数理解析研究所講究録
1125巻 2000年 172-182

変数の出現条件を用いた融合変換とその反復適用について

Fusion by occurrance analysis and its iterative application

湯浅 能史
Yoshifumi YUASA

早稲田大学 理工学総合研究センター
Advanced Research Center for Science and Engineering, Waseda University

概 要

合成プログラム $P \circ Q$ において、Q で生成され Pで消費される中間データを除去して、より効率的な実行形式に変換することは「プログラムの融合」と呼ばれている。特に Pが再帰子で Qが原始再帰関数の場合には、プログラム中の変数記号の出現状況を静的に分析することで、融合の可能性や変換後の形を特徴付けることができる。この方法によれば、融合変換をプログラムの構造に関する帰納法で定義することができるため、理論的に考察を加える上での見通しも良い。本講演ではこの特徴付けの具体的な形を提示すると共に、与えられた大きなプログラムに対しこれを反復適用して、全ての融合可能部を処理する「大域変換」とその停止性についても考察する。

1 はじめに

合成プログラム $f \circ g$の実行時には、プログラム gが生成するデータがプログラム fに受け渡され消費される、という形で処理が進む。ここで受け渡されるデータ構造のことを**中間データ**という。中間データを除去して二つのプログラムを一体化することは、プログラムの**融合**と呼ばれている。例えば：

$$\begin{aligned} f(n) &= if(n=0)\, then\, [] \, else\, [n|f(n-1)] \\ g(a) &= if(a=[])\, then\, 0\, else\, n+g(b) \\ &\qquad (where\ (a=[n|b])) \end{aligned}$$

とした時 $f \circ g$では中間データ $[1,\cdots,n]$ が gから fへと受け渡される。これを除去して消費者 fを生産者 gに融合すると、以下の h を得る。

$$h(n) = if(n=0)\, then\, 0\, else\, n+h(n-1)$$

プログラム融合の古典的な実現手続き（[3], [4]）としては「展開・畳みこみ法」が知られている。これは、関数評価の実行過程を模倣して関数呼出を展開し、その過程で現れる同じ実行パターンを畳み込んで新しく再帰構造を作成する方法である。しかし畳み込みは常に可能とは限らないため、融合手順の記述は複雑になる。従って、これを反復的に適用する戦略を立てることは困難である。これに対して、近年提案された「Foldr/build 法」（[1]）やこれをカテゴリ表現で一般化した「酸性雨定理」（[2]）では、プログラム中の構成子記号を所定の関数定義で置き換えることで融合を実現しており、変換そのものの記述は極めてシンプルで扱い易い。しかしこれらにおいても、プログラム中の置換すべき構成子を特定する方法は自明ではない。

本稿では対象とするプログラムを原始帰納関数に限定し、「強自由変数」と呼ぶ概念を用いて融合変換を記述する。これは展開・畳み込み法で言えば、畳み込みが成功するための条件を特徴付けることに相当する。また Foldr/build 法との関連では、置換すべき構成子を、項に関する単純な帰納法で特定する手順だと言える。更に本稿後半では、この方法に基づいて、与えられたプログラム中の融合可能部分の全てを融合する大域変換を提示する。

2 言語と表示的意味

本稿で扱うプログラミング言語は、自然数上の原始反復を記述する。この言語は、定数記号 '$\mathbf{0}$' と関数記号 '$\mathcal{S}$' を持つ。また原始反復を表すタグとして '$\mathcal{R}$' を用いる。以下プログラムの事を**項**と呼ぶことにする。項および変数記号の**自由出現**は、以下により帰納的に定義される。

定義 2.1 (項) (i) 文字列 $(\mathbf{0})$ は項であり変数の自由出現を持たない。変数記号 x に対し、文字列 (x) は項であり、これに自由出現する変数記号は x のみである。

(ii) 項 Nに対し、文字列 $(\mathcal{S}N)$ は項である。変数記号 yがこの項に自由出現するのは、項 Nに自由出現する時かつその時に限る。

(iii) 項 M_0, M_1, Nおよび変数記号 x に対し、以下は項である。

$$(\mathcal{R}(M_0, \lambda x M_1)N)$$

変数記号 yがこの項に自由出現するのは、項 $M_0, \lambda x M_1, N$の全てに自由出現する時かつその時に限る。但し yが$\lambda x M_1$に自由出現するとは、これが M_1に自由出現してかつ x と異なることをいう。

変数記号から自然数への有限部分関数を**環境**と呼ぶ。環境 Eにおける変数記号 x の値が kであることを 'xEk' と記する。環境 Eの定義域に項 Mの自由出現変数が全て含まれる時に「Eは Mの環境である」という。この時 Mはある自然数 $[\![M]\!]_E$に対応付けられる。これを環境 E下での Mの値と呼ぶ。

定義 2.2 (表示的意味) (i) 項 Mが $(\mathbf{0})$ の時、この項の値は（環境によらず）零に定める。項 Mが (x) の形で xEkの時、この項の値は kに定める。

(ii) 項 Mが $(\mathcal{S}N)$ の形なら以下の様に定める。

$$[\![M]\!]_E = [\![N]\!]_E + 1$$

(iii) 項 Mが $(\mathcal{R}(M_0, \lambda x M_1)N)$ の形の時、まず Nが (z) で変数 zが M_0と$\lambda x M_1$に自由に現れないケースを定義する。環境 E下での zの付値を n として、項 Mの値 $m(n)$ を n に関する帰納法で定める。

$$\begin{aligned} m(0) &= [\![M_0]\!]_E \\ m(k+1) &= [\![M_1]\!]_{E\cup\{(x,m(k))\}} \end{aligned}$$

一般の Nについては、環境 E下での Nの値を n とし、新変数 zをとって:

$$[\![M]\!]_E = [\![(\mathcal{R}(M_0, \lambda x M_1)(z))]\!]_{E\cup\{(z,n)\}}$$

と定める。

項 M中の自由変数 x に Nを（束縛変数の付け替えをした上で）代入して得られる項を $M[N/x]$ と書くことにすれば、以下が成り立つ。これは Mの構成に関する帰納法で証明される。

補題 2.3

$$[\![M[N/x]]\!]_E = [\![M]\!]_{E\cup\{(x,k)\}} \quad where\ [\![N]\!]_E = k$$

3 強自由変数による融合変換

文献 [1] の方法論によれば、融合変換はデータ生成側の関数定義式に含まれる構成子を所定の関数定義式で置換することで実現される。しかし、全ての構成子を置き換えると、一般には変換の前後で等価性が保たれなくなるので注意が必要である。

置換すべき構成子を選定するには、定義式中の変数記号に着眼する方法が有効である。変数記号の内で、束縛される値が反復子の反復回数に影響を与えないもの達を特徴付け、融合を部分項へと進めるに際しては、最終的にはそれらの変数のみに至るよう配慮する。このような変数を完全に（必要十分条件として）特定するのは不可能なので、適当な十分条件で特徴付ける。本稿ではこれを**強自由変数**と呼び、項の構成に関する帰納法で定義する。

定義 3.1 (強自由変数) (i) 項 Mが $(\mathbf{0})$ なら、全ての変数記号が Mで強自由である。項 Mが (x) の形なら、全ての変数記号が Mで強自由である。

(ii) 項 Mが $(\mathcal{S}N)$ の形なら、項 Nで強自由な変数かつそれらのみが Mで強自由である。

(iii) 項 Mが $(\mathcal{R}(M_0, \lambda x M_1)(N))$ の形の時には、変数 x が部分項 M_1で強自由か否かによって場合分けして定める。強自由の場合、変数 yが Mで強自由なのは、これが M_0と M_1の双方で強自由でありかつ Nに自由出現しない時、かつその時のみである。強自由でない場合、変数 yが Mで強自由なのは、これが Mに自由出現しない時、かつその時のみである。

次に融合変換を定義する。以下では：

$$K = \mathcal{R}(K_0, \lambda y K_1)$$

とし、反復子 Kを項 Mに融合する変換 $\mathcal{F}^K_U$ を定める。但し、ここで Uは Mの強自由変数からなる有限集合である。また各変数記号に対し新しい変数記号$\hat{x}$を導入する。これを x の**随伴変数**と呼ぶ。

定義 3.2 (融合変換) (i) 項 Mが $(\mathbf{0})$ なら:

$$\mathcal{F}_U^K M = K_0$$

と定める。項 Mが (x) の時には

$$\mathcal{F}_U^K M = \begin{cases} (\hat{x}) & (if\ x \in U) \\ (K(x)) & (if\ x \notin U) \end{cases}$$

と定める。

(ii) 項 Mが $(\mathcal{S}N)$ の形の時:

$$\mathcal{F}_U^K M = K_1[\mathcal{F}_U^K N/y]$$

と定める。

(iii) 項 Mが $(\mathcal{R}(M_0, \lambda x M_1)(N))$ の形の時には、変数 x が部分項 M_1で強自由か否かによって場合分けする。強自由の場合には:

$$\mathcal{F}_U^K M = (\mathcal{R}(\mathcal{F}_U^K M_0, \lambda\hat{x}\mathcal{F}_{U\cup\{x\}}^K M_1)(N))$$

と定める。強自由でない場合には:

$$\mathcal{F}_U^K M = (KM)$$

と定める。

4 融合変換の等価性

再帰子 Kを前節の様にとり Uを変数記号の有限集合とする。環境 Fが環境 Eの拡張で以下を満たす時、これは U上で Eの K-**随伴**であると言う。

$$x \in U \Leftrightarrow [\![(\hat{x})]\!]_F = [\![(K(x))]\!]_E$$

定理 4.1 (変換の等価性) 環境 Eを項 (KM) の環境とし Uを Mの強自由変数からなる集合とする。環境 Fが U上で Eの K-随伴なら:

$$[\![\mathcal{F}_U^K M]\!]_F = [\![(KM)]\!]_E$$

が成り立つ。

証明: 項 Mの構成に関する帰納法で証明する。

(i) 項 Mが $(\mathbf{0})$ の時:

$$\begin{aligned} [\![\mathcal{F}_U^K M]\!]_F &= [\![K_0]\!]_F = [\![K_0]\!]_E \\ &= [\![(K(\mathbf{0}))]\!]_E \\ &= [\![(KM)]\!]_E. \end{aligned}$$

項 Mが (x) で $x \in U$の時:

$$\begin{aligned} [\![\mathcal{F}_U^K M]\!]_F &= [\![(\hat{x})]\!]_F = [\![(K(x))]\!]_E \\ &= [\![(KM)]\!]_E. \end{aligned}$$

項 Mが (x) で $x \notin U$の時:

$$\begin{aligned} [\![\mathcal{F}_E^K M]\!]_F &= [\![(K(x))]\!]_F = [\![(K(x))]\!]_E \\ &= [\![(KM)]\!]_E \end{aligned}$$

(ii) 項 Mが $(\mathcal{S}N)$ の形とする。一般性を失わず変数 yが Fで付値されていないと仮定して良い。

$$\begin{aligned} [\![\mathcal{F}_U^K M]\!]_F &= [\![K_1[\mathcal{F}_U^K N/y]]\!]_F \\ &= [\![K_1]\!]_{F\cup\{(y,k)\}} \\ &\quad where\ [\![\mathcal{F}_U^K N]\!]_F = k \\ &= [\![K_1]\!]_{E\cup\{(y,k)\}} \\ &\quad where\ [\![(KN)]\!]_E = k \\ &= [\![(K(\mathcal{S}N))]\!]_E = [\![(KM)]\!]_E \end{aligned}$$

(iii) 項 Mが $(\mathcal{R}(M_0, \lambda x M_1)N)$ の形とする。

変数 x が項 M_1で強自由の場合、まず Nが (z) で変数 zが M_0と $\lambda x M_1$に自由に現れないケースを示す。環境 E下での zの付値 n に関する数学的帰納法で証明する。自然数 n が零の時:

$$\begin{aligned} [\![\mathcal{F}_U^K M]\!]_F &= [\![\mathcal{F}_U^K M_0]\!]_F \\ &= [\![(KM_0)]\!]_E = [\![(KM)]\!]_E \end{aligned}$$

自然数 n が $k+1$ の時、環境 $\bar{E}$と $\bar{F}$を以下で定める。一般性を失うことなく変数 x とその随伴は Fで付値されていないとして良い。

$$\begin{aligned} \bar{E} &= E \cup \{(x, m)\} \\ \bar{F} &= F \cup \{(x, m), (\hat{x}, \bar{m})\} \\ &\quad where\ [\![M]\!]_{E\setminus\{(z,n)\}\cup\{(z,k)\}} = m \\ &\quad\quad [\![\mathcal{F}_U^K M]\!]_{F\setminus\{(z,n)\}\cup\{(z,k)\}} = \bar{m} \end{aligned}$$

帰納法の仮定より $\bar{F}$は $U \cup \{x\}$ 上で $\bar{E}$に K-随伴である。

$$\begin{aligned} [\![\mathcal{F}_U^K M]\!]_F &= [\![\mathcal{F}_{U\cup\{x\}}^K M_1]\!]_{\bar{F}} \\ &= [\![(KM_1)]\!]_{\bar{E}} = [\![(KM)]\!]_E \end{aligned}$$

一般の Nに関しては、環境 E下での(従って F下での) Nの値を n とし、新変数 zにより項 M'を $(\mathcal{R}(M_0, \lambda x M_1)(z))$ と定めれば:

$$\begin{aligned} [\![\mathcal{F}_U^K M]\!]_F &= [\![\mathcal{F}_U^K M']\!]_{F\cup\{(z,n)\}} \\ &= [\![(KM')]\!]_{E\cup\{(z,n)\}} \\ &= [\![(KM)]\!]_E \end{aligned}$$

となる。

一方、変数 x が項 M_1で強自由でない場合には:

$$\begin{aligned} [\![\mathcal{F}_U^K M]\!]_F &= [\![(KM)]\!]_F \\ &= [\![(KM)]\!]_E \end{aligned}$$

となる。 □

項の集合から項への写像 $\mathcal{T}$が(その定義域上で)以下を満たす時、写像 $\mathcal{T}$を**プログラム変換**と呼ぶ。

$$[\![M]\!]_E = [\![\mathcal{T}M]\!]_E$$

但しここで Eは Mの任意の環境である。

環境 Eは空集合上では E自身の K-随伴であるから、定理 4.1の特別な場合として以下を得る。

系 1 変換 $(KM) \mapsto \mathcal{F}_\phi^K M$は(項全体で定義された)プログラム変換である。

5 レデクスと準レデクス

定義 3.2で定めた融合変換は:

$$(KM) \mapsto \mathcal{F}^K_\phi M$$

という簡約原理であるとみなすことができる。左辺の形をした項の出現を**前レデクス**という。更に上記簡約が恒等変換とならない時には、これを**レデクス**という。前レデクスがレデクスとなるのは、項 M が $(\mathbf{0})$ の時と $(\mathcal{S}N)$ の形の時および:

$$M = \mathcal{R}(M_0, \lambda x M_1)N$$

で x が M_1 が強自由の時である。レデクスを一つも含まない項のことを**既約項**と呼ぶ。本節以降では、与えられた項に融合変換を（適当な手順で）繰り返し適用することで規約項が得られることを示す。

まず補助概念として**準レデクス**を導入する。準レデクスとは一言でいえばレデクスの候補である。

例えば前レデクス $(K(\mathcal{R}(M_0, \lambda x M_1)N))$ は、変数 x が M_1 で強自由でなければレデクスではないが、部分項 M_1 中にレデクスがあれば、それを簡約した結果レデクスとなることもあり得る。また再帰子 L の L_1 中に前レデクス $(K(y))$ があった場合、これ自体はレデクスでないが、再帰子 L を他項に融合すると（$(\mathcal{S}N)$ の置き換えにより）$(K\bar{N})$ の形のレデクスを生じさせる可能性がある。

以下に述べる準レデクスの定義は、このように他レデクスの融合によってレデクスに変化する恐れのある前レデクスを（十分条件によって）特徴付けたものである。準レデクスは局所的概念ではなく、これが置かれた文脈に影響を受けるため「前レデクス (KM) は項 P で準レデクスである」という形で定義される。

定義 5.1 (準レデクス) レデクス (KM) は任意の項 P で準レデクスである。またレデクスでない前レデクス (KM) は、以下のいづれかの時に項 P で準レデクスである。

(a) 項 M が (y) の形で、かつ P の別の準レデクス $(\mathcal{R}(L_0, \lambda y L_1)N)$ において L_1 の部分項となる時。

(b) 項 M が $(\mathcal{R}(M_0, \lambda x M_1)N)$ の形で、かつ項 M が P の準レデクスを含む時。

今までと異なり、定義 5.1は項の構成に関する帰納法とはなっていない。しかし、項 P 中の前レデクスの集合 W_P に適切な半順序 '$\prec_P$' を定めれば、それに関する帰納法とみなすことができる。

定義 5.2 (i) 項 P が $(\mathbf{0})$ または (x) なら、集合 W_P は空なので半順序 $\prec_P$ は自明である。

(ii) 項 P が $(\mathcal{S}Q)$ の形なら W_P は W_Q に等しい。半順序 $\prec_P$ は $\prec_Q$ と同じものとする。

(iii) 項 P が $(\mathcal{R}(P_0, \lambda x P_1)Q)$ なら

$$W_P = W_{P_0} \cup W_{P_1} \cup W_Q \cup \{P\}$$

となる。半順序 $\prec_P$ は部分集合 W_{P_0}, W_{P_1}, W_Q 上ではそれぞれ半順序 $\prec_{P_0}, \prec_{P_1}, \prec_Q$ を継承する。更に前レデクス (LN) と (KM) がこれら部分集合のいづれか一つに同時に属さなければ、以下の順序関係に従う。

$$(LN) \prec_P (KM) \quad \Leftrightarrow \quad (LN) \in W_Q \vee ((LN) = P \wedge (KM) \notin W_Q)$$

項 Q が項 P の部分項である時、半順序 $\prec_P$ は W_Q 上で $\prec_Q$ と一致する。これに基づき以下の議論では、半順序の定義域を表す添字を省略して、単に '$\prec$' と書くことにする。

6 準レデクスの性質と項のランク

既に述べたように準レデクスであるか否かは、その置かれた文脈に依存する。本節ではまず、前レデクスがある項で準レデクスとなる事と、その部分項で準レデクスとなる事との関係を調べる。

命題 6.1 項 P の部分項を Q とする。項 Q での準レデクスは項 P でも準レデクスである。

証明： 準レデクスがレデクスの時には明らかである。そうでない時を背理法により示す。項 Q の準レデクスで P では準レデクスでない $\prec$-極小のものを (KM) と置く。

(a) 項 M が (x) であれば、項 Q の別の準レデクス (LN) において L_1 の部分項となる。最小性により (LN) は P でも準レデクスとなるが、これは (KM) が P の準レデクスとなることを導き、矛盾する。

(b) 項 M が $(\mathcal{R}(M_0, \lambda x M_1)N)$ の形なら、これは Q の準レデクスを含む。最小性により M は P の準レデクスを含むが、これは (KM) が P の準レデクスとなることを導き、矛盾する。 □

命題 6.1の逆が成り立つためには、若干の条件が必要となる。

命題 6.2 項 P の部分項を Q とする。また Q の自由変数は P でも自由と仮定する。項 Q 中の前レデクスが P で準レデクスなら、項 Q でも準レデクス

である。

証明： 命題 6.1と同様にして証明できる。変数の自由性に関する仮定はケース (a)を示す際に必要となる。 □

更に、項 P自体が前レデクス (KM) の形の時には次が成り立つ。証明の方針は上記の二命題と同様である。

命題 6.3 反復子 Kを $\mathcal{R}(K_0, \lambda x K_1)$ の形とし、項 (KM) は (KM) 自身で準レデクスでないとする。部分項 K_1中の前レデクスは、(KM) で準レデクスなら K_1でも準レデクスである。

項 Pにおける部分項 Q の**深さ**とは、Q を含む P の部分項の個数をいう。深さは必ず正の自然数となる。準レデクスとその深さを用いて項の複雑さに対する指標を定義する。

定義 6.1 (ランク) 項のランクとは、その準レデクスの深さの総和をいう。反復子 Kのランクとは、項 $K(\mathbf{0})$ のランクをいう。

レデクスは準レデクスなので、ランク零の項は既約項となる。ランク n 以下の項の集合をΩ_nと記する。また項全体をΩ_ωと書く。

$$\Omega_0 \subseteq \Omega_1 \subseteq \cdots \subseteq \Omega_\omega = \bigcup_{n=0}^{\infty} \Omega_n$$

準レデクスに関する上記の三命題より、ランクについての以下の性質を導くことができる。

補題 6.2 (1) 部分項のランクは親項のランク以下である。更に親項が正なら、真部分項のランクは親項のランクより真に小さい。

(2) 反復子 $\mathcal{R}(K_0, \lambda y K_1)$ のランクを n とする。項 Nが既約の時、項 $K_1[N/y]$ のランクは n より小さい。

(3) 項 Mおよび K_0と K_1のランクが零の時、適当な反復子 L に関して (LM) が準レデクスでなければ、項 (KM) のランクも零である。

7 融合変換の反復適用

前二節での議論を踏まえ、本節では与えられた項を等価な既約項に変換する手続きを提示する。これは二つのプログラム変換の相互呼びだしの形で実現される。それぞれ、**大域変換**および**局所変換**と呼ぶ。

大域変換の仕事は、与えられた項の構成に沿って前レデクスを探すことである。発見された前レデクスは局所変換によって融合が施される。ここで呼び出される局所変換の手続きは定義 3.2とほぼ同じだが、構成子の置換によって新たな項が生じた場合、そのまま出力せずに大域変換を用いて既約化する点が異なっている。

注意すべき点は、大域融合がその対象のランクによって階層化されている点である。ランク n 以下の項を処理する大域変換を $\mathcal{G}_n$と書くことにする。大域変換 $\mathcal{G}_n$に呼び出された局所変換が、再び大域変換を利用する際には、階層が一つ低い $\mathcal{G}_{n-1}$を呼び出す。このため再帰呼び出しが繰り返される度に、大域変換の階層は下がっていき、最終的には恒等写像である $\mathcal{G}_0$が呼び出される。この事実は変換手続きの停止性を保証する。

任意のプログラム変換 $\mathcal{T}$に対して、局所変換 $\mathcal{F}^K_U[\mathcal{T}]$ を定義する。記述の繁雑さを避けるため $\mathcal{T}$ は省略する。

定義 7.1 (局所融合) (i) 項 Mが $(\mathbf{0})$ なら:

$$\mathcal{F}^K_U M = \bar{K}_0$$

と定める。項 Mが (x) の時には

$$\mathcal{F}^K_U M = \begin{cases} (\hat{x}) & (if\ x \in U) \\ (\bar{K}(x)) & (if\ x \notin U) \end{cases}$$

と定める。但しここで:

$$\bar{K}_0 = \mathcal{T}K_0 \quad \bar{K}_1 = \mathcal{T}K_1$$
$$\bar{K} = \mathcal{R}(\bar{K}_0, \lambda y \bar{K}_1)$$

とする。

(ii) 項 Mが $(\mathcal{S}N)$ の形の時:

$$\mathcal{F}^K_U M = \mathcal{T}(K_1[\mathcal{F}^K_U N/y])$$

と定める。

(iii) 項 Mが $(\mathcal{R}(M_0, \lambda x M_1)N)$ の形の時には、変数 x が部分項 M_1で強自由か否かによって場合分けする。強自由の場合には:

$$\mathcal{F}^K_U M = (\mathcal{R}(\mathcal{F}^K_U M_0, \lambda \hat{x} \mathcal{F}^K_{U\cup\{x\}} M_1)N)$$

と定める。強自由でない場合には:

$$\mathcal{F}^K_U M = (\bar{K}M)$$

と定める。(反復子$\bar{K}$は (i)で決めたもの)

定理 7.2 ランク n 以下の再帰子 Kとプログラム変換 $\mathcal{T} : \Omega_{n-1} \to \Omega_0$ついて次が成り立つ。

(1) $\mathcal{F}^K_U[\mathcal{T}] : \Omega_0 \to \Omega_0$

(2) 変換 $(KM) \mapsto \mathcal{F}^K_\phi[\mathcal{T}]M$はプログラム変換である。

証明： (2)は一般の Uの場合を定理 4.1と同様にして示すことができる。以下、項に関する帰納法で (1)を証明する。

(i) 項 Mが $(\mathbf{0})$ なら $\mathcal{T}$に関する条件より自明。項 Mが (x) の時、変数 x が Uに属せば自明。属さない時は $\mathcal{T}$に関する条件と補題 6.2の (3)により示される。

(ii) 項 Mが $(\mathcal{S}N)$ なら補題 6.2の (2)と帰納法の仮定から示される。

(iii) 項 Mが $(\mathcal{R}(M_0, \lambda x M_1)N)$ の時。変数 x が M_1 で強自由なら、帰納法の仮定と補題 6.2の (3)により示される。強自由でなければ $\mathcal{T}$に関する条件と補題 6.2の (3)により示される。 □

更に、自然数 n に対して大域変換 $\mathcal{G}_n$を次のように定める。

定義 7.3 (局所融合) 変換 $\mathcal{G}_0$は恒等写像とする。零でない自然数 n に対しては以下で定義する。

(i) $\mathcal{G}_n(\mathbf{0}) = (\mathbf{0}), \quad \mathcal{G}_n(x) = (x).$

(ii) $\mathcal{G}_n(\mathcal{S}Q) = (\mathcal{S}\bar{Q}) \quad where\ \bar{Q} = \mathcal{G}_n Q.$

(iii) Pが $(\mathcal{R}(K_0, \lambda y K_1)Q)$ の形なら：

$$\mathcal{G}_n P = \mathcal{F}^K_\phi[\mathcal{G}_{n-1}]\bar{Q}.$$

定理 7.4 変換 $\mathcal{G}_n$はΩ_nからΩ_0へのプログラム変換である。

証明： 自然数 n に関する帰納法。各 n については項に関する帰納法で示す。本質的なのはケース (iii)である。項 (KQ) が準レデクスの時は Kのランクが n 以下なので、定理 7.2より明らか。これが準レデクスでない時には Q は既約で$\bar{Q} = Q$ であり、従って $(K\bar{Q})$ はレデクスでない。局所変換の定義により補題 6.2の (3)を用いて示す。 □

これに加えて、以下の図式が可換であることも（自然数 n に関する帰納法で）示される。従って n を無限にする極限で得られる $\mathcal{G}_\omega$はΩ_ωからΩ_0への写像となる。

$$\begin{array}{ccccccccc}
\Omega_0 & \xrightarrow{\subset} & \Omega_1 & \xrightarrow{\subset} & \Omega_2 & \xrightarrow{\subset} & \cdots & \xrightarrow{\subset} & \Omega_\omega \\
| & & | & & | & & & & | \\
\mathcal{G}_0 & & \mathcal{G}_1 & & \mathcal{G}_2 & & \cdots & & \mathcal{G}_\omega \\
\downarrow & & \downarrow & & \downarrow & & & & \downarrow \\
\Omega_0 & \xrightarrow{id} & \Omega_0 & \xrightarrow{id} & \Omega_0 & \xrightarrow{id} & \cdots & \xrightarrow{id} & \Omega_0
\end{array}$$

系 2 写像 $\mathcal{G}_\omega$はΩ_ωからΩ_0へのプログラム変換である。

8 まとめ

本稿では原始反復関数の融合手続きを提示した。融合可能性の判定には、項中に現れる変数の強自由性を利用する。また、融合法を適当な手順で繰り返し適用すれば、全ての融合可能部分の融合が可能であることを示した。この手続きの停止性は、項の簡約としての融合変換が弱正規化可能であることを表しているが、これは実際には強正規化可能であろうと予想している。

参考文献

[1] Gill,A., Launchbury,J., and Jones,S. (1993). A short cut to deforestation, *Proc. Conferenece on Functional Programming Languages and Computer Architecture*, ACP Press, pp. 223-232.

[2] Takano,A. and Meijer,E. (1995). Shortcut deforestation in calculational form. In *Peyton-Jones,S. editor, Functional Programming Languages and Computer Architecture*, pages 306-313, Association for Computer Machinery.

[3] Wadler,P.L. (1988). Deforestation: Transforming programs to eliminate trees. In *European Symposium on Programming*, Vol.300 of *LNCS*, Springer-Verlag.

[4] Wadler,P.L. (1990). Deforestation: Transforming programs to eliminate intermediate trees. *Theoretical Computer Science*, Vol.73, 231-248.

A 付録

本稿の大域融合を実現する ELisp プログラムを付する。主な関数の意味は以下の通りである。

- `occur-free`: 変数の自由出現を判定する。
- `substitute-var`: 代入を実行する。
- `strongly-free`: 変数の強自由性を判定する。
- `fuse-locally`: 局所融合を実行する。
- `fuse-globally`: 大域融合を実行する。

末尾に掲げた実行例は以下の変換である。

例題 A.1

$$\begin{aligned}
L &= \mathcal{R}((\mathcal{S}(\mathbf{0})), \lambda z(\mathbf{0}))(y) \\
K &= \mathcal{R}((\mathbf{0}), \lambda y L) \\
M &= (\mathcal{R}((\mathbf{0}), \lambda x(\mathcal{S}(\mathcal{S}(x))))(a)) \\
\mathcal{G}_\omega(KM) &= \mathcal{R}((\mathbf{0}), \lambda \hat{x}\bar{M})(a), \\
&\quad where\ \bar{M} = \mathcal{R}((\mathbf{0}), \lambda \hat{z}(\mathcal{S}(\mathbf{0})))(\hat{x}).
\end{aligned}$$

```
;;
;; Declare Symbols
;;

(defun reset-var-counter () (setq var-counter 0))

(defun get-new-var (meta-var)
  (let* ((var-symb (make-symbol (concat "v" (number-to-string var-counter))))
         (var-func (list 'lambda ()
                         (list 'list
                               (list 'quote var-symb)))))
    (progn (fset meta-var var-func)
           (set meta-var var-symb)
           (setq var-counter (+ var-counter 1)))))

(defun declare-const-zero (meta-var)
  (let* ((const-symb (make-symbol "Z"))
         (const-func (list 'lambda ()
                           (list 'list
                                 (list 'quote const-symb)))))
    (progn (fset meta-var const-func)
           (set meta-var const-symb))))

(defun declare-function-succ (meta-var)
  (let* ((const-symb (make-symbol "S"))
         (const-func (list 'lambda (list 'N)
                           (list 'list
                                 (list 'quote const-symb)
                                 'N))))
    (progn (fset meta-var const-func)
           (set meta-var const-symb))))

(defun declare-lambda (meta-var)
  (let* ((const-symb (make-symbol "lm"))
         (const-func (list 'lambda (list 'u 'M)
                           (list 'list
                                 (list 'quote const-symb)
                                 'u
                                 'M))))
    (progn (fset meta-var const-func)
           (set meta-var const-symb))))

(defun declare-recursor (meta-var)
  (let* ((const-symb (make-symbol "R"))
         (const-func (list 'lambda (list 'M0 'M1 'N)
                           (list 'list
                                 (list 'quote const-symb)
                                 'M0
                                 'M1
                                 'N))))
    (progn (fset meta-var const-func)
           (set meta-var const-symb))))

(defun declare-symbols ()
  (progn (reset-var-counter)
         (declare-const-zero 'Z)
         (declare-function-succ 'S)
         (declare-lambda 'lm)
         (declare-recursor 'R)))


;;
;; Parsing Terms
;;

(defun var-case-var (M)
  (car M))

(defun case-zero (M)
  (eq (car M) Z))

(defun case-succ (M)
  (eq (car M) S))

(defun arg-case-succ (M)
  (car (cdr M)))

(defun case-lambda (L)
  (eq (car L) lm))

(defun bvar-case-lambda (L)
  (car (cdr L)))
```

```
(defun term-case-lambda (L)
  (car (cdr (cdr L))))

(defun case-recursor (M)
  (eq (car M) R))

(defun base-case-recursor (M)
  (car (cdr M)))

(defun step-case-recursor (M)
  (car (cdr (cdr M))))

(defun arg-case-recursor (M)
  (car (cdr (cdr (cdr M)))))


;;
;; Free Variable
;;

(defun occur-free-case-var (v M)
  (eq v (var-case-var M)))

(defun occur-free-case-zero (v M)
  ())

(defun occur-free-case-succ (v M)
  (let ((N (arg-case-succ M)))
    (occur-free v N)))

(defun occur-free-case-lambda (v L)
  (let ((u (bvar-case-lambda L))
        (M1 (term-case-lambda L)))
    (and (not (eq v u))
         (occur-free v M1))))

(defun occur-free-case-recursor (v M)
  (let ((M0 (base-case-recursor M))
        (L (step-case-recursor M))
        (N (arg-case-recursor M)))
    (or (occur-free v M0)
        (occur-free-case-lambda v L)
        (occur-free v N))))

(defun occur-free (v M)
  (cond ((case-zero M) (occur-free-case-zero v M))
        ((case-succ M) (occur-free-case-succ v M))
        ((case-recursor M) (occur-free-case-recursor v M))
        (t (occur-free-case-var v M))))


;;
;; Substitution
;;

(defun subst-vars-case-var (substitutions M)
  (if substitutions
      (let ((rep (car substitutions))
            (rest (cdr substitutions)))
        (if (eq (car (cdr rep)) (var-case-var M))
            (car rep)
            (subst-vars-case-var rest M)))
      M))

(defun subst-vars-case-zero (substitutions M)
  M)

(defun subst-vars-case-succ (substitutions M)
  (let ((N (arg-case-succ M)))
    (let ((new-N (subst-vars substitutions N)))
      (S new-N))))

(defun subst-vars-case-lambda (substitutions L)
  (let ((u (bvar-case-lambda L))
        (M1 (term-case-lambda L)))
    (progn (get-new-var 'new-var)
           (let ((new-u new-var)
                 (new-M1 (subst-vars (cons (list (new-var) u)
                                           substitutions)
                                     M1)))
             (lm new-u new-M1)))))
```

```
(defun subst-vars-case-recursor (substitutions M)
  (let ((M0 (base-case-recursor M))
        (L (step-case-recursor M))
        (N (arg-case-recursor M)))
    (let ((new-M0 (subst-vars substitutions M0))
          (new-L (subst-vars-case-lambda substitutions L))
          (new-N (subst-vars substitutions N)))
      (R new-M0 new-L new-N))))

(defun subst-vars (substitutions M)
  (cond ((case-zero M) (subst-vars-case-zero substitutions M))
        ((case-succ M) (subst-vars-case-succ substitutions M))
        ((case-recursor M) (subst-vars-case-recursor substitutions M))
        (t (subst-vars-case-var substitutions M))))

(defun substitute-var (substitution M)
  (subst-vars (list substitution) M))


;;
;; Strongly Free
;;

(defun strongly-free-case-var (v M)
  t)

(defun strongly-free-case-zero (v M)
  t)

(defun strongly-free-case-succ (v M)
  (let ((N (arg-case-succ M)))
    (strongly-free v N)))

(defun strongly-free-case-lambda (v L)
  (let ((u (bvar-case-lambda L))
        (M1 (term-case-lambda L)))
    (or (eq v u)
        (strongly-free v M1))))

(defun strongly-free-case-recursor-sf (v M)
  (let ((M0 (base-case-recursor M))
        (L (step-case-recursor M))
        (N (arg-case-recursor M)))
    (and (strongly-free v M0)
         (strongly-free-case-lambda v L)
         (not (occur-free v N)))))

(defun strongly-free-case-recursor-nsf (v M)
  (not (occur-free v M)))

(defun strongly-free-case-recursor (v M)
  (let* ((L (step-case-recursor M))
         (u (bvar-case-lambda L))
         (M1 (term-case-lambda L)))
    (if (strongly-free u M1)
        (strongly-free-case-recursor-sf v M)
        (strongly-free-case-recursor-nsf v M))))

(defun strongly-free (v M)
  (cond ((case-zero M) (strongly-free-case-zero v M))
        ((case-succ M) (strongly-free-case-succ v M))
        ((case-recursor M) (strongly-free-case-recursor v M))
        (t (strongly-free-case-var v M))))


;;
;; Local Fusion
;;

(defun fuse-locally-case-var (K substitutions M)
  (if substitutions
      (let ((rep (car substitutions))
            (rest (cdr substitutions)))
        (if (eq (car (cdr rep)) (var-case-var M))
            (car rep)
            (fuse-locally-case-var K rest M)))
      (let* ((K0 (base-case-recursor K))
             (J (step-case-recursor K))
             (v (bvar-case-lambda J))
             (K1 (term-case-lambda J)))
        (let* ((new-K0 (fuse-globally K0))
```

```
                (new-K1 (fuse-globally K1))
                (new-J (lm v new-K1)))
            (R new-K0 new-J M)))))

(defun fuse-locally-case-zero (K substitutions M)
  (let ((K0 (base-case-recursor K)))
    (fuse-globally K0)))

(defun fuse-locally-case-succ (K substitutions M)
  (let* ((J (step-case-recursor K))
         (u (bvar-case-lambda J))
         (K1 (term-case-lambda J))
         (N (arg-case-succ M)))
    (let* ((new-N (fuse-locally K substitutions N))
           (new-K1 (substitute-var (list new-N u)
                                   K1)))
      (fuse-globally new-K1))))

(defun fuse-locally-case-lambda (K substitutions L)
  (let ((u (bvar-case-lambda L))
        (M1 (term-case-lambda L)))
    (progn (get-new-var 'new-var)
           (let ((new-u new-var)
                 (new-M1 (fuse-locally K (cons (list (new-var) u)
                                               substitutions)
                                       M1)))
             (lm new-u new-M1)))))

(defun fuse-locally-case-recursor-sf (K substitutions M)
  (let ((M0 (base-case-recursor M))
        (L (step-case-recursor M))
        (N (arg-case-recursor M)))
    (let ((new-M0 (fuse-locally K substitutions M0))
          (new-L (fuse-locally-case-lambda K substitutions L)))
      (R new-M0 new-L N))))

(defun fuse-locally-case-recursor-nsf (K substitutions M)
  (let* ((K0 (base-case-recursor K))
         (J (step-case-recursor K))
         (v (bvar-case-lambda J))
         (K1 (term-case-lambda J)))
    (let* ((new-K0 (fuse-globally K0))
           (new-K1 (fuse-globally K1))
           (new-J (lm v new-K1)))
      (R new-K0 new-J M))))

(defun fuse-locally-case-recursor (K substitutions M)
  (let* ((L (step-case-recursor M))
         (u (bvar-case-lambda L))
         (M1 (term-case-lambda L)))
    (if (strongly-free u M1)
        (fuse-locally-case-recursor-sf K substitutions M)
        (fuse-locally-case-recursor-nsf K substitutions M))))

(defun fuse-locally (K substitutions M)
  (cond ((case-zero M) (fuse-locally-case-zero K substitutions M))
        ((case-succ M) (fuse-locally-case-succ K substitutions M))
        ((case-recursor M) (fuse-locally-case-recursor K substitutions M))
        (t (fuse-locally-case-var K substitutions M))))


;;
;; Global Fusion
;;

(defun fuse-globally-case-var (M)
  M)

(defun fuse-globally-case-zero (M)
  M)

(defun fuse-globally-case-succ (M)
  (let ((N (arg-case-succ M)))
    (let ((new-N (fuse-globally N)))
      (S new-N))))

(defun fuse-globally-case-recursor (M)
  (let ((K0 (base-case-recursor M))
        (J (step-case-recursor M))
        (N (arg-case-recursor M)))
    (let* ((K (R K0 J ()))
           (new-N (fuse-globally N)))
```

```
      (fuse-locally K () new-N))))

(defun fuse-globally (M)
  (cond ((case-zero M) (fuse-globally-case-zero M))
        ((case-succ M) (fuse-globally-case-succ M))
        ((case-recursor M) (fuse-globally-case-recursor M))
        (t (fuse-globally-case-var M))))


;;
;; Example
;;

(declare-symbols)
(get-new-var 'a)
(get-new-var 'x)
(get-new-var 'y)
(get-new-var 'z)
(setq L (R (S (Z)) (lm z (Z)) (y)))
(setq M (R (Z) (lm x (S (S (x)))) (a)))
(setq KM (R (Z) (lm y L) M))

KM
> (R (Z) (lm v2 (R (S (Z)) (lm v3 (Z))
                   (v2)))
      (R (Z) (lm v1 (S (S (v1))))
         (v0)))

(fuse-globally KM)
> (R (Z) (lm v8 (R (Z) (lm v11 (S (Z)))
                   (v8)))
      (v0))
```