FGHC プログラムにおけるプロセスとメッセージの双対性

Duality of Processes and Messages in FGHC Programs

平田 圭二* 久門 耕一*
Keiji HIRATA Kouichi KUMON

概要 ある制限に従う FGHC プログラムに対して, そのプロセスとメッセージの役割を入れ替えた双対なプログラムが存在する. 本論文では, まず FGHC プログラムに対する制限と, FGHC プログラムのプロセスとメッセージの役割を置き換えるような変換規則を定義する. 次に, その変換規則の適用個所を同定するアルゴリズムと例を示す.

1 はじめに

ストリームで運ばれるような情報を, プロセスの内部状態として表現するのかメッセージの持つデータとして表現するのかは任意である [3]. 本論文では, ストリーム処理を行う FGHC プログラムに関して, アルゴリズムが等しく, プロセスの内部状態として持つ情報とメッセージのデータとして持つ情報を入れ替えたようなプログラム (双対なプログラム) が存在し互いに変換できることを示す. 双対なプログラムでは, ストリームの表現方法が異なるだけでアルゴリズムと計算の結果は等しい. 次に, 双対なプログラムを得るために抽象解釈を用いる手法を提案する. ある制限付きの FGHC プログラムを抽象解釈することで, ストリームの生成点, 消費点を発見し双対なプログラムへ変換するための書き換え個所を同定することができる.

2 FGHC プログラムの双対性

2.1 ストリーム

FGHC におけるストリームというデータ構造を定義する. メッセージにより構成されるストリームと, プロセスにより構成されるストリームの 2 種類がある. メッセージストリームは 2 種類のファンクタ (関数シンボル) を, プロセスストリームは 2 種類のリテラルをコンストラクタとして持つ. この 2 種類のコンストラクタの内, 1 つは継続 (continuation) を, もう 1 つは終端 (termination) を表現している (文献 [3] 図 4, 5). ストリームは木構造であり, 1 個以上の継続を持つことができる. 終端を表現するファンクタやリテラルからさらにポインタが延びていても, そのストリームは終端のファンクタかリテラルで完結しているとみなす.

2.2 プログラムに対する制限

本論文で扱う FGHC プログラムには次のような制限を設ける.
(a) リテラルやファンクタの引数のトップレベルには変数のみ現れる. ファンクタは = (単一化子) の引数のトップレベルにのみ出現する. 入れ子構造が現れない.
(b) 1 つの節あたり, 構造体の passive 単一化は高々 1 段である.
(c) ストリームを参照する変数の出現パターンは次の 3 通りである[1]: (i) ボディ部に 2 回出現, (ii) ガード部に 2 回出現, (iii) ボディ部, ガード部に 1 回ずつ出現.
(d) 各プロセスのストリーム入力は高々 1 個である. **(e)** ストリームを参照する変数に対して 1-writer 1-reader である.
(f) ストリームを参照する変数の writer は, 2 種類のストリームコンストラクタを接続する.

*(財) 新世代コンピュータ技術開発機構 Institute for New Generation Computer Technology
[1]これ以外の出現パターンの変数がストリームの生成, 消費に関与していても, 書き換え個所の対象とはなり得ない.

例として, メッセージストリームを生成する述語 p/1 とプロセスストリームを生成する述語 q/1 を制限付 FGHC プログラムにより記述する.

```
p(X) :- true|X=m(a,Y),Y=m(b,Z),Z=nil.
q(X) :- true|m(X,a,Y),m(Y,b,Z),nil(Z).
```

この例では, メッセージストリームもプロセスストリームも本質的には [a,b] という同一のストリームを表現している. 下は入力ストリームの分解も= を用いて陽に記述しなくてはいけないという例である.

```
r(X) :- X=s(Y) | r(Y).
```

次節では, 上述の制限付 FGHC プログラムで表現されるストリームの生成, 消費に関して, その双対性について議論する.

2.3 ストリームの生成に関する双対性

制限付 FGHC では, メッセージストリームは次のようなプログラムによってのみ生成される (証明略).

```
a(X) :- ··· | ···                                    (1)
··· :- ··· | ···,X=b(Y),c(Y),···                    (2)
··· :- ··· | ···,X=d,···                            (3)
```

a(X) がメッセージストリームを生成するトップレベルの述語である. (1) の X がリレーされて[2], (2) X として現れている. (1) と (2) は同一節に存在しても良い. (2) の Y を辿ると, いつかは Y=a(YY) という active 単一化が出現する. (2) c(Y) は a(Y) でも良いし, このレベルに Y=a(YY) が直接書かれても良い. (3) の d は終端を表わす.

一方, 制限付 FGHC では, プロセスストリームは次のようなプログラムによってのみ生成される (証明略).

```
a(X) :- ··· | ···                                    (4)
··· :- ··· | ···,b(X,Y),c(Y),···                    (5)
··· :- ··· | ···,d(X),···                           (6)
```

プログラム (4) ∼ (6) について, メッセージストリームの場合と同様な注釈を与えることができる.

上の各節は, (n) と $(n+3)$ $(n = 1..3)$ がそれぞれ対応している. (1)∼(3) と (4)∼(6) の 2 つのプログラム間には, ストリームの表現と操作に関して 1 対 1 の対応関係が存在し, (概念的に) 同じ計算を実現しているので, これらのプログラムは双対であると呼ぼう. (1)∼(6) から明らかなように, a(X) によるストリームの生成に関して双対なプログラムへの変換とは, ストリームに新たなコンストラクタを 1 つ接続する点を見出し, (2)↔(5), (3)↔(6) のように書き換えることであり, 変換後もまた制限付 FGHC プログラムである.

2.4 ストリームの消費に関する双対性

制限付 FGHC では, メッセージストリームは次のようなプログラムによってのみ消費される (証明略). ただし d(X) の第 1 引数には X=e(A),A=e(B),···,C=h というメッセージストリームが入力されることを仮定している.

```
d(X) :- X=e(Y),··· | ···                             (7)
··· :- ··· | ···,f(Y),···                           (8)
g(X) :- X=h,··· | ···                                (9)
```

(7) の Y を辿ると, いつかは (8) のような Y=e(YY) という passive 単一化を行う述語 f を呼び出す. (7) と (8) は同一節であっても良い. (8)(9) の述語名 f, g は d でも良い. (9) の h は終端に対応している.

一方, 制限付 FGHC では e(X,A),e(A,B),···,h(C) というプロセスストリームは, 次のようなプログラムによってのみ消費される (証明略).

```
e(X,Y) :- X=d,··· | ···                              (10)
··· :- ··· | ···,Y=f,···                            (11)
h(X) :- X=g,··· | ···                                (12)
```

[2] 1-writer 1-reader の関係を保存しながらポインタを受け渡していくこと.

プログラム (10)∼(12) についてメッセージストリームの場合と同様の (双対な) 注釈が与えられる. 上の各節は, (n) と $(n+3)$ $(n=7..9)$ がそれぞれ対応しており, (7)∼(9) と (10)∼(12) のプログラムは双対である.

(7)∼(12) から明らかなように, X が参照しているストリームの消費に関して双対なプログラムへの変換とは, ストリームからコンストラクタを 1 つ取り出す点を見出し, (7)↔(10), (8)↔(11), (9)↔(12) のように書き換えることである.

以上の議論よりストリームの生成と消費に関して FGHC のプロセスとメッセージには双対性があることが分かった.

3 双対なプログラムへの書き換え

3.1 FGHC プログラムの抽象解釈

第 2.2 節で挙げた制限の内, (a)∼(c) はプログラムから textual に判定し, (d)∼(f) はプログラムの抽象解釈により判定する. 以下, 抽象解釈の手順を簡単に述べる.

[1] 第 2.2 節 (a)(b) の制限を満たすプログラムに関して, (c) の制限を満たす変数のみに着目する.

[2] それらの変数が, リテラルのトップレベルからどのようなパスを通って出現しているかの関係を, 上田式モード解析 [4] の記法を拡張して記述する. 変数へのパス P は一般に $[f,i][g,j][h,k]\cdots$ や $\forall\xi\,[f,i]\xi$ のように書かれる. ここで f はリテラルを, g, h はファンクタを, i, j は引数の位置を表わす. パス式はパス同士の関係を記述し, $\forall\, P=P'$, $\forall\, P\doteq P'$, $P=M$ のいずれかの形をしている (ここで M は入出力モード). ここに例を 2 つ示す.

例 1: `r(X,Y) :- true | X=s(Y).`

`X` $[r,1]=out(s/1)$
`Y` $\forall\xi\,[r,2]\xi\doteq[r,1][s,1]\xi$

例 2: `t(X,Y) :- X=u(A) | Y=v(B),t(A,B).`

`X` $[t,1]=in(u/1)$
`XX` $\forall\xi\,[t,1][u,1]\xi=[t,1]^{+}\xi$
`Y` $[t,2]=out(v/1)$
`YY` $\forall\eta\,[t,2][v,1]\eta\doteq[t,2]^{+}\eta$

[3] 各変数から得たパス式の集合を元に抽象解釈する. パス左端に来る $[p,i]$ は FGHC のプロセスに対応しており, 抽象解釈時に世代番号 (generation number) が振られて行く. ボディのプロセスはヘッドのプロセスより必ず大きな世代番号を持ち, ボディのプロセスは各々異なる世代番号を持つ. 式の右辺に見られる $[t,1]^{+}$ の $+$ 記号は, その世代番号の振り方に対する指示である. 抽象解釈時には,

$$P_i=P_j \text{ かつ } P_j=P_k \Rightarrow P_i=P_k,$$
$$P_i\doteq P_j \text{ かつ } P_j\doteq P_k \Rightarrow P_i=P_k$$

のような推論が行われる. 抽象解釈の結果として, パスをノードに, $=$ と $\doteq$ をアークに持つようなグラフが得られる.

3.2 書き換え個所の同定

抽象解釈の結果として得られたグラフ中に, 以下に挙げるようなパターンが発見されると, それはストリームの生成点, 消費点に対応する.

• メッセージストリームの生成

$$[p,i]^{g}=out(c/m,t/n)$$
$$[p,i]^{g}[c,j]=out(c/m,t/n)$$
$$[p,i]^{g}[c,j][c,j]=out(c/m,t/n)\ \cdots$$

継続コンストラクタ c の arity は m, 終端コンストラクタ t の arity は n である. 各 $[c,j]$ を生成する個所が, 双対プログラムへ変換するための書き換えるべき所である. ストリームが一本鎖ではなく木構造の時も同様である.

• メッセージストリームの消費

$$[p,i]^{g}=in(c/m,t/n)$$
$$[p,i]^{g}[c,j]=in(c/m,t/n)$$
$$[p,i]^{g}[c,j][c,j]=in(c/m,t/n)\ \cdots$$

堀内の抽象解釈 [2] の枠組を使うとメッセージストリームの繰り返し構造が検出できる.

• プロセスストリームの生成

以下 $\forall$ を省略して記す.

$$P\xi=[t,k]^{h}\xi$$
$$P\xi=[c,j]^{g1}\xi,\ [c,j]^{g1}\eta\doteq[t,k]^{h}\eta$$
$$P\xi=[c,j]^{g1'}\xi,\ [c,j]^{g1'}\eta\doteq[c,i]^{g2'}\eta,[c,i]^{g2'}\zeta\doteq[t,k]^{h}\zeta\ \cdots$$

ただし $m < n \to g_m < g_n, i \neq j$. ここで P はプロセスストリームを出力する変数へのパス.

- **プロセスストリームの消費**

$$
\begin{aligned}
&[t,k]^h = in(m_1/a_{m_1})\\
&\quad [c,i]^{g_1'} = in(m_1'/a_{m_1'})\\
&\quad [c,j]^{g_1'} = out(m_2'/a_{m_2'})\\
&\quad [t,k]^{h'} = in(m_2'/a_{m_2'})\\
&\qquad [c,i]^{g_1''} = in(m_1''/a_{m_1''})\\
&\qquad [c,j]^{g_1''} = out(m_2''/a_{m_2''})\\
&\qquad [c,i]^{g_2''} = in(m_2''/a_{m_2''})\\
&\qquad [c,j]^{g_2''} = out(m_3''/a_{m_3''})\\
&\qquad [t,k]^{h''} = in(m_3''/a_{m_3''}) \cdots
\end{aligned}
$$

このパターンを検出するために, Codish がサスペンション解析 [1] のために導入した抽象解釈の枠組を応用する.
書き換えを行う際は, 対応する生成点と消費点を必ず同時に書き換えなければならない.

4 例題

4.1 Append プログラム

```
ap(X,Y,Z) :- X=c(E,XX) | Z=c(E,ZZ),ap(XX,Y,ZZ).
ap(U,V,W) :- U=n | V=W.
```

このプログラムの第 1, 2 節から得られるパス式を示す (以下 $\forall$ は全て省略する).

X	$[ap,1] = in(c/2)$
XX	$[ap,1][c,2]\xi = [ap,1]^{+}\xi$
Y	$[ap,2]\eta = [ap,2]^{+}\eta$
Z	$[ap,3] = out(c/2)$
ZZ	$[ap,3][c,2]\zeta = [c,3]^{+}\zeta$
U	$[ap,1] = in(n/0)$
V,W	$[ap,2]\xi \doteq [ap,3]\xi$

そして抽象解釈の結果を示す.

$$
\begin{aligned}
&[ap,1]^1 = in(c/2, n/0)\\
&[ap,1]^1[c,2] = in(c/2, n/0)\\
&[ap,1]^1[c,2][c,2] = in(c/2, n/0) \cdots\\
&[ap,3]^1 = out(c/2)\\
&[ap,3]^1\zeta = [ap,2]^1\zeta\\
&[ap,3]^1[c,2] = out(c/2)\\
&[ap,3]^1[c,2]\zeta = [ap,2]^1\zeta\\
&[ap,3]^1[c,2][c,2] = out(c/2)\\
&[ap,3]^1[c,2][c,2]\zeta = [ap,2]^1\zeta \cdots
\end{aligned}
$$

これより $[ap,1]$ がメッセージストリームの消費点であることが分かる. $[ap,3]$ はメッセージストリームの生成点ではない.

4.2 Naive Reverse プログラム

```
nr(X,A) :- X=c(E,XX) | nr(XX,Y), ap(Y,Z,A),Z=c(E,ZZ),ZZ=n.
nr(V,W) :- V=n | W=n.
```

このプログラム第 1, 2 節から得られるパス式を示す.

X	$[nr,1] = in(c/2)$
XX	$[nr,1][c,2]\xi = [nr,1]^{+}\xi$
A	$[nr,2]\alpha = [ap,3]^{++}\alpha$
Y	$[nr,2]^{+}\eta \doteq [ap,1]^{++}\eta$
Z	$[ap,2]^{++} = in(c/2)$
ZZ	$[ap,2]^{++}[c,2] = in(n/0)$
V	$[nr,1] = in(n/0)$
W	$[nr,2] = out(n/0)$

次に抽象解釈の結果 (の一部) を示す.

$[nr,1]^1 = in(c/2, n/0)$
$[nr,1]^1[c,2] = in(c/2, n/0)$
$[nr,1]^1[c,2][c,2] = in(c/2, n/0)$ ・・・
$[nr,2]^1 = out(c/2, n/0)$
$[nr,2]^1[c,2] = out(c/2, n/0)$
$[nr,2]^1[c,2][c,2] = out(c/2, n/0)$ ・・・

これより $[nr,1]$ がメッセージストリームの消費点, $[nr,2]$ がメッセージストリームの生成点であることが分かる.

以上の抽象解釈の結果に従い, その入力も出力もプロセスストリームであるような双対な naive reverse プログラムに変換する[3]. nr から呼ばれる ap の $[ap,3]$ はメッセージストリームの生成点になっている. ap も同時に変換しなければならない.

```
c(X,E,XX) :- X=nr(A) | XX=nr(Y), Y=ap(Z,A),c(Z,E,ZZ),n(ZZ).
c(X,E,XX) :- X=ap(Y,Z) | c(Z,E,ZZ),XX=ap(Y,ZZ).
n(V) :- V=nr(W) | n(W).
n(U) :- U=ap(V,W) | V=W.
```

入力のプロセスストリームに対し nr, ap というメッセージが通り抜けて行くような実行イメージである. この双対なプログラムをさらに双対なプログラムに変換すると, 元の naive reverse プログラムに戻る.

4.3 制限付 FGHC プログラムへの変換

制限付 FGHC プログラム (2.2 節) の条件 (b)(d) を満たすためには, プログラムの意味が変わらなければ, カリー化 (currying) を行えば良い. 例えば,

```
ad(X,Y,Z):- X=i(A,AA),Y=i(B,BB) | S:=A+B,Z=i(S,CC),ad(AA,BB,CC).
```

というプログラムは,

```
ad(X,Y,C) :- X=i(A,AA) | ad2(Y,A,AA,C).
ad2(Y,A,AA,C) :- Y=i(B,BB) | S := A+B,C=i(S,CC),ad(AA,BB,CC).
```

と変換される. また

```
h(X):- X=i(a,Y),Y=i(b,Z) | ···
```

のようなプログラムも中間の述語を設ける手法で制限を満たすようになる.

条件 (c)(e) を満たすためには, プログラムの意味が変わらなければ, 出力を陽に複製し, 例えば

```
··· :- ··· | X=p,q(X),r(X).
⇒··· :- ··· | X=p,Y=p,q(X),r(Y).
```

のように変換する.

条件 (f) を満たすためには, プログラムの意味が変わらなければ, ストリームコンストラクタを 1 段外側に被せれば良い. 例えば

```
X=i(1,Y), Y=j(2,Z), Z=k.
⇒X=h(i(1),Y), Y=h(j(2),Z), Z=k.
```

のように変換する.

5 おわりに

第 2.2 節で述べたような制限付 FGHC プログラムに関して, 双対なプログラムへの書き換えが機械的に行えることを示した. 現在, 本論文で述べた抽象解釈システムを KL1 で実装中である.

merge のような非決定的な述語は, 現在の枠組では双対なプログラムに変換できない. 非決定的な述語に関する双対性については別の論文で議論する予定である.

双対なプログラムへの変換で実行スレッドの制御ができる. 効率を考慮したスレッド制御を行うために, どの部分を双対なプログラムに変換すべきか判定するのは今後の課題であろう.

[3]入力か出力の一方だけを変換することも可能である.

謝辞

ICOT 上田和紀氏, 堀内謙二氏との大変有意義な議論のおかげで本論文を書くことができた. ここに感謝する.

参考文献

[1] Codish, M., Falaschi, M., and Marriott, K. : Suspension Analysis for Concurrent Logic Programs, *8th ICLP* (1991), pp. 331–345.

[2] Horiuchi, K.: Less Abstract Semantics for Abstract Interpretation of FGHC Programs, *FGCS'92, Vol. 2, ICOT* (1992), pp. 897–906.

[3] 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).

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