最大マーク付け問題の効率的プログラムの自動生成

横山 哲郎[†] 篠埜 功[††]
胡 振江[†,††] 武市 正人[†,††]

最大マーク付け問題とは，与えられたデータの要素に対するマーク付けのうち，ある条件を満たすマーク付けの中から，ある重み関数に関して最適なマーク付けを求める問題である．最大マーク付け問題はさまざまな最適化問題を含み，それを解く効率のよいプログラムの自動生成は意義深いが，実用的な自動生成システムを構築するのは難しいことが知られている．本論文では，最大マーク付け問題の効率的プログラムの自動生成時に有効な変換規則とプログラム変換を簡潔に記述できる言語を提案するとともに，それを支援するシステムの実現手法を示し，具体的な例を用いてその有効性を示す．

Automatic Generation of Efficient Programs for Maximam Multi-Marking Probrems

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

The maximum marking problem is the problem of marking the entries of some given data structure in such a way that a given constraint is satisfied and it maximizes the value computed from the marked entries with respect to a given weight function. This kind of problems are practically interesting enough to cover many optimization problems. This calls for automatic systems to generate efficient programs from the initial specification of the problems, but the construction of such system is known to be a challenge. In this paper, we shall propose a general framework for automatic generation of efficient programs for the maximum marking problems, showing how to naturally describe and efficiently implement transformation and its strategy, and explaining how to construct a practical system supporting both high level generation process as well as low level debugging. We illustrate our method with several interesting examples.

1. はじめに

プログラムを書くとき，実行時の効率とソースコードの簡潔さのどちらを優先させるかという難しい問題がある．理解しやすく高いモジュール性をもつプログラムは効率がよくないことが多く，逆に効率のよいプログラムが理解しにくいことも多い．理解しやすいプログラムを仕様とし，プログラム変換によって効率のよいプログラムを得ることでこの問題を解決できる．プログラム変換を利用したプログラムの開発においては，仕様の記述と，プログラム変換のための変換規則，変換戦略の記述に分割できることが利点である．

最大マーク付け問題[1)] という問題を考えてみる．最大マーク付け問題とは与えられたデータの要素に対するマーク付けの中から，ある条件 p を満たすマーク付けを取り出し，その中からある重み関数 w に関して最適なマーク付けを求める問題である．最大マーク付け問題はさまざまな最適化問題を含み，多くの研究がなされている[1)2)3)4)5)6)]．この問題は次のようなプログラムで解くことができる．

$$mmp\ p\ w \;=\; \uparrow_w / \circ \mathit{filter}\ p \circ gen$$

関数 gen は可能なマーク付けをすべて生成する．次にその中から性質 p を満たすものを $\mathit{filter}\ p$ で取り出す．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．

この最大マーク付け問題を解くプログラムは理解し易いが，計算量は指数オーダーであるため，効率が悪い．一方，性質 p，重み関数 w がそれぞれある条件を満たすときには線形時間アルゴリズムが存在する[2)3)1)]が，そのプログラムは理解し易いものではない．

† 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

したがって，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，実際多くの研究者を魅了してきた．例えば，文献 3) では線形プログラムが自動生成されている．しかし，この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．我々の知る限りにおいて，最大マーク付け問題を解く線形時間プログラムの自動生成を行うシステムは存在しない．

プログラムの自動生成システムを構築するには，効果的な変換規則と，変換規則と変換戦略を簡潔に記述できる言語が必重要である．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ンに適用できる汎用性をもつ規則であること．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「自然数 p, q が存在して $n = p * q$ を満たすならば, n を $p * q$ に書き変える」という規則は構成的でない例である．

変換戦略記述言語を用いることで変換規則をどのように適用するかを指定することができる[7)8)]．MAG システム[9)] のように戦略を記述しないものもあるが，ユーザの意図した導出をすることは難しい．そこで変換戦略の記述が重要になってくる．

本論文では, 最大マーク付け問題の効率的プログラムの自動生成時に有効な変換規則と変換戦略記述言語を提案し，それを支援する環境の実現を示す．本システムは次の 3 つの特徴を持つ．

- 我々の変換規則は, 最大マーク付け問題一般に適用できる汎用的な変換規則であり，多くの問題に適用できる．また，この変換規則は構成的であり，実現に向いている．
- 我々の変換戦略は，メタ言語である運算付随プログラム[7)]（Calculation Carrying Program，以下 CCP と略記）によって記述される．従来の変換記述言語[8)] の多くは 1 階マッチング機能しかなく，直接に記述可能な規則が限定されるが，CCP は高階マッチング機能をもつので広範囲の規則が記述できる．また CCP はプログラム変換の制御をしやすい．例えば，変換規則を適用できる場合にだけ適用を行なうことを記述したり，ある条件を満たすまで繰りかえし適用することを記述したり，変換戦略を別の変換戦略の一部で用いたりすることが簡単にできる．このように直観的に理解しやすい形で変換戦略を記述できることが，CCP が変換戦略の記述に向いている理由の一つである．
- 我々のシステムでは，CCP をコンパイルし中間言語であるスクリプトに変換できる．また得られたスクリプトを対話的に実行する環境があり, デバッグを行いやすい．

本論文の構成は次のようになっている．2 章で，最大マーク付け問題を定義し，線形時間アルゴリズムを得る変換規則を示す．3 章ではこのプログラムの自動生成を行うシステムの構成を示し，4 章でこのシステムを用いて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行う．5 章で結論を述べる．

なお以降本論文では 1) での表記法にあわせ関数型言語 Haskell[10)] によく似た記法を用いる．

2. 最大マーク付け問題

この章では，最大マーク付け問題を定義し，線形時間アルゴリズムを得る変換規則を示す．なお一般に最大マーク付け問題のマークの種類は多種類でもかまわない[1)] が，本論文では問題を簡単にするためにマークの種類を 2 種類とする．

2.1 最大マーク付け問題の仕様

最大マーク付け問題とは以下のような問題である．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 つ見つける問題と定義できる．最大マーク付け問題の解は, すべてのマーク付けの中から性質 p を満たすものを $\mathit{filter}\ p$ 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.

$$\mathit{mmp}\ p\ w \;=\; \uparrow_w / \circ \mathit{filter}\ p \circ \mathit{gen} \qquad (1)$$

性質を p, 重み関数を w，入力データを x とするとき, $\mathit{mmp}\ p\ w\ x$ 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.

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

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

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

$\uparrow_f$ は次のように定義される．

$$
\begin{array}{rcl}
a \uparrow_f b & = & a, \quad \textbf{if } f\ a > f\ b \\
 & = & b, \quad \textbf{otherwise}
\end{array}
$$

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はさまざまな問題を含んでいる．例えば，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．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ために，要素にマークを付けることによって部分列を表わす．例えば，上の例では解は

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

と表わされる．

最大部分列和問題を (1) 式の形で表わすためには，性質 p 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関数であり，次のように記述できる．

$$
\begin{array}{lcl}
\mathit{conn}_0\ [\,] & = & \mathit{True} \\
\mathit{conn}_0\ (x : \mathit{xs}) & = & \textbf{if } \mathit{marked}\ x \textbf{ then } \mathit{conn}_1\ \mathit{xs} \\
 & & \textbf{else } \ \mathit{conn}_0\ \mathit{xs} \\
\mathit{conn}_1\ [\,] & = & \mathit{True} \\
\mathit{conn}_1\ (x : \mathit{xs}) & = & \textbf{if } \mathit{marked}\ x \textbf{ then } \mathit{conn}_1\ \mathit{xs} \\
 & & \textbf{else } \ \mathit{conn}_2\ \mathit{xs} \\
\mathit{conn}_2\ [\,] & = & \mathit{True} \\
\mathit{conn}_2\ (x : \mathit{xs}) & = & \textbf{if } \mathit{marked}\ x \textbf{ then } \mathit{False} \\
 & & \textbf{else } \ \mathit{conn}_2\ \mathit{xs}
\end{array}
$$

重み関数 w はマークの付いた要素の和を求める関数とすればよい．

これらを用いて最大部分列和問題は

$$\mathit{mss} \ = \ \uparrow_{\mathit{wsum}} / \circ \mathit{filter}\ \mathit{conn}_0 \circ \mathit{gen} \qquad (2)$$

のように表わすことができる．

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

最大部分列和問題を解く (2) 式の関数 mss の仕様は単純であり人間には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．次節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．

2.2 変換規則

最大マーク付け問題はある条件を満たすとき線形アルゴリズムが存在する．ここでは，前節の最大マーク付け問題の仕様である (1) 式から線形プログラムを得るための変換規則を示す．

その準備として対象とする再帰データ型を定義し，その上の基本的再帰関数である catamorphism を定義する．

2.2.1 再帰データ型

$$
\begin{array}{lcl}
D\ \alpha & = & C_1\ (\alpha, D_1, \ldots, D_{n_1}) \\
 & | & C_2\ (\alpha, D_1, \ldots, D_{n_2}) \\
 & | & \cdots \\
 & | & C_k\ (\alpha, D_1, \ldots, D_{n_k})
\end{array}
$$

$D\ \alpha$ 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_i 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 $D_1\ \ldots\ D_{n_i}$ から $D\ \alpha$ 型のデータを構成する. この形で定義されるデータ型は polynomial データ型であり, リスト, 2 分木, 根つき木 (rooted tree)[2], 直列並列グラフ (series-parallel graph)[12], 木幅 (tree width) に上限を設けたグラフ[13] などを含む．

2.2.2 Catamorphism

再帰データを入力とする関数の中で, 基本的再帰関数のクラスに catamorphism がある. これは, プログラム変換において最も重要な概念の 1 つである[14),15)]. 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.

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

e, $\oplus$ を変えることにより, さまざまな関数を表すことができる. 関数 cata 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 $\oplus$ を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, 通常, $\mathit{cata} = (\![e \vartriangle \oplus]\!)$ と記述する.

定義 1 (Catamorphism)
次の等式により定義される関数 f を再帰データ $D\ \alpha$ 上の catamorphism と呼び, $(\![\phi_1, \ldots, \phi_k]\!)_{D\ \alpha}$ と表す.

$$
\begin{aligned}
& opt\ (\oplus, f)\ accept\ \phi_1 \ldots \phi_k\ x = \\
& \quad getdata\ (\uparrow_{snd} / [\,(c, w, r^*) \mid (c, w, r^*) \leftarrow (\!|\psi_1, \ldots, \psi_k|\!)_D\ x,\ accept\ c\,]) \\
& \quad \mathbf{where}\ \psi_i\ (e, cand_1, \ldots, cand_{n_i}) = \\
& \qquad eachmax\,[\,(\phi_i\ (e^*, c_1, \ldots, c_{n_i}), \\
& \qquad\qquad f\ e^*\ \oplus w_1 \oplus \ldots \oplus w_{n_i}, \\
& \qquad\qquad C_i\ (e^*, r_1^*, \ldots, r_{n_i}^*)) \mid \\
& \qquad\qquad\quad e^* \leftarrow [mark\ e, unmark\ e], \\
& \qquad\qquad\quad (c_1, w_1, r_1^*) \leftarrow cand_1, \cdots, (c_{n_i}, w_{n_i}, r_{n_i}^*) \leftarrow cand_{n_i}] \qquad (i = 1, \ldots, k) \\
\\
& mark\ x\ =\ (x, True) \\
& unmark\ x\ =\ (x, False) \\
& marked\ (\_, m) = \mathbf{if}\ m\ \mathbf{then}\ True\ \mathbf{else}\ False \\
\\
& weight\ (x, \_)\ =\ x \\
& getdata\ (\_, \_, x)\ =\ x \\
\\
& eachmax\ xs = foldl\ f\ [\ ]\ xs \\
& \quad \mathbf{where} \\
& \qquad f\ [\ ]\ (c, w, r) = [(c, w, r)] \\
& \qquad f\ ((c, w, r)\ :\ opts)\ (c', w', r') = \\
& \qquad\quad \mathbf{if}\ c == c'\ \mathbf{then} \\
& \qquad\qquad \mathbf{if}\ w > w'\ \mathbf{then}\ (c, w, r)\ :\ opts \\
& \qquad\qquad \mathbf{else}\ opts\ +\!\!+\ [(c', w', r')] \\
& \qquad\quad \mathbf{else}\ (c, w, r)\ :\ f\ opts\ (c', w', r')
\end{aligned}
$$

図 1　最適化関数 opt

$$f\ (C_i\ (e,\ x_1, \ldots, x_{n_i})) = \phi_i\ (e,\ f\ x_1, \ldots, f\ x_{n_i}) \quad (i = 1, \ldots, k)$$

$(\!|\phi_1, \ldots, \phi_k|\!)_{D\ \alpha}$ における添字の $D\ \alpha$ は, 文脈から明らかな場合には省略することもある. □

2.3　変 換 規 則

最大マーク付け問題の最適化規則として以下のものを用いる．これは文献 1) の最適化定理に対して，文献[5] の重み関数に関する拡張を行ったものである．

定理 **1** (最適化規則)
最大マーク付け問題

$$mmm\ p\ w\ =\ \uparrow_w /\ \circ\ filter\ p\ \circ\ gen.$$

は，もし (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，また，(2) 重み関数 w が

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

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

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

によって表される．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. □

前節で記述した最大部分列和問題の仕様の (2) 式は，最適化規則を適用することにより次の線形時間アルゴリズムに変換される．

$$
\begin{aligned}
& mss\ =\ opt\ (\oplus, f)\ accept\ \phi_1\ \phi_2 \\
& accept\ (t_0, t_1, t_2)\ =\ t_0 \\
& \phi_1 = (True, True, True) \\
& \phi_2\ x\ (t_0, t_1, t_2) = \mathbf{if}\ marked\ x \\
& \qquad\qquad \mathbf{then}\ (t_1, t_1, False) \\
& \qquad\qquad \mathbf{else}\ (t_0, t_2, t_2)
\end{aligned}
$$

この変換は機械的に行うことが可能であり，実際次章述べるシステムを用いて自動生成を行うことが可能である．

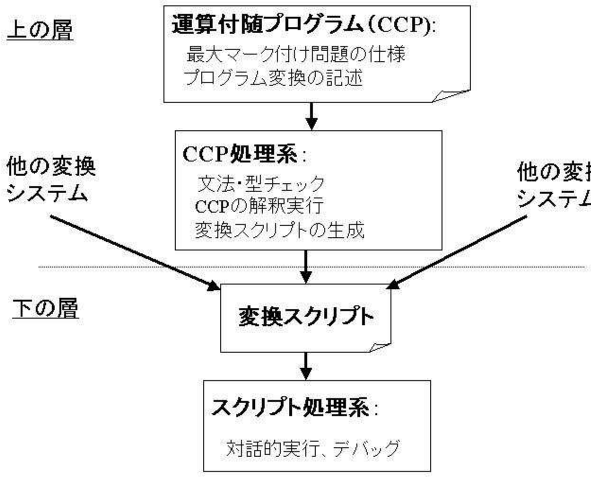

図 2　システムの構成

3. 自動生成のシステムの構成

この章では，MAG システムの上で Haskell 言語で開発した最大マーク付け問題を自動生成するシステムの構成と実現について説明する．

3.1 システムの概要

システムの概要は図 2 のように 2 つの階層からなる．上の層は変換戦略の仕様記述のための高レベルの記述の枠組で CCP[7] の記述を利用する（3.2 節）．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．下の層は細かい変換のステップを記述するためのスクリプトの枠組である（3.3 節）．CCP の文法，型をチェックした後，上の層の CCP から下の層のスクリプトへの変換が行われる（3.4）．

上の層の CCP は汎用的なメタプログラムを書くことに適している．スクリプトは，個別のプログラムの仕様に依存するので，少しでも異なるプログラムの仕様から効率のよいプログラムを得るためには用いることができない．しかし CCP によって記述されたその変換の戦略はほとんど変更することなく別の似たプログラムの仕様を変換することに再利用することができる．

下の層のスクリプトは，新たに変換規則，変換戦略を開発するときは，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（3.3 節）．

CCP と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スクリプトとして記録すれば，低レベルの階層で実行することが可能である．

3.2 運算付随プログラム CCP

CCP とは，運算規則 (calculation rule) の付随したプログラムである[16]．プログラム運算とは，運算規則を適用することによりプログラムを順次変換していき，効率のよいプログラムを得る過程を指す．一般に，プログラム運算は手動で行われ，用いられる運算規則や，運算規則の適用の仕方は運算対象のプログラムに応じて変わる．運算を計算機上で行うときには，この運算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，用いられる運算規則および運算規則の適用法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，プログラマが運算法を指定することが可能となる．また，融合変換，組化等，汎用的な運算については，一度記述すれば再利用が可能である．さらに，CCP には，高階パターンマッチングの機能が備わっており，通常行われる運算をそのまま記述することが可能である．

3.2.1 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 とはいかなるものであるかを示す．この問題は，各要素を 2 乗して足し合わせることにより解くことができる．2 乗する関数を sq，数列の和を求める関数を sum とすると，

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

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．そのような形に関数 $sumsq$ を変換することを考えてみる．これには，融合変換という変換を

行えばよい．

定理 2 (融合規則) 任意の a, x について

$$g\ a\ (f\ x) = f\ (step\ a\ x)$$

が成り立つならば，

$$f \circ foldr\ step\ e = foldr\ g\ (f\ e)$$

が成り立つ． □

上記規則中の関数 $foldr$ は，リストの畳み込み関数の 1 つであり，次のように定義される．

$$foldr\ step\ e\ [\,] = e$$
$$foldr\ step\ e\ (x : xs) = step\ x\ (foldr\ step\ e\ xs)$$

この定理を適用するために，まず，$map\ \ sq$ を $foldr\ step\ e$ の形で記述する．

$$map\ sq = foldr\ step\ [\,]$$
$$\quad \mathbf{where}\ step\ y\ ys = (sq\ y) : ys$$

次に，

$$g\ y\ (sum\ ys) = sum\ (step\ y\ ys)$$

を満たす関数 g を求める．次の関数 g は，この式を満たす．

$$g\ y\ s = sq\ y + s$$

$sum\ [\,] = 0$ であるので融合定理を適用することにより，

$$sum \circ map\ sq = foldr\ g\ 0$$

を得る．関数 $foldr\ g\ 0$ は，二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．

このような運算を計算機で行う場合には，運算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 においては，運算対象のプログラムに，運算法を付随させる．たとえば，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，上段部分が運算対象のプログラム，中段部分が運算規則 (fusion, applyFusion)，下段部分が，運算対象のプログラム (sumsq) からどのように効率のよい関数 (sumsqOpt) を得るかを示している．まず，上段部分では，sumsq の仕様を記述している．この仕様では，2 乗和を求めるときにリストを生成するが，融合変換によってリストを生成しないようにできる．こ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 2) を記述したものである．融合規則を適用する際には，関数融合 `f . g` の右側の関数 `g` が `foldr step e` の形で記述されていなければならないので，これを行うための運算規則として applyFusion を定義している．applyFusion では，関数融合の最も右側の関数を `foldr (:) []` と融合規則<fusion>を適用することによって `foldr step e` の形に変換し，その後，順に融合規則<fusion>を適用していくという変換が行われる．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．

CCP においては，関数定義と運算規則定義を混在させて記述できる．関数と運算規則は，運算規則名を<>で囲むことによって区別している．また，CCP においては，規則の定義中で他の運算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，基本的な運算を組み合わせて複雑な運算を記述することができるようになる．

また，CCP においては，パターン変数名を_で始めることにより，局所変数との区別を行っている．letm 式のマッチングの

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

においてにはアンダースコア_がついていないが，これは，a と x がパターン変数ではなく，局所変数であることを表している．具体的には，等式

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

が，変数_a,_x がそれぞれの型の任意の値をとったときに成り立つことを意味する．

3.2.2 CCP の記述言語

本論文においては，CCP の記述言語を図 4 のように定義する．CCP においては，関数定義 (*funDef*) と規則定義 (*ruleDef*) 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，メタラムダ抽象，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 (higher order pattern matching)[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 (pattern variable) に対する置換 (substitution) が求められる．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

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

  各式 $e_{p_1}, \ldots, e_{p_n}$ 中のパターン変数に対する置換を高階マッチングアルゴリズム[17] 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

```
-------------- 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;
foldr _step _e [] = _e;
foldr _step _e (_a:_x) = _step _a (foldr _step _e _x);
(_f . _g) _x = _f (_g _x);

-------------- 運算規則 -----------------
<fusion> _f (foldr _step _e _xs) =
    letm _g a (_f x) = _f (_step a x);
         _c = _f _e
    in foldr _g _c _xs;

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

-------------- 運算規則の適用 -----------------
sumsqOpt = <applyFusion> (<unfold> sumsq)
```

図 3 Calculation carrying program の例 — sumsq

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 $e_{p_1}, \ldots, e_{p_n}$ と順に高階パターンマッチングアルゴリズム[17] によってパターンマッチングを行い，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 $\langle r \rangle\ e$

規則 r を式 e に対して適用する．規則 r が

$$\langle r \rangle\ e_p = e_b$$

と定義されているとき，e と e_p を高階パターンマッチングアルゴリズム[17] によってパターンマッチングを行い，得られた e_p 中のパターン変数への置換によって 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[10),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，前節で述べたように，`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．

3.3 変換スクリプト

スクリプトの文法を図 5 のように定義する．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．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．スクリプトで用いる命令は環境設定命令（*EnvCommand*），実行命令（*IntCommand*），デバッグ用命令（*DebugCommand*）の 3 種類に大きく分かれる．

環境設定命令は以下の 4 種類ある．命令 *setPath* はプログラムと変換規則の置いてあるディレクトリへのパスを指定する．命令 *loadTheory* は変換規則のかかれたファイルを読み込む．命令 *setExp* は式を始式にセットする．

実行命令は以下の 5 種類である．step は適用するルール名と適用する部分式の位置をうけとってルールを部分式に適用して得られた結果を式にセットする．なお，部分式の位置の表わしかたとして，B は λ 式の

CCP:

$$ccp \quad ::= \quad def_1; \ldots; def_n \qquad \text{定義列}$$

定義:

$$\begin{array}{lll} def & ::= & funDef \quad \text{関数定義} \\ & | & ruleDef \quad \text{規則定義} \end{array}$$

関数定義:

$$funDef \quad ::= \quad f\ pats = e \qquad \text{関数定義}$$

規則定義:

$$ruleDef \quad ::= \quad \langle r \rangle\ e_p = e_b \qquad \text{規則定義}$$

式:

$$\begin{array}{llll} e & ::= & haskellExp & \text{関数型言語 Haskell の式} \\ & | & em & \text{高階パターンマッチングを含む式} \end{array}$$

高階パターンマッチングを含む式:

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

図 4 CCP の定義

本体を表わし，F は関数適用の関数部分を表わし，A は関数適用の引数部分を表わす．命令 $beginDerivation$ は適用するルール名と適用する部分式の位置をうけとってルールの条件式にマッチした式の部分導出を始める．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するかを調べ，マッチしたら得られた置換を記憶する．命令 $endDerivation$ はもし現在部分導出が終っていたら，いままでの部分導出によって得られた置換を用いてルールの右辺を適当に置きかえ命令 $beginDerivation$ で指定したルールを部分式に適用して得られた結果を式にセットすることで部分導出から脱出する．命令 $createRule$ は始式から現在の式へと変換するルールを指定した名前で追加する．

デバッグ用命令は対話的に実行するときに用いる．命令 $help$ で現在実行できる命令を表示する．命令 $showDerivation$ で今までの導出を表示する．命令 $quit$ で以下の命令をも実行せずに終了する．命令 $showBindings$ で現在の式における局所変数を表示する．命令 $showStack$ で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 $undo$ で 1 ステップ元に戻る．命令 $showSubExpressions$ で現在の式の部分式をすべて表示する．命令 $showTheory$ で現在使用可能なルールをすべて表示する．

スクリプトの例としてリストをうけとりリストの要素の自乗の和を返す関数命令 $sumsq$ の変換例をあげる．この変換のスクリプトは図 6 のようになる．なお左側の数字とコロンは説明のために書き加えたものである．1 行目の`#!`はこのファイルがスクリプトあることを表わしている．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` を読み込む．4 行目の命令 $setExp$ で式 `sumsqOpt` を始式をセットする．5-8 行目の命令 $step$ で式を変換していく．例えば，5 行目では `sumsq` という変換規則を現在の式の，λ 式の本体の，関数適用の関数部分に適用した結果を式にセットし直す．ここで，システムの内部では式はすべて η-展開してあるので，$sumsq$ は $\lambda xs\ .\ sumsq\ xs$ と表現されている．この場合，λ 式の本体は $sumsq\ xs$，さらにその関数適用の関数部分は $sumsq$ となる．この部分式は，図 7 の 2 行目に書いてある変換規則 `sumsq` の左辺とマッチす

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*	プログラムと変換規則のファイルのあるディレクトリへのパス指定
	|	*loadTheory TheoName*	変換規則の読込み
	|	*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 5　スクリプトの文法

るので，この部分式と変換規則 `sumsq` の右辺に置きかえる置換，すなわち $sumsq$ から $sum \circ map\ sq$ への置換得られる．この置換で部分式 $sumsq$ を置換した結果を元の式に戻すと

$$\lambda xs\ .\ sum\ (map\ sq\ xs)$$

が得られる．また，簡潔に表示するため，図 8 の 5 行目のように，この式は η-簡約されて表示される．9 行目の命令 *beginDerivation* で，変換規則 `fusion` を部分式"`BFABF`"に適用する前に，部分導出に移る．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．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．部分導出の中では命令 *match* で置換を得て，命令 *endDerivation* で部分導出から脱出する．21 行目の命令 *showDerivation* で今まで行った導出の出力（図 8）をする．22 行目の命令 *quit* でシステムの実行を終了する．

変換規則のかかれたファイルは図 7 のようになる．拡張子は.eq である．プログラムのかかれたファイルと同じく，ここでこのファイルは MAG システムの変換規則の入力ファイルと同じ形式である．このファイル

```
#!

setPath "/home/yokoyama/ys/examples"
loadTheory "sumsq"
setExp "sumsqOpt"

step "sumsqOpt" "BF"
step "sumsq" "BF"
step "applyFusion1" "BF"
step "applyFusion2" "BFA"

beginDerivation "fusion" "BFABF"
    step "map2" "BB"
    match
    step "map1" ""
    match
endDerivation

beginDerivation "fusion" "BF"
   step "sum2" "BB"
   match
   step "sum1" ""
   match
endDerivation

showDerivation
quit
```

図 6 スクリプトの例 – sumsq.ys

の記法も Haskell に非常によくにている．“{-”，“-}” でかこわ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いるときにオプショナルに記述する．条件式は “;” で区切られ，“=” の左辺が項を表わし，“=” の右辺がパタン変数を含む式であるパタンを表わす． 上記のスクリプトを実行して得られる出力は図 8 の通りである．“{-”，“-}” でかこわ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．

3.4 CCP の実行、スクリプトへの変換

本節では，CCP の実行を行う．CCP を実行の過程の記録としてスクリプトが得られる．これにより，CCP からスクリプトへの変換が行われる．

CCP の実行の際には，CCP 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．関数定義は，定義を展開する書き換え規則に変換する．運算規則定義は，右辺の中の規則をのぞ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

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

以下のような条件付式に変換する．

```
e, if {eb1 = ep1;
       eb2 = ep2;
       ...
       ebn = epn}
```

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

```
{- 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;
foldr1 : foldr step e []    = e;
foldr2 : foldr step e (a:x) = step a (foldr step e x);

fusion : _f (foldr _step _e _x) = foldr _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 . foldr (:) [];

sumsqOpt: sumsqOpt = sumsq
```

図 7　変換規則が書いてあるファイル – sumsq.eq

ンを右辺に書くためである．

– メタ case 式

　　casem e **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 e に変換する．

```
ep1 = e1;
ep2 = e2;
...
epn = en
```

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 の運算規則に指定された順で適用していくことによって行われる．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，スクリプトが得られる．CCP の実行過程は，以下で例を用いて示す．

ここでは，例として sumsq 問題の CCP からスクリプトへの変換を行う．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，スクリプトへの変換を行う．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，各運算規則は，規則の右辺中の規則名を機械的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 9 のような書き換え規則が得られる． 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 9 の書き換え規則を，図 3 の CCP において指定された順で適用することにより，運算が行われる．運算の過程は，図 8 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，次に，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 とマッチしないため，applyFusion2 が適用される．これにより，sum . (map sq . foldr (:) []) が得られるが，このとき，次には (map sq . foldr (:) []) 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 回適用されることとなり，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．

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

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

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

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

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

図 8 実行結果 – sumsq.out

この書き換えの過程において適用した規則，規則の適用箇所をスクリプトの形で記録すると，図 6 のようなスクリプトが得られる．これにより，CCP からスクリプトへの変換が行われる．

4. 最大マーク付け問題の自動生成の例

本章では,3 章で記述したシステムを用いて，いくつかの最大マーク付け問題の線形時間プログラムを自動生成する．

4.1 最大部分列和問題

ここでは最大部分列和問題の自動生成を行なう．最大部分列和問題は 2 章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．

$$mss \;=\; \uparrow_w / \;\circ\; \mathit{filter}\ conn_0 \;\circ\; gen \qquad (3)$$

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 $\mathit{filter}\ conn_0$ 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 $\uparrow_w /$ によって重み関数 w 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．(3) 式の定義は図 10 のプログラムの 1 行目に書かれている．な

```
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;
foldr1: foldr step e [] = e;
foldr2: foldr step e (a:x) = step a (foldr step e x);
compose: (_f . _g) _x = _f (_g _x);

fusion: _f (foldr _step _e _xs) = foldr _g _c _xs,
        if{ _f (_step a x) = _g a (_f x);
            _f _e  = _c}

applyFusion1: _f . _g = _f . _g;
applyFusion2: _f = _f . foldr (:) [];

sumsqOpt: sumsqOpt = sumsq;
```

図 9 sumsq の書き換え規則

お，$bmax\ w$ は $(\uparrow_w)$ を，$reduce$ は $(/)$ をそれぞれ表わす．

mmpRule は定理 1 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 をコンパイルすることで図 6 と同じようなスクリプトを生成する．

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について考える．第 2 章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である．図 10 の CCP の変更するべき部分が図??に記述されている．このように運算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った．

5. 結　論

本論文では，最大マーク付け問題の効率的プログラムの自動生成時に有効な変換規則，変換戦略記述言語を提案し，それを支援するシステムの実現手法を示した．我々のシステムはプログラム変換の戦略の記述を効果的にするためにシステムを 2 つの階層に分割していることが特徴である．また，いくつかの最大マーク付け問題対して，実用的線形時間アルゴリズムの自動生成を行うことによって我々の手法が有効であることを示した．

参 考 文 献

1) Sasano, I., Hu, Z., Takeichi, M. and Ogawa, 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).

2) 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).

3) 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).

4) Bird, R.: Maximum Marking Problems (2000). Available from http://www.comlab.ox.ac.uk/

oucl/work/richard.bird/publications/mmp.ps.
5) 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.)), Lecture Notes in Computer Science, Vol. 2196, Firenze, Italy, Springer-Verlag, pp. 72–91 (2001).
6) 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).
7) 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 (2000).
8) Visser, E.: Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5, *Rewriting Techniques and Applications (RTA'01)*, Springer-Verlag (2001).
9) de Moor, O. and Sittampalam, G.: Generic Program Transformation, *Proceedings of the 3rd International Summer School on Advanced Functional Programming (AFP'98)*, LNCS 1608, Braga, Portugal, Springer-Verlag, pp. 116–149 (1998).
10) Bird, R.: *Introduction to Functional Programming using Haskell (second edition)*, Prentice Hall (1998).
11) Bird, R.: Algebraic Identities for Program Calculation, *The Computer Journal*, Vol. 32, No. 2, pp. 122–126 (1989).
12) Takamizawa, K., Nishizeki, T. and Saito, N.: Linear-Time Computability of Combinatorial Problems on Series-Parallel Graphs, *Journal of the Association for Computing Machinery*, Vol. 29, pp. 623–641 (1982).
13) Arnborg, S., Courcelle, B., Proskurowski, A. and Seese, D.: An Algebraic Theory of Graph Reduction, *Journal of the ACM*, Vol. 40, No. 5, pp. 1134–1164 (1993).
14) Bird, R. and de Moor, O.: *Algebra of Programming*, Prentice Hall (1996).
15) 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)*, Lecture Notes in Computer Science, Vol. 523, Cambridge, Massachusetts, ACM Press, pp.124–144 (1991).
16) Hu, Z. and Takeichi, M.: Calculation Carrying Programs, Technical Report METR 99-07, Department of Mathematical Engineering, University of Tokyo, Japan (1999).
17) de Moor, O. and Sittampalam, G.: Higher-order Matching for Program Transformation, *Theoretical Computer Science*, Vol. to appear (1999).
18) Jones, S. P. and Hughes, J.(eds.): *The Haskell 98 Report* (1999). Available from http://www.haskell.org/definition/.

```
-------------- プログラム -----------------
mss = reduce (bmax w) . 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;
w = 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);

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

opt accept (oplus,f) = ... 省略

-------------- 運算規則 -----------------
<mmpRule> (reduce (bmax _w) . filter conn0 . gen) =
    letm _accept . _h = <tupling> conn0;
         reduce _oplus . map _f = _w;
         foldr _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 (foldr _step _e _xs) =
    letm _g a (_f x) = _f (_step a x);
         _c = _f _e
    in foldr _g _c _xs;

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

-------------- 運算規則の適用 -----------------
mssOpt = <mmpRule> (<unfold> mss)
```

図 10　mss に対する CCP

```
-------------- 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;

-------------- 運算規則 -----------------
...

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

図 11 mss に対する CCP