ICOT Technical Memorandom: TM-0986

TM-0986

GHC上のプロセス指向デバッガ

前田　宗則（富士通）、魚井　宏高、
都倉　信樹（大阪大学）

December, 1990

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03)3456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

GHC 上のプロセス指向デバッガ
Process and Stream Oriented Debugger for GHC Programs

前田宗則
Munenori MAEDA
富士通（株）国際情報社会科学研究所
IIAS-FUJITSU LIMITED
魚井宏高　　都倉信樹
Hirotaka UOI　　Nobuki TOKURA
大阪大学基礎工学部情報工学科
Faculty of Engineering Science, Osaka Univercity

e-mail: m-maeda@iias.fujitsu.co.jp

内容梗概

本論文では、GHC プログラミング技法としてよく用いられている、プロセスとストリーム計算モデルに基づく典型的なプログラムに対するデバッグ手法を提案する．本手法で用いられるデバッガはプロセス指向デバッガと呼ばれ、ターゲットプログラムの実行過程で生成されるプロセスとストリームを管理することで、ストリーム上のデータの参照と変更、プロセス間のデータ送受信におけるタイミングを変更する機能を提供し、実行状況に応じて変化するプロセスとストリームのネットワークグラフや各プロセスが入出力したデータの履歴を効果的に可視化する．これらの機能により、プログラマのイメージに沿った実行トレースの可視化と実行の制御が可能になる．

1　はじめに

これまでに、並列論理型言語 GHC(Guarded Horn Clauses) [Ueda 85] [1]プログラムのデバッグ手法がいくつか提案されている．それらのデバッグ手法は、“GHC ではプログラムが並行に実行されるので、単純に実行トレースを参照しただけでは、プログラム全体の動作が把握できない”との問題認識の下に開発されてきた．これらは大きく分けて、

1. 実行トレース情報をうまく可視化し、さらにその情報量を適度に制限する手段を提供する実行トレース法 [Hirano 89][PIMOS 89]
2. 論理的な推論により表示する情報を刈り込んで少なくするアルゴリズミックデバッグ手法 [Lloyd 86] [Takeuchi 86] [Takeuchi 87]

であり、その二つに共通する理念は、“多すぎる情報の提示は、デバッグに役立たない．箘いにかけて適当な情報のみを取り出すことが重要である．”というものである．

ところで、比較的規模の大きいプログラムを GHC で記述するとき、いくつかの機能モジュールをストリームと呼ばれる通信路で接続することで全体を構成することがよく行なわれる．同様に、各機能モジュールは、ストリームとプロセスから構成される．各プロセスは、ストリーム上のデータをすべて消費するまで連続してデータを読み込みを行ない、必要があればストリームにデータを順次送り出していくといった実行イメージの下で記述される．こうしたプログラムの根底にあるのは、オブジェクト指向プログラミング [Shapiro 83-2] とストリーム計算 [Kahn 77] のモデルであろう．

さて、従来のデバッガを用いてこのような作法に従ったプログラムをデバッグしたり、実行状況を可視化するとき、次のような点で不十分である．

- ゴールを実行の単位とするので、連続して実行されるプロセスという概念がない．それゆえに、プロセスはいつどのような入力を得て、どのような内部状態に変化し、何を出力したかという因果関係が明確ではない．

[1]今後、本論文では GHC を Flat GHC の意味で用いる．

- 注目するプロセスとそうでない内部の手続きが混在して表示される.
- ストリームは他のデータ構造と区別されないので、参照したい部分をのみを表示させることは困難である. 通常、全てのデータを冗長に表示したり、システム側で適当に決められた部分構造のみを表示してしまい、本当に必要な部分の情報が得られないことがある.

これらの問題は、利用者に適当な視点を与えないことに起因する. 適当な視点とは、一つは、各プロセスをブラックボックスと見立てて、内部でどのような処理が行なわれたかを関知することなく、ストリームからどのようなデータを読み込み、それによって内部状態がどう変化し、どのようなデータを出力したかというといった因果関係のみを対象にするようなマクロな視点のことであり、他は、各プロセスの処理をプリミティブな述語の実行に還元して、変数にどのような束縛が与えられたのかを対象にするようなミクロな視点である.

本論文では、1. に属する新しいデバッグ手法として、プロセス指向デバッグ手法を提案する. これは、"デバッグとは、対象とするプログラムに大きく依存するものである" という事実を積極的に利用することで、あるプログラミングパラダイムのもとで記述されたプログラムを利用者のイメージを重視しつつデバッグするパラダイム指向デバッグ手法 [Maeda 89] のうちの一手法と位置づけられるもので、典型的な "プロセスとストリーム" の実現法を用いたプログラムに対してのみ、マクロな視点でのデバッグ作業を支援するものである.

本論文の構成は、次の通りである. まず、2 章で本論文が対象とするプロセスとストリームの概念と、それらが GHC 上ではどのように実現されているかについて述べる. 次に 3 章では、プロセス指向デバッギングとはどうあるべきかということと、そのデバッガを構成する要素技術として、プロセスの設定、ストリームの管理技術、プログラムの実行制御機能について述べる. 4 章では、実行トレースの表示に対応するプロセスとストリームの可視化技法について述べる. 5 章では、試作されたデバッガの実行例を示す. 最後はまとめと今後の課題である.

2 プロセスとストリームのモデル

2.1 プロセスとストリームの概念

ストリーム ストリームとは、データの列である. ストリームに関する操作としては、先頭のデータを取り出す読み込み操作と末尾にデータを設置する書き込み操作が許される. ストリームの抽象化した定義に関しては、文献 [Tribble 87] に詳しい記述がある.

プロセス GHC で用いられるプロセスの概念には、次のような二つがある.

広義のプロセス 文献 [Ueda 86] で述べられている GHC プログラムのプロセス的解釈に従う.
プロセスとは、ゴールのことである. ゴールがリダクションされてサブゴールが生成することに対応して、あるプロセスはいくつかのサブプロセスを生成して、それらに実行を依託すると解釈する. 各プロセス間の通信は、対応するゴールの共有変数を通して行なう.

狭義のプロセス この意味でのプロセスは、文献 [Shapiro 83-2] におけるオブジェクトの概念に近い.
各プロセスは、他のプロセスによって直接に読み出し書き込みが許されない内部状態を持ち、入出力ポートに接続されるストリームを通して通信する. プロセスは、受け取ったデータによって内部状態を変更し、実行を継続する.

狭義のプロセスは、内部状態を持ちプロセスの継続を認めている点で、広義のプロセスより抽象性が高い概念になっている. 以後本論文で、特に断わらないでプロセスというときは、この狭義のプロセスを指すことにする.

2.2 プロセスとストリームの実現

GHC プログラムにおけるプロセス、ストリームの典型的な実現は、次のようになる.

1. プロセスの実現
プロセスは、プロセスを表す述語を呼び出すことによって生成される. 同様に、プロセスの継続は、そのプロセスの継続を表す述語を呼び出すことによって行なわれる. 多くの場合に再帰的に呼ばれるゴールがプロセスの継続を表す. 例を示す. 以下は、よく知られたフィボナッチ数列を生成するプログラムである.

```
go(X):- true | fib(0,1,X).
fib(N1,N2,X):- true | X=[N2|X1], N3 := N1+N2, fib(N2,N3,X1).
```

第一節に現われるゴール fib(0,1,X) は、プロセス fib を生成する. また、第二節のボディゴール fib(N2,N3,X1) は、プロセス fib の継続を表すゴールである.

2. ストリームの実現

ストリーム上のデータの並びは、未完成メッセージと呼ばれるデータ構造の連鎖によって実現される．一般的に利用されるのはリストである．そこで、以下ではストリームは必ずリストで実現されているとして、例にそってストリームに関するいくつかの用語を定義する．

```
(1) t(X,Y):- true | pA(X,U), pB([1,2|U],V), pC(U,[3,4|V],Y).
(2) pA([E|S],T):- true | T=[N|T1], pA(S,T1).
(3) pB([E1,E2|S],T):- true | T=[N1,N2|T1], pB([E2|S],T1).
(4) pC([E|S1],S2,T):- true | T=[N|T1], pC(S1,S2,T1).
(5) pC([],S,T):- true | T=S.
```

pA,pB,pC は、プロセスを表すゴールである．各々の引数は、ストリームが接続される入出力ポートの役割を果たす．

(a) **ストリームの生成と名前**

ストリームは、それが接続されるプロセスを表すゴールの呼び出しと同時に新しく生成される．新しく生成された各々のストリームは、異なる名前を持つ．また、ストリームの名前は、それを参照する変数によって伝播される性質を持つ．ストリーム St を変数 Var と単一化するとき、Var は St と同じデータの並びと同じ名前を持つストリームになる．

(b) **ストリームデータの読み込み**

内容がリスト $[Shead \mid Stail]$ であるストリーム St とリスト $[Head \mid Tail]$ を単一化するとき、一つのデータ Shead を読み込んだという．Tail が変数であるならば、Tail は St と同じストリーム名を持ち、St の先頭のデータを除いたデータの並びを持つストリームになる．この Tail を St の分流と呼ぶことにする．あるいは、Tail が変数でないならば、Stail と Tail についてデータの読み込みもしくは書き込みが再帰的に行なわれる．(2),(3),(4) 節において、節の頭部で行なわれる各ストリームと [E | S], [E1, E2 | S],[E | S1] の単一化によって、それぞれデータ E,E1 と E2,E が読み込まれる．また、変数 S,S,S1 は、それぞれのストリームの分流である．

(c) **ストリームデータの書き込み**

ストリーム St を伝播する変数 VSt に対して、リスト $[Head \mid Tail]$ を単一化するとき、ストリーム St に一つのデータ Head を書き込んだという．このとき、ストリーム St の内容は、リスト $[Head \mid Stail]$ となる．さらに Stail と Tail についてストリームデータの書き込み操作を繰り返す．(2),(3),(4) 節において、節のボディで行なわれる単一化操作 T = [N | T1],T = [N1, N2 | T1], T = [N | T1] によって各ストリームにそれぞれデータ N,N1 と N2,N が書き込まれる．また、各場合における変数 T1 は、それぞれのストリームの分流である．

(d) **ストリームの連結**

ストリームを表す項（変数またはリスト）S とストリームを表す項 T が単一化されたとき、二つのストリームは連結されたという．連結されたストリーム S と T は、同じストリーム名を持つ．(5) 節におけるボディで行なわれる単一化 T=S がこれにあたる．(a) で述べたように、(1) 節でプロセスを表すゴール pB,pC の呼び出しにより、それぞれの引数に設定されている [1, 2 | U],[3, 4 | V] は、新しく生成したストリームであると解釈される．このとき、U,V は既に異なる名前を持つストリームであるので、ストリームの連結が実行される．その結果、[1, 2 | U] は U と、[3, 4 | V] は V とそれぞれ同じ名前を持つストリームとして観測される．

以上のように定められたストリームに関して、次の性質が成り立つ．

<u>ストリームの性質</u>

X, Y, Z は、異なる名前を持つストリームであるとする．このとき、

$$X = [a_1, \cdots, a_m \mid Z], Y = [b_1, \cdots, b_n \mid Z]$$

の操作により、X, Y, Z は同じ名前を持つストリームとなる．一般に、与えられた同じ名前を持つストリーム X, Y には、関係 $^{\exists}A, B(append(A, Z, X) \wedge append(B, Z, Y))$ が成立する．Z は、ストリーム X, Y の分流である．■

3 プロセス指向デバッガ

まず、3.1 節でプロセス指向デバッガを用いたデバッグ方法について述べる．3.2 節以降では、プロセス指向デバッガの実現の要素技術について触れる．

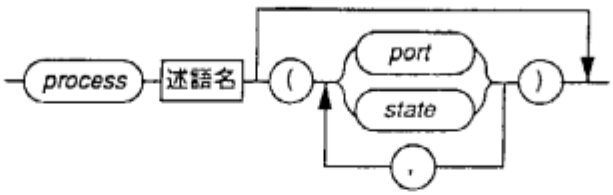


図 1: プロセス宣言

3.1 プロセス指向デバッギング

狭義のプロセスに注目し、以下のようなデバッグ情報と実行制御機能を提供するデバッガをプロセス指向デバッガと呼ぶことにする.

- デバッグ情報
  1. プロセスとストリームからなるネットワークグラフ
  2. 各プロセス単位に、入力あるいは出力したデータおよび内部状態の履歴
  3. ストリーム上に存在するデータ
- 実行制御機能
  1. 各プロセス毎に実行の中断・再開・放棄を行なうこと
  2. 各ストリームに対して、
     (a) データの流れをせき止めたり放出すること
     (b) データの内容を変更すること

プロセス指向デバッガでは、生成したプロセスと各プロセスへの入出力データ、及び内部状態の値をトレースすることで大まかなプログラムの実行が提示される. また、プロセス内部の実行について深く立ち入らない. プロセス内部で発生したエラーは、プロセスの継続、入力データの消費、あるいは出力データの生成に関する異常として検出されることになる. プログラマは、異常状態に陥ったプロセスに対するデータの供給を停止したり、出力されるデータをせき止め、その内容を変更した後、他のプロセスに送信することで実行を継続する. また、入出力データを保存した後、プログラムを修正し、生成されたプロセスに、先程の入力データを読み出して与え、出力されたデータを比較することで、バグを含むプロセスのテストが容易に行なえることが期待される.

3.2 プロセスの設定

プロセス指向デバッガにおいては、プロセスと考えられる述語とプロセスの内部で実行される局所的な述語を区別することが必要になる. GHC では構文的にそれらを区別することができないので、利用者のプロセス指定が必要になる.

ここでは、プロセスとその述語の引数に関する取り扱いを指定するプロセス宣言を導入する. 引数に関する取り扱い指定とは、ある引数がプロセスの内部状態を保持するためのものであるのか、あるいはストリームが接続するプロセスの入出力口であるのかをそれぞれ、state、port として与えることである. 図 1にこのプロセス宣言のダイアグラムを示す. 図 1中で四角で囲まれているものは非終端記号であり、ここに述語名が記述される.

生成したプロセスの継続をその述語の再帰によって行なうことが多いことは、2 章で述べた. しかしながら、これは、一般的、定性的な主張であり、実際のプログラムでは、他の述語を経由して再帰する間接再帰型プロセスや節のボディに複数の再帰ゴールが存在する場合等様々なものが考えられる. そこで、継続プロセスを表すゴールを利用者が直接指定することにする. 継続プロセスの指定は、プロセスを表す述語の節のボディにあるプロセスを表すゴールのうちの適当な一つを選び、そのゴールの頭に @ 記号を付けることで表現する. プロセス指定の具体例は、5 章で示される.

3.3 ストリームの認識と管理

2.2 節で述べたように、ストリームは、任意の未完成メッセージの連鎖で実現されるが、本デバッガでは、ストリームの統一的な認識と管理のために、リストで実現されているストリームのみを対象とする. これは、実際多くの GHC プログラムにおいて、ストリームがリストで表現されるという事実に拠っている.

本節の目的は、デバッガが

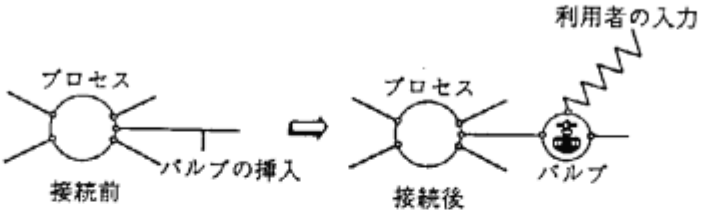


図 2: プロセスとバルブの接続

1. プログラム実行の過程で得られた項（リストまたは変数）が、ストリームを表現する項であるか否かを区別する

2. ストリームであることが知られた二つの項が、同一の名前を持つストリームを表現しているか否かを判定する

ための認識と管理の手続きを与えることである.

まず、1. に関しては、3.2 節で導入されたプロセス宣言を利用することができる. すなわち、プロセスの port 指定されている引数に与えられた項はストリームを表現していると言える.

次に 2. に関しては、2 章で述べたように、ストリームは変数を通して名前の伝播を行なうことから、変数とストリームの名前を管理する機構を用意することで可能となる. 具体的な実現は以下の通りである.

ストリーム管理のための型導入

1. 全てのデータに型のためのタグを設ける. タグとしては、ストリーム型のための stream とストリーム型以外を表す nonsense を用意する.

2. ストリーム型の変数、リスト構成子に対しては、ストリームの名前を格納するタグを設ける.

3. ストリームの名前は変数で実現する.

ストリームをタグ付きのデータで表現する場合、1. と 2. は自明であろう. 3. は、先に述べたストリームの名前に関する性質を満たすのに都合がよい実現技法である. ストリームの名前を論理変数で実現すると、異なるストリーム名を持つストリーム同士が連結すると同じ名前になるということとその影響はそれぞれのストリームの全ての分流に及ぶということが、二つの論理変数を単一化すると同じ実体になるということとその影響はそれぞれの変数を含む全ての項に及ぶということに素直に対応する.

ストリームの名前管理は、次のように行なう. プロセスを表すゴール $p(T_1,\cdots,T_n)$ が呼ばれたときに、デバッガはこのプロセスの引数を次のように置き換える.

$$V_k = \begin{cases} T_k, & \text{if } arg(k) = state\ ; \\ S_k & \text{elseif } arg(k) = port;\ S_k \text{ は、新しく生成したストリーム型未定変数.} \end{cases}$$

ここで、新しく生成されたストリーム変数 S_{k_i} は、各プロセス毎に用意されたプロセステーブルに登録される. プロセステーブルは、継続プロセスの呼び出し回数と内部状態値、ストリーム型データの表形式である. 続いて、デバッガは、ストリーム変数と実引数の単一化：$S_{k_1} = T_{k_1},\cdots,S_{k_i} = T_{k_i}$ を行なう. その後、プロセス $p(V_1,\cdots,V_n)$ を通常の GHC の動作に従って、定義節の頭部と単一化してガードゴールを実行する. ガード実行において、どのような束縛も呼び出し側に与えられないならば、節をコミットし、ボディゴールを実行する. ■

3.4 プログラム実行の制御

本システムでは、3.1 節において提案した実行制御の機能を“バルブ”と呼ばれるバッファの役割を果たすプロセスを用いることで実現している. バルブは、図 2のようにプロセスとストリームの間に挿入される. バルブは、データの入出力口、バルブ制御命令のための入口、読み込んだデータを保存するバッファ、バッファサイズ、バッファに書き込まれているデータの数、読み込みデータの条件の記述を持ち、利用者から入力される制御命令やあらかじめ設定された条件によって、自動データ入出力モード、条件付きデータ入出力モード、データ編集モードに遷移する. これを図 3に示す.

先に述べたバルブの各状態では、次のような処理が行なわれる.

- 自動データ入出力モード
  バルブに到着したデータは、バッファが一杯でないならばバッファに格納される. バッファが一杯であれば、バッファの先頭のデータをバルブから出力する.

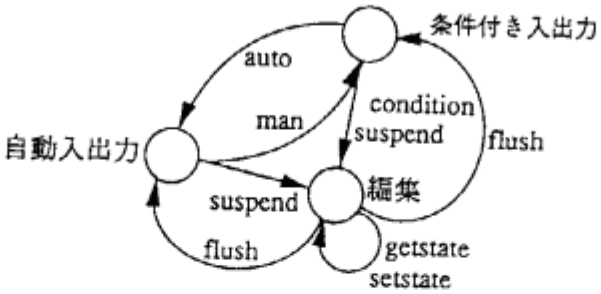


図 3: バルブの制御命令と状態の遷移

- 条件付きデータ入出力モード
  バルブに到着したデータは、バッファが一杯でないならばバッファに格納される．バッファが一杯になったか、あるいは、格納されたデータが設定された条件を満たさないならば、データ編集モードに遷移する．
- データ編集モード
  この状態では、バッファのサイズ、条件、現在までにバッファに読み込まれたデータの参照とそれらをテキストエディタ等によって変更することが可能である．編集後、バッファの全データを出力することで前の入力モードへ再開する．

さて、データの条件の評価は、実行中に得られたデータを D とするとき、以下のように定義された述語 $test(D)$ を実行することで行なわれる．

$$test(In_1) \leftarrow Exp_1 \mid true.$$
$$\vdots$$
$$test(In_k) \leftarrow Exp_k \mid true.$$

ここで、Exp_i は、テスト述語と呼ばれるものの並びである．利用者は、評価すべき述語の名前 $test$ をバルブに設定することで、データの条件を与える．

4 実行の可視化手法

実行の可視化対象として、次の二つを考える．

1. プロセスとストリームからなるネットワークグラフと、
2. プロセスが継続して実行されるとき、継続の前後における、内部状態とストリームの内容の変化．

1. は、実行状況に応じて変化してゆくのでストリームグラフと呼ばれる動画で示すことにする．また、2. は、実行のある時点までの履歴であり、プロセスチャートと呼ばれる静止画によって提示される．

4.1 実行状況とストリームグラフ

ストリームグラフの作画の規則は、次の通りである．

1. プロセス宣言されたゴールが呼び出されたときに、画面上にプロセスに対応する図形を表示する．
2. ストリーム名 A と B の連結が行なわれたときに、ストリーム A がつながるプロセスのポートとストリーム B につながるプロセスのポートを直線で連結する．
3. プロセスが継続する際に、継続前のプロセス図形に接続するストリームを表す直線を消去する．その後新しいプロセスの各ポートの引数を取り出し、他のプロセスの port 引数に同一の名前のストリームがあれば、それらを直線でつないで表示する．

4.2 実行履歴とプロセスチャート

プロセスチャートは、プロセスの計算履歴とプロセス間連結を提示する．プロセスの計算履歴とは、継続の各ステップにおける、

1. ストリームから読み出された、あるいは書き込まれたデータ

2. プロセス内部で起動されたプロセスの集合

3. 内部状態を表すデータ

の三つである．それぞれは、次のようにして得られる．

1. プロセス継続前後の同じ名前を持つストリーム X, Y の差分を取る．ストリーム X, Y の差分とは、2 章で述べたストリームの性質を用いて、$append(A, Z, X) \wedge append(B, Z, Y)$ で定められるストリーム Z を最大長となるように選ぶときの A, B であると定義する．ストリームの差分を取ることによって、

   (a) プロセスを継続する過程で、実際に消費あるいは生産されたデータ

   (b) ストリームを複数のプロセスが共有するマルチリーダ技法において、各プロセスが参照するストリームに初期設定された異なるデータの並び

   を明確にすることができる．例を挙げる．

```
(a') p1([a,b|X]):- true | p1([b|X]).
(b') p2(X):- true | q([a,b|X]), r([c,d|X]).
```

   (a') においては、ストリームから a,b が消費され b がストリーム先頭に押し戻される．それゆえ実質的に消費されるのは a のみである．(b') においては、プロセス q,r がストリームを共有しているが、ストリーム先頭の初期設定がそれぞれ a,b と c,d であって異なっている．

2. あるゴール P から生成されるプロセス集合 $process\_set(P)$ を次のように定義する．

$$process\_set(P) = \begin{cases} \emptyset & \text{if } P = true; \\ process\_set(A) \bigcup process\_set(B) & \text{if } P = (A, B); \\ \{P\} & \text{if } P \text{ is declared as a process;} \\ process\_set(reduced(P)) & \text{otherwise.} \end{cases}$$

$$reduced(P) = \{\theta B \mid^{\exists} (H \leftarrow G \mid B), {}^{\exists}\theta : \theta H = P \wedge \vdash \theta G\}$$

   プロセス Q から生成されるプロセス集合を $process\_set(reduced(Q))$ で与える．

3. プロセステーブルに登録されている内部状態を表す引数を参照する．

これらの計算履歴を用いて、プロセスチャートは、以下のような作画規則に従い描画される．

1. 継続の各ステップは、上から下に適当な距離をとって垂直方向に配置された水平線分で表現する．内部状態値は、それが与えられている引数の位置に合わせて、線分の下に配置する．

2. 継続以外の生成されたプロセスを表す線分は、継続プロセスの垂直方向に適当に右方向にずらして配置する．

3. 各プロセスを接続するストリームは、対応する引数の位置同士を接続する線分で表す．入力ストリームと出力ストリームは、異なる色の線分で表現し区別する．

4. ストリームで消費あるいは生産されたデータは、各線分上あるいはその近傍に配置する．

5 デバッガの実行例

本章では、Macintosh 上に試作したデバッガの実行例を示す．本デバッガは、上田のコンパイル方式 [Ueda 87] に準拠した GHC コンパイラ上で、GHC のメタインタプリタを拡張して記述されている．本デバッガは、マウスとアイコンによる直接操作インターフェース環境を提供しており、いくつかの制御コマンド: プロセスの実行停止と再開、バルブの設定と制御、プロセスチャートの表示、ストリームグラフ上からのプロセス図形の消去等を、それらに対応するアイコンを選択することで行なう．

まずプログラマは、デバッガの実行に先だってターゲットプログラムに対して、プロセスとして観察したい述語にプロセス宣言を与え、プロセス宣言される述語の定義節中に継続プロセスと見なされるゴールがあればそれに @ 記号を付ける操作を行なう．以下の素数生成プログラムにおいては、述語 gen,sift,filter をプロセスとする．

```
process gen(state,state,port).
process sift(port,port).
process filter(port,state,port).
prime(Max,Ps):- true | gen(2,Max,Ns), sift(Ns,Ps).
```

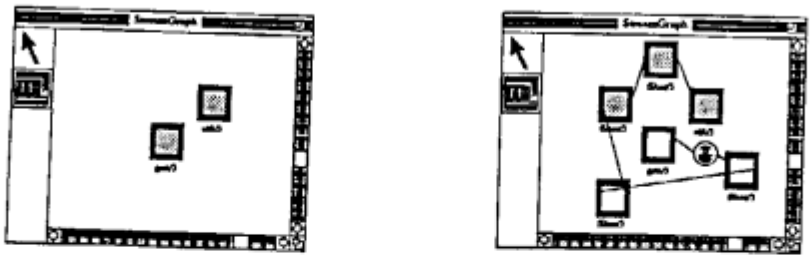
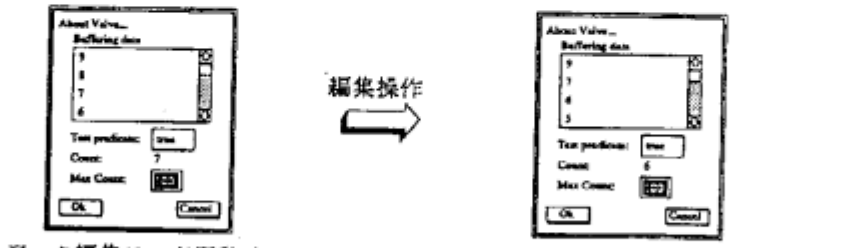


図 4: 素数生成プログラムのストリームグラフその 1 （左図）とその 2 （右図）



データ編集モード遷移時　　データ8を消去してバッファの内容を放出

図 5: バルブによるバッファの変更

```
gen(N,Max,Ns):- N >= Max | Ns =[].
gen(N,Max,Ns):- N < Max, N1 := N+1 | Ns=[N|Ns1], @gen(N1,Max,Ns1).
sift([],Ps):- true | Ps=[].
sift([P|Fs],Ps):- true | Ps=[P|Ps1], filter(Fs,P,Fs1), @sift(Fs1,Ps1).
filter([],P,Fs):- true | Fs=[].
filter([N|Ns],P,Fs):- N mod P =:= 0 | @filter(Ns,P,Fs).
filter([N|Ns],P,Fs):- N mod P =\= 0 | Fs=[N|Fs1], @filter(Ns,P,Fs1).
```

次にデバッガを起動し、ゴール prime(10,Ps) を与える．実行開始直後のストリームグラフは、図 4の左のようになる．実行途中で、プロセス gen がどのようなデータを生産しているのか調べるために、gen の出力ポートにバルブを設定する．最初バルブは、バッファサイズ 100 の自動入出力モードで起動される．十分な時間が経過すると、プロセス gen はデータを順次生成して実行を終了する．このとき、バルブのバッファサイズが実際到着するデータ数より大きいことから、バッファの内容が出力されずに実行がデッドロック状態に陥る．そのときのバッファの内容を参照し変更した上でデータをバルブから送信することを試みた．これを図 5に示す．

データ送信後、デッドロック状態だった実行は自動的に再開して、最終的には図 4の右のようなプロセスグラフを描いた後に全体の実行を終了する．ここで、各プロセスのプロセスチャートを参照したところ、図 6が得られた．このプロセスチャートより、

- gen プロセスはストリームの生産者である．一回の再帰でデータを一つずつ生産し、各引数は、同一のストリームと継続して接続する．
- filter プロセスは第一引数に与えれるストリームの消費者であり、第三引数のストリームの生産者である．これらのストリームは再帰による継続で維持され、一回の再帰で入力ストリームからはデータを一つずつ消費し、出力ストリームからはデータを一つ生産するかもしくはしない場合がある．
- sift プロセスは、一回の再帰で filter プロセスを一つずつ生産する．また、第二引数のストリームの生産者であり、再帰による継続で維持される．

が理解される．

6　まとめと今後の課題

本論文では、プロセスとストリーム計算に沿った典型的なプログラムに特化されたデバッガであるプロセス指向デバッガが提案された．本デバッガでは、実行の制御と実行トレースの提示にあたり、

- プロセスをブラックボックスとして扱うことで、プロセスの内部でどのような手続きが実行されたのかは問題にすることなく、ストリームから取り込まれるデータとその到着タイミングの変更によってプロセスを制御すること

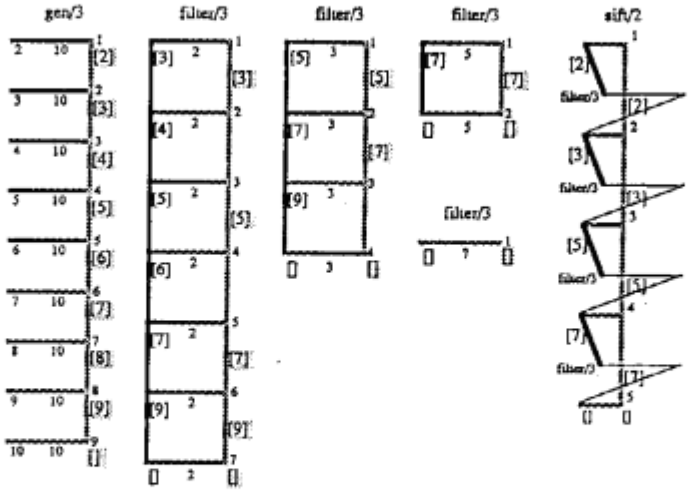


図 6: プロセスチャートの表示例

- 全実行状況を概観するために、プロセスとストリームの接続過程を動画によって可視化すること
- 各プロセス内部の実行過程を概観するために、プロセスの継続ステップ毎に行なわれるデータの入力と出力、内部状態の変更、新しいプロセスの生成を図的に可視化すること

を行なった．これによって、従来ではプログラム作成時にのみ意識された抽象的なプロセスとストリームの実行イメージを、デバッグ作業においても持ち込むことが可能となった．

また、広義のプロセスを対象とするデバッガ [Tatemura 89] と比較した場合、プロセスの継続が導入されたことで、実質的にトレース表示されるプロセス数が減少したこと、ストリームを他のデータ構造と区別し、ストリームの全データではなく、変化した部分のみがストリームの差分として提示されることの二点によって、無駄なトレース情報の刈り込みが改善された．加えて、バルブによる制御は、プロセス実行の中断再開を、プロセス名の直接指定やリダクション数といった低レベルな制御を越えてより抽象的なレベルでサポート可能である．

本デバッガを試作し実験を行なった結果、いくつかの問題点が明らかになった．これらの問題点とその解決あるいは改善の方針を述べて結びとする．

- 5 章の例題において、filter,gen といった他のプロセスを生成することなく、ストリームを生産あるいは消費するために再帰するような単純な構造を持つプロセスに比べ、動的にプロセスを生産してゆく sift プロセスでは、生産されたプロセスの詳細な動作内容は明らかではなく、各プロセスのチャートを参照する必要がある．そこで、全体的なプロセスのチャートを得る工夫として、三次元的に各チャートを配置して、ビューポイントを変えることで局所的なイメージから全体的なイメージまで様々に得ることができると思われる．これを sift に対して適用した例を図 7に示す．
- 新規に生成されたプロセスを全て表示する現在の方式では、全プロセス数の増加とともに実行状況の把握が困難になっていくことが予想される．また、メモリの問題もある．各プロセスに自身の履歴を管理することはメモリを急速に浪費する．これは、プロセス間に適当なスコープルールを導入し、現在の視点から見えるプロセスの数を制限することにより改善される．このようなスコープルールとして、最も単純なものはゴールの親子関係に注目するものであるが、それ以外にもプログラムが扱う問題の性質に応じた規則が考えられる．これは、プロセス指向を越えたいわば問題指向の抽象化であり今後の大きな課題である．

謝辞

本論文の構成にあたって御助言をいただいた富士通（株）国際情報社会科学研究所第二研究部第二研究室の皆様に感謝致します．なかでも、神田陽治氏、村上昌己氏、田中二郎氏には、活発な討議をいただき感謝致します．なお、本研究の一部は、第 5 世代コンピュータプロジェクトの一環として行なわれたものです．

参考文献

[Hirano 89] 平野喜芳、中越靖行、西崎慎一郎、宮崎芳枝、宮崎敏彦、近山隆: 汎用計算機上の KL1 処理系 - PDSS PROCEEDING OF THE LOGIC PROGRAMMING CONFERENCE '89、pp.193-202 (1985).

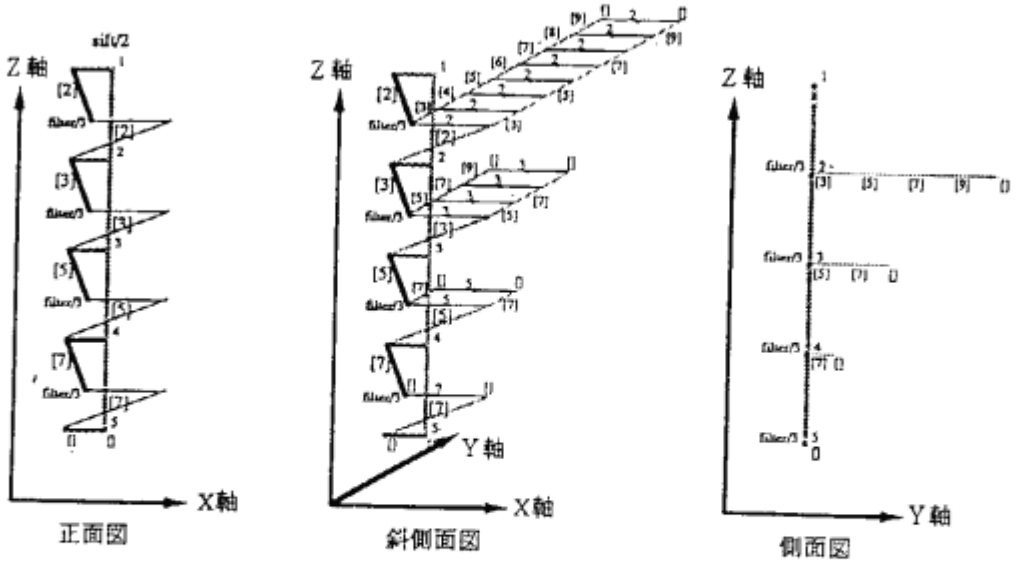


図 7: プロセスチャートの立体的な表示例

[Kahn 77] G.Kahn, D.B.MacQueen: Coroutines and Networks of Parallel Processes, Information Processing 77, North-Holland, pp.993-998 (1977).

[Lloyd 86] J.W.Lloyd, A.Takeuchi: A Framework of Debugging GHC, ICOT Technical Report TR-186, Institute for New Generation Computer Technology (1986).

[Maeda 89] 前田宗則、魚井宏高、都倉信樹: GHC のデバッガに関する考察、情報処理学会ソフトウェア基礎論研究会資料 89-SF-28 (1989 年 3 月 10 日).

[PIMOS 89] ICOT PIMOS 開発グループ: PIMOS マニュアル (第一版)、(1989 年 7 月 19 日).

[Shapiro 83-1] E.Shapiro: Algorithmic Program Debugging, The MIT Press (1983).

[Shapiro 83-2] E.Shapiro, A.Takeuchi: Object Oriented Programming in Concurrent Prolog, New Generation Computing, Vol.1, No.1, pp.25-48 (1983).

[Takeuchi 86] A.Takeuchi: Algorithmic Debugging of GHC programs and its Implementation in GHC, ICOT Technical Report TR-185, Institute for New Generation Computer Technology (1986).

[Takeuchi 87] 竹内彰一: GHC のプログラミング環境、淵 監修、古川、溝口 共編、並列論理型言語 GHC とその応用、共立出版株式会社、知識情報処理シリーズ 6、pp.191-215 (1987 年 9 月).

[Tatemura 89] 舘村淳一、田中英彦: 並列論理型言語 FLENG のデバッガ、PROCEEDING OF THE LOGIC PROGRAMMING CONFERENCE '89、pp.133-142 (1989).

[Tribble 87] E.D.Tribble, M.S.Miller, K.Kahn, D.G.Bobrow, C.Abbot and E.Shapiro :Channels: A Generalization of Streams, Proceedings of the Fourth International Conference Volume 2, pp.839-857 (1983).

[Ueda 85] K.Ueda: Guarded Horn Clauses, ICOT Technical Report TR-103, pp.1-12 (June 1985).

[Ueda 86] K.Ueda: Guarded Horn Clauses, Doctoral thesis, Information Engineering Course, Faculty of Engineering, Univ. of Tokyo (1986).

[Ueda 87] 上田和紀: Prolog 上の GHC 処理系の作成技法、淵 監修、古川、溝口 共編, 並列論理型言語 GHC とその応用、共立出版株式会社、知識情報処理シリーズ 6、pp. 218-238(1987 年 9 月).