FGHC の双対変換に基づく Continuation と Migration の記述

Representation of Continuation and Migration Based on Dual Transformation of FGHC Programs

久門 耕一* 平田 圭二†
Kouichi KUMON Keiji HIRATA

FGHC プログラムにおいて, プロセスとデータの役割を入れ換える双対変換を施すことによって, 計算の状態や途中結果をプロセスやデータに自由に保持させることができる. この双対変換によって, 実行過程を表すプロセスの状態が first-class object として扱えるようになり, process migration や continuation-passing が FGHC で記述できる. 本論文では, FGHC プログラム例を示し双対変換を施しそれらに考察を加える.

1 はじめに

我々は FGHC のソース上での双対な変換の手法を提案した [1][2]. この双対変換により FGHC の永続プロセスがストリームによりリスト構造を生成しているプロセスリストと, データ (例えば cons) がリスト構造を生成しているメッセージリストを相互に変換することが出来る. 変換には, 上田によるモード解析の手法 [3] を用いることでプログラム中のどの変数を書換えの対象にするかを決定できる.

ある時点に存在するプロセス数に比べ同時に存在するメッセージ数が少ない場合にはこの変換を行なうことによりプロセスとメッセージの関係が逆になり, プロセス数がメッセージ数よりも少なくなる. この結果, 一度実行状態になったプロセスが中断せずにいくつかのメッセージを処理する事になり, 実行効率を上げる事が出来た.

しかし, この双対変換は generator-consumer 型のプロセス, メッセージストリーム以外にも適用することが出来る. 一般に双対変換を行なうことにより, プロセスが first-class object として表現されプロセスの実行を陽に制御することが可能となる.

本稿では, 双対変換を用いたプロセスの実行制御の例として continuation-passing style(CPS) による実行の最適化および分散計算機環境下でのプロセス移動に関しての考察を述べる.

2 双対変換による Continuation の表現

図 1 の FGHC プログラムは深さ優先で左から右へ二分木を探索するプログラムである. このプログラムは指定されたキーが葉に存在するか否かによりその後の処理を変えている.

このプログラムでは, 左部分木の探索結果が `search1` の第一引数に与えられ, その値により右部分木の探索を行なうか行なわないかを決めている. `search1` の第一引数に関しての双対変換を行なったプログラムが次の図 2 である. この変数は, いわゆるメッセージストリームを構成する変数ではない.

元のプログラムでは, `search` の能動部に二つのユーザゴール呼び出しが存在しあたかも並列に実行できるように見えたが, 変換後の図 2 では, すべての述語の能動部は高々一つのユーザゴール呼び出しのみが書かれている. 元のアルゴリズムに並列度がなかったためである.

ボディー部のゴールは一つになったが, その後に実行するべきゴールはヒープ中に項データとしてスタックされているので continuation-passing style(CPS) と呼ぶことが出来る. 述語の実行順序を continuation により陽に指定できるという意味では [4] と同様である.

一般に FGHC の処理系ではボディーゴールが一つの場合には Prolog における終端最適化と同じように実行中のゴールの情報をボディーゴール用に使い回しすることで enqueue, dequeue を省く最適化が行なわれている. 二つ以上ある場合にはゴールの

*(財) 新世代コンピュータ技術開発機構 Institute for New Generation Computer Technology
†NTT 基礎研究所 NTT Basic Research Laboratories

```
?- search(Key,Tree,Res), cont(Res,Parm).

search(K,T,CP):- T=leaf(VV), K = VV |
        CP = found.
search(K,T,CP):- T=leaf(VV), K \= VV |
        CP = not_found.
search(K,T,CP):- T=node(T0,T1) |
        search(K,T0,CP0),
        search1(CP0,K,T1,CP).

search1(CP0,K,T1,CP):- CP0 = found |
        CP = found.
search1(CP0,K,T1,CP):- CP0 = not_found |
        search(K,T1,CP).

cont(CP,Param):- CP=found | found_proc(Param).
cont(CP,Param):- CP=not_found |
         not_found_proc(Param).
```

図 1: FGHC による 二分木探索プログラム例

```
?-search(Key,T,Res), Res=cont(Param).

search(K,T,CP):- T=leaf(VV), VV = K |
        found(CP).
search(K,T,CP):- T=leaf(VV), VV \= K |
        not_found(CP).
search(K,T,CP):- T=node(T0,T1) |
        search(K,T0,CP0),
        CP0=search1(K,T1,CP).

found(CP):- CP=search1(K,T1,CP0) |
        found(CP0).
found(CP):- CP=cont(Param) | found_proc(Param).

not_found(CP):- CP=search1(K,T1,CP0) |
        search(K,T1,CP0).
not_found(CP):- CP=cont(Param) |
        not_found_proc(Param).
```

図 2: 双対変換を施した二分木探索プログラム

キューへの enqueue, dequeue が必要であるが, ゴールキューの操作は重くなるのが普通である. 従って, ボディーゴールが一つになることは, ゴール操作を省くという点で好都合である.

変換によって左部分木探索の結果を待ち受けていた述語 search1 が search の第三引数に continuation を表すデータとして積まれている. search の第一引数に与えた探索の対象が見つかると found が呼ばれてこの continuation がたぐられる. 二分木の根の部分に積まれた continuation である cont/1 までたぐられた時に最後の処理が行なわれる.

しかし, プログラムの挙動を考えれば found が一つ一つ continuation をたぐるのは無駄なので, found が呼ばれた時に一気に根の部分の continuation に制御を移す事で実行の最適化を図ることが出来る. そのように書き換えたものが図 3 である.

このプログラムでは, found が呼ばれた時に実行する continuation を確保するため最初に search を呼び出した時の continuation を独立に search の第四引数として渡している.

ここで行なったように述語間の結果の引渡しに用いられた変数に対しての双対変換を行なうことにより, continuation を first-class object として陽に表す事ができ, 実行制御をソースレベルで扱う事が可能になる.

図 1 では found_proc や not_found_proc は cont から実行され, 図 2, 図 3 では search から派生したプロセス found, not_found から実行されることになる. この二つは, 論理的な意味は等価であるが並列計算機下での実行時にはどこで実行が行なわれるかが異なる場合がある.

次節ではこの点に注目し, ソースレベルでの書き換えにより実行プロセスを積極的に移動する事について考察する.

```
?-search(Key,T,Res,Res), Res=cont(Param).

search(K,T,CP,FC):- T=leaf(VV), VV = K |
        found(FC).
search(K,T,CP,FC):- T=leaf(VV), VV \= K |
        not_found(CP,FC).
search(K,T,CP,FC):- T=node(T0,T1) |
        search(K,T0,CPP,FC),
        CPP=search1(K,T1,CP).

found(FC):- FC=cont(Param) | found_proc(Param).

not_found(CP,FC0):- CP=search1(K,T1,CPP) |
        search(K,T1,CPP,FC0).
not_found(CP,FC):- CP=cont(Param) |
        not_found_proc(Param).
```

図 3: CPS の操作により最適化した二分木探索プログラム

3 Process Migration の実現

FGHC プログラムを並列計算機環境で実行することを考慮に入れ, FGHC 処理系に関して以下のような自然な条件を仮定する.

- ある Processing Element (PE) 上で親プロセスが実行されたら, (デフォールトでは) その子プロセスも同一 PE 上で実行される.
- 子プロセス起動時に allocate するデータは, その PE が直接アクセスできるメモリに allocate する.
- 他 PE が持つデータをアクセスする時, データはメッセージとなって他 PE に移動する. データを要求しているプロセスは (デフォールトでは) 移動しない.

従って, この仮定の元では, 負荷分散のためにプロセスを移動させたい時には, 何らかのメタ機能が必要になる. また通信相手のプロセスがどの PE に居るのかは一般にオブジェクトレベルでは識別できず, このためにも何らかのメタ機能が必要になる. しかし双対変換を適用しプロセスを first-class object に変換すると, メタ機能を使わずにメッセージとして他 PE に移動させること (migration) が可能となる.

まず二進探索木の左回転操作のプログラムを考えよう. ここで二進木の各ノードは `node(Val, Left, Right)` という項で表す. 例えば `node(a, X, node(b, Y, Z))` のような二進木が左回転により `node(b, node(a, X, Y), Z)` のような二進木に変換されるので, プログラムは素直に図 4 のように書ける. 文献 [2] のストリームの定義に従うと, このような `node`

```
left_rot(In, Out) :-
        In = node(A, X, node(B, Y, Z)) |
  Out = node(B, node(A, X, Y), Z).
```

図 4: 二進探索木の左回転操作

から構成される木構造もストリームと呼べる. しかし本プログラムのように木構造が動的に変形を受けるような場合, それが一般的な意味でのストリームと呼ばれることは少ないであろう.

このプログラムを `node` に関して双対なプログラムに変換し, 項の `node` をプロセスにすることができる. まず双対変換が適用できるような標準形に直す. `left_rot/2` を一引数化し, ボディ部の active unification を分解する (図 5).

そして図 5 の `left_rot/2` の第一引数, 第二引数, 及び `left_rot_sub/4` の第三引数, 第四引数に関して双対なプログラムに変換する (図 6).

図 6 のプログラムは, 親の `node/4` プロセスが `left_rot` というメッセージを受け取ると, 自分の内部状態を `left_rot_sub` というメッセージの中に埋め込み右側の子プロセス (第四引数) に送り付け, その子プロセスの子プロセスとして自分を再生するように読める. この時, 親プロセスは子プロセスを実行している PE に移動することができたので, migrate したと言って良いだろう. 次にこの技法を一般化しよう.

図 7 の簡単なプログラムを考える. ここで `p1/3`, `p2/3` というプロセスは, 異なる PE で実行されているとし, それぞれのプロセスの内部状態は `S1`, `S2` で表現されている. 親プロセス `p1/3` の子プロセス `p1/3` は親と同一の PE 上で実行される. 図 8 は,

```
left_rot(In, Out) :- In = node(A, X, BYZ) |
  left_rot_sub(A, X, BYZ, Out).

left_rot_sub(A, X, BYZ, Out) :-
        BYZ = node(B, Y, Z) |
  Out = node(B, AXY, Z),
  AXY = node(A, X, Y).
```

図 5: 双対変換に先立ち標準形に直す

```
node(In, A, X, BYZ) :- In = left_rot(Out) |
  BYZ = left_rot_sub(A, X, Out).

node(BYZ, B, Y, Z) :-
        BYZ = left_rot_sub(A, X, Out) |
  node(Out, B, AXY, Z),
  node(AXY, A, X, Y).
```

図 6: 双対な二進探索木の左回転操作

プロセス p1/3 を, p2/3 を実行している PE に migrate するプログラムである. p1/3 が t/1 というメッセージを受け取ると, その内部状態と子プロセスのメッセージ受口を t/2 メッセージに埋め込みストリーム通信を行なっていた変数を通じて, p2/3 の PE に移動させる. p2/3 側では受けとったメッセージから p1/3 を再生する.

図 9 は逆に p2/3 を p1/3 側に migrate するプログラムである. ただし, migrate するプロセス p2/3 の情報を運ぶ項 h/2 を供給するのは p1/3 である.

実際に通信を行なっている相手プロセスの居る PE に対して負荷分散を行う時, オブジェクトレベルで相手 PE を知るには何らかのメタ機能が必要となる. 本手法では通信に使っているストリームを migration のために兼用することで, そのようなメタ機能を使わずに済ませている.

上の二例では, プロセスが一旦他の PE に移動してしまうと元の PE 上での実行に戻ることはできない. そこで p1/3 と p2/3 を実行している PE を同時に入れ換えるプログラムを同様の手法で実現する (図 10).

4 おわりに

本論文では, FGHC 処理系の実行方式に関して現実的で自然な仮定をおいた上で, 双対変換による continuation, migration の記述手法を提案した. 本手法で実際に FGHC プログラムの実行効率を高めることができた. また本手法を新しい FGHC プログラミング方法論と位置付けることもできる.

双対変換をストリーム以外のデータ構造に適用し, その適用範囲の広さ, 有効性を示すことができた.

さらに双対変換を様々な局面に応用することで, もっと複雑なプロセスの挙動を制御することができるであろう. これは今後の課題としたい.

```
?- .., p1(X, S1, Y), p2(Y, S2, Z), ..

p1(X, S1, Y) :- X = nop(XX) |
  Y = nop(YY),
  p1(XX, S1, YY).

p2(Y, S2, Z) :- Y = nop(YY) |
  p2(YY, S2, Z).
```

図 7: Migration のためのプログラム例

```
p1(X, S1, Y) :- X = t(XX) |
  Y = t(S1, XX).

p2(Y, S2, Z) :- Y = t(S1, XX) |
  p1(XX, S1, M),
  p2(M, S2, Z).
```

図 8: p1 を p2 側に migrate するプログラム

```
p1(X, S1, Y) :- X = h(XX) |
  Y = h(S2, ZZ),
  p1(XX, S1, M),
  p2(M, S2, ZZ).

p2(Y, B, Z) :- Y = h(S2, ZZ) |
  S2 = B,
  ZZ = Z.
```

図 9: p2 を p1 側に migrate するプログラム

参考文献

[1] 久門耕一, 平田圭二: FGHC プログラムにおけるプロセスとメッセージの交換 – 実行スレッドに着目した効率改善, 日本ソフトウェア科学会第 9 回大会, A2-3 (1992).

[2] 平田圭二, 久門耕一: FGHC プログラムにおけるプロセスとメッセージの双対性, 日本ソフトウェア科学会第 9 回大会, A2-4 (1992).

[3] Ueda, K., and Morita, M. : Message-Oriented Parallel Implementation of Moded Flat GHC, *FGCS'92*, *Vol. 2*, *ICOT* (1992), pp. 799–808.

[4] Ueda, K. : Making Exhaustive Search Programs Deterministic, *New Generation Computing vol. 5*, *OHMSHA LTD.*(1987), pp. 29-44.


```
p1(X, A, Y) :- X = s(XX) |
  Y = s(A, XX, M, S2, ZZ),
  p2(M, S2, ZZ).

p2(Y, B, Z) :- Y = s(S1, XX, M, S2, ZZ) |
  S2 = B,
  ZZ = Z,
  p1(XX, S1, M).
```

図 10: p1 と p2 を実行する PE の交換