属性文法上の Shortcut Deforestation

Shortcut Deforestation on Attribute Grammar

森畑明昌, 筧一彦, 胡振江, 武市正人

Akimasa Morihata†, Kazuhiko Kakehi†, Zhenjiang Hu†‡, Masato Takeichi†

† 東京大学大学院情報理工学系研究科

Graduate School of Information Science and Technology

‡ 科学技術振興機構 さきがけ研究 21

PRESTO 21, Japan Science and Technology Agency

関数型言語では、関数を組み合わせることによってプログラムを構成するが、このとき結果に直接関係しない中間データ構造が発生し、効率を下げる原因となる。融合変換は、関数を融合することでこのような中間データ構造を削除するプログラム変換であり、運算や属性文法の枠組みを用いて多くの研究がなされている。しかし、両枠組みの研究が独立に行われ、互いの関係また両枠組みの融合に関する研究は少ない。本研究では、今まで運算手法の上で議論されてきた融合変換の 1 つである「shortcut deforestation」を属性文法の枠組みで記述し、その変換規則を与える。そして既成の融合変換との比較を行う。

1 はじめに

関数型言語では、小さな関数の組み合わせで大きな関数を構築する際、結果には直接必要でない中間データ構造が生じることがよくある。例えば、リストの各要素を 2 乗して足し合わせる関数 `sumsquare` は、値を 2 乗する関数 `square`, リストの各要素を足し合わせる関数 `sum`, そして高階関数 `map` を用いて、

```
sumsquare x = sum (map square x)
```

と記述することができるが、`map square x` の結果が中間データとして生じてしまい、効率を下げる原因となってしまう。これを避けるためには

```
sumsquare [] = 0
sumsquare (a:x) = square a + sumsquare x
```

という 1 つの関数にまとめてしまえばよい。このような中間データ構造の生成を抑制するプログラム変換を “deforestation” と呼ぶ。

Deforestation の方法は Wadler [Wad88] によって初めて示され、その後も多くの研究がある。それらの中には、運算手法を用いてプログラム変換のルールを記述する研究 [GLPJ93] [SF93] [LS95] [TM95] もあり、また属性文法を用いて関数を融合変換することで deforestation を行う研究 [CDPR99] [KGF01] もある。しかし両枠組みは独立に研究が行われ、互いの関係や融合についての研究はわずかであった。

本研究では、deforestation の 1 種である “shortcut deforestation” を取り上げる。shortcut deforestation は主に運算手法の上で議論されてきており、属性文法上での議論はほとんどなかった。本研究では、属性文法の枠組みでの shortcut deforestation の変換規則を与え、既成の deforestation の手法と比較する。

2 背景

2.1 Shortcut deforestation

Shortcut deforestation は Gill ら [GLPJ93] の示した deforestation の手法であり、以下の規則からなる。

```
g :: ∀β.(A → β → β) → β → β
⇒   foldr k z (build g) = g k z
 where build g = g Cons Nil
```

これは直感的には、データの構成を司る “`build`” に対して、構成時と同じ順序で各要素へ演算を行う関数 “`foldr`” を行うことは、すなわちデータの構成を行わずに演算を行うのと同じである、という意味である。Shortcut deforestation には関数の unfold-fold を用いて行う deforestation に比べて実装が容易であるという利点がある。Gill らの提案した shortcut deforestation で取り扱えるデータ型は *List* のみであったが、その後多くの研究によって拡張され [LS95] [TM95]、多くのデータ型に対して適用可能となっている。

例として `sum (map square x)` に shortcut deforestation を適用する。上記変換規則を適用するため `sum` を `foldr`, `(map square)` を `build` で書き直す。

```
sum x = foldr (+) 0 x
map square x = build g
  where g = (\c n->foldr (c.square) n x)
```

$map\ f\ x =$
$x \to L$ { **result** $= L.ms$ };
$L \to Cons\ a\ L_1$ { $L.ms = Cons\ (f\ a)\ L_1.ms$ };
$L \to Nil$ { $L.ms = Nil$ }

$reverse\ x\ =$
$x \to L$ { **result** $= L.rs \quad L.ri = Nil$ };
$L \to Cons\ a\ L_1$ { $L_1.ri = Cons\ a\ L.ri$
$L.rs = L_1.rs$ };
$L \to Nil$ { $L.rs = L.ri$ }

$flat\ x\ h\ =$
$x \to T$ { **result** $= T.fs \quad T.fi = h$ };
$T \to Node\ T_1\ T_2$ { $T.fs = T_1.fs$
$T_1.fi = T_2.fs$
$T_2.fi = T.fi$ };
$T \to Leaf\ a$ { $T.fs = Cons\ a\ T.fi$ }

図 1: *map*, *reverse*, *flat* の定義

g の型は $\forall\beta.(A \to \beta \to \beta) \to \beta \to \beta$ であり、shortcut deforestation の規則を適用できる。これにより以下のプログラムが得られる。

```
sumsquare x = foldr ((+).square) 0 x
```

`sumsquare x` は、確かに `sum (map square x)` と同じ値を返し、しかも中間データ構造を生成しない。

2.2 属性文法

属性文法とは、文脈自由文法をもとに構成される構文木の各節点に属性と属性の値を決定するための意味規則を付加したものである。例として図 1 に *map*, *reverse*, *flat* を属性文法で記述したものを示す。本論文で取り扱う属性文法の構文文法は図 2 に示す。

属性は自分の子節点へ値を伝播させる継承属性と親節点へ値を伝播させる合成属性の 2 つに分けられる。この 3 つの関数で用いられている属性のうち *ms*, *rs*, *fs* は合成属性であり、*ri*, *fi* は継承属性である。

開始記号の生成規則にあたる $x \to L$ の部分は “top case” と呼ばれ、構文解析の開始時、または終了時に必要となる意味規則を定義する。*map* の例では x を開始記号とし L を構文木として属性を計算し、結果として属性 *ms* の値を返すことが定義されている。$L \to Cons\ a\ L_1$, $L \to Nil$ の部分では非終端記号 L に対する生成規則とそれに伴う意味規則を表現している。*map* の例では「構文木 L が Cons a L_1 なら L の属性 *ms* として L_1 の属性 *ms* に $f\ a$ を *Cons* したものを返せ。L が *Nil* なら L の属性 *ms* として *Nil* を返せ」となる。これは、関数型言語におけるパターンマッチングと再帰呼び出しの構造に符合する。

agf ::= $f\ e_1 \cdots e_n\ =\ top;\ rule_1;\ \cdots;\ rule_n$
top ::= $prod \to$ **result** $= e\ (e = e)*$
$rule$ ::= $prod \to (e = e)*$
$prod$ ::= $(t_0 \to C\ t_1 \cdots t_n)$
e ::= $t \mid C\ t_1 \cdots t_n \mid f\ t_1 \cdots t_n$ (**where** agf)?
t ::= $c \mid v \mid t.a$

図 2: 対象とする属性文法

属性の計算を行う上で重要な性質に、「単一代入」と呼ばれるものがある。これは、変数への代入 (束縛) が 1 回に限られるという規則で、同じ文脈の変数が同じ値を持つことを保証するものである。すなわち、関数型言語における参照透明性に対応する。

2.3 属性文法上の deforestation

属性文法上での deforestation の手法の殆どは descriptional composition [GG84] を用いている。これは合成関数 $f \circ g$ を g の属性上で f の属性が計算されると考えることで融合変換する手法である。例えば $map\ (reverse\ x)$ であれば、*reverse* の属性 *rs*, *ri* 上で *map* の属性 *ms* が計算されるので、生成規則 $L \to Cons\ a\ L_1$ に対する意味規則は、*map* の属性 *ms* の計算規則により以下のように変換される。

$\{(L.ri).ms = (Cons\ a\ L_1.ri).ms$
$(L.rs).ms = (L_1.rs).ms$ };
$\Downarrow$
$\{L.ri.ms = Cons\ (f\ a)\ L_1.ri.ms$
$L.rs.ms = L_1.rs.ms$ };

他の意味規則も同様に変換することで両関数は融合され、中間データ構造を生成しない関数を表現する属性文法が得られる。

3 文法定義

図 2 に本論文での変換対象となる関数を表現する属性文法の構文規則を示す。ただし関数定義は非循環であるとする。また関数名、変数名、属性名等は適切な名前空間の元で唯一な名前を持つとする。

本論文では、属性文法の意味規則を “関係” と捉える。これはすなわち $e = e$ という文は「左辺と右辺の値が等しい」という状態を規定していると考えるのである。属性文法においては、代入は単一代入で

あるため、左辺の値も右辺の値も変化しない。よってこれを関係をとらえることに問題はない。

4 属性文法上の shortcut deforestation

4.1 変換規則

関数型言語では式が重要な役割を持つため、式を表現する “関数” による変換規則によって shortcut deforestation は達成される。しかし、属性文法では式の値だけでは十分でない。その式がどの属性に関係づけられるかも重要である。なぜなら、異なる生成規則に対する意味規則であっても、属性名は値の依存関係を示すよう適切に定められなければならないためである。そのため “関係” による変換規則でなければ変換を適切に表現できない。

我々の手法では関係を述語を用いて記述する。例えば *Cons*, *Nil* に対応する述語はそれぞれ

$Cons'\ a\ x\ t\ =\ (t = Cons\ a\ x)$

$Nil'\ t\ =\ (t = Nil)$

となる。つまり *Cons'* は変数 *a*, *x*, *t* の関係を表現している。また、述語に対して型を定義する。これは例えば *Cons'* であれば $a \to List\ a \to List\ a \to R$、*Nil'* であれば $List\ a \to R$ というものである。“関係” および “述語” を用いることにより、属性文法上での shortcut deforestation が表現できる。

定理 1 (属性文法上の shortcut deforestation)
関数 h が 2 関数 f, g の合成

$h\ x_1\ x_2 \cdots\ x_m\ =\ f\ (g\ x_1\ x_2 \cdots\ x_m)$

で定義され、かつ f および g が以下の属性文法で定義されているとする。

$f\ v_1\ v_2 \cdots\ v_n =$

$v_1 \to L \quad \{\ top_f\ \};$

$L \to Cons\ a\ L_1 \quad \{\ p_1\ a\ L_1\ L\ \};$

$L \to Nil \quad \{\ p_2\ L\ \}$

$g\ v_1\ v_2 \cdots\ v_m = body_g[Cons'/c,\ Nil'/n]$

$$body_g \equiv \left(\begin{array}{c} v_k \to pat \quad \{\ \mathbf{result} = e_r \quad top_g\ \}; \\ rule_1;\ \cdots;\ rule_l \end{array} \right)$$

このとき、関数 $g'\ c\ n = body_g$ の型が

$g'\ ::\ \forall\beta.\ (A \to \beta \to \beta \to R) \to (\beta \to R) \to \beta$

となるならば、h は以下になる。

$$\left(\begin{array}{c} v_k \to pat \quad \{\ (top_f)[e_r/v_1, v_{m+i-1}/v_i] \\ top_g\ \}; \\ rule_1;\ \cdots;\ rule_l \end{array} \right) [p_1/c, p_2/n]$$

述語によって g のコンストラクタが適切に抽象化されれば、すなわち型に関する条件を満たせば、抽象化されたコンストラクタを f の意味規則で置換し top case を併合することで融合変換が達成される。定理 1 では *List* に対する変換規則しか与えていないが、同様の規則によって多くのデータ型に対する shortcut deforestation を行うことができる。

4.2 変換例

上記の手法を用いた shortcut deforestation の適用例として *reverse*(*flat x h*) を考える。*reverse* および *flat* の属性文法での表現は図 1 に示してある。

reverse(flat x h) を shortcut deforestation する場合、*flat* の定義に現れる *Cons* を抽象化するだけでは十分でない。なぜなら、累積変数 h も関数 *flat* の返値の中に直接現れるからである。そのため h 中の *Cons* および *Nil* も抽象化しなければならない。以上をふまえ、h を *copy Cons' Nil' h* によって書き換え、コンストラクタを抽象化したものが図 3 である。これは型の条件を満たすため、定理 1 を適用し shortcut deforestation を行うことができる。

Shotcut deforestation の結果に対して、述語部分を定義で置き換えたものが図 4 である。確かに、累積変数まで含めて中間データ構造を生成しない属性文法の関数に融合変換されている。

変換結果の関数 *revflat* を評価するには多少の注意が必要である。属性 *fi.fs* 等は値であり、問題なく取り扱うことができるが、生成規則 $T \to Node\ T_1\ T_2$ の意味規則に登場する *T.fi*, *T.fs* は値ではない。属性、具体的には *rs* または *ri* をとって初めて値となる。そのため $e = e$ を「左辺への右辺値の代入」と捉えると値の評価順序が定まらない。前述のように、これは “関係 ”と捉えるべきである。すなわち、例えば文 $T_2.fi = T.fi$ は、「どのような属性 a に対しても $T_2.fi.a = T.fi.a$ である」という関係である。関係と捉えることによりこの関数は評価できる。

4.3 Descriptional Composition との比較

Shortcut deforestation は normalization algorithm [SF93] の具体化の 1 つであり、normalization algorithm と descriptional compositon の結果は等価である [DPRJ96]。そのため定理 1 は descripitional composition と殆ど同じ結果を生む。しかし両者の表現力には差がある。下記の関数 *foo* を考える。

$$
\begin{array}{l}
reverse\ x\ = \\
\quad x \to L \qquad\quad \{\ \mathbf{result} = L.rs \quad L.ri = Nil\ \} \\
\quad L \to Cons\ a\ L_1 \quad \{\ p_1\ L\ a\ L_1\ \} \\
\quad L \to Nil \qquad\quad \{\ p_2\ L\ \} \\
p_1\ x\ a\ t\ =\ (x.ri = Cons\ a\ t.ri \quad t.rs = x.rs) \\
p_2\ t\ =\ (t.rs = t.ri)
\end{array}
$$

$$
\begin{array}{l}
flat\ x\ h\ = \\
\left(\begin{array}{lll}
x \to T & \{\ \mathbf{result} = T.fs & \\
 & \quad T.fi = copy\ h & \\
 & \qquad \mathbf{where} \quad copy\ x\ = & \\
 & \qquad\qquad x \to L & \{\ \mathbf{result} = L.cs\ \} \\
 & \qquad\qquad L \to Cons\ a\ L_1 & \{\ c\ a\ L_1.cs\ L.cs\ \} \\
 & \qquad\qquad L \to Nil & \{\ n\ L.cs\ \}\ \} \\
T \to Node\ T_1\ T_2 & \{\ T.fs = T_1.fs \quad T_1.fi = T_2.fs \quad T_2.fi = T.fi\ \} & \\
T \to Leaf\ a & \{\ c\ a\ T.fi\ T.fs\ \} &
\end{array}\right)\ [Cons'/c,\ Nil'/n]
\end{array}
$$

図 3: コンストラクタの抽象化

$$
\begin{array}{lll}
revflat\ x\ h\ = & & \\
x \to T & \{\ \mathbf{result} = T.fs.rs \quad T.fs.ri = Nil & \\
 & \quad T.fi = copy'\ h & \\
 & \qquad \mathbf{where} \quad copy'\ x\ = & \\
 & \qquad\qquad x \to L & \{\ \mathbf{result} = L.cs\ \} \\
 & \qquad\qquad L \to Cons\ a\ L_1 & \{\ L.cs.rs = L_1.cs.rs \quad L_1.cs.ri = Cons\ a\ L.cs.ri\ \} \\
 & \qquad\qquad L \to Nil & \{\ L.cs.rs = L.cs.ri\ \}\ \} \\
T \to Node\ T_1\ T_2 & \{\ T.fs = T_1.fs \quad T_1.fi = T_2.fs \quad T_2.fi = T.fi\ \} & \\
T \to Leaf\ a & \{\ T.fs.rs = T.fi.rs \quad T.fi.ri = Cons\ a\ T.fs.ri\ \} &
\end{array}
$$

図 4: Shortcut deforestation の結果

$$
\begin{array}{lll}
foo\ x\ = & & \\
\quad x \to L & \{\ \mathbf{result} = L.s \quad L.i = Nil\ \}; & \\
\quad L \to Cons\ a\ L_1 & \{\ L_1.i = Cons\ a\ L.i & \\
 & \quad L.s = Cons\ L_1.i\ L_1.s\ \}; & \\
\quad L \to Nil & \{\ L.s = Nil\ \} &
\end{array}
$$

関数 *foo* の意味規則中には 2 つの *Cons* が登場する。$L_1.i = Cons\ a\ L.i$ の *Cons* が結果となる *List* の要素を生成するのに対し、$L.s = Cons\ L_1.i\ L_1.s$ の *Cons* は結果となる *List* を生成する。つまり、この関数 *foo* を中間データ構造の生産者とする deforestation を行う場合には、2 つの *Cons* は区別しなければならない。しかし descriptional composition では 2 つの *Cons* を区別できない。人手によって明示的に区別したとしても、その区別が正しいことを示す枠組みはない。これに対し、Shortcut deforestation では抽象化時に別々の抽象化を行うことで 2 つの *Cons* を区別でき、かつ抽象化の正しさも型情報から得ることができる。これは descriptional composition に比べて shortcut deforestation の方がより多くのプログラム中の情報を用いて変換を行っていることを意味する。換言すれば、descriptional composition の手法でもより多くの情報を取り扱うことでさらに表現力の高い融合変換が可能になると考えられる。

4.4 既知の問題点

我々の定義した属性文法では、特定のデータ構造を消費することで値を計算する関数以外の関数を記述することはできない。例えば、整数を引数としその値に等しい長さの *List* を生成する関数を記述することはできない。似た理由から、*List* の組を受け取り組の *List* を返す関数 *zip* は、2 つの引数両方を同

時に deforest することはできない。

また定理 1 の証明はできていない。変換の要旨は既成の shortcut deforestation とほぼ同じであるが、形式の整った証明はできていない。

5 おわりに

本研究では、属性文法の意味規則を “関係” と理解し述語の置換を行うことによって、属性文法上での shortcut deforestation が可能となることを示し、その変換規則を与えた。また、この手法は累積変数を持つ関数に対しても shortcut deforestation が可能であることを示した。さらに、この変換規則が descriptional composition の取り扱うことのできない関数を変換可能であることを示した。

本論文では属性値の評価戦略については何も述べなかった。これは、属性文法による関数表記はプログラムの意味を表現するのみであり、評価方法は規定していないと考えるためである。もちろん適切な評価戦略を用いれば冗長な中間データ構造を生成せずに評価を行うことができる。具体的には必要呼びを用いて値の依存関係を解決する方法やトポロジカルソートを用いて属性評価の順序を得る方法等がある。

今後の研究としては、まず属性文法の構文文法の拡張が考えられる。特定のデータ構造を持たない、例えば整数のようなデータ型に対しても同様の変換を行えるか否かについては興味のあるところである。また、本研究ではコンストラクタの抽象化は人手によって適切に与えられることを仮定していた。しかし抽象化を自動的に行うことができれば、コンパイル時の非常に有効な最適化となりうることが期待される。この点も今後の課題である。

最後に。Shortcut deforestation は関数型言語の枠組みで用いられる手法であるため、特に明示はしなかったが、本研究でも属性文法によって表現された関数は関数型言語の関数であった。しかし本研究の変換には、参照透明性以外の関数型言語特有の要素は用いていない。そのため shortcut deforestation は関数型言語以外の枠組みでも用いることができるのではないかと考えられる。実際、意味規則を関係と捉えるのは論理型言語では自然な理解である。関数型言語で培われた様々な手法の他の言語枠組みでの応用も興味深い研究課題の 1 つである。

参考文献

[CDPR99] L. Correnson, E. Duris, D. Parigot, and G. Roussel. Declarative program transformation: A deforestation case-study. In *Principles and Practice of Declarative Programming*, pp. 360–377, 1999.

[DPRJ96] E. Duris, D. Parigot, G. Roussel, and M. Jourdan. Attribute grammars and folds: Generic control operators. Technical Report 2957, 1996.

[GG84] H. Ganzinger and R. Giegerich. Attribute coupled grammars. In *Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, pp. 157–170. ACM Press, 1984.

[GLPJ93] A. Gill, J. Launchbury, and S. L. P. Jones. A short cut to deforestation. In *Conference on Functional Programming Languages and Computer Architecture*, pp. 223–232, 1993.

[KGF01] K. Kakehi, R. Gluck, and Y. Futamura. On deforesting parameters of accumulating maps, 2001.

[LS95] J. Launchbury and T. Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *Conf. Record 7th ACM SIGPLAN/SIGARCH Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA'95, La Jolla, San Diego, CA, USA, 25–28 June 1995*, pp. 314–323. ACM Press, New York, 1995.

[SF93] T. Sheard and L. Fegaras. A fold for all seasons. In *Proceedings 6th ACM SIGPLAN/SIGARCH Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA'93, Copenhagen, Denmark, 9–11 June 1993*, pp. 233–242. ACM Press, New York, 1993.

[TM95] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Conf. Record 7th ACM SIGPLAN/SIGARCH Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA'95, La Jolla, San Diego, CA, USA, 25–28 June 1995*, pp. 306–313. ACM Press, New York, 1995.

[Wad88] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *ESOP '88. European Symposium on Programming, Nancy, France, 1988 (Lecture Notes in Computer Science, vol. 300)*, pp. 344–358. Berlin: Springer-Verlag, 1988.