

6Q-2

並列論理型言語 Fleng のデバッガ HyperDEBU における視覚化支援機能

館村 純一, 小池 汎平, 田中 英彦

{tatemura,koike,tanaka}@mtl.t.u-tokyo.ac.jp

東京大学工学部*

1 はじめに

プログラムをデバッグする場合には、まず実行の様子を把握する必要がある。特に細粒度で高並列なプログラムは実行の流れが多数あるので、実行の巨視的な状態をまず理解することがより重要な課題になる。このためには、実行情報を抽象化したグローバルな視野が必要となり、プログラム実行の視覚化手法の問題を解決しなければならない。

デバッグ時には、どの部分をどのように見たいかというユーザの意図が状況によって変化するので、これを反映させることが重要である。我々の視覚化の方針では、ユーザの意図を反映した「付加的な知識」を与えこれを利用して視覚化を行ない、プログラム全部について完全な知識を与えなくても、知識の与え方に応じて高レベルなデバッグを可能にする。

我々は、並列論理型言語の一種である Committed-Choice 型言語 (CCL) の Fleng を対象とするデバッガ HyperDEBU を開発している [1]。HyperDEBU は、ユーザの意図に応じたコントロール / データフローの視覚化機能を持ち、プログラムの視覚的な観察・操作による効果的なデバッグを可能にする。ユーザはブレークポイントによって自分の意図を伝え、デバッガはこの情報を実行の視覚化に用いる。HyperDEBU ではユーザのブレークポイント設定を支援するため、プログラムの解析・ブラウジング機能を備えている。本論文では、このうちデータフローの視覚化支援機能について述べる。

2 データフローの視覚化

HyperDEBU ではデータフローを視覚化するためにストリーム通信に着目する。このストリーム通信は共有変数を用いて行なわれる。一つのゴールが何らかの構造データを変数に代入する (ストリーム出力)。他のゴールはその値が確定すると、それを読んで処理を行なう (ストリーム入力)。構造データは中に新たな変数を含み、これを介して次の通信が行なわれる。

プログラム中にはこのようなデータフローが多数存在するが、HyperDEBU では、ユーザが実行状況を把握するために注目するストリームのみを視覚化する。

ストリーム通信の様子を表現するために、ストリームが生成される様子、それが分配される様子、データの入出力が行なわれる様子を図1のように視覚化する。

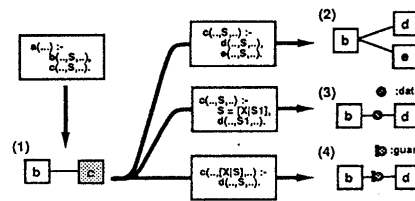


図1: ストリームの生成・分配・入出力

- 図1-(1)はストリームが生成された時に画面に視覚化される図形の様子を表したものである。これは、定義節のボディゴールに共有変数が生まれた時である。
- 図1-(2)はストリームが分配される様子を示している。これは、図にあるような定義節によってゴールcがリダクションされ、変数Sを参照するゴールが増えた時である。
- 図1-(3)はストリームに出力が行なわれた様子を示している。これは、変数Sに構造データが代入された時である。
- 図1-(4)はストリームからの入力が行なわれた様子を示している。定義節のヘッド部に書かれたような構造データを待ってリダクションが行なわれた時である。

3 視覚化支援

着目するストリームを視覚化するには、ストリームに用いられる変数について、図1にあるような各動作にあたる部分をプログラム中で指定する必要があるが、このような各部分指定するのは複雑である。ユーザは着目点を示すのに必要最低限の情報のみを与え、デバッガはこれに基づき必要十分な実行情報を提示することが望まれる。このためには次の点を明確にしなければならない。

1. ユーザの見たいものは何か。
2. これを伝えるのにユーザがどのように指定するか。
3. デバッガが何をどのように見せるか。

ユーザは、述語が引数として持つストリームに着目する。このストリームを用いてどこでデータのやりとりをしているのか、および、この述語がストリームに対していかなるデータ操作を行なっているのかユーザの知りたい点である。

ユーザはストリームとして着目する述語の引数をブレークポイントで指定する。デバッガは、ユーザが指定したこの着目部分がどこどのように通信しているかを示すため、ストリームが生成されてから、ブレークポイントに至り、そこでデータ処

*Facilities of a Debugger HyperDEBU to Visualize Parallel Logic Programs

Junichi TATEMURA, Hanpei KOIKE, Hidehiko TANAKA,
the University of Tokyo

理が行なわれるまでをトレースして表示する。

また、ユーザの指定したポイントをサブゴール中に持たないゴールについては、リダクションされてもそのままゴールとして表示しゴールの内部を抽象化することで、必要十分な情報のみを与える。その内部が見たい場合には、その部分にもブレイクポイントを指定すればよい。

ユーザの指定したブレイクポイントから視覚化対象を特定するため、デバッガ側はプログラムを解析して次の情報を得る。

- プログラムコード中に現れる変数のうち、どれが視覚化されるストリームに該当するのか。
- リダクションによるデータ操作を視覚化するか、あるいはゴールの内部を抽象化するか。

これに基づきデバッガはストリームの各動作を視覚化する。

4 ユーザの指定方法

ユーザは、ストリームを視覚化するためにブレイクポイントでその位置(述語の引数)と型を指定する。ここでストリームの型とは、ストリーム変数に与えられるデータの集合を表す。変数に与えられるデータの中には、次に用いられる新しいストリーム変数が含まれている。データ中の着目すべきストリーム変数は、どのストリームの流れに着目するかによって変わるのでこれをストリームの型として指定する必要がある。

多くのプログラムの場合、ストリームとして用いられるの是一列のリストである。これは次のように表せる。

$$S ::= [_ | S] | \square$$

これは、ストリーム変数で表されるデータがリスト $[_ | _]$ か、 $\square$ かであり、リストの場合その CDR 部が同じ型のストリームであることを表す。

この情報により、ストリームに与えられたデータの中での次のストリーム変数を知ることができる。ストリーム変数 S のデータの中に変数 $S1$ が存在して、これが次のストリーム変数であるということを $\text{next}(S, S1)$ と表すと、上記のストリームの型の定義では、例えば次のことがいえる。

$$\text{next}([X | S], S), \text{next}([X, Y | S], S), \dots$$

このようなストリームの型指定を導入すれば、木構造をしたデータをストリームとみなすなどの一般的なストリームにも対応できる。

5 解析手法

解析手法の概略説明のため、ここでは以下の術語を用いる。

- $\text{arg}(G, i)$: ゴール G の i 番目の引数。
- $\text{var}(T, k)$: データ T の中に含まれる k 番目の変数。
- $\text{breakpoint}(A)$: 述語の引数 A にブレイクポイントが設定されている。
- $\text{stream}(A)$: 述語の引数 A はストリームである
- $\text{equiv}(X, Y)$: 変数 X と Y が同一である。

また、定義節のヘッドを H 、ボディを B_i とする。

ストリーム変数を発見するための推論規則は次のようになる。

1. 定義節中でストリームと同じ変数として現れる部分はストリームである。

$$\begin{aligned} & \text{breakpoint}(\text{arg}(B_m, i)) \rightarrow \text{stream}(\text{arg}(B_m, i)). \\ & \text{stream}(\text{arg}(B_m, i)), \text{equiv}(\text{arg}(H, j), \text{arg}(B_m, i)) \\ & \quad \rightarrow \text{stream}(\text{arg}(H, j)). \\ & \text{stream}(\text{arg}(B_m, i)), \text{equiv}(\text{arg}(B_k, j), \text{arg}(B_m, i)) \\ & \quad \rightarrow \text{stream}(\text{arg}(B_k, j)). \\ & \text{stream}(\text{arg}(H, j)), \text{equiv}(\text{arg}(H, j), \text{arg}(B_m, i)) \\ & \quad \rightarrow \text{stream}(\text{arg}(B_m, i)). \end{aligned}$$

2. ストリーム入力が行なわれ、次のストリーム変数が存在する。

$$\begin{aligned} & \text{stream}(\text{arg}(H, i)), \text{next}(\text{arg}(H, i), \text{var}(\text{arg}(H, i), k)), \\ & \quad \text{equiv}(\text{var}(\text{arg}(H, i), k), X) \\ & \quad \rightarrow \text{stream}(X). \end{aligned}$$

3. ストリーム変数にデータが代入され、その中に次のストリーム変数が存在する。ここで、 $U(X, Y)$ は X と Y をユニファイする述語とする。

$$\begin{aligned} & \text{stream}(\text{arg}(U, 1)), \text{next}(\text{arg}(U, 2), \text{var}(\text{arg}(U, 2), k)), \\ & \quad \text{equiv}(\text{var}(\text{arg}(U, 2), k), X) \rightarrow \text{stream}(X). \end{aligned}$$

ゴール G のリダクション時のデータ操作が視覚化されることを $\text{visible}(G)$ で表すと、これを推論する規則は次のようになる。

$$\begin{aligned} & \text{breakpoint}(\text{arg}(B_m, i)) \rightarrow \text{visible}(B_m). \\ & \text{stream}(\text{arg}(B_m, i)), \text{visible}(B_m), \text{stream}(\text{arg}(H, j)) \\ & \quad \rightarrow \text{visible}(H). \end{aligned}$$

6 実装方式

このような Fleng プログラム解析機能を Fleng 自身によって並列実装する。各述語に対応するプロセスを割り当て、述語の呼びだし関係に基づいてネットワークを構成する。これらが推論で得られた知識をメッセージとして通信し合いながら並列に解析を行なう。各述語毎の知識を各プロセスが分散管理することで、並列度の高い処理が行なわれる。

7 おわりに

本論文で述べた視覚化支援機能に対して、さらに以下の機能拡張が考えられる。

- 静的デバッグ: ストリーム変数に正しい型のデータが与えられていない、ストリーム変数が共有されていない、入出力モードのエラーなどのバグを検出する。
- 静的データフローの表示: 述語の呼びだし関係の中で、ストリーム変数がどのように伝搬しているかを示す。

参考文献

- [1] 館村, 小池, 田中: マルチウィンドウデバッガ HyperDEBU における細粒度高並列プログラムの実行のデータフローの視覚化, 並列処理シンポジウム JSPP'92 (1992).