

GHCのデバッグに関する考察

A Debugging Method for GHC Programs

前田宗則

魚井宏高

都倉信樹

Munenori MAEDA Hiroataka UOI Nobuki TOKURA

大阪大学基礎工学部

Osaka University

あらまし 一般にGHCプログラムのデバッグは、逐次型言語のプログラムのデバッグに比べて非常に困難である。これは、実行が並列に行われるため、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ためである。本論文では、この点を改善する新しいデバッグ手法として、パラダイム指向デバッグング(Paradigm Oriented Debugging: POD)を提案した。PODとは、プログラムがあるプログラミングパラダイムに従って記述されたとき、プログラムの各部がそのパラダイムを反映したスタイルで記述されることを用いて、実行トレースを整理しパラダイムのイメージに沿った表示と実行制御によりデバッグする手法である。さらに、プロセスとストリーム並列処理のパラダイムを対象とするPODを取り上げ、プログラムとプロセスの対応、プロセスの実行制御の方法、さらにプロセスの実行状況の可視化について考察を行った。

ABSTRACT GHC(Guarded Horn Clauses) is a parallel language based on Horn Clause logic. Generally it is difficult to debug GHC programs comparing with one written by another sequential logic programming language, e.g. Prolog, because parallel execution of a program makes the execution traces more complex. To improve debugging environment, we propose a new debugging method named "Paradigm Oriented Debugging". Each GHC program is written under some programming paradigm, and it has specific style reflecting the paradigm. The Paradigm Oriented Debugger knows about the paradigm of the target program, so it can arrange the execution traces for "virtual execution images" which are easy to understand for the user, and the user can control the execution under the images.

1 まえがき

並列論理型言語Guarded Horn Clauses (GHC) [10]は、新世代コンピュータ開発機構(ICOT)において、第五世代コンピュータの核言語としてPrologをもとに開発された言語である。GHCは誕生して間もない言語であるので、種々のアプリケーションを実際に記述してその言語の有効性を評価検討することが行われている。また、GHCを効率よく実行する並列マシン及びその処理系の研究が行われている一方、逐次マシンで稼働するGHC処理系がいくつか試作されているがまだ普及するまでに至っていない。さらに、デバッグを含めたプログラム開発環境の整備はこれからの課題である。

一般にGHCプログラムのデバッグは、

(1) 計算がバラレに進行し、かつ各々の計算が他の計算結果に依存することが多いこと

(2) デッドロックが発生すること

(3) 非決定的な計算が含まれる場合があること

などの理由により、逐次型言語のプログラムのデバッグに比べて非常に困難であるとされている。これは、

(a) プログラムが、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こと

(b) 非決定的なプログラムは再現性がないこと

などの問題が生じるからである。したがって、並列プログラムのデバッグを実行トレースに基づいて行う方法は、近い将来深刻な問題に直面するだろうと言われている[8]。

従来の単純な実行トレースに代わる新しいデバッグ法として、Shapiroによりアルゴリズムックデバッグング[6]が提案されている。アルゴリズムックデバッグは、実行した述語のインスタンスを表示し、それがプログラムの意図に一致しているかプログラマに問い合わせ、その返答によってバグの潜むホーン節を絞り込み、最終的に

バグを見つけ出す対話的システムである。アルゴリズムックデバッグングでは、システムから質問される個々の述語のインスタンスの正しさのみを決定すれば良いので、プログラムの詳細な動作を知る必要がなく、プログラムが大規模になってもプログラムの負担は少ないと言われる。このアルゴリズムックデバッグングを並列論理型言語に持ち込んだ研究がいくつか報告されている[7,4]。

GHCにおけるアルゴリズムックデバッグングの問題点は、

- (1) 非決定的な実行におけるバグの再現性を考慮していないこと
- (2) 無限に計算を続けるプログラムのデバッグが困難であること
- (3) 計算の中断(suspension)の取扱いがむずかしいこと
- (4) プログラムを手続き的に解釈するとき、並列プログラミングで重要な手続きの同期や実行順序の情報が、全く得られないことがあげられる。

(1)～(3)の問題点の改善のために、現在もアルゴリズムックデバッグ法を拡張する方向で研究が続けられている[9]。

このうち(4)の問題点に関しては、アルゴリズムックデバッグの対象外として研究されていないが、プログラマにとって、この問題は重要であると考えられる。

本来デバッグとは、対象とするプログラムに大きく依存するものである。GHCプログラムは、既存のいくつかのプログラミングパラダイムにそって記述されることが多い。あるプログラミングパラダイムにそって記述されたプログラムは、パラダイムを反映した特徴のあるプログラミングスタイルを持つ。そこで、パラダイムを強く意識したプログラムにおいては、パラダイムの情報をデバッグに与えることにより、パラダイムのイメージにそった表示と実行制御が可能となると思われる。本稿では、このようなデバッグ手法をパラダイム指向デバッグング(Paradigm Oriented Debugging: POD)と呼ぶことにする。

ところで、GHCは、並列モデルとしてHoareの提案したCSP[2]を採用している。CSPでは並列処理の単位としてプロセスというものを考え、複数のプロセスがチャンネルと呼ばれるFIFO通信路(ストリーム:stream)をとおして他のプロセスにデータを送信することによって計算が行われると考える。また、動的なプロセス生成は認めないという立場をとる。しかし、それでは、記述力の点で大きな制限となるので、GHCを含めた並列論理型言語では、積極的に動的なプロセス生成を認める立場をとっている。従って、CSPで書かれた例題をGHCで記述することは容易であり、実際に多くのGHCプログラムが、CSPのプロセスを意識して記述されている。そこで、GHCにおいては、プロセスとストリーム並列処理のパラダイムを対象とするPODを考察することが重要であると思われる。このプロセスを対象とするPODをプロセス指向デバッグと呼ぶことにする。

プロセス指向デバッグでは、プロセスをデバッグの単位とするので、プログラムとプロセスの対応、プロセスの実行制御の方法、さらにプロセスの実行状況の可視化について考察することにする。特にGHCでは、CSPと異なりプロセスの動的な生成を許すので、プロセスを動的に管理することが必要となる。また、プロセスの実行制御を行うには、プロセスがデータの送受信によって状態変更を行うことから、データの入出力を制御できればよい。

本稿では、プロセス指向デバッグの実現にあたり、

- (1) プロセスの定義によるプロセスの認識
- (2) データのプロセスへの入力と出力の制御
- (3) プロセスの非決定性の制御
- (4) 実行状態の表現

を中心に考察する。

2 GHC上の従来のデバッグ手法

現在、多くのGHC処理系が開発されているが、上田によって開発されたProlog処理系上のGHC[11]を標準GHCと呼ぶことにする。標準GHCでは、ゴールのスケジューリングを、制限付き深さ優先で行っている。あるゴールとそれから導出された全てのゴールに対する制限深さまでの実行を1サイクルと呼ぶ。標準GHCでは、1リダクションと1サイクルが並列実行の単位となっている。

2.1 単純実行トレース法

多くのProlog処理系上のデバッグは、制御の流れをボックスモデル[1]に基づいて実行のトレースを可視化する。GHCを含めた並列論理型言語において、まだ標準とされるようなデバッグが存在せず、Prolog上のデバッグにならった実行トレースを出力するデバッグが用いられている。

標準GHCに装備されているデバッグは、

- (1) 呼び出されたゴールを表示する。
- (2) ゴールが実行に失敗した場合は、その通告を行う。
- (3) サスペンドしている全てのゴールをユーザの指示で表示する。
- (4) 実行を強制停止(abort)したり、ゴールの1リダクション、1サイクル毎に利用者からの入力待ちのため実行を中断する。等の機能を持つ。

単純トレース法の長所は、その実行での詳細な情報が得られることにある。詳細な情報とは、ゴールはどの節を選択したか、どこで実行がサスペンドしたか、実行の結果どのような束縛が変数に与えられたかといったことである。しかし、情報が未整理のまま利用者に提示されるので、そこからデバッグに必要な情報の選択、整理は利用者が行わなければならない。

ところで、従来の逐次型言語のデバッグ作業では、

- (1) プログラムの入力データをいくつか用意し、入力データを変えると実行系列のどこが変化するか
- (2) プログラムの一部を変更し、ある入力データについて、プログラム変更前と変更後で実行系列のどこが変化したかを観察することが中心的作業となる。ここで、実行系列とは、トレースが出力したゴールを時間順に並べたものである。

一般のGHCにおいては、その並列性のためにプログラムの実行がテキストに記述されているような「上から下へ、左から右へ」といった固定された順番に行われないため、テキストとトレース出力を照らし合わせてもどこにバグがあるのかすぐに見つけ出すことが難しい。これは、複数の実行系列が存在するからである。図1は、あるプログラムの2つの実行系列を图示したものである。図の横軸は時間軸であり、縦軸は、複数の手続きを順不同に並べたものである。ここでいう手続きとは、あるまとまった処理をおこなう節の集合のこととする。この例で実行系列1では、実行開始→d→b→a→c→b→d→a→b→d→終了と実行されたとする。x→yは、手続きxの実行から手続きyの実行に移行したことを表す。また、実行系列2では、開始→c→d→b→a→b→d→a→b→終了と実行されたとする。手続きxの実行の流れのみに注目するなら、図で手続きxを横軸に沿って視線を動かし、処理がxに移ってから抜けるまでをつなぎ合わせればよい。

デバッグ作業においては、ある手続きがどのように実行されたかを手続きごとに整理する必要があるが、これは利用者に任される。実際には、複数の実行系列についてトレース情報を整理することは単純な作業ではない。

従って、GHCプログラムを単純トレース手法でデバッグすることとは、Prologプログラムの場合と比較してかなり難しいといえる。

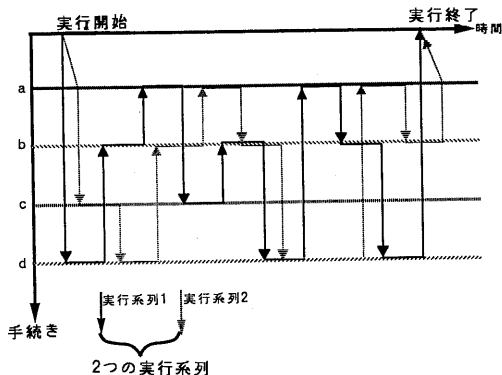


図1 複数の実行系列

2.2 アルゴリズムックデバッグ

アルゴリズムックデバッグは、プログラムの宣言的意味に基づくデバッグ手法としてShapiro[6]によって提案されたものである。このデバッグ手法では、プログラムの実行が大規模になっても、質問の対象となるのはある述語のインスタンスがプログラムの意図を満たしているかどうかという点のみであるということ、およびデバッガから与えられる質問の間の関係について利用者は知らなくてもよいこと、さらに質問の数は実行されたプログラムの計算木の深さに比例したものであり、述語の呼び出し回数の対数のオーダーになるという3点から、単純トレース法に比べ利用者の負担が非常に小さくなっているのが特徴である。

アルゴリズムックデバッグの問題点は、

(1) GHCの意味モデルが、プログラムの意図を表現するのに十分ではないことが報告されている[4,9]。また、ある述語が成功するかどうかは、その述語が有限回の実行で停止して初めて判明するので、ある述語が無限ループとなるならば、アルゴリズムックデバッグではあつかえない。しかし、GHCを含めた並列論理型言語においては、述語が成功して停止しなくてもデータを出力することができるので、積極的に無限ループとなる述語を記述することができる。

(2) プログラムを論理式として解釈するデバッグ手法であるので、プログラムを手続き的解釈による述語間の同期や実行順序に関する情報が得られない。

(3) デバッグの対象となるプログラムが、どのようなパラダイムのもとで何を実現するために記述されたのかということをいっさい対象としないため、たとえば、述語とプロセスの概念を対応づけるとき、複数のプロセスがデータの送受信を行うことにより、プログラムが実行されるという視点を持ち込めない。ことがあげられる。

(1)に関しては、現在もGHCの意味論から研究が行われている。(2)に関してはアルゴリズムックデバッグの範囲外であり、他の方法によって改善する以外ないと思われる。また、(3)は(2)と重複するところもあるが、プログラムが持つプログラムの実行のイメージでのデバッグという作業と、論理式の宣言的意味からのデバッグという作業が必ずしも重なり合うものではないということである。これは、プログラムが大規模になりオブジェクト指向プログラミング等の構造的なプログラミングを行う際に大きな問題点となるとと思われる。

従って、デバッグ作業は、論理式としての側面と手続き的な側面の両方の立場から行う必要があることがわかる。この点からアルゴリズムックデバッグに代わる新しいデバッグ手法が求められる。

次に述べるパラダイム指向デバッグでは、抽象度の高いトレース情報の提示と実行制御が行えるものである。

3 パラダイム指向デバッグ

一般にデバッガの役割には、次の2つがあげられる。

(a) プログラムにプログラムの制御の流れ、変数に格納された値を分かりやすく整理して表現する。

(b) バグがある位置に、できるだけ早くプログラムを誘導する。

前節で述べたアルゴリズムックデバッグは、

(b)において優れた手法であるといえる。(a)は、デバッガ利用者にとって実用上非常に重要であるが、具体的に何をどのように表現するかは各デバッガによって異なっており、デバッガのユーザーインターフェイスの問題として、デバッグ手法という点からは対象外とされてきた。

本稿では、プログラムに依存した情報を与えることで(a)の機能の向上を行える新しいデバッグ手法としてパラダイム指向デバッグを提案する。

プログラミングパラダイムとは何かを明確に述べることは困難であるが、プログラムを行う上での考え方のパターンである[3]と考えられる。これは、

- (1) 計算モデル
- (2) プログラミングテクニック
- (3) プログラミングスタイル

に關した枠組みであるともいえる。

プログラムが、あるパラダイムにそってプログラムを記述するとき、その計算モデルでどのように実行が行われるのかを考慮しながら、制御構造を選び、あるプログラミングスタイルに従って行う。このプログラミングスタイルは、多くの場合特徴あるものとなる。例えばプログラムに段付けを行ったり、特定のデータ構造や制御構造を用いたりする。本稿では、このパラダイムを反映した個別のテクニックや構造をパラダイム要素と呼ぶことにする。

従来のデバッグ作業では、デバッガから出される出力トレースを参照しながらプログラムがその責任において、仮想的な実行を想像しデバッグを進めていた。例えば、並列処理では、プロセス、ストリーム、データフローといった概念があるが、直接にそれらの概念を表現できないGHCでは、それらを述語論理に射影してプログラムすることになる。これらのプログラムのデバッグでは、実行トレースをながめて、述語論理からある概念への逆射影を行うことになる。これを図2に示す。パラダイムがGHCで素直に表現できるものであったとしても、大規模な実行ではデバッガ利用者の負担は大きいものになる。このように、デバッグする際、プログラムの制御の流れやデータの構造を直接に提示することは、利用者の素朴なイメージとかなり異なるため、煩雑でありプログラムの本質を見にくくするだけである。

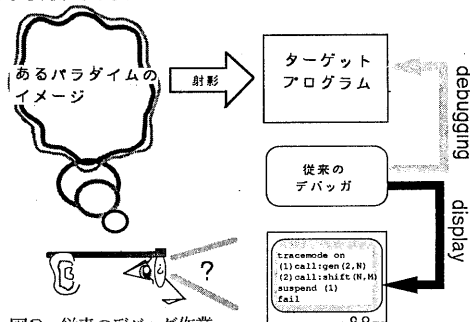


図2 従来のデバッグ作業

これに対し、パラダイムに従ってデバッグ情報を整理して提示することにより、プログラマの負担を小さくできないかというのが、本稿でパラダイム指向デバッグを提案する動機である。またターゲットとしてGHCを選択したのは、GHCが単純な構文と動作の意味を持つこと、初めての実用的並列論理型言語として複数のパラダイムがGHC上でどのように記述するかが研究され、プログラミングスタイルとしていくつか確定していること、及びPrologの流れをくむのでデータとプログラムが同等に扱えデバッグをGHC上に実現することも容易であることがあげられる。

パラダイム指向デバッグは、従来の単純トレース法やアルゴリズムックデバッグのようにどのプログラムにも適用可能な広範囲をカバーするものではなく、あるパラダイムに従った特徴的な部分のデバッグを助けるものであるから、他のデバッグ手法と併用することが大切である。また、このようなデバッグ手法の階層化がプログラマに必要な環境である。図3にパラダイム指向デバッグにおけるデバッグ作業を示す。

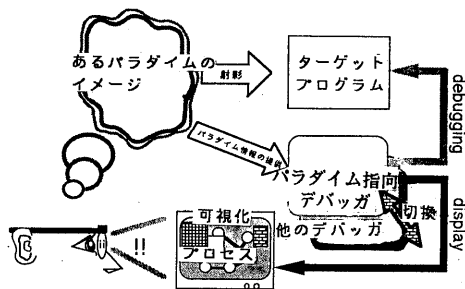


図3 パラダイム指向デバッグ

本稿では、ストリーム並列処理とプロセスに関するパラダイム指向デバッグについて考察する。このプロセス主体のデバッグをプロセス指向デバッグと呼ぶことにする。

プロセス指向デバッグについて考察することは、一般的なパラダイムを対象とするパラダイム可変デバッグの実現において、どのような課題が存在し、それをどのようにして解決するかについての情報を与える。

4 プロセス指向デバッグ

GHCプログラムの例として素数の生成プログラムを取り上げる。

プログラム例 素数の生成プログラム

```
gen(N,Out):- true | N1 := N+1, Out=[N|Out1], gen(N1,Out1).
shift([P|Is],Out):- true | Out=[Is|Out1], filter(Is,P,Is),
                    shift(Is,Out1).
```

```
filter([N|Is],P,Out):- N mod P /= 0 | filter(Is,P,Out).
```

```
filter([N|Is],P,Out):- N mod P == 0 | Out=[N|Out1],
                    filter(Is,P,Out1).
```

```
outterms([X|Is],Out):- true | Out=[write(X),nl|Out1],
                    outterms(Is,Out1).
```

```
:- gen(2,N), shift(N,Ps), outterms(Ps,Os), outstream(Os).
```

このプログラムは、5種類のCSP風のプロセスgen,filter,shift,outterms,outstreamがデータのやりとりをして実行が行われると手続的に解釈することができる。プロセスgenは、他のプロセスとは独立に2以上の整数を生産し、プロセスfilterは、入力データのうちある素数の倍数以外を出力するふるいとして働く。またshiftは、プロセスfilterを動的に生産し、前に作ったfilterの後につなぐことを目的とするプロセスである。outtermsは、素数列を取り込んで出力フォーマットに変換したのち出力プロセスoutstreamに送る。こ

の例では、各プロセスは自主的に実行を進め、入力データが到着するまで実行を中断すること以外に、他のプロセスによって実行が直接干渉されることはない。■

ここで、プログラマが素材にイメージするプロセスとは、次のようなものである。

(1) プロセスは、他のプロセスからデータを入力するための入力ポート、または他のプロセスへデータを出力するための出力ポートを持つ。両方のポートを持つ場合もあるが、ポートを持たないものはプロセスではない。

(2) 複数のプロセスは、ストリームと呼ばれるFIFOの通信路によって接続される。プロセスからみて、入力ポートにつながるストリームを入力ストリーム、また出力ポートにつながるストリームを出力ストリームと呼ぶ。ストリームに対して、プロセスは、入力ストリームの先頭からデータを取り出し、出力ストリームの後尾にデータを送り出すことのみが許される。

(3) プロセスは、データの送受信によってのみ同期をとることができる。

(4) プロセスの内部状態は、他のプロセスが直接参照したり更新したりすることができない。つまりプロセス間に共有メモリはない。このプロセスのイメージを図4に示す。

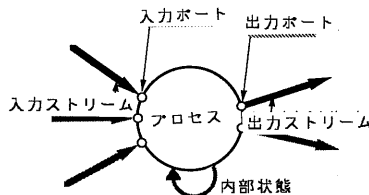


図4 プロセスのイメージ

GHCでは、このプロセスを次のように実現していることが多い。

(a) プロセスは、それ自身を再帰的に呼び出す述語である。多くのGHCプログラムでは、データと制御が左から右に流れるようにゴールを配置し、またプロセスの再帰を表すゴールは、その節の左端におかれる(末尾再帰型)のが普通である。

(b) プロセスのポートは引数に対応し、引数を通してデータを送受信する。

(c) ストリームは、プロセスに共有された変数を末尾に引数であるリストに具体化し、その末尾変数を新たな共有変数として用いることによって実現される。また、プロセスのネットワークは、ある節のボディに並べられた各プロセスを表す述語を共有変数で接続することで行われる。

(d) 入力ストリームからデータを取り出すためには、ストリームを表すリストを分解してデータを取り出す処理を節のガードに記述する。出力ストリームにデータを送り出すためには、出力ストリームを表す変数にデータをセットしたリストを単一化する。各入出力ストリームに継続して接続されるためには、ストリームの末尾変数を各ポート引数に設定し再帰呼び出しする。

(e) 内部状態は引数に割り当て、その内部状態の継続は再帰呼び出しにおいて、特定の引数に新しい内部状態を設定することで行われる。

(f) プロセスの生成は、プロセスを表す述語を呼び出すことによって行われる。■

プロセスとプログラムの対応を図5に示す。

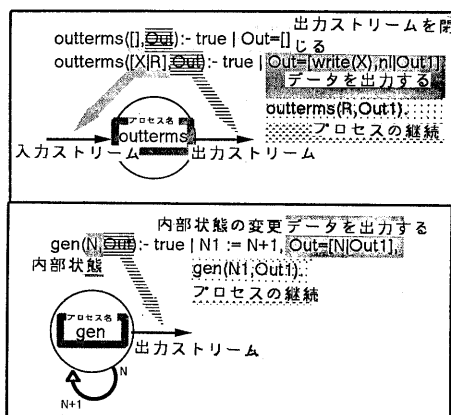


図5 プロセスとプログラムの対応

プロセスに関するバグには次のようなものが考えられる。

- (1) プロセスのネットワークがプログラムの意図と異なる。ここで、プロセスのネットワークとは、プロセスを頂点、ストリームを辺と見立てたグラフのネットワークのことである。
- (2) 入力データがそのプロセスが受理可能なデータではない。あるデータが受理可能であるとは、そのデータを含むデータ列が入力されたとき、どれかの節がコミットされるようなデータ列が存在することである。
- (3) プロセスが出力すべきでないデータを出力した。
- (4) 変更された内部状態の値が意図するものでない。
- (5) プロセス間の同期がとれなかった。
- (6) デッドロックが生じた。
- (7) 入力されたデータからコミットされた節が意図したものではなかった。
- (8) プロセスの内部手続きが失敗したか、停止しなかった。プロセスの内部手続きとは、プロセスを表す節のボディで実行されたゴールのうち、プログラマがプロセスであると意図しないゴールのことである。

(8) のバグは、プロセス内部の実行に深く関わるものでありプロセス指向デバッグでは対象としないことにする。3節でも述べたとおり、デバッグ手法は階層的に用いられることが望ましく、このバグは、他のデバッグ手法（アルゴリズムックデバッグ等）で取り除かれるものとする。

以上のような (1) ~ (7) のバグに対処するため、実行系列及び変数束縛の情報を整理して、以下のようなプロセスに関する情報を利用者に提示する。

- (a) 利用者及びデバッガが動的に与える名前を持つプロセス自体
- (b) プロセスの内部状態
- (c) プロセスのネットワーク図
- (d) 各プロセスが生産及び消費したデータ
- (e) ストリーム上に存在する未処理のデータ

このような実行系列からプロセスに関する情報を取り出し利用者に提示することを受動的なデバッグとすると、能動的なデバッグとして利用者からの入力によってプロセス単位に全体の実行を制御することが考えられる。プロセス単位に実行を制御することによって、システム依存の実行のスケジューリングを変更することと同じ効果を生むことができる。

各プロセスに関する実行制御として、

- (A) プロセス名で指定されるターゲットプロセスを実行する。ターゲットプロセスがすでに生成されており、強制的に実行中断させ

られているものであるならば、内部状態を継続して実行を再開する。また、プロセスが生成されていないならば、指定された内部状態を持つプロセスを新たに生成する。

(B) プロセスのデータ入力 (出力) に次のような条件を導入し、この条件が満たされたときデータ入力 (出力) を強制的に中断する。

(B1) 入力 (出力) されたデータ数が指定された数に達した。

(B2) 指定された形式のデータが入力 (出力) された。

(C) プロセスの持つ内部状態が指定された値をとるまで実行し、その後実行を強制的に中断する。

(D) 節の選択を利用者からの指示に従って決定する。

(B) の実行制御を行うために、バルブという概念を新たに導入する。バルブは、図5のようにプロセスとストリームをつなぐ接点の役割を果たす。プロセスの入力ポートに接続しているバルブを入力バルブ、出力ポートに接続しているバルブを出力バルブと呼ぶことにする。データは、入力ストリームから入力バルブを通り、プロセスに入力される。また、プロセスから出力されたデータは、出力バルブを通り出力ストリームに送り出される。バルブは、開状態と閉状態の2つの状態を持つと定める。バルブが開いているとき、データがバルブを自由に通抜けことができる。また、バルブが閉まっているとき、データはバルブを通ることができない。(B1)、(B2) は入力 (出力) バルブの開閉条件と見なすことができる。また全てのプロセスの全てのポートにバルブを設置することで、全体の実行を細かく制御することができる。

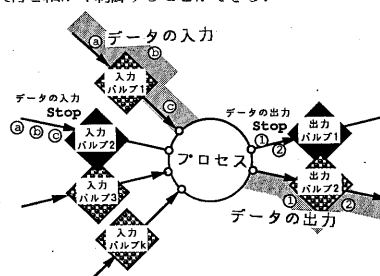


図6 プロセスとバルブ

(D) は、(7) のバグを考慮して導入される実行制御である。非決定性から生じるバグについていろいろな対処の方法が考えられるが、各プロセスごとに利用者が節選択について指示を与えることによって非決定性の管理を行う方法を取り上げる。他に、実行の履歴をとることや実行の全ての可能性を調べるという全探索的なアプローチも考えられるが、いずれも実行時の負担が大きいに加えて、プログラマはGHCの非決定性を知っている上でプログラムしているので、必要最小限の非決定性の管理で十分であると思われる。そこで、非決定性の管理に関しては、実行時にプログラマに節選択を問い合わせるようにする。本稿では、利用者からの節選択の指定をオラクルと呼ぶことにする。各プロセスごとに、オラクルを求めるオラクルON状態であるか、オラクルを求めず節選択はGHCの意味に従い非決定的にオラクルOFF状態であるかを指定する。オラクルON状態では、ガードが成功した節が1つ以上あり、かつ他にガードがサスペンドしているか成功している節が1つ以上存在している場合、ガードが成功しているかサスペンドしているすべての節を節番号と共に表示し、利用者は、次の (a)、(b) の

いずれかを行うことになる。

(a) ガードが成功した節のうちの一つの節番号を入力することで、その節を選択する。ガードがサスペンドした節を選択することはできない。

(b) ガードが成功あるいはサスペンドした節に優先番号付けを行う。実際の節選択はデバッグが行う。デバッグは、優先上位の節のガードが成功しているときは直ちにそれを選択し、またその節のガードがサスペンドしている場合は、その節のガードが失敗しない限りどの節も選択しない。その節のガードが失敗した場合は、次の優先順位の節を調べる。

本稿では、オラクルによる非決定性実行の管理の実現方法については紙面の制約により述べられないので、文献[5]を参照のこと。

4. 1 プロセスの認識方法

GHCにはモジュール化機能がなく、プロセスの内部で実行される手続きとプロセスの定義が混在してプログラムを構成するため、プロセスと内部手続きを構文的、機能的に区別することは困難である。

そこで、プログラムで定義される全てのプロセスは、利用者の責任において宣言することにする。プロセス宣言は、入出力引数の区別にProlog等がよく用いられるモード宣言にならったものである。プロセス宣言:- process <プロセス名> (ポートモード [, ポートモード]). [] 内は0回以上の繰り返しを表す。<プロセス名>は、プロセス定義を行っている述語名である。

ポートモードは、次の3つのうちのいずれかである。

(1) in この引数は、入力ポートである。

(2) out この引数は、出力ポートである。

(3) state ... この引数は、内部状態を表す。

プロセスを表す述語は複数の節から構成されるが、プロセス宣言で与えられたポートモードは、一貫していなければならない。ポートモードが一貫しているというのは、プロセスを表す述語の節AとBにおいて、第N番目の引数をAではinとし、Bではoutとして用いるようなことがないということである。もしそのような必要性があるならば異なる述語名を与えて、異なるプロセスとして宣言する。

どの述語がプロセスを表すかは、プロセス宣言により知ることができるが、述語名は同じでつながるストリームが異なるプロセスは動的に区別する必要がある。動的に生成するプロセスの管理は次のようになる。

(1) プログラムの実行に先だって、プロセス管理表を用意する。

プロセス管理表には、

(1a) 生成しているプロセスを表すゴール。このゴールをプロセスゴールと呼ぶことにする。

(1b) デバッグが実行時に与えるユニークなID番号。

(1c) プロセスを1リダクションした結果生じた、プロセスではないがサスペンドしたゴール。このゴールをサスペンドゴールと呼ぶことにする。

(1d) パルプ指定。

を記入する。

(2) 述語がプロセス宣言された述語ではないならば、それをリダクションする。その後、生成した複数のゴールについて (2) を行う。Pがプロセス宣言された述語であるならば、プロセス管理表にゴールPとID番号を登録する。

(3) プロセス管理表に記入されたプロセスPをリダクションする。このとき得られた複数のゴールについて、ゴールのうちプロセス宣言されていない述語は、サスペンドしてしまうか、プロセスゴールのみになるまでリダクションを続ける。そこで生成したプロセス

ゴールP1,...,Pkは各々プロセス管理表に登録する。生成したプロセスゴールであって、右端のゴールPk+1がPと同じ名前であるならば、このゴールPk+1をPが再帰したものとしてPに代わって登録する。サスペンドゴールはPのサスペンドゴール欄に登録する。

(4) プロセスPが再帰せず、かつサスペンドゴールが生成しなければ、このプロセスは消滅したと考えプロセス登録表から削除する。このとき、プロセス本体は再帰せずにサスペンドゴールが生成した場合、そのサスペンドゴールは、内部手続きのバグである可能性が大きい。そこで、このとき利用者に実行を中断するかどうかを問い合わせ、バグであると判断された場合、プロセス登録表からそのプロセスのレコードを削除しサスペンドゴールのリダクションは行わない。バグでないと判断された場合は、その中に含まれる変数が具体化されるごとにリダクションを行う。そこから生成されるゴールに関しては (3) に従う。■

4. 2 プロセスのデータ入力認識

入力ストリームに流れるデータをプロセスが取り込むとき、プロセスがいくつかのデータを取り込んだのかをどのようにして調べられるかについて考察する。ここで、あるデータをプロセスが取り込むとは、そのデータを含む複数の連続したデータの列Isによって、プロセスを表す述語のある節がコミットされることである。その詳細に対して2つの立場がある。

(a) コミットされた節を構文的に見たとき、節のヘッドで入力リストを分解して、いくつかのデータと末尾変数に分離しているはずである。この入力引数に含まれるデータ数が取り込まれるデータ数である。取り込まれるデータ数は、実行前にプロセスを表す述語の各節ごとに調べておくことができる。

process p(In).

(1) p([a|Is]) :- true | p(Is).

(2) p([c,d,e|Is]) :- true | p(Is).

(1)は、入力ストリームから1個のデータを取り込む。(2)は、3個である。

この場合、再帰するときにデータを入力ストリームに戻すようなプロセスを取り扱うのは難しい。

(3) p([f,g|Is]) :- true | p([g|Is]).

(3)の場合、プロセスは、f,gという2個のデータを取り込んだ後、gを入力ストリームの先頭に戻している。次にプロセスが、gから始まるデータ列を取り込むことを考えれば、(3)の節で実際に取り込まれたのはfのみであると考えられることもできる。(3)は、2つのデータを取り込むのではなく、1データを先読みしていると考えるのが自然である。

(b) コミットした直後に入力ポート引数をプロセス管理表に登録する。次に再帰したとき、プロセスの入力引数のデータ列とプロセス管理表に登録した前の入力データ列の差分をとり、前にコミットしたときのデータ列を決定する。また節のボディで入力ストリームに戻したデータ列を決定する。この立場では、取り込まれるデータは実行時にわかる。今、登録されている入力ポート引数の内容を、[x0,x1,...,xn|X]、再帰したゴールの入力ポート引数の内容を[y0,y1,...,ym|Y]とする。

(b1) XとYが等価な変数である場合。

具体化した末尾から順に、データxiとyjが等価であるかどうか調べる。初めi=n,j=mである。xiとyjが等価であるならばi=i-1,j=j-1として繰り返す。xiとyjが等価でないとき、プロセスが取り込んだ入力データは、x0,...,xiであり、y0,...,yjは、プロセスが入力ストリームに戻したデータである。さらにxi+1,...,xn及びyj+1,...,

ymは、プロセスがまだ読み込んでいない入力ストリーム上のデータであるとする。

例2)

(4) $p([f,g,h,il|Is]) :- \text{true} \mid p([h,g,il|Is])$.

(4)では、取り込まれたデータはf,g,h, 入力ストリームに戻されたデータはh,gと見なされる。

(b2)XとYが等価な変数でない場合。

このとき、このポートに接続する入力ストリームが変更されたと考える。入力ストリームが変更された場合は、読み込まれたデータはないと考えることにする。入力ストリームの変更がバグでないときは、このプロセスが子プロセスQを生成し、入力ストリームからのデータを子プロセスQがフィルタし、それが再帰したプロセスに入力される。素数の生成プログラムにおけるプロセスshiftがそれに当たる。

(a), (b)の2つの立場のどちらを選択するかは、実行時のオーバーヘッドとプロセスのデータ利用の詳しい情報の間のトレードオフにより決まる。以下では、実行時のオーバーヘッドを問題より、よりプロセスの実行状況を詳しく参照できることのほうが重要であるとするので、(b)のデータ入力認識手続きを採用している。

4.3 バルブの実現

バルブの実現方法を、入力バルブと出力バルブの場合に分けて考察する。

(A) 入力バルブの実現

次の2つの方法が考えられる。

(A1)プロセスにデータが読み込まれた後、読み込まれたデータがバルブ条件を満たしているかチェックする。バルブ条件は後述するが、実行時にバルブを閉じるためにあらかじめ利用者が設定する条件のことである。

(A2)プロセスにデータが読み込まれる前に、ストリームの先頭に具体化しているデータがバルブ条件を満たしているかどうかチェックする。

まず、(A1)の方式の入力バルブ実現方法を述べる。

(A1.1)プロセスpが生成したとき、各入力ポート引数に代入されているストリーム変数Isを取り出し、ストリーム変数表に登録する。

(A1.2)Isにデータが具体化するような単一化操作がおこなわれようとするまで何もしない。Isにデータを具体化する単一化は、節のボディで行われ、かつIsとリストが単一化されるものである。

(A1.3)入力バルブが開状態ならば、その単一化操作をそのまま実行する。

入力バルブ閉状態ならば、その単一化操作をサスペンドさせる。このサスペンドした単一化操作は、入力バルブが開状態になるまで続く。

(A1.4)プロセスを1単位実行する。どの節もコミットされなかったら(A1.2)へ。

(A1.5)入力バルブの開閉条件のテストを行う。

開閉条件は、

④データ数dが、上限値d_maxに達したかどうか

⑤入力されたデータdataが、指定した項d_patternであるかどうかの2つである。dは、バルブが開状態になってから、これまでにこのプロセスに取り込まれたデータ数を記録する変数とする。

・d:=d+プロセスに取り込まれたデータ数

・s=プロセスに取り込まれたデータの集合

④については、d>d_maxであるならばバルブを閉じる。

⑤については、sの要素としてd_patternが存在するかを調べる。そのデータが存在するならばバルブを閉じる。

(A1.6) (A1.2)以降をプロセスが消滅するまで繰り返す。■

指定データが存在するかどうかは、dataとd

patternを単一化することでテストする。単一化が失敗したときは、条件が不成立であり何もしない。ここで単一化の方法には、dataに変数が含まれているとき、その変数にd_patternの項が束縛されるのを認めるか否かで2つの方法がある。どちらの単一化を選択するかは、利用方法による。前者は、データに影響を与えるために、プログラム実行を変更する機能を持つ。後者にはそのような機能はない。

(A1)によるバルブ実現の概念を図7に示す。

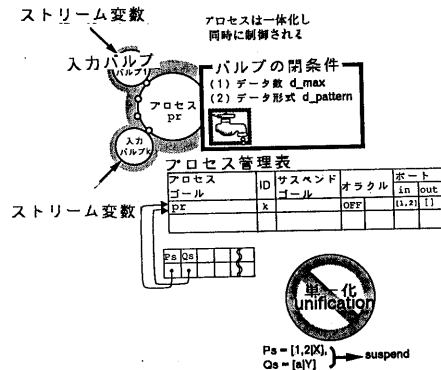


図7 入力バルブの実現法: A1

(A2)の入力バルブ実現法は、入力バルブをプロセスと考慮し、対象とするプロセスpの入力ポートに接続する形で設置することによって実現することに対応する。この実現の概念を図8に示す。

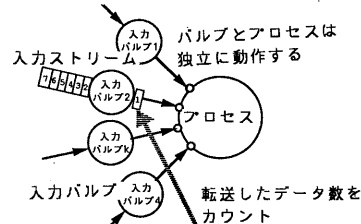


図8 入力バルブの実現法: A2

バルブ手続き(A2)は、(A1.1)~(A1.3)と同じ手順をとるが、

(A1.4)のようなプロセス実行を直接制御することはない。

④条件は、Isに具体化したデータ数dをカウントしてゆく。

d=d_max-nのとき、Is=[x0,...,xn|Is']なる単一化を実行するとd>d_maxとなってしまう。このときは、Is=[x0,...,xi|Temp]を実行し、dをd_maxと等しくする。また、Temp=[xi+1,...,xn|Is']というゴールをサスペンドさせる。このゴールは入力バルブが開状態になったとき再開する。ここで、Tempは、システムが一時的に生成する変数である。d<d_max-nのとき、ゴールIs=[x0,...,xn|Is']をそのまま実行する。

⑤条件は、ゴールIs=[x0,...,xn|Is']の要素、x0,...,xnに指定したデータが存在するかどうかリスト先頭から調べ、もし存在しなければそのまま実行する。また、存在しxiがそうであるならば、Is=[x0,...,xi|Temp]を実行し、ゴールTemp=[xi+1,...,xn|Is']をサスペンドさせる。

(A1)は、プロセスが取り込んだデータについてバルブ条件を検査するため、条件④で指定したデータ数に等しいかより多くのデータを取り込む。どれだけのデータが読み込まれるかは実行時でないとわからない。(A2)はプロセスに取り込まれる前の具体化したデータ

について検査するので、指定したデータ数がプロセスに送信される。しかし、プロセスがデータをストリーム上にことがあるなら、バルブとプロセスをつなぐストリーム上に未処理のデータがバッファリングされたまま残されるということが起こる。

(B) 出力バルブの実現

出力バルブの実現についても2つの方法がある。

(B1) (A2) と基本的に同じである。

(B2) プログラムテクニックとして、バルブを実現する。

ここでは、紙面の都合上 (B2) の実現方法のみを述べる。

出力バルブは、要素が変数であるリストを生成し、それを対象のプロセスPに流す機能を持つプロセスとして実現する。このとき、プロセスPは、変数に他のプロセスQに送るべきデータを具体化する。これは、未完成メッセージの利用によって出力バルブを実現するプログラムテクニックである。図9にその概念図を示す。

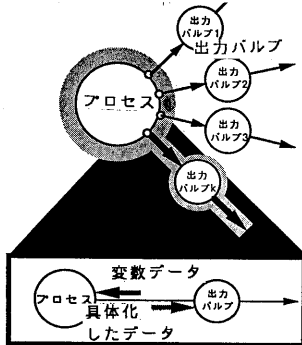


図9 出力バルブの実現法2

より詳細には、プロセスの出力ポートを入力引数として利用するために、プロセスの定義を書き換える。まず例を示す。次の述語genxは、genの出力ポートである第二引数を書き換えて、入力引数として変更したものである。

プロセス定義の変更例

genx(N, [X|Od]) :- true | X=N, N1:=N+1, genx(N1, Od).

出力ポートを入力引数に変更する手順は次のとおり。

- (1) 出力ポート引数で与えられている出力変数Outを少なくとも一つの変数要素Vを持つリスト[V|Out1]で置換する。
- (2) その節のボディにOut=[V|Out1]というゴールを新たに加える。
- (3) (1)で行われた置換は、プロセスを定義する各節において、すべての出力ポート引数を対象に行う。ただし、空リストを具体化する引数についてはこの限りではない。プロセスとしての出力バルブは、o_valve(ToOutputStream, User_Control)として与えられる。

User_Controlは、利用者からの

④バルブを直接開閉する命令

⑤ストリームToOutputStreamを伸ばす、つまり変数要素を生成する命令

⑥バルブ条件設定

を入力するためのストリームである。また、ToOutputStreamは、このバルブを設定するプロセスの出力ストリームとして用いられる。閉状態の出力ポートからデータを生成しようとする全体の実行をサスペンドさせることができるので、元のプログラムに簡単な手直しを行った上で、データ生成、マニュアル開閉といった機能と、出力データ数、特定のデータにより自動的にバルブを閉めるような機能を加えたバルブプロセスを用いることによってデバッグを行うこともできる。しかし、この方法では、テキストプログラムを変更することが必要でその正当性が明確でなければ用いられない。たとえば、データを全く生成することなく出力ストリームを閉じるとき、出力ポート引数を1つ以上の要素をもつリストで書き換えることに

より、矛盾が生まれ実行時に失敗する。それ故データを少なくとも1つ以上生成することが明らかであるプロセスにこの手法を適用するには問題ないが、データを生産するかどうかかわからないプロセスにこの手法を直接に用いることはできない。この問題を解決するために、データを出力することなく出力ストリームを閉じようとする単一化に対して、ダミーデータを1つ挿入する方法も考えられるが、それはGHC内では定義できない述語(メタ述語)を導入することにより実現され、GHCの単純さを損なう点でありよい方法ではない。ゆえに特定のプロセスに対する制御テクニックとしては有効であるが、一般的に用いることは難しい。

5 あとがき

並列論理型言語GHCにおけるデバッグ手法としてパラダイム指向デバッグを提案した。また、多くのGHCプログラムでプロセスによるストリーム並列処理が行われているので、パラダイムとしてプロセス指向を選びそれに従って記述されたプログラムをターゲットとするプロセス指向デバッグを提案した。本稿では、紙面の制約上、可視化の技術については触れられなかったが、詳しくは文献[5]に述べられているので参照して下さい。

本研究で提案した手法と従来のデバッグ手法を併用することにより、プログラムの負担が軽減されると思われる。

今後の課題として、プロセス指向デバッグの有効性を確かめること、パラダイムとしてオブジェクト指向プログラミングを選び、オブジェクト指向デバッグについて考察すること、また、より自由度の大きいパラダイム可変型のデバッグについて考察を深めることが考えられる。

参考文献

- [1] W.F.Clocksins, C.S.Mellish, 中村 訳: "Prologプログラミング", マイクロソフトウェア, pp.197-222 (1983).
- [2] C.A.R.Hoare: "Communicating Sequential Processes", CACM, Vol.21, No.8, pp.666-676 (Aug.1978).
- [3] 井田哲雄: "新しいプログラミングパラダイム: 総論", 共立出版株式会社, bit, Vol.19 No.12, pp.36-42 (1987年11月).
- [4] J.W.Lloyd, A.Takeuchi: "A Framework of Debugging GHC", ICOT Technical Report TR-186, Institute for New Generation Computer Technology (1986).
- [5] 前田宗則: "GHCプログラムのデバッグ手法", 大阪大学大学院基礎工学研究科 修士学位論文 (1989年3月).
- [6] E.Shapiro: "Algorithmic Program Debugging", The MIT Press (1983).
- [7] A.Takeuchi: "Algorithmic Debugging of GHC programs and its Implementation in GHC", ICOT Technical Report TR-185, Institute for New Generation Computer Technology (1986).
- [8] 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.193 (1987年9月).
- [9] 竹内彰一: "GHCプログラムの意味について", 情報処理学会ソフトウェア基礎論研究会資料 86-SF-19 (1986).
- [10] K.Ueda: "Guarded Horn Clauses", ICOT Technical Report TR-103, pp.1-12 (June 1985).
- [11] 上田和昭: "Prolog上のGHC処理系", 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.217-270 (1987年9月).