数理解析研究所講究録
902巻 1995年 117-132

FGHC プログラムの意味を考える道具としての π 計算

NTT 基礎研究所 平田 圭二 (Keiji Hirata)

1 はじめに

並行論理型言語の効率的な実行のためには, ソースプログラムの静的解析を行い, その結果をもとに最適化やプログラム変換などを施すことが必要である. 我々は, その効率化のためのプログラム変換手法の 1 つとして双対変換を提案した [3]. 次に解決すべき問題は双対変換を適用すべき個所を正しく見い出すことである. 誤った個所に双対変換を適用すると逆に効率が低下してしまう可能性がある. 双対変換を施したプログラムの抽象実行により双対変換の適用個所, 及びどの程度の効率化が達成できるかの見積もりが得られると考えている. そこで, まず双対変換を形式的に記述するのに適した FGHC の操作的意味を定義する必要があった. 本論文では, FGHC プログラムを Polyadic π 計算 (PPC) へ翻訳する方法を述べるが, FGHC の述語や項がともに PPC のプロセスとして統一的に表現されるので, 双対変換との親和性が高いと考えられる.

また並行論理型言語では, 変数が一度具体化されたらそれ以降その値は不変であるという論理性を壊すような操作は, 一般に許されていない. ところが, 論理性を壊す破壊的な書き換えでも制限しつつ安全に用いることで, 実用的には効率的なアルゴリズム, 効率的な実行を達成することができる. 例えば, 通常のコーディングではリストを用いるところを配列を用いたアルゴリズムで実現したり, 破壊書き換えの行える構造体を用意して入力本数に依存しない定数オーダのマージャを実現することができる. しかし, 従来より提案されている並行論理型言語の意味論を, このような破壊的な書き換え操作をプリミティブとして持つような言語の意味論へと拡張するのはかなり困難に思える. そこで我々は π 計算に着目した; 並行論理型言語ではプロセス間でデータ構造自体を通信するのに対し, π 計算では name つまりデータ構造への参照を通信することが基本となっている. 実際に, プロセス間で共有されるデータ構造に対して破壊書き換えを行う FGHC のプリミティブを PPC で記述することで, その操作的意味を定義することを試みる.

本論文の構成は次のようになっている. まず FGHC のサブセットを Polyadic π 計算へ翻訳する方法について述べる; この翻訳は埋め込み (embedding) と言い換えてもよいだろう. この翻訳方法を検討することは, FGHC の持つ並行計算に関する概念をより単純な PPC の立場から考察することにつながる. 次に, この翻訳により, PPC のプログラムの上で FGHC プログラムの動作の等価性を議論する. ここでは append プログラムと merge プログラムの例を用いて双対なプログラムの動作が等しいことを検証する. 最後に, FGHC の新しい組込み述語の仕様を PPC で記述する. ここでは例として, 共有データをアクセスするプリミティブを FGHC に導入すると, 非決定的マージャと同じ動作が実現できることを示す. さらに, 共有データをアクセスする他のプリミティブも PPC で記述し比較を行う.

2 FGHC^{+-} と Polyadic π 計算

2.1 FGHC

FGHC の操作的意味は上田式遷移システム [9] に従う. 即ち FGHC のゴール :- P. が与えられると, 同一のヘッドリテラルを持つ複数の節 H_i :- $G_i \mid B_i$ において, ガード部 (H_i, G_i) の受動単一化 (passive unification) が並行に行われる. ガード部の単一化に成功した節のどれか 1 つだけが非決定的にコミットオペレータ (|) を通り抜けることができる. ガードを通り抜けた節のボディ部 (B) は, ガード部の単一化で得られた変数具体化で置換され新たなゴール群 $(B\theta)$ となる. 複数のゴール :- $P_1, \cdots, P_k$. の実行は並行に行われる.

FGHC^{+-} は, FGHC の論理変数に対し入出力のモードを付与し, 論理変数に対する操作を単一書込み・複数読出しに制限した well-moded FGHC[11] にさらに多少の制限を加えたものである. 従って FGHC^{+-} では, ボディ部の能動単一化 (active unification) を代入とみなすことができ, 能動単一化は失敗しない[1]. FGHC^{+-} プログラムの標準型では, リテラル, 項のトップレベルには変数しか出現しないと定める (即ち flat form); 従って構造は必ず変数を介して参照される. FGHC^{+-} によるプログラム例を示す:

$$
\begin{array}{lll}
nrev(X^i, Y^o) \text{ :- } X^i = nil & \mid & Y^o = nil. \\
nrev(X^i, Y^o) \text{ :- } X^i = cons(H^m, T^o) & \mid & nrev(T^i, R^o), append(R^i, C^i, Y^o), \\
 & & C^o = cons(H^{\bar{m}}, N^i), N^o = nil.
\end{array}
$$

本論文では, リストを構成する [|] や [ ] を陽に関数記号を用いて $cons/2$, $nil/0$ などと書くこととする. また各変数にはモード解析の結果として ^{i} (in) と ^{o} (out) が付いている. m, $\bar{m}$ は, 現在は未定だがさらにモードに関する情報が増えると in, out に具体化されるかも知れないモード変数である; ただし $m \neq \bar{m}$ である.

2.2 Polyadic π 計算

次に Polyadic pi 計算 (PPC)[5] について概説する. まずベースとなる π 計算があり, 次のような構文規則を持つ:

$$P ::= \textstyle\sum_{i \in I} \pi_i.P_i \mid P|Q \mid !P \mid (\nu x)P \mid \mathcal{A}(x) \mid \mathbf{0}$$

ここでプロセス $\pi.P$ の π は prefix と呼ばれ, atomic action を表わしている. この prefix には 2 つの基本型がある: $x(y_1 \cdots y_n)$, $\bar{x}y_1 \cdots y_n$; これらがそれぞれ polyadic 入力と出力を表わしている[2]. PPC に変換したプログラムは PIC システム[3] で実際に実行して確認する. しかし PIC では複製子 (replicator, !) をサポートしていないために, PPC への翻訳に際して複製子は導入せずに, ガードされた再帰呼出しを導入する.

---

[1] FGHC+$Mode - Unification\ Failure$ より FGHC^{+-} と (とりあえず) 命名した.

[2] 本論文では [6] の構文を一部取り入れて, 以降 $x([y_1, \cdots, y_n])$, $\bar{x}[y_1, \cdots, y_n]$ と記述する.

[3] Polyadic π 計算を基礎に設計されたプログラミング言語 [6]. 本研究には PIC 1.8 版を用いている.

PPC では, 構造同値性 (Structural Congruence, $\equiv$ で表わす) という同値関係があり, 意味的に等しい PPC の項は構文上も等しいと見なすための規則により表現される. 構造同値性の例を 2 つあげる:

もし x が P 中で自由出現していなければ $(\nu x)P \equiv P$.
もし $P \equiv (\nu y)\bar{x}y$ ならば, $x(z).\bar{y}z \mid !P \equiv (\nu y')(x(z).\bar{y}z \mid \bar{x}y') \mid !P$.

そして PPC では, 簡約則 (reduction rules) によって, 通信による状態遷移 ($\longrightarrow$ で表わす) が表現される. 遷移の例を 2 つあげる:

$\bar{x}y \mid x(u).\bar{u}v \mid \bar{x}z \longrightarrow \bar{y}v \mid \bar{x}z$ あるいは $\bar{x}y \mid \bar{z}v$.
$(\nu xw)(\bar{x}y \mid (x(u).\bar{u}v + w(u))) \mid \bar{x}z \longrightarrow \bar{y}v \mid \bar{x}z$.

3 FGHC^{+-} の Polyadic π 計算への変換

本節では, FGHC を構成するいくつかの基本操作を PPC でどのように実現すればよいかを考察する.

3.1 項

FGHC の 0-ary (アトム) も含む関数記号は, 文献 [5] に従って翻訳する. 例えばリストを構成する *cons* や *nil* を翻訳する場合は
$\{TERM \mapsto (CONS, NIL),\ CONS \mapsto (TERM, TERM),\ NIL \mapsto ()\}$ のようなソーティングを考える. これより以下のように翻訳する:

$$cons[t,x,y] \stackrel{\text{def}}{=} t([c,n]).\bar{c}[x,y]$$
$$nil[t] \stackrel{\text{def}}{=} t([c,n]).\bar{n}()$$

FGHC における項が PPC のプロセスに翻訳されているが, これらを項プロセスと呼ぶ. 項プロセスは内容を 1 回読み出されたら終了する. ここで, タプル $[c,n]$ は, その FGHC^{+-} プログラムに出現する関数記号全部の個数分以上の長さを持ち, ソートタプルと呼ぶこととする. ここでは *cons* と *nil* しかないので, 長さは 2 である.

さらに, atomic なデータとして整数を持つようなプログラムに対して, Milner の Numeral コーディング [5] を採用しても良いが, 本論文では便宜上 $INT \mapsto (V)$ というソーティングを新たに加えることで対処した. 従って整数は次のような項プロセス (関数記号) に翻訳される:

$$int[t,v] \stackrel{\text{def}}{=} t([c,n,i]).\bar{i}v$$

3.2 論理変数と能動単一化

FGHC^{+-} の論理変数の機能は以下のようである:

- 高々 1 回だけ書き込まれた値を 0 回以上読み出す (複製)

- 書き込みより先に発行された読み出しは待たされる (同期)

さらに FGHC^{+-} では出現する論理変数の in, out のモードが決まっているので, チャネル w から書き込みを 1 回だけ受け付け (これは PPC の + 構文で保証される), チャネル r から読み出される回数分だけ項プロセスの内容を複製するように翻訳を行う:

$$\begin{aligned} var[w,r] &\stackrel{\text{def}}{=} (\nu cn)(\bar{w}[c,n] \mid (c([x,y]).rep\_cons[r,x,y] + n().rep\_nil[r])) \\ rep\_cons[p,x,y] &\stackrel{\text{def}}{=} p([c,n]).(\bar{c}[x,y] \mid rep\_cons[p,x,y]) \\ rep\_nil[p] &\stackrel{\text{def}}{=} p([c,n]).(\bar{n}() \mid rep\_nil[p]) \end{aligned}$$

ここでは関数記号として $cons$ と nil だけ出現すると仮定した. 内容の複製は rep_cons, rep_nil が行う.

Well-moded FGHC (FGHC^{+-}) の標準型 (flat form) における論理変数の出現パターンは次の 3 つに分類できる (以下 H, G, B で各々ヘッド部, ガード部, ボディ部を表わす):

a. ガード部での検査による節選択のため

b. チャネル変数として

c. ゴール間通信のため

この内, **a.** は, ある節がコミットできるかどうかを知るため, ガード部で行う検査に必要な値を読み出すための変数を指す. このパターンの変数は H と G にのみ read-only 変数として出現する; モードは全て in である. このパターンでは, 最初の論理変数の出現で, 他の節で生成された $var[w,r]$ の r チャネルを得て, 節内の他の出現個所ではそれを共有する.

b. のチャネル変数とは, 文献 [11] に従い, FGHC のゴールとサブゴールの間で情報を運ぶチャネルの役割をする論理変数のことを指す. PPC のチャネルとは全く異なる. B での全出現 (1 回以上) のモードが in で, かつ H, G で in 出現か G で out 出現する場合と B で 1 回 out 出現し, かつ H で 1 回 out 出現か G で 1 回 in 出現する場合がある. いずれの場合も, **a.** 同様に $var[w,r]$ の r チャネルか w チャネルを持ち回るだけで十分である.

c. のパターンは一般に, B に含まれる複数のゴールが論理変数を共有して通信を行う場合を指す; 論理変数の機能であるデータの複製と同期を行う必要があるので, $var[w,r]$ を生成しなければならない. その時, B に複数回出現する論理変数のモードは, out (1 回) と in (1 回以上) である. H, G に出現 (高々 1 回) があっても構わない. 例えば以下のような FGHC^{+-} の標準型において, ボディ部で first class のデータ構造 17 を生成する能動単一化を考える:

$$p(X^o) :\text{-}\ true \mid q(X^i), X^o = 17.$$

17 は必ず論理変数 X 経由でアクセスされる. 翻訳の結果生成された $var[w,r]$ の w 経由で 17 が書き込まれ, $p(X)$ と $q(X)$ の X には r が渡され他の節において値が読み出される.

例として, $var[w,r]$ と項プロセスを用いた能動単一化の翻訳を示す:

$$Y^o = X^i, X^o = nil \implies (\nu px_w)(var[p,y_r] \mid var[x_w,p] \mid nil[x_w])$$

ここで FGHC^{+-} の論理変数と PPC で表現された論理変数の対応を見ると, $X^o \Leftrightarrow x_w$, $X^i \Leftrightarrow p$ (即ち x の r チャネル), $Y^o \Leftrightarrow p$ (即ち y の w チャネル) となっている. ボディ部で新しく生成される変数に対する能動単一化は, 単に対応するチャネルを共有するだけでよい. 一方, 他の節で生成されてチャネルだけが渡されてくるような節がある. 例えば以下のようである:

$$u(X^o, Y^i) \text{ :- } true \mid X^o = Y^i.$$

これらの論理変数 X, Y に対する能動単一化は forwarder に翻訳される:

$$X^o = Y^i \implies x_w(t).\bar{y}_r t$$

ここで, 書き込まれる変数 X の翻訳を $var[x_w, x_r]$, 読み出される変数 Y の翻訳を $var[y_w, y_r]$ とした. この節の中では PPC で新たに論理変数を生成する必要はない. つまり, FGHC^{+-} での論理変数の出現は $var[w, r]$ 個々の出現とは対応せず, そのチャネル w, r の出現に対応している.

3.3 述語呼出しと定義節

FGHC^{+-} ボディ部に並んだ述語呼出し $p(\vec{x_p}), \cdots, q(\vec{x_q})$ は, 素直に PPC の並行なプロセス呼出し $p[\vec{x_p}] \mid \cdots \mid q[\vec{x_q}]$ に翻訳される. 述語 $p(\vec{X}, \vec{Y})$ の定義節は次のように翻訳される (ここで $\vec{X}$ は in, $\vec{Y}$ は out にモード付けされていると仮定):

$$p[\vec{x}, \vec{y}] \stackrel{\text{def}}{=} (\nu g)(p_1[\vec{x}, g, \vec{y}] \mid p_2[\vec{x}, g, \vec{y}] \mid \cdots \mid \bar{g}())$$

k 番目の定義節の翻訳が p_k に対応する. p_k の $\vec{x}$ へのデータは $var[w, r]$ が複製して供給する. p_k の $\vec{y}$ に対する出力は, ガードがあるので競合しない. チャネル g は FGHC^{+-} のコミットオペレータに対応している (後述). FGHC^{+-} の再帰呼出しは PPC でも再帰呼出しに翻訳される.

3.4 ガード部

FGHC^{+-} ガード部の受動単一化は, 入力された項プロセスを分解するだけであり, 外部の変数に対する具体化は行わない (副作用がない). 受動単一化は, 項プロセスと対称な (逆の) 通信を行うように翻訳すればよい. 例えば $a(X^i)$:- $X^i = cons(H, T) \mid \cdots$ という節のガード部は以下のように翻訳する:

$$a[x, g] \stackrel{\text{def}}{=} (\nu cn)(\bar{x}[c, n] \mid c([h, t]).g().\cdots) \tag{1}$$

また例えば変数同士の受動単一化を含む $a(X_1^i, X_2^i)$:- $X_1^i = X_2^i \mid \cdots$ という節は以下のように翻訳すればよい:

$$a[x_1,x_2,g] \stackrel{\text{def}}{=} (\nu s)(matcher[x_1,x_2,s] \mid s().g().\cdots) \tag{2}$$

$$\begin{aligned} matcher[a,b,s] \stackrel{\text{def}}{=}\ & (\nu c_a n_a i_a c_b n_b i_b)(\bar{a}[c_a,n_a,i_a] \mid \bar{b}[c_b,n_b,i_b] \mid \\ & ((c_a([x_a,y_a]).c_b([x_b,y_b]).(\nu s_x s_y)(matcher[x_a,x_b,s_x] \\ & \mid matcher[y_a,y_b,s_y] \\ & \mid s_x().s_y().\bar{s}())) \\ & + (n_a().n_b().\bar{s}()) \\ & + (i_a(v_a).i_b(v_b).(\nu b\_)(=[v_a,v_b,b] \mid b(b_r).\mathtt{if}[b_r,s,\_])))) \end{aligned}$$

ここでは, ソートは $cons$, nil, int の 3 種類あるとした. $=[v_a,v_b,b]$, $\mathtt{if}[b_r,t,f]$ は PIC システムの組込みプロセスである. また, 簡単のために, ガード部には高々 1 つの受動単一化あるいはテスト述語が出現すると仮定している. 受動単一化の suspend/resume は, 変数に対して読み出しを行った時, w チャネルに書き込み (具体化) が行われるまで待たせることで実現する. また我々の翻訳では, 変数が具体化されるまで内容が読めず待たされるので, ガード部における未定義変数同士の受動単一化 (例えば, 未定義変数 X に対する $X = X$ という受動単一化) も suspend してしまう[4].

FGHC の 1 つの特徴である非決定性とは,

- ガード部の条件判定に十分な具体化が得られたら節を選べる,
- どちらの節も選べるならどちらの節を選んでもよい,
- どれかの節を選べるならいつかは必ずどれか選ばなくてはいけない,
- 選ばれなかった節は副作用を起こしてはいけない,

ということを意味するが, 我々の翻訳はこの意味において正しく動作する.

コミットオペレータの翻訳は (1), (2) のように, PPC プログラムの対応する位置に $g()$ を置くだけである. ガードというのは共有資源 (上述の $\vec{Y}$, $\vec{y}$) に対する排他制御の役割を果たしており, 排他制御すべきスレッドが節の形で並置されているともみなせる[5]. 通常, 排他制御はガード部の条件判定とコミットオペレータで実現されているが, 決定的な述語ならコミットオペレータがなくとも条件判定だけで排他制御が実現できる. 従って, 条件判定だけでは排他制御できない述語のためにコミットオペレータが存在していると考えられる. これより我々は, FGHC^{+-} のコミットオペレータを, critical region に入ったことを知らせるシグナルを非決定的に 1 回だけ通信する操作に翻訳する. 一方, PPC は $\pi_1.P_1 + \cdots + \pi_n.P_n$ という構文 (ここで π_i は $x(y)$ か $\bar{x}y$) を持ち非決定的な選択を表現することができるが, 我々の翻訳においてこの強力な構文を用いる必然性はない.

---

[4] ボディ部の $X = X$ という能動単一化は ill-moded である.

[5] 共有資源を持たない非決定的な述語は, 独立な 2 つのプロセスに置き換えられる.

3.5 append プログラムの翻訳

本節では, ストリーム通信を行う FGHC^{+-} プログラムを PPC プログラムに翻訳する. 整数を要素とするリストを生成し ($intList/2$), それを append する ($app/3$) プログラムを取り上げる.

元の FGHC^{+-} プログラム:

$$
\begin{aligned}
&:\text{-}\ intList(N^i, L^o), app(L^i, Y^i, Z^o), \cdots \\
&intList(N^i, L^o) :\text{-}\ N^i =< 0 \ \mid\ L^o = nil. \\
&intList(N^i, L^o) :\text{-}\ N^i > 0 \ \mid\ L^o = cons(N^i, \underline{T^i}), intList(N{-}1^i, \underline{T^o}). \\
&app(X^i, Y^i, Z^o) :\text{-}\ X^i = nil \ \mid\ Y^i = Z^o. \\
&app(X^i, Y^i, Z^o) :\text{-}\ X^i = cons(E^m, Xs^o) \ \mid\ Z^o = cons(E^{\bar{m}}, \underline{T^i}), app(Xs^i, Y^i, \underline{T^o}).
\end{aligned}
$$

翻訳された PPC のプログラム:

$$
\begin{aligned}
\text{トップレベル}:\ & (\nu l_w l_r n_w n_r z_w z_r)(intList[n_r, l_w] \mid var[n_w, n_r] \mid int[n_w, n] \\
& \qquad \mid var[l_w, l_r] \mid app[l_r, y_r, z_w] \mid var[z_w, z_r] \mid \cdots) \\
intList[x, l] \stackrel{\text{def}}{=}\ & (\nu g)((\nu cni)(\bar{x}[c, n, i] \\
& \qquad \mid i(k).(\nu bt\_)(\texttt{=<}[k, 0, b] \mid b(b_r).\texttt{if}[b_r, t, \_] \mid t().g().nil[l])) \\
& \quad \mid (\nu cni)(\bar{x}[c, n, i] \\
& \qquad \mid i(k).(\nu bt\_)(\texttt{>}[k, 0, b] \mid b(b_r).\texttt{if}[b_r, t, \_] \\
& \qquad\quad \mid t().g().(\nu t_r t_w n_w n_r)(cons[l, x, t_r] \mid var[t_w, t_r] \Longleftarrow \star 1 \\
& \qquad\qquad \mid int[n_w, k{-}1] \mid var[n_w, n_r]\dagger \\
& \qquad\qquad \mid intList[n_r, t_w]))) \\
& \quad \mid \bar{g}()) \\
app[x, y, z] \stackrel{\text{def}}{=}\ & (\nu g)((\nu cni)(\bar{x}[c, n, i] \mid n().g().fw[z, y]) \\
& \quad \mid (\nu cni)(\bar{x}[c, n, i] \\
& \qquad \mid c([e, x_s]).g().(\nu t_r t_w)(cons[z, e, t_r] \mid var[t_w, t_r] \mid app[x_s, y, t_w])) \\
& \quad \mid \bar{g}())
\end{aligned}
$$

ここで $fw[z, y] \stackrel{\text{def}}{=} z(t).\bar{y}t$ (forwarder) であり, $\texttt{=<}[a, b, r]$, $\texttt{>}[a, b, r]$ は PIC システムの組込みプロセスである. FGHC^{+-}, PPC とも呼出しの引数の位置に N–1, k–1 とは書けないが簡単のためここでは略記する. $var[w, r]$ は基本的に第 3.2 節の翻訳方法に従いコーディングする. ここで $var[w, r]$ が生成されるのは元の FGHC^{+-} プログラムにおいて __ (下線) を付した変数である.

また第 3.1 節で述べたように, 整数は便宜的に $int/1$ という関数記号を用いて表現されている. 従ってここで用いられる $var[w, r]$ は, $cons$, nil, int の 3 つの関数記号が読み書きできるように 3 要素のソートタプルを持つ必要がある. さらに, 整数がボディ述語の引数として in

モードで出現する時には, 他の関数記号と同様の扱いをする必要があり, PPC への翻訳において $var[w,r]$ を生成する (上のプログラムにおける $var[n_w,n_r]$†).

4 双対なプログラムの PPC への翻訳

FGHC プログラムにおいて, そのアルゴリズムには変更を加えず, データを運ぶメッセージとプロセスの役割を入れ換えるだけの変換を双対変換 [3] と呼ぶ. 直観的には双対変換を施してもプログラムの意味は不変であると考えられるが, まだ形式的には証明されていない. 本章では双対な FGHC^{+-} プログラムを PPC に翻訳し, 双対変換されたプログラム間の等価性を考察する.

4.1 append の双対変換

第 3.5 節と同じ例題に対して, $app/3$ の第 1 引数に関して双対なプログラムを示す:

$:\!-\ intList(N^i,L^i), L^o = app(Y^i,Z^o),\cdots$

$intList(N^i,L^i) :\!-\ N^i =< 0 \ \mid\ p\_nil(L^i).$

$intList(N^i,L^i) :\!-\ N^i > 0 \ \mid\ p\_cons(L^i,N^i,T^o), intList(N{-}1^i,T^i).$

$p\_nil(X^i) :\!-\ X^i = app(Y^o,Z^i) \ \mid\ Y^i = Z^o.$

$p\_cons(X^i,E^m,Xs^o) :\!-\ X^i = app(Y^o,Z^i) \ \mid\ Z^o = cons(E^{\bar{m}},T^i), Xs^o = app(Y^i,T^o).$

関数記号 $cons, nil$ を述語に変換した際, 便宜上, 述語名, プロセス名を p_cons, p_nil とした. 以下は PPC への翻訳である.

$$
\begin{aligned}
\text{トップレベル}: \quad & (\nu l_w l_r n_w n_r z_w z_r)(intList[n_r, l_r] \mid var[n_w, n_r] \mid int[n_w, n] \\
& \qquad \mid var[l_w, l_r] \mid app[l_w, y_r, z_w] \mid var[z_w, z_r] \mid \cdots) \\
intList[x, l] \stackrel{\text{def}}{=} \quad & (\nu g)((\nu cni)(\bar{x}[a, c, n, i] \\
& \qquad \mid i(k).(\nu bt\_)(=<[k, 0, b] \mid b(b_r).\mathtt{if}[b_r, t, \_] \mid t().g().p\_nil[l])) \\
& \quad \mid (\nu cni)(\bar{x}[a, c, n, i] \\
& \qquad \mid i(k).(\nu bt\_)(>[k, 0, b] \mid b(b_r).\mathtt{if}[b_r, t, \_] \\
& \qquad\qquad \mid t().g().(\nu t_r t_w n_w n_r)(p\_cons[l, x, t_w] \mid var[t_w, t_r] \Longleftarrow \star 2 \\
& \qquad\qquad\qquad \mid int[n_w, k\text{-}1] \mid var[n_w, n_r] \\
& \qquad\qquad\qquad \mid intList[n_r, t_r]))) \\
& \quad \mid \bar{g}()) \\
app[x, y, z] \stackrel{\text{def}}{=} \quad & x([a, c, n, i]).\bar{a}[y, z] \\
p\_nil[p] \stackrel{\text{def}}{=} \quad & (\nu g)((\nu acni)(\bar{p}[a, c, n, i] \mid a([y, z]).g().fw[z, y]) \\
& \quad \mid \bar{g}()) \\
p\_cons[p, e, x_s] \stackrel{\text{def}}{=} \quad & (\nu g)((\nu acni)(\bar{p}[a, c, n, i] \\
& \qquad \mid a([y, z]).g().(\nu t_r t_w)(cons[z, e, t_r] \mid var[t_w, t_r] \mid app[x_s, y, t_w])) \\
& \quad \mid \bar{g}())
\end{aligned}
$$

$intList[n, l]$ に関しては, 第 3.5 節のプログラムの $\star 1$ を $\star 2$ のように, 変数に対するモードを逆転させるだけでよい. トップレベルに関しても, 変数のモードを逆転させれば良い. さらにこの場合は p_cons, p_nil というプロセス名への書き換えも必要である. ここでソートタプルが $[a, c, n, i]$ となっている点に注意; a は双対変換後の FGHC^{+-} プログラムにおける関数記号の $app/2$ に対応している.

元の PPC プログラムでは $cons$, nil という項プロセスがデータを供給し, データを受け取った app が計算を行っていた. 双対変換されたプログラムでは逆に app が項プロセスとしてデータを供給し, p_cons, p_nil の中で計算が行われる. 元の $intList/2$ での $L^o = cons(N^i, T^i)$ と双対変換後の $p\_cons(L^i, N^i, T^o)$ という述語呼出しが, PPC ではテキスト上は全く同じプロセス呼出しに翻訳されている点に注意; ただし l, t に関するデータ送受のモードは逆である. また, データ供給の向きを逆転させるために書き換えられたのは各述語のガード部に対応する部分だけであり, ボディ部は変数のモード逆転を除いて変更はない.

PPC において, この元プログラムと双対なプログラムでは同じ振舞いが観測できる. 例として, 次の 2 つの翻訳の reduction を行う:

$$
\begin{aligned}
&\text{元の append プログラムの翻訳:} \quad (\nu x_w x_r y z d \cdots)(app[x_r, y, z] \mid var[x_w, x_r] \mid cons[x_w, n, d] \mid \cdots) \\
&\text{双対な append プログラムの翻訳:} \quad (\nu x_w x_r y z d \cdots)(app[x_w, y, z] \mid var[x_w, x_r] \mid p\_cons[x_r, n, d] \mid \cdots)
\end{aligned}
$$

これらはともに $\longrightarrow^{\star} (\nu t_r t_w)(cons[z,n,t_r] \mid var[t_w,t_r] \mid app[d,y,t_w] \mid \cdots)$ となり，チャネル z に対して同じプロセス構造を生成することが分かる．

4.2 merge の双対変換

非決定性を持つ述語の 1 つとして $merge/3$ を PPC プログラムに変換する．まず元となる FGHC^{+-} で記述されたメッセージストリームをマージするプログラムは以下のようである:

$$
\begin{array}{lll}
merge(X^i,Y^i,Z^o) \text{ :- } X^i = cons(E^m, Xs^o) & \mid & Z^o = cons(E^{\bar{m}}, Zs^i), merge(Xs^i, Y^i, Zs^o). \\
merge(X^i,Y^i,Z^o) \text{ :- } Y^i = cons(E^m, Ys^o) & \mid & Z^o = cons(E^{\bar{m}}, Zs^i), merge(X^i, Ys^i, Zs^o). \\
merge(X^i,Y^i,Z^o) \text{ :- } X^i = nil & \mid & Z^o = Y^i. \\
merge(X^i,Y^i,Z^o) \text{ :- } Y^i = nil & \mid & Z^o = X^i.
\end{array}
$$

これは，以下のような PPC プログラムに翻訳される:

$$
\begin{array}{lll}
merge[x,y,z] \stackrel{\text{def}}{=} & (\nu g)((\nu cn)(\bar{x}[c,n] \mid c([e,x_s]).g().(\nu rw)(cons[z,e,r] & \\
& \qquad \mid var[w,r] \mid merge[x_s,y,w])) & \Longleftarrow \star 3 \\
& \mid (\nu cn)(\bar{y}[c,n] \mid c([e,y_s]).g().(\nu rw)(cons[z,e,r] & \\
& \qquad \mid var[w,r] \mid merge[x,y_s,w])) & \\
& \mid (\nu cn)(\bar{x}[c,n] \mid n().g().fw[z,y]) & \\
& \mid (\nu cn)(\bar{y}[c,n] \mid n().g().fw[z,x]) & \\
& \mid \bar{g}()) &
\end{array}
$$

現在の双対変換の枠組みでは，非決定性を持つ FGHC^{+-} の述語は双対変換できない．そこで，前出の $app/3$ プログラム例にならって，PPC レベルで双対変換 (のような書き換え) を施す．$merge/3$ の第 1 引数に関して双対変換を行い，そのデータ供給の方向を逆転させる:

$$\text{トップレベル:}\quad (\nu x_w x_r y_w y_r z t_w t_r)(merge[x_w, y_r, z] \mid var[y_w, y_r] \mid intList[y_w, n]$$
$$\mid var[x_w, x_r] \mid p\_cons[x_r, e, t_w] \mid var[t_w, t_r] \mid p\_nil[t_r] \mid \cdots)$$

$$merge[x, y, z] \stackrel{\text{def}}{=} (\nu g)(x([m, c, n]).\bar{m}[y, z].g() \qquad \Longleftarrow \star 4$$
$$\mid (\nu mcn)(\bar{y}[m, c, n] \mid c([e, y_s]).g().(\nu rw)(cons[z, e, r]$$
$$\mid var[w, r] \mid merge[x, y_s, w]))$$
$$\mid (\nu mcn)(\bar{y}[m, c, n] \mid n().g().fw[z, x]) \qquad \Longleftarrow \star 5$$
$$\mid \bar{g}())$$

$$p\_cons[x, e, x_s] \stackrel{\text{def}}{=} (\nu g)((\nu mcn)(\bar{x}[m, c, n] \mid m([y, z]).g().(\nu rw)(cons[z, e, r]$$
$$\mid var[w, r] \mid merge[x_s, y, w])) \Longleftarrow \star 6$$
$$\mid \bar{g}())$$

$$p\_nil[p] \stackrel{\text{def}}{=} (\nu g)((\nu mcn)(\bar{p}[m, c, n] \mid m([y, z]).g().fw[z, y]) \qquad \Longleftarrow \star 7$$
$$\mid \bar{g}())$$

双対変換後のトップレベルでも同様に, 第 1 引数を受ける論理変数に対応する $var[w, r]$ の w, r の方向が逆転している. 双対変換前には入力の分解及び非決定的なコミットが $merge/3$ のガード部で行われていたが, PPC の $merge/3$ の第 1 引数に関する双対変換を行うと, 元のボディ部 ($\star 3$) の部分は, $\star 4$ のように項プロセスのボディ部と入れ換わる. 一方, 元のボディ部は p_cons ($\star 6$) と p_nil ($\star 7$) のボディ部に分かれて移動している.

この双対な $merge/3$ は[6], z への多重書き込みが生じて動作しない. 条件判定だけでは排他制御できない非決定的な述語に双対変換を施すと, 排他制御すべき操作が $merge/3$ のガードにかからない所に出現してしまうからである (この場合 p_cons, p_nil の $g()$ 以降のボディ部). 例えば, $\star 4$ でソート m の情報を出力し, 決定的に p_cons か p_nil のボディ部で共有資源 z への書き込みが生じているにもかかわらず, $\star 5$ の $g()$ が先にシグナルを受けとる可能性がある. 従って, 非決定性の満たすべき性質である, $merge/3$ の 4 つの定義節のどれも選べない時はどれか選べるようになるまで (副作用を起こさずに) 待たねばならず, かつ選ばれなかった節は共有資源に対して副作用を起こしてはならない, ということが満たされていない. また $\star 4$ の行が $g().x([m, c, n]).\bar{m}[y, z]$ となっていても, 同様に動作しない.

このような場合, $\star 4$ で出力された情報をバックトラックによりキャンセルできれば良いのだが, 我々の翻訳の枠組みではできない. また, 異なる述語の定義節間で排他制御を行うような仕組みがあっても良いのだが, FGHC^{+-} は言語としてそのような機能を備えていない.

---

[6] これは間違った変換なので本当は双対ではない.

5 新しい組込み述語の仕様記述

5.1 共有データ操作のための組込み述語

FGHC^{+-} に共有データ操作のための新しいプリミティブ $swap\_shared\_data(SD_0, Old, New, SD_1)$ ($swap\_sd$ と略す) を導入することを考える[7]. 本述語の引数 SD_0, SD_1 は共有データへの参照 (共有データの存在する場所) を表す. SD_0 に V という値が記憶されている時, $swap\_sd$ を実行すると, SD_0 上の V は New に置き換えられ, V は Old から読み出される. この一連の swap 操作は排他的に実行され, swap が終了すると SD_1 に新しい共有データへの参照が具体化される. KL1 の $set\_vector\_element/5$ [10] と異なるのは, SD_0, SD_1 が共有されていてもコピーを生成せず破壊代入を行い副作用を残すことである.

この $swap\_sd$ と, 共有データをアロケートする述語 $alloc\_sd$ の仕様を PPC で記述する.

$$
\begin{aligned}
swap\_sd[p,o,u,q] &\stackrel{\text{def}}{=} (\nu cnis)(\bar{p}[c,n,i,s] \\
&\qquad \mid s(m).(\nu r)(\bar{m}[r,u].(r(v).fw[v,o] \mid q([c,n,i,s]).\bar{s}m))) \Longleftarrow \star 8 \\
alloc\_sd[p] &\stackrel{\text{def}}{=} p([c,n,i,s]).(\nu m\_)(\bar{s}m \mid swap[m,\_]) \\
swap[m,v] &\stackrel{\text{def}}{=} m([r,u]).(\bar{r}v \mid swap[m,u])
\end{aligned}
$$

ソートタプル $[c,n,i,s]$ に, 新しいソートに対応して s という要素が増えたが, チャネル s 上で送受されるデータは常に $alloc\_sd$ で最初に定義された m (定数) である点に注意. この $swap\_sd$ のモードは o が in で u が out である. 逆の o が出力で u が入力であるような $swap\_sd$ は, $\star 8$ の所を $fw[v,o] \Rightarrow fw[o,v]$ とすればよい. また, 共有メモリを複数セルに拡張するのも容易である.

5.2 merge プログラムの実現

$swap\_sd$ 述語を用いると, FGHC^{+-} の節間の非決定性を利用せずに非決定的マージャが記述できる[8]:

$$
\begin{aligned}
&merge(X^i, Y^i, Z^o) \text{ :- } true \mid alloc\_sd(SD_0^o), swap\_sd(SD_0^i, dum^i, Z^o, SD_1^o), \\
&\qquad\qquad mg\_in(X^i, SD_1^i), mg\_in(Y^i, SD_1^i). \\
&mg\_in(X^i, \_^i) \text{ :- } X^i = nil \mid true. \\
&mg\_in(X^i, S_0^i) \text{ :- } X^i = cons(H^m, T^o) \mid swap\_sd(S_0^i, A^i, R^o, S_1^o), A^o = cons(H^{\bar{m}}, R^i), \\
&\qquad\qquad mg\_in(T^i, S_1^i).
\end{aligned}
$$

この $swap\_sd$ を用いた PPC マージャのプログラムを示す:

---

[7] 本組込み述語のオリジナルアイデアは久門耕一氏 (ICOT) による.

[8] $swap\_sd$ を用いたマージャは正しく双対変換を行うことができる.

$$
\begin{aligned}
merge[x,y,z] \stackrel{\text{def}}{=}\ & (\nu g)(g().(\nu s_{0w}s_{0r}s_{1w}s_{1r}\_)(alloc\_sd[s_{0w}] \mid var[s_{0w},s_{0r}] \\
& \qquad \mid swap\_sd[s_{0r},dum,z,s_{1w}] \mid var[s_{1w},s_{1r}] \\
& \qquad \mid mg\_in[x,s_{1r}] \mid mg\_in[y,s_{1r}]) \\
& \mid \bar{g}()) \\
mg\_in[x,p] \stackrel{\text{def}}{=}\ & (\nu g)((\nu cnis)(\bar{x}[c,n,i,s] \mid n().g()) \\
& \mid (\nu cnis)(\bar{x}[c,n,i,s] \\
& \qquad \mid c([h,t]).g().(\nu q_w q_r a_w a_r r_w r_r)(swap\_sd[p,a_r,r_w,q_w] \mid var[q_w,q_r] \\
& \qquad\qquad \mid fw[a_r,a_w] \\
& \qquad\qquad \mid cons[a_w,h,r_r] \mid var[r_w,r_r] \Longleftarrow \star 9 \\
& \qquad\qquad \mid mg\_in[t,q_r])) \\
& \mid \bar{g}())
\end{aligned}
$$

ここで, 単一書込み単一読出しであることが分かっているような変数 (この場合 mg_in の A) の場合は, $\star 9$ には本来なら $var[a_w,a_r]$ が来る筈であるが, 値の複製をする必要がなく同期だけをとれば良いので $fw[a_r,a_w]$ $(= a_r(t).\bar{a_w}t)$ で十分機能する.

本節で定義した $merge/3$ が, $swap\_sd$ を用いない従来のマージャと同じ動作をすることは, 第 4.1 節同様に, PPC 上で reduction $(\rightarrow^*)$ を行うことで確認できる[9].

5.3 PPC による Port の仕様記述

並行論理型言語における共有データに対するアクセスプリミティブは, $swap\_sd$ の他にも, port[1], mutual reference cell[7] などがあり, これらを PPC で記述することで統一的に比較することが可能となる.

ここでは port を取り上げる. port を生成するプリミティブは $open\_port/2$ であり, port に 1 つのデータを送信するプリミティブは $send/2$ である. 以下のように動作する[10]:

$$:\text{-}\ open\_port(P,S), send(P,2), send(P,3), send(P,5), \cdots.$$

この時 S は非決定的に $[2,3,5,\cdots], [3,2,5,\cdots], [3,5,2,\cdots]\cdots$ のいずれかに具体化される. これら各プリミティブは FGHC^{+-} の $alloc\_sd$, $swap\_sd$ で以下のように定義することができるだろう:

---

[9] ストリームが nil で閉じられた場合も含めると従来の $merge/3$ とは厳密には等しくない. 等しくするために, マージャの入力本数を記録するカウンタを $swap\_sd$ で実現する方法と, 全ての mg_in プロセスが終了したことを short circuit で検出する方法がある.

[10] port の仕様では, ある port への参照がなくなるとシステムがそれを検出し, 出力ストリームを自動的に閉じることになっているが, ここでは簡単のため考慮しない.

$$
\begin{aligned}
open\_port(P^o, Z^o) &\equiv alloc\_sd(S^o), swap\_sd(S^i, dum^i, Z^o, P^o) \\
send(P^i, E^m) &\equiv swap\_sd(P^i, A^i, B^o, \_^o), A^o = cons(E^m, B^i)
\end{aligned}
$$

上述の FGHC^{+-} における $open\_port/2$ と $send/2$ の定義が正しいかどうかを検証するために, PPC に翻訳し reduction を行う:

$$
\begin{aligned}
open\_port[p, z] &\stackrel{\text{def}}{=} (\nu m)(p([c, n, i, s]).\bar{s}m \mid swap[m, z]) \\
send[p, e] &\stackrel{\text{def}}{=} (\nu a_w a_r b_w b_r)(swap\_sd[p, a_r, b_w, \_] \\
&\qquad \mid fw[a_r, a_w] \mid cons[a_w, e, b_r] \mid var[b_w, b_r])
\end{aligned}
$$

これらの定義より

$$
\begin{aligned}
&(\nu p_w p_r)(open\_port[p_w, z] \mid var[p_w, p_r] \mid send[p_r, e_1] \mid send[p_r, e_2]) \\
&\longrightarrow^\star (\nu m p_r a_w a_r b_w b_r)(fw[z, a_r] \mid fw[a_r, a_w] \mid cons[a_w, e_1, b_r] \mid var[b_w, b_r] \\
&\qquad \mid swap[m, b_w] \mid rep\_sd[p_r, m] \mid send[p_r, e_2]) \\
&\longrightarrow^\star (\nu m p_r a_w a_r b_w b_r c_w c_r d_w d_r) \\
&\qquad (fw[z, a_r] \mid fw[a_r, a_w] \mid cons[a_w, e_1, b_r] \mid var[b_w, b_r] \\
&\qquad \mid fw[b_w, c_r] \mid fw[c_r, c_w] \mid cons[c_w, e_2, d_r] \mid var[d_w, d_r] \\
&\qquad \mid swap[m, d_w] \mid rep\_sd[p_r, m]) \\
&\longrightarrow^\star (\nu a_w a_r b_r d_w d_r) \quad (fw[z, a_r] \mid fw[a_r, a_w] \mid cons[a_w, e_1, b_r] \mid \\
&\qquad \mid rep\_cons[b_r, e_2, d_r] \mid var[d_w, d_r])
\end{aligned}
$$

ここで, $var[p_w, p_r]$ の中から呼び出される $rep\_sd[p_r, m]$ プロセスが値 m を複製すると仮定した.

mutual reference cell に関しても, それを生成するプリミティブ ($allocate\_mutual\_reference/2$) とアクセスするプリミティブ ($stream\_append/3$) を PPC で記述することができる. この時, $open\_port/2$ の仕様と $allocate\_mutual\_reference/2$ の仕様は全く同じであり, $stream\_append/2$ の仕様と $send/2$ の仕様は非常によく似ていることが分かる.

6 おわりに

本論文では, まず FGHC を π 計算に翻訳 (embedding) する方法を非形式的に述べた. 次にその応用として, 双対変換はプログラムの意味を変えない変換であることの検証, 共有変数を破壊的に書き換える組込み述語の仕様定義を行った. これらの実例より, 本翻訳の応用範囲の広さを示すことができた.

関連研究としては, まず [4] がある. Prolog の実行における, バックトラックを含む述語の実行制御 (SLDNF resolution) とバックトラック付き単一化を, π 計算にコーディングする方法を示した. この研究に用いられている π 計算では, チャネル同士の等価性をテストするプリ

ミティブや, 型検査のプリミティブの存在を仮定している. また, 論理変数が未定義であることはタグにより判定している. WAM による実行を比較的素直に表現していると言えよう.

本研究は, プログラミング言語 Oz [2] とその意味論 [8] の研究にも重要な関連がある. Oz は, 並行制約, オブジェクト指向, 高階という特徴をもった言語である. その意味論の考察を通して γ 計算という新しい並行計算モデルが得られている. γ 計算は, 大まかに言うと, π 計算にプリミティブとして論理変数を加えたようなものである. Smolka らのアプローチ [8] は, 表現力の高い言語の意味を記述するためにはモデルを拡張する必要がありそのために新しいプリミティブを導入するというものである. 一方, 我々のアプローチはその逆で, PPC に埋め込むことのできる FGHC のサブセット FGHC^{+-} を考えて, その範囲で翻訳を行うというものである.

以下, 今後の課題を述べる.

矛盾なくモードが付与できる FGHC (FGHC^{+-}) から PPC への我々の翻訳方法をもっと形式的に記述する必要がある. そして, 上田式遷移システムに関して我々の翻訳が正しいことを示すことができれば, 翻訳の結果得られた PPC プログラムを FGHC^{+-} の操作的な意味と見なすことができる. ただし, 上田式遷移システムで unification failure が生じると無意味な計算状態に陥り, reduction failure が生じると suspension (deadlock) に陥る. 一方, 我々の翻訳の結果生成された PPC プログラムでは, これら failure が区別できず全て deadlock してしまう. しかし, FGHC^{+-} に制限することで unification failure を回避することができると考えている.

また, 翻訳された PPC プログラムの型は, 元の FGHC^{+-} プログラムの型と関連を持っている. PPC 上の型システムから得られる情報が, 元 FGHC^{+-} プログラムの実行効率化にどの程度寄与するのか, についても調べたいと思う.

謝辞: FGHC のモード解析について, 早稲田大学 上田和紀助教授より大変貴重な助言を数多く頂きました. また本研究を行うにあたり, NTT 基礎研究所 尾内 *K.* 理紀夫氏は著者を常に暖かく励まして下さいました.

参考文献

[1] S. Haridi, S. Janson, J. Montelius and M. Nilsson, Ports for Objects in Concurrent Logic Programs (DRAFT). *unpublished document* (Dec. 1991).

[2] M. Henz, G. Smolka and Jörg Würtz, Oz – A Programming Language for Multi-Agent Systems, In *Proc. of IJCAI'93*, pp.404–409 (1993).

[3] K. Kumon and K. Hirata, A New Transformation Based on Process-Message Duality for Concurrent Logic Languages. In *Proc. of ICLP 1994*, pp.684–698 (1994).

[4] B. Li, A π-calculus Specification of Prolog. In *Proc. of ESOP 1994* (1994).

[5] R. Milner, *The Polyadic π-Calculus: a Tutorial.* ECS-LFCS-91-180, University of Edinburgh (1991).

[6] B. Pierce, *Programming in the Pi-Calculus.* ftp.dcs.ed.ac.uk より anonymous FTP 可能 (1993).

[7] E. Shapiro and S. Safra, Multiway Merge with Constant Delay in Concurrent Prolog. *New Generation Computing*, Vol. 4, pp.211–216 (1986).

[8] G. Smolka, *A Foundation for Higher-order Concurrent Constraint Programming.* RR-94-16, DFKI (1994).

[9] K. Ueda, Designing a Concurrent Programming Language. In *Proc. of InfoJapan'90*, pp.87–94 (1990).

[10] K. Ueda and T. Chikayama, Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, Vol.33, No.6, pp.494–500 (1990).

[11] 上田, 並列論理型言語の言語処理系の効率化に関する調査研究報告書, ICOT (March 1994).