修 士 論 文

完備化手続きを用いたプログラム合成の研究

指導教官　外山芳人 教授

北陸先端科学技術大学院大学
情報科学研究科情報処理学専攻

吉政洋美

1999 年 2 月 15 日

要 旨

本論文は、切り払い手法を適用させたプログラム変換の有効性を調査するために、Knuth-Bendix の完備化手続きを用いて自動的にプログラム合成させる実験を行なった結果を主に報告するものである。このときの項書き換え系の切り払い手法の表現方法・表記や完備化手続きなどの概要を述べた後、46 種類の入力データに対して、順序付けを変えて Knuth-Bendix 完備化手続きの実験を行なった 731 回に及ぶプログラム合成の中から成功例や失敗例を示し、その結果を解析することを目的とする。

目 次

第 1 章

はじめに

1.1 研究の背景・目的

P.Wadler によって提案された切り払い手法は、関数型プログラミング言語において、畳み込み-合成規則（The foldr/build rule）を用いることによって不必要であると思われるデータ構造を削除する手法である [1]。そして、関数型言語のプログラム変換法として融合変換を用いた切り払い手法が有効であることが知られている [1, 2]。F.Bellegarde の論文より、項書換え系で切り払い手法のための融合ルール (Fusion rule) が表現できることが示されている。ここでの融合ルールは、定義された書き換え規則の右辺の中での融合項 (Fusion term) が構成子に基づいた系 R の実行によって見つけられることにより、構成子に基づいた系 R_h が統合されるというものである。これは、ある書き換え規則を関数型プログラミング言語において定義し、それに対して合成規則を付け加えることによって、融合ルール $s \to h(x_1, x_2, \cdots, x_n)$ が生成されるシステムである。F.Bellegarde は項書換え系に対する融合変換が完備化手続きに基づいて実現できることを示した [2]。F.Bellegarde の方法は、完備化手続きをそのまま利用するのではなく、融合変換を行なう対象も制限されている。本研究では、完備化手続きをそのまま利用しても、項書換え系の融合変換が有効であり、プログラム合成の対象も拡張可能であることを示す。

1.2 本論文の構成

本論文の構成として、まずはじめに、切り払い手法についての文献調査について、2 章で述べた後に、切り払い手法で用いられる項書き換え系の定義と表現を 3 章でまとめる。そして、本研究で用いた完備化手続き（Knuth-Bendix 完備化手続き）のシステムについて、4 章で詳しく説明する。その上で、本研究の主題である Knuth-Bendix 完備化手続きによる切り払い手法を用いたプログラム合成の結果について 5 章でまとめる。そこで、実際的に順序付けや合成の等式をよく考えた上で行なった Knuth-Bendix 完備化手続きでのプログラム合成の実験の結果での考察を行なう。最後に、項書き換え系での切り払い手法の応用としての他のプログラム言語による切り払い手法への展望とまだ未解決である問題を今後の課題として述べる。

第 2 章

切り払い手法についての文献調査

本章では、研究の動機となった切り払い手法についての文献調査での内容とそこで認められた切り払い手法の有効性についての概要を述べる。

2.1 合成戦略について

切り払い手法 (Deforestation) は、合成戦略 (Composition strategy) の一つであり、関数型プログラムの実行途中に生成されるリスト、木 (tree) などのデータ構造を除去することで計算の効率化を実現するプログラム変換である [1]。合成戦略とは、プログラム変換の中で Burstall & Darlington(1977) が提案した定義 (definition)、畳み込み (folding)、展開 (unfolding) を主たる規則とする 7 つの規則からなる戦略である [4]。ここで、定義とは、既に定義された関数を用いて新しい関数を定義することであり、畳み込みとは、項の中に出現している一部分がある関数の定義式の右辺と一致しているならば、その部分を左辺の関数に置き換えてまとめる操作である。そして、展開とは、たたみ込みの逆で、項の中に出現している一部分がある関数の定義式の左辺と一致しているならば、その部分を右辺で展開する操作である。

この合成戦略を項書き換え系に適用させた例が、F.Bellegarde の論文 [2] の完備化手続きを用いた切り払い手法によるプログラム自動生成の研究にある。ここでの展開-畳み込み手法 (Unfold-fold Method) は、定義 (definition)、具体化 (instantiation)、展開 (unfolding)、畳み込み (folding)、抽出 (abstraction)、法 (law) の 6 つの規則からなる。これは、Burstall

& Darlington の提案した規則を使ったものである。

2.2 切り払い手法によるプログラム変換の有効性

P.Wadler は、このような合成戦略を関数型プログラミング言語に適用させた切り払い手法を試みることによって、そのプログラム変換の有効性を調べている。数値型やブール型なら計算途中に現れても計算効率を悪化させないことに着目し、印付切り払い (blazed deforestation) という手法を提案している [1]。これは、変換すべきでない項に印をつけ、それらの項を変換対象から外すことにより効率化をはかる手法である。この印付切り払い手法は、佐賀の拡張された切り払い手法における研究にも利用されている [5]。

切り払い手法の文献調査を行なうと、文献 [6] では、畳み込み-合成規則 (foldr/build rule) などの理論的な考察や畳み込み-合成変換 (The fold/build Transformation) の簡単な例、実装された簡易切り払い手法 (Cheap Deforestation) の有効性などに関して詳しい。一般に切り払い手法は、プログラムに応じた規則を次々に適用するだけで変換が実行されるという単純な処理であるため、変換過程の自動化に適している。しかし、アルゴリズムの効率が不十分であること、変換対象のプログラムの性質が限定されること、高階プログラムを直接扱えない等の問題を含んでいる。

2.3 F.Bellegarde の切り払い手法について

様々な切り払い手法の論文を調査した結果、完備化手続きを用いたプログラム自動生成についての F.Bellegarde の論文 [2] が、項書き換え系に合成戦略や切り払い手法を用いた論文だった。そこで、この論文を基にして Knuth-Bendex 完備化手続きを用いたプログラム合成の実験において、様々な順序付けやプログラムリストの合成例を試すことによって、融合ルールを導出させて検証を行なうことになった。

F.Bellegarde の切り払い手法に関する項書き換え系での定義・表現は、3 章で述べる。

2.3.1 F.Bellegarde のプログラム自動生成

F.Bellegarde の切り払い手法によるプログラム自動生成 [2] は、完備化手続きのアルゴリズムより導出される。 $h(x_1, x_2, ..., x_n)$ を、左辺におき、合成させたい融合項 t に対して、新しい関数記号 (Fresh symbol) h を用いて、等式 $h(x_1, ..., x_n) = t$ を作り、これを E の等式とする。等式 E を書き換え規則 $R_h : t \to h(x_1, x_2, ..., x_n)$ に変換させる手続きをとる。本研究は、Knuth-Bendix 完備化手続きを用いて等式の左辺と右辺が入れ替わった形の書き換え規則と合成された書き換え規則の集合が得られるような E の等式と順序付けについての例を収集するところにある。合成された書き換え規則が得られるような成功例には、文献 [2] によると融合ルール (Fusion rule) が適用される。

切り払い手法によるプログラム合成の手順を図 1-1 に示す。

R : { 項書換え規則 }

$E : h(x) = t$

⇓ 順序付け

融合ルール $R_h : t \to h(x)$

$R \cup R_h$

⇓ 完備化

$$\begin{cases} h([]) \to t_1 \\ h(x :: xs) \to t_2 \end{cases}$$

図 1-1 プログラム合成手順

2.3.2 Knuth-Bendix 完備化手続きへの拡張

4 章で述べる Knuth-Bendix 完備化手続きへの拡張によって F.Bellegarde による融合ルールにある制限が取り外せることが本研究でわかった。例えば、3 章で述べる融合ルール (Fusion rule) の定義にある (3) の融合ルール R_h の融合項 t が線形でなければならない点

が、$x + x \to d(x)$ での $x + x$ のような非線形のものでもうまく合成されるということである。これは、Knuth-Bendix 完備化手続きによる計算機上での実験でいえることである [7]。

例　融合項が非線形項であるプログラム合成例を次に示す。

$$R_+ : \begin{cases} x + 0 \to x \\ 0 + y \to y \end{cases}$$

$$R_s : \begin{cases} x + (s(y)) \to s(x + y) \\ (s(x) + y) \to s(x + y) \end{cases}$$

$$E : d(x) = x + x$$

$\Downarrow$ 順序付け ($d > + > s$ または $+ > d > s$)

融合ルール $R_h : x + x \to d(x)$

$$R \cup R_h$$

$\Downarrow$ Knuth-Bendix 完備化

$$\begin{cases} d(0) \to 0 \\ d(s(x)) \to (s(s(d(x)))) \end{cases}$$

このような、非線形の融合項で行なった実験の成功例や失敗例を 5 章で更に詳しく述べる。

第 3 章

項書き換え系について

3.1 項書き換え系の定義・表現について

理論的な切り払い手法に対する項書き換え系の定義・表現について、F.Bellegarde の論文 [2, 3] を基にして述べ挙げる。

3.1.1 基本的な用語の定義

定義 1 **定義された関数記号** *(defined function symbol)* **の定義**

Σ 上の項書き換え系 R の定義された関数記号の集合 Σ_d :

$\Sigma_d = \{f | f(l_1, l_2, ...l_n \rightarrow r \in R\}$

$\Sigma = \{a, b, c, f, g\}$

$$R = \begin{cases} f(g(a), x) \rightarrow c(x) \\ g(b) \rightarrow f(b, a) \\ a \rightarrow b \end{cases}$$

$\Sigma_d = \{f, g, a\}$

定義 2 **構成子記号** *(constructor symbol)* **の定義**

Σ 上の項書き換え系 R の構成子記号の集合 Σ_c :

$\Sigma_c = \Sigma \backslash \Sigma_d$

$= \{a, b, c, f, g\} \backslash \{f, g, a\}$
$= \{b, c\}$

定義 3 **構成子項** *(constructor term)* **の定義**
構成子記号の集合を Σ_c、変数の集合を V とすると、構成子項の集合 T_c は、
$T_c = T(\Sigma_c, V)$
項の中のすべての記号が、構成子記号か変数である。
例えば、$x, c(x), c(b)$ は構成子項で、$c(g(x))$ は、構成子項ではない。

定義 4 **構成子に基づいた項書き換え系** *(constructor-based TRS)* **の定義**
すべての書き換え規則 $f(t_1, t_2, ..., t_n) \to r$ について、$t_1, ..., t_n$ が構成子項のとき、R を、構成子に基づいた項書き換え系と呼ぶ。

3.1.2 位置 (position) の定義

定義 5 **帰納的位置** *(inductive position)* **の定義**
記号 f は帰納的位置 i においてデータ構造を消費する。
もし、R_f の中に $l \to r$ の規則 が存在するならば、$l|_i$ は構成子項であって、変数でもない。
項 t の中の記号に定義された帰納的位置に従って項 t から帰納的にデータ構造を使わずに生成する記号が含まれる。
これらの記号は、項 t の基幹位置に位置する。

定義 6 **基幹位置** *(spine position)* **の定義**
項 t の中の位置 w は、基幹位置とすると、
w は、ϵ または、$w = u.i$ ここで u は、基幹位置 $t|_u$ D のルート f と i は f の帰納的位置である。

例えば、下の図のような木構造の項 t を考える。

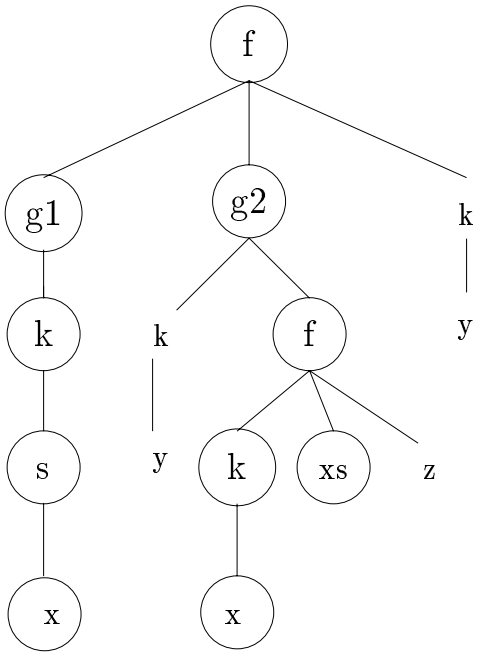
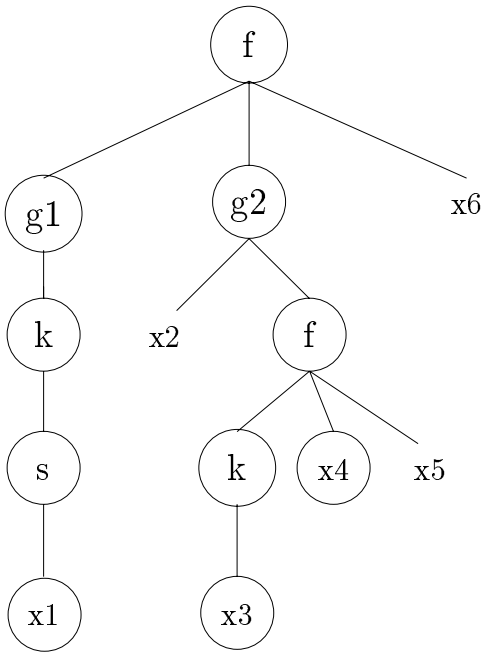


図 3-1 項 $t = f(g_1(k(s(x))), g_2(k(y), f(k(x), xs, z)), k(y))$

関数記号の帰納的位置 は、$f : \{1, 2\}, g_1 : \{1\}, g_2 : \{2\}, k : \{1, 1, 1\}$ そして、s は構成子である。○の部分 $\{\epsilon, 1, 1.1, 1.1.1, 1.1.1.1, 2, 2.2, 2.2.1, 2.2.1.1, 2.2.2\}$ は、基幹位置を表している。

構成子に基づいた系 R を実行すると定義した書き換え規則の右辺の中での融合項を見つけることが必要である。だから、融合ルール r が構築される。最終的に新しい記号 h の構成子に基づいた系 R_h を統合する。例えば項 t 上の R の中の書き換え規則の左辺である。系は、図 3-2 のような融合ルールを生成する。



⇓

h((x1), x2 ,(x3), (x4), x5 , x6)

図 3-2 融合ルール $f(g_1(k(s(x_1))), g_2(x_2, f(k(x_3), x_4, x_5)), x_6) \rightarrow h(x_1, x_2, x_3, x_4, x_5, x_6)$

新しい関数記号 h の帰納的位置は、その融合ルールを解析することによって定められる。これは、危険対の生成を制御するための鍵となる。

記号 h は、融合ルール によって導入されて、○で囲った場所の擬縮退変数 x_1, x_3, x_4 を持つ。統合の目的は、統合された記号 h の 定義 R_h を発見するためである。融合ルール と R との間の危険対を計算する。危険対の興味深いところは、x の擬縮退変数において、構成子項を持つ融合項 s の擬縮退変数 x を代入することである。右辺のリスト $(x_1, x_2, ..., x_n)$ の中の x の位置は、h の帰納的位置を示している。更に、もし、x_i が s の擬縮退変数であり、p が s の中の x_i に対する擬縮退位置のとき、$i.p$ は、h の帰納的位置である。前ページのような項 t に対して、h の帰納的位置は、{1,3,3.1,4 } となる。統合の過程は、h

の帰納的位置が空ではないときのみ動く。少なくとも、融合項と R との間で重なり合うものがなければいけない。これは、融合項に少なくとも一つの擬縮退部分項を含んでいるときに保証される。

3.1.3 融合ルール（Fusion rule）の定義

定義 7 **擬縮退項** *(surreductive term)* **の定義**

擬縮退項は、f の帰納的位置にある構成子項を持つ項 $f(t_1, t_2, ..., t_n)$ である。

R_f が完備化されているとき、縮退可能な R_f である擬縮退項 $f(t_1, t_2, ..., t_3)$ の構成子項によって、タイプ付き可能な具体例が存在する。だから、R によって、少なくとも一つの危険対 かリダクションのどちらか一方が存在することは確かである。だから、融合ルールを定義することができる。

定義 8 **擬縮退変数** *(surreductive variables)* **の定義**

$t = C[f(t_1, t_2, ..., t_n)]$ とする。f の帰納的位置における構成子項 t_i の中にある変数 x は、f に関連した t の擬縮退変数と呼ばれる。もし、p が、項 t の中の f と関連する擬縮退変数 x の位置であって、$p.u$ が p よりも帰納的位置が深いか、もしくは等しければ、u は、x と関連した擬縮退位置である。もし、変数 x が基幹位置の融合項の中に現れるならば、それは、いくつかの定義された記号 f に対する擬縮退変数である。

F.Bellegarde の切り払いのための統合規則の定義である融合ルールの定義を以下に示す。

定義 9 **融合ルール** *(Fusion rule)* **の定義**

$f(tc_1, tc_2, \cdots, tc_n) \to r$ を構成子に基づく R が、停止性を持ち、既約であるとする。もし、r の中に位置 u が以下のように存在するならば、融合項 s は r より抽出される。

(1)$r\mid_u$ は、s の実例 (instance) である。（encompassment）

(2) s のルート記号は、定義された記号 (defined symbol) である。そして、その他の s の中に現れる関数記号は、s の基幹位置に位置する。(deforestation)

(3)s は、線形である。(linearity)

(4)f は、s の中に現れない。(skip over recursive calls)
(5)1 よりも大きい $length$ の s の基幹位置がある。 (fusion existence)
(6)s が、少なくとも１つの擬縮退部分項に含まれている。(synthesizability)

3.2 項書き換え系での切り払い手法

合成戦略の１つとしての切り払い手法は、Burstall & Darlington の展開-畳み込み手法 (Unfold-fold Method) の 6 つの規則が用いられるが、その 6 つの規則の中で、基本的に具体化 (Instantiation) と畳み込み (Folding) の組み合わせは、危険対の生成によって得られ、展開 (Unfolding) と法 (Law) は、書き換えによって簡約化される。そして、定義 (Definition) は、統合規則 (Synthesis rule) を導き出すものである。

前節で述べた F.Bellegarde の融合ルール (Fusion rule) は、切り払い手法のための統合規則であるが、項書き換え系において適用させる時、定義を踏まえることによって、合成させる式の項を解析することが可能となる。そして、完備化手続きによって、生成される融合規則が停止性を持ち、新しい関数記号 (Fresh symbol) を含んだ規則を生成できるかを解析できる。しかし、本研究では Knuth-Bendix 完備化手続きを用いて実際に実験により、考察を行なう。データが得られると理論的に項書き換え系で用いられる表記により、解析できると考えられる。

第 4 章

完備化手続きについて

項書き換え系は、ある書き換え規則を与えてその等式に対して次々と等価交換していき、項を変化させていく。計算機上で完備化されたアルゴリズムで自動変換させたとき何回かのステップ数で完備化できるものと、完備化に失敗して停止してしまうものやアルゴリズムの上でループに陥り発散して停止しない３つの場合に分けられる。同じ規則を与えたプログラム上においてもその書き換え規則に現れる関数記号の順序付けによって、この３つの場合に分けられる。そこで、完備化手続き、特に Knuth-Bendix 完備化手続きとその順序付けについての概要についてまとめる [8]。

4.1 Knuth-Bendix 完備化手続き

Knuth-Bendix の基本的な考え方は項書き換え系 R を停止性と合流性を持つように、発散する危険対をなくしていき、新たな書き換え規則を含めたとしても繰り返し変形し完備な項書き換え系へ近づけていくことである。歴史的に Knuth-Bendix は、普遍代数 (universal algebra) の語の問題 (word problem) を解くアルゴリズムとして、60 年代後半に提案されたものである。その後 80 年代に入り、項書き換え系を通じて関数型言語や代数的仕様などの応用として拡張・改良などが施されている [9, 10, 11]。

E は等式の集合、R は書き換え規則の集合であり、$>$ は停止性保証のための順序である。新たに獲得された等式の両辺が $>$ で比較できないとき、つまり、R の停止性が保証できなくなったときは、失敗に終わる。完備である R が有限集合として存在しない場合は新

たな等式を獲得し続け停止しないので、正確には半アルゴリズムである [9]。

Knuth-Bendix の完備化手続きのアルゴリズムを図 4-1 に示す。

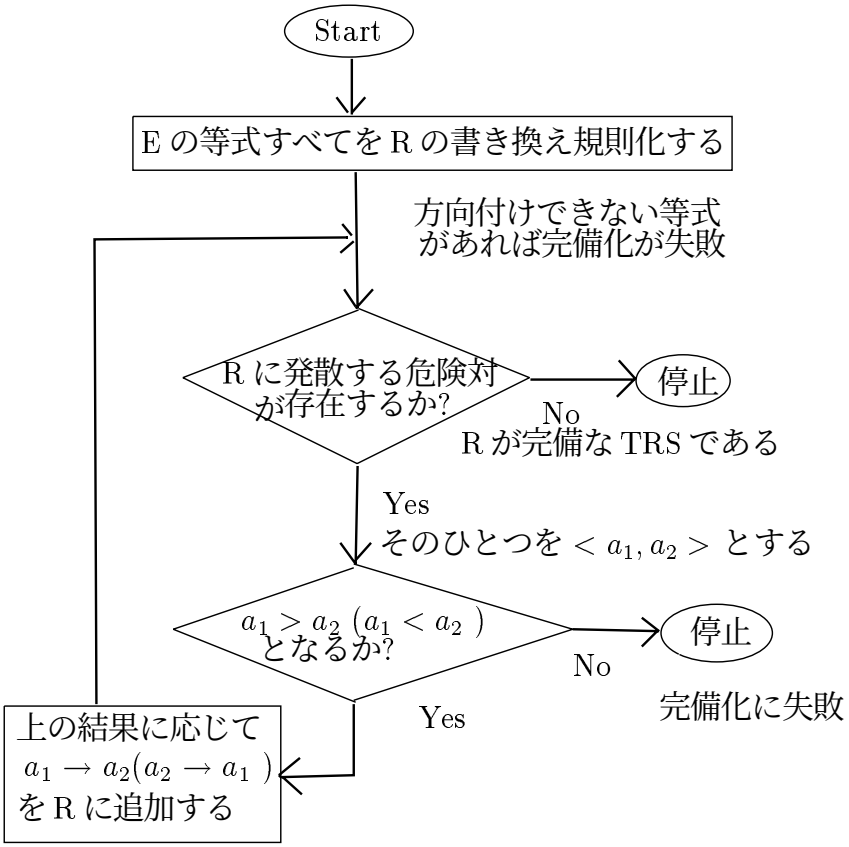
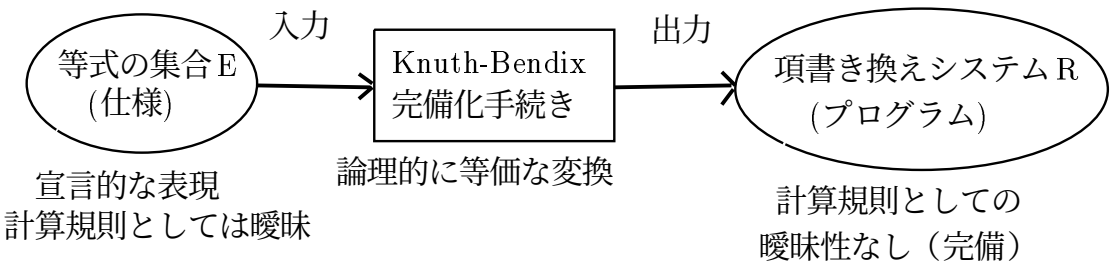


図 4-1 Knuth-Bendix アルゴリズム

初期状態では、常に等式のみが与えられ、書き換え規則は空であるとすると、Knuth-Bendix は等式による仕様を入力し、完備化された項書き換え系プログラムを出力する変換システムと見なされる。



Knuth-Bendix 完備化手続きを用いて、完備化させた例が文献 [12] に示されている。

例1　等式集合Eを３つの群の公理とし、それを完備化した例を示す。

$$\text{等式集合 } E_{group}: \begin{cases} 1 * x = x & \text{(左単位元の存在)} \\ I(x) * x = 1 & \text{(左逆元の存在)} \\ (x * y) * z = x * (y * z) & \text{(結合律)} \end{cases}$$

$$\Downarrow \text{Knuth-Bendix 完備化}$$

$$\text{項書き換え系 } R_{group}: \begin{cases} I(1) \to 1 \\ 1 * x \to x \\ x * 1 \to x \\ I(I(x)) \to x \\ I(x) * x \to 1 \\ x * I(x) \to 1 \\ I(x * y) \to I(y) * I(x) \\ (x * y) * z \to x * (y * z) \\ I(x) * (x * y) \to y \\ x * (I(x) * y) \to y \end{cases}$$

10 個の規則からなる完備な項書き換え系 R_{group} を得ることが出来る。

例 2　完備化手続きを用いてリストの反転関数 reverse を完備化した例を示す。

$$\text{等式集合 } E_{rev}: \begin{cases} []@y = y \\ (x :: y)@z = x :: (y@z) \\ reverse([]) = [] \\ reverse(x :: y) = reverse(y)@(x :: []) \\ reverse(reverse(x)) = x \end{cases}$$

$\Downarrow$ knuth-Bendix 完備化

$$\text{項書き換え系 } R_{rev}: \begin{cases} []@y \to y \\ (x :: y)@z \to x :: (y@z) \\ reverse([]) \to [] \\ reverse(reverse(x)) \to x \\ reverse(x :: y) \to reverse(y)@(x :: []) \\ reverse(x@(y :: [])) \to y :: reverse(x) \end{cases}$$

4.2　完備化手続きの順序付けについて

構文順序 (syntactical ordering) による方法は、関数記号上に半順序を導入し、項の構造と関数記号の大きさから 2 項の相対的な重さを比較するもので、単純化順序 (simplification ordering) の方法がその代表である。単純化順序による方法は、機械的な手順で 2 項の比較ができることから実装も容易で、複雑な項にも十分対応できる。そして、前提となる関数記号上の半順序は項の構造から逆に推論できるなど自動化に対して有益な方法である。そのために停止性保証の方法として広く用いられている。具体的な単純化順序としては、

- 再帰経路順序（recursive path ordering)
- 再帰分割順序 (recursive decomposition ordering)
- 辞書式経路順序 (lexicographic path ordering)
- 辞書式部分項順序 (lexicographic subterm ordering)

などがある。停止性能力については、どの方法が優位であるか判断できないので、単純化順序では無理の時、多項式解釈を持ち出すといったように、状況に応じてこれらの方法を使い分けるものである。本研究では、辞書式経路順序を使用している。

定義 **辞書式経路順序** *(lexicographic path ordering)*
辞書式経路順序は、与えられた整礎な関数記号間の半順序 $>_F$ から構成される。
項 $s = f(s_1, ..., s_n)$, $t = g(t_1, ..., t_m)$ ($n, m \geq 0$) 間の半順序 $s >_{lpo} t$ で、以下のどちらかの条件を満たす。

- $\exists i.s_i \geq_{lpo} t$
- $\forall i.s >_{lpo} t_i$ で、次のどちらかの条件を満たす。
  - $f >_F g$
  - $f = g$ で、$\exists i.s_1 = t_1, ..., s_{i-1} = t_{i-1}$, $s_i >_{lpo} t_i$

プログラム合成を実験する際の順序付けは、関数記号を一つずつ手動で行なったので、システムの改良により自動的に順序付けが出来るように開発する必要を感じる。また、同じプログラム合成の種類によって、順序付けが異なると、途中で合成に失敗したり、発散して停止しなくなる例が成功する例よりも多く観察された。

第 5 章

完備化手続きを用いた切り払い手法・プログラム合成

5.1 プログラム合成の実験について

Knuth-Bendix 完備化手続きによる切り払い手法を用いたプログラム合成は主に、長さを表す関数の集合 R_{length} 、和を表す関数の集合 R_{+} 、積を表す関数の集合 $R_{\times}$ にあらゆる関数記号を合成させた形のものを出力させて実験を行なった。その結果を次の節で述べる。これは、関数記号の順序付けを注意深く考慮した上で集合の組み合わせを吟味して融合項を作り、完備化させるものである。この実験を行った際、完備化せずに停止しないものや完備化する前に失敗してしまうケースが多く観察される。そこで、どうすれば完備化を成功させるかその法則性を発見するために 46 通りの入力データに対して、順序付けを変えることによって、731 通りのプログラム合成を Knuth-Bendix 完備化手続きのシステムを用いて実験を行なった。そのシステムは、関数型プログラミング言語 standard ML of New Jersey (1993) を用いて作成されたものである。そして、Knuth-Bendix 完備化の実験で用いた計算機は、主に Dell 社のノート型パソコン Inspiron 3000 であり、データ処理は ML of New Jersey (1994) 上で行なった。その実験において、121 通りの入力データに対して完備化手続きの出力が成功し、それぞれの出力データの中に生成規則が得られた。付録には、実験を行なったそのプログラムとその結果の一部を記載するが、ここでは特に成功した入力データとそれに対する順序付けやその入力データに対する出力データによって得られた生成規則について報告する。

5.2 プログラム合成に成功した実験例とその結果

5.2.1 関数記号 $length$ を合成させた例

例1.1 $length$ と @ はリストの長さと追加を求める関数とし、等式$E: h(x,y) = length(x@y)$を満たす関数 h のプログラム合成を試みる。

$$\mathrm{R}_{+}: \begin{cases} x+0 \to x \\ x+s(y) \to s(x+y) \end{cases}$$

$$R_{length}: \begin{cases} length[] \to 0 \\ length(x :: y) \to length(y) + s(0) \end{cases}$$

$$R_{@}: \begin{cases} []@y \to y \\ (x :: y)@z \to x :: (y@z) \end{cases}$$

h の順序を $@ > h > length$ と定めると合成規則 $length(x@y) \to h(x,y)$ が得られる。ここで、$R_{+} \cup R_{length} \cup R_{@} \cup R_{h} : \{length(x@y) \to h(x,y)\}$ に対して完備化を行なうと、手続きは成功して 8 ステップで以下の書き換え規則が合成される。

$h([],y) \to length(y)$
$h(x :: y, z) \to s(h(y,z))$

注意 h の順序を $h > length$ とすることが重要である。$length > h$ とすると、書き換え規則 $length(y) \to h([],y)$ が完備化によって生成されてしまうので、プログラム生成は失敗する。ここで提案したプログラム生成法は、F.Bellegarde と異なり t に対して、特に制限を設けていないので、融合変換が適用できない場合にもプログラム生成が可能である。

例1.2 リストの長さを表す関数 $length$ と追加を表す関数 @、そして $flat$ の再帰的プログラムの集合を合成させる。等式 $E: h(x) = length(flat(x))$ を満たす関数 h のプログラム合成を試みる。

$$\mathrm{R}_{+}:\begin{cases} x+0 \to x \\ x+s(y) \to s(x+y) \end{cases}$$

$$R_{length}:\begin{cases} length[] \to 0 \\ length(x :: y) \to length(y)+s(0) \end{cases}$$

$$R_{@}:\begin{cases} []@y \to y \\ (x :: y)@z \to x :: (y@z) \end{cases}$$

$$R_{flat}:\begin{cases} flat([]) \to [] \\ flat(x :: y) \to x@flat(y) \end{cases}$$

$R_{+} \cup R_{length} \cup R_{@} \cup R_{flat} \cup R_{h} : \{length(flat(x)) \to h(x)\}$ に対して完備化を行なうと、手続きは成功して以下の書き換え規則が合成される。

$h(x :: y) \to length(x@flat(y))$
$h([]) \to []$

注意　関数記号の順序付けを $h > flat > length > @ > + > [] >::> 0 > s$
または、$h > length > flat > @ > + > [] >::> 0 > s$ としなければいけない。
すると、このプログラム合成は 10 ステップで完備化する。順序付けを異なったものにすると、発散して停止しなくなる。

例 1.3　リストの長さを表す関数 $length$ と追加を表す関数 @、そして連結を表す関数 :: の再帰的プログラムの集合を合成させる。等式 $E : h(x, y, z) = length((x@y) :: z)$ を満たす関数 h のプログラム合成 $R_{+} \cup R_{length} \cup R_{@} \cup R_{h} : \{length((x@y) :: z) \to h(x, y, z)\}$ を試みると、$h > length > @ > + > [] >::> 0 > s$ の順序付けでのみ、8 ステップで完備化することがわかった。そして、以下の書き換え規則が生成される。

$h(y, z, x) \to s(length(x))$

例 1.4　リストの長さを表す関数 $length$ と追加を表す関数 @ 、そして、和を表す関数 $+$ の再帰的プログラムの集合を合成させる。等式 $E : h(x, y, z) = length((x@y) + z)$ を満たす関数 h のプログラム合成 $R_+ \cup R_{length} \cup R_@ \cup R_h : \{length((x@y) + z) \rightarrow h(x, y, z)\}$ を試みると、$h > length > @ > + > [] >::> 0 > s$ の順序付けでのみ、10 ステップで完備化できる。そして、以下のような書き換え規則が、生成される。

$h(x :: y, z, w) \rightarrow length((x :: (y@z)) + w)$
$h(x, y, z) \rightarrow length(s(x@y) + z)$
$h(x, y, 0) \rightarrow length(x@y)$
$h([], x, y) \rightarrow length(x + y)$

5.2.2　関数記号 $+$ を合成させた例

関数記号の $+$ の中に様々な書き換え規則を合成させた結果を次に示す。

例 2.1　$length$ と @ は、リストの長さと追加を求める関数とし、和を表す関数 $+$ と合成させる。例えば、等式 $E : h(x, y, z) = (length(x@y)) + z$ を満たす関数 h のプログラム合成を試みる。

$$\mathrm{R}_+ : \begin{cases} x + 0 \rightarrow x \\ x + s(y) \rightarrow s(x + y) \end{cases}$$

$$R_{length} : \begin{cases} length[] \rightarrow 0 \\ length(x :: y) \rightarrow length(y) + s(0) \end{cases}$$

$$R_@ : \begin{cases} []@y \rightarrow y \\ (x :: y)@z \rightarrow x :: (y@z) \end{cases}$$

$R_+ \cup R_{length} \cup R_@ \cup R_h : \{(length(x@y)) + z \to h(x,y,z)\}$ に対して完備化を行なうと、$h > @ > + > length > [] >::> 0 > s$ の順序付けで、10 ステップで完備化できることがわかった。そして、以下の書き換え規則が生成される。

$h(x :: y, z, w) \to s(length(y@z)) + w$
$h(x, y, s(z)) \to s(h(x, y, z))$
$h(x, y, 0) \to length(x@y)$
$\mathrm{h}([],\mathrm{x},\mathrm{y}) \to length(x) + y$

例 2.2　リストの長さを表す関数 $length$ と追加を求める関数 @ に、和を表す関数 + を合成させた例 2.1 に対して応用させた形を検証してみる。例えば、 等式 $E : h(x,y,z,w) = (length(x@y))+(z@w)$ を満たす関数 h のプログラム合成を試みる。ここで、例 2.1 の書き換え規則をそのまま用いる。$R_+ \cup R_{length} \cup R_@ \cup R_h : \{(length(x@y))+(z@w) \to h(x,y,z,w)\}$ に対して完備化を行なう。h の順序付けをを一番大きくもってくると完備化が成功する。

例えば、$h > length > @ >::> [] > + > 0 > s$ で、手続きは成功して以下のような融合規則が 10 ステップで合成される。

$h(x, y, z :: w, v) \to length(x@y) + (z :: (w@v))$
$h(x :: y, z, w, v) \to s(length(y@z)) + (w@v)$
$h([], x, y, z) \to length(x) + (y@z)$
$h(x, y, [], z) \to length(x@y) + z$

例 2.3　リストの長さを表す関数 $length$ と追加を求める関数 @ と連結を表す :: に、和を表す関数 + を合成させる。等式 $E : h(x,y,z,w) = (length(x@y) :: z) + w$ を満たす関数 h のプログラム合成 $R_+ \cup R_{length} \cup R_@ \cup R_h : \{(length(x@y) :: z) + w\}$ を試みると、$h > length > @ > [] >::> 0 > s$ の順序付けでのみ 8 ステップで完備化できることがわかった。そして、以下のような書き換え規則が生成される。

$h(z, w, x, y) \to s(length(x)) + y$

5.2.3 関数記号 $\times$ を合成させた例

関数記号の $\times$ の中に様々な書き換え規則を合成させた結果を次に示す。

例 3.1 $length$ と@はリストの長さと追加を求める関数とし、積を表す関数 $\times$ と合成させる。例えば、等式 $E : h(x, y, z) = (length(x@y)) \times z$ を満たす関数 h のプログラム合成を試みる。

$$\mathrm{R}_{\times} : \begin{cases} x \times 0 \to 0 \\ x \times s(y) \to (x \times y) + x \end{cases}$$

$$R_{+} : \begin{cases} x + 0 \to x \\ x + s(y) \to s(x + y) \end{cases}$$

$$R_{length} : \begin{cases} length[] \to 0 \\ length(x :: y) \to length(y) + s(0) \end{cases}$$

$$R_{@} : \begin{cases} []@y \to y \\ (x :: y)@z \to x :: (y@z) \end{cases}$$

$R_{\times} \cup R_{+} \cup R_{length} \cup R_{@} \cup R_h : \{(length(x@y)) \times z \to h(x, y, z)\}$ に対して完備化を行なうと、手続きは成功して以下のような書き換え規則が生成される。

$h(x :: y, z) \to s(length(y@z)) \times w$
$h(x, y, s(z)) \to h(x, y, z) + length(x@y)$
$h([], x, y) \to length(x) \times y$
$h(x, y, 0) \to 0$

注意　このプログラム合成は、関数記号の順序付けを$h > length > @ > \times > ::> [] > + > 0 > s$ とすると 12 ステップで完備化できる。
その他の順序付けだと、途中で完備化に失敗するか発散して停止しなくなる。

例 3.2　例 2.2 と類似した形の式を合成させる。リストの長さを表す関数 $length$ と追加を求める関数 $@$ に、積を表す関数 $\times$ を合成させた例 3.1 に対して応用させた形を検証してみる。例えば、等式 $E:\ h(x,y,z,w) = (length(x@y)) \times (z@w)$ を満たす関数 h のプログラム合成を試みる。ここで、例 3.1 の関数記号の規則の集合をそのまま用いる。$R\times \cup R_{+} \cup R_{length} \cup R_{@} \cup R_{h} : \{(length(x@y)) \times (z@w) \to h(x,y,z,w)\}$ に対して完備化を行なう。h の順序付けをを一番大きくもってくると完備化が成功する。

例えば、$h > length > @ > \times > ::> [] > + > 0 > s$ で、手続きは成功して以下のような融合規則が 12 ステップで合成される。例 2.2 では、10 ステップで完備化したが、生成された規則の形がよく似ている。

$h(x,y,z :: w,v) \to length(x@y) \times ((x :: w)@v)$
$h(x :: y,z,w,v) \to s(length(y@z)) \times (w@v)$
$h([],x,y,z) \to length(x) \times (y@z)$
$h(x,y,[],z) \to length(x@y) \times z$

例 3.3　$length$ と $@$ は、リストの長さと追加を求める関数であり、$::$ は連結を表す関数として、積を表す関数 $\times$ と合成させる。等式 $E:\ h(x,y,z,w) = (length(x@y) :: z) \times w$ を満たす関数 h のプログラム合成を試みる。

$R_{\times} \cup R_{+} \cup R_{length} \cup R_{@} \cup R_{h} : \{(length(x@y) :: z) \times w \to h(x,y,z,w)\}$　に対して完備化を行なうと、$h > \times > length > @ > + > [] > ::> 0 > s$ の順序付けでのみ、10 ステップで手続きは成功して以下のような書き換え規則が生成される。

$h(z, w, x, y) \to s(length(x)) \times y$

5.2.4 関数記号 $+$ と $\times$ を合成させた例

和を表す関数 $+$ の集合 R_+ と積を表す関数 $\times$ の集合 $R_\times$ を合成させた例を示す。

例4.1 和を表す関数 $+$ の集合 R_+ と積を表す関数 $\times$ の集合 $R_\times$ と等式 $E: h(x, y, z, w) = (x \times y) + (z \times w)$ を満たす関数 h のプログラム合成を試みる。

$$\mathrm{R}_+ : \begin{cases} x + 0 \to x \\ x + s(y) \to s(x + y) \end{cases}$$

$$R_\times : \begin{cases} x \times 0 \to 0 \\ x \times s(y) \to (x \times y) + x \end{cases}$$

$R_+ \cup R_\times \cup R_h : \{(x \times y) + (z \times w) \to h(x, y, z, w)\}$ に対して完備化を行なうと、$h > \times > + > s$ の順序付けでのみ 8 ステップで完備化が成功する。生成される書き換え規則は、以下の通りになる。

$h(x, s(y), z, w) \to ((x \times y) + x) + (z \times w)$
$h(x, y, z, s(w)) \to (x \times y) + ((z \times w) + z)$
$h(z, 0, x, y) \to 0 + (x \times y)$
$h(x, y, z, 0) \to x \times y$

例4.2 和を表す関数 $+$ の集合 R_+ と積を表す関数 $\times$ の集合 $R_\times$ と、等式 E: $h(x, y, z, w) = (x + y) \times (z + w)$ を満たす関数 h のプログラム合成を試みる。$R_+ \cup R_\times \cup R_h : \{(x + y) \times (z + w) \to h(x, y, z, w)\}$ に対して完備化を行なうと、例 4.1 と同様に $h > \times > + > s$ の

順序付けで完備化が 8 ステップで成功する。以下の書き換え規則が生成される。

$h(x, y, z, s(w)) \to h(x, y, z, w) + (x + y)$
$h(x, s(y), z, w) \to s(x + y) \times (z + w)$
$h(x, 0, y, z) \to x \times (y + z)$
$h(x, y, z, 0) \to (x + y) \times z$

例 4.3　和を表す関数記号 $+$ と積を表す関数記号 $\times$ を複雑に合成した例を示す。
例えば、R_+ と $R_\times$ と、等式 $E : h(x, y, z, w) = (x + (y \times z)) + w$ を満たす関数 h のプログラム合成を試みる。$R_+ \cup R_\times \cup R_h : \{(x + (y \times z)) + w \to h(x, y, z, w)\}$ に対して完備化を行なうと、$h > \times > + > s$ の順序付けで 8 ステップで完備化が成功する。以下の書き換え規則が生成される。

$h(x, y, s(z), w) \to (x + ((y \times z) + y)) + w$
$h(x, y, z, s(w)) \to s(h(x, y, z, w))$
$h(x, y, z, 0) \to x + (y \times z)$
$h(x, z, 0, y) \to x + y$

例 4.4　R_+ と $R_\times$ と、等式 $E : h(x, y, z, w) = (x + (y \times z)) \times w$ を満たす関数 h のプログラム合成を試みる。$R_+ \cup R_\times \cup R_h : \{(x + (y \times z)) \times w \to h(x, y, z, w)\}$ に対して完備化を行なうと、$h > \times > + > s$ の順序付けで 8 ステップで完備化が成功する。以下の書き換え規則が生成される。

$h(x, y, s(z), w) \to (x \times ((y \times z) + y)) + w$
$h(x, y, z, s(w)) \to s(h(x, y, z, w))$
$h(x, y, z, 0) \to x \times (y \times z)$
$h(x, z, 0, y) \to 0 + y$

5.2.5 関数記号 map を合成させた例

リストの各要素に関数を適用させる map を用いたプログラム合成を試みた。map とは、次のような再帰的プログラムで表現できる [13]。

```
map f [] = []
map f (x::xs) = fx :: Map f xs
```

その map を用いたプログラム合成の中で、成功した結果を次に示す。

例 5.1 等式 $E : h(x, y) = map(length(x) + y)$ を満たす関数 h のプログラム合成を試みる。

$$\mathrm{R}_{+} : \begin{cases} x + 0 \to x \\ x + s(y) \to s(x + y) \end{cases}$$

$$R_{length} : \begin{cases} length[] \to 0 \\ length(x :: y) \to length(y) + s(0) \end{cases}$$

$$R_{map} : \begin{cases} map(+([])) \to [] \\ map(+(x :: y)) \to +(x) :: map(+(y)) \end{cases}$$

$R_{+} \cup R_{length} \cup R_{map} \cup R_{h} : \{map(length(x) + y) \to h(x, y)\}$ に対して完備化を行なうと、例えば、$h > map > + >::> length > [] > 0 > s$ の順序付けなどにおいて、10 ステップで完備化が成功する。そして、次のような書き換え規則が生成する。

$h(z :: x, y) \to map(s(length(x)) + y)$
$h(x, s(y)) \to map(s(length(x) + y))$
$h([], x) \to map(0 + x)$
$h(x, 0) \to map(length(x))$

例 5.2　等式 $E : h(x, y) = map(length(x) \times y)$ を満たす関数 h のプログラム合成を試みる。

$$\mathrm{R}_{+} : \begin{cases} x + 0 \to x \\ x + s(y) \to s(x + y) \end{cases}$$

$$R_{\times} : \begin{cases} x \times 0 \to 0 \\ x \times s(y) \to (x \times y) + x \end{cases}$$

$$R_{length} : \begin{cases} length[] \to 0 \\ length(x :: y) \to length(y) + s(0) \end{cases}$$

$$R_{map} : \begin{cases} map(\times([])) \to [] \\ map(\times(x :: y)) \to \times(x) :: map(\times(y)) \end{cases}$$

$R_{+} \cup R_{\times} \cup R_{length} \cup R_{map} \cup R_{h} : \{map(length(x) \times y) \to h(x, y)\}$ に対して完備化を行なうと、例えば、$h > map > \times > + >::> length > [] > 0 > s$ の順序付けなどにおいて、12 ステップで完備化が成功する。そして、次のような書き換え規則が生成する。

$h(z :: x, y) \to map(s(length(x)) \times y)$
$h(x, s(y)) \to map((length(x) \times y) + length(x))$
$h([], x) \to map(0 \times x)$
$h(x, 0) \to map(0)$

5.2.6　非線型項のプログラム合成の例

F.Bellgarede の論文 [2] では、等式の融合項が線形でなければいけないといった制限があったが、Knuth-Bendix 完備化手続きにより、この条件を取り払って、等式の融合項が非線形項であるプログラム合成の実験を行なった。その結果を次に述べる。

例 6.1　等式 $E : d(x) = x + x$ のプログラム合成を考える。

$$\mathrm{R}_{+} : \begin{cases} x + 0 \to x \\ 0 + y \to y \end{cases}$$

$$R_s : \begin{cases} x + (s(y)) \to s(x + y) \\ (s(x) + y) \to s(x + y) \end{cases}$$

等式の右辺は、$x+x$ は非線形項であり、本来、F.Bellegarde の定義にある融合変換には、適用できない。しかし、Knuth-Bendix 完備化手続きにより、$d > + > s$ と、$+ > d > s$ に順序を定めて完備化を行なうと、合成規則 $R_d : x + x \to d(x)$ が得られる。そして、$R_{+} \cup R_s \cup R_d : \{x + x \to d(x)\}$ に対して完備化を行なうと 5 ステップで以下の書き換え規則が生成される。

$d(0) \to 0$
$d(s(x)) \to (s(s(d(x))))$

例 6.2
上記の例 5.1 の書き換え規則に関数 $+$ を一つ増やした非線形項を考える。新しい関数記号を t として、等式 $E : t(x) = x + (x + x)$ の合成を行なう。
$R_{+} \cup R_s \cup R_t : \{x + (x + x) \to t(x)\}$ に対して完備化を行なうと、$t > + > s$ と $+ > t > s$ の順序付けにより、5 ステップで完備化が成功する。そして、完備化したときに生成される規則は、以下のようになる。

$t(0) \to 0$
$t(s(x)) \to s(s(s(t(x))))$

例 6.3　和を表す関数記号　$+$ を３つに増やして等式をつくる。

- 等式 $E : q_1(x) = (x + x) + (x + x)$ のとき、
  $R_+ \cup R_s \cup R_{q_1} : \{(x + x) + (x + x)\}$ を合成して、完備化を行なうと以下の書き換え規則が生成される。順序付けは、$q_1 > + > s$ と $+ > q_1 > s$ のときのみ完備化が成功し、5 ステップで完備化される。

  $q_1(0) \to 0$
  $q_1(s(x)) \to s(s(s(s(q_1(x)))))$

- 等式 $E : q_2(x) = x + (x + (x + x))$　のとき、
  $R_+ \cup R_s \cup R_{q_2} : \{x + (x + (x + x))\}$ の完備化を行なうと、以下のような書き換え規則が生成される。順序付けは、同じく $q_2 > + > s$ と $+ > q_2 > s$ のときのみ完備化が成功する。これも、5 ステップで完備化される。

  $q_2(0) \to 0$
  $q_2(s(x)) \to s(s(s(s(q_2(x)))))$

例 6.4　融合項に、$\times$ を含み、非線形項である等式 $E : d(x) = x \times x$ のプログラム合成を考える。

$$\mathrm{R}_+ : \begin{cases} x + 0 \to x \\ 0 + y \to y \end{cases}$$

$$R_s : \begin{cases} x + (s(y)) \to s(x + y) \\ (s(x) + y) \to s(x + y) \end{cases}$$

$$R_\times : \begin{cases} x \times 0 \to 0 \\ x \times s(y) \to x \times y + x \end{cases}$$

$R_\times \cup R_+ \cup R_s \cup R_d : \{x \times x \to d(x)\}$ に対して完備化を行なうと、$d > \times > + > s$　の順序付けでのみ 6 ステップで完備化が成功する。それ以外の順序付けであると発散して、停止しない。完備化が成功すると以下の書き換え規則が生成する。

$d(0) \to 0$
$d(s(x)) \to s((s(x) \times x) + x)$

例 6.5　例 6.4 の等式 E に、関数記号を 2 つに増やして、右辺が非線形項である等式をプログラム合成する。

- 等式 $E : d(x) = x + (x \times x)$ のとき、

  $R_\times \cup R_+ \cup R_s \cup R_d : \{x + (x \times x) \to d(x)\}$ に対して完備化を行なうと、$d > + > \times > s$ の順序付けでのみ 7 ステップで完備化が成功する。それ以外の順序付けであると発散して、停止しない。完備化が成功すると以下の規則が生成する。

  $d(0) \to 0$
  $d(s(x)) \to s(s(x + ((s(x) \times x) + x)))$

- 等式 $E : d(x) = (x + x) \times x$ のとき、

  $R_\times \cup R_+ \cup R_s \cup R_d : \{(x + x) \times x \to d(x)\}$ に対して完備化を行なうと、$d > \times > + > s$ の順序付けでのみ 7 ステップで完備化が成功する。それ以外の順序付けであると発散して、停止しない。完備化が成功すると以下の書き換え規則が生成する。

  $d(0) \to 0$
  $d(s(x)) \to s(s((s(s(x + x))) \times x) + (x + x))$

例6.6　例6.4の等式Eに、関数記号を3つに増やして、右辺が非線形項である等式をプログラム合成する。

- 等式 $E:\ d(x) = (x \times x) + (x \times x)$ のとき、

  $R_{\times} \cup R_{+} \cup R_{s} \cup R_{d}\ :\ \{(x \times x) + (x \times x)\ \rightarrow\ d(x)\}$ に対して完備化を行なうと、$d > \times > + > s$　の順序付けでのみ7ステップで完備化が成功する。それ以外の順序付けであると発散して、停止しない。完備化が成功すると以下の書き換え規則が生成する。

  $d(0) \rightarrow 0$
  $d(s(x)) \rightarrow s(s((s(x) \times x) + x) + ((s(x) \times x) + x))$

- 等式 $E:\ d(x) = (x + x) \times (x + x)$ のとき、

  $R_{\times} \cup R_{+} \cup R_{s} \cup R_{d}\ :\ \{(x + x) \times (x + x)\ \rightarrow\ d(x)\}$ に対して完備化を行なうと、$d > \times > + > s$　の順序付けでのみ7ステップで完備化が成功する。それ以外の順序付けであると発散して、停止しない。完備化が成功すると以下の書き換え規則が生成する。

  $d(0) \rightarrow 0$
  $d(s(x)) \rightarrow s(s(s(s(x + x)) \times (x + x)) + (x + x) + (x + x))$

- 等式 $E:\ d(x) = (x \times x) + (x + x)$ のとき、

  $R_{\times} \cup R_{+} \cup R_{s} \cup R_{d}\ :\ \{(x \times x) + (x + x)\ \rightarrow\ d(x)\}$ に対して完備化を行なうと、$d > \times > + > s$ の順序付けでのみ7ステップで完備化が成功する。完備化が成功すると次のような書き換え規則が生成する。

  $d(0) \rightarrow 0$

$d(s(x)) \to s(s(s((((s(x) \times x) + x) + (x + x)))))$

- 等式 $E : d(x) = (x \times x \times x) + x$ のとき、

  $R_\times \cup R_+ \cup R_s \cup R_d : \{(x \times x \times x) + x \to d(x)\}$ に対して完備化を行なうと、$d > \times > + > s$ の順序付けでのみ 7 ステップで完備化が成功する。完備化が成功すると次のような書き換え規則が生成する。

  $d(0) \to 0 \times 0 \times 0$
  $d(s(x)) \to s((s(x) \times s(x) \times s(x)) + x)$

- 等式 $E : d(x) = (x + x + x) \times x$ のとき、

  $R_\times \cup R_+ \cup R_s \cup R_d : \{(x + x + x) \times x \to d(x)\}$ に対して完備化を行なうと、$d > \times > + > s$ の順序付けでのみ 7 ステップで完備化が成功する。完備化が成功すると次のような書き換え規則が生成する。

  $d(0) \to 0$
  $d(s(x)) \to ((s(x) + s(x) + s(x)) \times x) + (s(x) + s(x) + s(x))$

5.3 プログラム合成に失敗した実験例

5.2 節では、プログラム合成が成功した例を示したが、本節では、合成に失敗した例について述べる。

例 1 特に、例 1.2、2.2、3.1、3.2、3.3、4.1、4.2、4.3、4.4 において、新しく定義する関数記号 h の順序付け一番大きく設定しなければ、プログラム合成に失敗することがわかっ

た。その他のプログラム合成の例に関しても、比較的に同様のことが言える。

例 2　リストの長さを表す $length$ と追加を表す @ 、反転を表す $reverse$ などの関数を用いて、等式 $E:\ h(x,y) = length(reverse(x,y))$ を満たす関数 h のプログラム合成を試みる。

$$\mathrm{R}_{+}:\begin{cases} x+0 \to x \\ x+s(y) \to s(x+y) \end{cases}$$

$$R_{length}:\begin{cases} length[] \to 0 \\ length(x :: y) \to length(y)+s(0) \end{cases}$$

$$R_{@}:\begin{cases} []@y \to y \\ (x :: y)@z \to x :: (y@z) \end{cases}$$

$$R_{reverse}:\begin{cases} reverse(x :: y) \to reverse(y@(x :: [])) \\ reverse(reverse(x)) \to x \end{cases}$$

$R_{+} \cup R_{length} \cup R_{@} \cup R_{reverse} \cup R_{h} : \{length(reverse(x,y)) \to h(x)\}$ に対して完備化を行なうと、$length(reverse(x,y)) \to h(x,y)$ の書き換え規則しか合成されない。新たな生成規則が得られないので、この例は失敗だと見なされる。

例 3　融合項が非線形項であっても、成功する例を前節で述べたが、完備化できないで失敗する例も見られる。例えば、5.2.1 の例 1.1 の $h(x,y) = length(x@y)$ の等式では 8 ステップで完備化に成功しても、それを非線形にした $d(x) = length(x@x)$ を満たす関数 $d(x)$ のプログラム合成は完備化できない。

$$\mathrm{R}_{+}:\begin{cases} x+0 \to x \\ 0+y \to y \end{cases}$$

$$R_s:\begin{cases} x+(s(y)) \to s(x+y) \\ (s(x)+y) \to s(x+y) \end{cases}$$

$$R_{length}:\begin{cases} length[] \to 0 \\ length(x :: y) \to length(y)+s(0) \end{cases}$$

$$R_{@}:\begin{cases} []@y \to y \\ (x :: y)@z \to x :: (y@z) \end{cases}$$

$R_+ \cup R_s \cup R_{length} \cup R_@ \cup R_h : \{length(x@x) \to d(x)\}$ に対して完備化を行なうと、停止せずにループに陥るか、途中で失敗する場合が多く観察される。完備化に成功した場合、7 ステップで $length(append(x)) \to d(x)$ の規則が生成されるが、$R_h : length(x@x) \to d(x)$ の形ではないのでプログラム合成ができない。

- 等式 $E : d(x) = x@x$ のプログラム合成の成功例も、まだ観察されていない。$R_@ \cup \{x@x \to d(x)\}$ に対して完備化手続きを行なっても、生成規則が得られるような順序付けや合成する良い関数の集合が見つかっていない。

- 等式 $E : d(x) = length(x + x)$ のプログラム合成も、成功した例が観察されていない。

例 4　map 関数のプログラム合成に関して、5.2.5 節で成功した例を示したが、合成できない例を示す。等式 $E : h(x, y, z) = x + map(length(y :: z))$ を合成させる時、次の関数の集合ではうまくいかないことがわかった。

$$\mathrm{R}_{+}:\begin{cases} x+0 \to x \\ 0+y \to y \end{cases}$$

$$R_s:\begin{cases} x+(s(y)) \to s(x+y) \\ (s(x)+y) \to s(x+y) \end{cases}$$

$$R_{length}:\begin{cases} length[] \to 0 \\ length(x :: y) \to length(y)+s(0) \end{cases}$$

$$R_{map}:\begin{cases} map(length[]) \to [] \\ map(length(x :: y)) \to length(x) :: map(length(y)) \end{cases}$$

$R_+ \cup R_s \cup R_{length} \cup R_{map} \cup R_h : \{x + map(length(y :: z)) \to h(x, y, z)\}$ に対して完備化手続きを行なうと、成功しない。

第 6 章

まとめ

6.1 本論文のまとめ

本研究では、主に Knuth-Bendix 完備化手続きを用いて再帰的なプログラムの形をした関数の集合の合成を行ない、切り払いのための融合規則が得られるか否かの実験を繰り返してきた。これまでの結果を順序付けに着目して考察してみると、線形項の融合項の合成に関して言えば、新しく導入する関数記号 (Fresh symbol) は、一番上に持ってくると成功しやすいことがわかる。そして、順序付けが異なるが完備化が成功する例を見てみると、完備化するまでのステップ数が異なることが多い。そして、実際は、順序付けが異なると同じプログラム合成でも、完備化される途中で失敗して停止したり、発散して停止しなくなったりすることが多い。このようにして、順序付けがプログラム合成の完備化に対して、大きく左右していることがよくわかる。また、合成を行なう時、その再帰的プログラムの形や融合項の構造に着目して、プログラム合成の完備化が可能かどうかが判断できるように、第 3 章に触れたような表記を用いて、項書き換え系の計算モデルとして理論的に解析できると考えられる。

6.2 今後の課題

今回の実験では、Knuth-Bendix 完備化手続きを行なう際に、一つずつ手動で関数記号の順序付けを設定して行なった。この研究の今後の課題の一つとして、用いた完備化手続きのシステムを改良して、順序付けが自動化できると更に研究が早く進められる。しか

し、一度にすべての順序付けについて調べるとなると、関数記号が n 個ある場合、$n!$ 通りの組み合わせが考えられるので、莫大な出力になる。だから、ある固定された順序付けに対して、一つの関数記号を一つずつ上からずらしていくような自動化であると、n 個の関数記号に対して、 n 通りの組み合わせの出力になる。そのようなシステムの改良だと実験の効率化が図れる。

本研究では、主に融合規則 (Fusion rule) を導出させて、プログラム合成の実験を行なったが、文献 [2] にあるようなタプリング規則 (Tupling rule) や 2 回の融合 (Secondary fusion) の実験を Knuth-Bendix 完備化手続きで行なえると考えられる。今のところ、このような戦略で停止するプログラム合成は、まだ行なっていない。そして、プログラム合成に対して、第 3 章に記した項書き換え系での定義・表現を用いた項書き換えモデルとしての理論的な考察を進めることが、今後の課題として残っている。後、切り払い手法としての未解決な問題である高階プログラムへの適用や項書き換え系での切り払いの推論規則の新しい提案などが、研究課題として残っている。

謝辞

本研究を行なうのにあたって、終始変わらぬ適切な御指導を賜りました外山芳人教授に心から感謝致します。そして、良き助言でもって励まして戴いた鈴木大郎助手にも感謝致します。また、輪講や論文紹介などを通して、学業を共にした外山研究室の皆さんと北陸にいる私を影ながら暖かく支えてくれた実家の父母に感謝の意を表します。そして、何より、研究のために計算機などの環境を整えて戴いた福井工業大学の佐村敏治博士にお礼を申し上げます。

参考文献

[1] P.L.Wadler, "Deforestation: transforming programs to eliminate trees", Theoretical Computer Science 73, pp.231-248, 1990.

[2] F.Bellegarde, "Automating synthesis by completion", Journees Francophones des Langages Applicatifs, pp.177-203, 1995.

[3] F.Bellegarde, "Program trasformation and Rewriting", Rewriting Techniques and Applications, pp.226-239, 1991.

[4] Burstall, R.M. & Darlington, J., "A transformation system for developing recursive programs", J.ACM 24(1), pp.44-67, 1977.

[5] 佐賀正芳, "関数型プログラムにおけるプログラム変換の研究", 平成9年度北陸先端科学技術大学院大学 修士論文, 1998.

[6] A.J.Gill, "Cheap deforestation for non-strict fuctional languages", PhD thesis, University of Glasgow, 1996.

[7] 吉政洋美, 外山芳人, "完備化手続きを用いたプログラム合成", 平成10年度電気関係学会北陸支部連合大会講演論文集(E-7), p.257, 1998.

[8] Jan Willem Klop, Aart Middeldorp, "An Introduction to Knuth-Bendix Completion", pp.31-52.

[9] 井田 哲雄 編, "新しいプログラミング・パラダイム", 共立出版 1989.

[10] 田中 二郎, 井田 哲雄 編, "続・新しいプログラミング・パラダイム", 共立出版 1990.

[11] 井田 哲雄, "計算モデルの基礎理論", 岩波書店, 1991.

[12] 外山芳人, “項書き換えシステム入門”, 電子情報通信学会技術研究報告, SS98-15, pp.32-38, 1998.

[13] Bird.R, P.L.Wadler, “Introduction to Functional Programming”, Prentice-Hall, 1988. (武市正人訳 「関数プログラミング」近代科学社,1991).

[14] 淵 一博監修, “プログラム変換”, 共立出版社, 1987.

付録

本研究ではDell社のノート型パーソナルコンピュータInspiron 3000において、関数型プログラミング言語Standard ML of New Jersey (1993)で作成したKnuth-Bendix完備化手続きを実装して実験を行なった。その時に、実際に使った入力データと得られた出力データについての一部の例を記載する。

例1 $length(x@y) \to h(x,y)$ を合成させるKnuth-Bendix完備化の実験例

実際に実行を行なった入力データ

```
  fun kbtest 1  =
       KB.kb (Order.grter_lpo [("H",8),("Plus",7),("Append",6),("Length",5),
 ("Cons",4),("S",3),("0",2),("Nill",1)])
(IO.readeqs
     [ "Length(Append(x, y)) = H(x, y)" ])
     (IO.readeqs
     [ "Plus(x, 0) = x" ,
       "Plus(x ,S(y)) = S(Plus(x , y))",
       "Length(Nil) = 0",
       "Length(Cons(x,y)) = Plus(Length(y), S(0))",
       "Append(Nil, y) = y",
       "Append(Cons(x, y), z) = Cons(x, Append(y, z))" ])
```

Knuth-Bendix 完備化手続きを実行した時の出力データ

```
- kbtest 1;
***  0 ステップ目 ***

E =
[ Length(Append(x, y)) = H(x, y) ]

R =
[ Plus(x, 0) -> x
, Plus(x, S(y)) -> S(Plus(x, y))
, Length(Nil) -> 0
, Length(Cons(x, y)) -> Plus(Length(y), S(0))
, Append(Nil, y) -> y
, Append(Cons(x, y), z) -> Cons(x, Append(y, z)) ]

*** 1 ステップ目 ***

E =
[ Cons(x, Append(y, z)) = Append(Cons(x, y), z)
, y = Append(Nil, y)
, Plus(Length(y), S(0)) = Length(Cons(x, y))
, Plus(x, 0) = x
, S(Plus(x, y)) = Plus(x, S(y)) ]

R =
[ Length(Nil) -> 0
, Length(Append(x, y)) -> H(x, y) ]

*** 2 ステップ目 ***
```

```
E =
[ Plus(x, S(y)) = S(Plus(x, y))
, x = Plus(x, 0)
, Length(Cons(x, y)) = Plus(Length(y), S(0))
, Append(Cons(x, y), z) = Cons(x, Append(y, z))
, Length(x) = H(Nil, x) ]

R =
[ Append(Nil, y) -> y
, Length(Nil) -> 0
, Length(Append(x, y)) -> H(x, y) ]

*** 3 ステップ目 ***

E =
[ H(Nil, x) = Length(x)
, Cons(x, Append(y, z)) = Append(Cons(x, y), z)
, Plus(Length(y), S(0)) = Length(Cons(x, y))
, Plus(x, S(y)) = S(Plus(x, y)) ]

R =
[ Plus(x, 0) -> x
, Append(Nil, y) -> y
, Length(Nil) -> 0
, Length(Append(x, y)) -> H(x, y) ]

*** 4 ステップ目 ***

E =
```

```
[ S(Plus(x, y)) = Plus(x, S(y))
, Length(Cons(x, y)) = Plus(Length(y), S(0))
, Append(Cons(x, y), z) = Cons(x, Append(y, z)) ]

R =
[ H(Nil, x) -> Length(x)
, Plus(x, 0) -> x
, Append(Nil, y) -> y
, Length(Nil) -> 0
, Length(Append(x, y)) -> H(x, y) ]

*** 5 ステップ目 ***

E =
[ Cons(x, Append(y, z)) = Append(Cons(x, y), z)
, S(Length(y)) = Length(Cons(x, y)) ]

R =
[ Plus(x, S(y)) -> S(Plus(x, y))
, H(Nil, x) -> Length(x)
, Plus(x, 0) -> x
, Append(Nil, y) -> y
, Length(Nil) -> 0
, Length(Append(x, y)) -> H(x, y) ]

*** 6 ステップ目 ***

E =
[ Append(Cons(x, y), z) = Cons(x, Append(y, z)) ]
```

```
R =
[ Length(Cons(x, y)) -> S(Length(y))
, Plus(x, S(y)) -> S(Plus(x, y))
, H(Nil, x) -> Length(x)
, Plus(x, 0) -> x
, Append(Nil, y) -> y
, Length(Nil) -> 0
, Length(Append(x, y)) -> H(x, y) ]
```

*** 7 ステップ目 ***

```
E =
[ S(H(y, z)) = H(Cons(x, y), z) ]

R =
[ Append(Cons(x, y), z) -> Cons(x, Append(y, z))
, Length(Cons(x, y)) -> S(Length(y))
, Plus(x, S(y)) -> S(Plus(x, y))
, H(Nil, x) -> Length(x)
, Plus(x, 0) -> x
, Append(Nil, y) -> y
, Length(Nil) -> 0
, Length(Append(x, y)) -> H(x, y) ]
```

*** 8 ステップで完備化に成功しました! ***

```
R =
[ H(Cons(x, y), z) -> S(H(y, z))
, Append(Cons(x, y), z) -> Cons(x, Append(y, z))
, Length(Cons(x, y)) -> S(Length(y))
```

```
, Plus(x, S(y)) -> S(Plus(x, y))
, H(Nil, x) -> Length(x)
, Plus(x, 0) -> x
, Append(Nil, y) -> y
, Length(Nil) -> 0
, Length(Append(x, y)) -> H(x, y) ]
```

以下の例では、途中の実行過程に関しては省略し、入力データと得られた生成規則の出力データの結果のみを示す。

例 2　$length((x@y)+z) \to h(x,y,z)$ を合成させる Knuth-Bendix 完備化の実験例

Knuth-Bendix 完備化手続きを行なった入力データ

```
fun kbtest 0 =
    KB.kb (Order.grter_lpo  [("H",8),("Length",7),("Append",6),
("Plus",5),("Nil",4),("Cons",3),("0",2),("S",1)]  )
    (IO.readeqs
     [ "Length(Plus(Append(x, y),z)) = H(x, y, z)" ])
     (IO.readeqs
     [ "Plus(x, 0) = x" ,
       "Plus(x ,S(y)) = S(Plus(x , y))",
       "Length(Nil) = 0",
       "Length(Cons(x,y)) = Plus(Length(y), S(0))",
       "Append(Nil, y) = y",
       "Append(Cons(x, y), z) = Cons(x, Append(y, z))" ])
```

実際に得られた生成規則を含む出力データ

```
*** 10 ステップで完備化に成功しました! ***

R =
[ H(Cons(x, y), z, w) -> Length(Plus(Cons(x, Append(y, z)), w))
, H(x, y, S(z)) -> Length(S(Plus(Append(x, y), z)))
, Append(Cons(x, y), z) -> Cons(x, Append(y, z))
, Length(Cons(x, y)) -> S(Length(y))
, Plus(x, S(y)) -> S(Plus(x, y))
, H(x, y, 0) -> Length(Append(x, y))
, H(Nil, x, y) -> Length(Plus(x, y))
, Plus(x, 0) -> x
, Append(Nil, y) -> y
, Length(Nil) -> 0
, Length(Plus(Append(x, y), z)) -> H(x, y, z) ]
```

例3　非線形項　$x + x \to d(x)$ を合成させる Knuth-Bendix 完備化の実験例

Knuth-Bendix 完備化手続きを行なった入力データ

```
fun kbtest 1 =
    KB.kb (Order.grter_lpo [("D",3),("Plus",2),("S",1)])
    (IO.readeqs
     [ "Plus(x, x) = D(x)" ])
    (IO.readeqs
     [ "Plus(x, 0) = x",
       "Plus(y, 0) = y",
       "Plus(x,S(y)) = S(Plus(x,y))",
       "Plus(S(x),y) = S(Plus(x,y))"])
```

実際に得られた生成規則を含む出力データ

```
*** 5 ステップで完備化に成功しました! ***

R =
[ Plus(x, S(y)) -> S(Plus(x, y))
, D(S(x)) -> S(S(D(x)))
, Plus(S(x), y) -> S(Plus(x, y))
, D(0) -> 0
, Plus(x, 0) -> x
, Plus(x, x) -> D(x) ]
```