プログラミング－言語・基礎・実践－ 15－8
（1994. 1. 21）

古典線形論理のプログラミング言語への応用

小方一郎
電子技術総合研究所

線形論理のプログラミング言語への応用について考察する。変数の線形性を利用し、証明の変形を実行と見る仕組みについて述べる。線形論理は、その証明の正規形（cut のない証明）が一意に定まる構成主義論理のため、cut 除去の過程を計算の実行と見ることが可能である。また、仮定の自由なコピーを許さないという線形論理の性質は、プログラミング言語での変数の出現の線形性との対応関係がある。変数に線形性があれば、変数を参照することはそこへ結合された構造体の破壊を意味するから、その参照の時点で構造体を再利用可能な資源として回収する。つまり、どこからも参照のない構造体をシステム全体から探してきて回収するプロセス（つまりゴミ集め）の必要がないシステムとなる。また、参照が一回であるという性質は、データフローによる実行の可能性も示唆する。

Application of Linear Logic to Programming Language

Ichiro OGATA
Electrotechnical Laboratory
1-1-4 Umezono, Tsukuba, 305 Japan

This paper gives an overview of application of Linear Logic to programming language. This language uses the proof normalization (cut elimination) step as computation. It also make use of the linearity of variables, which means every bound name is referenced exactly once. Because of this fact, we can collect every cell at the point of reference. It has to have no garbage collecter. It dose not have a dangling reference problem also. This also leads to dataflow execution of symbolic language.

1 はじめに

古典線形論理 Classical Linear Logic は、複数の帰結 multiple conclusion を持つ構成論理 constructive logic として注目を集めている。古典線形論理は構成論理であるため、証明を正規化 normalize する過程を計算のステップに対応させる、という手法が使える。

また、 Linear Logic の線形性も、プログラミング言語の大事な性質のひとつである。プログラム開発の過程で最も発見・修正が困難なバグのひとつは、記憶領域の重複使用によるものである。 Lisp では自動記憶管理（ゴミ集め）を導入してこの問題を解決しているわけだが、ここでは一歩進んで変数の参照の線形性を利用してゴミ集め garbage collection の必要のないという性質についても考察する。

2 線形なラムダ式での表現

ゴミ集めが不必要というなのは線形性から来ている。線形なラムダ式では、 body 部での変数の出現 occurrence が1回（ i.e. 0回でも、2回以上でもなく）に制限したものである。

つまり、変数の値は、決して捨てられたり、 share されたりすることがないのである。このため、 Linear Lisp で扱えるデータ構造は、“share のない完全二進木”のみとなる。

[1] [2] の Linear Lisp の記法で append を定義すれば、

```
(defun append (x y)
 (if-atom x (progn x y)
  (dlet* (((carx . cdrx ) x))
   `(,carx ,@(append cdrx y)))))
```

となる。 dlet は destructive let の意味で、変数 x に bind された cons cell を破壊して、car cdr それぞれの part を、変数 carx cdrx に代入する操作である。

Linear Lisp で cons cell を参照するとは、それを破壊することを意味する。そして、その cons cell は free list に即座に回収できる。 share がないから、この cons cell への参照が存在しないことが明らかであるし、名前の出現が線形であるため、そのプログラムの他の変数から参照されている可能性もないから、これが可能となる。

cons cell はこのように明示的に破壊されるため、ゴミ集めを独立した別プロセスとして実行する必要がないのである。

2引数の reverce2 も append と同じようにかける。

```
(defun reverse2 (x y)
 (if-atom x (progn x y)
  (dlet* (((carx . cdrx ) x))
   (reverse2 `(carx.y) cdrx))))
```

recursive な定義もしてみる。実は、このような定義でも効率の良い実行を許すようなルールの追加ができるかどうかが、この言語のポイントである。

```
(defun reverse (x)
 (if-atom x x
  (dlet* (((carx . cdrx ) x))
   (append (reverse cdrx) '(,carx)))))
```

3 グラフの変形としての表現

さて、上記の線形のプログラムを、グラフの変形で表現してみる。

ルール：

$$\frac{\overline{()}^{\,nil} \quad Y}{X}\,app \;\Rightarrow\; X\cdots Y$$

$$\frac{\dfrac{V \quad W}{(V.W)}\,cons \quad Y}{X}\,app \;\Rightarrow\; \frac{V \quad \dfrac{W \quad Y}{Z}\,app}{X}\,cons$$

実際に変形してみる。

$$\frac{(1) \quad (23)}{X}\,append \quad \frac{1 \quad \dfrac{nil \quad (23)}{Y}\,cons}{X}\,append \quad \frac{1 \quad \dfrac{2 \quad (3)}{(23)}\,cons}{(123)}\,cons$$

reverse2 も全く同じようにできる。

ルール：

$$\frac{\dfrac{Y \quad Z}{(Y.Z)}\,cons \quad W}{X}\,rev2 \;\Rightarrow\; \frac{Z \quad \dfrac{Y \quad W}{(Y.W)}\,cons}{X}\,rev2$$

$$\frac{\overline{()}^{\,nil} \quad Y}{X}\,rev2 \;\Rightarrow\; X\cdots Y$$

reverse では、 reverse の展開だけでなく、 append - append という連鎖の解消も並行に試みる。

ルール：

$$\frac{\overline{()}^{\,nil} \quad Y}{X}\,reverse \;\Rightarrow\; X\cdots Y$$

$$\frac{(Z.W)}{X}\,rev \;\Rightarrow\; \frac{\frac{W}{Y}\,rev \quad (Z)}{X}\,app$$

$$\frac{\frac{Z \quad (W)}{Y}\,app \quad V}{X}\,app \;\Rightarrow\; \frac{Z \quad (W.V)}{X}\,app$$

変形してみる。

$$\frac{(123)}{X}\,rev \quad \frac{\frac{(23)}{Y}\,rev \quad (1)}{X}\,app \qquad \frac{\frac{\frac{(3)}{Z}\,rev \quad (2)}{Y}\,app \quad (1)}{X}\,app$$

$$\frac{\frac{(3)}{Z}\,rev \quad (21)}{X}\,app \qquad \frac{\frac{\frac{()}{W}\,rev \quad (3)}{Z}\,app \quad (21)}{X}\,app \quad \frac{\overline{()}\,nil \quad (321)}{X}\,app$$

4 項書換え系での表現

上記の議論は、項書換えシステムを使って次のようにも書ける。

項書換え系の言葉で言えば、左辺も右辺も linear term （i.e. 変数の出現が 1 回）であることが条件で、しかも、かつ左辺と右辺の変数が同じであることである。項書換え系では、left linear （左辺が linear term であること）を条件にしている。

だから、ここまで述べてきた「線形」という条件は、項書換え系の言葉では、「left linear かつ right linear 」で、しかも変数の出現が left と right で同じもののことである。

```
(a0) append(nil,y) -> y
(a1) append(cons(carx,cdrx),y) -> cons(carx,append(cdrx,y))

(a3) append(append(x,cons(a,nil)),y) -> append(x,cons(a,y))

(r0) reverse(nil) -> nil
(r1) reverse(cons(carx,cdrx)) -> append(reverse(cdrx),cons(carx,nil))
```

（冗長なルールである）(a3) を除けば critical pair は発生せず、また当然 left linear でもあるので、この TRS は orthogonal である。（orthogonal な TRS は confluent である）

```
append(append(cons(carx,cdrx),cons(a,nil)),z)
```

という term を考えて、 critical pair を構成してみると、

```
a1: append(cons(carx,append(cdrx,cons(a,nil))),z)
a1: cons(carx,append(append(cdrx,cons(a,nil)),z))
a3: cons(carx,append(cdrx,cons(a,z)))

a3: append(cons(carx,cdrx),cons(a,z)))
a1: cons(carx,append(cdrx,cons(a,z)))
```

となり、結局、ciritical pair は発生しない。そこで、(a3) を加えても confluent 性は保たれている。これは、当然で、もともと、 (a3) は (r1) を 2 回展開した形に合わせて、 append の仕様と適合するように作ったからである。

```
reverse(cons(a,cons(b,c)))
 -> append(reverse(cons(b,c),cons(a,nil)))
 -> append(append(reverse(c),cons(b,nil)),cons(a,nil))
```

5 principal port

Lafont の Interaction Nets [Lafont90] では、 principal port という「注目点」を考えていた。 principal port という性質があるために、 strongly confluent

$$P \to Q \& P \to R \Rightarrow Q \to S \& R \to S$$

という性質があった。

一方、ここまでのグラフの変形のルールの性質を考えてみると、これは完全にローカルであり、 confluent である。しかし、 (a3) のルールを導入して principal port の考えを放棄したことにより、すでに strong confluent ではなくなっている。

また、 Lafont のシステムでは、その Type 付けにより Linear Logic の Multiplicative Proof Net との対応がつけられ、 Linear Logic が cut elimination を accept することと、simple net は simple rule の適用に関して closed であることの証明ができた。これも principal port に付随している。

全く同じ理由で、(a3) は type 付けに失敗し、当然 proof net との対応もとれない。それでも、(項書換え系の言葉で言えば) critical pair を生まない (a3) というルールは、これを付け加えてもなお confluent である。実行効率の面からは、このルールが付け加えられるようなシステムであって欲しい。

しかし、これはやはりできない。

(a3) というルールは、 append - append - cons という 3 者の alive set となっているからである。 binary intersection でなくなっているわけである。まさにこれが strongly confluency を失わせている原因であり、 confluent であることの証明が別に必要になってしまっている。

結局、この時点で interaction net の、この方向への拡張は失敗している。

6 結論

まず、この言語は、計算が局地的なものに限定されているので、データフロー型の計算に向いている。データの発火条件も principal port の結合、つまり alive pair であるかどうかを2つのデータ間で確かめるだけである。また、データの共有もおこらないので、上記の cons や append や reverse などは、トークンとしてそのまま扱える。

さて、ここまで述べたことは、固まったプログラミング言語からはほど遠い、原理的なことだけである。この原理をもとに、実際のプログラミング言語を設計するためには、まだ考慮すべき点がたくさんある。

もっとも重要なのは、おそらく効率の問題であろう。Lisp のほとんどの応用では、ほとんどの cons cell の参照は一回である（つまり、線形性がある）とは言え、share を有効に利用して効率的な実行をするプログラムもあるからである。特に database の世界では、巨大な data の share は本質的であろうから、この言語の適用には、実際にはかなり検討が必要である。

参考文献

[1] H.G.Baker, "Lively Linear Lisp - 'Look Ma, No Garbage!"',*ACM Sigplan Notices* 27,8 (Aug.1992), 91-95

[2] H.G.Baker, "Linear Logic and Permutaion Stacks - The Forth Shall Be First",to appear *ACM Sigarch Computer Architecture News.*

[3] G.Berry and G.Boudol, "The chemical abstract machine",*proc. of ACM POPL 90*,95-108, Jan. 1990

[4] J.-Y. Girard, "Linear Logic". *Theoretical Computer Science*, 50, 1-102, 1987.

[5] W.A. Howard, "The Formulae-as-types notion of construction" in [6] 479-490

[6] J.R. Hindley and J.P. Seldin, *To H.B. Curry: Essays on combinatory logic, Lambda Calculus and Formalism*, Acadmic Press, 1980.

[7] Y. Lafont, "The Linear Abstract Machine" *Theoretical Computer Science*, 59, 157-180, 1988.

[8] Y. Lafont, "Interaction Nets" *proc. of ACM POPL 90*,95-108, Jan. 1990

[9] A. Scedrov,"A Brief Guide to Linear Logic", *Bull fo ETACS*, 41, 154-166, 1990.

[10] S. Abramsky. "Computational interpretations of linear logic", *Theoretical Computer Science*, 111, 3-57,1993.