ICOT Technical Memorandom: TM-0991

TM-0991

プロセス指向プログラムのための GHCデバッガ

前田 宗則（富士通）、魚井 宏高、
都倉 信樹（大阪大学）

November, 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

プロセス指向プログラムのためのGHCデバッガ

Debugger for Process Oriented Programs in GHC

前田宗則

Munenori MAEDA

富士通（株）国際情報社会科学研究所

IIAS-SIS 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

しばしば用いられるGHCプログラミング技法に，プロセスとストリーム計算を基礎とするものがある．本論文では，この技法を用いた典型的なプログラムを対象とするデバッグ手法とそのデバッガを提案する．従来のGHCデバッガをこういったプログラムに適用した場合，仮想的なプロセス，ストリームとGHCのプリミティブのレベルに大きなギャップが存在し実行状況の把握が困難であった．そこで本デバッガでは，プログラム実行によって変化するプロセスとストリームからなるネットワークグラフと，各プロセスの入出力履歴をグラフィカルに示すことでプログラマのイメージに沿った視覚化を行なう．さらに，プロセス処理の中断と再開，ストリーム上に存在するデータの検査，変更，挿入，保存といった操作を利用者に提供することによりプログラム全体のテストからプロセス単体のテストまでをより抽象度の高いレベルで支援する．

1　まえがき

これまでに提案されている並列論理型言語GHC(Guarded Horn Clauses)[10]プログラムのデバッグ手法は，プログラムの論理的な側面に注目したアルゴリズミックデバッグ法[7]と実行経過のヒストリを実時間あるいは実行後に提示する実行トレース手法[1]に大別される．本論文では，GHCプログラムに対する後者の手法に属する新しいデバッグ手法を提案する．

GHCでは，オブジェクト指向プログラミング[6]やストリーム計算[2]のモデルを基礎としたプログラミングがしばしば行なわれる．こうしたプログラムでは，各機能モジュールがプロセスとストリームによって構成される．プロセス[1]とは並行処理を行なう主体であり，ストリームとはプロセス間を接続するFIFO通信路である．各プロセスは，ストリーム上のデータを読み込み，それによって内部状態を更新し，必要があれば他のプロセスに繋がるストリーム上にデータを書き込む動作を繰り返す．

さて，GHCで記述されたこうしたプロセスとストリームは特徴ある制御構造とデータ構造を持つ．従来のGHCデバッガは，それらを表現するプリミティブを直接に提示するので，プログラム作成時のストリームに繋がれた複数のプロセスによる実行といったイメージと，デバッグ時に提示されるプリミティブには大きなギャップが存在していた．そこで，デバッガに特定の制御構造やデータ構造を抽象化して，プロセスとストリームを陽に扱う機能を持たせるならば，各プロセスが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ったプロセス間の因果関係を中心とした視点からデバッグ作業を支援することが可能になると期待される．

2 プロセス指向プログラムとデバッギング

2.1 GHCによるプロセスとストリームの表現

GHCにおける計算のプリミティブは，手続き呼び出しにあたるゴールと単一化操作である．ゴールはすべて並列に動作し，変数への読み書きは単一化操作によって行なわれる．本節では，仮想的なストリーム，プロセスが，GHCプログラムでどのように表現されるかについて述べる．

2.1.1 ストリームの表現

ストリームとは，データの並びである．ストリームに関する操作としては，先頭のデータを順次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は文献[9]に譲るとして，ここでは，GHCプログラムにおけるストリームの表現とそれに関する操作を述べた上で，ストリーム上の同値関係を定義する．

ストリーム上のデータの並びは，リスト構造体の連鎖で表現される．このためのストリームの簡便な表記をリストの表記法にならって導入する．これを表2.1.1に示す．

ストリームの生成 ストリームは，ゴールの引数に割り当てられる．異なる引数に割り当てられたストリームは，異なるストリームを表す．ストリームは，ゴールの起動とともに新たに生成される．

ストリームデータの読み書き ストリーム (SH‖ST) に対して，リスト [H | T] を単一化するとき，ストリーム上の先頭要素SHが読み込まれる．また，ストリームを表す未定の変数SVに対して，リスト [H | T] を単一化するとき，ストリーム上に要素Hが書き込まれる．書き込み操作後，SVは，ストリーム (H‖ST) なる構造体として参照される．さらに，ストリームへのデータの読み書き操作は，末尾ストリームSTと末尾リストTについて再帰的に行なわれる．

[1]GHCでは，ゴールをプロセスと解釈する立場[8]もあるが，本論文では，CSPやUNIXでいうところのまとまった計算単位をプロセスと見なす．

表 1: リスト表記とストリーム表記の対応

構成要素	リスト $\mathcal{L}$	ストリーム $\mathcal{S}$
未定変数	V	SV
先頭の要素と残り	[H | T]	⟨SH‖ST⟩
空リスト	[]	⟨⟩

ストリームの接続 ストリーム S_1, S_2 が単一化されるとき，二つのストリーム S_1, S_2 は接続されたという．接続された二つストリームは同一のストリームとして参照される．

以上のように表現されたストリームに対して，ストリーム同値関係 $\cong$ を定義する．ストリームの同値関係は，4 節においてストリームの視覚化に利用される．

プログラム実行によって得られた変数，リスト，空リストから成るストリームの集合 $\mathcal{S}$ と，変数への置換集合 σ のもとで，次のような関係 $\simeq_\sigma$ を定義する．

1. $S \simeq_\sigma S$; $^\forall S \in \mathcal{S}$
2. $\langle H \| S \rangle \simeq_\sigma S$; $^\forall S \in \mathcal{S}$
3. $S_1\sigma = S_2\sigma \Rightarrow S_1 \simeq_\sigma S_2$; $^\forall S_1, S_2 \in \mathcal{S}$

2. は，ストリームデータの読み書き操作を行なった時の残りのストリームが元のストリームと同値類に含まれることを表す．また，3. は，接続が行なわれた二本のストリームは，同値類に含まれることを表す．さて，関係 $\simeq_\sigma$ に対して，対称推移閉包をとって関係 $\cong_\sigma$ を定義する．関係 $\cong_\sigma$ が同値関係を与えることは容易に示せる．今後，関係 $\cong_\sigma$ を単に $\cong$ と書く．

2.1.2 プロセスの表現

本論文では，文献 [6] におけるオブジェクトの概念に対応するものをプロセスと呼ぶ．

プロセスの構成 プロセスは，プロセスを表すゴールとプロセス内部で用いられる補助的なゴールから構成される．プロセスは，内部状態と入力ポートもしくは出力ポートを持つ．これらはゴールの引数に割り当てられる．プロセス間通信は，ストリームを用いて行なわれる．

プロセスの生成 プロセスは，プロセスを表すゴールを呼び出すことによって生成される．

プロセスの処理 プロセスは，入力ストリームの先頭からデータを読み出し，それらのデータと内部状態値から出力データと新しい内部状態値を，補助手続きにより計算する．出力データは，出力ストリームに書き込まれる．

プロセスの継続と終了 更新された内部状態値とストリームの残りを引数に設定して，そのプロセスの継続を表すゴールを呼び出すことで処理を続行する．また，継続を表すゴールがなければプロセスは実行を終了する．

2.2 プロセス指向デバッギング

前節のプロセスモデルに基づくGHCプログラムをプロセス指向プログラムと呼ぶことにする．また，プロセス指向プログラムに特化されたデバッガをプロセス指向デバッガと呼ぶことにする．本節では，従来のGHCデバッガをプロセス指向プログラムに適用した場合に生じる実行トレース表示における問題点を列挙し，プロセス指向デバッガが備えるべき表示機能について考察する．次に，プロセス表示機能を用いたデバッグ手法を述べ，それを支援する実行制御機能について考察を進める．

従来のGHCデバッガの実行トレース表示は，次の二つの問題点によりプロセス指向プログラムの実行過程を提示するに十分ではなかった．

1. 実行トレースを整理してプロセス毎に表示する機能がない．

2. 相互に通信路で接続しているプロセスのネットワークを提示する機能がないので，プロセス相互の入出力依存関係が明確ではない．

プロセス継続までの実行を1実行ステップと呼ぶことにするとき，プロセスの振る舞いは，ある1実行ステップでストリーム上のどのデータを読み書きしたかということによって明らかにされる．1.は，得られた全実行トレース列から特定プロセスの各実行ステップにおける入出力データ列を決定する作業が利用者任せになってしまう点が問題である．

また，2.の問題点は通信路の接続誤りにおいて顕在化する．プロセス指向プログラムにおいては，しばしば，通信路の接続誤りからデッドロックエラーを経験する．このとき，従来のデバッガは，エラー報告と共に原因となった呼び出し手続きを提示するが，この情報からただちにどのプロセス間で接続誤りが存在するのかを判断することは容易ではない．

これらの問題点を踏まえ，プロセス指向デバッガはトレース表示機能に関して，

1. 各プロセス毎に，1実行ステップ単位で補助手続きの実行トレース，入出力データ，内部状態の履歴を管理し，整理して提示すること

2. プロセスとストリームからなるネットワークグラフを視覚化するために，ストリームを直接に管理すること

の機能が求められる．

上で述べた実行トレース表示技法を用いることによって次のような手順でデバッグできよう．

1. 利用者は，初期データのもとでプログラムを実行する．

2. プログラム実行過程において生成した各プロセスについて，内部状態の更新，入出力データの読み書きの履歴を調べる．また，プロセスとストリームによるネットワークグラフが意図したものであるかどうかを調べる．意図に合わない動作はバグによるエラーを表す．

3. エラーが観測された場合，エラーを起こしたプロセスについて詳しく調べる．また，このエラーは誤った入力によって生じていることがありうるので，エラーを起こしたプロセスの入力ポートに接続するプロセスについても調べる．

```
(1)  process gen(state,state,port),sift(port,port),filter(port,state,port).
(2)  prime(Max,Ps):- true |  gen(2,Max,Ns),sift(Ns,Ps).
(3)  gen(N,Max,Ns):- N>=Max | Ns =[].
(4)  gen(N,Max,Ns):- N<Max, N1:=N+1 | Ns=[N|Ns1],@gen(N1,Max,Ns1).
(5)  sift([],Ps):- true | Ps=[].
(6)  sift([P|Fs],Ps):- true | Ps=[P|Ps1],filter(Fs,P,Fs1),@sift(Fs1,Ps1).
(7)  filter([],P,Fs):- true | Fs=[].
(8)  filter([N|Ns],P,Fs):- true | sw(N,P,Fs1,[N|Fs1],Fs), @filter(Ns,P,Fs1).
(9)  sw(N,P,Fs1,Fs2,Fs):- N mod P=:=0 | Fs=Fs1.
(10) sw(N,P,Fs1,Fs2,Fs):- N mod P=\=0 | Fs=Fs2.
```

図 1: デバッグ情報が付加された素数生成プログラム

4. バグを含むと疑われるプロセスの入出力データ系列を書き出しておき，プロセス定義の変更後，単体テストの際にテストデータとして用いる．このときプログラム変更前後の出力データ系列を比較することで意図する振る舞いを行なっているかどうかの検査が容易になる．

特に，4. のステップは重要である．一般に，期待される入力データを与えて行なう各プロセスの単体テストでは，表面化しないエラーが，全プロセスを接続した場合に起こることがある．こういったエラーの原因の一つは，実行前に予期しえなかったデータが与えられることにある．そこで，実際に入力されたデータ系列を保存してテストに用いることには意味がある．ゆえに，こうした入出力データ系列を自動的かつ統一的に保存する機能をデバッガに与える．

また，3. のステップにおいて，エラーが生じたプロセスの以降の実行を中断し，誤ったデータがストリーム上に存在する場合には，他のプロセスが読み込む前に変更したり，また要求されるデータを利用者が新たに挿入することによりなるべく実行を継続することも考えられる．こうした実行継続のメリットは，致命的なエラー以外であればプログラムの全実行に関してテストすることが可能になることである．そこで，プロセス指向デバッガに求められる実行制御機能は以下のようになろう．

1. 各ストリームに対して，データのバッファリング及び変更の支援

2. 各プロセスの実行中断・再開・放棄

3 プロセス指向デバッガの実現

図 1は，素数の生成プログラムとして知られているものにデバッグ情報を付加したプログラムである．各行先頭の (1) ～ (10) は説明のために与えられた番号であり，プログラム及びデバッガの動作には無関係である．3.1 節では，本例題に付加されたデバッグ情報について述べる．また，5 節では，本例題によるデバッガの実行例を示す．

3.1 プロセス宣言

プロセスのモデルでは，プロセスを表す述語と内部で補助的に用いられる述語を区別している．しかしながら GHC では構文的にそれらを区別することができないので，利用者によるプロセス宣言が必要となる．プロセス宣言は，引数モード指定，継続ゴール指定から成る．引数モード指定は，キーワード process の後にプロセスを表す述語名と，その各引数の位置に合わせて state もしくは port というキーワードを与えることである．state はプロセスの内部状態を表す引数を，port はストリームが接続される入出力ポートを表す引数を明示する．継続ゴール指定は，定義節のボディにあるゴールのうちの組込みでない適当な述語を一つ選び，そのゴールの前に @ 記号を付けることで行なう．例題プログラムの (1) において，gen，sift，filter がプロセスとして宣言されている．例

えばgenプロセスは，第一，第二引数が内部状態を表し，第三引数が入出力ポートを表す．(3) ～ (8) は，各プロセスの定義である．定義節 (4),(6),(8) のボディ部に各プロセスの継続を表すゴールが指定されている．定義節 (3),(5),(7) のボディ部には継続ゴールが存在しないので，これらの節が選択された場合はプロセスの実行が終了する．また (2),(9),(10) は，補助的な述語prime,swの定義節である．

3.2 ストリームの認識と管理

2.1.1 節で述べたように，ストリームは，変数，リスト構造，空リストアトムによって実現され，その実体はプログラム中で陽に示されるものではない．本デバッガでは，タグ付きデータ構造を導入することで，実行時に生成されるストリームを認識し管理する．タグ付き構造の詳細は次の通りである．

1. 全てのデータに型のためのタグフィールドを設ける．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defaultを用意する．

2. ストリーム型の変数，リスト構造体，空リストアトムに対しては，ストリーム同値関係制約を格納するタグを設ける．このタグは論理変数で実現する．

ストリームをタグ付きのデータで表現する場合，1. は自明であろう．2. は，先に述べたストリームの同値類を表すのに都合がよい実現技法である．実行時に与えられたストリームを表す2つの項が，単一化操作によって同値関係 $\cong$ で結ばれることは，各々のタグフィールドに格納される変数に単一化操作を施し，同一の論理変数に置換することに対応する．また，ストリームを表す2つの項が同値関係で結ばれるか否かは，タグ変数の同一性をチェックすることで容易に調べられる．

デバッガは，実行時に出現するストリームの認識と管理を次のように行なう．まず，ゴールリダクションの過程において，プロセス宣言された名前を持つゴールが与えられたときport指定されている実引数を取り出す．この実引数は，利用者によって陽にストリームであると指定された項である．このとき，デバッガは，新しいストリーム変数のための領域を確保し，その変数の型を表すタグフィールドにstreamを初期設定する．そのうえでこの新しく確保された変数と本来の実引数を単一化する．このとき用いられる単一化手続きは，タグ付きデータ構造を用いて，2.1 節で述べたストリームの読み，書き，接続操作を行なえるように拡張されている．

また，生成されたストリーム変数は，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．プロセステーブルは，4 節で述べられる実行の視覚化手続きにおいて利用される．

以上によって，プログラム実行の過程で得られた項がストリームを表現する項であるか否かを区別すること，及び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ことが可能になっている．

3.3 プログラム実行の制御

本デバッガでは，2.2 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新たに導入することで実現している．既存のプロセスとストリームの間に挿入されたバルブは，データの入力ポート，出力ポート，利用者から与えられる制御命令のための入力ポート，読み込んだデータを保存するバッファと

バルブ制御情報の記述を持つ．ここでバルブ制御情報の記述は，予め設定されたバッファのサイズ，その時点までに書き込まれたデータの数，バッファに格納可能なデータの条件の記述からなる．

バルブは，制御命令の実行やデータ条件の評価によって，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し，各モードにおいて次のような処理を行なう．

- 自動データ入出力モード

  バルブは，データを１つ読み込み，バッファに格納する．バッファが一杯であれば，先頭データを出力する．

- 条件付きデータ入出力モード

  バルブは，データを１つ読み込み，バッファに格納する．バッファが一杯になったか，データ格納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，バルブは新たにデータを読み込まない．利用者は，バッファのサイズ，データ格納条件，既に格納されたデータを参照し，テキストエディタによって変更することができる．編集終了後，バッファの全データを出力し，本モード遷移前のモードへ再開する．

利用者は，データ格納条件の評価に用いられる１引数述語を GHC プログラムで定義する．述語名は任意でよい．ここでは，非終端記号 TestP で表すことにする．データ格納条件は，予め利用者により設定された名前 TestP と入力データ D により，TestP(D) をメタインタプリタで実行することで評価される．

4　実行の可視化手法

4.1　実行履歴とストリームチャート

本デバッガは，2.2 節で述べたプロセスの実行履歴のうち，補助手続きの実行トレースを表示しない．これは，補助手続きの振る舞いを隠蔽することで利用者に提示する情報を少なくし，プロセスの振る舞いを明確にすることを可能とする．

プロセスの実行履歴は，各実行ステップにおける，

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集合
3. 内部状態を表すデータ

の三つである．2. と 3. の履歴情報は，プロセステーブルに保存された実行トレースから容易に得られる．また，1. は，プロセスの１実行ステップにおいて観測された，$X \cong Y$ を満たす二つのストリーム X, Y について，ストリームの差分を求めることで与えられる．ここで，二本のストリームについて，最後尾の変数もしくはアトムから順次取り出して，等価でないデータが見つかるまで比較したときのストリームの先頭からその位置までのデータ列をストリームの差分という．具体例を挙げる．あるプロセスの１実行ステップにおいて，観測されたストリームの内容が，[a,b | X] から [b | X] に更新された時，$\langle a, b \| X \rangle \cong \langle b \| X \rangle$ であり，データ [b | X] を共有している．よって，これらのストリームの差分は，それぞれ $\langle a \rangle, \langle\rangle$ である．

これらの履歴情報を用いて，以下のような作画規則に従い描画されるグラフをストリームチャートと呼ぶ．

1. 各実行ステップは，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. 生成された子プロセスを表す線分は，実行ステップを表す線分の間に配置する．

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．

4.2 実行状況とプロセスグラフ

実行時における，プロセスとストリームからなるネットワークグラフの変遷は，プロセスグラフと呼ばれる動画によって提示される．プロセス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. 二つのストリームの接続が行なわれたときに，各々が繋がるプロセスのポート同士を直線で接続する．

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．

5 デバッガの実行例

本デバッガは，上田のコンパイル方式 [11] 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おり，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いる．利用者は，各制御コマンドに対応するアイコンを選択することにより，プロセスの実行停止と再開，バルブの設定と制御，ストリームチャートの表示，プロセスグラフ上からのプロセス図形の消去を行なう．

ここでは，3 節で与えた素数生成プログラムについて実行をトレースしてみる．まずデバッガを起動し，ゴール prime(10,Ps) を与える．実行開始直後のプロセスグラフは図 2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中断する．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 3に示す．

データ送信後，中断していた実行は自動的に再開して，最終的には図 4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．この時点での各プロセスのストリームチャートを図 5に示す．このストリームチャートより，

- gen は，1実行ステップで第三引数に割り当てられたストリームにデータを一つ書き込む．全実行を通して一本のストリームが維持される．

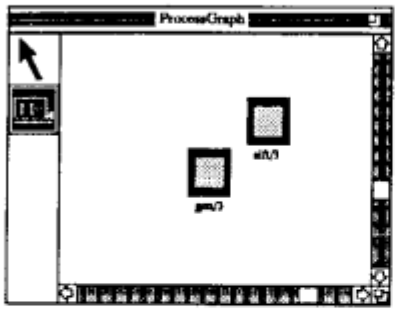

図 2: プロセスグラフの初期状態

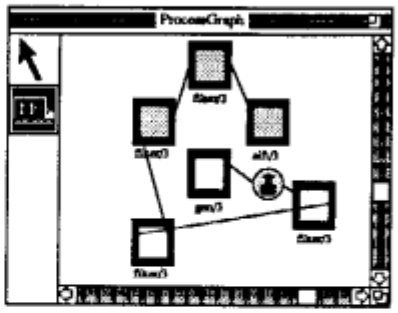

図 4: プロセスグラフの最終状態

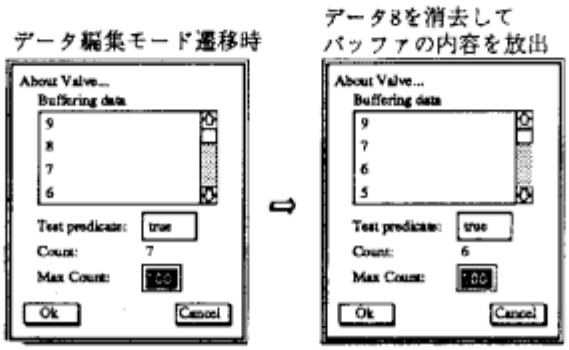

図 3: バルブによるバッファの変更

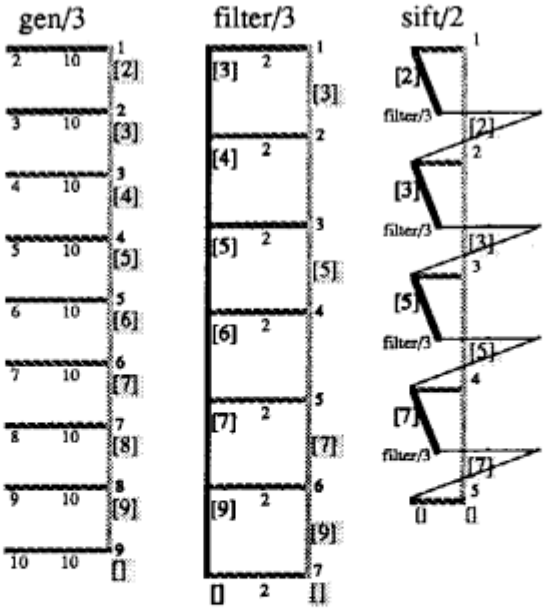

図 5: ストリームチャートの表示例

- filterは，1実行ステップで第一引数に割り当てられたストリームからデータを一つ読み出し，第三引数に割り当てられたストリームにデータを一つ書き込むか，もしくは何もしない場合がある．全実行を通して二本のストリームが維持される．

- siftは，1実行ステップで第一引数に割り当てられたストリームからデータを一つ読み込み，第二引数に割り当てられたストリームに等価なデータを書き込む．さらに，filterを生成し，入力ストリームの残りをfilterの第一引数に，またfilterの第二引数と次の実行ステップで第一引数のストリームと接続する．

ことが確認される．

6 議論

本論文で提案されたプロセス指向デバッガは，プロセスのモデルを基礎とすることにより，従来のゴールトレースを中心とするデバッガよりもより抽象度の高い実行トレースの獲得が可能となった．しかしながら，抽象度を高めることは実行の細部の情報を隠蔽することで行なわれる点に注意する必要がある．本デバッグ手法は，ゴール単位の実行に無関心であるために，文献 [8] で議論されているゴールの入出力代入やそのタイミングについての情報を与えない．そこで，利用者にとって真に使いやすいデバッグシステムとは，プロセス指向デバッガと他の従来型デバッガの併用システムであると思われる．併用システムでは，抽象度の低いレベルから高いレベルまで，様々な実行履歴のトレースと実行制御が可能になる．

こうした抽象度の問題以外に，本デバッガの評価から以下の問題点が明らかになっている．

- ストリーム上のデータがさらにストリームであるような再帰的な技法が，多入力マージャ [5] やオブジェクト指向プログラミング [6] においてたびたび用いられる．このような“ストリームのストリーム”に対して，入出力の制御や表示が考慮されていない．

- 新規に生成されたプロセスを全て表示するため，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る．

この二点の改善案として，プロセス間に適当なスコープルールを導入して，ビューポイントから観測されるプロセスの数を制限することを現在検討している．このようなスコープルールのうち単純なものはプロセスの親子関係に基づくものである．具体的には，ビューポイントから観測されるプロセスとストリームの接続は，現在の方式をそのまま継承し，スコープで隠蔽されるプロセスと一部を除くストリームに関しては，制御，描画を行なわないようにする．また，隠蔽されたプロセス内から，ビュー内にある他のプロセスに接続されるストリームは，制御，描画の対象とする．このように，プロセス間にモジュラリティを導入することでプロセス指向デバッガの適用範囲が拡大される．

7 むすび

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と，それを用いたデバッグ手法を提案した．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い，ストリームから読み込まれるデータとその到着タイミングを変更することによってプロセスを制御すること

- 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出力，内部状態の変更，新しいプロセスの生成を図示すること

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．加えて，バルブによる制御は，プロセス実行の中断再開放棄をゴール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度の高いものとなっている．また，バルブを用いることで単体テストのためのデータの確保も容易になる．

今後の課題は，6 節で述べたスコープルールの導入によるプロセスのグループ化である．基本的な枠組みは決定しており，現在試作を進めている．

謝辞

日頃より討論の相手を引き受けて下さる村上昌己研究員，田中二郎研究員をはじめとする富士通（株）国際情報社会科学研究所の皆様に感謝いたします．なかでも神田陽治研究員には活発なご討議をいただき深謝致します．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

[1] 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 -”, Proc. of the Logic Programming Conference(LPC)'89, pp.193-202 (1989-07).

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

[3] 前田宗則, 魚井宏高, 都倉信樹:“GHC のデバッガに関する考察”, 情処学ソフトウェア基礎論研報 89-SF-28 (1989-03).

[4] 前田宗則, 魚井宏高, 都倉信樹:“GHC 上のプロセス指向デバッガ”, Proc. of LPC'90, pp.169-178 (1990-07).

[5] ICOT PIMOS 開発グループ:“PIMOS マニュアル（第一版）”, (1989-07).

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

[7] A. Takeuchi:“Algorithmic Debugging of GHC programs and its Implementation in GHC”, ICOT Tech. Rep. TR-185, Institute for New Generation Computer Technology (1986).

[8] 舘村淳一, 田中英彦:“並列論理型言語 FLENG のデバッガ”, Proc. of LPC'89, pp.133-142 (1989).

[9] E. D. Tribble, M. S. Miller, K. Kahn,
D. G. Bobrow, C. Abbot and E. Shapiro: "Channels: A Generalization of Streams", Proc. of 4th International Conference of Logic Programming(ICLP)'87 Vol.2, pp.839-857 (1987).

[10] K. Ueda:"Guarded Horn Clauses", ICOT Tech. Rep. TR-103, pp.1-12 (1985-06).

[11] 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(1987-09).