

解説

関数型プログラムにおけるストリーム計算†



田 中 二 郎††

1. はじめに

ストリームとは、要素の値が先頭から順に時間の経過とともに順番に与えられていくような一次元状の値の列である。この値の列は有限の列でも良いし、無限であっても良い。たとえば 1, 2, 3, ... という列であれば、これは正の整数の無限ストリームを表している。

通常のプログラミング言語では、有限の列を記述することはできても、無限ストリームを表現することはできない。Lisp などのプログラミング言語では、リストというデータ構造をもつが、リストとはあらかじめ要素の値を計算し、それを束ねるデータ構造である。したがって有限の列しか記述できないし、要素の値がすべて揃っていないようなものは記述できない。

しかしながら、今後のプログラミング言語について考えた場合、こうした無限ストリームを言語の枠内で取り扱うことのメリットは大きいと考えられる。

歴史的にみた場合、こういったストリームの概念と関数型言語とは密接に結びついている。本稿では、このストリームについて、歴史的経緯から出発し、逐次型の関数型言語におけるストリーム計算や並列言語におけるストリーム計算について簡単に解説を試みる。

1.1 Landin のストリーム

最初にストリームの概念を言語の枠内に持ち込んだのは Landin である。彼は 1965 年、Algol 60 のプログラムの意味をチャーチのラムダ計算との対応で示すことを試みた¹⁾。プログラムの意味をラムダ計算で示すという試みが、このように早くから行われていたことは興味深い。この試みはやがて ISWIM²⁾ という関数型言語の提案へとつながっていくが、この Landin の論文には、現在の関数型言語において重要な概念、すなわち Continuation (継続) やストリーム、遅延評価、無限データ構造、値としての関数などがすでに提

案されていることに驚かされる。

さて、ストリームであるが、Landin は Algol 60 における for 文の意味を説明するためにストリームを導入した。すなわち for 文においては、たとえば、

```
for i:=1 step 1 until n do
  begin...end
```

などという文を実行すると、begin...end が n 回実行されることになるが、こうしたループにおける変数の値の変化を表現するのにストリームの概念が必要であった。for 文などではループにおける変数の値などをあらかじめすべて揃えておくわけにはいかないし、またループの計算自体が無限に続いて終わらないかもしれない。そのため Lisp のリストとは違った構成要素を考える必要があったのである。

Landin の記法ではストリームは 0 引数の関数であり、その定義域と値域は以下のように記述できる。

$nil \rightarrow element \times stream$

すなわち、ストリームは、nil を受け取って最初の要素と（残りのストリームを表す）ストリーム関数の対を返す特殊な関数である。このストリームを使えば、ストリーム処理の基本関数 hs, ts, prefixes は以下のように定義される。（これは Lisp でいえば CAR, CDR, CONS を定義することに相当する。）

```
def hs s = first(s())
def ts s = second(s())
def prefixes x s = λ(). (x. s)
```

ここでは、ストリームを s で、また引数 nil を $()$ で表している。したがって $s()$ はストリームの最初の要素とストリーム関数の対に相当している。また $\lambda(). f$ は引数 nil を受け取って f を返す 0 引数関数、 $()$ は対を示している。

こうしたストリームを用いれば、プログラムのループで生成される変数の値の変化を表現することはそれほど困難ではない。たとえば、“for $i:=a$ step b until c do” で生成される変数 i のストリーム step (a, b, c) は以下のように表現できる。

† Stream Computation in Functional Programming by Jiro T. NAKA (International Institute for Advanced Study of Social Information Science.).

†† 富士通国際情報科学研究所

```

def rec step (a, b, c) =
  λ(). if (a' - c') * sign (b') > 0
    then ()
    else (a'. step (λ(). a' + b', b, c))
  where a' = a()
        b' = b()
        c' = c()

```

ここで、**rec** はこの定義が再帰的な定義であることを示しており（すなわち **step** の定義に **step** が使われている）、 $\lambda()$ は 0 引数の関数を定義するのに使用されている。また **step** の引数 a, b, c は 0 引数の関数として与えられており、引数 **nil** を与えることにより、それぞれ値が計算されて a', b', c' になる。

なお、この変数 i のストリームは 0 引数の関数として定義されており、あらかじめストリームの要素を計算することではなく、必要に応じ引数 **nil** が適用されてその値が計算される。このような計算メカニズムのことを遅延評価あるいは要求駆動というのが、Landin はこの遅延評価を実現するための道具として 0 引数関数を使用したのである。

1.2 Friedman と Wise の遅延 CONS

さて、Landin により提案されたストリームではあったが、このストリームの概念と Lisp などにおけるリストとの関係については必ずしも明確ではなかった。この両者の関係を明確にしたのは Friedman と Wise の論文である³⁾。彼らは、Lisp で通常の CONS の代わりに引数の評価を行わない遅延 CONS を用いれば Landin のストリームが実現できることを示した。CONS は CAR 部分と CDR 部分の二つの引数をもつが、Landin のストリームは実は CAR 部分の引数を評価し、CDR 部分の引数の評価を行わない半遅延 CONS に相当することを明らかにした。また、両方の引数の評価を行わない遅延 CONS を仮定すれば、単にデータが一次的に並んだ場合だけでなく、より広いクラスの問題も扱うことが可能となることも示した。

Friedman と Wise の論文では遅延 CONS の例題として無限列 $1/1, 1/4, 1/9, \dots$ の例が示されている。

```

def (terms n) =
  (cons (reciprocal (square n))
        (terms (add1 n)))

```

ここで、**square** は与えられた引数を 2 乗し、**reciprocal** は与えられた引数の逆数を取る関数である。このストリームの要素の値は実際に要素の値がプログラム

によって必要とされるまで計算されない。たとえば $(\text{car} (\text{cdr} (\text{cdr} (\text{terms } 1))))$ などの場合は **terms** の第三要素 $1/9$ だけが計算され、第一要素や第二要素は計算されない。したがって、 $(\text{car} (\text{cdr} (\text{cdr} (\text{terms } 0))))$ などのように第一要素の計算が発散して値が出ないような場合でも答えを出すことができる。

この遅延 CONS によるストリームの表現を Landin の 0 引数関数を用いた方法と比較してみると、Landin の方法がやや技巧的な表現であるのに比べ、より自然なストリームの実現手段であるといえることができる。

1.3 ストリームの表現

一般に、プログラム言語のデータ構造の表現には、ストラクチャ・モデルとトークン・モデルの二つがあると考えられる⁴⁾。ストラクチャ・モデルとはデータの構造を CONS などで結合された構造として表現するもので、Lisp のリスト構造や Landin のストリームなどがこれにあたると考えられる。一方トークン・モデルとはデータの構造をデータ・トークンの流れとして表現するもので、いわゆる純粋なデータフロー・モデル⁵⁾がこれにあたる。

ストリームはトークン・モデルでは、データ・トークンの流れ、あるいはメッセージの流れとして解釈することができる。たとえばある入力データ列に一連の処理を施す場合など、入力データ列に対応するストリームを処理に対応するプログラム構造の中を流してやればよい。この場合、同じプログラム構造をストリームの各要素に対して繰り返し使うことができるのでメモリの効率が向上する。また、一つの処理が終わらないうちにパイプライン的に次の処理が開始できるので処理効率も向上する。

このようにトークン・モデルにも良い点はあるが、トークン・モデルには、ストリームのストリームが表せないなどプログラムの表現力に限界があるのでストラクチャ・モデルと併用されるのが普通である。

一方、ストラクチャ・モデルとはプログラムのデータ構造の表現としては通常の方法であり、ストリームもこのストラクチャ・モデルで表現することができる。ストラクチャ・モデルでストリームを記述する際の問題は、ストリームにおいては要素がすべて揃っているわけではなく、要素が先頭から順番に与えられ、無限列となる可能性がある点である。

こうした無限構造の表現としては、前にあげた Landin の **step** の定義や、Friedman と Wise の **terms** の定義のように再帰的な関数定義を利用するの

が通常であるが、もう一つの可能性としてはストリームを変数を使って表現し、それが徐々に具体化されていくようにする方法がある。

たとえば、1, 2, 3, 4, ... という無限ストリームであればそれを変数 X とおく。それを $X = (1, X')$ と具体化し、次に $X' = (2, X'')$ と具体化する。これをつぎつぎと繰り返していけば無限ストリーム 1, 2, 3, 4, ... を生成することができる。こうしたストリームの作り方は前にあげた再帰的な関数定義によるものと同じだが、変数が陽に記述されるところが相違点である。

2. ストリーム計算

さて、次の問題はストリームをどのようにプログラムの中で使うかという問題である。ストリームを使ったプログラムについては、Burge の文献⁶⁾にいくつかのストリーム処理用の関数が定義されている。一方、Henderson は、データ再帰により定義される循環構造を用いた無限ストリーム計算の例を示している⁷⁾。また、井田や筆者らは Backus の FP⁸⁾ にストリーム機能を組み込み、一般的なストリーム計算のパラダイムについて研究している^{9)~11)}。本章ではこれらのストリーム計算の動向について簡単に述べる。

2.1 Burge のストリーム計算

Burge は Landin と同様、ストリームを 0 引数の関数 (nil を受け取って最初の要素とストリームの対を返す関数) として定義している。以下、Burge の提案したいくつかのストリーム処理用の関数⁶⁾のうち興味深いものを紹介する。

```
def rec generate f x
  = λ(). (x . generate f(fx))
```

まずストリームを生成する関数がこの generate である。例えば、正の無限整数列を表すストリーム integer は、この generate を使い integer = (generate successor 1) と定義でき、integer() = 1, 2, 3, 4, ... となる。

```
def rec maps f s = λ(). (fx . maps f y)
  where (x . y) = s()
```

次にストリームのそれぞれの要素に与えられた関数 f を適用する関数が maps である。たとえば、(maps square integer)() = 1, 4, 9, 16, ... となる。

```
def rec filter p s =
  if p x
  then λ(). (x . filter p y)
  else filter p y
```

where (x . y) = s()

また、与えられたストリームからある条件 p を満たす要素だけを選びだし、新しくストリームを作る関数が filter である。

このほかにも Burge は、Lisp のさまざまなリスト処理用の関数からの類推で、さまざまなストリーム処理用の関数を定義している。しかしながら彼はリスト処理と同様なことがストリームでも可能であることを示しただけで、それから一歩踏み出してストリーム計算のパラダイムを提案するには到らなかった。

2.2 Henderson のストリーム計算

ストリーム計算のメリットを示したという点で注目できるのは、Henderson のストリーム計算 (あるいは循環構造計算)⁷⁾ の提案である。Henderson や Keller¹²⁾ はストリーム計算のメリットとして、循環構造をうまく利用することにより、同一の計算の繰り返しを避けることができることをあげている。ここではそれらの動向について簡単に解説する。

たとえば、再帰的プログラミングの例として良く使われる例題に以下の Fibonacci 数のプログラムがある。

```
Fib(0) = 1
Fib(1) = 1
Fib(i) = Fib(i-1) + Fib(i-2), i > 1
```

このプログラムは再帰的プログラムの例としてはあまり良いとはいえない。なぜなら Fib(n) を計算するとき、たこ足的に Fib が呼ばれるからである。これでは同一の引数に対して同じ計算が繰り返されることになり効率が良くない。

この効率の悪さを防ぐにはさまざまな方法が考えられる。たとえば、一度呼ばれた関数呼び出しとその計算結果をどこかに表の形で記憶しておき、次からはそれを使う方法もある (素表計算)。ここでは Fibonacci 数の無限ストリームを使う方法^{7), 12)} について紹介する。(この方法は、ある意味で無限ストリームを表の代わりに使うもので、一種の素表計算といえないこともない。)

いま Fibonacci 数の無限ストリームを Fib とすると、

```
Fib = Fib(0), Fib(1), Fib(2), ...
```

ここで上記の Fibonacci 数の定義の 3 行目の i に数を代入すると以下の関係が成り立つ。

```
Fib(2) = Fib(1) + Fib(0)
Fib(3) = Fib(2) + Fib(1)
```

$$\text{Fib}(4) = \text{Fib}(3) + \text{Fib}(2)$$

$$\vdots \quad \vdots \quad \vdots$$

これらの式を統合すれば,

$$\text{cdr}(\text{cdr}(\text{Fib})) = \text{cdr}(\text{Fib}) + \text{Fib}$$

ただしここで $+$ はストリームの要素ごとの加算であることに注意されたい. この式に Fibonacci 数の定義の1行目と2行目を加えれば, 上の式は,

$$\text{Fib} = 1 \wedge 1 \wedge (\text{cdr}(\text{Fib}) + \text{Fib})$$

ようになる. なおここで $\wedge$ は遅延 CONS を中置記法 (infix notation) で表したものである.

この式は見た目には多少変わって見えるが, これで無限 Fibonacci ストリームの定義になっている (こういった定義をデータ再帰と呼ぶ). この Fib を図に示したのが図-1 である. この図で分かるように Fib は循環構造になっている.

この無限ストリームでは, ストリームの何番目かの要素が呼ばれると, 必要に応じてこの循環構造が計算される. (データ再帰は遅延 CONS を想定して初めて意味をもつことに注意されたい.) 一度計算された構造は値に置き代わってしまうので, この無限ストリームを使えば, 再帰的なプログラムの呼び出しの際に生ずる再計算を防ぐことができる.

Henderson はこうした循環構造の例としてほかに Hamming の問題¹³⁾ を解くプログラムをあげている. これは以下のような問題である.

Hamming の問題

2^a 3^b 5^c の形 ($a, b, c = 0$) の数を順に小さいものから並べた数の列を作れ.

この問題は, そもそも Kahn¹⁴⁾ によって解かれた問題であるが, これをストリームを使って解くと以下のようになる.

```
hamming = X
```

```
where rec
```

```
  X = (1 . X235)
```

```
  X235 = merge(X23, mul5(X))
```

```
  X23 = merge(mul2(X), mul3(X))
```

ここで merge は, 入力として要素を小さい順に並べた二つのストリームを取り, 出力としてその二つのストリームを合流し要素を小さい順に並べたストリームを出力するような関数である. この merge は以下のように記述できる.

```
def rec merge(s1, s2) =
```

```
  if x < y
```

```
  then (x . merge(sx, s2))
```

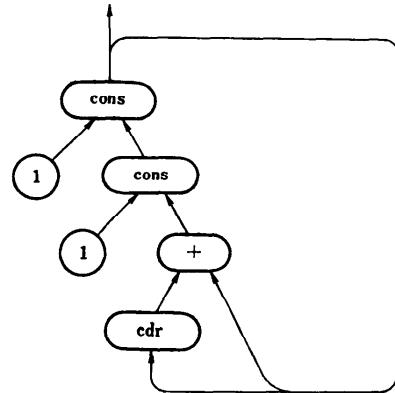


図-1 Fibonacci ストリームの定義

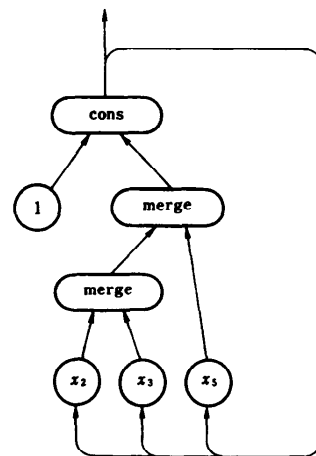


図-2 Hamming の数列

```
else (y . merge(s1, sy))
```

```
where (x . sx) = s1
```

```
(y . sy) = s2
```

また, mul2, mul3, mul5 はストリームの要素の値をそれぞれ2倍, 3倍, 5倍する関数である.

この数列を求めるプログラムを図に示したのが図-2 である. この図も循環構造になっており, 最初は要素1から出発し, その要素をそれぞれ2倍, 3倍, 5倍しながら数列が作られていくことが分かる. こうした循環構造の応用については Keller がいくつかの提案を行っている¹²⁾. 一つは古くから Dynamo¹⁵⁾ などで試みられてきた相互作用する系のシミュレーション, また属性文法による parsing の問題などのように評価の順序が複雑に絡み合っている系の評価, もう一つはプログラムをどの程度並列に評価するかの動的な制御などである.

2.3 井田、田中のストリーム計算

このようにストリームについてはさまざまな視点から研究がなされていたが、それらはストリームを使っている新しいプログラミング・パラダイムの提案とまではいかなかった。

ストリームによるプログラミングの特徴を考えてみると、再帰関数によるプログラミングがスタック的であるのに較べ、ストリームによるプログラミングはキュー的もしくはパイプライン的であることに気付く。こういったストリームの特性を生かすようなプログラミング・パラダイムはないかということが問題となる。

これに関連して Wadler が、有限リストについてプログラムをリストの生成、変換、消滅という観点でとらえ、リストのへる変換過程をプログラム変換により縮退させるような研究¹⁶⁾を行っていた。

この Wadler の研究にヒントを得て、井田や筆者らは、それをストリームに応用し、無限ストリームを言語レベルでサポートするような方式について提案^{9)~11)}を行った。またそれらの機能を Backus の提案した関数型言語である FP⁸⁾に組み込んだ。

FP ではファンクションとデータ・オブジェクトとははっきり区別されており、ファンクションはプログラム、データ・オブジェクトは入力データにそれぞれ対応し、他の関数型言語のように両者が混じり合うことはない。筆者らはストリームをストリーム・オブジェクトという特殊なデータ・オブジェクトとして実現した。

ストリーム計算のパラダイムを図に示したのが図-3である。

すなわちストリーム計算の典型的な例は、

reducer : transformer : generator : seed

と書ける。ストリーム・オブジェクトは、ストリームの種となる seed に generator を適用することにより生成される。生成されたストリーム・オブジェクトは、transformer により変形され、やがて reducer により消滅させられて出力値にかわる。このストリーム・オブジェクトの変遷の過程がすなわち計算というわけである。ただし、ここで注意する必要があるのは、generator が生成するのはストリーム・オブジェクトであって、実際のストリームの要素を計算するわけではない。(generator は必ず無限ストリームを生成するのでそもそもそのすべての要素を計算するのは不可能である。) また transformer が変形するのもストリーム・オブジェクトであり、reducer ではじめてス

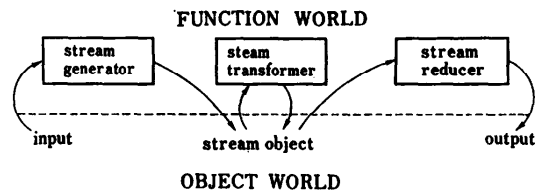


図-3 ストリーム・プログラミング

トリーム・オブジェクトから要素の値が計算されるのである。

さて generator であるが、generator は $\langle \rangle$ で表され、生成関数 g と選択関数 s が与えられる必要がある。たとえば $\langle g, s \rangle$ という generator にストリーム・シード i を適用すると、ストリーム・オブジェクト $\{g, s, i\}$ が生成され、これは

$s:i, s:g:i, s:g^2:i, s:g^3:i, \dots$

という無限ストリームに対応すると考えられる。ここで選択関数 s を仮定したのは、ストリームのもつ内部状態と出力とが必ずしも一致しなくてもよいと考えたからである。

こうして選択関数 s を仮定すれば、ストリームの各要素に関数 f を適用するようなストリームの変換には、 $(\text{map } f): \{g, s, i\} \rightarrow \{g, f \circ s, i\}$ のようにストリーム・オブジェクトの選択関数だけを変えればよくなる。この map はストリームの transformer と考えられるが、われわれはストリームの transformer として、このほかにもストリームの合流や分配などに関しさまざまなものを用意した。

また、ストリームを消滅させ出力値を計算する reducer としてはストリームの先頭の n 個を取り出しそれから出力値を計算するものと、ある条件を満たすまでストリームの先頭の要素を取り出し出力値を計算するものの二種類を用意した。(これは逐次プログラミングにおける DO 文、WHILE 文にそれぞれ対応する。)

こうして、井田や筆者らは、ストリームの生成、変換、消滅の各種の関数を提示し、ストリームのライフ・サイクルをプログラミング・パラダイムとして実現するようなプログラミング方式について提案を行い、各種のストリーム・プログラミングの例題を提示した^{9)~11)}。

3. 並列プログラムにおけるストリーム

前章までは主として逐次プログラムにおけるストリーム・プログラミングについて解説した。こうした

逐次プログラミングにおいては、ストリームの生成・変換・消滅といった三つの側面が重要な役割を演じたが、ここでストリームを生成・変換・消滅させる主体をプロセスと考え、プロセスを主体としてストリーム・プログラミングを捉えなおしてみることも可能である。そのときプロセスを逐次的に作用させるのではなく、プロセスは常に存在し、ほかのプロセスからのメッセージを受け取って作動する（ここでストリームはメッセージの列に対応する）と考えれば、プログラムは逐次プログラムではなく並列プログラムとなる。

このような計算システムについてはさまざまな提案がなされており、一つ一つのプログラムの記述方式（逐次プログラムか並列プログラムか）、メッセージの送出・受取方式（送出と受取が同期するか、また同期の方式）、動的にプロセスの生成を許すか、などでさまざまなバリエーションが可能である。すでに提案されたこの種の計算システムとしては、Kahn の並列プロセスのネットワーク^{14), 17)}, Hoare の CSP¹⁸⁾, Actor ふうのオブジェクト指向言語¹⁹⁾, 各種の並列論理型言語^{20)~22)}などがこれに該当する。

また手近に利用可能な並列ストリーム言語として、Pascal にストリームを組み込んだ、中田、久世らによる Stella^{23)~25)} がある。ここでは Pascal プログラムであるモジュール群がストリームを介して相互に結合され、それらのモジュール群はそれぞれストリーム通信を行いながら逐次的に実行される。

本稿ではこれらの試みの中から、Kahn の並列プロセスのネットワークと、並列論理型言語で探索問題を解く試みの中で開発された技法であるレイヤード・ストリーム²⁶⁾について特に述べる。

3.1 Kahn の並列プロセスのネットワーク

並列プロセスが同時に起動しながら、相互にメッセージを交換しつつ計算を進めていくような計算システムの提案を最初に行ったのは Kahn^{14), 17)}である。彼は、一つ一つは Algol ふうの逐次プログラムであるプロセスが、同時に起動しながら計算を進めていくような計算システムの提案を行った。ここでは一つ一つのプロセスは、いわば入出力のような形で、他のプロセスにメッセージを出力できる、また他のプロセスからメッセージを受け取ることができる。

Hamming の問題についてはすでに述べたが、この問題についても、前章で扱ったような遅延評価によるストリーム計算を主体とした立場ではなく、むしろストリームを合流する merger やストリームの各要素を

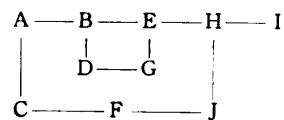
整数倍する multiplier をプロセスと考え、そうしたプロセスのネットワークがメッセージをやりとりしながら計算を行うと考えたほうが自然であろう。（事実、Kahn によって最初に示された解法¹⁴⁾はそのような方法であった。）また彼はこのような並列プロセスのネットワークの意味に関しても考察を行った¹⁷⁾。

3.2 レイヤード・ストリームによる探索問題のプログラミング

並列プログラムにおけるストリーム計算について考えた場合、どのような問題をどのように解くかが問題となる。これにもさまざまな可能性が考えられるが、ここでは探索問題について考える。探索問題とは人工知能の分野でよく扱われる問題で、通常は、解の候補を生成し、その中から条件に合った解を選びだすような問題である。ここでは探索問題の例として、以下のようなグッドパス問題をレイヤード・ストリームを用いて解いた例²⁶⁾について考える。

グッドパス問題

下の図のようなグラフ上の指定された二点間を結び、ループを含まない（同じ点を二度通らない）経路を（すべて）みつつけよ。



この問題のモデル化の仕方であるが、ここでは与えられたグラフの（始点以外の）ノードをプロセスと考え、プロセスどうしが探索の情報をメッセージとして交換すると考える。こうしたモデル化は現実の事象そのものをプロセスと考えるわけできわめて自然なものと思われる。

いま、Aを出発してIに到るグッドパスを求めるとき、与えられたグラフは以下のようにモデル化できる。

goodpath=I

where rec

A=a

B=node (b, (A E D))

C=node (c, (A F))

D=node (d, (B G))

E=node (e, (B G H))

F=node (f, (C J))

G=node (g, (D E))

H=node (h, (E I J))

$I = \text{node}(i, (H \ J))$

$J = \text{node}(j, (F \ H))$

ここで小文字 $a \sim j$ はノード名を表している。また大文字 $A \sim J$ は (レイヤード・ストリームに具体化され) 探索情報を持ち運ぶ変数であってノード間の道に対応している。(すなわち小文字は定数, 大文字は変数に対応している。)

この問題の解法としては出発点から探索を開始し, 隣接点にそれらが未到達であることを確認しながら進み, 終着点につくまで探索するというのが通常の解法であろう。ここではこういった探索に対応して, まず始点の a では " $A=a$ " でパスに a という情報を入れて流す。各ノードではパスの情報を受け取ると, いままでの部分パスがグッドパスになっているかのチェックをしたあと自分のノード情報を加え隣接したノードにわたす。たとえばノード b の処理であれば " $B = \text{node}(b, (A \ E \ D))$ " と表せる。そして終点の i に流れるストリーム I が解となる。

このようなアルゴリズムを素直に実現すると, それぞれのパスには, それまでに作られた部分パスのストリームが流れることになる。しかしながらこうした部分パスには似たようなものも多く, そのそれぞれに部分パスのチェックをするのは大変である。そこでこういった似たような部分解が多く現れる探索問題用に特に提案されているのがレイヤード・ストリームである。(このレイヤード・ストリームはそもそも並列論理型言語で提案された概念であるが, レイヤード・ストリームの本質は部分解構造の共有にあり, 特に並列論理型言語の特性を使用しているわけではないので本稿では関数型言語の記法を用いて説明を行う。)

レイヤード・ストリームは共有構造ストリームとも呼べるものであり, 一般には $(X_1 \ X_2 \ \dots \ X_n)$ の形をしている。ここで X_i は単なる定数か, もしくは $S_i * L S_i$ のような形をしている。ここで S_i は定数, $L S_i$ は再びレイヤード・ストリームである。* は S_i と $L S_i$ の各要素のつながりを表現する形式であるが, ここで $()$ と * について数学の分配則のようなものが成り立っていると解釈できる。ただし * の後ろが空リスト $()$ であるときには, その構造は空リストを表現していると考えられる。たとえば $(1 * (2 * (3 \ 4 * ()) 5))$ は通常のリストでは $((1 \ 2 \ 3) (1 \ 5))$ に相当する。

このレイヤード・ストリームを使えば, 解を探索する過程で現れる共有構造を $()$ で括りだすことができ, なおかつ部分解のテストはこの括りだされた構造

について一回だけやればよい。

これを見るかぎり, レイヤード・ストリームは一種のリストのように見えるが, 上の構造にしても, すべての構造が同時に与えられるわけではなく, レイヤード・ストリームに対応する変数が, 最上層から計算の進行とともに徐々に具体化されていくのであり, これはストリームの一種と考えられる。

このレイヤード・ストリームを用いれば, node のプログラムは以下のように記述できる。

```
node(X, List) = X * filter(X, List)
filter(X, List) =
  if List = () → ()
  if List = ((). In) → filter(X, In)
  if List = (a . In)
    → (a . filter(X, In))
  if List = (X * Xs . In) → filter(X, In)
  if List = (Y * Xs . In), X ≠ Y →
    (Y * filter(X, Xs) . filter(X, In))
```

すなわち node は, まず filter で List のレイヤード・ストリームから不都合な部分パスを読みとばし, 自分のノード名を $X *$ の形で付け加える。

また filter はレイヤード・ストリームになっている List のそれぞれの要素について, グッドパスになっているかどうかの検査を行っている。(すなわち List やその要素が nil であれば読みとばす。要素が出発点 a であればグッドパスになっている。レイヤード・ストリームが自分のノード名 X を含めばそのパスを読みとばす, 含まなければグッドパスになっている。)

なお, ここで示したプログラムは各 node や filter が並列に作動することを前提としている。このプログラムでは filter の条件部でパターンマッチを行っているが, List の内容のデータが到着していないときにはプログラムはそこでデータ待ちの状態に入り, データが到着ししだい計算を再開させる必要がある。いいかえればプログラミング言語自体が何らかの同期のメカニズムをもつ必要がある。

またこのプログラムを逐次プログラムと比べると, ここではストリームの生成→変換→消滅の順序関係がプロセスの間で明らかではない。実際の計算は出発点のノード a から始まるのではなく, むしろ並列プロセスがメッセージのキャッチボールをしながらトータルとしての計算を行っていくというのに近い。レイヤード・ストリームにしても最上層から計算の進行に従い, 時間とともに徐々に変数が具体化されていくの

である。

このレイヤード・ストリームを使えば、一般の探索問題についても効率よく問題を解くことができる。奥村らは、ここに示したグッドパスの問題のほかにも、 n クイーン問題、4色キューブ問題や構文解析の問題を解いている^{26), 27)}。

4. ま と め

以上、関数型プログラムにおけるストリーム計算について現状を概観した。前半では逐次型の関数型言語におけるストリーム計算について、また後半では並列プログラミングにおけるストリーム計算について解説を行った。

ここで関数型プログラミングにおけるストリームの役割についてまとめてみると、ストリームには大きく二つの役割があることが分かる。それは「関数型言語への順序関係の導入」及び「データの流れにしたがったプログラムのパイプライン処理」の二つである。

関数型言語は通常の Fortran や C などのプログラミング言語とは異なり表現指向 (expression-oriented) であり、コントロールの流れをプログラムの実行文の列としてプログラムに陽に記述できない。そのため順序関係をストリームとして記述しなければならない。

また、レイヤード・ストリームの例で示したようにストリームは単に部分解の候補を流すために使われることがある。この場合はデータが各プロセスで並列あるいはパイプライン処理され、ストリームを流れるデータの順序は本質的ではない。

こうしてストリームの役割を煮詰めていくと、同時に現在のストリーム計算の問題点も明らかになってくる。それらは 1) ストリームは順序関係として、全順序関係をもつがそれでは強力すぎないか、2) ストリーム計算を実現するための言語としては究極には並列言語であることが望ましいが、その場合プロセス間の連絡や同期の方法としてどのようなものが望ましいか、などである。

このうち 1) については Tribble らはストリームの一般化として、全順序の代わりに半順序関係を導入し、一つのストリームに複数の書き込み、読み出しを許すようなチャンネル²⁸⁾という概念を提唱している。2) についても、ストリーム処理専用言語ではないが、本稿でも紹介したようなさまざまな並列言語が提案されインプリメントされつつある。

計算機ハードウェアの研究開発が、今後、さまざま

な形の並列・分散ハードウェアの開発にシフトしていくことを考えれば、こうした並列言語や、さらには本稿で述べたストリーム計算を促進するような並列ストリーム処理専用言語について、一層の研究開発が進められる必要があろう。

参 考 文 献

- 1) Landin, P. J.: A Correspondence between Algol 60 and Church's Lambda-Notation: Part I, Comm. ACM, Vol. 8, No. 2, pp. 89-101 (1965).
- 2) Landin, P. J.: The Next 700 Programming Languages, Comm. ACM, Vol. 9, No. 3, pp. 157-166 (1966).
- 3) Friedman, D. P. and Wise, D. S.: CONS Should Not Evaluate its Arguments, Automata, Languages and Programming, Edinburgh University Press, pp. 257-284 (1976).
- 4) Davis, A. L. and Keller, R. M.: Data Flow Program Graphs, Computer, Vol. 15, No. 2, pp. 26-41 (1982).
- 5) Dennis, J. B.: First Version of a Data Flow Procedure Language, Lecture Notes in Computer Sci., Vol. 19, Springer-Verlag, pp. 362-376 (1974).
- 6) Burge, W. H.: Recursive Programming Techniques, Addison-Wesley (1975).
- 7) Henderson, P.: Functional Programming Application and Implementation, Prentice-Hall (1980).
- 8) 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).
- 9) Ida, T. and Tanaka, J.: Functional Programming with Streams, in Information Processing 83, North Holland, pp. 265-270 (1983).
- 10) Ida, T. and Tanaka, J.: Functional Programming with Streams—part II—, New Generation Computing, Vol. 2, No. 3, pp. 261-275 (1984).
- 11) Tanaka, J. and Ida, T.: Stream Objects in a Functional Language, Research Report No. 56, International Institute for Advanced Study of Social Information Science, Fujitsu (1985).
- 12) Keller, R. M. and Lindstrom, G.: Applications of Feedback in Functional Programming, in Proc. of 1981 Conference on Functional Languages and Computer Architecture, ACM, pp. 123-130 (1981).
- 13) Dijkstra, E. W.: A Discipline of Programming, Prentice-Hall, New Jersey (1976).
- 14) Kahn, G. and MacQueen, D. B.: Coroutines and Networks of Parallel Processes, in Infor-

- mation Processing 77, North-Holland, pp. 993-998 (1977).
- 15) Forrester, J. W.: Industrial Dynamics, The MIT Press (1961).
- 16) Wadler, P.: Applicative Style Programming, Program Transformation and List Operators, in Proc. of 1981 Conference on Functional Languages and Computer Architecture, ACM, pp. 25-32 (1981).
- 17) Kahn, G.: The Semantics of a Simple Language for Parallel Programming, in Information Processing 74, North Holland, pp. 471-475 (1974).
- 18) Hoare, C. A. R.: Communicating Sequential Processes, Comm. ACM, Vol. 21, No. 8, pp. 666-678 (1966).
- 19) 米澤: オブジェクト指向型プログラミングについて, コンピュータソフトウェア, Vol. 1, No. 1, pp. 29-41 (1984).
- 20) Clark, K. and Gregory, S.: PARLOG: Parallel Programming in Logic, Research Report DOC 84/4, Dept. of Computing, Imperial College of Science and Technology (1984).
- 21) Shapiro, E.: A Subset of Concurrent Prolog and Its Interpreter, ICOT Technical Report TR-003, ICOT (1983).
- 22) 古川, 溝口編: 並列論理型言語 GHC とその応用, 共立出版 (1987).
- 23) Nakata, I. and Sassa, M.: Programming with Streams, IBM Research Reports, RJ 3751 (1983).
- 24) 久世和資: ストリームを扱う言語 Stella による在庫管理システムの記述, 情報処理, Vol. 25, No. 5, pp. 497-505 (1985).
- 25) Kuse, K., Sassa, M. and Nakata, I.: Modelling and Analysis of Concurrent Processes Connected by Streams, Journal of Information Processing, Vol. 9, No. 3, pp. 148-158 (1986).
- 26) 奥村, 松本: レイヤードストリームを用いた並列プログラミング, The Logic Programming Conference '87, ICOT, pp. 223-232 (1987).
- 27) 奥村, 松本: レイヤードストリームを用いた並列構文解析, 情報処理学会第 35 回全国大会論文集 5R-2, 情報処理学会 (1987).
- 28) Tribble, E. D. et al.: Channels: A Generalization of Streams, Logic Programming: Proceedings of the Fourth International Conference, Vol. 2, The MIT Press, pp. 839-857 (1987).

(昭和 63 年 7 月 1 日受付)