5.6}$$

$f\ 0$は, (5.5)の規則を適用すると1, (5.6)の規則を適用すると2になる.

しかし, 正規戦略が存在し, 正規形が一意に定まるような「良いふるまい」をする書換え規則を与えることも容易である.

正規形が一意に定まる一例として(5.1)(5.2)によって与えられる項書換え計算系で, $gcd\ 12\ 8$を求めてみよう. リダクションは次のように行なわれ, 唯一の正規形4が求まる. なお, ここでmodの書換え規則は既に与えられているものと仮定した.

$$\begin{aligned} gcd\ 12\ 8 &\xrightarrow[(5.2)]{} gcd\ 8\ (mod\ 12\ 8) \\ &\xrightarrow[mod]{} gcd\ 8\ 4 \\ &\xrightarrow[(5.2)]{} gcd\ 4\ (mod\ 8\ 4) \\ &\xrightarrow[mod]{} gcd\ 4\ 0 \\ &\xrightarrow[(5.1)]{} 4 \qquad (\text{正規形}) \end{aligned}$$

ここで$\rightarrow$は書換え規則によって左辺から右辺に書き換えられることを示し, $\rightarrow$の下の添字は適用される書換え規則を示す.

5.3 項書換え計算系の合流性

項書換え計算系の書換え規則の左辺の項の構成法は一般には何ら制限を受けない. したがって, Rの合流性は, 限られた場合にしか保証されない. 合流性を実現するため, 新たに書換え規則をつけ加えるアルゴリズムにKnuth-Bendixアルゴリズム[24]が知られているが, Knuth-Bendixアルゴリズムを用いても, 合流性が保証される場合は限られている. この分野に関しては, 現在活発な研究が行なわれており, 近い将来もう少し見通しのよい議論ができるようになろう.

5.4 項書換え計算系と関数型言語との結びつき

項書換え計算系は抽象的なリダクションシステムとして, 今までは主として理論的な研究が行なわれてきた. 計算機にインプリメントされた実験的システムは,

(1) 定理証明システム
(2) 一般的な等号論理におけるリダクションシステム
(3) 純粋に項書換え計算系の性質を研究するソフトウェア環境

などの目的をもったものが作成されている. システムとしてはReve[30]がよく知られている. (2)の目的をもったシステムとしては, 等号論理を基礎にした仕様記述言語の処理系(たとえばOBJ 2[12])がある. しかし, プログラマとのインタフェイスには, 項書換え計算系としての特徴は明確には表われてこない.

上のアプローチとは異なったものとして, 項書換え計算系をそのままプログラミング言語として考える見方がある.

5.1, 5.2の項書換え計算系の議論はこの見方を念頭においてなされたものである. これは項書換え規則の左辺に記述できる項を制限することで, 項書換えシステムの合流性や正規戦略の存在の保証を行なおうとする考えに基づく. 項の制限は, 構文上の制限の他に, 言語を型付きシステムとすることによっても行なうことができる.

書換え規則の集合をプログラムと考える関数型言語にはLazy ML[1], HOPE[6], KRC[46], Miranda[47], O'Donnelの等式言語[37]などが知られている†11.

†11 ここでは, 言語の設計者の意図にかかわらず, これらの言語で書かれたプログラムが書換え規則とみなすことができるという意味で, HOPE, KRC, Lazy MLを含めた. これらの言語のセマンティックスを細かく検討してみると, 書換え計算系のセマンティックスで, 単純明快に解釈できない点もあるが, 書換え規則であるとして, 新たな解釈を加えてみることも有益であろうと思われる.

わが国においても，長田は実験的等式システムをインプリメントし，プログラミング言語としての有効性の評価を行なっている[35]．稲垣らの代数的仕様記述によるプログラミング言語の処理系の自動生成システムも，項書換え計算系を内蔵したものとなっている[21]．また検証系を念頭においた阪大のシステム ASL[22]もある．筆者らの項書換え言語 T[20] については，既にふれた．

プログラミング言語としての項書換え計算系の有効性(プログラミング・スタイルとしての有効性および実行効率)の評価は今後の研究の成果を待ちたい．

6 カテゴリーに基づく計算系

カテゴリーは今まで議論してきた関数を射(ある対象から他の対象への矢印(arrow))としてとらえ，抽象的に関数の空間を扱うものである．林の論文[14]に指摘があるように，カテゴリー論は計算機科学においても，仕様記述に用いられる様々な論理を統一的に扱う上で有効であることが示された．

関数型言語では関数をプログラムの基本単位としていることから，カテゴリーと関数型言語との結びつきが考えられる．ここでは，結びつきの手がかりとなる研究のいくつかにふれよう．

(1) Backus は 1978 年チューリング賞受賞記念論文において，関数の組合せを主体とするプログラミング・スタイルの提唱を行なった．彼は FP とよぶ言語をその論文において提示し，FP のプログラム代数の可能性を示した．

[例] gcd のプログラムは，$-_1 \rightarrow -_2; -_3$ という関数を結合する操作(if $-_1$ then $-_2$ else $-_3$ に相当する)，関数の合成 $\circ$，関数の対を作る操作$[_, _]$を用いて，FP で次のように書ける．

$$gcd = eq0 \circ 2 \rightarrow 1; gcd \circ [2, mod]$$

ここで，$eq0$ は 0 を判断する述語関数，1, 2 は対の選択関数で各々 1 番目，2 番目を選択する．

FP は関数の合成を主として，プログラムを組み立てていく点，関数を射としても容易に解釈できる点でカテゴリーと類似している．

(2) 横内は FP をカリー化と作用演算子で拡張した体系が β, η-規則をもつ型付きラムダ計算系と同等の能力をもつことを示した．この体系は Cartesian Closed Category (CCC) とよばれる一つのカテゴリーになっている[48]．

(3) 理論面では，CCC が β, η-規則をもつラムダ計算系のモデルになっていることを，Lambeck[26] と Koymans[25]は示している．

(4) Cousineau らは関数型言語 ML[13]を CCC に翻訳し，翻訳した結果を CAM (Categorical Abstract Machine)によって実行する試みを発表している[10]．

3.6 において，ラムダ計算系のインプリメントについて述べたが，そこにおける課題の一つは，変数をいかに効率よく扱うかであった．カテゴリーにおいては，変数は捨象され，関数どうしの結合方法のみを問題にすればよいので，カテゴリーを用いた単純化された効率のよい計算モデルが考えうる．しかし，一引数関数の結合のみでは，計算の動的側面をとらえられないので，直積構造やラムダ計算系の λ-抽象化や作用(apply)に相当するプリミティブを導入することとなる．そのような性質をもったカテゴリーが CCC である．

カテゴリーを関数型言語の計算のモデルとして用い，具体的な処理系を作成したのは Cousineau らの仕事が始めてであると考えられるが，彼らの仕事は一つの新しい方向を示した点で興味深い．

7 まとめ

本稿ではリダクションの概念を中心に，関数型言語の計算のモデルについて解説した．リダクションの概念によらず，公理と推論規則でもって，計算系を考えていこうとするアプローチもあるが(その中で最も成功しているのは Prolog における一階述語論理であろう)，リダクションに基づく計算系は，その概念的単純性，実現の容易さの点から，計算の過程を抽象化し，さらに言語処理系をインプリメントする上で最もすぐれたものの一つである．

研究の歴史の古いラムダ計算系，コンビネータ論理と関数型言語との関連については，多くの知見もえられ，概念の整理もすすんでいる．したがって，本稿では，これら(特にラムダ計算系)と関数型言語との関係を論じるのに多くのページ数をさいた．項書換え計算系やカテゴリーと関数型言語との関連については，本稿では十分に論じられなかった．まだ概念が十分に展開されていないことや現在活発な研究が行なわれていることもあって，

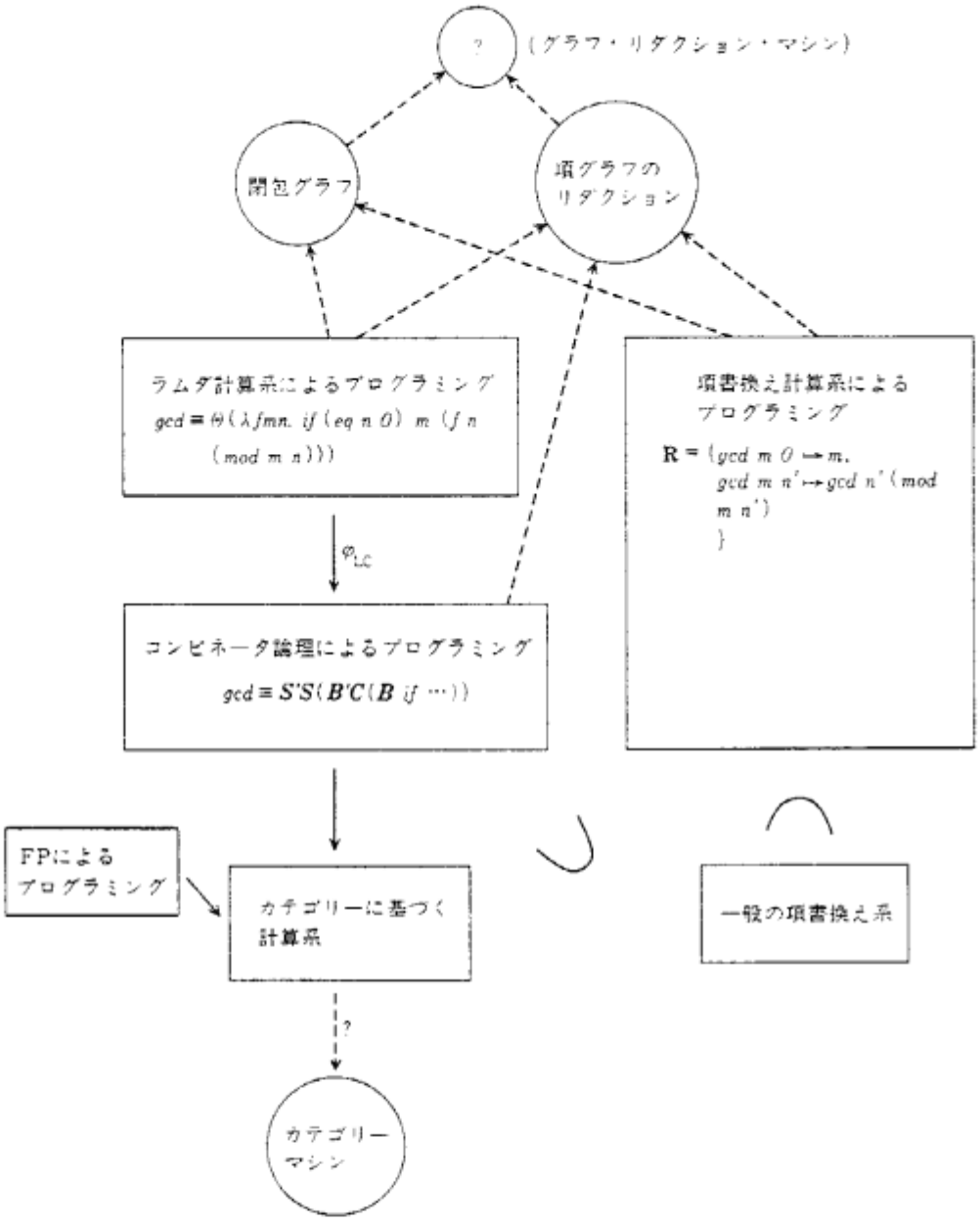

図 5 関数型言語の計算モデル——まとめ

記述がやや断片的であったと思われる．しかし，この魅力ある一分野により多くの研究者が目を向けられることを願って，あえて，本稿にこれらの分野の記述を含めた．

なお本稿では型付きシステムはとりあげなかった．ML のように多相型チェックがコンパイル時に行なえる言語では，型の議論は計算モデルにまでもちこまれない．しかし，Martin-Löf の型理論[31]のような形式的体系では，計算モデルに型の影響がはいるのかどうか今後の研究をまたなくてはならない．

本稿の草稿を読んで下さった佐々政孝氏は，図 5 に似た計算モデルの鳥瞰図ともいうべきものを示唆された．本稿のまとめとして有効であると思われるので，最後に付け加えたい．

謝 辞

本稿でふれた筆者らの研究に関して，筆者の一人井田は，日本学術振興会および文部省科学研究補助金制度(課題番号 59580027)の補助を受けた．また，草稿の段階から本稿に目を通して下さり，有意義なコメントを下さった次の諸氏に感謝する．佐々政孝氏(筑波大学)，疋田輝雄氏(都立大)，武市正人氏(電通大)および坂井公氏

(ICOT).

参考文献

[11], [19]は本文では引用されないが，本チュートリアルの背景となる問題を扱っているので参考文献として含めた．

[1] Augustsson, L.: A Compiler for Lazy ML, *Proc. 1984 ACM Sym. on Lisp and Functional Conference*, 1984, pp. 218-227.

[2] Backus, J.: Can Programming be Liberated from the von Neumann Style?, *Comm. ACM*, Vol. 21, No. 8 (1978), pp. 613-641.

[3] Barendregt, H. P.: *The Lambda Calculus: Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics, Vol. 103, North Holland, revised edition, 1984.

[4] Beeson, M.: *Foundation of Constructive Mathematics*, Mathematical Studies, Springer, 1985.

[5] Burge, W. H.: *Recursive Programming Techniques*, Addison-Wesley, 1975.

[6] Burstall, R. M. *et al.*: HOPE: An Experimental Applicative Language, *Record of 1980 LISP Conference*, 1980, pp. 136-143.

[7] Church, A.: *The Caliculi of Lambda-conversion*, Princeton University Press, 1941.

[8] Clack, C. *et al.*: Strictness Analysis——A Practical Approach, in *Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science 201, Springer, 1985, pp. 35-49.

[9] Clarke, T. J. W. *et al.*: SKIM——The S, K, I Reduction Machine, *Proc. 1980 ACM Lisp Conf.*, Aug. 1980, pp. 128-135.

[10] Cousineau, G. *et al.*: The Categorical Abstract Machine, in *Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science 201, Springer, 1985, pp. 50-64.

[11] Eisenbach, S. *et al.*: Declarative Languages: An Overview, *BYTE*, Vol. 10, No. 8 (Aug. 1985), pp. 181-197. 邦訳，宣言型言語とその概要，日経バイト，Jan. 1986, pp. 75-86.

[12] Futatsugi, K. *et al.*: Principles of OBJ 2, *Proc. 12th Annual Sym. on POPL*, New Orleans, 1985.

[13] Gordon, M. J. *et al.*: *Edinburgh LCF*, Lecture Notes in Computer Science 78, Springer, 1979.

[14] 林晋：カテゴリーと代数仕様の基礎，コンピュータソフトウェア，Vol. 2, No. 4 (1985)，日本ソフトウェア科学会，pp. 30-40.

[15] Henderson, P.: *Functional Programming Application and Implementation*, Prentice-Hall, 1980.

[16] 定田晃雄：コンビネータによる言語処理系，コンピュータソフトウェア，Vol. 2, No. 3 (1985)，日本ソフトウェア科学会，pp. 27-35.

[17] Hudak, P. *et al.*: A Combinator-based Compiler for a Functional Language, *Proc. 11th ACM Sym. on Principles of Programming Languages*, 1984, pp. 122-132.

[18] Hughes, R. J. M.: Super-combinators: New Implementation Method for Applicative Language, *Proc. 1982 ACM Symposium on LISP and Functional Programming*, ACM, Pittsburgh, Pennsylvania, 1982, pp. 1-10.

[19] 井田哲雄，関数的プログラミング，bit, Vol. 16, No. 2～No. 10 (1984)に連載．

[20] 井田哲雄，外山芳人，相場亮：項書換え言語 T，理研シンポジウム，関数プログラミング，理化学研究所，1986年1月，pp. 34-44.

[21] 稲垣康善他：代数的手法に基づくプログラム言語の仕様記述法と処理系の自動生成，文部省特定研究「多元知識情報」C班研究集会，1985.

[22] 井上克郎他：関数型言語 ASL/F とその最適化コンパイラ，電気通信学会誌，Vol. J 67-D, No. 4 (1984), pp. 458-465.

[23] Johnsson, T.: Lambda Lifting: Transforming Programs to Recursive Equations, in *Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science 201, Springer, 1985, pp. 190-203.

[24] Knuth, D. E. and Bendix, P. B.: *Simple Word Problems in Universal Algebras, Computational Problems in Abstract Algebra*, Pergamon Press, 1970.

[25] Koymans, K.: Models of Lambda Calculus, *Inf. Control*, Vol. 52 (1982), pp. 306-332.

[26] Lambeck, J. *et al.*: Cartesian Closed Categories and Lambda Calculus, preprint, Dept. Mathematics, McGill University, 1982.

[27] Landin, P.: The Mechanical Evaluation of Expressions, *Comput. J.*, Vol. 6, No. 4 (1964), pp. 308-320.

[28] Landin, P.: A Correspondence between Algol 60 and Church's Lambda Notation, *Comm. ACM*, Vol. 8, No. 2 (1965), pp. 89-101.

[29] Landin, P.: The Next 700 Programming Languages, *Comm. ACM*, Vol. 9, No. 3 (1966), pp. 157-166.

[30] Lescanne, P.: Computer Experiments with the REVE Term Rewriting System Generator, *10th Sym. on POPL*, Austin, 1983.

[31] Martin-Löf, P.: *Intuitional Type Theory*, Bibliopolis, 1984.

[32] McCarthy, J. *et al.*: *LISP 1.5 Programmer's Manual*, MIT Press, 1962.

[33] Muchnick, S. S. *et al.*: A Fixed-Program Machine for Combinator Expression Evaluation, *Proc. 1982 ACM Symposium on LISP and Functional Programming*, ACM, Pittsburgh, Pennsylvania, 1982, pp. 11-20.

[34] Mycroft, A.: Abstract Interpretation and Optimising Transformations for Applicative Programs, Dept. of Computer Science, Univ. of Edinburgh, Dec. 1983.

[35] 長田博泰：等式システムとそのインタプリタ，理研シンポジウム，関数プログラミング，理化学研究所，1986年1月，pp. 28-33.

[36] Noshita, K. and Hikita, T.: The BC-chain Method for Representing Combinators in Linear Space, *New Generation Computing*, Vol. 3 (1985), pp. 131-144.

[37] O'Donnel, M. J.: *Equational Logic as a Programming Language*, MIT Press, 1985.

[38] Sasaki, H. and Katayama, T.: Global Storage Allocation in Attribute Evaluation, *Proc. 13th Annual*

Sym. on POPL. St. Peters Beach, Florida, 1986.

[39] 佐々政孝：属性文法のチュートリアル，第16回ソフトウェア基礎論，第4回プログラミング言語合同研究会，情報処理学会，1986年2月．

[40] 篠田陽一，楠浩志，片山卓也：属性文法に基づいた関数型言語AGのプログラム作成・実行支援システム，第16回ソフトウェア基礎論，第4回プログラミング言語合同研究会，情報処理学会，1986年2月7日．

[41] Steele Jr., G. L.: *Common Lisp: The Language*, Digital Press, 1984.

[42] Stoy, J.: *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, 1977.

[43] Tanaka, J.: Optimized Concurrent Execution of an Applicative Language, Ph. D. Thesis, Dept. of Computer Science, Univ. of Utah, Mar. 1984.

[44] Turner, D. A.: SASL Language Manual, St. Andrews University Technical Report, Dec. 1976.

[45] Turner, D. A.: A New Implementation Technique for Applicative Languages, *Softw. Pract. Exper.*, Vol. 9(1979), pp. 31-49.

[46] Turner, D. A.: The Semantic Elegance of Applicative Languages, *Proc. 1981 Conf. on Functional Programming Languages and Computer Architecture*, Oct. 1981, Portsmouth, New Hampshire, pp. 85-92.

[47] Turner, D. A.: Miranda: A non Strict Functional Language with Polymorphic Types, in *Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science 201, Springer, 1985, pp. 1-16.

[48] Yokouchi, H.: Application and Composition in Functional Programming, to appear in *JIP* Vol. 8, No. 3.