JAIST Repository
https://dspace.jaist.ac.jp/

Title	関数型プログラムにおけるプログラム変換の研究
Author(s)	佐賀，正芳
Citation	
Issue Date	1998-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/1160
Rights	
Description	Supervisor:外山 芳人，情報科学研究科，修士

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

修 士 論 文

関数型プログラムにおけるプログラム変換の研究

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

佐賀正芳

1998 年 2 月 13 日

Copyright © 1998 by Masayoshi Saga

目 次

第 1 章

はじめに

1.1 研究の背景・目的

プログラム変換とは、あるプログラムから、その正しさを保持しつつ、効率化された新たなプログラムを得るものである。自動化されたプログラム変換では、人間の手を介さずにプログラムの実行時間の短縮をはかれるので、プログラム開発において非常に有効である。また、コンパイラとは異なり、ときにはアルゴリズムそのものの変換も可能なので、変換を通して新しいアルゴリズムの開発に寄与できる [2, 8, 13]。

プログラム変換が注目を浴びるきっかけとなったのは Burstall & Darlington(1977) の研究である。彼らは、与えられたプログラムに変換規則を次々と適用して変換を繰り返すことで、様々なプログラム変換が可能になることを示した [2, 13]。

プログラム変換で、Burstall & Darlington が示した手法のうち定義 (definition)、たたみ込み (folding)、展開 (unfolding) を主たる規則とする戦略は、合成戦略 (composition strategy) と呼ばれる [8]。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る。また、たたみ込みとは、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。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。

Wadler(1990) は、合成戦略のひとつとして切り払い (deforestation) と呼ばれる手法を提案した [15]。関数型プログラムでは、多くの関数の合成によってプログラミングが行なわれるが、これら関数をつなぐ役割を果たしているのがリストや木といったデータ構造で

ある。だが、このようなデータ構造を計算途中で生じさせるプログラムは、それらを生じさせないものと比較して一般に計算効率が悪い。そこで、前者から後者へとプログラムを変換することを目的として切り払いの戦略が考え出された。

切り払いは、7つの変換規則をプログラムに適用することで実現される。変換規則は、展開、たたみ込みのような従来の規則を含むもので、比較的理解しやすい。また、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。

アルゴリズムの効率化に関しては、Wadler 自身が、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 [15]。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。また Sϕrensen(1994)、Seidl(1996) らは、変換によって大幅に増大するパラメータと関数呼び出しの効率化について研究している [11, 9]。

Wadler の切り払いは、対象のプログラムを、計算途中に木等のデータ構造を生じさせない関数の組合せから成るものに限定しているので、そのままではごく一部のプログラムにしか対応できない。Chin(1994) は印付切り払いを発展させ、全ての一階のプログラムに対応し停止性も保証されたアルゴリズムを提案した [3]。さらに、Sϕrensen(1994) は、Chin より効率の良い印付けの方法を提案している [11]。

関数型プログラムは、高階関数を直接扱えるので、プログラミングしやすく、出来上がったプログラムも理解しやすいという特長を持つ。しかし、Wadler の切り払いでは高階関数をそのまま扱えない。そこで、Wadler は高階関数を一階で定義する手法でこの問題の解決を試みた [15]。また Marlow & Wadler(1992) は高階に対応した方法を提案したが、その停止性は保証されていない [5]。Seidl & Sϕrensen(1997) は制約型高階制御フロー解析 (constraint-based higher-order control-flow analysis) を用い、高階に対応し停止性も保証された手法を提案している [10]。

これらの手法は、基本的にすべての関数呼び出しを展開し、次々に切り払いを試みるという戦略である。しかし、再帰関数を無限に展開させないために、展開した関数呼び出しの形をメモしておき、同じ形が再び現れたら新たな関数を定義するという複雑な処理が必

要であった。

これに対し、Gill ら (1993) はリストを扱う基本的な関数 $foldr$で関数の再帰構造を抽象化し、抽象化された関数だけを切り払いの対象とする手法を提案した [4]。この手法ではメモ化が不要であり、単純な変換規則を局所的に繰り返し適用することで切り払いを行う。Takano & Meijer(1995) は、この手法を抽象化することによって、除去される中間データ構造の種類をリストから一般のデータ構造へと拡張した手法を提案した [12]。実装の面では、これらの手法を基に尾上ら (1997) によりＨＹＬＯシステムと呼ばれるプログラム融合変換システムが開発、研究されている [6]。しかし、彼らの再帰関数の抽象化では、計算ステップ数削減に効果のあるリスト高速反転のような再帰関数は考慮されていない。

また、従来の研究の多くは、遅延評価 (lazy evaluation) の言語上でなされ、先行評価 (strict evaluation) の言語上では、ほとんど研究されていない。そのため、先行評価の言語における切り払いの有効性は確認されておらず、言語の簡約規則の違いという視点からの研究も進んでいない。

そこで本研究では、除去される中間データ構造がリストに限定されていた Gill らの手法を改良することにより、Takano & Meijer とは異なる手法によって、より多くのデータ構造が扱えるような切り払い方法を提案する。その際、プログラム変換の対象言語を値呼び (call-by-value) の言語とする。さらに実装を通じて本方法の評価を行なうとともに、その問題点についても検討する。

1.2 構成

第 2 章では必要となる用語の解説を行なう。続いて第 3 章では、データ生成過程を構成子に限定した部分計算的切り払いを提案し、実装を通して有効性を評価する。ここで提案する切り払いでは、リスト高速反転のような入力のデータ構造転換を伴う再帰関数も、特別な引数を導入することで取り扱うことが可能である。第 4 章では、入出力が異なる型にも対応した部分計算的切り払いの拡張を試み、問題点と解決策を検討する。最後に第 5 章で本研究を概観し、今後の研究方向について言及する。

第 2 章

準備

本章では、[15] に準じて本稿に必要となるプログラムに関する諸定義を与える。なお、本研究で用いる関数型言語の簡約は、値呼びとする。

2.1 項・関数定義・再帰的データ型

定義 2.1.1 (項) 変数記号、関数記号、構成子記号をそれぞれ v, c, f と表すとき、項 t を以下のように定義する。

$$
\begin{array}{rcll}
t & \stackrel{def}{=} & v & (\text{変数}) \\
 & | & c(t_1, t_2, \ldots, t_k) & (\text{構成子式}) \\
 & | & f(t_1, t_2, \ldots, t_k) & (\text{関数式}) \\
 & | & \mathbf{case}\ v_0\ \mathbf{of}\ p_1 \Rightarrow\ t_1 \mid p_2 \Rightarrow\ t_2 \mid \ldots \mid p_k \Rightarrow\ t_k & (\mathbf{case}\ \text{式}) \\
p & \stackrel{def}{=} & c(v_1, v_2, \ldots, v_k) & (\text{パターン})
\end{array}
$$

但し、**case** 式の $k \geq 1$ とし、$p_x \Rightarrow\ t_x$を分岐 (branch) と呼ぶ。

定義 2.1.2 (再帰的データ型) $'a$ を型変数 (type variable) とする。このとき以下のように定義されるデータ型を再帰的データ型と呼ぶ。

$$
\begin{array}{rcll}
'a\ data & \stackrel{def}{=} & c_1 & (\text{基底データ}) \\
 & | & c_2('a_1, \ldots, 'a_l, 'a\ data_1, \ldots, 'a\ data_m) & (\text{帰納データ})
\end{array}
$$

但し $m \geq 1$ とし、帰納データの引数$'a_1, \ldots, 'a_l, 'a\ data_1, \ldots, 'a\ data_m$の位置は任意。

例 2.1.1 (再帰的データ型) リストは、再帰的データ型として次のように表現できる。

$$\begin{aligned} 'a\ list\ &=\ \ Nil \\ &\quad \mid Cons('a,\ 'a\ list) \end{aligned}$$

定義 2.1.3 (関数定義、プログラム) 関数 f は、以下のように項 t で定義される。特に **case** 式で定義される場合、その定義を **case** 式による関数定義と呼ぶ。また、関数定義の集合をプログラムと呼ぶ。

$$f(v_1, v_2, \ldots, v_k) = t$$

例 2.1.2 (プログラム) リスト xs にリスト ys を連接する関数 $append$ は次のように定義される。

$$\begin{aligned} append(ys, xs) = \mathbf{case}\ xs\ \mathbf{of}& \\ Nil &\Rightarrow ys \\ \mid Cons(z, zs) &\Rightarrow Cons(z, append(ys, zs)) \end{aligned}$$

定義 2.1.4 (関数間の呼び出し) 関数 f が関数 g を呼び出すことを、以下のように再帰的に定義する。

f が g を呼び出すとは、

i) $(f(v_1, \ldots, v_k) = t)\ \wedge\ (g \in t)$、
または
ii) $\exists f' \in t$ (f' は g を呼び出す)。

但し、$g \in t$ は、t の中に関数 g が出現していることを表す。

定義 2.1.5 (分岐変数、格納変数) **case** 式による関数定義において、そのパターンが再帰的データ型である次のような関数定義を考える。但し、c_1 は基底データ、$c_2(\mathbf{v_d}, \mathbf{v_D})$ は帰納データ、$\mathbf{v_d} = (v_{d_1}, v_{d_2}, \ldots, v_{d_l})$ は帰納データにおけるデータの要素を表す変数列、$\mathbf{v_D} = (v_{D_1}, v_{D_2}, \ldots, v_{D_m})$ は帰納データにおける再帰的データを表す変数列とする。

$$\begin{aligned} f(v_1, v_2 \ldots, v_k, v_{k+1}, v_{k+2}) = \mathbf{case}\ v_{k+2}\ \mathbf{of}& \\ c_1 &\Rightarrow v_{k+1} \\ \mid c_2(\mathbf{v_d}, \mathbf{v_D}) &\Rightarrow t \end{aligned}$$

このとき v_{k+2}のように **case** 式の判定対象となる変数を分岐変数、v_{k+1}のようにその値が関数の結果に直接反映される変数を格納変数と呼ぶ。

定義左辺において、どれが分岐変数で格納変数であるかは、**case** 式、再帰的データ型を手がかりにして右辺を探索すれば即座に識別できる。よって、ここでは一般性を失うことなく、定義左辺における分岐変数は最後尾に、格納変数は分岐変数の直前に位置することとした。

第 3 章

部分計算的切り払い

本章では、まず Gill ら [4] が提案したリスト限定切り払いの特徴を述べる。次に、リスト限定切り払いを基にした部分計算的切り払いを提案する。そして、部分計算的切り払い手続きの停止性及び等価性を示し、実装を通じて本方法の評価を行なう。

3.1 リスト限定切り払い

リスト限定切り払いは Gill らによって提案された手法である [4]。彼らは $foldr$, $build$ という関数をリストに関する再帰関数の構造を抽象化するテンプレートと見なした。そして、$foldr$, $build$ で定義される関数だけを切り払いの変換対象とした。図 3.1 は、Gill らが示した $foldr$, $build$ を用いた切り払いの例である。$foldr$はリストを入力とする関数を定義し、$build$ はリストを出力とする関数を定義する。入力、出力ともにリストであるような関数は $build$ によって定義される。$build$ は、Nil, $Cons$ といった構成子をあたかもリスト生成演算子であるかのように扱う。そして、$build$ で表現された項が $foldr$の入力であるときに限り変換を行なうのである。つまり、関数をつなぐ中間データがリストのときに限り切り払いを行なう。重要な点は、$foldr$によって出力データ構造を規定する基底データと帰納データを生成する関数または構成子を抽出している点である。例えば、図 3.1 の sum の場合は、0 と $(+)$ である。これにより、変換後のデータ構造を確実に把握できる。

$<$ $foldr, build$ の定義と変換規則$>$

$$
\begin{aligned}
&foldr\ k\ z\ y\ =\ \textbf{case}\ y\ \textbf{of}\\
&\qquad\qquad Nil \Rightarrow z\\
&\qquad\quad \mid Cons(x, xs) \Rightarrow k\ x\ (foldr\ k\ z\ xs)\\
&build\ g\ =\ g\ Cons\ Nil\\
&foldr\ k\ z\ (build\ g)\ =\ g\ k\ z
\end{aligned}
$$

$<$ $foldr, build$ で定義可能な関数例$>$

$$
\begin{aligned}
&sum\ xs\ =\ foldr\ (+)\ 0\ xs && (\text{リスト } xs \text{ の総和})\\
&from'\ a\ b\ c\ n\ =\ \textbf{case}\ a > b\ \textbf{of}\\
&\qquad\qquad true \Rightarrow n\\
&\qquad\quad \mid false \Rightarrow c\ a\ (from'\ (a+1)\ b\ c\ n)\\
&from(a, b)\ =\ build\ (from'\ a\ b) && (a \text{ から } b \text{ のリスト})
\end{aligned}
$$

$<$変換例$>$

$$
\begin{aligned}
sum\ (from\ (a,\ b)) &= foldr\ (+)\ 0\ (build\ (from'\ a\ b))\\
&= from'\ a\ b\ (+)\ 0
\end{aligned}
$$

図 3.1: リスト限定切り払い

3.2 リストの高速反転と切り払い

リストを反転させる関数 $reverse$ を考える。

反転： $reverse(xs) = \mathbf{case}\ xs\ \mathbf{of}$

$$\begin{aligned} & Nil \Rightarrow Nil \\ \mid\ & Cons(y, ys) \Rightarrow append(Cons(y, Nil), reverse(ys)) \end{aligned}$$

$reverse$ は補助関数 $shunt$ を用いることにより、効率の向上を得られることが知られている [1]。

反転と連接： $shunt(j, xs) = \mathbf{case}\ xs\ \mathbf{of}$

$$\begin{aligned} & Nil \Rightarrow j \\ \mid\ & Cons(z, zs) \Rightarrow shunt(Cons(z, j), zs) \end{aligned}$$

反転： $rev(xs) = shunt(Nil, xs)$

$shunt$ は分岐変数 xs が帰納データの場合、格納変数 jに最終的な値を蓄積し自分自身を呼び出す。こうすれば反転と連接を一度に行なえるので計算効率が向上する。

データ構造を変化させる反転のような処理と、データの要素を操作する連接のような処理を一度に行なう $shunt$ のような再帰関数は、プログラムの効率を向上させる上で極めて有効な関数である。しかし、これまでの切り払いの手法ではほとんど取り扱われていない。

リスト限定切り払いでは、簡約が 1 段なされる度に最終結果のデータ構造の要素が生成される。これに対しリスト高速反転では、最終結果のデータ構造は引数で受け渡され、最終簡約を待たないとそのデータにアクセスできない。だが、リスト高速反転関数の定義では、受け渡される引数が確実にデータ構造の要素を生成しているのがわかる。したがって、このデータ生成過程に注目すれば、リスト限定切り払いが出力データ構造を抽出していたように、リスト高速反転のような 2 つの働きを行う関数の出力データ構造を抽出することで、切り払いが可能になると考える。

そこで本章ではリスト限定切り払いのアイデアを基に、ある条件の下ではリスト高速反転のような 2 つの働きを行う関数に対しても、切り払いが可能であることを示す。

3.3 逐次形関数・一括形関数

部分計算的切り払いでは、主として逐次形関数・一括形関数の 2 種類の再帰関数の合成を変換する。ここではこれら 2 種類の関数を定義する。

両者に共通する特徴は次の通りである。

- 同位置のデータには同一の処理。
- 関数が呼び出される度に、構成子によって最終結果の一部を形成する。
- 分岐変数と出力の型が一致。
- 関数 $f_0, f_1, \ldots, f_l$は関数 fを呼び出さない。

定義 3.3.1 (逐次形関数) 以下のように定義される関数を逐次形関数と呼ぶ。但し $\mathbf{v} = (v_1, v_2, \ldots, v_k)$ とする。

$$
\begin{aligned}
f(\mathbf{v}, v_x) = \mathbf{case}\ v_x\ \mathbf{of}& \\
& c_1 \Rightarrow f_0(\mathbf{v}, c_1) \\
& \mid c_2(\mathbf{v_d}, \mathbf{v_D}) \Rightarrow c_2(f_1(\mathbf{v}, v_{d_1}), \ldots, f_l(\mathbf{v}, v_{d_l}), f(\mathbf{v}, v_{D_1}), \ldots, f(\mathbf{v}, v_{D_m}))
\end{aligned}
$$

$S \stackrel{def}{=} \{ f \mid f$の定義は逐次形 $\}$

定義 3.3.2 (一括形関数) 以下のように定義される関数を一括形関数と呼ぶ。

$$
\begin{aligned}
f(\mathbf{v}, v_z, v_x) = \mathbf{case}\ v_x\ \mathbf{of}& \\
& c_1 \Rightarrow v_z \\
& \mid c_2(\mathbf{v_d}, v_{D_1}) \Rightarrow f(\mathbf{v}, c_2(f_1(\mathbf{v}, v_{d_1}), \ldots, f_l(\mathbf{v}, v_{d_l}), v_z), v_{D_1})
\end{aligned}
$$

$P \stackrel{def}{=} \{ f \mid f$の定義は一括形 $\}$

逐次形関数、一括形関数における入出力間の型は同一に制限してある。それは、一括形関数における型の制限をなくした場合、再帰呼出時の引数の操作についても制限をなくさなければ、有用な関数を表現できないからである。詳しくは第 4 章で述べることとし、ここでは以降の議論を簡単にするため、両関数について入出力間の型を制限した。

補題 3.3.1 (一括形関数の性質) 一括形関数は、分岐変数に帰納データが与えられた場合、その値が存在するなら帰納データである。

(証明) 一括形関数 fが次のように定義されていたとする。

$$
\begin{aligned}
f(\mathbf{v}, v_z, v_x) = \mathbf{case}\ v_x\ \mathbf{of}& \\
& c_1 \Rightarrow v_z \\
& \mid c_2(\mathbf{v_d}, v_{D_1}) \Rightarrow \overbrace{f(\mathbf{v}, \underbrace{c_2(f_1(\mathbf{v}, v_{d_1}), \ldots, f_l(\mathbf{v}, v_{d_l}), v_z)}_{\text{格納項 } B}, \underbrace{v_{D_1}}_{\text{分岐項 } C})}^{\text{帰納データ分岐右辺 A}}
\end{aligned}
$$

このとき帰納データ分岐右辺 A を考える。A は C の値が定まらない限り、fの定義による簡約ができない。しかし B は帰納データであることがわかっている。A の値が定まるとき、その値は最終簡約の直前の格納項そのものの値である。つまり、一度でも帰納データ分岐の処理がなされる $f(\mathbf{t}, t_z, t_x)$ の値は、帰納データとなる。よって、分岐変数 v_xに入力として帰納データが与えられた場合、すなわち $f(\mathbf{t}, t_z, c_2(\mathbf{t_d}, t_{D_1}))$ の値は、t_{D_1}が基底データか帰納データであるかに関わらず、帰納データである。 □

3.4 変換規則

リストを入出力とする逐次形関数は $foldr$を用いて定義可能であり、一括形関数はリスト高速反転のような再帰関数を表現できる。そこで、逐次形・一括形関数に注目して、プログラムを効率のよいプログラムへと変換することを目指す。

ここで提案する変換方法は、従来の切り払いと比較して次のような特長を持つ。

- 値呼び簡約の言語に対応。
- 入出力がリストに限定されない。
- リスト高速反転のような再帰関数の取り扱いが可能。

このような変換を部分計算的切り払いと名付ける。部分計算的切り払いは、複数個の関数記号を含む項を入力とし、切り払いされた項を出力する。以下では、プログラムを効率のよいプログラムに変換する規則について説明する。

部分計算的切り払いでは、他の切り払いの手法と同じく、任意の項が中間データを生成する項なら、その項を中間データを生成しない項へと変換する。このとき使用される 7 個の変換規則を図 3.2 に示す。各規則は左辺を右辺へと変換し、$[\![t]\!]$ は t を変換した結果を表す。但し、$\mathbf{t_i} = (t_{i_1}, t_{i_2}, \ldots, t_{i_k})$、$\mathbf{t_j} = (t_{j_1}, t_{j_2}, \ldots, t_{j_n})$ とする。変換規則 (3)〜(6) によ

(1) $[\![v]\!] = v$ (変数)

(2) $[\![c(t_1, \ldots, t_k)]\!] = c([\![t_1]\!], \ldots, [\![t_k]\!])$ (構成子式)

(3) $[\![f(\mathbf{t_i}, g(\mathbf{t_j}, t_q))]\!] = [\![h(\mathbf{t_i}, \mathbf{t_j}, t_q)]\!]$ (逐次・逐次)
where $f, g, h \in S$, h is a new defined function.

(4) $[\![f(\mathbf{t_i}, g(\mathbf{t_j}, t_r, t_q))]\!] = [\![h(\mathbf{t_i}, \mathbf{t_j}, f(\mathbf{t_i}, t_r), t_q)]\!]$ (逐次・一括)
where $f \in S$, $g, h \in P$, h is a new defined function.

(5) $[\![f(\mathbf{t_i}, t_r, g(\mathbf{t_j}, t_q))]\!] = [\![h(\mathbf{t_i}, \mathbf{t_j}, t_r, t_q)]\!]$ (一括・逐次)
where $f \in P$, $g \in S$, h is a new defined function.

(6) $[\![f(\mathbf{t_i}, t_{r1}, g(\mathbf{t_j}, t_{r2}, t_q))]\!] = [\![f(\mathbf{t_i}, h(\mathbf{t_i}, \mathbf{t_j}, t_{r1}, t_q), t_{r2})]\!]$ (一括・一括)
where $f, g \in P$, $h \in S$, h is a new defined function.

(7) $[\![f(t_1, \ldots, t_k)]\!] = f([\![t_1]\!], \ldots, [\![t_k]\!])$ (その他)
where (the head of $t_k \notin S \cup P$) $\vee$ ($f \notin S \cup P$).

図 3.2: 部分計算的切り払いの変換規則

る変換は、逐次形関数・一括形関数間において、中間データが生成される場合を網羅したものである。また、変換規則 (3)〜(6) は新関数の定義を伴う。この定義方法については、次節において述べる。

3.5 新関数導出過程

前節では変換規則を述べた。変換規則 (3)〜(6) では新たに関数を定義している。ここでは新関数の導出方法について述べる。但し $\mathbf{v_i} = (v_{t_{i_1}}, \ldots, v_{t_{i_k}})$、$\mathbf{v_j} = (v_{t_{j_1}}, \ldots, v_{t_{j_n}})$ とする。$c_2(\mathbf{v_e}, \mathbf{v_E})$ は $c_2(\mathbf{v_d}, \mathbf{v_D})$ を変数の名前替えしたものとする。

新関数導出の基本方針は次の 3 つである。これら基本方針に各場合個別の処理を加えることにより、新関数が導出される。

(1) 展開と適用: 内側の関数を展開し、各分岐の右辺を外側の関数に適用する。

(2) 簡約: 分岐項が構成子である項は、関数定義に基づき簡約する。

(3) たたみ込み: 変換対象の関数合成とマッチする項は、新関数を用いて書き換える。

3.5.1 変換規則 (3) 逐次・逐次

逐次形関数の分岐項に逐次形関数が現れる場合の変換規則について、ここでは説明する。逐次形関数 $f,\ g$が次のように定義されていたとする。このとき、$f(\mathbf{t_i}, g(\mathbf{t_j}, t_q))$ を新関数を用いた項 $h(\mathbf{t_i}, \mathbf{t_j}, t_q)$ へと変換させてもよいことを示す。

$$
\begin{aligned}
f(\mathbf{v_i}, v_{x1}) = \mathbf{case}\ v_{x1}\ \mathbf{of}& \\
& c_1 \Rightarrow f_0(\mathbf{v_i}, c_1) \\
& \mid c_2(\mathbf{v_e}, \mathbf{v_E}) \Rightarrow c_2(f_1(\mathbf{v_i}, v_{e_1}), \ldots, f_l(\mathbf{v_i}, v_{e_l}), f(\mathbf{v_i}, v_{E_1}), \ldots, f(\mathbf{v_i}, v_{E_m}))
\end{aligned}
$$

$$
\begin{aligned}
g(\mathbf{v_j}, v_{x2}) = \mathbf{case}\ v_{x2}\ \mathbf{of}& \\
& c_1 \Rightarrow g_0(\mathbf{v_j}, c_1) \\
& \mid c_2(\mathbf{v_d}, \mathbf{v_D}) \Rightarrow c_2(g_1(\mathbf{v_j}, v_{d_1}), \ldots, g_l(\mathbf{v_j}, v_{d_l}), g(\mathbf{v_j}, v_{D_1}), \ldots, g(\mathbf{v_j}, v_{D_m}))
\end{aligned}
$$

まず基本方針 1 に従い、$f(\mathbf{v_i}, g(\mathbf{v_j}, v_{x2}))$ について、gを展開し分岐右辺を fに適用する。

$$
\begin{aligned}
&\mathbf{case}\ v_{x2}\ \mathbf{of} \\
&\quad c_1 \Rightarrow f(\mathbf{v_i}, g_0(\mathbf{v_j}, c_1)) \\
&\mid c_2(\mathbf{v_d}, \mathbf{v_D}) \Rightarrow f(\mathbf{v_i}, c_2(g_1(\mathbf{v_j}, v_{d_1}), \ldots, g_l(\mathbf{v_j}, v_{d_l}), g(\mathbf{v_j}, v_{D_1}), \ldots, g(\mathbf{v_j}, v_{D_m})))
\end{aligned}
$$

分岐右辺に対し、基本方針 2 による簡約を行なう。

$$
\begin{aligned}
&\mathbf{case}\ v_{x2}\ \mathbf{of} \\
&\quad c_1 \Rightarrow f(\mathbf{v_i}, g_0(\mathbf{v_j}, c_1)) \\
&\mid c_2(\mathbf{v_d}, \mathbf{v_D}) \Rightarrow c_2(\ f_1(\mathbf{v_i}, g_1(\mathbf{v_j}, v_{d_1})), \\
&\qquad\qquad\qquad\qquad \vdots \\
&\qquad\qquad\qquad\quad f_l(\mathbf{v_i}, g_l(\mathbf{v_j}, v_{d_l})), \\
&\qquad\qquad\qquad\quad f(\mathbf{v_i}, g(\mathbf{v_j}, v_{D_1})), \\
&\qquad\qquad\qquad\qquad \vdots \\
&\qquad\qquad\qquad\quad f(\mathbf{v_i}, g(\mathbf{v_j}, v_{D_m})))
\end{aligned}
$$

基本方針 3 により変換前の項とマッチする $f(\mathbf{v_i}, g(\mathbf{v_j}, v_{D_m}))$ のような項は、新関数名 h を用いて $h(\mathbf{v_i}, \mathbf{v_j}, v_{D_m})$ と書き換える。h は上記 **case** 式を基に下記のように定義される。

$$
h(\mathbf{v_i}, \mathbf{v_j}, v_{x2})\ =\ \mathbf{case}\ v_{x2}\ \mathbf{of}
$$

$$
\begin{aligned}
& c_1 \Rightarrow f(\mathbf{v_i}, g_0(\mathbf{v_j}, c_1)) \\
& \mid c_2(\mathbf{v_d}, \mathbf{v_D}) \Rightarrow c_2(\ f_1(\mathbf{v_i}, g_1(\mathbf{v_j}, v_{d_1})), \\
& \qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
& \qquad\qquad\qquad\qquad\qquad\qquad f_l(\mathbf{v_i}, g_l(\mathbf{v_j}, v_{d_l})), \\
& \qquad\qquad\qquad\qquad\qquad\qquad h(\mathbf{v_i}, \mathbf{v_j}, v_{D_1}), \\
& \qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
& \qquad\qquad\qquad\qquad\qquad\qquad h(\mathbf{v_i}, \mathbf{v_j}, v_{D_m}))
\end{aligned}
$$

変換させたかった項 $f(\mathbf{t_i}, g(\mathbf{t_j}, t_q))$ は、この定義を用いて $h(\mathbf{t_i}, \mathbf{t_j}, t_q)$ へと変換される。

例 3.5.1 (逐次・逐次) 木の左右を交換する逐次形関数 $reflect$ を 2 度呼び出す項は次のように変換される。

木： $'a\ tree\ =\ \ Lf$
$\qquad\qquad\qquad\mid Br('a,\ 'a\ tree,\ 'a\ tree)$

逐次形関数： $reflect(v) = \mathbf{case}\ v\ \mathbf{of}$
$\qquad\qquad\qquad\qquad\qquad Lf \Rightarrow Lf$
$\qquad\qquad\qquad\qquad\mid Br(v1, v_{t1}, v_{t2}) \Rightarrow Br(v1, reflect(v_{t2}), reflect(v_{t1}))$

変換前： $reflect(reflect(v))$

変換後： $h(v)$

新関数： $h(v) = \mathbf{case}\ v\ \mathbf{of}$
$\qquad\qquad\qquad\qquad Lf \Rightarrow Lf$
$\qquad\qquad\qquad\mid Br(v1, v_{t1}, v_{t2}) \Rightarrow Br(v1, h(v_{t1}), h(v_{t2}))$

3.5.2 変換規則 (4) 逐次・一括

次に、逐次形関数の分岐項に一括形関数が現れる場合の変換規則について説明する。逐次形関数 f、一括形関数 gが次のように定義されていたとする。このとき、合成項 $f(\mathbf{t_i}, g(\mathbf{t_j}, t_r, t_q))$ が新関数を用いた項 $h(\mathbf{t_i}, \mathbf{t_j}, f(\mathbf{t_j}, t_r), t_q)$ へ変換できることを示す。

$$
\begin{aligned}
& f(\mathbf{v_i}, v_{x1}) = \mathbf{case}\ v_{x1}\ \mathbf{of} \\
& \qquad\qquad\qquad\qquad c_1 \Rightarrow f_0(\mathbf{v_i}, c_1) \\
& \qquad\qquad\qquad\mid c_2(\mathbf{v_e}, \mathbf{v_E}) \Rightarrow c_2(f_1(\mathbf{v_i}, v_{e_1}), \ldots, f_l(\mathbf{v_i}, v_{e_l}), f(\mathbf{v_i}, v_{E_1}), \ldots, f(\mathbf{v_i}, v_{E_m}))
\end{aligned}
$$

$$
\begin{aligned}
g(\mathbf{v_j}, v_{z2}, v_{x2}) \;=\; & \mathbf{case}\ v_{x2}\ \mathbf{of} \\
& \quad c_1 \Rightarrow v_{z2} \\
& \mid c_2(\mathbf{v_d}, v_{D_1}) \Rightarrow g(\mathbf{v_j}, c_2(g_1(\mathbf{v_j}, v_{d_1}), \ldots, g_l(\mathbf{v_j}, v_{d_l}), v_{z2}), v_{D_1})
\end{aligned}
$$

基本方針１に従い、$f(\mathbf{v_i}, g(\mathbf{v_j}, v_{z2}, v_{x2}))$ について、gを展開し分岐右辺を fに適用する。

$$
\begin{aligned}
& \mathbf{case}\ v_{x2}\ \mathbf{of} \\
& \quad c_1 \Rightarrow f(\mathbf{v_i}, v_{z2}) \\
& \mid c_2(\mathbf{v_d}, v_{D_1}) \Rightarrow \overbrace{f(\mathbf{v_i}, \underbrace{g(\mathbf{v_j}, c_2(g_1(\mathbf{v_j}, v_{d_1}), \ldots, g_l(\mathbf{v_j}, v_{d_l}), v_{z2}), v_{D_1})}_{\text{分岐項 B}})}^{\text{帰納データ分岐右辺 A}}
\end{aligned}
$$

このとき帰納データ分岐右辺 A の分岐項 B は、補題 3.3.1 より帰納データである。B の値が帰納データであるなら、A は fの定義を用いて簡約できる。ここで、次のように定義される一括形関数 h を考える。

$$
\begin{aligned}
h(\mathbf{v_i}, \mathbf{v_j}, v_s, v_{x2}) \;=\; & \mathbf{case}\ v_{x2}\ \mathbf{of} \\
& \quad c_1 \Rightarrow v_s \\
& \mid c_2(\mathbf{v_d}, v_{D_1}) \Rightarrow h(\ \mathbf{v_i}, \\
& \qquad\qquad \mathbf{v_j}, \\
& \qquad\qquad c_2(f_1(\mathbf{v_i}, g_1(\mathbf{v_j}, v_{d_1})), \ldots, f_l(\mathbf{v_i}, g_l(\mathbf{v_j}, v_{d_l})), v_s), \\
& \qquad\qquad v_{D_1})
\end{aligned}
$$

この関数は格納項 v_sを利用し、分岐項 v_{x2}のデータ構造を変えながら、データの要素各々に対して g, fの処理を行なう。変換規則の左辺 $f(\mathbf{t_i}, g(\mathbf{t_j}, t_r, t_q))$ を確認すると、t_rに対しては fの処理、t_qに対してはデータ構造を変えながらの gの処理と fの処理がなされる。したがって、t_rに対してあらかじめ fの処理を行った項を格納項、t_qを分岐項とした $h(\mathbf{t_i}, \mathbf{t_j}, f(\mathbf{t_i}, t_r), t_q)$ は、データ t_r,t_qを参照する機会が一度で済む。

このように、逐次形関数 f、一括形関数 gの合成項 $f(\mathbf{t_i}, g(\mathbf{t_j}, t_r, t_q))$ を、新たに定義される一括形関数 h を用いて $h(\mathbf{t_i}, \mathbf{t_j}, f(\mathbf{t_i}, t_r), t_q)$ へと変換する。

例 3.5.2 (逐次・一括) リストの要素に n を加算する逐次形関数 $plusN$ とリストを高速反転する一括形関数 $shunt$ を含む項 $plusN(n, shunt(ys, xs))$ は次のように変換される。

変換前： $plusN(n, shunt(ys, xs))$

変換後： $h(n, plusN(n, ys), xs)$

新関数： $h(n, ys, xs) = \mathbf{case}\ xs\ \mathbf{of}$

$$\begin{aligned} & Nil \Rightarrow ys \\ \mid\ & Cons(z, zs) \Rightarrow h(n, Cons(plus(n, z), ys), zs) \end{aligned}$$

3.5.3 変換規則 (5) 一括・逐次

ここでは、一括形関数の分岐項に逐次形関数が現れる場合の変換規則について説明する。一括形関数 f、逐次形関数 gが次のように定義されていたとする。このとき、合成項 $f(\mathbf{t_i}, t_r, g(\mathbf{t_j}, t_q))$ が新関数 h を用いた項 $h(\mathbf{t_i}, \mathbf{t_j}, t_r, t_q)$ へと変換できることを示す。

$$\begin{aligned} f(\mathbf{v_i}, v_{z1}, v_{x1}) \ =\ & \mathbf{case}\ v_{x1}\ \mathbf{of} \\ & c_1 \Rightarrow v_{z1} \\ \mid\ & c_2(\mathbf{v_e}, v_{E_1}) \Rightarrow f(\mathbf{v_i}, c_2(f_1(\mathbf{v_i}, v_{e_1}), \ldots, f_l(\mathbf{v_i}, v_{e_l}), v_{z1}), v_{E_1}) \end{aligned}$$

$$\begin{aligned} g(\mathbf{v_j}, v_{x2}) = & \mathbf{case}\ v_{x2}\ \mathbf{of} \\ & c_1 \Rightarrow g_0(\mathbf{v_j}, c_1) \\ \mid\ & c_2(\mathbf{v_d}, v_{D_1}) \Rightarrow c_2(g_1(\mathbf{v_j}, v_{d_1}), \ldots, g_l(\mathbf{v_j}, v_{d_l}), g(\mathbf{v_j}, v_{D_1})) \end{aligned}$$

基本方針１により、$f(\mathbf{v_i}, v_{z1}, g(\mathbf{v_j}, v_{x2}))$ について、gを展開し分岐右辺に fを適用する。

$$\begin{aligned} & \mathbf{case}\ v_{x2}\ \mathbf{of} \\ & \quad c_1 \Rightarrow f(\mathbf{v_i}, v_{z1}, g_0(\mathbf{v_j}, c_1)) \\ & \mid\ c_2(\mathbf{v_d}, v_{D_1}) \Rightarrow f(\mathbf{v_i}, v_{z1}, c_2(g_1(\mathbf{v_j}, v_{d_1}), \ldots, g_l(\mathbf{v_j}, v_{d_l}), g(\mathbf{v_j}, v_{D_1}))) \end{aligned}$$

続いて基本方針２、３に従い、新関数 h を得る。

$$\begin{aligned} h(\mathbf{v_i}, \mathbf{v_j}, v_{z1}, v_{x2}) \ =\ & \mathbf{case}\ v_{x2}\ \mathbf{of} \\ & c_1 \Rightarrow f(\mathbf{v_i}, v_{z1}, g_0(\mathbf{v_j}, c_1)) \\ \mid\ & c_2(\mathbf{v_d}, v_{D_1}) \Rightarrow h(\mathbf{v_i}, \\ & \qquad \mathbf{v_j}, \\ & \qquad c_2(f_1(\mathbf{v_i}, g_1(\mathbf{v_j}, v_{d_1})), \ldots, f_l(\mathbf{v_i}, g_l(\mathbf{v_j}, v_{d_l})), v_{z1}), \\ & \qquad v_{D_1}) \end{aligned}$$

この関数を利用し、一括形関数 f と逐次形関数 g の合成 $f(\mathbf{t_i}, t_r, g(\mathbf{t_j}, t_q))$ は $h(\mathbf{t_i}, \mathbf{t_j}, t_r, t_q)$ へと変換できる。

例 3.5.3 (一括・逐次) 一括形関数 $shunt$ と逐次形関数 $plusN$ を含む項は次のように変換される。

変換前： $shunt(ys, plusN(n, xs))$
変換後： $h(n, ys, xs)$
新関数： $h(n, ys, xs) = \mathbf{case}\ xs\ \mathbf{of}$

$$\begin{aligned} & Nil \Rightarrow ys \\ \mid\ & Cons(z, zs) \Rightarrow h(n, Cons(plus(n, z), ys), zs) \end{aligned}$$

3.5.4 変換規則 (6) 一括・一括

次に、一括形関数の分岐項に一括形関数が現れる場合の変換規則について説明する。一括形関数 $f,\ g$ の合成 $f(\mathbf{t_i}, t_{r1}, g(\mathbf{t_j}, t_{r2}, t_q))$ を新関数 h を含む項 $f(\mathbf{t_i}, h(\mathbf{t_i}, \mathbf{t_j}, t_{r1}, t_q), t_{r2})$ へと変換してよいことを示す。

$f,\ g$ 共にその分岐項のデータ構造を変化させる。変換前の項 $f(\mathbf{t_i}, t_{r1}, g(\mathbf{t_j}, t_{r2}, t_q))$ について、t_{r1}、t_{r2}、t_q が最終結果 R へどのように反映しているかを見る。t_{r1} はそのままの形で R の一部を形成する。t_{r2} は f のデータ構造変換及びデータ要素処理後、R の一部を形成する。t_q は g のデータ構造変換及びデータ要素処理、f のデータ構造変換及びデータ要素処理後、R の一部を形成する。すなわち、t_{r2} の処理には f があれば十分である。t_q についてはデータ構造変換が 2 回行われるので、t_q のデータ構造は入力時と等しくなる。したがって t_{r1} と t_q については、t_{r1} を格納項、t_q を分岐項とし、t_q に対し $f,\ g$ のデータ要素処理を行うような逐次形関数 h を定義すればよい。よって変換すべき合成項は、あらかじめ t_{r1} を格納項、t_q を分岐項とした $h(\mathbf{t_i}, \mathbf{t_j}, t_{r1}, t_q)$ を用意し、これを格納項とする項 $f(\mathbf{t_i}, h(\mathbf{t_i}, \mathbf{t_j}, t_{r1}, t_q), t_{r2})$ へと変換させればよい。新関数 h の定義を以下に示す。

$$\begin{aligned} h(\mathbf{v_i}, \mathbf{v_j}, v_s, v_x)\ =\ & \mathbf{case}\ v_x\ \mathbf{of} \\ & c_1 \Rightarrow v_s \\ \mid\ & c_2(\mathbf{v_d}, v_{D_1}) \Rightarrow c_2(f_1(\mathbf{v_i}, g_1(\mathbf{v_j}, v_{d_1})), \\ & \qquad\qquad \vdots \\ & \qquad f_l(\mathbf{v_i}, g_l(\mathbf{v_j}, v_{d_l})), \end{aligned}$$

$$h(\mathbf{v_i}, \mathbf{v_j}, v_s, v_{D_1}))$$

例 3.5.4 (一括・一括) 一括形関数 $shunt$ を 2 度呼び出す項は次のように変換される。

変換前： $shunt(zs, shunt(ys, xs))$
変換後： $shunt(h(zs, xs), ys)$
新関数： $h(ys, xs) = \mathbf{case}\ xs\ \mathbf{of}$

$$Nil \Rightarrow ys$$
$$\mid Cons(z, zs) \Rightarrow Cons(z, h(ys, zs))$$

3.6 停止性

部分計算的切り払い手続きの変換規則 (3)〜(6) では、変換後の項を再び変換するので、手続きの停止性が保証されていないように見える。しかし、実際には手続きの停止性が保証されることを以下で示す。

定理 3.6.1 (部分計算的切り払い手続きの停止性) 部分計算的切り払い手続きは停止する。

(証明) 項 t の構造に関する帰納法で示す。

＜場合 1 (変数)： $t = v$**＞** 変換規則 (1) より明らか。

＜場合 2 (構成子式)： $t = c(t_1, \ldots, t_k)$ **＞** 部分項 $t_1, \ldots, t_k$ の変換手続きがそれぞれ停止すると仮定する。$c(t_1, \ldots, t_k)$ は変換規則 (2) より $c([\![t_1]\!], \ldots, [\![t_k]\!])$ に変換される。仮定より $t_1, \ldots, t_k$ の変換手続きはそれぞれ停止するので、$c(t_1, \ldots, t_k)$ の変換手続きも確かに停止する。

＜場合 3 (関数式)： $t = f(t_1, \ldots, t_k)$ **＞** 部分項 $t_1, \ldots, t_k$ の変換手続きがそれぞれ停止すると仮定する。

＜場合 3a： (the head of $t_k \notin S \cup P$) $\vee$ ($f \notin S \cup P$) **＞** このとき $f(t_1, \ldots, t_k)$ は、変換規則 (7) より $f([\![t_1]\!], \ldots, [\![t_k]\!])$ に変換される。帰納法の仮定より $t_1, \ldots, t_k$ の変換手続きはそれぞれ停止するので、$f(t_1, \ldots, t_k)$ の変換手続きも確かに停止する。

＜場合 3b：$(f \in S \cup P) \wedge (\text{the head of } t_k \in S \cup P)$ **＞** 関数式 $f(t_1, \ldots, t_k)$ は、変換規則 (3)〜(6) により各規則の右辺 $[\![h(\ldots, t_q)]\!]$ あるいは $[\![f(\ldots, t_{r2})]\!]$ へと変換される。このとき t の変換過程は、以下に示す (A) 及び (B) の性質を持つ。

(A) 右辺の t_q あるいは t_{r2} が左辺の t_k の部分項であることは、容易に示せる。したがって、変換規則 (3)〜(6) が繰り返し適用される項と言えども、いずれ頭部が逐次形・一括形関数以外の分岐項を持つ関数式を得る。この場合、次に適用される変換規則は変換規則 (7) であり、右辺の各部分項に対する変換手続きを行なう。

(B) 右辺の各部分項は、左辺 $f(t_1, \ldots, t_k)$ より小さい。

よって、(A)、(B) 及び帰納法の仮定より、右辺の各部分項に対する変換手続きは停止するので、$f(t_1, \ldots, t_k)$ の変換手続きも停止する。

□

3.7 等価性

２つの項が存在し、任意の入力に対し双方の項の計算過程が停止するなら停止したときの値が等しく、一方の計算過程が停止しないならもう一方の計算過程も停止しない場合、２つの項は等価であるとする。ここでは、部分計算的切り払いにおいて、変換前後の項が等価であることを示す。

定理 3.7.1 (部分計算的切り払い手続きの等価性) 部分計算的切り払い手続きにおいて、変換規則の左辺と右辺は等価である。

(証明) 変換規則 (1)(2)(7) の場合は明らか。

＜変換規則 (3) 逐次・逐次の場合＞ $f(\mathbf{t_i}, g(\mathbf{t_j}, t_q)) = h(\mathbf{t_i}, \mathbf{t_j}, t_q)$ であることを示す。以下、関数 f, g, h の定義は 3.5 新関数導出過程に基づく。

$t_q = c_1$ の場合、(左辺)=(右辺) を示す。

$$
\begin{aligned}
(\text{左辺}) &= f(\mathbf{t_i}, g(\mathbf{t_j}, c_1)) \\
&= f(\mathbf{t_i}, g_0(\mathbf{t_j}, c_1) \\
(\text{右辺}) &= h(\mathbf{t_i}, \mathbf{t_j}, f(\mathbf{t_i}, t_r), c_1)
\end{aligned}
$$

$= f(\mathbf{t_i}, g_0(\mathbf{t_j}, c_1))$

よって (左辺)=(右辺)。

$t_q = c_2(\mathbf{t_d}, \mathbf{t_D})$ の場合、(左辺)=(右辺) を示す。

(左辺) $= f(\mathbf{t_i}, g(\mathbf{t_j}, c_2(\mathbf{t_d}, \mathbf{t_D})))$

$= f(\mathbf{t_i}, c_2(g_1(\mathbf{t_j}, t_{d_1}), \ldots, g_l(\mathbf{t_j}, t_{d_l}), g(\mathbf{t_j}, t_{D_1}), \ldots, g(\mathbf{t_j}, t_{D_m}))$

$= c_2(f_1(\mathbf{t_i}, g_1(\mathbf{t_j}, t_{d_1})), \ldots, f_l(\mathbf{t_i}, g_l(\mathbf{t_j}, t_{d_l})), f(\mathbf{t_i}, g(\mathbf{t_j}, t_{D_1})), \ldots, f(\mathbf{t_i}, g(\mathbf{t_j}, t_{D_m})))$

$= c_2(f_1(\mathbf{t_i}, g_1(\mathbf{t_j}, t_{d_1})), \ldots, f_l(\mathbf{t_i}, g_l(\mathbf{t_j}, t_{d_l})), h(\mathbf{t_i}, \mathbf{t_j}, t_{D_1}), \ldots, h(\mathbf{t_i}, \mathbf{t_j}, t_{D_m}))$

(右辺) $= h(\mathbf{t_i}, \mathbf{t_j}, c_2(\mathbf{t_d}, \mathbf{t_D}))$

$= c_2(f_1(\mathbf{t_i}, g_1(\mathbf{t_j}, t_{d_1})), \ldots, f_l(\mathbf{t_i}, g_l(\mathbf{t_j}, t_{d_l})), h(\mathbf{t_i}, \mathbf{t_j}, t_{D_1}), \ldots, h(\mathbf{t_i}, \mathbf{t_j}, t_{D_m}))$

よって (左辺)=(右辺)。

＜変換規則 (4) 逐次・一括の場合＞ $f(\mathbf{t_i}, g(\mathbf{t_j}, t_r, t_q)) = h(\mathbf{t_i}, \mathbf{t_j}, f(\mathbf{t_i}, t_r), t_q)$ であることを示す。

$t_q = c_1$の場合、(左辺)=(右辺) を示す。

(左辺) $= f(\mathbf{t_i}, g(\mathbf{t_j}, t_r, c_1))$

$= f(\mathbf{t_i}, t_r)$

(右辺) $= h(\mathbf{t_i}, \mathbf{t_j}, f(\mathbf{t_i}, r), c_1)$

$= f(\mathbf{t_i}, t_r)$

$t_q = c_2(\mathbf{t_d}, t_{D_1})$ の場合、(左辺)=(右辺) を示す。

(左辺) $= f(\mathbf{t_i}, g(\mathbf{t_j}, t_r, c_2(\mathbf{t_d}, t_{D_1})))$

$= f(\mathbf{t_i}, g(\mathbf{t_j}, c_2(g_1(\mathbf{t_j}, t_{d_1}), \ldots, g_l(\mathbf{t_j}, t_{d_l}), t_r), t_{D_1}))$

$= h(\mathbf{t_i}, \mathbf{t_j}, f(\mathbf{t_i}, c_2(g_1(\mathbf{t_j}, t_{d_1}), \ldots, g_l(\mathbf{t_j}, t_{d_l}), t_r)), t_{D_1})$

$= h(\mathbf{t_i}, \mathbf{t_j}, c_2(f_1(\mathbf{t_i}, g_1(\mathbf{t_j}, t_{d_1})), \ldots, f_l(\mathbf{t_i}, g_l(\mathbf{t_j}, t_{d_l})), f(\mathbf{t_i}, t_r))), t_{D_1})$

(右辺) $= h(\mathbf{t_i}, \mathbf{t_j}, f(\mathbf{t_i}, t_r), c_2(\mathbf{t_d}, t_{D_1}))$

$= h(\mathbf{t_i}, \mathbf{t_j}, c_2(f_1(\mathbf{t_i}, g_1(\mathbf{t_j}, t_{d_1})), \ldots, f_l(\mathbf{t_i}, g_l(\mathbf{t_j}, t_{d_l})), f(\mathbf{t_i}, t_r))), t_{D_1})$

よって (左辺)=(右辺)。

＜変換規則 (5) 一括・逐次の場合＞ $f(\mathbf{t_i}, t_r, g(\mathbf{t_j}, t_q)) = h(\mathbf{t_i}, \mathbf{t_j}, t_r, t_q)$ であることを示す。

$t_q = c_1$の場合、(左辺)=(右辺) を示す。

$$
\begin{aligned}
(\text{左辺}) &= f(\mathbf{t_i}, t_r, g(\mathbf{t_j}, c_1)) \\
&= f(\mathbf{t_i}, t_r, g_0(\mathbf{t_j}, c_1)) \\
(\text{右辺}) &= h(\mathbf{t_i}, \mathbf{t_j}, t_r, c_1) \\
&= f(\mathbf{t_i}, t_r, g_0(\mathbf{t_j}, c_1))
\end{aligned}
$$

よって (左辺)=(右辺)。

$t_q = c_2(\mathbf{t_d}, t_{D_1})$ の場合、(左辺)=(右辺) を示す。

$$
\begin{aligned}
(\text{左辺}) &= f(\mathbf{t_i}, t_r, g(\mathbf{t_j}, c_2(\mathbf{t_d}, t_{D_1}))) \\
&= f(\mathbf{t_i}, t_r, c_2(g_1(\mathbf{t_j}, t_{d_1}), \ldots, g_l(\mathbf{t_j}, t_{d_l}), g(\mathbf{t_j}, t_{D_1}))) \\
&= f(\mathbf{t_i}, c_2(f_1(\mathbf{t_i}, g_1(\mathbf{t_j}, t_{d_1})), \ldots, f_l(\mathbf{t_i}, g_l(\mathbf{t_j}, t_{d_l})), t_r), g(\mathbf{t_j}, t_{D_1})) \\
&= h(\mathbf{t_i}, \mathbf{t_j}, c_2(f_1(\mathbf{t_i}, g_1(\mathbf{t_j}, t_{d_1})), \ldots, f_l(\mathbf{t_i}, g_l(\mathbf{t_j}, t_{d_l})), t_r), t_{D_1}) \\
(\text{右辺}) &= h(\mathbf{t_i}, \mathbf{t_j}, t_r, c_2(\mathbf{t_d}, t_{D_1})) \\
&= h(\mathbf{t_i}, \mathbf{t_j}, c_2(f_1(\mathbf{t_i}, g_1(\mathbf{t_j}, t_{d_1})), \ldots, f_l(\mathbf{t_i}, g_l(\mathbf{t_j}, t_{d_l})), t_r), t_{D_1})
\end{aligned}
$$

よって (左辺)=(右辺)。

＜変換規則 (6) 一括・一括の場合＞ $f(\mathbf{t_i}, t_{r1}, g(\mathbf{t_j}, t_{r2}, t_q)) = f(\mathbf{t_i}, h(\mathbf{t_i}, \mathbf{t_j}, t_{r1}, t_q), t_{r2})$ であることを示す。

$t_q = c_1$の場合、(左辺)=(右辺) を示す。

$$
\begin{aligned}
(\text{左辺}) &= f(\mathbf{t_i}, t_{r1}, g(\mathbf{t_j}, t_{r2}, c_1)) \\
&= f(\mathbf{t_i}, t_{r1}, t_{r2}) \\
(\text{右辺}) &= f(\mathbf{t_i}, h(\mathbf{t_i}, \mathbf{t_j}, t_{r1}, c_1), t_{r2}) \\
&= f(\mathbf{t_i}, t_{r1}, t_{r2})
\end{aligned}
$$

よって (左辺)=(右辺)。

$t_q = c_2(\mathbf{t_d}, t_{D_1})$ の場合、(左辺)=(右辺) を示す。

$$
\begin{aligned}
(\text{左辺}) &= f(\mathbf{t_i}, t_{r1}, g(\mathbf{t_j}, t_{r2}, c_2(\mathbf{t_d}, t_{D_1}))) \\
&= f(\mathbf{t_i}, t_{r1}, g(\mathbf{t_j}, c_2(g_1(\mathbf{t_j}, t_{d_1}), \ldots, g_l(\mathbf{t_j}, t_{d_l}), t_{r2}), t_{D_1})) \\
&= f(\mathbf{t_i}, h(\mathbf{t_i}, \mathbf{t_j}, t_{r1}, t_{D_1}), c_2(g_1(\mathbf{t_j}, t_{d_1}), \ldots, g_l(\mathbf{t_j}, t_{d_l}), t_{r2})) \\
&= f(\mathbf{t_i}, c_2(f_1(\mathbf{t_i}, g_1(\mathbf{t_j}, t_{d_1})), \ldots, f_l(\mathbf{t_i}, g_l(\mathbf{t_j}, t_{d_l})), h(\mathbf{t_i}, \mathbf{t_j}, t_{r1}, t_{D_1})), t_{r2}) \\
(\text{右辺}) &= f(\mathbf{t_i}, h(\mathbf{t_i}, \mathbf{t_j}, t_{r1}, c_2(\mathbf{t_d}, t_{D_1})), t_{r2})
\end{aligned}
$$

$$= f(\mathbf{t_i}, c_2(f_1(\mathbf{t_i}, g_1(\mathbf{t_j}, t_{d_1})), \ldots, f_l(\mathbf{t_i}, g_l(\mathbf{t_j}, t_{d_l})), h(\mathbf{t_i}, \mathbf{t_j}, t_{r1}, t_{D_1})), t_{r2})$$

よって (左辺)=(右辺)。

□

3.8 効率性

切り払い手続きによるプログラム変換は、複数回行なっていたデータの参照が 1 回で済むため、一般に効率のよいプログラムを生成すると考えられている。しかし、変換後の項が線形 (linear) でなく、複数回現れる項の計算ステップ数が著しく多い場合には、効率が上がることは明らかとは言えない。ここでは、部分計算的切り払いを適用することにより、変換されるプログラムの効率がどの程度向上するかについて考察する。以下では、計算プロセスとしての簡約、すなわち計算ステップ数を基準として、効率を比較する。

ここでは、ML(Standard ML of New Jersey) を用い SS5(SunOS 4.1.4) 上で、部分計算的切り払い手続きを実装し、実際のプログラムでどの程度計算ステップ数を減少させることができるかを実験を通して考察する。

3.8.1 部分計算的切り払いの実装

図 3.3 に処理の流れを示す。本実装では、ML で記述されたプログラムを入力とし、出力も ML で記述されたプログラムとした。但し、プログラムにおいて使用される関数は、一階の関数に限定した。

図 3.3 において、関数定義を逐次形・一括形に分類する処理を除き、全て自動化できた。この関数定義の分類処理は、既に分類済みの定義を入力として与えることで代用した。また、プログラム変換時の入出力ユーティリティを考慮し、データ型、関数定義といった記述に労力を要する入力はファイル形式での入力を可能とした。さらに、変換後、即座に新関数定義を読み込み、実行させることを可能にするため、出力される新関数定義の集合もファイル形式で出力されるようにした。実装した部分計算的切り払い手続きの入出力を、以下に示す。

入力　1. データ型の定義ファイル

2. 逐次形関数の定義ファイル

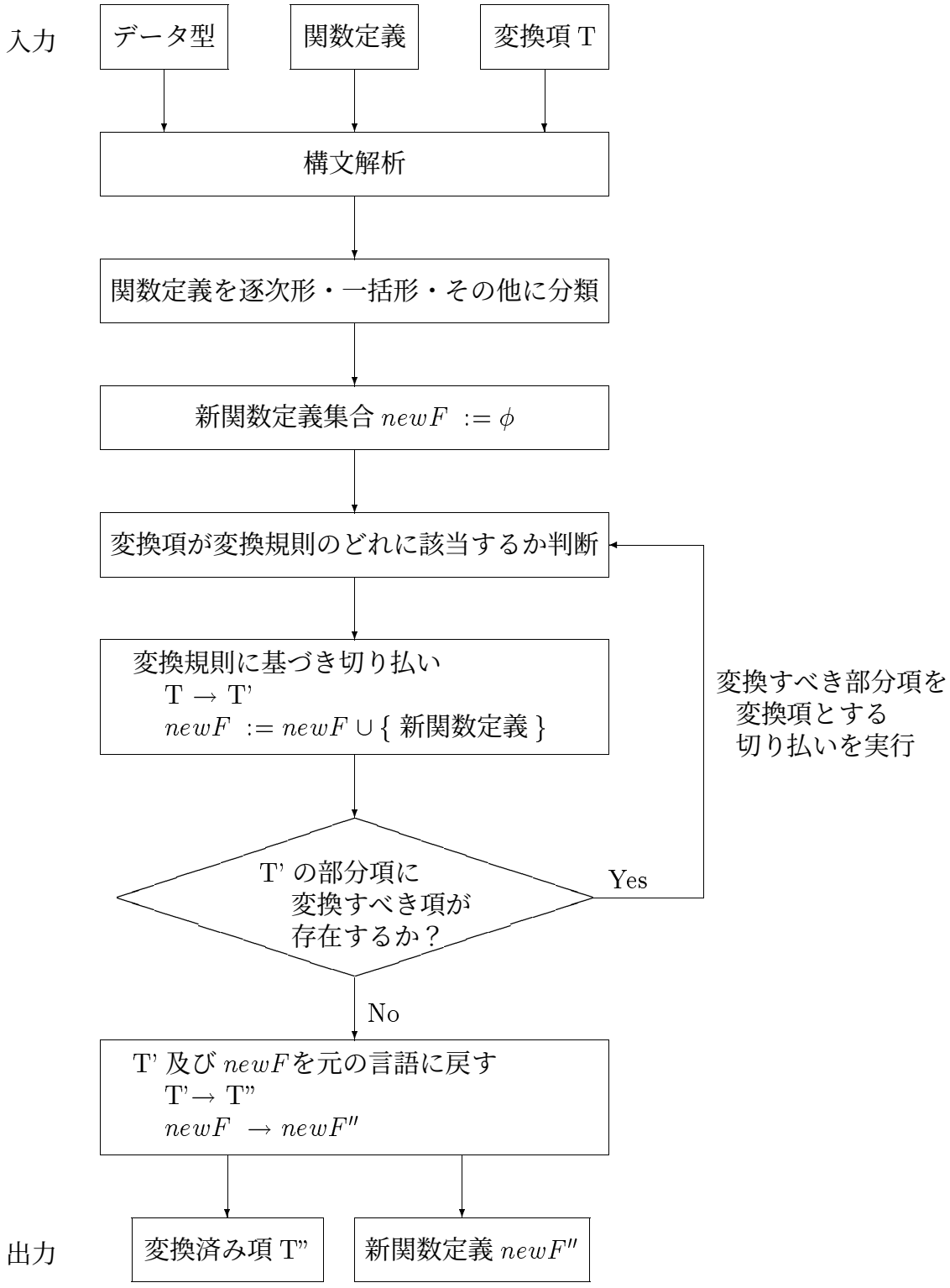

図 3.3: 部分計算的切り払いの流れ

3. 一括形関数の定義ファイル

4. 逐次形・一括形以外の関数の定義ファイル

5. 新関数定義の出力先ファイル

6. 変換対象項

出力　1. 変換済項 (新関数定義は入力 5 で指定した出力先ファイルに書き込まれる)

項 $f(t_1, \ldots, g(\ldots))$ が変換規則 (3)〜(6) のどれに該当するかの判断処理では、f, gが逐次形、一括形、逐次形・一括形以外の関数のどれに属するかが判断できればよい。本実装では、次のような探索手続きを採用した。逐次形定義の集合、一括形定義の集合、逐次形・一括形以外の関数定義の集合の順に、f, gを探索していく。その際、fがどこに属するかを判断してから gの所属を判断するという 2 段階の探索ではなく、f, gについて同時探索を行なった。具体的には、3 つの関数定義の集合を探索する際、1 つの関数定義に対し fがマッチするか、gがマッチするかの 2 つのマッチング処理を行なう。f, gのどちらもマッチしなければ、次の関数定義と関数名のマッチング処理を行なう。f, gのどちらかが関数定義の関数名とマッチしたら、f, gのもう一方に対しては未探索の関数定義の集合についてのみ探索する。これは、切り払いの基本となる概念、データ参照を 1 回にすることで効率の向上を図るというアイデアにヒントを得たものである。付録 A にプログラムリストを添付したので参照されたい。

さらに、部分計算的切り払いを実現する上で、変換前後で項の等価性を保証するため、次の処理が重要な役割を果たすことが判明した。

- 変数の名前替え
- 新関数名の取得

部分計算的切り払いでは、言わば 2 つの関数定義を融合する。このとき起こり得る問題が変数名の衝突である。双方の関数定義で、同一の変数名が用いられていた場合、変数名を考慮しないと変換前後で等価性が保証されない可能性がある。本実装では、呼び出す毎に新変数名を取得できる変数名取得関数 $getV$を用意した。 そして、2 つの関数が与えられた時点で、変数名の衝突を避けるため、双方の関数定義に対し、$getV$を用いた変数の名前替え処理を行なった。

表 3.1: 変換前後における計算ステップ数比較

変換項	入力	計算ステップ	
		変換前	変換後
逐次・逐次：reflect(reflect(x))	x=深さ 15 の二分木	131070	65535
逐次・一括：plusN(n,shunt(ys,xs))	n=5000, xs=ys=$[\underbrace{1,\ldots,1}_{10000}]$	70002	60002
一括・逐次：shunt(ys,plusN(n,xs))	n=5000, xs=ys=$[\underbrace{1,\ldots,1}_{10000}]$	40002	30001
一括・一括：shunt(zs,shunt(ys,xs))	zs=ys=xs=$[\underbrace{1,\ldots,1}_{10000}]$	30002	20002

新関数も、生成される毎に異なる名称を関数に与えなければ、変数の衝突と同様の問題が生じる。ここでも $getV$ のように、呼び出す毎に新関数名を取得できる関数名取得関数 $getF$ を用意し新関数生成時に使用した。

なお、本実装では、既に使用されている変数名あるいは関数名をある場所に記憶し、そこに記憶されていない名称を新変数名あるいは新関数名として使用するという方法はとらなかった。実装を簡単にするため、新変数名は v'、新関数名は FUN' で始まるものとし、生成順に 1, 2, 3, ... のように番号を割り当て、v'10、FUN'23 のように生成した。

3.8.2 実装の評価

第 3 章 3.5 新関数導出過程で使用した例について、計算機上で部分計算的切り払いを行ない、変換前と変換後の計算ステップ数を比較した。表 3.1 に実験結果を示す。また、付録 B に計算機上での実行例を添付したので参照されたい。

但し、ML には計算ステップ数を測定する機構が装備されていない。そこで、C や Pascal の変数に相当する参照型 (reference type) を計算ステップ数測定用のカウンターとして利用することとした。プログラムにおいて、計算が 1 ステップ行なわれる箇所にカウンターを 1 加算するよう記述しておき、プログラムの実行終了後のカウンター値を計算ステップ数と見なした。

入力データについては、木 (x) は深さ 15、自然数 (n) は 5000、リスト (xs, ys, zs) は要素が 1 で長さが 10000 のリストとした。

表 3.1 において、いずれの場合も、内側の分岐項のデータ要素数に相当する計算ステップ数が減少し、効率の改善が見られ、確かに切り払いがなされたことが確認できる。

逐次・逐次に関しては、木の要素全てを参照する回数が２回から１回に減少したため、ちょうど木の要素に相当する計算ステップ数の減少が見られる。逐次・一括、一括・逐次、一括・一括においても、リスト xs の参照回数が２回から１回へと減少したので、リスト xs の長さに相当する計算ステップ数の減少が見られる。

これらの事実から、部分計算的切り払いによる効率の向上が認められた。但し、効率はデータの要素数に左右されるので、一般的な改善率を予測することはできず、入力データに大きく依存する。

3.9 問題点

部分計算的切り払いは、$foldr$, $build$ を用いて関数を抽象化する必要がなく、実装の面からもその有効性が確認できた。

しかし、部分計算的切り払いは入出力の型が一致していなければならず、リストから自然数といった型の変化を伴う sum のような関数には対応していない。したがって、リスト限定切り払いでは扱えなかったリスト高速反転関数を扱えるようになったものの、型の変化を伴う関数は扱えなくなってしまった。

そこで次章では、型変化を伴う関数にも対応した部分計算的切り払いの拡張を試みる。

第 4 章

部分計算的切り払いの拡張

本章では、部分計算的切り払いの 2 段階の拡張について議論する。第 1 段階では、入出力間での型の一致制限をなくすこと、第 2 段階では、構成子に限定されていた出力を関数を用いた出力に拡張することを目指す。さらに、予測される問題点と解決策を検討する。

4.1 異なる構成子における拡張

入出力間の型を制限しなかった場合、入出力間のデータ構造は一般に異なる。したがって、再帰呼び出し時の引数操作を制限したままでは、入力のデータ要素数が出力のそれよりも少ない場合、どのデータ要素も等しくなるような出力しか得られない。

例 4.1.1 (入力のデータ要素数が出力よりも少ない関数) 自然数の帰納データ $Succ(v)$ は 1 引数、木の帰納データ $Br(v, v1, v2)$ は 3 引数はであり、自然数のデータ要素数は木のデータ要素数と比較して少ない。

関数 $comptree$ は、自然数 n を入力とし、$k = 1$ の場合ノードが 1 から 2^n、深さ n の木を出力する。定義は [7] を参照した。

自然数から木

$$
\begin{aligned}
comptree(k, n) = \mathbf{case}\ n\ \mathbf{of}& \\
& Zero \Rightarrow Lf \\
& \mid Succ(m) \Rightarrow Br(k,\ comptree(2 * k, m), \\
& \qquad\qquad\qquad\qquad comptree(2 * k + 1, m))
\end{aligned}
$$

$comptree$ は再帰的に自分自身を呼び出す際、引数 kを操作することによりノードのラベルを変化させて木を出力する。もし、再帰呼び出し時に引数 kを操作できなければ、全てのノードが kであるような木を出力することになる。

このように、入出力間の型を制限しない場合、引数の自由な操作をも制限しない方が、取り扱える関数集合を拡大できる。

$comptree$ のように引数の自由な操作を許可した関数を逐次形として再定義する。同様に一括形も引数操作の制限をなくす。このとき部分計算的切り払いの３つの基本方針により、次の例のような逐次・逐次、一括・逐次の変換が可能である。

例 4.1.2 (逐次・逐次 (構成子拡張版)) 関数 $preord$ は木を入力とし、左分木から順にリストを作る。$comptree$ を分岐項とする $preord$ は次のように変換される。$preord$ の定義は [7] を参照した。

木からリスト (逐次)：

$$
\begin{aligned}
&preord(vs, x) = \mathbf{case}\ x\ \mathbf{of}\\
&\qquad\qquad Lf \Rightarrow vs\\
&\qquad\quad \mid Br(v, t1, t2) \Rightarrow Cons(v, preord(preord(vs, t2), t1))
\end{aligned}
$$

変換前： $preord(vs, comptree(k, n))$

変換後： $h(vs, k, n)$

新関数： $h(vs, k, n) = \mathbf{case}\ n\ \mathbf{of}$

$$
\begin{aligned}
&\qquad\qquad Zero \Rightarrow vs\\
&\qquad\quad \mid Succ(m) \Rightarrow Cons(k,\\
&\qquad\qquad\qquad\qquad h(h(vs, k*2+1, m),\\
&\qquad\qquad\qquad\qquad\ \ k*2,\\
&\qquad\qquad\qquad\qquad\ \ m)
\end{aligned}
$$

例 4.1.3 (一括・逐次 (構成子拡張版)) 関数 $inord$ は木を入力とし、左から順にノードをリストにする。$inord$ の定義は [7] を参照した。

木からリスト (一括)：

$$
\begin{aligned}
&inord(vs, x) = \mathbf{case}\ x\ \mathbf{of}\\
&\qquad\qquad Lf \Rightarrow vs
\end{aligned}
$$

$$\mid Br(v, t1, t2) \Rightarrow inord(Cons(v, inord(vs, t2)), t1)$$

変換前：$inord(vs, comptree(k, n))$

変換後：$h(vs, k, n)$

新関数：$h(vs, k, n) = \textbf{case}\ n\ \textbf{of}$

$$\begin{aligned} & Zero \Rightarrow vs \\ \mid & Succ(m) \Rightarrow h(Cons(k, h(vs, 2 * k + 1, m)), \\ & \qquad\qquad 2 * k, \\ & \qquad\qquad m) \end{aligned}$$

しかし、一括形が分岐項に現れる場合、入出力間のデータ構造が異なるので注意が必要である。この場合、一括形関数 gの入力時点で上部にあったデータは出力時には下部へと移動する。この結果を入力とする逐次形あるいは一括形関数 fを考える。fは入力データの上部から処理を行なう。fが引数の自由な操作を伴う関数である場合、gの結果を待たずして f の処理は行なえない。なぜなら、データの要素個々に対する処理情報は引数によって継承されるため、部分計算的切り払いでは考慮せずに済んだデータの処理順序も考慮する必要があるからである。部分計算的切り払いでは、データの要素に対する処理は均一であったから、処理順序は重要な問題ではなかった。しかし、引数の自由な操作を想定すると、引数によるデータ要素処理情報の受渡しは重要な意味を持つ。この処理順序を伴うデータ要素処理情報を得られれば、異なる構成子においても切り払いは成功する。

4.2 関数における拡張

関数を用いた出力であっても、部分計算的切り払いの基本方針にしたがえば切り払いが成功する例がある。

例 4.2.1 (逐次・逐次 (関数拡張版)) 関数 sum の定義において、帰納データ分岐右辺の頭部は関数 $plus$ であるが、$sum(plusN(n, xs))$ の切り払いは成功する。

リストの総和 (逐次)：

$sum(xs) = \textbf{case}\ xs\ \textbf{of}$

$$Nil \Rightarrow Zero$$

$$\mid Cons(y, ys) \Rightarrow plus(y, sum(ys))$$

変換前： $sum(plusN(n, xs))$
変換後： $h(n, xs)$
新関数： $h(n, xs) = \mathbf{case}\ n\ \mathbf{of}$

$$Nil \Rightarrow Zero$$
$$\mid Cons(z, zs) \Rightarrow plus(plus(n, z), h(n, zs))$$

しかし、同じ逐次形どうしの合成であっても、次のように切り払いがなされていない例を見ることができる。

例 4.2.2 (逐次・逐次 (関数拡張版)) 関数 $reverse$ の定義において、帰納データ分岐右辺の頭部は関数 $append$ である。このとき $length(reverse(xs))$ の切り払いは不完全である。

リストの長さ (逐次)：
$length(xs) = \mathbf{case}\ xs\ \mathbf{of}$

$$Nil \Rightarrow Zero$$
$$\mid Cons(y, ys) \Rightarrow Succ(length(ys))$$

変換前： $length(reverse(xs))$
変換後： $h(xs)$
新関数： $h(xs) = \mathbf{case}\ xs\ \mathbf{of}$

$$Nil \Rightarrow Zero$$
$$\mid Cons(z, zs) \Rightarrow length(append(Cons(z, Nil), reverse(zs)))$$

次に、中間データの除去が完全な場合と不完全な場合を比較する。中間データの除去が完全な場合は、内側に位置する関数は構成子により明示的にデータを生成している。構成子によるデータ生成であれば、部分計算的切り払いと同じく先々の処理を見通すことができるので、中間データの除去に成功する。これに対し、中間データの除去が不完全な場合は、内側に位置する関数は関数によるデータ生成を採っている。したがって、最外関数のみが関数によるデータ生成ならば、切り払いは成功するように思えるが、次の例では切り払いは不完全である。

例 4.2.3 (逐次・逐次 (関数拡張版)) $plusN(n, preord(vs, x))$ は $preord$ のデータ生成は構成子によるものであるが、切り払いが不完全である。

変換前：$plusN(n, preord(vs, x))$

変換後：$h(n, vs, x)$

新関数：$h(n, vs, x) = \mathbf{case}\ x\ \mathbf{of}$

$$\begin{aligned} & Lf \Rightarrow plusN(n, vs) \\ \mid\ & Br(v, t1, t2) \Rightarrow Cons(plus(n, v), h(n, preord(vs, t2), t1)) \end{aligned}$$

部分計算的切り払いでは基本方針の他に、それぞれの場合に応じた処理抽出あるいは処理の組み換えにより、切り払いを可能にした。部分計算的切り払いでは逐次・逐次の場合、そのような特別な処理は必要なかった。しかし、拡張版では対策を講じなければうまく変換することができない。

切り払いがうまくいかなかった $plusN(n, preord(vs, x))$ の例では、$preord$ に切り払いを困難にする原因が隠されている。$preord$ は帰納データ分岐右辺で自分自身を２度呼び出している。このことは切り払いを行なう際、単に２つの関数定義を１つにまとめるだけでなく、自己呼び出しの数に応じた工夫が求められていることを示す。現に $plusN(n, preord(vs, x))$ の例では、入力の右分木は切り払いされていないが、左分木に対しては確実に切り払いが行なわれている。

このように、部分計算的切り払いを拡張するためには、少なくとも次の情報に基づく新たな処理が必要である。

- データ生成は構成子によるか関数によるか。
- 自己呼び出しはどこでどのようにおこなわれているか。
- データの要素に対する処理は均一か。

第 5 章

まとめ

本研究では $foldr$, $build$ の精神を継承しながら、$foldr$, $build$ によってプログラムを抽象化しない切り払いとして、部分計算的切り払いという新しい変換方法を提案した。さらにデータ構造の転換を伴う関数も、ある条件の下ではこれまでの関数と併せて切り払いが可能であることを示した。また、ML 上でこの変換手法を実装し、実験を通して、部分計算的切り払いの有効性の確認を行なった。

しかし、関数定義が構成子によってデータ生成過程を明示的に記述されていない場合は、切り払いが困難となる。今後の課題としては、このデータ生成過程の関数の問題とデータ構造の転換を伴う関数の問題があげられる。

データ生成過程の関数の問題については、データ生成過程が明示的でない関数の場合も切り払いが行える例があることから、どのような関数ならデータ生成過程に現れてもよいかについて研究する価値がある。

また、本研究ではデータ構造の転換を伴う関数について、リスト以外の型についてはあまり想定しなかった。だが、引数を変化、継承することによる再帰関数処理の研究がより一般的な型においてもなされるなら、様々な型の間におけるデータ構造の転換を伴う型変化関数の切り払いが可能になり、切り払いはさらに有用なものとなる。

高階関数についての考察も不十分であった。本研究を開始した当初、高階関数を切り払いで扱うことは困難と思われた。だが、逐次形・一括形関数の定義が、分岐変数以外の引数として関数を型とする引数をとっても、部分計算的切り払いの枠組に影響はない。こうした関数も扱える事例をいくつか発見し、それらに共通する性質を手がかりにすれば、高階関数に対応した部分計算的切り払いの拡張に貢献できる。

これらの課題が克服されれば、部分計算的切り払いは $foldr$, $build$ で扱える関数をカバーできる上、データ構造の転換を伴う関数についても扱えるようになる。さらに、記述したプログラムを抽象化することなく、そのままの形でプログラム変換機構により、より効率の良いプログラムへと自動的に変換することが可能になる。この方向での研究は、今後の興味深い課題である。

従来は、遅延評価の言語上での研究が多く、値呼び簡約などの先行評価の言語上での研究は、ほとんどなかった。部分計算的切り払いは、値呼び簡約の言語上での切り払いである。そのため、従来の切り払いの変換規則に含まれるような遅延評価を前提とした簡約は、先行評価の言語上では行なえないことが、本研究を進めて行く過程で明らかになった。本研究では、簡約規則の違いが切り払い手法に与える影響を解明するには至らなかった。言語の簡約規則の違いという視点からの切り払い研究は、ほとんど行なわれていないので、この方向での研究も興味深い課題である。

謝辞

本研究及び学生生活全般に渡って、多大なる御指導を頂いた外山芳人教授に謹んで感謝致します。また、本研究に関する議論の場を与えて下さった外山研究室の皆さんに心から感謝します。最後に、様々な形で学生生活を支えてくれた家族に感謝します。

参考文献

[1] Bird,R. & Wadler,P., "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)

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

[3] Chin,W.-N., "Safe fusion of functional expressions II: Further improvements", Journal of Functional Programming 4(4), pp.515-555, October 1994.

[4] Gill,A. & Launchbury,J. & Peyton Jones,S.L., "A Short Cut to Deforestation", FPCA '93, pp.223-232.

[5] Marlow,S. & Wadler,P., "Deforestation for Higher-Order Functions", Proc. 5th Annual Glasgow Workshop on Functional Programming, pp.154-165, July 1992.

[6] 尾上能之, 胡振江, 武市正人, "HYLO システムによるプログラム融合変換の実現", 日本ソフトウェア科学会第 14 回大会論文集, pp.447-480, 1997.

[7] Paulson,L.C., "ML for the Working Programmer"2nd edition, Cambridge University Press, 1993.

[8] Pettorossi,A. & Proietti,M., "Rules and Strategies for Transforming Functional and Logic Programs", ACM Compt. Surveys 28, pp.360-414, 1996.

[9] Seidl,H., "Integer Constraints to Stop Deforestation", Proc.ESOP '96, LNCS 1058, pp.326-340, 1996.

[10] Seidl,H. & Sϕrensen,M.H., "Constraints to Stop Higher-Order Deforestation", POPL '97, 1997.

[11] Sørensen,M.H., "A Grammar-based Data-flow Analysis to Stop Deforestation", Proc.CAAP '94, LNCS 787, pp.335-351, 1994.

[12] Takano,A. & Meijer,E., "Shortcut Deforestation in Calculational Form", Proc.FPCA '95, pp.306-313, 1995.

[13] Toyama,Y., "How to prove equivalence of term rewriting systems without induction", Theoretical Computer Science 90, pp.369-390, 1991.

[14] Ullman,J.D., "Elements of ML Programming", Prentice-Hall, 1993.

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

付録 A

変換規則判断処理

部分計算的切り払い手続きにおける規則判断処理を以下に記す。この処理は関数を頭部とする項に対し、変換規則 (3)〜(7) のいずれを適用すべきかの判断を決定する。

```
(*******************************
項と関数定義
摘要          :Var 変数
               Con 構成子
               Fun 関数
               Cas case
               Def 関数定義
*******************************)
type name = string;
datatype expr = Var of name
              | Con of (name * (expr list))
              | Fun of (name * (expr list))
              | Cas of expr * ((expr * expr) list)
              | Def of (abt * expr)
and abt = Abt of (name * (expr list))

(*******************************
関数名：read_Compo
入力   ：関数を頭部とする項
```

出力　：(頭部 (1〜3), 頭部関数定義, 分岐項の頭部 (1〜3,8,9), 分岐項の頭部関数定義)

摘要　： 1 逐次形関数

2 一括形関数

3 その他関数

8 構成子

9 変数

参照型 Fdf1,Fdf2,Fdf3 は逐次形、一括形、その他の関数定義のリスト

**********)

```
exception read_Compo_Error;
exception read_Compo_Error1;
exception read_Compo_Error2;
exception read_Compo_Error3;
exception read_Compo_Error4;
exception read_Compo_Error5;
fun read_Compo(cmp) =
    let
        val dmyDf = Def(Abt("dummy",[]),Var("dummy"));
        (* 関数定義 *)
        val F1 = !Fdf1;
        val F2 = !Fdf2;
        val F3 = !Fdf3;
        (* 外・内を同時に探索 *)
        fun ptChk1(_,_,[],[],[]) = raise read_Compo_Error1
          | ptChk1(Fun(nm1,ls1),Fun(nm2,ls2),[],[],
                              (dfn as Def(Abt(nm3,_),_))::xs) =
            if nm1=nm3 then
                ((true,3,dfn),(false,0,dmyDf),([],[],dfn::xs))
            else
                if nm2=nm3 then
```

```
          ((false,0,dmyDf),(true,3,dfn),([],[],xs))
      else ptChk1(Fun(nm1,ls1),Fun(nm2,ls2),[],[],xs)
| ptChk1(Fun(nm1,ls1),Fun(nm2,ls2),[],
                (dfn as Def(Abt(nm3,_),_))::xs,D3) =
  if nm1=nm3 then
      ((true,2,dfn),(false,0,dmyDf),([],dfn::xs,D3))
  else
      if nm2=nm3 then
          ((false,0,dmyDf),(true,2,dfn),([],xs,D3))
      else ptChk1(Fun(nm1,ls1),Fun(nm2,ls2),[],xs,D3)
| ptChk1(Fun(nm1,ls1),Fun(nm2,ls2),
             (dfn as Def(Abt(nm3,_),_))::xs,D2,D3) =
  if nm1=nm3 then
      ((true,1,dfn),(false,0,dmyDf),(dfn::xs,D2,D3))
  else
      if nm2=nm3 then
          ((false,0,dmyDf),(true,1,dfn),(xs,D2,D3))
      else ptChk1(Fun(nm1,ls1),Fun(nm2,ls2),xs,D2,D3)
(* 変数 *)
| ptChk1(_,Var(x),_,_,_,_) =
      ((false,9,dmyDf),(false,9,Var(x)),([],[],[]))
(* 構成子 *)
| ptChk1(Fun(nm1,ls1),Con(nm2,ls2),[],[],
                   (dfn as Def(Abt(nm3,_),_))::xs) =
  if nm1=nm3 then
      ((false,3,dfn),(false,8,Con(nm2,ls2)),([],[],dfn::xs))
  else ptChk1(Fun(nm1,ls1),Con(nm2,ls2),[],[],xs)
| ptChk1(Fun(nm1,ls1),Con(nm2,ls2),[],
                (dfn as Def(Abt(nm3,_),_))::xs,D3) =
  if nm1=nm3 then
```

```
        ((false,2,dfn),(false,8,Con(nm2,ls2)),([],dfn::xs,D3))
    else ptChk1(Fun(nm1,ls1),Con(nm2,ls2),[],xs,D3)
  | ptChk1(Fun(nm1,ls1),Con(nm2,ls2),
                (dfn as Def(Abt(nm3,_),_))::xs,D2,D3) =
    if nm1=nm3 then
        ((false,1,dfn),(false,8,Con(nm2,ls2)),(dfn::xs,D2,D3))
    else ptChk1(Fun(nm1,ls1),Con(nm2,ls2),xs,D2,D3)
  | ptChk1(_,_,_,_,_) = raise read_Compo_Error2;

(* 外・内どちらかを探索 *)
fun ptChk2(_,[],[],[]) = raise read_Compo_Error3
  | ptChk2(Fun(nm1,ls1),[],[],(dfn as Def(Abt(nm3,_),_))::xs) =
    if nm1=nm3 then (3,dfn)
    else ptChk2(Fun(nm1,ls1),[],[],xs)
  | ptChk2(Fun(nm1,ls1),[],(dfn as Def(Abt(nm3,_),_))::xs,D3) =
    if nm1=nm3 then (2,dfn)
    else ptChk2(Fun(nm1,ls1),[],xs,D3)
  | ptChk2(Fun(nm1,ls1),(dfn as Def(Abt(nm3,_),_))::xs,D2,D3) =
    if nm1=nm3 then (1,dfn)
    else ptChk2(Fun(nm1,ls1),xs,D2,D3)
  | ptChk2(_,_,_,_) = raise read_Compo_Error4;

(* 分岐項取得 *)
fun infGet(Fun(_,vars)) = hd(rev vars)
  | infGet(_) = raise read_Compo_Error5;
val inF = infGet cmp;

(* 外・内の関数を同時探索 *)
val ((b1,n1,d1),(b2,n2,d2),(D1,D2,D3)) =
ptChk1(cmp,inF,F1,F2,F3)
```

```
in
    if b1 then
        let val (n3,d3) = ptChk2(inF,D1,D2,D3)
        in  (n1,d1,n3,d3)
        end
    else
        if b2 then
            let val (n3,d3) = ptChk2(cmp,D1,D2,D3)
            in  (n3,d3,n2,d2)
            end
        (* 変数 or 構成子 *)
        else (n1,d1,n2,d2)
end;
```

付録 B

部分計算的切り払いの実行例

部分計算的切り払いの計算機上の実行例を以下に記す。

```
(*******************************************************
関数名：DEFO_file
入力　：Data.sml(データ型の定義ファイル)
        P1.sml　(逐次形関数の定義ファイル)
        P2.sml　(一括形関数の定義ファイル)
        P3.sml　(逐次形・一括形以外の関数の定義ファイル)
        Rst.sml (新関数定義の出力先ファイル)
        object　(変換対象項)
出力　：変換済項
摘要　：・object に対し部分計算的切り払いを実行
        ・新関数定義を Rst.sml に書き込む
*******************************************************)

(*** 逐次・逐次：reflect(reflect(x)) の実行例 ***)
- val object = "reflect(reflect(x))";
val object = "reflect(reflect(x))" : string
- DEFO_file(("Data.sml","P1.sml","P2.sml","P3.sml",object),"Rst.sml");
val it = "FUN'1(x)" : string
-
(* 出力ファイル (Rst.sml) に書き込まれた新関数定義
```

```
fun FUN'1(v'5) =
        case v'5 of
          Lf => Lf
        | Br(v'6,v'7,v'8) => Br(v'6,FUN'1(v'7),FUN'1(v'8));
*)
(*** 逐次・一括：plusN(n,shunt(ys,xs)) の実行例 ***)
- val object = "plusN(n,shunt(ys,xs))";
val object = "plusN(n,shunt(ys,xs))" : string
- DEFO_file(("Data.sml","P1.sml","P2.sml","P3.sml",object),"Rst.sml");
val it = "FUN'2(n,plusN(n,ys),xs)" : string
-
(* 出力ファイル (Rst.sml) に書き込まれた新関数定義
fun FUN'2(v'9,v'13,v'14) =
        case v'14 of
          Nil => v'13
        | Cns(v'15,v'16) => FUN'2(v'9,Cns(plus(v'9,v'15),v'13),v'16);
*)
(*** 一括・逐次：shunt(ys,plusN(n,xs)) の実行例 ***)
- val object = "shunt(ys,plusN(n,xs))";
val object = "shunt(ys,plusN(n,xs))" : string
- DEFO_file(("Data.sml","P1.sml","P2.sml","P3.sml",object),"Rst.sml");
val it = "FUN'6(n,ys,xs)" : string
-
(* 出力ファイル (Rst.sml) に書き込まれた新関数定義
fun FUN'6(v'45,v'41,v'46) =
        case v'46 of
          Nil => v'41
        | Cns(v'43,v'44) => FUN'6(v'45,Cns(plus(v'45,v'43),v'41),v'44);
*)
(*** 一括・一括：shunt(zs,shunt(ys,xs)) の実行例 ***)
```

```
- val object = "shunt(zs,shunt(ys,xs))";
val object = "shunt(zs,shunt(ys,xs))" : string
- DEFO_file(("Data.sml","P1.sml","P2.sml","P3.sml",object),"Rst.sml");
val it = "shunt(FUN'8(zs,xs),ys)" : string
-
(* 出力ファイル (Rst.sml) に書き込まれた新関数定義
fun FUN'8(v'57,v'62) =
        case v'62 of
          Nil => v'57
        | Cns(v'63,v'64) => Cns(v'63,FUN'8(v'57,v'64));
*)
```