変換戦略の記述に基づくプログラムの自動生成システムの実装

横山哲郎[†] 篠埜功[††]
胡振江[†,†††] 武市正人[†]

プログラムの自動生成においては，変換規則の汎用性を高めるために，変換規則と変換規則の適用順序を制御することが重要である．変換規則と変換戦略を記述する理論的枠組である CCP(Calculation Carrying Program) についてはすでに報告しているが，筆者らの知る限りにおいては計算機上で稼動されたという報告はなく，CCP による大きなプログラムの変換についての報告もない．本論文では，CCP の処理系を実装する方法を考察し，計算機上で実際に最大マーク付け問題のプログラムの自動生成について検討を行い，本システムの有効性を示す．

Automatic Generation of Programs based on High Level Strategy Description

Tetsuo Yokoyama,[†] Isao Sasano,[††] Zhenjiang Hu[†,†††] and Masato Takeichi[†,††]

To relax the tension between clarity and efficiency in programming, we have proposed a theoretical framework called calculation carrying program, which accompanies straightforward specification with calculation specifying the intention (strategy) in a highly abstract way. In this paper, we give its first implementation, showing the system which not only automatically derives efficient programs from initial inefficient specification, but also interactively helps programmers to debug derivation steps. Furthermore, to show its power, we demonstrate how to use our system to generate efficient programs for solving maximum marking problems.

1. はじめに

高性能なソフトウェア開発においてプログラム変換は大きな役割を果たしている[1)]．プログラム変換は，変換規則をプログラムに順次適用することでプログラムの意味を変えずに，実行ステップと記憶容量の点で効率を向上させる．プログラムの自動生成システムを構築するには，効果的な変換規則が必要である．一般に，効果的な変換規則とは次の 2 つの性質をもつものである．第一に，多くのプログラムのパターンに適用できる汎用性があること．第二に，実装を可能にするために変換規則が構成的であり現実的な時間で規則の適用が可能であること．例えば，「素数 p, q が存在して $n = p * q$ を満たすならば, n を $p * q$ に書き換える」という規則は効果的な変換規則とはいえない．汎用的な規則ではあるもののこの規則はどのように素数 p，q を求めるかについて言及しておらず，構成的でないからである．

また，変換規則をどのような順序でどの部分に適用していくかという変換規則の適用の仕方を制御する変換戦略をプログラムの自動生成システムに伝える必要がある．既存のプログラムの自動生成システムのほとんどはあらかじめシステムに用意されている固定された変換戦略を用いて変換規則の適用を行っている．例えば，MAG システム[2)] は変換戦略をユーザが陽に指定できないプログラムの自動生成システムであるが，プログラム変換の停止性を意識して変換規則の記述を行わなければならず，ユーザの意図した導出をすることは難しい．

この問題を解決する直接的な方法は，1 ステップのプログラム変換を変換規則と変換規則の適用場所に

† 東京大学 大学院 情報理工学系研究科 数理情報学専攻
Department of Mathematical Informatics, Graduate School of Information Science and Technology, University of Tokyo
†† 東京大学 大学院 工学系研究科 情報工学専攻
Department of Information Engineering, School of Engineering, University of Tokyo
††† さきがけ研究 21，科学技術振興事業団
PRESTO21, Japan Science and Technology Corporation

よって表現し，それらを順に並べて連続するプログラム変換を表すものである．本論文では，まずこの方法を実現する変換スクリプトを提案する．この方法では変換規則の適用場所，適用順序をすべて明示的に指定するので，プログラム変換の停止性を意識して変換規則を記述する必要性がなくなるという利点がある．筆者らは MAG システムを対話的実行の点で拡張し，変換スクリプトの処理系の実装を行った．

しかし，プログラム変換を 1 ステップずつ指定することは煩雑であるので，変換規則をどのように適用するのかを簡潔に指定できる変換戦略記述言語[3)4)] が必要である．従来の変換戦略記述言語[4)] の多くは 1 階マッチング機能しかなく，直接に記述可能な変換規則が限定されている．本論文では，メタ言語である運算付随プログラム[3)]（Calculation Carrying Program，以下 CCP と略記）によって変換戦略を記述することとした．CCP は高階マッチング機能をもつので広範囲に適用可能な変換規則を簡潔に記述できる．プログラム変換の制御をしやすく，直観的に理解しやすい形で変換戦略を記述できるので，CCP は変換戦略の記述に向いているといえる．

本論文では，これらの変換戦略の実現とデバッギングを支援する環境の実現を示す．また，変換戦略の記述に基づくプログラムの自動生成システムの有効性を，最大マーク付け問題[5)] の例を通して示す．

最大マーク付け問題はさまざまな問題を含み，多くの研究がなされている[5)6)7)8)9)10)]．入力としては，要素に重みの与えられたある再帰データが与えられる．その入力データ中のいくつかの要素にマークをつけるという操作を考え，これをマーク付けと呼ぶ．最大マーク付け問題とは，性質 p を満たすマーク付けの中で，重み関数 w の値が最も大きなものを 1 つ見つける問題である．実際，最大マーク付け問題を解く効率のよいプログラムの自動生成は意義深いことで，多くの研究者を魅了してきた．例えば，文献 7) では線形プログラムの自動生成法が示されている．しかし，生成される線形プログラムは巨大なサイズのテーブルを持っており，まったく実用にならない．本論文では，筆者らが開発したプログラムの自動生成システムで最大マーク付け問題を解く効率的プログラムを自動生成できることを示す．

最大マーク付け問題の例

例えば，最大マーク付け問題の一例である最大部分列和問題[11)] を考えてみる．最大部分列和問題とは，与えられた列中の連続する部分列の中で和が最大となるものを返す問題である．例えば，

$$[1, -2, 4, -1, -2, 5, -1]$$

というリストが与えられたときの最大部分列和問題の解は

$$[4, -1, -2, 5]$$

となる．要素にマークを付け，部分列を表すことによって，この最大部分列和問題を最大マーク付け問題として定式化することができる．具体的には性質 p をマークの付けられた要素が連続しているかを判定する関数，重み関数 w をマークの付けられた要素の和を求める関数とすればよい．

2. 仕様の記述と変換規則

本節では，仕様の記述に用いる表記法を述べ，最大マーク付け問題の例を通して仕様記述法を説明する．また，その仕様から線形時間アルゴリズムを得る変換規則を示し，それを用いて，変換規則について説明する．

2.1 表 記 法

本論文では 5) での表記法にあわせ関数型言語 Haskell[12)] に類似した記法を用いる．

2.1.1 関数と演算子

関数適用は括弧を省略した形で表記する．つまり数学でよく使用する表記法の $f(x)$ ではなく，$f\ x$ と表記する．関数はカリー化 (currying) する．カリー化された関数適用は左結合的である．例えば $f\ a\ b = (f\ a)\ b$ である．関数適用は 2 項演算子よりも結合の順序が高いものとする．例えば，$f\ a \oplus b$ は括弧を明示的に表すと，$(f\ a) \oplus b$ である．2 項演算子は $\oplus, \otimes, \ominus, \oslash, \odot$ などで表す．2 項演算子はセクション (section) 化により

$$x \oplus y = (x\oplus)\ y = (\oplus y)\ x = (\oplus)\ x\ y$$

のように関数として用いることができる．関数合成は $\circ$ で表し，

$$(f \circ g)\ x = f\ (g\ x)$$

と定義する．関数合成は結合的である．

2.1.2 リ ス ト

リストは同一の型の要素を一次元的に並べたデータである．n 個の要素 $a_1, a_2, ..., a_n$ からなるリストは $[a_1, a_2, ..., a_n]$ と表記する．0 個の要素のリスト，空

リストは [ ] と表記する．要素は $a, b, c, \ldots$，リストは $x, y, z, \ldots$，リストのリストは $xs, ys, zs, \ldots$，リストのリストのリストは $xss, yss, zss, \ldots$ のように表記する．2 つのリスト x, y の連接は演算子 $+\!\!+$ を用い $x +\!\!+ y$ と表記する．例えば

$$[1] +\!\!+ [3,4] +\!\!+ [2] = [1,3,4,2]$$

である．また内包表記は，集合を記述する数学の形式をもとにした構文を用いるものであり，

$$[x * x \mid x \leftarrow [1,2,3,4,5,6,7,8,9], even\ x]$$

の値は

$$[4, 16, 36, 64]$$

である．

2.1.3 再帰データ型

本論文では，次の形で定義される再帰データ型を扱う．

$$\begin{array}{lcl} D\ \alpha & = & C_1\ (\alpha, D_1, \ldots, D_n) \\ & | & C_2\ (\alpha, D_1, \ldots, D_n) \\ & | & \cdots \\ & | & C_k\ (\alpha, D_1, \ldots, D_n) \end{array}$$

$D\ \alpha$ は定義される型，α は型変数を表す．D_i は $D\,\alpha$ を表す．C_i の引数の $D\,\alpha$ の個数が n 個であることを表すためにこのように表記する．C_i はデータ構成子と呼ばれ，α 型の要素と，ある決まった個数の再帰データ $D_1\ \ldots\ D_n$ から $D\ \alpha$ 型のデータを構成する．

特に，C_i の引数が 1 つのときは $C_i\ (\alpha, _, \ldots, _)$，引数が 0 のときは $C_i\ (_, _, \ldots, _)$ であり，便宜上，それぞれ $C_i\ \alpha$，C_i で表す．$_$ は don't care を表す．

例えば，リストのデータ型は

$$List\ \alpha = Nil \mid Cons\ (\alpha, List\ \alpha)$$

と表せる．本論文の表記法では，Nil は [ ]，$Cons$ は $(:)$ にあたる．これらのデータ構成子によるリスト $[a_1, a_2, \ldots, a_n]$ の表記は $(a_1 : (a_2 : \cdots (a_n : [\,]) \cdots))$ である．

2.1.4 Catamorphism

再帰データ上の基本的再帰関数のクラスに catamorphism がある．例えば，リスト上の catamorphism は，以下のように定義される関数を表す．

$$\begin{array}{lcl} cata\ (\oplus)\ e\ [\,] & = & e \\ cata\ (\oplus)\ e\ (x : xs) & = & x \oplus cata\ (\oplus)\ e\ xs \end{array}$$

具体的に，e, $\oplus$ を与えることにより，さまざまな関数を表すことができる．例えば，e を 0，$\oplus$ を $+$ とすると，リストの和を求める関数になる．関数 $cata\ (\oplus)\ e$ は，リストを入力として受けとり，その入力リスト中の [ ] を e で，: を $\oplus$ に置き換えて評価したものを結果として返す．関数 $cata$ は，e, $\oplus$ によって唯一に定まるので，$cata = ([\lambda().e, (\oplus)])$ と記述する．

定義 1 (catamorphism)
再帰データ $D\ \alpha$ 上の関数 f が $i = 1, \ldots, k$ に対して

$$f\ (C_i\ (e,\ x_1, \ldots, x_{n_i})) = \phi_i\ (e, f\ x_1, \ldots, f\ x_{n_i})$$

と定義されるとき，f を catamorphism という． □

この f を $([\phi_1, \ldots, \phi_k])_{D\,\alpha}$ と表す．添字の $D\ \alpha$ は，文脈から明らかな場合には省略することもある．catamorphism は，プログラム変換において最も重要な概念の 1 つである[13) 14)]．

2.2 最大マーク付け問題の仕様の記述

最大マーク付け問題を解くには，すべてのマーク付けの中から性質 p を満たすものを $filter\,p$ で取り出し，その中で重み関数 w の値が最大のものを $\uparrow_w /$ を用いて 1 つ取り出すことによって得られる．

$$mmp\ p\ w\ =\ \uparrow_w / \circ\ filter\ p \circ gen \qquad (1)$$

性質を p，重み関数を w，入力データを x とするとき，$mmp\ p\ w\ x$ が 1 つの解を与える．関数 gen は，入力データを引数にとり，すべてのマーク付きデータからなるリストを返す．

$$\begin{array}{lcl} gen\ [\,] & = & [[\,]] \\ gen\ (x : xs) & = & [x^* : xs^* \mid \\ & & \quad x^* \leftarrow [mark\ x, unmark\ x], \\ & & \quad xs^* \leftarrow gen\ xs] \end{array}$$

ここで，関数 $mark$, $unmark$ は，要素にマークを付ける関数，マークを付けない関数を表す．

$$\begin{array}{lcl} mark\ x & = & (x, True) \\ unmark\ x & = & (x, False) \end{array}$$

gen は catamorphism では，

$$\begin{array}{l} ([\lambda().[[\,]], \\ \quad \lambda x\ xs.[x^* : xs^* \mid x^* \leftarrow [mark\ x, unmark\ x], \\ \qquad\qquad xs^* \leftarrow xs]) \end{array}$$

と表せる．また，$\uparrow_w$ は次のように定義される．

$$\begin{array}{lcll} a \uparrow_w b & = & a, & \textbf{if}\ w\ a > w\ b \\ & = & b, & \textbf{otherwise} \end{array}$$

また，演算子 / は

$$\oplus\ /\ [a_1, a_2, \ldots, a_n] = a_1 \oplus a_2 \oplus \cdots \oplus a_n$$

と定義される．

最大マーク付け問題の一例として最大部分列和問題について考えることとする．最大部分列和問題を (1) 式の形で表すためには，性質 p と重み関数 w を決め

ればよい．まず p は，マークのついた要素が連続しているかどうかを判定する以下の関数 $conn_0$ である．

$$
\begin{array}{lcl}
conn_0\ [\,] & = & True \\
conn_0\ (x:xs) & = & \textbf{if}\ marked\ x\ \textbf{then}\ conn_1\ xs \\
 & & \textbf{else}\ \ conn_0\ xs \\
conn_1\ [\,] & = & True \\
conn_1\ (x:xs) & = & \textbf{if}\ marked\ x\ \textbf{then}\ conn_1\ xs \\
 & & \textbf{else}\ \ conn_2\ xs \\
conn_2\ [\,] & = & True \\
conn_2\ (x:xs) & = & \textbf{if}\ marked\ x\ \textbf{then}\ False \\
 & & \textbf{else}\ \ conn_2\ xs \\
marked\ (_,m) & = & \textbf{if}\ m\ \textbf{then}\ True\ \textbf{else}\ False
\end{array}
$$

重み関数 w はマークの付いた要素の和を求める以下の関数 $wsum$ とすればよい．

$$
\begin{array}{lcl}
wsum & = & +/ \circ map\ f \\
 & & \textbf{where} \\
 & & \quad f\ x = \textbf{if}\ marked\ x \\
 & & \qquad\qquad \textbf{then}\ weight\ x \\
 & & \qquad\qquad \textbf{else}\ 0 \\
 & & \quad weight\ (x,m) = x
\end{array}
$$

関数 $conn_0$，$wsum$ を用いると，最大部分列和問題は次のように (1) 式の形で表すことができる．

$$mss\ =\ \uparrow wsum\ /\ \circ\ filter\ conn_0 \circ gen$$

この仕様は単純であり理解しやすい仕様といえるが，計算量が指数オーダーとなるため効率が悪い．以下でこのプログラムを線形プログラムに変換する変換規則を示す．

2.3 変換規則

プログラム変換において，ある関数とある catamorphism を融合して，新たに catamorphism を作すことによって，中間データを生成しないようにすることはよく行われる．そのとき用いられるのが次の融合規則である．

定理 1 (融合規則) 任意の a, x について

$$a \otimes f\ x = f\ (a \oplus x)$$

ならば，

$$f \circ ([\lambda().e, (\oplus)]) = ([\lambda().f\ e, (\otimes)])$$

である． □

また，次の最大マーク付け問題の最適化規則も変換規則の一例である．これは文献 5) の最適化定理に対して，文献 9) の重み関数に関する拡張を行ったものである．

定理 2 (最適化規則)

最大マーク付け問題

$$mmp\ p\ w\ =\ \uparrow w\ /\ \circ\ filter\ p\ \circ\ gen$$

は，

(1) 性質 p が有限の値域をもつ関数 $\phi_i (i = 1, \ldots, n)$ と述語 $accept$ を用いて

$$p = accept \circ ([\phi_1, \ldots, \phi_k])$$

と分解され，

(2) 重み関数 w が

$$w = \oplus/\ \circ\ map\ f$$

の形に分解でき，$\uparrow$ が $\oplus$ に対して分配法則

$$x \uparrow (y \oplus z) = (x \uparrow y) \oplus (x \uparrow z)$$

を満たし，$\oplus$ が結合的であるという条件を満たすとき，最適化関数 opt (図 1 参照) を用いて

$$mmp\ p\ w\ =\ opt\ (\oplus, f)\ accept\ \phi_1 \ldots \phi_k$$

と書くことができる．この関数 opt によって，最大マーク付け問題は線形時間で解ける． □

図 1 のプログラムの概要は以下のとおりである．関数 $mark$ は要素に $True$ とマーク付けをし，関数 $unmark$ は要素に $False$ とマーク付けをする．関数 $marked$ は要素を受け取りその要素のマークを返す．関数 $weight$ はマーク付けのされた要素のマーク付けされる前の値を返す．関数 $getdata$ はクラス，重み，再帰的データの 3 つ組を受け取り再帰的データを返す．関数 $eachmax$ はクラス，重み，再帰的データの 3 つ組のリストを受け取りそれぞれのクラスで重みが最大の 3 つ組のリストを返す．関数 opt 中の関数 $leftunit$ は 2 項演算子を受け取り，その 2 項演算子の左単位元を返す関数である．最適化関数 opt の詳しい例は[5] にある．

上で記述した最大部分列和問題の仕様は，最適化規則を適用することにより次のような線形時間アルゴリズムに変換される．

$$
\begin{array}{l}
mss\ =\ opt\ (+, f)\ accept\ \phi_1\ \phi_2 \\
\quad \textbf{where} \\
\qquad f\ x\ =\ \textbf{if}\ marked\ x\ \textbf{then}\ weight\ x \\
\qquad\qquad\qquad\qquad \textbf{else}\ \ 0 \\
\qquad accept\ (t_0, t_1, t_2)\ =\ t_0 \\
\qquad \phi_1\ () = (True, True, True) \\
\qquad \phi_2\ (x, (t_0, t_1, t_2)) = \textbf{if}\ marked\ x \\
\qquad\qquad\qquad\qquad \textbf{then}\ (t_1, t_1, False) \\
\qquad\qquad\qquad\qquad \textbf{else}\ \ (t_0, t_2, t_2)
\end{array}
$$

これらの変換は，システムを用いて自動的に行うことができる．

3. 自動生成システムの構成

システムの概要を図 2 に示してある．変換戦略の仕

```
opt (⊕, f) accept φ1 ... φk x =
    getdata (↑snd / [(c, w, r*) | (c, w, r*) ← ([ψ1, ..., ψk])D x, accept c])
  where ψi () = [(φi, leftunit (⊕), Ci)]
        ψi (e) = [(φi (e*), f e*, Ci e*) | e* ← [mark e, unmark e]]
        ψi (e, cand1, ..., candni) =
            eachmax [(φi (e*, c1, ..., cni),
              f e* ⊕ w1 ⊕ ... ⊕ wni,
              Ci (e*, r1*, ..., rni*)) |
                    e* ← [mark e, unmark e],
                    (c1, w1, r1*) ← cand1, ···, (cni, wni, rni*) ← candni]      (i = 1, ..., k)

mark x = (x, True)
unmark x = (x, False)
marked (_, m) = if m then True else False

weight (x, _) = x
getdata (_, _, x) = x

eachmax xs = foldl f [] xs
    where
        f [] (c, w, r) = [(c, w, r)]
        f ((c, w, r) : opts) (c', w', r') =
            if c == c' then
                if w > w' then (c, w, r) : opts
                else opts ++ [(c', w', r')]
            else (c, w, r) : f opts (c', w', r')
```

図 1 最適化関数 opt

様記述のための高レベルの記述の枠組である運算付随プログラム CCP[3] によって，プログラムの仕様とプログラムの変換規則の記述を行う．CCP は高階マッチング機能をもっているので直接に広範囲の変換規則の記述が可能である．CCP 処理系で文法・型チェックを行い解釈実行すると，変換結果のプログラムと共に変換の過程を記述した変換スクリプトが得られる．

変換スクリプトは，新たに変換規則，変換戦略を開発するときに用いると有効であり，変換の 1 ステップごとに対話的な実行，変換の取消し，現在の変換結果，変数の束縛，現在使用可能な変換規則，関数の型などの確認の作業を支援する．

CCP と変換スクリプトは独立しており，他のシステムでの変換手順を変換スクリプトとして記録すれば，変換スクリプトの処理系で実行することが可能である．

3.1 運算付随プログラム CCP

CCP とは，変換規則 (運算規則) が付随したプログラムである[15]．プログラム変換において，用いられる変換規則や，変換戦略 (変換規則の適用の仕方) は変換対象のプログラムに応じて変化する．変換を機械的に行うときには，この変換法を何らかの方法で記述しなければならないが，その 1 つの方法を与えるのが CCP である．CCP は，用いられる変換規則および変換戦略を明示的にプログラムに付随させたものであり，これにより，プログラマが変換法を指定することが可能となる．また，融合変換，組化等，汎用的な変換については，一度記述すれば再利用が可能である．

3.1.1 例 — sumsq

ここでは，リストの 2 乗和を求める問題 (sumsq) という問題を通して，CCP の具体例を示す．この問題を解くには，各要素を 2 乗して足し合わせればよい．2 乗する関数を sq，数列の和を求める関数を sum とすると，

$$sumsq = sum \circ map\ sq$$

```
-------------- プログラム -----------------
sumsq = sum . map sq;
sum []     = 0;
sum (x:xs) = x + sum xs;
sq x = x * x;
map f []     = [];
map f (x:xs) = f x : map f xs;
cata oplus e []    = e;
cata oplus e (a:x) = oplus a (cata oplus e x);
(f . g) x = f (g x);

-------------- 変換規則 -----------------
<fusion> (_f (cata _oplus _e _xs)) =
    letm _g a (_f x) = _f (_oplus a x);
         _c = _f _e
    in cata _g _c _xs;

<applyFusion> (_f . _g) = <fusion> (_f . <applyFusion> _g);
<applyFusion> _f = <fusion> (_f . cata (:) []);

-------------- 変換規則の適用 -----------------
sumsqOpt = <applyFusion> (<unfold> sumsq)
```

図 3 Calculation carrying program の例 — sumsq

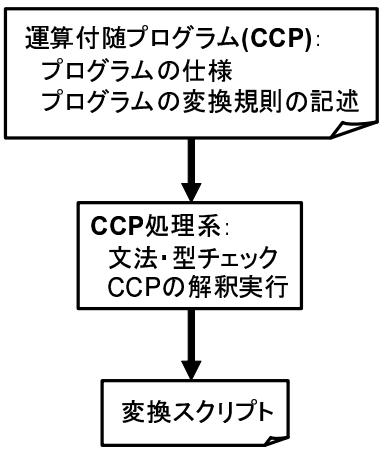


図 2 システムの構成

が 1 つの解法を与える．これは，各要素が 2 乗されたリストを $map\ sq$ により生成してしまうが，実際には，このようなリストを生成しないように記述することができる．関数 $sumsq$ をそのような形に変換するには，融合規則 (定理 1) を用いればよい．

融合規則を適用するために，まず，$map\ sq$ を catamorphism の形で記述する．

$$map\ sq = ([\lambda().[\,], (\oplus)])$$
$$\textbf{where}\ y \oplus ys = sq\ y : ys$$

次に，

$$y \otimes sum\ ys = sq\ y + sum\ ys$$

を満たす関数 $(\otimes)$ を求める．次の関数 $(\otimes)$ は，この式を満たす．

$$y \otimes s = sq\ y + s$$

$sum\ [\,] = 0$ であるので融合定理を適用することにより，

$$sum \circ map\ sq = ([\lambda().0, (\otimes)])$$

を得る．catamorphism $([\lambda().0, (\otimes)])$ は，2 乗和を計算する過程においてリストを生成しないので，効率が改善されている．

$sumsq$ の CCP は，図 3 のように簡潔に書くことができる．CCP は 3 つの部分に分けることができ，図 3 においては，上段部分が変換対象のプログラム，中段部分が変換規則 (<fusion>, <applyFusion>)，下段部分が，変換対象のプログラム (sumsq) から効率のよい関数 (sumsqOpt) を得る過程を示している．上段部

分では，$sumsq$ の仕様を記述している．融合変換の変換過程を記述したのが中段部分である．<fusion>が融合規則 (定理 1) を記述している．融合規則を適用する際には，関数合成 f . g (関数合成の演算子 ∘ は CCP と変換スクリプトでは . と表記される) の右側の関数 g が cata oplus e の形で記述されていなければならないので，これを行うための変換規則として<applyFusion>を定義している．<applyFusion>では，関数融合の最も右側の関数を cata (:) [] と融合規則<fusion>を適用することによって cata oplus e の形に変換し，その後，順に融合規則<fusion>を適用していく．この<applyFusion>規則を sum . map sq に適用すれば，中間リストを生成しない形が得られる．それを記述しているのが下段部分である．ここでは，まず sumsq の定義を<unfold>によって展開し，それに対して<applyFusion>を適用する．これによって，中間リストを生成しない関数 sumsqOpt が得られる．<unfold>は，関数定義を展開する組み込みの変換規則である．

3.1.2 CCP の記述言語

本論文においては，CCP の記述言語を図 4 のように定義している．CCP では，関数定義と規則定義を混在させて記述することができる．また，式中で，規則適用，メタ let 式，メタ case 式を用いることができ，そこでは高階パターンマッチング[16)17)] が行われる．パターンマッチングにおいては，パターン変数に対する置換を求める．

本システムで使用しているパターンマッチングはシンプルマッチング[16)17)] と 2 ステップマッチング[16)17)] である．例えば $p \circ q$ を sum にマッチさせたときに 2 ステップマッチングでは置換 $\{p \to \lambda x.x, q \to sum\}, \{p \to sum\}$ を返すが，シンプルマッチングではマッチングが失敗する．

高階パターンマッチングを含む式の意味は以下のように定める．

- メタ let 式

  $$\textbf{letm } e_{p_1} = e_{b_1}; \ldots; e_{p_n} = e_{b_n} \textbf{ in } e$$

  各式 $e_{p_1}, \ldots, e_{p_n}$ 中のパターン変数に対する置換を 2 ステップマッチングによって求め，それを用いて式 e を評価し，得られた値を全体のメタ let 式の値とする．
- メタ case 式

  $$\textbf{casem } e \textbf{ of } e_{p_1} \to e_1; \ldots; e_{p_n} \to e_n$$

  式 e について，式 $e_{p_1}, \ldots, e_{p_n}$ と順にシンプルマッチングを行い，最初にマッチングが成功した式 e_{p_i} の，パターン変数への置換によって e_i を評価し，得られた値を全体のメタ case 式の値とする．
- 規則適用 $< r >\ e$

  変換規則 $< r >$ を式 e に対して適用する．変換規則 $< r >$ が

  $$< r >\ e_p = e_b$$

  と定義されているとき，e と e_p をシンプルマッチングを行い，得られた置換によって e_b を評価し，得られた値を全体の規則適用式の値とする．

なお，上記 CCP の定義中で $haskellExp$ は関数型言語 Haskell[12)18)] の式であるが，現在のシステムではその一部のみを実現している．なお，本システムでは，パターン変数の変数名は，アンダースコア _ で始め，それ以外の変数の変数名は，アンダースコア _ では始めないこととする．この区別は，定数とパターン変数とを区別するためと，letm 式中で，局所変数とパターン変数を区別するためのものである．例えば，letm 式のマッチングの

```
_g a (_f x) = _f (_oplus a x)
```

において a と x にはアンダースコア_ が a と x についていないが，これは，a と x がパターン変数ではなく，局所変数であり，等式全体において a と x が全称的に束縛されていることを表している．変換規則は変換規則名を<>で囲むことによって関数と区別している．また，CCP 中では，規則の定義中で他の変換規則や自分自身を呼び出すことができる．このような記述を許すことにより，<applyFusion>規則のように，基本的な変換を組み合わせて複雑な変換を記述することができるようになる．

3.1.3 CCP のセマンティクス

CCP のセマンティクスを図 5 の評価関数 $\mathcal{E}$ によって定める．CCP を評価した値は **Exp** 型である．**Exp** はプログラム変換の対象言語 Haskell の式を表す．パターン変数の束縛を保持するために，置換の環境 **Env** を用い，$\mathcal{E}$ はメタ式を環境下で評価する．$\sqcup$ はリストのリストを平坦化することを表している．$unlet$ は let 式を λ 式に置き換え，$\beta norm$ は出現するすべての β-redex に対して β-簡約を行う．$\rho_1 \oplus \rho_2$ は環境 ρ_1 を ρ_2 で以下のように拡張することを表している．

$$\begin{aligned}(\rho_1 \oplus \rho_2)\ x &= \rho_2\ x, \quad x \text{ is defined in } \rho_2 \\ &= \rho_1\ x, \quad \text{otherwise.}\end{aligned}$$

シンプルマッチング[17)] $match\text{-}s$ と 2 ステップマッチング[17)] $match\text{-}2$ はパターンマッチングをされる側の項の中に自由変数が現れていないときにパターンマッチングを行う．式が変数 v のときはそのままを返す．式が

CCP:

ccp ::= $def_1; \ldots; def_n$ 定義列

定義:

def ::= $funDef$ 関数定義
| $ruleDef$ 規則定義

関数定義:

$funDef$ ::= $f\ pats = e$ 関数定義

規則定義:

$ruleDef$ ::= $\langle r \rangle\ e_p = e_b$ 規則定義

式:

e ::= $haskellExp$ 関数型言語 Haskell の式
| em 高階パターンマッチングを含む式

高階パターンマッチングを含む式:

em ::= $\mathbf{letm}\ e_{p_1} = e_{b_1}; \ldots; e_{p_n} = e_{b_n}\ \mathbf{in}\ e$ メタ let 式
| $\mathbf{casem}\ e\ \mathbf{of}\ e_{p_1} \to e_1; \ldots; e_{p_n} \to e_n$ メタ case 式
| $\langle r \rangle\ e$ 規則適用

図 4 CCP の記述言語の定義

$$
\begin{array}{lcl}
\mathcal{E}[\![e]\!] & : & \mathbf{Env} \to [\mathbf{Exp}] \\
\mathcal{E}[\![v]\!]\rho & = & [v] \\
\mathcal{E}[\![_v]\!]\rho & = & [\rho\ v] \\
\mathcal{E}[\![n]\!]\rho & = & [n] \\
\mathcal{E}[\![\lambda v.e]\!]\rho & = & [\lambda v.e' \mid e' \leftarrow (\mathcal{E}[\![e]\!]\ \rho)] \\
\mathcal{E}[\![e_1\ e_2]\!]\rho & = & [e'_1\ e'_2 \mid e'_1 \leftarrow \mathcal{E}[\![e_1]\!]\rho,\ e'_2 \leftarrow \mathcal{E}[\![e_2]\!]\rho] \\
\mathcal{E}[\![< rule >\ e]\!]\rho & = & \sqcup\ [\mathcal{E}[\![e_b]\!]\ (\rho \oplus \phi) \mid \phi \leftarrow match\text{-}s\ \rho\ e_p\ (\beta norm\ e)] \\
\mathcal{E}[\![\mathbf{let}\ x_1 = e_1; \ldots; x_n = e_n\ \mathbf{in}\ e]\!]\rho & = & \mathcal{E}[\![unlet\ (\mathbf{let}\ x_1 = e_1; \ldots; x_n = e_n\ \mathbf{in}\ e)]\!]\rho \\
\mathcal{E}[\![\mathbf{letm}\ e_p = e_b\ \mathbf{in}\ e]\!]\rho & = & \sqcup\ [\mathcal{E}[\![e]\!]\ (\rho \oplus \phi) \mid e'_b \leftarrow \mathcal{E}[\![e_b]\!]\rho, \phi \leftarrow match\text{-}2\ \rho\ e_p\ (\beta norm\ e'_b)] \\
\mathcal{E}[\![\mathbf{casem}\ e\ \mathbf{of}\ e_{p_1} \to e_1; \ldots; e_{p_n} \to e_n]\!]\rho & = & \sqcup\ [[\mathcal{E}[\![e_1]\!]\ (\rho \oplus \phi_1) \mid e'_b \leftarrow \mathcal{E}[\![e]\!]\rho, \phi_1 \leftarrow match\text{-}s\ \rho\ e_{p_1}\ (\beta norm\ e'_b)] \\
 & & \quad +\!\!+ \cdots +\!\!+ \\
 & & \quad [\mathcal{E}[\![e_n]\!]\ (\rho \oplus \phi_n) \mid e' \leftarrow \mathcal{E}[\![e]\!]\rho, \phi_n \leftarrow match\text{-}s\ \rho\ e_{p_n}\ (\beta norm\ e')]]
\end{array}
$$

図 5 CCP の評価関数

パタン変数 $_v$ のときは環境のもとで評価する．定数 n のときはそのままである．λ 式のときは本体の中を評価した結果を返す．関数適用は関数，引数のそれぞれを評価した結果を返す．変換規則 $< rule >\ e_p = e_b$ の式 e への適用において，まず e_p と e をシンプルマッチングすることで置換を得てそれを環境に追加し，次に e_b をその環境下で評価する．let 式はすべて λ 抽象で置き換える．letm 式は 2 ステップマッチングによって置換を得る．casem 式はそれぞれのパターンと e をシンプルマッチングして得られた置換をそれぞれ環境に加える．

3.2 変換スクリプト

変換スクリプトは図 6 に示す文法に従う．変換スクリプトは，変換のステップを細かく記述したものである．変換スクリプトを用いるとデバッグを行うための対話的実行が行える．変換スクリプトで用いる命令は環境設定命令，実行命令，デバッグ用命令の 3 種類に大きく分かれる．

変換スクリプト

Command	::=	*EnvComm*	環境設定命令
	|	*IntComm*	実行命令
	|	*DebugComm*	デバッグ用命令

環境設定命令

EnvComm	::=	setPath *PathName*	プログラムと規則定義と関数定義の書かれたファイルのあるディレクトリへのパス指定
	|	loadTheory *TheoName*	規則定義と関数定義の書かれたファイルの読込み
	|	setExp *ExpName*	始式のセット

実行命令

IntComm	::=	step *LawName Path*	変換規則適用
	|	beginDerivation *LawName Path*	部分導出開始
	|	match	高階マッチングを行い置換を求める
	|	endDerivation	部分導出終了
	|	createRule *RuleName*	新規変換規則作成
Path	::=	$Loc_1\ Loc_2\ \cdots$	部分式へのパス
Loc	::=	B	λ 式の本体
	|	F	関数適用の関数部分
	|	A	関数適用の引数部分

デバッグ用命令

DebugComm	::=	help	ヘルプ表示
	|	showDerivation	今までの導出の出力
	|	quit	終了
	|	showBindings	局所変数の表示
	|	showStack	すべての部分導出表示
	|	undo	取り消し
	|	showSubExpressions	すべての部分式を表示
	|	showTheory	すべての変換規則を表示

図 6 変換スクリプトの文法

変換スクリプトの例として，リストを受け取りリストの要素の 2 乗の和を返す関数命令 *sumsq* の変換を考えることとする．用いる変換規則の書かれたファイルは図 7 のようである．“{-”，“-}” で囲まれた部分はコメントである．変換規則は “;” で区切られている．それぞれの変換規則は “:” の左側が変換規則の名前で “=” の左側から右側への変換を表わしている．`if` 文は変換規則の適用に条件が必要なときに記述する．条件式は “;” で区切られ，“=” の左辺が項を表わし，“=” の右辺がパターン変数を含む式であるパターンを表す．この規則に基づく，変換スクリプトは図 8 のようになる．図 8 の中の変換スクリプトの左側の数字と”:” は説明のために書き加えたものである．図 8 の 1 行目の “`#!`” はこのファイルが変換スクリプトであることを表わしている．

3.2.1 環境設定命令

環境設定命令は setPath，loadTheory，setExp の 3 種類である．命令 setPath はプログラムと変換規則の書かれたファイルの置いてあるディレクトリへのパスを指定する．命令 loadTheory は変換規則の書かれた

```
{- sumsq.eq -}

sumsq : sumsq = sum . map sq;
sum1  : sum []       = 0;
sum2  : sum (_x:_xs) = _x + sum _xs;
sq    : sq _x = _x * _x;
map1  : map _f [] = [];
map2  : map _f (_x:_xs) = _f _x : map _f _xs;
cata1 : cata _step _e []      = _e;
cata2 : cata _step _e (_a:_x) = step _a (cata _step _e _x);

fusion : _f (cata _step _e _x) = cata _g _c _x,
         if{ \ x a -> _f (_step a x) = \ x a -> _g a (_f x);
             _f _e = _c };
applyFusion1: _f . _g = _f . _g;
applyFusion2: _f      = _f . cata (:) [];

sumsqOpt: sumsqOpt = sumsq
```

図 7 変換規則が書いてあるファイル − sumsq.eq

ファイルを読み込む．命令 setExp は式をプログラム変換を始める式に指定する．例えば，図 8 では 2 行目の命令 setPath で変換規則の書かれたファイルのあるディレクトリへのパスを指定し，3 行目の命令 loadTheory で変換規則の書かれたファイル sumsq.eq を読み込んでいる．図 8 の 4 行目の命令 setExp は式 sumsqOpt (図 3 参照) をプログラム変換を始める式を指定している．

3.2.2 実 行 命 令

実行命令は step，beginDerivation，match，endDerivation，createRule の 5 種類である．命令 step は適用する変換規則名と適用する部分式のパスをうけとって変換して得られた結果を次の変換対象とすることを表す．B は λ 式の本体，F は関数適用の関数部分，A は関数適用の引数部分を表す．図 8 の 8 行目の命令

```
step "applyFusion2" "BFA"
```

を例にとって処理内容を述べる．$\lambda x.M\ x$ のような形の λ 項を，関数表記である M に書き換えることを η-簡約という．η-簡約の逆操作を η-展開という．変換スクリプトの処理系の内部では式はすべて η-展開されており，表示のときはすべて η-簡約されている．例えば，式 sum . map sq は処理系の内部では

```
\ a -> ((\ b -> sum b) .
        (\ c -> map (\ d -> sq d) c)) a
```

と表現されている．この η-展開に用いる型の情報は図 7 中に定義された変換規則の中の関数の型推論から得たものである．式:

```
\ a -> ((\ b -> sum b) .
        (\ c -> map (\ d -> sq d) c)) a
```

のパス BFA の部分式は

```
\ c -> map (\ d -> sq d) c
```

である．図 7 の下から 2 行目の applyFusion2 を適用すると

```
(\ c -> map (\ d -> sq d) c) .
        (cata (:) [])
```

を得る．具体的には，変換規則 applyFusion2 の左辺

```
_f
```

と部分式

```
\ c -> map (\ d -> sq d) c
```

のマッチングにより処理系の内部では置換

```
{_f := \ c -> map (\ d -> sq d) c}
```

が得られ環境に追加され，これを変換規則 applyFusion2 の右辺に代入することで上の式を得ている．これによって，変換結果は

```
\ a -> ((\ b -> sum b) .
        ((\ c -> map (\ d -> sq d) c) .
         (cata (:) [])))
```

```
 1:  #!

 2:  setPath "/home/yokoyama/ys/ex"
 3:  loadTheory "sumsq"
 4:  setExp "sumsqOpt"

 5:  step "sumsqOpt" "BF"
 6:  step "sumsq" "BF"
 7:  step "applyFusion1" "BF"
 8:  step "applyFusion2" "BFA"

 9:  beginDerivation "fusion" "BFABF"
10:      step "map2" "BB"
11:      match
12:      step "map1" ""
13:      match
14:  endDerivation

15:  beginDerivation "fusion" "BF"
16:     step "sum2" "BB"
17:     match
18:     step "sum1" ""
19:     match
20:  endDerivation

21:  showDerivation
22:  quit
```

図 8 変換スクリプトの例 – sumsq.ys

であり，表示されるのはこれを η-簡約した

```
sum . (map sq . cata (:) [])
```

という式となる．

命令 beginDerivation は適用する変換規則名と適用する部分式へのパスを受け取って，if{...}で書かれた部分の式の部分導出を始める．このとき指定する変換規則は，必ず図 7 の変換規則 fusion のように if に続く条件式をもたねばならない．左辺では出現していないパターン変数を束縛する．これによって規則の部分導出で，この if の中の式の左辺から右辺へと変換し，置換を得る．それらの置換を使って変換規則 fusion を適用する．例えば，図 8 の 9 行目では，変換規則 fusion を式 sum . (map sq . cata (:) []) の η-展開した式:

```
\ a ->
  ((\ b -> sum b) .
   (\ c -> ((\ d -> map (\ e -> sq e) d) .
           (\ f -> cata (\ g h -> g : h )
                        [] f))
            c)) a
```

のパス BFABF の部分式:

```
(\ d -> map (\ e -> sq e) d) .
        (\ f -> cata (\ g h -> g : h ) [] f)
```

と変換規則 fusion の左辺との高階マッチングにより，処理系の内部では置換:

```
{ _f := \ d -> map (\ e -> sq e) d,
  _step := \ g h -> g : h,
  _e := []}
```

が環境に追加され，if{...}で書かれた部分の式の部分導出を行うことによって，変換規則 fusion の左辺には現れないパターン変数_g，_c を束縛する．具体的には，if 文の 2 行目

```
_f _e = _c
```

の場合には左辺に先ほどの環境に追加された置換を代入することで

```
(\ d -> map (\ e -> sq e) d) []
```

が得られ，これに図 8 の 12 行目で変換規則 map1 を適用して

```
[]
```

が得られる．

命令 match は部分導出中で条件式のパターンと現在の式とでマッチングを行い得られた置換を記憶する．例えば，図 8 の 13 行目で match を行うことで処理系の内部では置換

```
{_c := []}
```

が得られる．命令 endDerivation は部分導出によって得られた置換を用いて変換規則の右辺を適当に置きかえ，命令 beginDerivation で指定した変換規則を部分式に適用して得られた結果を式に与えて部分導出から脱出する．例えば，図 8 の 14 行目では，11 行目と 13 行目の match によって処理系の内部では置換

```
{_g := \ g -> (:) (sq g), _c := []}
```

が環境に追加されているので，これを変換規則 fusion の右辺に代入することで

```
cata (\ g -> (:) (sq g)) []
```

の η-展開した式が得られる．

命令 createRule は始式から現在の式へと変換する変換規則を指定した名前を追加する．

3.2.3 デバッグ用命令

デバッグ用命令は対話的に実行するときに用いる．命令 help は現在実行できる命令を表示する．命令 showDerivation はこれまでの導出を表示する．例え

ば，図 8 の 21 行目の命令 showDerivation は導出結果を出力 (図 9) をする．命令 quit はそれ以降の命令を実行せずに終了する．命令 showBindings は現在の式における局所変数を表示する．命令 showStack は部分導出に入るまでの情報や部分導出で得られた置換など記憶しているものを表示する．命令 undo によって 1 ステップ元に戻る．命令 showSubExpressions は現在の式の部分式をすべて表示する．命令 showTheory は現在使用可能な変換規則をすべて表示する．

上記の変換スクリプトを実行して得られる出力は図 9 の通りである．“{”，“}” で囲まれた部分は変換規則名と変換に必要な置換を得るために必要な部分導出が記述されている．

3.3 CCP から変換スクリプトへの変換

CCP の実行の際には，書き換え規則の右辺から他の規則を呼び出さない書き換え規則を求めるという前処理が行われる．この前処理の結果，右辺の規則の適用，メタ let 式，メタ case 式が現れない変換規則が求められる．関数定義は，定義を展開する書き換え規則に変換する．変換規則定義は，右辺の中の規則を除いた形の規則に変換する．これらの変換は，高階パターンマッチングを含む式を以下のように変換することにより行う．

- メタ let 式

  $$\mathbf{letm}\ e_{p_1} = e_{b_1}; \ldots; e_{p_n} = e_{b_n}\ \mathbf{in}\ e$$

  以下のような条件付式に変換する．

  $$e,\ \mathit{if}\ \{e_{b_1} = e_{p_1}; \ldots; e_{b_n} = e_{p_n}\}$$

  ここでマッチングの等式の左辺と右辺は入れ換えられているのは，書き換え規則においては，パターンを右辺に書くためである．
- メタ case 式

  $$\mathbf{casem}\ e\ \mathbf{of}\ e_{p_1} \to e_1; \ldots; e_{p_n} \to e_n$$

  以下の n 個の書き換え規則を生成し，メタ case 式は以下のような変換規則群に変換する．

  $$e_{p_1} = e_1; \ldots; e_{p_n} = e_n$$

  メタ case 式は e によって適用する変換規則が異なるが，変換スクリプトの処理系では命令によって陽に指定することでこの制御を行う．

CCP の実行は，この前処理によって得られた書き換え規則を，CCP の変換規則に指定された順に適用していくことによる．その過程において適用された規則，適用箇所を記録することにより，変換スクリプトが得られる．

ここでは，例として sumsq 問題の CCP から変換スクリプトへの変換の例を示す．図 3 に sumsq 問題の CCP を記述したが，これを上記の手順に従って実行し，変換スクリプトを得る．まず，前処理として，各関数定義は，定義を展開する変換規則に変換され，各変換規則は，規則の右辺中の規則名を機械的に取り除くことによって，他の規則を呼び出さない書き換え規則に変換される．これにより，図 7 のような書き換え規則が得られる．

```
    sumsqOpt
  = { sumsqOpt }
    sumsq
  = { sumsq }
    sum . map sq
  = { applyFusion1 }
    sum . map sq
  = { applyFusion2 }
    sum . (map sq . cata (:) [])
  = { fusion

        (\ a x -> map sq (a : x ))
    = { map2 }
        (\ b d -> sq b : map sq d)

        map sq []
    = { map1 }
        []
    }
    sum . cata (\ g -> (:) (sq g)) []
  = { fusion

        (\ a x -> sum (sq a : x))
    = { sum2 }
        (\ b c -> sq b + sum c)

        sum []
    = { sum1 }
        0
    }
    cata (\ f -> (+) (sq f)) 0
```

図 9 実行結果 – sumsq.out

applyFusion1 は，<fusion> 規則, <applyFusion> 規則を適用する以外のことは行わないため，何も書き換えない規則が得られている．

次に，得られた図 7 の書き換え規則を，図 3 の CCP において指定された順に適用することにより，変換が

行われる．変換の過程は，図 9 のようになる．まず，sumsqOpt が sumsq に変換され，それが<unfold>によって展開される．これに対して<applyFusion>が適用されるが，そのとき，sum ． map sq が_f ． _g とマッチングすることを調べ，applyFusion1 を適用する．ここでは何も変換が行われていないが，これは規則の適用以外何も行われないためである．このとき，システム内部では，<applyFusion> を map sq に対して適用し，その後全体に対して<fusion>を適用するということが記憶される．次は<applyFusion> が map sq に対して適用されるが，map sq が_f ． _g の形に適合しないため，applyFusion2 が適用される．これにより，sum ． (map sq ． cata (:) []) が得られるが，このとき，次には (map sq ． cata (:) []) に対して<fusion>規則を適用することが記憶される．その後，<fusion>規則が 2 回適用され，その結果，中間リストを生成しないプログラムが得られる．

この書き換えの過程において適用した規則，規則の適用箇所を変換スクリプトの形で記録すると，図 8 のような変換スクリプトが得られる．これにより，CCP から変換スクリプトへの変換が行われたことになる．

なお，変換スクリプトの実行における変換規則の書いてあるファイルの形式は MAG システムとまったく同じものである．これは筆者らが開発した変換スクリプトの処理系が MAG システムを対話的実行の点で拡張を行ったものであるからである．

4. 最大マーク付け問題の自動生成の例

本節では，いくつかの最大マーク付け問題の線形時間プログラムの自動生成を例示して，本システムの有効性を示す．

4.1 最大部分列和問題

最大部分列和問題は第 2 節で述べたように，最大マーク付け問題として次のように定式化できる．

$$mss \;=\; \uparrow wsum\ /\ \circ\ filter\ conn_0\ \circ\ gen \quad (2)$$

まず gen によって 2^n 個のすべてのマーク付きリストを生成する．次に $filter\ conn_0$ によって性質 $conn_0$ を満たすマーク付きリストのみを取り出す．最後に $(\uparrow wsum\ /)$ によって重み関数 $wsum$ の値が最大となるマーク付きリストを 1 つ返す．

効率の悪いこの定義から線形時間アルゴリズムに変換する CCP を図 10 のように記述する． 最大部分列和問題の仕様である (2) 式の定義はプログラムの 1 行目に書かれている．なお，$bmax\ wsum$ は $(\uparrow wsum)$ を，$reduce$ は $(/)$ をそれぞれ表す．

mmpRule は定理 2 の最適化規則を表わしており，最適化関数 opt に変換する．最適化関数の前提条件を満たすと<tupling>より，性質 $conn_0$ が分解できる．この CCP を実行することで実行過程の記録として図 8 と同じような変換スクリプトを得ることができる．

4.2 最大部分列和問題を拡張した問題

最大部分列和問題の解の満たすべき条件を厳しくした問題へ拡張した場合の処理を考えることとする．第 2 節で述べた最大部分列和問題の中の性質 $conn_0$ と，解のリストに付いたマークの数が偶数であるという性質 $even$ を同時に満たすという性質 p は，

$$p\ xs \;=\; conn_0\ xs \wedge even\ xs$$

と表わせる．

この問題も最大マーク付け問題の例になっている．図 10 の CCP の変更するべき部分が図 11 に記述されている．このように変換規則を変更しなくても別のプログラム変換を行うことが可能であることがわかる．

5. 結　　論

本論文では，変換戦略と変換規則を同時に記述する理論的枠組である CCP の実現法を示した．また，最大マーク付け問題に適用して，その効率的プログラムの自動生成を行い，本システムの有効性を示した．本システムは次の 3 つの特徴を持つ．

- 変換規則の記述法が汎用的であり，再利用が可能である．
- 高階マッチング機能によって構成される変換記述言語 CCP を用いるため，抽象度の高いメタプログラミングが可能である．
- CCP の実行の過程の記録として変換スクリプトが得られ，これを対話的に実行する環境があり，デバッグに有効である．

参 考 文 献

1) Pettrossi, A. and Proietti, M.: Rules and Strategies for Transforming Functional and Logic Programs, *Comp.Surveys*, Vol. 28, No. 2, pp. 360–414 (1996).
2) de Moor, O. and Sittampalam, G.: Generic Program Transformation, *Proceedings of the 3rd International Summer School on Advanced Functional Programming (AFP'98)*, LNCS, Vol.1608, Braga, Portugal, Springer-Verlag, pp. 116–149 (1998).
3) Takeichi, M. and Hu, Z.: Calculation Carrying Programs: How to Code Program Transformations (Invited Paper), *International Symposium on Principles of Software Evolution (ISPSE 2000)*, Kanazawa, Japan, IEEE Press

```
-------------- プログラム -----------------
mss = reduce (bmax wsum) . filter conn0 . gen;

reduce f [x]    = x;
reduce f (x:xs) = f x (reduce f xs);
bmax f a b = if f a > f b then a else b;
wsum = reduce (+) . map fc
    where fc (x,m) = if marked (x,m) then x
                                     else 0;
filter p []     = [];
filter p (x:xs) = if p x then x : filter p xs
                         else filter p xs;
conn0 []     = True;
conn0 (x:xs) = if marked x then conn1 xs else conn0 xs;
conn1 []     = True;
conn1 (x:xs) = if marked x then conn1 xs else conn2 xs;
conn2 []     = True;
conn2 (x:xs) = if marked x then False else conn2 xs;
marked (x,b) = b;
gen [] = [[]];
gen (x:xs) = [ x':xs' | x' <- [mark x,unmark x],
                        xs' <- gen xs ];
mark x = (x,True);
unmark x = (x,False);

cata _step _e []      = _e;
cata _step _e (_a:_x) = _step _a (cata _step _e _x);

opt accept (oplus,f) ... 省略

-------------- 変換規則 -----------------
<mmpRule> (reduce (bmax _w) . filter _p . gen) =
    letm _accept . _h = <tupling> _p;
         reduce _oplus . map _f = _w;
         cata _phi2 _phi1 = <applyFusion> _h
    in opt (_oplus,_f) _accept _phi1 _phi2;

<tupling> conn0 =
    letm _h xs = (conn0 xs,conn1 xs,conn2 xs);
         _accept (t0,t1,t2) = t0;
    in _accept . _h;

<fusion> (_f (cata _step _e _xs)) =
    letm _g a (_f x) = _f (_step a x);
         _c = _f _e
    in cata _g _c _xs;

<applyFusion> (_f . _g) = <fusion> (_f . <applyFusion> _g);
<applyFusion> _f        = <fusion> (_f . cata (:) []);

-------------- 変換規則の適用 -----------------
mssOpt = <mmpRule> (<unfold> mss)
```

図 10 mss に対する CCP

(2000).
4) Visser, E.: Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5, *Rewriting Techniques and Applications (RTA'01)* (Middeldorp, A.(ed.)), LNCS, Vol. 2051, Springer-Verlag, pp. 357–362 (2001).
5) Sasano, I., Hu, Z., Takeichi, M. and Ogawa,

```
-------------- プログラム -----------------
mss' = reduce (bmax w) . filter p . gen;
...
p xs = conn0 xs && even xs;
even []     = True;
even (x:xs) = if marked x then odd xs else even xs;
odd []     = False;
odd (x:xs) = if marked x then even xs else odd xs;
h xs = (p xs,even xs,odd xs,conn0 xs,conn1 xs,conn2 xs);
...
accept (t0,t1,t2,t3,t4,t5) = t0;

-------------- 変換規則 -----------------
...

-------------- 変換規則の適用 -----------------
mssOpt' = <mmpRule> (<unfold> mss')
```

図 11 mss に対する CCP

M.: Make it Practical: A Generic Linear-Time Algorithm for Solving Maximum-Weightsum Problems, *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, Montreal, Canada, ACM Press, pp. 137–149 (2000).

6) Bern, M. W., Lawler, E. L. and Wong, A. L.: Linear-Time Computation of Optimal Subgraphs of Decomposable Graphs, *Journal of Algorithms*, Vol. 8, pp. 216–235 (1987).

7) Borie, R. B., Parker, R. G. and Tovey, C. A.: Automatic Generation of Linear-Time Algorithms from Predicate Calculus Descriptions of Problems on Recursively Constructed Graph Families, *Algorithmica*, Vol. 7, pp. 555–581 (1992).

8) Bird, R.: Maximum Marking Problems (2000). Available from http://www.comlab.ox.ac.uk/oucl/work/richard.bird/publications/mmp.ps.

9) Sasano, I., Hu, Z. and Takeichi, M.: Generation of Efficient Programs for Solving Maximum Multi-Marking Problems, *Semantics, Applications, and Implementation of Program Generation (SAIG'01)* (Taha, W.(ed.)), LNCS, Vol. 2196, Firenze, Italy, Springer-Verlag, pp. 72–91 (2001).

10) 篠埜功, 胡振江, 武市正人, 小川瑞史: ナップサック問題およびその発展問題の統一的解法, コンピュータソフトウェア, Vol. 18, No. 2, pp. 59–63 (2001).

11) Bird, R.: Algebraic Identities for Program Calculation, *The Computer Journal*, Vol. 32, No. 2, pp. 122–126 (1989).

12) Bird, R.: *Introduction to Functional Programming using Haskell (second edition)*, Prentice Hall (1998).

13) Bird, R. and de Moor, O.: *Algebra of Programming*, Prentice Hall (1996).

14) Meijer, E., Fokkinga, M. and Paterson, R.: Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire, *Proceedings of the 5th International Conference on Functional Programming Languages and Computer Architecture (FPCA'91)*, LNCS, Vol. 523, Cambridge, Massachusetts, Springer-Verlag, pp. 124–144 (1991).

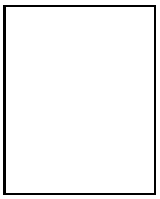
15) Hu, Z. and Takeichi, M.: Calculation Carrying Programs, Technical Report METR 99-07, Department of Mathematical Engineering, University of Tokyo, Japan (1999).

16) de Moor, O. and Sittampalam, G.: Higher-order matching for program transformation, *Theoretical Computer Science*, Vol. 269, No. 1-2, pp. 135–162 (2001).

17) Sittampalam, G.: *Higher-order matching for program transformation*, PhD Thesis, University of Oxford (2001).

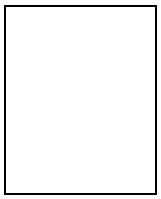
18) Jones, S. P. and Hughes, J.: The Haskell 98 Report (1999). Available from http://www.haskell.org/definition/.

(平成？年？月？日受付)
(平成？年？月？日採録)



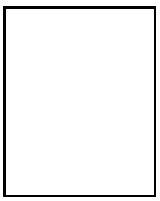
横山　哲郎

昭和 52 年生．平成 13 年東京大学工学部計数工学科卒．同年東京大学情報理工学系研究科入学．プログラム変換に興味を持つ．



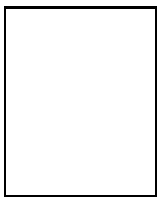
胡　　振江（正会員）

1966 年生．1988 年中国上海交通大学計算機科学系を卒業．1996 年東京大学大学院工学系研究科情報工学専攻博士課程修了．同年日本学術振興会特別研究員を経て，1997 年東京大学大学院工学系研究科情報工学専攻助手，同年 10 月同専攻講師，2000 年同専攻助教授．2001 年より東京大学情報理工学系研究科助教授，同年 12 月より科学技術振興事業団さきがけ 2 1 研究者を兼任．博士（工学）．日本ソフトウエア科学会，情報処理学会，ACM 各会員．



篠埜　　功

1973 年生．1997 年東京大学工学部計数工学科卒．1999 年同大学大学院工学系研究科情報工学専攻修士課程修了．同年博士課程に進学，現在に至る．2001 年 4 月より日本学術振興会特別研究員．関数プログラミング，プログラム変換，アルゴリズムの導出，グラフアルゴリズムなどに興味を持つ．日本ソフトウェア科学会，ACM 各会員．



武市　正人（正会員）

1948 年生。1972 年東京大学工学部助手、講師、電気通信大学講師、助教授、東京大学工学部助教授、を経て 1993 年東京大学大学院工学系研究科教授（情報処理工学講座）、2001 年より同大学大学院情報理工学系研究科教授、現在に至る。工学博士。プログラミング言語、関数プログラミング、構成的アルゴリズム論の研究・教育に従事。日本ソフトウェア科学会、情報処理学会、日本応用数理学会、ACM 各会員。