数理解析研究所講究録
第480巻 1983年 33-52

構成的数学とプログラム合成

都立工科短大　林　晋
(Susumu Hayashi)

本論文では、構成的数学の計算機科学的側面についての概観と、著者が開発中のプログラム合成系(抽出系)の解説を行なう。著者は、数年前に日本数学会基礎論分科会の特別講演で、構成的数学について話す機会に恵まれたが、その時には、計算機科学について無知であった為、構成的数学と計算機科学の交流と言う興味深い分野について言及する事ができなかった。本論文をもって、その補足としたい。

§1　構成的数学と計算機科学

計算機科学における重要な問題に、プログラムの検証と合成がある。これ等の問題は多くの困難を含み、解決には程遠いと言うのが現状ではあるが、幾つかの注目すべき研究もある。([GMW 79] の文献表参照)特に、

proof-checkerに関して言えば、Landauの「解析学の基礎」を検証したAUTOMATH [BJ79]、タイプ付けされた関数型言語MLによって記述され、かつMLによって証明の戦略を容易に記述できるEdinburgh LCF [GMW79]、PL/I系のプログラム言語PL/CSの検証系で、構成的論理に基づくPL/CV [CJE82]が良く知られている。(他にWeyhrauchのFOLが特に有名であるが、本題と直接的関係は無い。)これ等の内、陽に構成的数学に基づいているものは、PL/CVのみであるが、他の二つも構成的数学と深い関連を持つ。AUTOMATHにおける排中律の使用は、古典論理に基づく定理を検証する時に追加される機能であり、本来のAUTOMATHは直観主義数学におけるconstructionの理論に密接に結びついており、非常に構成的な性格を持つ。(cf. [Sc70], [B80]) また、Edinburgh LCFはD. Scottの計算の理論と古典論理に基づく検証系であるが、tactic等を無視して純粋に数学的に見るとMLはMartin-Löf [ML79]のsystemを関数型言語と見なした時、そのsubsetと考える事ができる。Scott理論をMartin-Löfの理論に置き換える事により、ML及びそれ自身を記述可能にするLCTの構想もあるようだ。([C82]) LCTはtactic等の正当性も扱える検証系だが、単にEdinburgh LCFのScott理論(PPλ)を、

Martin-Löfの理論に置き換えたものは、SwedenのGöteberg大学で実現されているようである。(c.f. [P82] 以下、Göteberg type theoryと呼ぶ。) この様に、proof checkerに関する重要な研究には、構成的数学と密接な関係を持つものが多い。さらに、検証系の理論的基礎となる「計算の意味論」においても構成的数学との関連が見られる。例えば、構成的数学の一分野とも考えられるλ-計算の理論と計算の意味論との結び付きは言うまでもない。また、D. Scottの計算理論の形成に、彼の構成的数学の研究が影響を与えなかったとは考えられない。「計算可能な関数は連続である」と言うScott理論の基本的着想は、Kleene, Kreiselのcontinuous functionalや彼自身によるtopological modelの研究と無縁ではあるまい。

話題をプログラム合成に移そう。プログラム合成とは、プログラムの仕様書が与えられた時、その仕様どおりに働くプログラムを自動的に作製するプログラムを作製する問題である。ここではdeductive approachと呼ばれる一つの接近法を解説しよう。話を単純にして、プログラムの仕様書とは、入力と出力との間に成立すべき関係式の事とする。これを $\varphi(x,y)$ とする。また、入力は一定のデータタイプに属すると考えて良いだろ

う。これをDとする。仕様(specification)とは、対<D, φ>の事である。プログラムPが仕様を満たすとは、

$$\forall x \in D\ \exists y\ (Px \simeq y\ \&\ \varphi(x,y))$$

となる事を言う。この時、$\forall x \in D\ \exists y\ \varphi(x,y)$は正しいのだから、仕様が解を持つ時、適当な形式系でこの式が証明可能であるだろう。その形式系がChurchの法則をみたせば、recursiveに計算できる項$t(x)$があり$\forall x \in D\ \exists y\ (t(x) \simeq y\ \&\ \varphi(x,y))$がやはりその形式系で証明できる。したがって、原理的にはプログラム合成はChurchの法則をみたす形式系の自動証明の問題となる。以上がdeductive approachの考え方だが、最も難しい部分は自動証明を行なう部分である。また、自動証明を有効に行なうアルゴリズムが完成したとき、そのアルゴリズムを使って証明を経由せずにプログラムの合成を行なえるのではないかとも考えられる。しかし、証明からプログラムを抽出(extract)する、あるいは証明そのものを実行すると言う問題も興味深い問題を含んでいる。例えば、proof checkerを使って構成した証明からプログラムを抽出したとすると、そのプログラムは検証ずみと考える事ができる。つまり、仕様の構成的証明を作製すると言う事は、プログラムの作製とその検証を同時に行なっていると考えられる。

さらに、この考え方をすすめれば、構成的形式系自体を一つのプログラム言語とみなす事ができる。(これについては[ML79]に詳しい解説がある。) また、抽出されたプログラムのoptimizationや、プログラムの世界の概念を自然な形で、数学ー論理の世界の言葉で書けるかと言う問題も面白い問題である。

以下に、その様なプログラムの抽出あるいは証明そのものをプログラムとして実行するという試みを紹介しよう。(この二つを合わせて、単に「証明を実行する」と言う。) プログラム合成のdeductive approachはかなり古くからある考え方だが、最初に実現されたものは、おそらく後藤[Gt79]のものであろう。後藤はHAの検証系を作製し、Gödel functional interpretationによってprimitive recursive functionalを抽出し、これをLisp codeに変換する事によってプログラムを得た。後藤の処理系は、剰余定理から、商と余りを求めるプログラムを抽出している。しかし、佐藤は[Sa79]でrecursive functionalをLisp codeに変換する時、妥当性の問題が生じる事に注意している。その為、佐藤は、primitive recursive functionalの項を、reductionによって処理する方法を提言している。(また、数学的に見ても妥当性の問題が生じない様なLispや、

Prolog等も、佐藤,萩谷,桜井によって実現されている。cf.[SH81],[SS82]) 他方、Goad[Gd80]は、natural deductionの証明をnormalizationによって実行する際、証明図の持つ論理的情報を利用する事により、論理的情報を含まないプログラムでは不可能なoptimizationを行なえる事を示した。GoadはFOL風の検証系を実現し、bin-packing algorithmに対して実験を行ない、その方法の有効性を示している。また、Goadは証明図のcompactな表現法や、normalizationの効率化について興味深い結果を得ている。萩谷[Hg83]は、call by need reduction [L80]の方法を取り入れてnormalizationの一層の効率化を実現している。

また、Martin-Löf [ML79]は、彼のtype theoryが、一種のタイプ付き関数型言語と見なし得る事を主張しているが、先に述べたGöteberg type theoryで作製した証明(construction)を実行するinterpreterも報告されている([N81])。

§2. Pure Lispに基づく構成的形式系と検証系

前節で紹介した手法で合成されるプログラムは、関数型でrecursionを含むが、iterationを含まない。これは、通常の数学的形式系には、iteration, dynamicな変数等の

概念が無いからである。(Floyd-Hoare流の理論, e.g. PL/CV はこの限りでない。[CJE82]で引用されているBatesの処理系はproceduralなPL/CSのプログラムを抽出するのだろう。)例えば、ユークリッドのアルゴリズム等の本質的に手続き的なアルゴリズムを抽出しても、プログラムの実効性についての問題が生じる。ところが、証明論におけるアルゴリズムは本質的に帰納的(i.e. stackを必要とする)である事が多い。また、それ等のアルゴリズムは証明の形で記述され具体的なアルゴリズムとしては記述されない方が多い。さらに、これ等のアルゴリズムが扱うデータは本質的にはリストである。以上の様な事実も考慮に入れ、著者はリストを扱うアルゴリズムの記述言語としてPure Lispを採用する構成的形式系を作製し、証明論の定理の証明から、そのアルゴリズムを実行するLisp Programを抽出する実験を計画した。この様な実験を行なう際、最も重要な点は形式系の、designをどのようなものにするかと言う事にある。

ところで、前節で述べたMartin-Löfのtype theoryは本来Bishopの構成的数学を形式化する為に考えられたものであるが、同じ目的の為に考えられた形式系として、Feferman [Ff78], Myhill 及び Friedman の形式系があ

る。この内、FefermanのT_0はアルゴリズムとその性質の記述の直截さにおいては、最も優れている。また、T_0はMartin-Löfの理論と比較する時、仕様記述の柔軟性と拡張性において優れている。（これはT_0がtype freeである事に由来するが、この事を欠点とみなす考えもあるだろう。）T_0とその変種に関しては種々のrealizability interpretationが知られており、そのrealizerはT_0におけるalgorithm記述子として用いられる条件文付きのcombinatorである。そこで、T_0のdomainをLispのS-式で解釈し、combinatorをLisp codeで置き換えた形式系が考えられる。著者は[Hy83]において、その様な形式系LMとLisp codeによるrealizability interpretationを与えた。S-式に関する構造帰納法と有限的inductive definitionを持つ実験的な処理系は完成しており、これによって、数のリストから最大数を求めるプログラムを抽出する実験を行なった。[Hy83]では、古典命題論理の決定問題のアルゴリズム（Wang algorithm）を手計算で抽出したが、現在これを実際に計算機で実現する為に、新しい処理系（PX program extracterと呼ぶ）を作製中である。以下、このsystemの概要を述べる。

1. term (項)

termは基本的にはPure Lispのformである。但し、関数と式は分離する。自己引用はapplyを通して行なう。applyは基本関数(predefined function)である。変数は個体変数、類変数、項変数にわかれる。束縛変数と自由変数は区別しないが項変数は自由変数としてしか使えない。また、宣言により、atomに関数の定義をbindできる。これをdefined functionと言う。

2、式

原子式は$E\alpha, \alpha:\beta$, $\alpha=\beta$, $CL\,\alpha$, $\alpha<\beta$ である。α,βは項で、$\alpha<\beta$は「αとβの値が存在し、αの値がβの値の部分式(S-式として)である。」と言う事を表わす。その他は順に[Hy83]の$\alpha\downarrow$, $\alpha\in\beta$, $\alpha\simeq\beta$, $Cl\,\alpha$に対応する。命題論理記号は、$\wedge$, $\vee$, $\rightarrow$, $\neg$, $\perp$, $\top$ だが、実際の処理系では &, +, ->, -, FALSE, TRUE を使用している。限量子は

$$(\begin{smallmatrix}\forall\\ \exists\end{smallmatrix}\ x_1 \cdots x_m:\alpha \quad y_1 \cdots y_n:\beta \cdots)$$

の形のものをゆるす。これは、$\begin{smallmatrix}\forall\\ \exists\end{smallmatrix}\, x_1\in\alpha \cdots x_m\in\alpha\, y_1\in\beta \cdots y_n\in\beta \cdots$の事である。実際の処理系では、$\forall,\exists$はUN EXである。また、$\alpha,\beta\cdots$中に、この限量子で束縛される変

数は無いものとする。

3、論理

基本的には、Scott[Sc79]のものと同じである。項変数(%で初まる変数)は、一般のtermを表わす。個体変数はS-式を表わし、常に値を持つとする。したがって

$$\frac{A(\%1)\quad E\ \%1}{\exists x Ax}\ ,\qquad \frac{A(\%1)}{A(\alpha)}$$

は正しいが、

$$\frac{A(x)}{A(\alpha)}\ ,\qquad \frac{A(\%1)}{\exists x Ax}$$

は正しくない。CLαはαが値vを持ち、vが類(class)のcodeである事を主張する。類変数は、単に個体変数をCLで相対化したものである。Eαはαが値を持つ事、$\alpha:\beta$はαとβが値を持ち、それをu,vとすると、uがcode vの類の元である事を示す。他方、$\alpha=\beta$は、「αが値を持てば、βも同じ値を持ち、βが値を持てば、αも同じ値を持つ。」と言う事を示す。処理系はLispで書かれているので、α,βが値を持つ時は$\alpha=\beta$か否かは、Lispの処理系で判断できる。

4. アルゴリズムに関する公理

[Hy 83] では deep-binding の interpreter を記述したが、PX では、a-list なしの apply を記述する。この時、environment は陽に現われない。また、shallow-binding の処理系とも整合的となる。我々の、apply の記述は、本質的に不完全なものになる。environment を使って計算を行なう処理系を、environment 無しで記述しようとする事も、その理由の一つではあるが、これはあまり本質的な事ではない。apply の記述が不完全になる最大の理由は、PX では適当な atom に関数の定義を bind する事が許されているからである。関数の定義の前後で apply の定義域は異なる。つまり、apply は open-ended でなくてはならない。関数の定義に関する environment を導入する方法もあるが、Lisp は本来 open-ended であるべきだから apply 等の完全な記述には、あまり、こだわらない事にする。

car, cdr 等の関数の coding は 'car, 'cdr 等とする。したがって

(apply 'car (list x)) = (car x)

等が公理となる。また、λ-式 α で表現される関数も、'α で code する。この時、次の様な funarg 問題が生じる。

(apply '(lambda (x) (cons x y)) (list z)) = ((lambda (x) (cons x y)) z)

は公理となるべきだが、左辺の最初の引数'(lambda...)は定数である。よって、この式にyに他の変数y'を代入すると、左辺は変わらず矛盾が生じる。この為に、*functionと言うexper関数を導して、

```
(apply (*function '(lambda(x) (cons x y)) (list 'y y)) (list z))
            = ((lambda (x) (cons x y)) z)
```

を公理とする。*functionはprog形式を使えば容易に実現できる。また, back quote macroを使えば、([FS82])

```
(*function '(lambda (x) α) (list 'x1 y1))
```

は

```
'(lambda (x) α[',y1/x1])
```

であると考えて良い。

5 証明と推論

PXは基本的にはLispである。証明や式は特殊なリストとして表現される。推論はLispの関数となる。例えば、α, α&α→βが式の時、

```
(prog (p1) (setq p1 (assume "α"))
   (return (impe (conji p1 p1) (assume "α&α→β"))))
```

を実行すると、α, α&α→βを仮定とするβの証明図が値となる。実際の処理系では、リストとしての証明

は display 上には表示されず、[GMW79] のように

$$-: \;..\;]\!-\beta$$

が表示される。]- は ⊢ 、二つのピリオドは、仮定が二つである事、-: はタイプ β の証明である事を表示する。また、この例で、assume の引数中にある double quote「"」は、二つの「"」間の S-exps を式と見て内部表現に変換せよと言うマクロである。PX の top-level は、Franz-Lisp の cmu-top-level と同じであるが、ML のように史を持ち、各 event は ex や vi でも編集でき、p? f? コマンドで、event 中で証明、式を値とするものを
する事ができる。

6. クラス

式 φ の自由変数が x のみで、φ が elementary な(-)型の式 (cf [Hy 83]) の時、

(ECA-def (x: name) φ)

により、name にクラスの定義が bind される。つまり、CL name, x: name $\Leftrightarrow \varphi$ が公理となる。φ が他に、変数達 $\bar{y}, \bar{z}$ を含み、$\bar{z}$ に関して elementary である時には、

(ECA-def (x: (name $\bar{y}$ $\bar{z}$)) ($\bar{z}$) φ)

とすると、name には、total function の定義が bind される。

$\bar{z}$ が class 達である時、(CL (name $\bar{y}$ $\bar{z}$)), x :(name $\bar{y}$ $\bar{z}$)⇔ φ が公理となる。例えば、$B^A \equiv$ (exp A B) は、$\bar{y} \equiv$ ( ), $\bar{z} \equiv A, B$, $\varphi(x, A, B) \equiv (\forall y : A)$((apply x (list y)) : B) により定義される。

[Hy83]では、$ECA^{(-)}$ のみが、class を生成する原理であったが、PXでは T_0 の $IG^{(-)}$ (inductive generation) も使用できる。$IG^{(-)}$ の特殊な形式として CIG (conditional inductive generation) も使用できる。"*definiendum*" を name あるいは、(name 変数列) とする時、

(CID-def (x : definiendum) [in t1] clauses の列)

によって、ECA-def の場合と同様に、definiendum に、class あるいは、class を値とする関数が bind される。但し、clause とは

$$(ci) \cdots\cdots \quad pi \rightarrow Fi1 \ [\& \ Fi2 \ \& \ \cdots \ \& \ Fin_i]$$

Piは term, F ij は (term : definiendum) か (-)型の式となるものである。但し、t1 と clauses 中の自由変数はすべて、definiendum に現われるとする。CIG は、$IG^{(-)}$ で書くと、clause の列を C1, ---, Cm とするとき、i(t1, R) である。但し、R(y, x) は

$$p1 = t \ \& \left\{ \bigwedge_{F \in M1} F \ \wedge \ \bigwedge_{t \in Pr1} Et \ \wedge \left(\bigvee_{t \in Pr1} y = t \right) \right\}$$

$$\vdots$$

$$p1 = \cdots = pn\text{-}1 = nil \ \& \ pn = t \ \& \left\{ \bigwedge_{F \in Mn} F \ \wedge \ \bigwedge_{t \in Prn} Et \ \wedge \left(\bigvee_{t \in Prn} y = t \right) \right\}$$

但し、M_iは$F_{i1}, \cdots, F_{in_i}$の内、(−)型の式の集合、Pr_iは同じく(term:definiendum)の形のtermの集合である。例えば、

(CID-def (x:(List X))

(atom x) → x = nil

t → (car x):X & (cdr x):(List X))

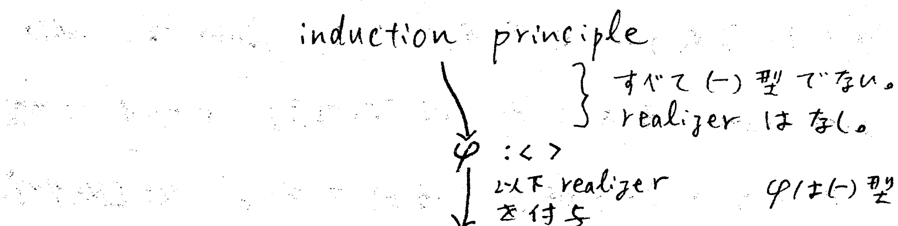
によって、Xの要素のリストのクラス(List X)が定義される。以上の$IG^{(-)}$, CIGには、それぞれinduction principleが付属する。

また、PXではT_0のjoinも使用できる。Joinは、index付きのdisjoint sumと言えるが(cf.[Ff78])、recordの様なデータ・タイプを作る時に、便利である。(cf.[ML79])

[Hy83]では、IG, Joinに関するrealizationの証明を与えていないので、これについて少し述べよう。まず、$IG^{(-)}$, Joinに関しては、Fefermanによる証明どおりで問題は無い。また、CIGでは、$\bigvee_{t \in Pr_i}$が使用されてはいるが、これは、y=tと言う形の式に関してであり、Et ($t \in Pr_i$)を仮定しているから、決定できる。また、(p1=t&…)∨(p1=nil & p2=t & …)∨ …… の∨も、p1, …, pnを順に評価してゆけば決定できる。一般のinductive definitionは、Prologの様なback trackを行なう処理系でないと、

realizerを構成できない。(もちろん、Lispでback trackを行なうprogramを書いても良いが、あまりに間接的になる) CIGを導入したのはこの為である。

この問題を解決する他の方法として、「inductive definitionに対応するinduction principleにはrealizerを付与しない」と言う方法もある。つまり、induction principleの帰結の内、induction principleとその間の帰結がすべて(-)型でない時はrealizearを付けず、(-)型となった後はrealizerを付与するわけである。



実際に、この方法で自然なprogramをextractする事は可能である。以前述べた、リスト中の最大数を見つけるprogramは、この方法でextractされている。この方法を取る時は、任意の式に対し、それを(-)式に変換するoperatorがあると便利である。例えば、¬¬を◇として、◇をmodal operatorとする論理体系が一つの候補である。

7. Induction Principles (帰納法)

帰納法は、$IG^{(-)}$, CIGに付属するものの他、数学的帰納

法と、S-式に関する帰納法、及び構造帰納法の推論法則が使える。(cf. [Hy83]) また、S-式に関する帰納法としては、[Hy83]のものだけでなく、$x < y$に関するcourse of values induction も公準とする。

最後に、PXと前節で述べた他の処理系を比較しよう。PXを[Gt],[Sa79],[Gd80],[Hg83],[N81]と比較すると、[Gt79]が最も近い。PXや[Gt79]の方法の長所は、object program (抽出されたprogram)がLisp programである為、Lispの処理系さえあれば動く事と、compileも可能である事にある。(Lisp codeを基本に考えれば、PXや[Gt79]のextracter 一種のcompilerと考えられる。)また、公準を追加した時にも、処理系も変更する事が容易である。例えば、新しい公理とそのrealizerの対を[Hy83]のようにR-axiomとして追加する事により処理系を拡張できるようにする事はたやすい。また、この際、realizerの正当性(拡張された処理系の正当性)を形式化し、検証する事も、他の方法に比較する時、はるかに容易であろう。

他方、[Gd],[Hg83],[N81]の方法では、normalizer (reducer)を必要とする。(これ等では、証明をアルゴリズムの表

記とみなし、それを直接実行するので一種のinterpreterと考えられる。)この方式の長所は、実行の際に、証明図としての情報を利用できる点にある。また、処理系の正当性は、それが実現された時には、比較的明白であると言える。(原理的には、計算が終了した際、その正当性の証明も計算機中に存在する。)反面、処理系の拡張を処理系自身の言語で記述検証する事はかなり困難を伴う。また、normalization theoremが成立する形式系は、必ずしも柔軟でない事も問題であろう。(直観主義的ZFのrealizability interpretationは存在するが、Gentzen typeの自然なZFの形式化でnormalizationの成立するものは存在しない。但し、inductive definitionの理論の場合のように逆の事例もある。)

Lisp interpreter自身ある種のreducerと考える事もできる事を考慮すれば、reducerをinterpreterとして持ち、extracter(-compiler)をcompilerとして持つ処理系も考えられよう。

文献

[GMW79] Gordon, Milner & Wadsworth, Edinburgh LCF, Springer Lecture Notes in Computer Science 78, 1979

[BJ79] van Benthem Jutting, Checking Landau's "Grundlagen" in the AUTOMATH system, Mathematical Center Tracts 83, Amsterdam, 1979.

[CJE82] Constable, Johnson & Eichenlaub, An Introduction to the PL/CV2 Programming Logic, Springer Lecture Notes in Comp. Sci. 135, 1982.

[Sc70] D. Scott, Constructive Validity, in Springer Lecture Notes in Math. 125, 1970.

[B80] de Bruijn, A survey of the project AUTOMATH, in To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, Academic Press 1980.

[ML79] Martin-Löf, Constructive Mathematics and Computer Programming, to appear in Logic, Methodology and Philosophy of Science VI, 1979.

[C82] Constable, Formalizing Metamathematics in Type Theory, preprint, 1982.

[P82] Petersson, A Programming System for Type Theory, University of Göteborg - Chalmers University of Technology, 1982.

[Gt79] Goto, Program Synthesis from Natural Deduction Proofs, IJCAI, 1979.

[Sa79] Sato, Towards a mathematical theory of Program Synthesis, IJCAI, 1979.

[SH81] Sato & Hagiya, HyperLisp, in Algorithmic Languages, North-Holland, 1981.

[SS82] Sato-Sakurai, Qute: A Prolog/Lisp type Languages for Logic Programming, preprint, 1982.

[Gd80] Goad, Computational uses of the manipulation of formal proofs, Ph.D. Thesis, Stanford University, 1980.

[Hg83] Hagiya, A Proof Description Language and its reduction system, to appear in Publications of R.I.M.S., 1983.

[L80] Lévy, Optimal Reductions in the Lambda-Calculus, in To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, 1980.

[N81] Nordström, Programming in Constructive Set Theory: Some Examples, ACM Conf. on Functional Programming Lang. and Comp. Arch., 1981.

[Ff78] Feferman, Constructive theories of function and classes, in Logic Colloq. 78, North-Holland, 1978.

[Hy83] Hayashi, Extracting Lisp Programs from Constructive Proofs, to appear in Publications of RIMS, 1983.

[Sc79] Scote, Identity and existence in intuitionistic logic, in Springer Lecture Notes in Math. 753, 1979.