6N-4

Committed-Choice 型言語 FLENG のデバッガ DEBU

舘村 純一, 田中 英彦
{tatemura,tanaka}@mtl.t.u-tokyo.ac.jp
東京大学工学部*

1 はじめに

並列論理型言語のデバッギングにおいて、従来の逐次論理型言語のようにトレースしていくことは困難である。同時にいくつものゴールが存在して、互いに通信し合いながらそれぞれリダクションされていくので、これをわかり易くトレースする必要がある。

また、論理型言語としての特徴を生かした宣言的なデバッグも考えられるが、GHC や Concurrent Prolog のような並列論理型言語は、ホーン節に同期の機構を付加した Committed-Choice 型言語であるから、純粋な論理プログラムとはいえない。そのため、宣言的意味の中にも操作的な要素が入ってくるため、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。

我々の研究室で開発した Committed-Chioce 言語 FLENG を対象としたデバッガ DEBU は、プログラムの実行の様子を、通信をしあうプロセスとしてモデル化し、これに基づいてデバッグを行なうものである。

2 プロセス型実行モデル

ここで提案するプロセス型実行モデルにおける「プロセス」は、一つのゴールだけでなく、そのゴールからリダクションを繰り返すことによって生成される全てのゴール(これをサブゴールと呼ぶ)を含めて考える。つまり、プロセスの実体はある一つのゴールから生成されるゴールの集合である。プロセスの内部でリダクションが行なわれ、ゴールが生滅してもプロセスを外部から見れば同じ一つのプロセスとしてとらえられる。その動作の様子は外部からはプロセスの入出力としてとらえられる。

ゴールがリダクションされていくつかのサブゴールができるのに対応して、プロセスの内部はまたいくつかのサブプロセスとして表現できる。これを計算木と対応付けて考えてみる。計算木の内部には、あるノードをルートとするような部分木が存在するわけであるが、このサブトリーは一つのプロセスにあたる。計算木が部分木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。このようにプロセスを考えると、定義節はプロセスとサブプロセスの関係を規定するものであると考えることができるであろう。

プログラム中で実行されるあるゴール G と、そこから生成されるゴールからなる集合を実体とするプロセスを、「G に対応するプロセス」という。ゴール g_1 が g_2, g_3 にリダクションされるとすれば、g_2 に対応するプロセス、および g_3 に対応するプロセスは g_1 に対応するプロセスのサブプロセスとなる。

ゴール G に対応するプロセスは外部から見た場合以下のように表現できる。

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

G_{skel} は G の *skeletal predicate* であり、ゴールの引数が全て個別の変数となっているもので、この変数が外部との通信の窓口と言える。

I,O は *Input*/*Output* であり、これがプロセスの入出力を表す。G_{skel} の変数がプロセスの外部及び内部からどのようにユニファイされ具体化されていくかを示したものである。これについては3節において説明する。

S は *Status* で、プロセスの状態を表す。これには次の状態がある。

- *terminate* : プロセス内のゴールの実行が全て終了している状態を示す。
- *suspend*(*deadlock*) : 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示す。外部からの入力によって内部のゴールがアクティベートされた場合は *active* に変化する。外部からのどのような入力によっても *active* に変化する可能性のない場合、この状態を特に *deadlock* とよぶこともできよう。
- *active* : プロセスの内部にアクティブなゴールが残っている状態を示す。

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。

3 入出力因果関係

並列論理型言語における宣言的意味を入出力の集合で表すことがあるが、入出力の集合が同じものでも、非決定的なプログラムと合わせると結果が違ってくる可能性があり、入出力の因果関係も考慮しなければならない [1]。そこで、入出力因果関係がわかるようにプロセスの入出力を定義する。

あるゴール P に対応するプロセスの G_{skel} が、

$$p(V_1, \cdots, V_n)$$

とすると、$V_1, \cdots, V_n$ は入出力の窓口と考えられるから、それぞれに対応した入出力の組が存在する。よって P に対応するプロセスの入出力 I_P, O_P は次のようになる。

$$\begin{cases} I_P = I_{PV_1}, \cdots, I_{PV_n} \\ O_P = O_{PV_1}, \cdots, I_{PV_n} \end{cases}$$

I_{PV_i}, O_{PV_i} は P の V_i における入出力である。

*DEBU: The Debugger of the Committed Choice Language FLENG
Junichi TATEMURA, Hidehiko TANAKA,
the University of Tokyo

定義節はプロセスとサブプロセスの関係を規定するものとして位置付けられる。プロセスの入出力とサブプロセスの入出力の関係も定義節によって決まる。これにより、プロセスの入出力は再帰的に定義できる [2]。

FLENG にはガードゴールがないので、ガードの役割をする部分はヘッドの部分だけである。ヘッドの引数に変数でない項が書かれている場合、これが入出力の因果関係を決める要因となる。例えば、

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

というプログラムで、append([1],[2],X) というゴールが実行された場合、このゴールに対応するプロセスの第三引数の(実行終了後の)出力は以下のようになる。

```
Gskel: append(A,B,C)
out(C): cond(A = [D|E])
          |-C = [D|F]
            |-cond(E = [])
              |-F = B
```

この cond という部分が入出力の因果関係を表す部分である。

4 実行トレース

並列論理型言語で実行トレースを行なう場合、実行順序が決っている Prolog と同様にトレースするわけにはいかない。逐次的に順序を決めて実行するとしても、サスペンド・アクティベートが起きた場合実行の流れが追いにくい。そこで、ゴールをキューに入れて管理しておき、これををリストしてどのゴールをリダクションするかをユーザが選び、リダクションの結果生成されたゴールをキューに加えることが考えられる。しかしこの方法ではゴール間の関係の理解が困難で、全体の動作が把握しにくい。

本デバッガでは、階層的なキューを用いてゴールを管理し、プロセス型実行モデルに対応付けた。つまり、キューの中に更にキュー(サブキュー)を持つことが可能で、これが一つのプロセスにあたるわけである。キューの中にあるゴール G を、サブキューをつくってその中に入れると、以後、このゴールからリダクションされてできるサブゴールはこのサブキューの中にできる。通常はサブキューの中のゴールは見えず、「G に対応するプロセス」として扱われることになる。プロセスの外側からは内部の各ゴールに対してでなく、プロセス全体に対して操作を行なうことになる。そのプロセスの内部のゴールを細かく操作したければそのサブキュー(プロセス)の中にはいることができる。そうすると、今までみえていた(プロセスの)外側のゴールは見えず、サブキュー内のゴール(及びサブキューのサブキュー)が直接見えるようになる。

実行のある時点で存在するゴールはキューの中にあるが、それまでの経過はサブキュー(プロセス)毎に分割された計算木の形で表示することができる。

本デバッガにおけるゴール実行に対する操作は、基本的には以下のようなものがある。

- キューの中のゴール(とサブキュー)をリストする。
- 計算木を表示する。
- キューから出る(一つ上のキューに行く)。
- ゴールを一つ 選んで、
  - リダクションする。
  - サブキューに入れる。
- サブキューを一つ選んで、
  - リダクションする(回数、スパイポイントなど指定)。
  - サブキューの中にはいる。
  - サブキュー内のゴールをリストする。
  - サブキュー内の計算木を表示する。
  - サブキューを消去する。
  - 入出力を調べる。
  - アルゴリズミックデバッギングを行なう。

5 アルゴリズミックデバッギング

Shapiro は論理型言語の宣言的なデバッギング手法としてアルゴリズミックデバッギングを提案した [3]。従来のアルゴリズミックデバッギングではインスタンスを提示して、これがプログラマの意図するものであるかを答えることによってバグを発見していくものであるが、並列論理型言語では、インスタンスを示すだけでは不十分であり、入出力の因果関係が含まれたモデルを提示しなければならない。そこでプロセス型実行モデルを用いたアルゴリズミックデバッギングが考えられる。プロセス型実行モデルを提示して、それが正しく出力を行なっているかをプログラマに問うようにする。現在、プログラマの答えに、入出力の「どこがどう違うか」という情報を持たせ、より効率的にバグを探索する方法を検討中である。

6 解析ツール

プロセスの入出力は 3節のように表されるが、プログラムが大きくなってくるとその構造がより複雑になる。そこで、この入出力を解析して、欲しい情報だけをわかりやすく示すことが考えられる。現在、このような機能を各種検討開発中である。

また、実行前にプログラムの静的解析を行なうこともできる。述語の呼び出し関係や、述語内の変数参照関係を調べ、スペルミスやタイプ・モードのミスをチェックする [4]。

7 おわりに

今後の課題としては、ウインドウ・グラフィック環境をとりいれた視覚的なデバッグ環境の構築と、FLENG にオブジェクト指向をとりいれた FLENG++ におけるデバッギングが挙げられる。

参考文献

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

[2] 舘村, 田中, " 並列論理型言語 FLENG のデバッガ ", Logic Programming Conference '89, 1989.

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

[4] 小中, 田中, " Committed-Choice 型言語 FLENG のタイプ / モード推論 ", 情報処理学会第 38 回全国大会, 6Q-2, Mar. 1989.