ICOT Technical Memorandum: TM-0800

TM-0800

色効果を利用したGHCデバッガ

前田宗則(富士通)

September, 1989

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

色効果を利用したGHCデバッガ

A Color based GHC Debugger

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

あらまし 従来の実行トレースのような時系列に沿った情報の表示では，GHCプログラムのデバッグ情報としては不十分である．そこで，本論文では，色の概念を導入することにより，それらの問題点を改善する新しい実行トレーサを提案する．このトレーサは，注目するゴールに選択的に色を付けること，及び変数を通して色を伝播させることが可能である．この実行トレーサを単純なメタインタプリタから段階的に導出し，その実行例を示す．

Abstract Sequential tracing obtained from the ordinary GHC debugger is not sufficient to debug GHC programs. Therefore, we propose a new "Color GHC Tracer" to solve this problem. By using this tracer, it becomes possible to paint a "color" for certain goals. Propagating a color through variable bindings is also possible. We show how this "Color GHC Tracer" is derived from the simple meta-interpreter of GHC. A tracing example is also shown.

1 まえがき

並列論理型言語 GHC(Guarded Horn Clauses) [Ueda 85] は，ホーン節に基づくコミットチョイス型の並列プログラミング言語である．現在，新世代コンピュータ技術開発機構 (ICOT) では，第5世代コンピュータプロジェクトの一環として，GHC[1]，に基づいた核言語 KL1 の処理系と分散並列計算機，及びアプリケーションプログラムの開発を精力的に行っている．

ところで，GHC (KL1) プログラミング環境の中で重要な研究課題となるのが，そのデバッガである．これは，GHC プログラムの実行において，

1. 計算がパラレルに進行し，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したり，非決定的な計算が含まれる場合があること

といった従来のプログラム言語にはない動作が行われるためデバッグすることがより困難であるからである．

現在，提案されているデバッグ手法を大別すると，次の2つに分類される．

- 実行トレース法
- アルゴリズミックデバッグ法

実行トレース法は，ゴールと呼ばれる実行手続きを時系列に沿って表示する．デバッグ作業は，この表示されたゴールからプログラム全体の制御の流れを把握し，異常動作の直接の原因を推測することである．しかし，デバッグに関する情報が未整理のまま提示されるので，そこから，必要な情報の選択，整理は，プログラマが行わなければならない．プログラムの規模が大きくなるほど，出力される情報が増大し，これに伴ってデバッグの手間も確実に大きくなる．

[1]今後，本論文では GHC を Flat GHC の意味で用いる．

また，アルゴリズミックデバッグ法 [Shapiro 83] は，pure Prolog（ホーン節論理）に対するデバッグ手法として提案された．この手法は，デバッガが，プログラム実行の全過程を実行木として保存し，実行終了後に，実行木に現れる各手続きがプログラマの意図に従っているかどうかをプログラマに問い合わせることを行う．このデバッグ法の特徴は，

- デバッガが内蔵するバグ探索のアルゴリズムによるが，一般に実行トレースで無制限に出力したゴール数と比較すると，より少ないステップでバグを持つ節を決定できること

であり，これより，実行トレース法に比べプログラマの労力の負担が小さくなると予想されている．しかし，まだ実行トレース法にとって代わってデバッガの主流には成っていない．これは，

- 実際的な大規模プログラム，例えば Operating System 等において，実行木を生成することは，大量の計算機資源を浪費すること
- GHC のいくつかの応用プログラムでは，利用者が強制的に実行停止するまで，無限ループするものがあり，これらにこの手法は適用することは困難であること

といった問題点を抱えているためである．大規模なプログラムであるほど有利であると予想されながら，実際には利用されていない理由がこれであろう．

以上より，提案されている 2 つのデバッグ手法のいずれも，大規模プログラムのデバッグに向かない．そこで，本論文では，現在の主流となっている実行トレース法に色の概念を導入することで，実用的な大規模プログラムにも適用可能なデバッグ手法を提案する．色を導入する理由としては，従来のデバッグ作業は，

1. 実行する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ゴール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ゴール系列のどこが変化したか

について観察することを主目的としており，色を用いることでこれらの変化を選択的に観察することが可能であると考えたからである．

本論文の構成は次の通りである．まず，第 2 章でプログラムで起こるエラーとその原因となるバグについて，これまでのデバッガの研究に基づいてまとめてみる．次に第 3 章で，色効果とは何かについて具体例を交えて説明する．第 4 章では，色効果を持つ GHC デバッガを段階的に構成し，適用範囲を拡大することを試みる．第 5 章は，本デバッガの実行例を示す．最後に，まとめと今後の方針について述べる．

2 バグの分類と従来のデバッグ手法

まず，GHC プログラムに含まれるバグを分類し，それらのバグに対して，どのようなデバッグ手法とデバッグツールが用いられるかについて，Prolog，GHC のデバッガに関するこれまでの研究 [高橋 85] を踏まえて以下にまとめてみる．

エラーの原因となるバグには，多数の要因が考えられる．このバグをまず大きく 2 つに分類する．一つは，プログラム実行の前にエラーの発生のチェックが行えるもので，これを“静的なバグ”と呼ぶことにする．他方は，実行時までエラー発生が判明しないバグであり，これを“動的なバグ”と呼ぶことにする．

まず，静的なバグは次のように分類できよう．

静的なバグ 1 シンタックスに関するバグ．パーザ，シンタックスチェッカ等が，シンタックスエラーとして検出するシンタックスに関するバグであり，最も容易に訂正可能である．

静的なバグ 2 プログラムの解析で発見可能なバグ．

- 述語記号のスペリングミス，述語の引数の数（arity）の不一致．
- 明らかに失敗するような述語の呼出し．実引数と仮引数のデータ型の不一致．
- 引数の入出力モードの誤り．

1.，2. のバグに対する Prolog 上のデバッグ手続きが，文献 [高橋 85] で提案されている．また，これらに対する利用可能なツールとしては，クロスリファレンサ，型チェッカ，入出力モードチェッカ等がある．

次に，動的なバグであるが，これは以下のように分類できる．

動的なバグ 1 GHC の並列性と非決定性の問題．
GHC の特徴として，

- ゴールの実行順序は実行時まで明らかではないこと
- 複数の節が選択可能性である場合に，それらの一つを非決定的に選択すること

がある．これらの特徴を十分に考慮されていないプログラムは，プログラマの意図に反した動作をする場合がある．

動的なバグ 2 デッドロックの発生．
デッドロックは，以下のような原因で生じる．

- 決して具体化されない変数を参照する手続きの存在．
- どの節の条件部も成功しない手続きの呼び出し．

動的なバグ 3 有限で停止すべき手続きの無限ループ．

動的なバグ 4 既に値の具体化した変数に，異なる値を再度代入した．

動的なバグ 5 組込み述語の誤った使用．

これらの動的なバグに対しては，前章で述べたように実行トレーサとアルゴリズミックデバッガが用いられる．実行トレーサ [Clocksin 83][PIMOS 89] の機能としては，スパイポイントの設定，ステップ実行がある．また，アルゴリズミックデバッガの機能は，利用者への質問とその応答からバグを持つ手続きを絞り込むものである．これを GHC（もしくは，そのサブセット言語）上で実現したものが，文献 [Takeuchi 86][竹内 87][舘村 89] で報告されている．

3 デバッグにおける色効果とは何か

まえがきで述べたように実行トレース法は，ターゲットプログラムの実行制御，実行状況（変数の束縛情報等）の提示を対話的に行うことができる点で自由度が大きい手法である．しかしながら，多くの場合，計算機の出力情報は，無駄が多いか，もしくは足りないことで，デバッグがうまく行えないことを経験する．無駄が多いということは，情報は大量にあるが，知りたい情報がどこにあるのか分からない場合であり，足りないのは，出力するトレース情報を制限しすぎてしまい知りたい情報までも枝刈りしてしまったからである．加えて，第 2 章の動的なバグの項で述べたように，GHC の並列性の問題がある．GHC プログラムを分散並列計算機上で実行する場合，複数のプロセッサが非同期に手続きを実行しているので，時系列で出力されるトレース情報には意味がない．また，単一プロセッサの計算機上においても，Prolog のようにプログラムテキストに記述された順序で実行されるわけではなく，注目する実行系列とそうでない系列が組合わさって表示されるので実行過程を理解することが容易ではない [前田 89]．

そこで，出力トレース情報に色付き重要度をつけて表示することにより，大量の出力の中から常に注目する情報を見つける工夫ができないか．これが本研究の動機である．

研究の指導原理としては，神田の主張する“意味ありげ主義”[Kohda 89] を採用する．意味ありげ主義は，情報をある基準（利用度，優先度等）で分類し，その度合いの大きいものから順に提示することによって，無基準で提示されるよりも早く必要な情報を得ることができるというものである．神田は，意味ありげ主義を実現している具体例として，ColorEmacs[Lippman 85] を取り上げている．ColorEmacs は，文字に色の属性を付加している点で，従来の Emacs テキストエディタと異なり，その文字色に，それが入力されてからの経過時間を反映させることで，

1. どこがいつ変更されたのか
2. 注目する場所はどこか

を周囲の文字色の変化によって提示する．特にここで強調するべきは，2. の特徴であり，テキストに付けられた色の段階的な変化によって，テキスト中のどの部分をどの程度注目しているのかといった情報を表現することが可能になっている点である．

本論文は，ColorEmacs にならい，色の技法をデバッグに利用することによって，従来のトレーサでは明確ではなかった，以下の 3 つのデバッグ情報を効果的にプログラマに提示することが目標である．

1. ある注目するゴールは，どのゴールのリダクション過程で生じたのか.

2. ある注目するゴールは，どのくらいのリダクション数を経過したか.

3. 変数の具体化は，どのゴールのリダクション過程で行われたのか

1. は，同じ名前の述語を複数実行している時に，その先祖にあたるゴールが明らかにすることで，他の同じ名前のゴールと区別する情報を与えるものである．2. は，従来の言語でいうところの何回ループしたか，あるいは，構造化された定義のどのレベルかといった情報を与える．最後の 3. は，データフロー計算を意識した GHC プログラムで重要となる情報である．多くの GHC プログラムが，ゴールをプロセス，共有変数を通信路と見なして，プロセスが他のプロセスと通信を行うことで，計算を進めるというモデルに基づいて記述されている．これらのプログラムのデバッグにおいて重要なことは，変数を共有する 2 つのプロセスのどちらが値を書き込んで，どちらが読み出したのかという，データの流れた方向を明らかにすることである．このとき，プロセスごとに異なる色を着色し，そこで生成されたデータに同じ色を付けられれば，データの流れを理解し易くなると思われる．

これまでに述べたように，デバッグ作業における，色を使用することのメリットは，注目する情報を目立たせるということであり，これは，日頃我々がドキュメントの重要なキーワードに色マーカで印を付けることに対応するものである．

4 色付きデバッガの試作

本章の方針は，まず，4.1 節で，メタインタプリタによるデバッガの記述例を示し，4.2 節で色付きシンボルの表現形式について述べる．次に，4.3 節で，ゴールに色を付けて表示する色付きゴールトレーサの実現について論じる．さらに，4.4 節では，ゴールの色をデータに着色するために単一化を拡張することを行う．最後に 4.5 節において，色付けトレーサに具体的な例題を適用する．

4.1 準備

まず，Prolog のバニラインタプリタに相当する，GHC のメタインタプリタを示す．

```
(1) call((A,B)):- true | call(A), call(B)
(2) call(A):- A \= (_,_), sys(A) |
        sys_exe(A).
(3) call(A):- A \= (_,_), usr(A) |
        clauses(A,B), call(B).
```

(1) 節は，兄弟ゴールの展開を行うものである．(2) 節は，true，単一化操作を含めた組込み述語の処理を行う．組込み述語の処理を実際に行うのは，メタ述語 sys_exe が行う．最後の (3) 節は，メタ述語 clauses が，利用者定義述語 A と単一化可能な節の頭部を持ち，なおかつガードの述語が全て成功する節のうち一つを非決定的に選択し，そのボディゴール B を取り出す操作を行う．その結果，得られた新しいゴール B について再帰的に実行を継続する．

Prolog バニラインタプリタを拡張することにより，デバッガ [Shapiro 83] を構成する試みや，GHC プログラム環境の改善を目的として，上のメタインタプリタを順次拡張する試み [田中 88] [Tanaka 88] が以前から提案されている．本論文も同様にメタインタプリタを拡張することで，色効果をサポート可能なデバッガを構成する．次節で色効果をサポートするデバッガの構成する前に，利用者の指示により実行の中断，再開が行え，かつ出力情報を抽象的な表示装置に送り出すようなデバッガを示す．

```
(1) debug((A,B),I,O):- true |
        debug(A,I,O1),
        debug(B,I,O2),
        merge(O1,O2,O).
(2) debug(A,I,O):- A \= (_,_), sys(A) |
        O = [flush_write(sys(A),X)|O1],
        syncro(X,sys_exec(A,R)),
        syncro(R,
              O1 = [write(exec(A,R))]).
(3) debug(A,I,O):- A \= (_,_), usr(A) |
        clauses(A,B),
        O = [flush_write(usr(A),X)|O1],
        syncro(X,debug(B,I,O1)).
(4) debug(A,[suspend|I],O):- true |
        susp_mode(I,A,O).
(5) susp_mode([resume|I],A,O):- true |
        debug(A,I,O).
```

このデバッガの述語 debug は，前のメタインタプリタの述語 call に利用者からの入力と表示のため

に2つの引数を追加したものになっている．(1)節は，ゴールを展開し，各ゴールを実行した結果を述語 merge で統合する．(2)節は，組込み述語の実行であり，実行前後の組込み述語をデータとして出力する．ここで，データ flush_write が出力装置に送られると，出力装置は，その第一引数のデータを物理的な表示装置に出力し，出力が終了したら第二引数に与えられた変数をアトム ok に具体化する．次に述語 syncro は，第一引数が具体化するまで実行を中断しており，それが具体化した後，第二引数のデータをゴールとみなして実行する．二引数述語 sys_exec は，第一引数で与えれる組込み述語の実行結果 true または false をその第二引数に具体化する．これは，メタインタプリタで用いられている一引数 sys_exec を拡張したものである．これらの flush_write, sys_exec, syncro を同期の機構として用いることにより，ゴールの実行前後の変化を表示することが可能となっている．(3)節は，利用者定義ゴールの実行であり，節のボディゴール実行前に，flush_write で，利用者定義のゴールを出力する．(4)節は，利用者から実行中断の命令 suspend が送られた場合である．実行中断の状態は，(5)節で示されているように利用者から新たに実行再開の命令 resume が送られるまで続く．

このデバッガの起動は，以下のようなゴール列を与えることで始まる．

```
?- debug(Target,I,O),input(I),display(O).
```

ここで，Target は，ターゲットプログラムのトップレベルのゴールを表している．述語 input は，利用者からの入力をデバッガに送り込む入力プロセスであり，述語 display は，デバッガの送り出すデータを端末に表示する出力プロセスである．

このデバッガでは，紙面の関係もあり，一リダクション単位の実行（ステップ実行）や，予め指定された特定の述語が呼び出されるまで連続に実行しその述語が呼び出された時に実行を中断するスパイ機能を省略しているが，これらは，メタインタプリタの拡張で簡単に実現できる．そこで，4.3節で示される，拡張されたデバッガにおいては，これらの機能が提供されていると仮定している．

4.2 色付き表現の導入

はじめに，色付き表現を定義する．色付き表現は，プログラム中に現れる定数，構成子，述語記号といったシンボルに対して，それがどのような色を持つかを指定する．各シンボルに付けられた色は，論理的に見た場合には，無視できる．

色は，GHC で用いられる任意の項である．色付きシンボルは，色構成子 *color* と色 c を用いて以下のように表現される．

- 色付き定数 a は，$color(a, c)$ と表現される．
- 色付き構成子 f を持つ n 引数の複合項は，$color(f(T_1, \cdots, T_n), c)$ と表現される．
- 色付き述語記号 p を持つ m 引数の述語は，$color(p(T_1, \cdots, T_m), c)$ と表現される．

ここで，T_i は，項である．

色付きシンボルを含む項の具体例を示す．
color(a,green) と表現される定数 a の色は，green である．
color(f(a,b),(red,100)) と表現される複合項 f(a,b) において，構成子 f の色は，(red,100) である．
color(p(f(color(a,green),b),blue)) と表現される述語 p(f(a,b)) において，述語記号 p の色は，blue であり，定数 a の色は，green である．

4.3 色付きゴールトレーサ

ここでは，3章で述べたトレーサに必要な3つのデバッグ情報のうちの最初の2つ，

1. ある注目するゴールは，どのゴールのリダクション過程で生じたのか．
2. ある注目するゴールは，どのくらいのリダクション数を経過したか．

を明らかにするために，色付きゴールトレーサを構成する．色付きゴールトレーサは，ターゲットプログラムの実行過程で現れた普通のゴールに，4.2節で導入した色付き表現された色付きゴールをリダクションした結果生じた新しいゴールに再び薄めた色を着色することによって，2つのデバッグ情報を出力されたトレース系列に反映させることを可能とする．すなわち，濃淡を無視した色付きトレースの系列は，リダクションの系列に明らかに一致するからである．また，リダクションを行う度に色の濃

淡は単調に減少することにより，リダクションの実行回数と色の濃度は一対一対応する．

色付きゴールトレーサを用いたデバッグ作業のシナリオは，次のようになる．

- 利用者は，デバッガの実行を一時中断し，実行待ちゴールのいくつかに色を付ける．ゴールの色付け作業が終了したら，もとのように実行を再開する．デバッガは，色付きゴールを上で述べたように処理して行くので，画面上には，異なった色と濃淡を持つ実行系列が，次々に出力されることになる．この情報を参照しつつ，利用者は，実行中断，色付け操作，実行再開といった作業を繰り返す．

このシナリオに従って，色付きゴールトレーサを GHC で実現したのが次のプログラムである．

```
(1') debug2(color(Goal,(C,N)),I,O):-
       Goal \= (_,_), usr(Goal) |
       clauses(Goal,NewGoals),
       N1:= N-1,
       O = [flush_write(usr(color(Goal,
                          (C,N))),X)|O1],
       syncro(X,debug2(color(NewGoals,
                          (C,N1)),I,O1)).
(2') susp_mode([set_color|I],Goal,O):-
       true | O =
     [input(['color for ',Goal],Q)|O1],
       set_color(Q,Goal,I,O1).
(3') set_color(color(C),Goal,I,O):-
       Goal \= color(_,_) |
       susp_mode(I,color(Goal,
                          (C,1000)),O).
(4') set_color(no,Goal,I,O):- true |
       susp_mode(I,Goal,O).
```

ここでは，色は，2 項組（識別名，整数）で表現されており，識別名が色の種類を表し，整数が色の濃淡を表す．色は，薄くなるほどに小さな整数が割り当てられる．述語 debug2 の定義は，4.1 節で示された述語 debug とほぼ同様であるので，(1') 節を除いて省略している．この節は，4.2 節の定義 (3) 節に対応しており，利用者定義の色付きゴールをリダクションし，薄めた色を持つサブゴールを起動する処理を定義している．(2') 節は，前に説明した述語 susp_mode に新たに加えられる定義である．

利用者からデータ set_color が入力されると，実行を中断しているゴールを表示して，そのゴールに色付けを行うか否かの応答を求める．変数 Q は，利用者からの応答 color (色識別名) かもしくは，no が具体化するためのものである．(3'), (4') 節は，ゴールに色付け操作を行う述語 set_color の定義である．(3') 節で利用者から color(C) が入力された場合には，色の種類 C と色の濃淡として初期値 1000 を持つ色付きゴールを構成し，実行開始命令 resume を待つ．また，利用者から no が入力された場合は，何もしない．

4.4 色付きデータの利用

本節では，3 章で示したトレーサに必要なデバッグ情報の 3 番目，

3. 変数の具体化は，どのゴールのリダクション過程で行われたのか

を明らかにするために，4.3 節で示された色付きゴールが，ある節をリダクションして新しいサブゴールを生成する時，サブゴールの引数に含まれるデータのシンボルにも色が分配されるようなモデルを考える．このモデルの実現は，以下 2 つの側面，

1. プログラムの変更を行うこと
2. 4.3 節のデバッガを拡張すること

から行われる．

初めに，1. の側面であるが，データのシンボルにどのような色が分配されるかは，実行時まで明らかではないので，実行時に，そのシンボルに着色するべき色を決定することが必要となる．プログラム変更の手続きとしては，以下のようなものを考察した．

- 4.2 節の色付き表現に従って，定義節 cl のゴールの述語記号とその引数に存在する全てのシンボルを，色として未定変数 C を持つ色付き表現に変更した新しい節 cl' を得る．手続きの適用後，clause(変換後の定義節，未定変数 C) の形式でプログラムデータベースに登録する．

この手続きの適用を具体例で示す．

```
(cl) p(u,X):- true |
                  q(X,s(a,b)), r(f(Y),g).
(cl') p(u,X):- color(true,C) |
```

```
color(q(X,color(s(color(a,C),
                  color(b,C),C)),C),
color(r(color(f(Y),C),
                       color(g,C)),C).
```

clause((cl'),C) のような形式で保存しておくことにより，サブゴール q，r に着色する色 cg が明かになった時点で，次に述べる拡張されたデバッガが未定色変数 C と単一化することですべてのデータシンボルに色が伝播される．

次に 2. の側面である，拡張されたデバッガ（の一部）を示す．

```
(1'') debug3(color(Goal,(C,N)),I,O):-
        Goal \= (_,_), usr(Goal) |
        N1:= N-1,
  (a)   c_clauses(Goal,(C,N1),NewGoals),
        O = [flush_write(usr(color(Goal,
                           (C,N)),X)|O1],
  (b)   syncro(X,debug3(NewGoals,I,O1)).
```

4.3 節で示した述語 debug2 の定義 (1') を拡張して得られたのが述語 debug3 の定義節 (1'') である．変更された部分は，(a)，(b) でマークされたゴールである．(a) の c_clauses では，プログラムデータベース中から clause(V,cl) で登録されている変更された節を取り出し，そのうちの一つをコミットして新しいサブゴール NewGoals を取り出す．また，第二引数で与えられた新しい色 (C,N1) を未定色変数 V と単一化する．(b) では，述語 syncro の第一引数の内容が異なっている．これは，プログラム変更手続きで述語を色表現に書き換えているからである．プログラム変更手続きを述語の引数のみを評価するものに変更すれば，4.3 節と同じままでよい．

色表現されたデータを用いる場合には，節のガード及びボディで使用される組込み述語を変更しなければならない．特に，節のボディで行われる単一化述語では，未定義変数に色付き項が単一化されることによって色の伝播が行われるようにする必要がある．この単一化述語は，通常の単一化操作を拡張することで比較的容易に実現できる．また，色の伝播には影響を及ぼさないが，節頭部で行われる一方向単一化についても拡張することが求められる．

5 色付き実行トレーサの実行例

4.4 節で与えた色付き実行トレーサに次のプログラムを適用した実行例を示す．

```
(0) ?- debug3((color(ex20(X),(red,10)),
          color(ex2(X),(blue,10))),I,O1),
       append(O1,[write(data(X))],Out),
       input(I), display(Out).
(1) ex20(X):- true |
       X = mes(hello,Y), ex2(Y).
(2) ex2(mes(hello,X)):- true |
       X = mes(fine,Y), ex2(Y).
(3) ex2(mes(fine,X)):- true |
       X = mes(bye,Y), ex2(Y).
(4) ex2(mes(bye,X)):- true | X = ok.
(5) ex2(ok):- true | true.
```

まず，このプログラムの説明を行う．最初の (0) 節で，ターゲットゴールを設定して，デバッガを起動する．ターゲットゴールは，red ex20 と blue ex2 である．デバッガへの入力は，ゴール input から共有変数 I を通して与えられる．また，デバッガの出力は，述語 append でそのデータの末尾に ex20 と ex2 の共有変数 X の出力命令 [write(data(X))] を加えた上で表示装置に送られる．X の内容を表示することで具体化した内容を確認することができる．

(1)~(5) 節は，述語 ex20 と ex2 の定義である．このプログラムは，データフローモデルでの要求駆動計算に対応したものとなっており，動的に生成されるゴール ex2 をプロセスと見なすことができる．このとき，2 つのプロセス ex2 が，互いに共有変数を通して，データの交換を行なうことで計算が進行するわけである．(1) 節は，ex20 が共有変数 X を mes(hello,Y) に具体化，プロセス ex2 生成して処理を継続する．(2) 節は，データ mes(hello,X) を受け取ったプロセス ex2 の処理の定義であり，X にデータ mes(fine,Y) を具体化し，変数 Y に新しいデータが到着するまで処理を中断する．(3) 節は，データ mes(fine,X) を受け取ったプロセス ex2 の処理の定義であり，X にデータ mes(bye,Y) を具体化し，変数 Y に新しいデータが到着するまで処理を中断する．(4) 節は，データ mes(bye,X) を受け取ったプロセス ex2 の処理の定義であり，X に ok を具体化して処理を終了する．最後の (5) 節は，

データ ok を受け取った ex2 が直ちに終了することを表わしている．このプログラムの実行結果を示す．

```
usr(color(ex20(X),(red,10)))
sys(color(X = color(mes(color(hello,
      (red,9)),A),(red,9)),(red,9)))
exec(省略)
usr(color(ex2(color(mes(color(hello,
      (red,9)),A),(red,9))),(blue,10)))
sys(color(A = color(mes(color(fine,
      (blue,9)),B),(blue,9)),(blue,9)))
exec(省略)
usr(color(ex2(color(mes(color(fine,
      (blue,9)),A),(blue,9))),(red,9)))
sys(color(A = color(mes(color(bye,
      (red,8)),B),(red,8)),(red,8)))
exec(省略)
usr(color(ex2(color(mes(color(bye,
      (red,8)),A),(red,8))),(blue,9)))
sys(color(A = color(ok,(blue,8)),
                              (blue,8)))
exec(省略)
usr(color(ex2(color(ok,(blue,8))),
                              (red,8)))
sys(color(true,(red,7)))
exec(color(true,(red,7)))

data(color(mes(color(hello,(red,9)),
     color(mes(color(fine,(blue,9)),
     color(mes(color(bye,(red,8)),
     color(ok,(blue,8))),(red,8))),
     (blue,9))),(red,9)))
```

省略した部分は，単一化の結果を報告している部分であり，等号の左辺，右辺とも等しくなっていることを報告する．

このままでは，多少読みづらいが，color で始まる色の表現形式をうまく可視化すれば，実行結果から

1. 赤色 ex20 が青色 ex2 に hello を送信する．
2. 青色 ex2 が赤色 ex2 に fine を送信する．
3. 赤色 ex2 が青色 ex2 に bye を送信する．
4. 青色 ex2 が赤色 ex2 に ok を送信する．

という実行のようすを直ちに理解することができる．

これまでは，色付き項がどのように表示されるか全く述べなかった．実際，指定された色で項を表示することができるかどうかは，計算機と端末に依存するので，各々の環境に応じてうまく可視化することが望まれる．例えば，複数の文字フォントが使用できるならば，色の種類を使用可能な文字フォントに写像することができる．

6 まとめと今後の方針

本論文では，従来の実行トレース法に色の効果を導入することによって，デバッグ環境を改善することを提案した．また，単純なメタインタプリタを順次拡張することによって，色付き実行トレーサを構成する手法を述べた．今後，さらに以下のような色の利用法を取り込んだ実行トレーサを作成し，いくつかの具体例に対して適用することでその有効性を確かめる予定である．

1. 4.3 節で導入した色の着色法では，リダクション回数に逆比例して，色が薄くなる一方である．これに対して，スパイ設定されている述語が出現するたびに，色の濃度を初期値に再設定することで他の述語と区別する方法が考えられる．
2. 実行時にプログラマが着色操作を行って，注目するゴールを陽に明らかにする手法にたいして，例えば，デバッグ作業で変更された定義節，もしくはゴールが注目する対象であると捉える手法も考えられる．そこで，節のバージョン番号と色を対応させることも興味深いテーマである．
3. デバッグとは多少レベルが異なるが，その他の色の利用法として，ハードウェアの状態を色で表現し，それをトレース表示に反映させることが考えられる．例えば，各々の分散プロセッサに固有の色を対応させて，各色付きプロセッサでリダクションされたゴールに，その色を着色して表示すれば，負荷分散の状況をトレース表示に反映できる．

謝辞 本研究のきっかけを与えて下さった大阪大学基礎工学部情報工学科の都倉信樹教授，魚井宏高助手に感謝致します．また，本論文の構成にあた

て多数の御助言をいただいた神田陽治氏，田中二郎氏，及び国際研第二研究部第二研究室の皆様に感謝致します．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われたものです．

参考文献

[Brock 81] J.D.Brock,W.B.Ackermann: Scenarios:A model of Nondeterminate Computation,Lecture Notes in Computer Science, No.107 (1981).

[Clocksin 83] W.F.Clocksin, 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).

[平野 89] 平野喜芳,et.al.: 汎用計算機上の KL1 処理系 - PDSS -, PROCEEDING OF THE LOGIC PROGRAMMING CONFERENCE '89, pp.193-202 (1985).

[Kohda 89] Y.Kohda: MASAE Project Overview, Fujitsu IIAS Research Report No.89 (March 1989).

[Lippman 85] A.Lippman,et.al.: Color Word Processing,IEEE Computer Graphics & Applications (June 1985).

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

[前田 89] 前田宗則: GHC プログラムのデバッグ手法，大阪大学基礎工学研究科 修士学位論文 (1989年3月).

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

[高橋 85] 高橋秀久，柴山悦哉: Prolog のデバッグ支援環境に対する一提案, PROCEEDING OF THE LOGIC PROGRAMMING CONFERENCE '85, pp.107-118 (1985).

[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).

[竹内 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.193 (1987年9月).

[田中 88] 田中二郎，的野文夫: 変数管理をする GHC の自己記述, 電子情報通信学会誌 COMP88-5, pp.41-49 (1988年5月26日).

[Tanaka 88] J.Tanaka: Meta-interpreters and Reflective Operations in GHC, PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON FIFTH GENERATION COMPUTER SYSTEMS 1988, pp.774-783 (Nov.28-Dec.2 1988).

[舘村 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).

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