並列論理型言語 FLENG のデバッガ

舘村 純一　田中 英彦

概要

並列論理型言語のプログラミング環境としてデバッガが必要とされるが、並列プログラムは実行の様子を順にトレースするのは困難である。その解決策として、論理型言語であるということから、宣言的なモデルでデバッグをすることが考えられる。しかし、GHC のような並列論理型言語は純粋な論理プログラムではないので、宣言的意味の中にも手続き的な要素が入ってくる。並列論理型言語のセマンティクスの諸研究で入出力のタイミング、因果関係なども考慮しなければならないことが論じられている。そこで本論文ではプログラムの実行を通信をするプロセスの形で表し、入出力が行なわれている様子を表現した。この様なモデルでアルゴリズミックデバッギングを行なうことができる。ここでは並列論理型言語として FLENG を対象とする。

1　はじめに

近年、PARLOG[Clark86]、Concurrent Prolog[Shapiro83a]、GHC[Ueda85a] など、Horn 論理に基づく並列プログラミング言語が提案されている。現在これらの並列論理型言語の処理系、プログラミング環境が構築されつつある。このプログラミング環境として重要なものにデバッガがある。しかし、並列プログラムのデバッグは困難なものとされている。その大きな原因として、実行が半順序的なため逐次プログラムのように単純にトレースするのが困難であることがあげられる。

そもそもデバッガの役割とはなにかと考えてみると、それは、プログラムの実行からその様子を抽出してプログラマに示すことであるといえよう。デバッガがプログラムの実行をモデル化し、プログラマはそれと自分の意図するモデルとを照らし合わせて、その違いからバグを発見する。それゆえ並列論理型言語に適したデバッガを構築するためには、それに適した実行モデルを構築することが重要である。

並列プログラムを従来のようなトレースによってデバッグするのには限界があるが、一方従来の宣言的モデルでは並列論理型言語に適さない。そのため、宣言的意味に手続き的要素を加えたものが実行モデルとして望まれる。そこで本論文では並列論理型言語のセマンティクスに関する諸研究の成果をとり入れた上で、並列論理型言語の実行の様子をプロセスとその入出力によって表現し、この実行モデルをプログラムのデバッグに用いる。

2　Committed-Choice 型言語 FLENG

Committed-Choice 型言語は、論理型プログラミングで通信、同期が記述できるように制御機能を強化したもので、汎用並列プログラム言語を目指したものである。GHC や Concurrent Prolog は、この Committed-Choice 型言語である。

我々の研究室で研究されている FLENG[Nilsson86] は GHC をより単純化して計算機上への実装を容

易にしたCommitted-Choice型言語である。GHCでは、ゴールの成功、失敗が他のゴールに影響するために並列計算機上での処理が複雑になる。FLENGでは、ゴールの実行は論理的な真理値とは無関係に行なわれる。そのため、ゴールの成功、失敗が他のゴールに影響を与えない。ゴール間の論理的な関係が必要な時は、共有変数によって記述する。FLENGでは、ヘッドのマッチングが完了した時点で定義節がコミットされるので、定義節はガードを持たない。

本論文ではこのFLENGを扱う。Commtted-Choice型言語は、純粋な論理型言語ではないが、なかでもFLENGは、以上のようなことから、さらに純粋な論理としての意味合いが薄くなっている。本論文では、このFLENGのプログラムの実行をプロセスが通信しているものとしてとらえている。

3 並列論理型言語のデバッギング

論理型プログラムは、手続き的にも宣言的にも解釈できるという特徴を持つ。プログラムの実行を追っていくトレーサは、手続き的なモデルをプログラマに示すものといえる。一方、論理プログラムは宣言的側面もあるのだから、宣言的なモデルを用いたデバッギングも考えられる。Shapiroは、Prologにおいてアルゴリズミックデバッギングを提案した[Shapiro83b]。また、Prologにおいては、宣言的セマンティクスの問題に関して様々な研究がなされている。しかしこれは純Horn論理に適用できるもので、そのままでは並列論理型言語には適用できない。それは、並列論理型言語において、同期、通信の概念を導入するために、Horn節に新たに意味を付加したためである。

逐次論理型言語Prologにおいてなされた研究の成果は、並列論理型言語においても適用され、並列言語に適するように拡張されている。例えばPrologのトレーサと同様のものが並列論理型言語でも開発されている。これらは逐次型処理系において実装されている[Ueda85b]。また、Prologにおけるボックスモデルに対応するものとして新しいモデルも考えられ、トレーサに応用されている[Ezaki86]。しかしこれらは逐次的に実行を追っていくもので、並列言語に適用するには限界がある。

さらに、Prologと同様なアルゴリズミックデバッガも、GHCにおいて研究されている[Takeuchi86]。これはPrologにおけるアルゴリズミックデバッガと同様のものをモデルとして用いているが、この様なモデルでは充分でないことが述べられている。

これらのデバッガは、Prologのデバッガを逐次処理系において拡張したものである。Prologも、並列論理型言語も、論理型言語という点では共通である。しかし、並列論理型言語の特徴は論理型言語という点だけではない。Prologと違う点は、当然のことながらその並列性である。しかしながら、並列言語に適したといえるデバッガはまだこれからの課題といえる。

このような中で、並列論理型言語におけるセマンティクスの問題が論じられているわけであるが、そこには純粋な論理型言語にはないような問題が出てくる。

並列論理型言語において、入力や出力を論じることがあるが、この入出力とは次のようなものをいう。例えば、append/3はFLENGで、以下のように定義される。

```
append([A|X], Y, Z) :- Z = [A|Z1], append(X, Y, Z1).
append([], Y, Z) :- Z = Y.
```

ここで、append([1],[2],X)というゴールがあれば、その結果はappend([1],[2],[1,2])となる。append(X,Y,Z)において、X、Yは入力、Zは出力であるといえる。上の例では、入力として{X = [1], Y = [2]}があり、出力として{Z = [1,2]}があったといえる。

しかし、入出力の結果をこのように示しただけでは充分でない場合も存在することが知られている。例えば次のp1/2とp2/2を比較してみる。

```
p1([A|In],O) :- O = [A|Out], p11(In, Out).
p11([A|In], O) :- O = [A].

p2([A,B|In], O) :- O = [A,B].
```

それぞれ入出力を考えるとどちらも

```
p1(X,Y) (p2(X,Y))

Input  = {X = [A,B|_]}
Output = {Y = [A,B]}
```

となり、区別できない。しかし、p1 と p2 は、出力を具体化するタイミングが違う。この二つのプログラムが非決定的なプログラムと並列に実行された場合にそれぞれの結果がことなる可能性があるということが指摘されている [Brock81]。そこで、GHC のような並列論理型言語においては、入力と出力のタイミング、因果関係も問題にされ、GHC のセマンティクスに関する諸研究 [Takeuchi87][Murakami88b] で論じられている。

本論文では、このような問題もふまえた上で、プログラムの実行においてコミットされる定義節から入出力を定義し、これをもってプログラムの実行の様子を表すことを考える。

4 プロセス型実行モデル

4.1 プロセスの概念

従来から、並列論理型言語の動作を説明するのにプロセスという考え方が用いられてきた。しかし、GHC などで考えられている従来の「プロセス」は一つのゴールを指すものである。ゴールがリダクションされることによって、プロセスは幾つかの新しいプロセスに変換される。このようなモデルでは、多数のプロセスが生滅していく様子を追っていかなければならない。このモデルを用いてデバッグをすることを考えると、ダイナミックに、しかも並列に移り変わるプロセスの通信の様子を把握しなければならず、大変困難なものとなるであろう。

本論文で述べるプロセスは、ある一つのゴールと、そこからリダクションを繰り返すことにより生成される全てのサブゴールを合わせたものを一つのプロセスと見るものである。これを計算木と対応付けて考えてみる。計算木はプログラムの実行された様子を木の形で表したものである。その内部には、あるノードをルートとするようなサブトリーが存在するわけであるが、このサブトリーが一つのプロセスにあたる。トリーがサブトリーに分割されるように、プロセスはいくつかのサブプロセスに分割される。このプロセスとサブプロセスの関係を規定するものが定義節であると考えることができるであろう。

このようなモデルを用いることにより、プログラムの実行をトップダウンに解析することができる。プログラムの実行はプロセス間の通信と見ることができ、プロセスの内部はまたサブプロセスで表現できる。

4.2 実行モデル

ここで述べるプロセスは、以下のように表現できる。

$$\langle G_{skel},\ I,\ O,\ S,\ G_{ins} \rangle$$

G_{skel} は *skeletal predicate* であり、引数がすべて個別の変数である述語である。これは計算木のサブトリーのルートのゴールの引数をすべて変数にしたものをさす。すなわち、

$$p(v_1, \ldots, v_i)$$

$v_1, \ldots, v_i$ はそれぞれ個別の変数で、この変数が外部との通信の窓口といえる。

I,O は *Input/Output* であり、プロセスの入出力を表す。G_{skel} の変数がプロセス (計算木のサブトリー) の外部、及び内部からどのようにユニファイされていくかを示したものである。

S は *Status* で、現在のプロセスの状態を表す。これには *terminate*, *suspend*, *active*(*stop*) の状態がある。*terminate* は、プロセスのサブプロセスがすべて終了したことを示す。*suspend* は、プロセスの内部にアクティブなゴールがなく、サスペンドしたゴールのみとなった状態をいう。これは、外部からの入力によってそれらのゴールがアクティベートされる可能性がある。この場合、プロセスの状態は *active* (もしくは *stop*) に変化する。しかし、トップレベルのプロセスなどで、入力が新たに入らないような場合はその可能性はない。この時は状態を *deadlock* と表すことができる。*active* とは、プロセスがまだ終了も、デッドロックもせず、アクティブなゴールが残っている状態のことである。プログラムを走らせながら観察する場合にはこのように *active* と表現できるが、プログラム全体が停止させられた状態で観察する場合、これは *stop* と呼ぶことができるであろう。このようにプロセスの状態を定義すれば、実行途中のプログラムや、無限に続くプログラムも扱うことができる。

G_{ins} はゴールの *instance* である。G_{skel} に対して入出力を行なって変数が束縛されてできた結果を表したものと考えることもできる。

4.3 プロセスの入出力

プロセスの入出力は、トップレベルのイニシャルゴールとコミットされた定義節より定義される。

まず、トップレベルで与えられたゴール (イニシャルゴール) について、その入力が定義できる。すなわち、

$$P(t_1(X_{11}, \cdots, X_{1n_1}), \cdots, t_k(X_{k1}, \cdots, X_{kn_k}), V_{k+1}, \cdots, V_m)$$

において、

$$G_{skel} = P(V_1, \cdots, V_m), \quad I = \{I_1, \cdots, I_m\}$$

$$I_i = \begin{cases} [V_i = t_i(X_{i1}, \cdots, X_{in_i})] & i \leq k \\ nil & k < i \leq m \end{cases}$$

となる。ただし、$t_i(X_{i1}, \cdots, X_{in_i})$ は変数でない項で、X_{ij} は項に含まれる変数である。

次に、コミットされた定義節から、ゴールとサブゴールの入出力の関係が決まる。そこでまず次のような定義節を考える。

$$H \ :- \ B_1, \cdots, B_i.$$

$$\begin{cases} H(t_1(X_{11}, \cdots, X_{1n_1}), \cdots, t_k(X_{k1}, \cdots, X_{kn_k}), V_1, \cdots, V_m) \\ B_i(t'_{i1}(X_{11}, \cdots, X_{1n_1}, \cdots, X_{kn_k}, V_1, \cdots, V_m, Y_1, \cdots, Y_l), \cdots, t'_{ij}(\cdots), \cdots) \end{cases}$$

つぎに、この定義節を以下のようにガードつきのホーン節に展開することにより、述語の引数をみな変数にしてしまう。

$$\begin{cases} H(t_1,\cdots t_k,V_1,\cdots,V_m) & \Rightarrow H(W_1,\cdots,W_k,V_1,\cdots V_m),\ [W_1=t_1,\cdots,W_k=t_k] \\ B_i(t'_{i1},\cdots,t'_{ij_i},X,\cdots) & \Rightarrow B_i(U_{i1},\cdots,U_{ij_i},X,\cdots),\ U_{i1}=t'_{i1},\cdots,U_{ij_i}=t'_{ij_i} \end{cases}$$

このようにして、*skeletal predicate* である H 、B_i と、ガード部に展開された $W_i = t_i$ 、ボディー部に展開された $U_{ij} = t'_{ij}$ 、また、もとから定義節に書かれたシステム述語で定義節が構成されることになる。ここでは簡単のため、システム述語として、

$$V_x = t''_{V_{xi}}(\vec{Z})$$

を考える。ただし、$\vec{Z} = Z_1,\cdots,Z_n$ で、V_x 、Z_i は X_{ij} 、V_i などの変数である。

ここで、ゴールの出力

$$O_H = \{O_{HW_1},\cdots,O_{HW_k},\ O_{HV_1},\cdots,O_{HV_m}\}$$

と、サブゴールの入力

$$I_{B_i} = \{I_{B_iU_{ij}},\ I_{B_iX_{jk}},\ I_{B_iV_j},\ I_{B_iY_j}\}$$

を決めればよい。

まず、ゴール H の出力は以下のように、H の入力と B_i の出力から決められる。ただし、

$$\prod_i A_i = [A_1,\cdots,A_n]$$

と約束する。

$$O_{HW_i} = \{Guard,\ \{W_i = t_i(X_{i1},\cdots,X_{in_i}),\ \prod_j O_{X_{ij}}\}\}$$

$$O_{HV_i} = \{Guard,\ [V_i = t''_{V_i1}(\vec{Z}),\ \prod_k O_{Z_k}],\cdots[V_i = t''_{V_in}(\vec{Z}),\ \prod O_Z],\ \prod_j O_{B_jV_i}\}$$

ここで、

$$Guard = \prod_j \{W_j = t_j,\ I_{HW_j}\}$$

$$O_{X_{ij}} = \prod_k O_{B_kX_{ij}}$$

$$O_{Z_k} = \{\prod_j O_{B_jZ_k},\ I_{HZ_k}\}$$

である。

つぎに、サブゴール(ボディーゴール)B_i の入力は、以下のようになる。

$$\begin{cases} I_{B_iU_{ij}} = \{Guard,\ [U_{ij} = t'_{ij}(\vec{Z}),\ \prod_k O_{Z_k}]\} \\ I_{B_iX_{jk}} = \{Guard,\ [[t_j(\cdots X_{jk}\cdots) = W_j,\ I_{HW_j}],\ \prod_m [X_{jk} = t''_{X_{jk}m}(\vec{Z}),\ \prod_l O_{Z_l}]\ \prod_{l\neq i} O_{B_lX_{jk}}]\} \\ I_{B_iV_j} = \{Guard,\ [I_{HV_j},\ \prod_m [V_j = t''_{V_jm}(\vec{Z}),\ \prod_l O_{Z_l}]\ \prod_{k\neq i} O_{B_kV_j}]\} \\ I_{B_iY_j} = \{Guard,\ [\prod_m [Y_j = t''_{Y_jm}(\vec{Z}),\ \prod_l O_{Z_l}]\ \prod_{k\neq i} O_{B_kY_j}]\} \end{cases}$$

この入出力について、簡単な例をみてみる。

```
h(A) :- g1(A,B), g2(A,B).        (1)
h(A,B) :- A = t(B), g(A,B).      (2)
h(t(A)) :- g(A).                 (3)
```

(1) では、

$$\begin{cases} \langle h(A),\ [I_{hA}],\ [O_{hA}]\rangle \\ \langle g_1(A,\ B),\ [I_{g_1A},\ I_{g_1B}],\ [O_{g_1A},\ O_{g_1B}]\rangle \\ \langle g_2(A,\ B),\ [I_{g_2A},\ I_{g_2B}],\ [O_{g_2A},\ O_{g_2B}]\rangle \end{cases}$$

において、

$$\begin{cases} O_{hA} = [O_{g_1A},\ O_{g_2A}] \\ I_{g_iA} = [I_{hA},\ O_{g_jA}] \\ I_{g_iB} = [O_{g_jB}] \quad i = 1,2,\ j \neq i \end{cases}$$

となる。同様に、(2) の場合は、

$$\begin{cases} O_{hA} = [[A = t(B),\ \{[O_{gB}],\ I_{hB}\}],\ O_{gA}] \\ I_{gA} = [I_{hA},\ [A = t(B),\ \langle[O_{gB}],\ I_{hB}\rangle]] \\ I_{gB} = [I_{hB}] \end{cases}$$

(3) の場合は、

$$\begin{cases} \langle h(X),\ [I_{hX}],\ [O_{hX}]\rangle \\ \langle g(A),\ [I_{gA}],\ [O_{gA}]\rangle \end{cases}$$

において、

$$\begin{cases} O_{hX} = \{[\{X = t(A), I_{hX}\}],\ \{X = t(A), [O_{gA}]\}\} \\ I_{gA} = \{[\{X = t(A), I_{hX}\}],\ [t(A) = X,\ I_{hX}]\} \end{cases}$$

となる。

5 モデルの実例

5.1 プロセス間通信の例

例として次のようなプログラムを考えてみる。

```
qq(L, R) :- q(L, M), q(M, R).

q([b(X)|L], R):- R = [b([a|X])|R1], q(L, R1).
q([end|L], R) :- R =[end].
```

プロセス qq は二つのプロセス q に分割される。これらは変数を共有していて、ストリームによる通信を行なう。q(L,R) は L にストリームで入力を受け取り、R にそれに応じて出力し、となりのプロセスに通信を送る。ここではプロセス q は二つだけだが、いくつつなげてもよい。

ここで、qq([b([]),end],X) というゴールが与えられたとする。すると結果は X = [b([a,a]),end] となる。このプロセスと、二つのサブプロセスについて、入出力を見てみる。

今回作成したデバッガでは、まずメタインタプリタによるプログラムの実行を行なう。4.3節で定義した入出力の様子はそこからリスト構造として得られる。デバッガではこれを更に見やすいように処理して表示する。その表示方法は次のようである。

$$\prod_i A_i = [A1, A2, A3]$$

は、

```
|-A1
|-A2
|-A3
```

となり、サブゴールがフォークしたことにより、入出力の流れ関係が分割されたことを表す。ただし、サブゴールが変数を共有した場合に限る。

$$[V = term(\vec{Z}), \prod O_Z]$$

は、

```
V = term
 |-Z1
 |-...
```

となる。これは、V に term を出力しようとしていることを表す。term に含まれる変数に関する出力が後に続く。

$$\{Guard,\ X\},\ Guard = \{V = t,\ I\}$$

は、

```
guard{V = t}
  | <-I
  |-X
```

となる。これが入出力間の因果関係を表している。

```
qq(_1,_2) --- q(_1,_3)
          |
          |- q(_3,_2)
```

```
goal : qq(_1,_2) / qq([b(nil),end], [b([a,a]),end])

OUTPUT OF _2 :
```

```
guard{_3 = [b(_4)|_5]}
  |          <-guard{_1 = [b(_6)|_7] / [b(nil),end]}
  |              |-_3 = [b([a|_6])|_8]
  |                |-guard{_7 = [end|_9]}
  |                    |-_8 = [end]
  |-_2 = [b([a|_4])|_10]
    |-guard{_5 = [end|_11]}
        |-_10 = [end]

goal : q(_1,_3) / q([b(nil),end],[b([a]),end])

OUTPUT OF _3 :
guard{_1 = [b(_6)|_7] / [b(nil),end]}
  |-_3 = [b([a|_6])|_8]
    |-guard{_7 = [end|_9]}
        |-_8 = [end]

goal : q(_3,_2) / q([b([a]),end],[b([a,a]),end])

OUTPUT OF _2 :
guard{_3 = [b(_4)|_5] / [b([a]),end]}
  |-_2 = [b([a|_4])|_10]
    |-guard{_5 = [end|_11]}
        |-_10 = [end]
```

q(_1,_3) を見てみる。_1 に入力 _3 に出力が存在する。_1 が [b(_6)|_7] であることを条件に (ここで同期がとられる) _3 = [b([a|_6])|_8] が出力され、さらに _7 が [end|_9] であることを条件に _8 = [end] が出力されたことがわかる。q(_3,_2) も同様な動作をしたことがわかる。qq(_1,_2) の _2 の出力を見てみると、これらを合わせたようになっている。

5.2 非決定的なプログラム

```
p(X,Y) :-  p1(X), p2(X),p3(X,Y).

p1(X) :- X = a.
p2(X) :- X = b.
p3(a,Y) :- Y = 1.
p3(b,Y) :- Y = 2.
```

上のプログラムは X = a と X = b のどちらが先に実行されるかによって結果が違ってくる。先に実行された方がユニファイに成功し、後の方が失敗する。この結果は次の二通りがある。このように、一つの変数を、複数のゴールが具体化しようとしている様子がわかる。FLENG では一つのゴールの失敗 (真理値) が他のゴールに影響しないが、GHC の場合は全体が失敗することになる。

```
goal : p(_1,_2) / p(a,1)

OUTPUT OF _2 :
guard{_1 = a}
  |         <-_1 = a
  |         <-_1 = b
  |-_2 = 1

goal : p(_1,_2) / p(b,2)

OUTPUT OF _2 :
guard{_1 = b}
  |         <-_1 = a
  |         <-_1 = b
  |-_2 = 2
```

5.3 Brock Ackermann の例

```
p1([A|In],O) :- O = [A|Out], p11(In, Out).
p11([A|In], O) :- O = [A].

p2([A,B|In], O) :- O = [A,B].

dup([A|I], O) :- O = [A,A].

merge([A | Ix], Iy, O) :- O = [A|Out], merge(Ix, Iy, Out).
merge(Ix, [A | Iy], O) :- O = [A|Out], merge(Ix, Iy, Out).
merge(Ix, [], O) :- O = Ix.
merge([], Iy, O) :- O = Iy.

s1(Ix, Iy, Out) :-
        dup(Ix, Ox), dup(Iy, Oy),
        merge(Ox, Oy, Oz), p1(Oz, Out).
s2(Ix, Iy, Out) :-
        dup(Ix, Ox), dup(Iy, Oy),
        merge(Ox, Oy, Oz), p2(Oz, Out).

t1(In, Out) :- s1(In, Mid, Out), plus1(Out, Mid).
t2(In, Out) :- s2(In, Mid, Out), plus1(Out, Mid).

plus1([A|In], O) :- add(A, 1, A1), O = [A1].
```

上のプログラムが前述したBrock Ackermannのプログラムである。ここでは、p1(X,Y)とp2(X,Y)はX = [A,B|_]という入力に対してY = [A,B]を出力するという点では同じで区別されない。しかしこれを上のように非決定的なプログラムmerge/3とともに用いると、それぞれ違った動作を示す可能性がある。t1(In,Out)とt2(In,Out)にそれぞれIn = [5]という入力を与えると、t2の場合はOut =[5,5]という可能性しかないが、t1の場合はOut = [5,5]とOut = [5,6]の二つの可能性がある。このため、入力と出力の同期の関係(因果関係)もプログラムの意味として記述する必要があるということが[Takeuchi87]や[Murakami88b]等で論じられている。

まず、t2(X,Y)の方を見てみる。計算木を見てみると、プロセスは次のようにわかれることがわかる。

```
t2([5],[5,5])
  |-s2([5],[6],[5,5])
  | |-dup([5],[5,5])
  | |-dup([6],[6,6])
  | |-merge([5,5],[6,6],[5,5,6,6])
  | |-p2([5,5,6,6],[5,5])
  |-plus1([5,5],[6])
```

t2のサブプロセスとしてs2とplus1があって互いに通信しあっているわけであるが、問題となるs2(X,Y,Z)についてその入出力を見てみる。XとYに入力があり、Zに出力がある。plus1との通信に用いられているのはYとZである。

```
goal : s2(_1,_2,_3) / s2([5],[6],[5,5])

OUTPUT OF _3 :
guard{_4 = [_5,_6|_7]}
  |          <-guard{_8 = [_9|_10]}
  |              |          <-guard{_1 = [_11|_12] / [5]}
  |              |              |-_8 = [_11,_11]
  |              |-_4 = [_9|_12]
  |                |-guard{_10 = [_13|_14]}
  |                    |-_12 = [_13|_15]
  |                      |-guard{_14 = nil}
  |                          |-_15 = _16
  |                            |-guard{_2 = [_17|_18] / [6]}
  |                                |-_16 = [_17,_17]
  |-_3 = [_5,_6]


INPUT OF _2 :
guard{_3 = [_19|_20] / [5,5]}
```

```
  |-_2 = [_21]
    |-add(_19,_22,_21) -> _21 = 6
        |-_22 = 1
```

このプログラムでは Var = term 以外のシステム述語として add/3 が出てくるが、これに関しては _2 の入力のように表せばよいと考えられる。

サブプロセス p2 の出力は以下のようになる。

```
goal : p2(_4,_3) / p2([5,5,6,6],[5,5])

OUTPUT OF _3 :
guard{_4 = [_5,_6|_7]}
  |-_3 = [_5,_6]
```

次に、t1 を見てみる。まず t1([5],[5,5]) となった場合は次のようになる。

```
t1([5],[5,5])
  |-s1([5],[6],[5,5])
  | |-dup([5],[5,5])
  | |-dup([6],[6,6])
  | |-merge([5,5],[6,6],[5,5,6,6])
  | |-p1([5,5,6,6],[5,5])
  |-plus1([5,5],[6])
```

このサブプロセス s1 は以下のような入出力を行なっている。

```
goal : s1(_1,_2,_3) / s1([5],[6],[5,5])

OUTPUT OF _3 :
guard{_4 = [_5|_6]}
  |         <-guard{_7 = [_8|_9]}
  |             |         <-guard{_1 = [_10|_11] / [5]}
  |             |             |-_7 = [_10,_10]
  |             |-_4 = [_8|_12]
  |               |-guard{_9 = [_13|_14]}
  |                   |-_12 = [_13|_15]
  |                     |-guard{_14 = nil}
  |                         |-_15 = _16
  |                           |-guard{_2 = [_17|_18] / [6]}
  |                               |-_16 = [_17,_17]
  |-_3 = [_5|_19]
    |-guard{_6 = [_20|_21]}
        |-_19 = [_20]
```

```
INPUT OF _2 :
guard{_3 = [_22|_23] / [5,5]}
  |-_2 = [_24]
    |-add(_22,_25,_24) ->_24 = 6
        |-_25 = 1

goal : p1(_4,_3) / p1([5,5,6,6],[5,5])

OUTPUT OF _3 :
guard{_4 = [_5|_6] /  [5,5,6,6]}
  |-_3 = [_5|_19]
    |-guard{_6 = [_20|_21]}
        |-_19 = [_20]
```

t2 との違いは、_4 = [_5, _6|_7] となってから初めて _3 に出力が出るのか、_4 = [_5|_6] となった時点で出力があるのかという点である。_3 = [_5|_9] という出力があれば、_2 に入力が入る可能性がある。_2 = [_17|_18] という入力があれば、_16 = [_17,_17] となって _4 のリストの二番目の要素にこの _17 が入る可能性が出る。するとこれが _3 のリストの二番目の要素になる。このような場合は t1([5],[5,6]) となる。実際にその例を見てみる。

```
goal : s1(_1,_2,_3) / s1([5],[6],[5,6])

OUTPUT OF _3 :
guard{_4 = [_5|_6]}
  |         <-guard{_7 = [_8|_9]}
  |             |         <-guard{_1 = [_10|_11] / [5]}
  |             |             |-_7 = [_10,_10]
  |             |-_4 = [_8|_12]
  |               |-guard{_13 = [_14|_15]}
  |                   |          <-guard{_2 = [_16|_17] / [6]}
  |                   |              |-_13 = [_16,_16]
  |                   |-_12 = [_14|_18]
  |                     |-guard{_15 = [_19|_20]}
  |                         |-_18 = [_19|_21]
  |                           |-guard{_20 = nil}
  |                               |-_21 = _9
  |-_3 = [_5|_22]
    |-guard{_6 = [_23|_24]}
        |-_22 = [_23]

goal : s1(_1,_2,_3) / s1([5],[6],[5,6])
```

```
INPUT OF _2 :
guard{_3 = [_25|_26] / [5,6]}
  |-_2 = [_27]
    |-add(_25,_28,_27) -> _27 = 6
        |-_28 = 1
```

このような t1 と t2 の違いは、結局 p1 と p2 の動作の違いから来る。

6 デバッグの実際

6.1 アルゴリズミックデバッギング

本論文で述べたモデルを、アルゴリズミックデバッギングに適用する。アルゴリズムは Shapiro の "Divide and Query" を基本とする。これは Prolog のためのものであり、プログラマには計算木のノードであるインスタンスを提示して、その正誤を問う。この質問を繰り返すことにより、バグを探索する。

しかし、ここでいう「正しい」とは、論理式である述語に対してプログラマが意図した意味の集合に、そのインスタンスが含まれているということである。ということは、ここで扱う計算木は、純粋な論理プログラムによるものでなければならない。前述したように、並列論理型言語は純粋な論理プログラム言語ではない。それゆえ、インスタンスを提示してアルゴリズミックにバグを発見するという方法は、並列論理型言語においてはうまくいかないことが起こる。

ここで構成されるデバッガではインスタンスではなくそれに対応するプロセスを提示する。これが正しい場合は、少なくともこの実行においては、このプロセスに相当するサブトリーの中にはバグは存在しないと考えられる。一方、提示されたプロセスの例がプログラマの意図するものと違う時、プログラマの答え方は従来の場合と異なってくる。それは、間違ったプロセスの例には次の場合があるからである。

1. 入力は正しいのにそれに対する出力が違う場合。
2. 意図されない入力が存在する場合。

1の場合、このプロセスに対応するサブトリーの中にバグがあると考えられる。この場合は、Shapiro の "Divide and Query" で no と答える場合と同様である。しかし、2の場合は、サブトリーの外にバグがあり、そこから間違った通信が届いていると考えられる。これは yes と答える場合に相当する。そこで、1の場合は出力が違うとして out と答え、2の場合は入力が違っているとして in と答えることにして区別しなければならない。

これより、このアルゴリズミックデバッギングのアルゴリズムは、以下のようになる。

1. あるプロセス P において、各サブプロセスについて問い合わせる。
2. 1でプログラマから受けた答について、
   (a) すべてのサブプロセスが正しければ、P に関する定義節を答えとして返す。
   (b) もし正しくないサブプロセスが存在したら、それを新たに P として、1を繰り返す。

ここで、「正しい」とは、

- プロセスの入出力が正しい。(yes と答える場合)

- プロセスにプログラマの意図しない入力が入っている。(in と答える場合)

のことをさす。P に関する定義節とは、プロセス P とサブプロセスの関係を定義する定義節、すなわち、ルートのゴールのリダクションに用いた定義節である。

6.2 入出力の解析

本論文で述べた入出力はプログラムの大きさにつれて構造が複雑になる。これをアルゴリズミックデバッギングにおいて表示しても理解が困難となり、返って能率が悪くなる。そのため、この入出力を部分的に計算して簡略化することが考えられる。これをより詳しく見たいという時にそれに応じて表示すればよい。これには色々なレベルが考えられる。

1. guard で区切られていない部分はまとめる。
2. 外部から見えない変数による guard はなくしてまとめる。
3. 全部まとめる。

1は、例えば次のようなものである。

```
guard{A = t(X)}                    guard{A = t(X)}
  |-B = [X|C]          -->           |-B = [X,X|D]
    |-C = [X|D]                        |-guard ...
      |-guard ...
```

2は、内部的な動作を取り除いて、外部に関係する所だけを示すもので、そのプロセスの外部から見た意味としては必要十分な情報といえる。例えば、5.1の qq(_1,_2) の _2 の出力は次のようになる。

```
guard{_1 = [b(_6)|_7] / [b(nil),end]}
  |-_2 = [b([a,a|_6])|_10]
    |-guard{_7 = [end|_9]}
        |-_10 = [end]
```

またこの他にも、ある一つの変数に注目してそれがどのように束縛されていったかを追っていったり、入出力を対応付けて、そのタイミングを解析したりすることが考えられる。実行によって得られた入出力のデータの中にリダクションされたゴールの名前も埋め込んでおけば、ある出力がどのサブゴールによるものかなども調べられ、また例えば5.3の例で次のような表し方もできるであろう。

```
guard{_4 = [_4|_6]}
  |          <-merge(_7,_16,_4) : _4 = [5,6,6,5]
  |              <-guard{_1 = [_10|_11] / [5]}
  |                  |-_7 = [_10,_10]
  |              <-guard{_2 = [_17|_18] / [6]}
  |                  |-_16 = [_17,_17]
  |-_3 = [_5|_19]
    |-guard{_6 = [_20|_21]}
        |-_19 = [_20]
```

6.3 デバッグ例

実際にデバッガを実装して、デバッグを試みてみた。このデバッガはUNIX(SUN)上の逐次処理系に実装されたが、並列推論マシン上への実装を考えてFLENG自身で記述されている。ここではアルゴリズミックデバッギングの例をいくつか見てみる。

6.3.1 入出力のモードのバグの例

まずクイックソートのプログラムを例にあげる。qsort(X,Y)のXに整数のリストを与えると、Yにソートされた答えが返ってくるものである。

```
qsort(X,Y) :- qsort(X,Y,[]).
qsort([X | L], R, R0) :-
    partition(L, X, L1, L2),
    qsort(L2, R1, R0), qsort(L1, R, [X | R1]).
qsort([], R1, R2) :-
    R1 = R2.
partition([X | L], Y, L1, L2) :-
    gt(X, Y, Z), p10(Z, [X | L], Y, L1, L2).
partition([], X, L1, L2) :-
    L1 = [], L2 = [].
p10(false, [X | L], Y,  L1, L2) :-
%     L1 = [X | L3], partition(L, Y, L3, L2). %right
      L1 = [X | L3], partition(L, Y, L1, L3). %buggy
p10(true, [X | L], Y, L1, L2) :-
    L2 = [X | L3], partition(L, Y, L1, L3).
```

ゴールとしてqsort([2,1,3],X)を与えてみると、このプログラムはデッドロックを起こす。プログラムが正しければ、Xは[1,2,3]になるはずである。そこでこれに対してアルゴリズミックデバッギングを行ってみる。

デバッガは以下のような実行モデルを提示する。この実行モデルは簡略化されていて、最終的に何が入力されて何が出力されたかだけを示している。これは6.2の3にあたるものである。これだけでは正誤を判断するのに十分な情報とは言えないことは前述した通りだが、大抵の場合これで十分なので効率を考えてこのように示す。これで十分でないとき、疑問を感じた時に、より詳しく見せるようにすればよいであろう。

```
<Goal> : qsort(_2,_3,_4)
[_2]  IN = [2,1,3]              ((no output))
[_3] ((no input))              OUT =[1,3,2|$V(5)]
[_4] IN = nil                  ((no output))
<Ins>  : qsort([2,1,3],[1,3,2|_0],nil)
query>
```

<Goal>は、前述したG_{skel}のことで、<Ins>がG_{ins}である。入出力はG_{skel}の変数(引数)ごとに表示しているが、変数_3に関する出力が、本来[1,2,3]になるはずが、間違った出力を行なっている。

そこで、プログラマは out と答える。この繰り返しを以下のように行なう。

```
<Goal> : partition(_6,_7,_8,_9)
[_6]    IN = [1,3]              ((no output))
[_7]    IN = 2                  ((no output))
[_8]    ((no input))            OUT =[1,3], OUT =nil
[_9]    ((no input))            ((no output))
<Ins>  : partition([1,3],2,[1,3],_1)
query>out.
<Goal> : p10(_10,_11,_7,_8,_9)
[_10]   IN = false              ((no output))
[_11]   IN = [1,3]              ((no output))
[_7]    IN = 2                  ((no output))
[_8]    ((no input))            OUT =[1,3], OUT =nil
[_9]    ((no input))            ((no output))
<Ins> :  p10(false,[1,3],2,[1,3],_1)
query>out.
<Goal> : partition(_12,_7,_8,_13)
[_12]   IN = [3]                ((no output))
[_7]    IN = 2                  ((no output))
[_8]    IN = [1,3]              OUT =nil
[_13]   ((no input))            OUT =[3]
<Ins> :  partition([3],2,[1,3],[3])
```

上の partition/4 のプロセスで、第3引数に意図しない入力が入って、第3引数を [1,3] に具体化している。そこでプログラマは in と答える。

```
query>in.
Buggy clause:
p10(false,[1,3],2,[1,3],_1) :-
      [1,3] = [1,3],
      partition([3],2,[1,3],[3]).

p10(false,[_0|_1],_2,_3,_4) :-
      _3 = [_0|_5],
      partition(_1,_2,_3,_5).
```

するとデバッガは、バグのある定義節を発見する。最初に表示したのがインスタンスを定義節に当てはめたもの (clause instance) で、その後のものが定義節である。従来の Prolog と同様な計算木のモデルでは入出力を考えずに partition([3],2,[1,3],[3]) というインスタンスを提示するだけなので、これはプログラマにとって意図しないインスタンスであるから no と答えてしまい、デバッガは正しくバグを発見することはできなくなってしまう。

6.3.2 入出力の因果関係によるバグの例

前節の例は入力か出力かそのモードを解析することによって解決することが出来るバグであり、6.2の3の表示の仕方で十分であった。しかし、次に示すバグの例はBrock Ackermannの例と同様なもので、このような表示だけでは不十分である。このような時には入出力の因果関係も調べなければならない。

以下のプログラムは複数のプロセスがあって通信をし合うものであるが、全てのプロセスの出力がマージされて、それを各プロセスが入力として受け取るようになっている。あるプロセスが出力すると、それは自分を含めた全てのプロセスに渡される。各プロセスはそれに応じて出力を出して応答する。これは図1のように表せる。ここでは簡単のためプロセスは二つで、一つはターミナルで出力として他のプ

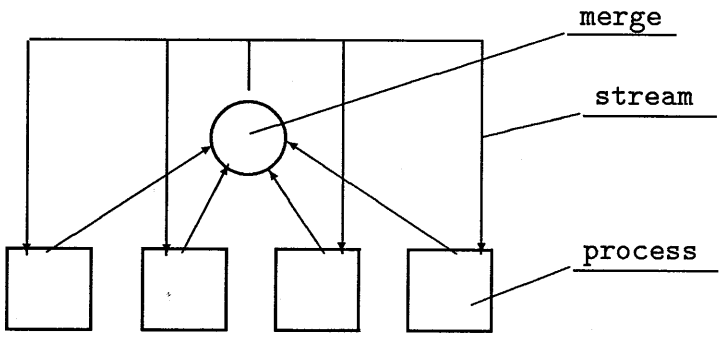

図 1: プログラムの例の説明

ロセスに命令を出し、受け取ったストリームはそのままモニタする。もう一つのプロセスは命令を受け取ってそれに応じた出力を出す。リストの要素二個分が一つの命令となっていて、例えば [getstr,ab] という命令には [string,[97,98]] を、 [getsym,[120,121,122]] という命令には [symbol,xyz] を出力する。

```
term1(I,O):- merge(I, Proc, O), proc1(O,Proc).
term2(I,O):- merge(I, Proc, O), proc2(O,Proc).

proc1([getstr,X|_], O) :- O = [string,Y], compute(string,X,_,Y).
proc1([getsym,X|_], O) :- O = [symbol,Y], compute(symbol,X,_,Y).

proc2([getstr|I], O) :- O = [string|O1], proc2_str(I,O1).
proc2([getsym|I], O) :- O = [symbol|O1], proc2_sym(I,O1).
proc2_str([X|_],O) :- O = [Y], compute(string,X,_,Y).
proc2_sym([X|_],O) :- O = [Y], compute(symbol,X,_,Y).

merge([A|X],Y,Z) :- Z = [A|Z1], merge(X,Y,Z1).
merge(X,[A|Y],Z) :- Z = [A|Z1], merge(X,Y,Z1).
merge([],Y,Z) :- Z = Y.
merge(X,[],Z) :- Z = X.
```

ここで、term2([getstr,abc],X) を実行する。結果は X = [getstr,abc,string,[97,98,99]] となるはずだが、実行によってはそうならない場合もある。その場合の結果は以下のようになる。

```
term2([getstr,abc],[getstr,string,[115,116,114,105,110,103],abc])
  |-merge([getstr,abc],[string,[115,116,114,105,110,103]],
          [getstr,string,[115,116,114,105,110,103],abc])
  |-proc2([getstr,string,[115,116,114,105,110,103],abc],
          [string,[115,116,114,105,110,103]])
```

これは二つのサブプロセスとも最終結果としての入出力を見ただけではプログラムの意味の集合に含まれている。しかし実際には proc2 は間違っていて、正しくは proc1 のようでなければならない。これは 6.2の 3の表示の仕方では分からないバグである。このような場合は以下のように入出力の因果関係も調べる。

```
<Goal> : term2(_1,_2)
[_1]   IN = [getstr,abc]
       ((no output))
[_2]   ((no input))
       OUT = [getstr,string,[115,116,114,105,110,103],abc]
<Ins> : term2([getstr,abc],[getstr,string,[115,116,114,105,110,103],abc])
query>output(2).
OUTPUT OF _2 :
guard{_1 = [_3|_4] / [getstr,abc]}
  |-_2 = [_3|_5]
    |-guard{_6 = [_7|_8]}
        | <-guard{_2 = [getstr|_9]}
        |     |-_6 = [string|_10]
        |       |-guard{_9 = [_11|_12]}
        |           |-_10 = [_13]
        |             |-compute(string,_11,_14,_13)
        |                |-> _13 = [115,116,114,105,110,103]
        |-_5 = [_7|_15]
          |-guard{_8 = [_16|_17]}
              |-_15 = [_16|_18]
                |-guard{_17 = nil}
                    |-_18 = _4
query>out.
```

これは _2 のリストの二番目の要素として abc が来るべきところを先に string が来てしまって、これを計算した結果を返してしまっている。それは _2 の一番目の要素が決まった時に _6 に出力があることが原因である。

```
<Goal> : proc2(_2,_6)
```

```
[_2]   IN = [getstr,string,[115,116,114,105,110,103],abc]
       ((no output))
[_6]   ((no input))
       OUT = [string,[115,116,114,105,110,103]]
<Ins> : proc2([getstr,string,[115,116,114,105,110,103],abc],
              [string,[115,116,114,105,110,103]])
query>output(2).
OUTPUT OF _6 :
guard{_2 = [getstr|_9] / [getstr,string,[115,116,114,105,110,103],abc]}
  |-_6 = [string|_10]
    |-guard{_9 = [_11|_12]}
        |-_10 = [_13]
          |-compute(string,_11,_14,_13)
             |-> _13 = [115,116,114,105,110,103]
query>input(1).
INPUT OF _2 :
guard{_1 = [_3|_4]}
  | <-_1 = [getstr,abc]
  |-_2 = [_3|_5]
    |-guard{_6 = [_7|_8] / [string,[115,116,114,105,110,103]]}
        |-_5 = [_7|_15]
          |-guard{_8 = [_16|_17]}
              |-_15 = [_16|_18]
                |-guard{_17 = nil}
                    |-_18 = _4
query>out.
```

結局 _2 = [getstr|_9] となった時点で _6 = [string|_10] としているところが間違っているので、これも出力が間違っていることになる。

```
<Goal> : proc2_str(_9,_10)
[_9]   IN = [string,[115,116,114,105,110,103],abc]
       ((no output))
[_10]  ((no input))
       OUT = [[115,116,114,105,110,103]]
<Ins>  : proc2_str([string,[115,116,114,105,110,103],abc],
                   [[115,116,114,105,110,103]])
query>yes.
Buggy clause:
proc2([getstr,string,[115,116,114,105,110,103],abc],
     [string,[115,116,114,105,110,103]]) :-
    [string,[115,116,114,105,110,103]] = [string,[115,116,114,105,110,103]],
```

```
    proc2_str([string,[115,116,114,105,110,103],abc],[[115,116,114,105,110,103]]).

proc2([getstr|_1],_2) :-
      _1 = [string|_3],
      proc2_str(_2,_3).
```

7 比較検討

並列論理型言語におけるアルゴリズミックデバッギングの研究としてはまず [Takeuchi86] があげらあれる。これはGHC のアルゴリズミックデバッガであるが、ここではインスタンスを成功、中断、失敗の三つの集合に分けた形でプログラムの意味を表している。しかしここでのインスタンスは入出力の概念をとり入れていないのでプログラムの意味としては不十分であることが [Takeuchi87] に述べられており、入出力のモードの間違いによるバグには対応できない。

Concurrent Prolog におけるアルゴリズミックデバッギングの研究としては [Lichtenstein88] があげられる。これは入出力は表しているので、モードの間違いによるバグは対応できるが、入出力の因果関係は示していない。

[Takeuchi87] では入出力の因果関係をとり入れたセマンティクスを示し、これがデバッグに応用できることを示唆しているが、本論文ではそれを実現したことになる。

8 結論

本論文では並列論理型言語 FLENG のプログラムの実行の様子を、通信をするプロセスの形で表し、これをデバッグに用いた。並列論理型言語の宣言的意味として入出力の因果関係が問題にされているが、このモデルはそれを表現することができる。プログラマが意図するプログラムの意味をそのような宣言的意味として考え、プログラムの実行をこのモデルで表せばそれがその意味の集合に含まれるかわかる訳である。

ここでは、プログラムの意味的解析というより、実際にプログラムを実行してみて、その様子を表現することでそれとプログラムの意図された意味とを照らし合わせるという立場をとっている。それがデバッグの意味と考えている訳であるが、非決定的なコミットでその全ての場合を解析するようにするなどすれば、プログラムの検証などにも適用できるであろう。

参考文献

[Brock81] Brock,J.D. and Ackermann,W.B.: *Scenarios: A Model of Nondeterminate Computation*, Lecture Notes in Computer Science, No.107, 1981.

[Clark86] Clark,K. and Gregory,S.: *PARLOG: Parallel Programming in Logic*, In ACM Transactions on Programming Language and Systems, Vol.8, No.1, 1986.

[Ezaki86] 江崎令子, 宮崎敏彦, 太細孝: *GHC サブセット逐次型処理系のデバッグ環境*, 情報処理学会第33回全国大会, 1986.

[Lichtenstein88] Lichtenstein,Y. and Shapiro E.: *Abstract Algorithmic Debugging*, Logic programming: proceedings of the fifth international conference and symposium,1988.

[Murakami88a] Murakami,M: *An Axiomatic Verifivation Method for Synchronization of Gurded Horn Clauses Programs*, Technical Report TR-339, 1988.

[Murakami88b] 村上昌己: 無限プロセスを含む並列論理型プログラムの宣言的意味論, ICOT,1988.

[Nilsson86] Nilsson,M. and Tanaka,H.: *Fleng Prolog - The Language which turns Supercomputers into Prolog Machines*, In Wada,E. (Ed.): Logic Programming '86, LNCS 264, Springer-Verlag, 1986.

[Shapiro83a] Shapiro,E: *A Subset of Concurrent Prolog and Its Interpreter*, Technical Report TR-003, ICOT, 1983.

[Shapiro83b] Shapiro,E: *Algorithmic Program Debugging*, MIT Press, 1983.

[Takeuchi86] Takeuchi,A: *Algorithmic Debugging of GHC Programs and Its Implementation in GHC*, Technical Report TR-185, ICOT, 1986.

[Takeuchi87] Takeuchi,A: *A Semantic Model of Gurded Horn Clauses* Technical Report, ICOT, 1987.

[Ueda85a] Ueda,K.: *Guarded Horn Clauses*, Technical Report, TR-103, ICOT, 1985.

[Ueda85b] Ueda,K.: *GHC Compiler User's Guide*, NEC Corporation, 1985.