キューマシンにおける不動点プログラミング

山口 文彦

キューマシンは先入れ先出しの記憶装置を持つ抽象計算機である。キューマシンは演算の並列実行に相性が良いとされている一方で、関数などの部分プログラムの呼出しを含むプログラムを単純なキューマシンの命令列で表現することに困難がある。ここで、部分プログラムはインライン展開し、再帰的な部分については不動点を用いることで、さまざまな計算手続きが単純なキューマシン上で表現できるのではないかと考えた。しかし、キューマシンにおける不動点プログラミングについて知見がない。そこで本稿では、スタックマシンにおける不動点プログラミングを参考に、キューマシンにおける不動点プログラミングについて考察する。

Queue machine is a type of computer which has first in, first out memory. Queue machine have good potential to concurrent execution, however, there is a difficulty on describing function call in simple queue machine codes. This paper discusses fixed point programming on simple queue machine, which may enable representing various calculation including recursive programs.

1 はじめに

キューマシン [1] は先入れ先出しの記憶装置 (キュー) を持つ抽象機械である。演算子を節とし演算子の引数を子とする木 (以下、これを計算の木と呼ぶ。単純な式の場合は構文木に等しい) を考えたとき、キューマシンの命令列は、計算の木を幅方向に巡回することで得られる。したがって、ある演算や関数の引数となる演算のうちで最上位のものが、命令列中に続けて登場する。副作用のない純粋な関数型言語においては、演算や関数の複数の引数は並列に計算することができることが知られているが、キューマシンにおいては、このような並列に計算できる演算が命令列の中に連続して現れる。このことから、キューマシンは細粒度の並列実行を考える際の計算モデルとして有用であると考えられる。

しかし、関数呼び出しなどのように、計算の途中で計算の木の葉ノードが別の木に展開されるような仕組みを単純なキューマシンで実行することは難しい。なぜなら、葉の一つが高さのある木に置き換わるとすると、置き換えられる木の節 (葉を含む) は命令列中の離れた箇所 (それはすでに実行済の箇所であるかもしれない) に挿入されなければならないからである。

元々提案されたキューマシンアーキテクチャ [1] においては、命令キューを引数と呼出元の依存関係を保持したままセグメントに分割し、子セグメントが全て実行済みになるまで親セグメントの実行を待つという同期の仕組みを導入している。これによって、関数呼び出しが実行されると、その関数の実行が終了するまで、関数の呼び出し元が待機することになる。同じセグメントに配置された命令は並列に実行できるため、並列性を高めるためには、セグメントはできるだけ大きい方がよい。極論として、計算の木の高さごとに高々一つのセグメントしかないとすれば、それは単純なキューマシンと変わらない。

本稿では、単純なキューマシンに立ち返り、計算を表現する方法について考える。関数呼び出しが大きなオーバーヘッドとなっていることから、実行前にすべての関数呼び出しを展開しておくことを考える。再帰

---

* Fixed-point Programming on Queue Machine.

Fumihiko YAMAGUCHI, 長崎県立大学情報システム学部情報セキュリティ学科, Dept. Information Security, University of Nagasaki.

的な関数が使われていなければ、この展開によって計算の木が作られ、キューマシンの命令列を完成することができると考える。このとき、再帰的な関数の扱いが課題として残る。ここで、定義された関数の呼び出しを使わずに再帰的な計算を表現することができれば、課題の解決につながる。そのような計算の表現として、不動点を用いた計算の表現がある。関数型言語においては、匿名関数を用いた式として、再帰的な計算を表現することができる。しかし、キューマシンにおける不動点プログラミングについては知見がない。手掛かりとして、スタックマシンにおける不動点プログラミングが挙げられる。とくに、スタックを用いたインタプリタ言語である Postscript において、関数不動点に相当するプログラムを書くことができ、それによって `def` (ブロックに名前を付ける、定義のための命令) を用いずに再帰的な計算が表現できることが分かっている [5]。本稿ではキューを用いた計算を表現する言語である FIFOL を、適宜拡張しながら議論する。FIFOL は Postscript に似た言語である。

以下、キューマシン、不動点およびスタックマシンにおける不動点プログラミングについて述べたのち、キューマシンにおける不動点プログラミングの例をいくつか示し、課題を明らかにする。

2 研究の背景

2.1 キューマシン

ある式の値を計算するキューマシンの命令列は、その式の構文木を巡回して得られる。構文木の巡回によって命令列が得られる点では、スタックマシンと同様であるが、スタックマシンの命令列が構文木を深さ優先で巡回したときの帰りがけ順にノードを記録して得られる (式をこのように記述する方法は逆ポーランド記法として知られている) のに対して、キューマシンの命令列は構文木を下方から幅優先で巡回する順にノードを記録することで得られる。例として中置記法で `2*3+(2+5)*4` と書かれる式の構文木を図 1 に挙げ、スタックマシンおよびキューマシンの命令列として表現する場合の構文木のたどり方と、それぞれの命令列を示す。

図 2 には、図 1 に示したキューマシンの命令列

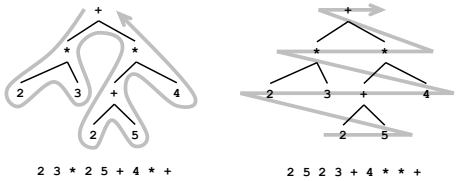

図 1　スタックマシンの命令列 (左) と キューマシンの命令列 (右)

データキュー	命令キュー
[]	[2 5 2 3 + 4 * * +]
[2]	[5 2 3 + 4 * * +]
[2 5]	[2 3 + 4 * * +]
[2 5 2]	[3 + 4 * * +]
[2 5 2 3]	[+ 4 * * +]
[2 3 7]	[4 * * +]
[2 3 7 4]	[* * +]
[7 4 6]	[* +]
[6 28]	[+]
[34]	[]

図 2　２５２３＋４＊＊＋ の実行の様子

２５２３＋４＊＊＋ が実行される様子を示している。図 2 の各行は、ある時点のデータキューと命令キューが書かれている。いずれのキューも、中身の列が [ ] で括られ、右から入って左から出るものとする。実行の 1 ステップでは、命令キューの先頭を取り出し、それがデータならデータキューに格納し、演算子なら、必要なだけのデータをデータキューから取り出して、演算結果をデータキューに格納する。命令キューが空になった時点で、データキューには計算結果が格納されている。

FIFOL はキューを使った計算を記述するプログラミング言語であり、2001 年夏のプログラミング・シンポジウムにおいて課題として (パズルとして) 登場した [2]。そのインタプリタの scheme による実装が公開されている [6]。

キューマシンの並列性に着目した研究として、FIFOL の SIMD 拡張に関する研究 [4] がある。この中でベクトル長さやマルチスカラの繰り返し数につい

ても言及があるが、これらを可変にする繰り返しの構造については未解決となっている。

2.2 不動点プログラミング

関数 F の不動点とは、$x = F(x)$ となるような x のことである。ここで、関数を引数とする関数の不動点によって、再帰的な関数を表現することができる。このことは次のように説明される。

再帰的な関数 f はその定義の中で自分自身である f を呼び出す関数である。f が、$f \equiv D$ と定義されるとすると、この f の定義式の本体 D の中には識別子 f が自由に登場する。この f を λ 抽象して得られる $\lambda f.D$ という関数を考える。この関数に x (この x は関数である) を適用して $(\lambda f.D)x$ とすると、これは元々の f とほぼ同様な計算をしながら、f を再帰呼び出しする代わりに x を呼ぶ計算を表す。ここで、元の再帰的な関数 f は、$f \equiv D$ と定義される関数であるから、$x = (\lambda f.D)x$ を満たす x と同じ計算を表す。すなわち、f は $\lambda f.D$ の不動点である。

純粋な λ 計算においては、式 F の不動点を得る不動点演算子を構成することができる。不動点演算子としては、Y オペレータ $(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))$ などが知られている。

多くのプログラミング言語で、関数などの部分プログラムを定義する仕組みが用意されており、再帰的な計算を行うときには、そうした仕組みを使って、再帰的な部分プログラムを定義することが一般的である。しかし、不動点を用いることで、再帰的な計算を一つの式として扱うことができる。関数の呼び出しを式に置き換えるという点で、不動点プログラミングは、インライン展開と同様な意味合いを持つものと考えられる。

2.3 スタックマシンにおける不動点プログラミング

スタックマシンの例として、Postscript インタプリタ [3] が挙げられる。

Postscript には、命令の列をブロックという形でデータとしてスタックに積む機能がある。ブロックは、命令列を { } で括って表す。これは、条件分岐を表す際や命令列に名前を付ける (定義する) 際などに使われる。また、スタックの先頭にあるブロックを実行する (実行すべき命令列の先頭にブロックの中身を追加する) `exec` という命令が用意されている。

スタックマシンでは、関数の引数の相当するデータをスタックに積んで計算を行う。そこで、ブロックをスタックに積めば、計算手順を引数とする計算を表現することができる。そのような計算の不動点であるような計算手順を構成することで、再帰的な関数に相当するプログラムを書くことが可能であることが分かっている [5]。具体的には Postscript における不動点プログラミングは次のように説明される。

F, G を Postscript の命令列であるとする。F が G の不動点であることを、$\{F\}G$ と F が同じ計算をすることであるとする。G の中には、スタックの先頭に積まれた $\{F\}$ を実行するための `exec` があると考えられる。このとき、G 中のこの `exec` を `dup exec` に置き換えて得られる命令列を G' とすると、命令列 $\{G'\}\ G'$ は G の不動点となっている。このことを示すには、$\{\{G'\}\ G'\}\ G$ と $\{G'\}\ G'$ が同じ計算をすることを示せばよい。前者はスタックに $\{\{G'\}\ G'\}$ を積んで G を実行し、後者はスタックに $\{G'\}$ を積んで G' を実行する。ここで、G と G' はスタックの先頭の積まれたブロックに対する動作だけが異なる。前者は $\{\{G'\}\ G'\}$ に対して `exec` を行うので、ここで $\{G'\}\ G'$ の計算が行われる。後者は、$\{G'\}$ に対して `dup exec` が行われるので、(スタックの先頭に $\{G'\}$ を二つ積んで `exec` を行うので) やはり $\{G'\}\ G'$ の計算が行われる。したがって、$\{\{G'\}\ G'\}\ G$ と $\{G'\}\ G'$ は同じ計算をする。以上から $\{G'\}\ G'$ は G の不動点となっていることが分かる。なお、$\{G'\}\ G'$ は $\{G'\}$ `dup exec` と書くことができる。

この方法を使って、Postscript において、`def` を用いることなく、再帰的な計算を表現することができる。なお、上記のように不動点を構成することはできるが、Postscript の命令列からその不動点を得る不動点演算子を Postscript で書くことはできていない。

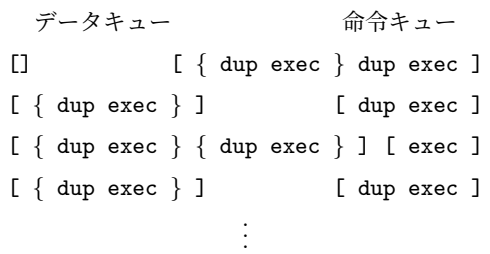

図 3 { dup exec } dup exec **の実行の様子**

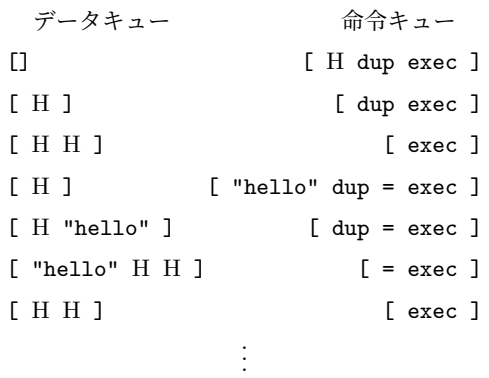

図 4 { "hello" dup = exec } dup exec **の実行の様子**

3 キューマシンにおける不動点プログラミング

キューマシンインタプリタである FIFOL には、Postscript におけるブロックと同様に、命令列をキューに格納する構文が用意されている。これを用いて、if などの条件分岐や loop という繰り返しが実行される。しかし、キューの先頭にある命令列を命令キューに入れる命令は存在しない。ここでは、そのような exec という命令を追加して FIFOL を拡張し、この拡張された FIFOL 上で不動点プログラミングを考える。

3.1 単純なループ

FIFOL の命令 dup は、キューの先頭から要素を取り出して、取り出した要素を 2 回キューに格納する。したがって、{ dup exec } dup exec という命令列は、命令列 { dup exec } をキューに格納して、これを複製したのち、キューの先頭の命令列を実行する。こうして exec によって dup exec が実行されるとき、キューには { dup exec } が格納されているので、実行を無限に繰り返すプログラムとなる。また、このプログラムを実行したときのデータキューと命令キューの様子を図 3 に示す。

これはただ無限ループするだけのプログラムだが、キューマシンにおいても (FIFOL には exec の導入という拡張を必要としたものの) ある種の不動点プログラミングが可能であることが分かる。

さて次に、上記のような無限の繰り返しの中に別の計算を入れることを考える。例として hello と表示しつづけるプログラムを考えるが、実は FIFOL には文字列を扱う機能がないので、¨ (ダブルクォーテーション) で括られた文字列は、そのままデータキューに格納されることにする。このとき、hello と表示しつづけるプログラムは、次のように書くことができる。

```
{ "hello" dup = exec } dup exec
```

このプログラムを実行したときのデータキューおよび命令キューの様子を図 4 に示す。なお、図 4 では、紙面の大きさと見やすさのために、ブロック { "hello" dup = exec } を H で表している。

このプログラムのブロック中で、dup と exec の間に = がある。これはもちろんこの時点でキューの先頭にある "hello" を出力するためのものである。ここで、exec の引数は dup されてできるブロック { "hello" dup = exec } であり、= の引数は "hello" という文字列である。図 4 の 4 行目でこれらが交互に出現しているように、キューマシンの命令列においては、ある演算を行おうとする演算の引数と演算子が必ずしも近くにない。

3.2 階乗の計算

もう少し意味のある計算として、階乗計算を行うプログラムを次に示す。

```
{ dup { rot pop pop }
      { rot rot dup rot rot dec mul
        dup rot rot rot exec }
      rot rot rot ifzeroelse }
dup 10 1 rot exec
```

このプログラムでは 10! を計算するが、プログラム中最下段にある 10 を別の非負整数値にすれば、その値の階乗を計算する。

ここで、 rot はキューの先頭のデータを取り出して改めてキューに格納する (末尾に追加する) 命令である。このような命令が必要なのは、キューの中のデータの順序を調整するためである。

スタックマシンであれば、ある計算 (関数) に必要な計算 (引数) は、その計算の近くで求められる。引数の順序を変更したい場合には、スタックの上部だけで入れ替えが可能である。しかし、キューマシンでは他の計算の途中経過がデータキュー上にあるために、キューの中身を回すだけの命令によって、データキュー全体をほぼ一周させる必要が生じることがある。

こうした理由もあって、上記のプログラムでは、以下の二つの機能拡張をしている。dec は整数一つを引数としてキューから受け取り、その整数から 1 を減じた値をキューに格納する命令である。また、ifzeroelse は整数と二つのブロックを引数としてキューから受け取り、整数が 0 であれば最初のブロックの中身を、そうでなければ二番目のブロックの中身を命令キューに追加する命令である。

スタックマシンであれば dec は 1 sub などと書かれ、また ifzeroelse も 0 eq および ifelse を用いて書かれるところだが、キューマシンにおいては、この 1 や sub、0、eq および ifelse を適宜配置したり、結果を使う計算に渡すためにキューを回す必要が生じる。

この階乗計算プログラムでは、exec が 3 行目と 5 行目に登場するが、いずれも最初のブロック (およびその複製) を実行する。このブロックを実行する時点で、データキューには、階乗計算の引数、計算の途中経過、ブロックの 3 つのデータが格納されていることを仮定している。すなわち、データキュー内のデータ数が (あまり) 変化しないということである。こうすることで、データキュー内のデータの位置調整をする場合にも rot の個数が定数個になる。

4 結論

キューマシンにおける不動点プログラミングについて考察した。いくつかのプログラム例を示して、キューマシンにおいても関数定義ないしブロックの名前付けを行うことなく、ある種の繰り返しを表現できることを示した。

しかし、本稿で示した繰り返しは、繰り返しの中でデータキューに格納されるデータ数が変らないものに限られている。これは、一般の再帰的な計算の表現ではなく、単純な繰り返ししか表現できていないことを意味する。残念ながら、現在のところ、そのような計算しか表現することができていない。この点について解決するような仕組みを考えることが今後の課題となる。

参 考 文 献

[1] 前田敦司: 新しい計算モデル キューマシンとその並列関数型言語への応用, 情報処理学会論文誌, Vol.38, No.3, pp.574–583, 1997

[2] 田中哲朗: プログラミングコンテスト実施報告, 夏のプログラミング・シンポジウム「プログラミングの鉄人—プログラミングの技」, pp. 3–14, 2001

[3] Adobe Systems 著, 桑沢清志 訳: Postscript リファレンスマニュアル第三版, アスキー, ISBN: 978-4756138224, 2001

[4] 大日向大地: SIMD 拡張 FIFOL を使った科学技術計算, 情報処理学会, 夏のプログラミング・シンポジウム報告集 (2005 年), pp. 97–102, 2006

[5] 山口文彦: ML で不動点演算子を定義する, 情報処理学会, 夏のプログラミング・シンポジウム報告集 (2005 年), pp. 103–108, 2006

[6] Eiichi Wada: fifol simulator, http://www.iij.jp/~ew/sim.html, (last accessed: Sep. 4, 2017)