「いま欲しいブレークスルー」
関数プログラミング

武市正人

コンピュータサイエンス誌 bit の 30 歳を記念して、「なぜか伸び悩んでいる」関数プログラミングに「なにが欲しいか」ということを書くことになったとき、ふと、考えてしまった。この特集のほかの分野は、どうみても、より広い範囲を対象としているのに、なぜ、「関数プログラミング」がとりあげられたのであろうか。たしかに、歴史が長いわりに、ひろく認知されるに至っていないところが、ブレークスルーを必要としていると言えなくもない。bit の読者にもなじみのない分野かも知れない。本稿では、関数プログラミングの概要を述べる余裕はないので、ご興味のあるむきはあとに出てくるテキストを参考にしていただきたい。関数プログラミングをひとことでいえば、関数の定義と関数の呼出しだけでプログラムを書く、というものである。ここでいう関数は C などのプログラム言語でよく使われている「関数」ではなく、数学の関数である。そこには、手続き型プログラミングに代表的な代入操作や、いくつかの文の逐次実行という概念はない。むしろ、そのような言語のなかの式だけでプログラムを書くというようなものである。そんなことができるのか、また、なぜ、それほどまでに無理をするのか、関数プログラミング偏執者？から多くのことを聞くことができようが、ここでは触れまい。最後のほうで、それとなく擁護者としての考えをお伝えすることにして…。

関数プログラミングの歴史

こうした関数プログラミングに、いま、求められるブレークスルーといっても、いきなり出てくるわけでもない。まず、関数プログラミングの分野の歴史を大雑把にたどってみよう。なかには、その時期のブレークスルーとなったものがある。いま求められるブレークスルーを考える上で、先人たちが解決してきたものを見ておくことは重要であろう。ときどきのブレークスルーなくしては、いまがなかったのだから…。

ラムダ計算法：

関数プログラミングの考え方は、1930 年頃の Church のラムダ計算法(lambda calculus)[1] に始まるといってよかろう。もちろん、その頃には「プログラミング」とはいわれなかったし、あとになって関連があると分かったことである。当時は計算機(コンピュータ)自体も存在しなかったが、「計算を捨てた数学」から、「計算を科学する」ことを復活させた功績も大きい。集合の元の対応関係を表現する関数と、その対応を具体的に計算する機

構を与えて、計算に関わる理論を構築したからである。ラムダ計算法は、プログラミングではなく、計算そのものを形式的な対象として捉えて計算可能性などを論じるひとつの枠組みとして考案されたものではあるが、最近の関数プログラミングにおいても、ラムダ計算法を基礎に置いているということに異論をはさむことはなかろう。

LISP, ISWIM, …:
しかし、関数プログラミングの悲哀もまた、ここにあるのかも知れない。関数プログラミングといえばラムダ計算法が顔を出すので、それとはなしに、プログラミングに興味のあるプログラマを遠ざけてしまったのではなかろうか。1960 年頃の McCarthy による LISP [2] がラムダ計算法に基礎を置くというふうに言われることもあるが、どうやらそうでもないらしい。細かいことはともかくとして、LISP の表記法には、たしかにラムダ計算法のものを借用したところがあるが、「関数に市民権を与える」といった考え方は初期の LISP には存在しなかったようである。名前の示すように、人工知能の研究のためにリスト処理を主眼として、むしろ、実用的な機能としての代入文や逐次実行制御など、ときの FORTRAN の伝統との妥協があった。Lisp より少し後になって、Landin が ISWIM(If You See What I Mean) [3] によって、「伝統との妥協から LISP を解放」したあたりが、純粋な関数プログラミングの始まりといえるであろう。しかし、LISP の処理系はすでに実用的になっていたが、ISWIM 風の言語のまともな処理系は存在しなかった。

John Backus:
手続き的な側面をもつ LISP プログラミングは研究者によって盛んに行なわれたが、純粋な関数プログラミングへの関心が引き起こされるまで 10 年余を要した。1978 年のチューリング賞の受賞講演 [4] で、Backus は Can Programming be Liberated from the von Neumann Style と題して、「関数プログラミングがなぜ良いか、また、手続き型(命令型)プログラミングがなぜ悪いのか」と語ったことが、関数プログラミングの提唱として注目を集めた。手続き型言語の FORTRAN の設計に功績があり、ALGOL60 の開発に影響を与え、それによって受賞した際の講演であった。歴史には、こうした皮肉もあってよかろう。

ML, Miranda, …:
Edinburgh 大学で行なわれていた定理証明生成系の研究の中から産まれた ML(Meta-Language)は証明系に対するメタ言語であったが、これ自体が関数型言語 [5] として関数プログラミングの世界に出されてきたのもこの頃である。これもまた、少々、純粋性に問題があったが、実用的な処理系が用意されたし、なによりも、その後に大きな影響を与えた型体系の重要性を知らしめた功績はきわめて大きい。イギリスでは、この頃にはいくつかの言語と処理系、それに、関数プログラミングの教科書も現れている。今から 20 年近

く前の話である。なかには、「初めての商品としての」関数プログラミング言語(の処理系)Miranda［6］というものもあった。Turner によるもので、きわめて斬新なアイデアによる目的コードを生成(して、それを解釈実行)するというものであった。そのアイデアはラムダ計算法とも関連するコンビネータ論理の成果に基づくもので、それが応用されたことに驚きがあった。

型：
上にあげた ML の型体系は、関数プログラミングにとっての大きなブレークスルーであった。否、むしろ、プログラミングの世界における一大ブレークスルーといってよかろう。単純に「型がきちんと整合しているプログラムは正しい」とは言いきれないが、型の概念はプログラミングの種々の側面にあらたな視点を与えることとなり、かなりの程度に型の役割と有効性が評価されるようになってきたことも事実である。

プログラミングの教科書：
1988 年に出版された Bird&Wadler の教科書［7］は、関数プログラミングをプログラミングの立場で明確に位置付けたという意味で、ひとつのブレークスルーを与えたといえよう。それまでの多くの書物は、ラムダ計算法や計算機構を中心に据えて書かれたものか、あるいは、それらだけを述べたものであった。10 年前に出版されたこの教科書には、プロ
グラミング方法論としての関数プログラミングが展開されていて、アルゴリズムの記述に重点が置かれている。

プログラミング言語と実用的な言語処理系：
前出の ML は 1980 年の半ばに SML(Standard ML)［8］として定められ、効率のよい処理系も作られた。しかし、その一方で、どの分野の言語についても見られるように、1970 年代から 1980 年代にかけて、関数型言語の小さい世界でもバベルの塔の状況を呈していた。1988 年には、共通の言語として Haskell［9］が作られ、その処理系も公開されるよう
になった。これはひとつのブレークスルーであるというのは、近代的な関数型言語に特徴的な機能がよく整理された形で入ったということにある。たとえば、それまでの言語や処理系では、入出力を操作として捉えていたので、関数プログラムとはいえ、「こっそりと」命令型の考え方を使ったり、副作用を扱えるようにしたりして、不純な一面を必要悪としてもっていた。しかし、Haskell ではこれを遅延ストリームモデルで扱い、純粋な関数型言語としてさまざまなプログラムを書くことができることを実証した。遅延ストリームの考え方は、端的にいえば、プログラムは、個別の入力データから出力を得る関数ではなく、時間経過を考慮した入力列から出力列への関数として捉えるということである。

求められるブレークスルー

いま、関数プログラミングに欲しいブレークスルー、その答えを得ることはきわめて難しいことであるが、浅学非才を顧みずに、大胆に…。

効率的な実行システム：

なんといっても、プログラムの効率的な実行を実現するシステムということであろう。これまでにも多くの試みがなされてきており、実際、実用的なシステムも存在する。場合によっては、伝統的な手続き型プログラムに匹敵するコード生成も可能である。しかし、ふつうの von Neumann 型計算機の機構と手続き型プログラムの間には密接な対応関係が見られるが、Backus の指摘を待つまでもなく、このような計算機と関数プログラムのセマンティックギャップは依然として大きい。このような問題点を解消しようとして、関数プログラムの計算に適した計算機構を実現するためのハードウェアの設計が試みられてきたが、標準的ハードウェアの発展の速度は、このセマンティックギャップをコンパイラによって埋める方向に向かわせつつある。プログラミングの立場から冷静にみれば、ありとあらゆる問題解決にひとつの言語やスタイルでことが足りるというわけでないこともあきらかであろう。関数プログラミング愛好家といえども、トイプログラムはともかく、実用的なシステムを関数プログラムだけで完全に作り上げられるとは考えていないのではなかろうか。さまざまな計算モデルの共存共栄というのが現代的なプログラミングであろう。そういう考え方に立てば、標準的な計算機アーキテクチャの上で関数プログラムを実行するという姿は当然の帰結である。

関数プログラミングの実践：

それでは、関数プログラミングの位置付けはどこにあるのか。10 年ほど前に、ある研究者・プログラマから、「関数プログラミングは論文題材の宝庫のようだ」といわれたことがある。実際、論文の主張する理論的な成果はともかく、プログラムという点からみれば、いつも関数 append がどうのこうの、とか、せいぜい、8-queens のようなトイプログラムが現れる成果というのでは、このような指摘もあながち的外れではなかろう。しかし、最近の 10 年ほどで、関数プログラミングの方向も変わってきた。ハードウェアの能力の向上とともに、昔は不可能であった実行システムが、遅いながらもプログラムの実証に役立つようになったからである。Backus の指摘した「von Neumann から解放される」プログラミングの試みができるようになったということができよう。Glasgow Haskell Compiler(GHC) [10] に見られるように、コンパイラ自体がそれで記述され、かなりの程度のコード生成を行なうようになったことも実用的プログラミングのための関数プログラ

ミングを刺激した。
すでに、関数プログラミングによる実用的なシステム開発の事例も多く出てきている[11]。「トイプログラムのための関数プログラミング」という迷信の払拭も、欲しいブレークスルーといえようか。同時に、関数プログラミングの考え方を広めることも求められよう。プログラムを書く人がいないのではプログラミングになりはしない。科学技術計算にも、システムプログラムにも、関数プログラミングは有効な手法であることが確信できるようにすることが「関数プログラミングのよさ」、「関数プログラミングの拓く未来」のために必要なことである。

プログラム運算：
関数プログラミング愛好家といえども、なにからなにまで、純粋な関数で表現することが(原理的にはできるにしても)現実的でないことは承知しているであろう。すべての物質が原子から成っているのと同様に、関数プログラムは基本構成要素を決まった方法で組み上げた式によって表現する。しかも、ある式の表わすものは、その構成要素だけから決まるという作用的な(applicative)意味づけが基礎となっている。それ以外の構成法はないので、動的な構成変更などは考えられない。たとえていえば、一般的なグラフに対して、その部分集合として素直な構造だけをもつ木の集合が関数プログラムということになろうか。こうした限界を認識し、それを活かすこともまた、求められるひとつのブレークスルーである。
関数プログラムは素直なだけに、扱いやすく、プログラム操作によっていろいろと改善できる。効率ということもあろうし、標準形を得るということもあろう。数式を変形するのとおなじように、プログラムを変形することのできるのは関数プログラムの特徴である。プログラムの運算(calculation)によって、どの程度まであらたなアルゴリズムの導出が可能であるか未知ではあるが、いくつかの成果が出ていることも事実である。プログラミングの世界におけるブレークスルーのひとつとも考えられよう。

これらのブレークスルーによって、関数プログラミングがλ(ラムダ)のように末広がりに発展することを期待しよう。

関数プログラミングに欲しいブレークスルー、当然のことといえようが、プログラミング一般の世界に共通のものがなくはない。当然のことではあるが…。ここにあげたブレークスルー、実現すればまた、あらたなものが欲しくなる。いつになっても欲しいブレーク

スルーはなくならない。ブレークスルーの皮肉的側面かも…。

[1] A. Church: The Calculi of Lambda Conversion. Princeton University Press, 1941.
[2] J. McCarthy: Recursive Functions of Symbolic Expressions and Their Computation by Machine. Comm. ACM 3 (1960), pp.184-195.
[3] P. J. Landin: The Next 700 Programming Languages. Comm. ACM 9 (1966), pp.157-166.
[4] J. Backus: Can Programming be Liberated from the von Neumann Style? ? A Functional Style and Its Algebra of Programs. Comm. ACM 21 (1978), pp.613-641.
[5] M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth: A Metalanguage for Interactive Proof in LCF. Proc. 5th ACM Symp. POPL, pp.119-130, 1978.
[6] D. A. Turner: Miranda- A Non-strict Functional Language with Polymorphic Types. Functional Programming Languages and Computer Architecture, LNCS 201, pp.1-16, Springer-Verlag, 1985.
[7] R. Bird, and P. Wadler: Introduction to Functional Programming. Prentice-Hall, 1988.(邦訳：武市正人訳「関数プログラミング」、近代科学社、1991). なお、原著の第2版が1998年に出版されている。
[8] A. Wikstrom: Standard ML. Prentice-Hall, 1988.
[9] P. Hudak, and P. Wadler, eds.: Report on the Functional Programming Language Haskell. Tech. Rep. YALEU/DCS/RR656, Dept. Computer Science, Yale University, 1988. これの改訂版V1.2がACM SIGPLAN Notices 27(1992), No.4にある。最近の情報は[10]から得られる。
[10] http://www.dcs.gla.ac.uk/fp/software/ghc/
[11] P. Wadler: An Angry Half-Dozen. ACM SIGPLAN Notices 33 (1998), pp.25-30.