データ構造に着目したコンパイラ最適化の形式化

佐々木　真弓　　浅井　健一
お茶の水女子大学大学院人間文化研究科
mayumi@pllab.is.ocha.ac.jp
asai@is.ocha.ac.jp

概 要

再帰的データ構造に対する一般的な変換手法として fold によるコンストラクタの置き換えを挙げることができる。しかし、この手法をコンパイラの構文木に対しても用いるには既存のパラメータ以外にコンパイラの中で行われる変数の置き換えを把握する環境が必要となってくる。本研究では、既存の fold に環境をパラメータとして１つ増やしても同様に形式化できることを示し、実際に fold および build を用いてコンパイラの各種最適化の形式化を行う。この形式化は各変換を自然に表現したものになっている。さらに fold/build 規則を用いて最適化関数を合成することにより中間データの削除を行った。

1　はじめに

再帰的データ構造に対する一般的な変換手法として fold によるコンストラクタの置き換えを挙げることができる。fold はデータ構造のもつ再帰構造を内包しているため、fold を使うと再帰構造を気にすることなく抽象度を上げてプログラムを書くことが可能である。

一方、我々が使用しているコンパイラは多くの変換を行っている為に複雑である。故に各種変換の入力と出力が等価であることの証明や、変換プログラムの処理効率の劣化が起こっているかなどの検証も現在、課題として残されている。そうした検証への足がかりとして、本研究ではコンパイラの最適化の fold による形式化を試みる。形式化を行うことでコンパイラをより深く理解し、本質を捉える上で重要な役割を果たすのではないかと考えた。

fold を使って形式化する際の問題点は環境の扱いである。既存の fold は普通、データのみを受け取り、コンパイラの中で行われる変数の置き換えを把握する環境などは出てこない。そこで fold で扱われている既存のパラメータ以外に、環境を新たに fold のパラメータに加えることを考える。本研究では、環境を加えても以前と同様の fold を使って形式化できることを示す。その上で実際にコンパイラの中で使われる各種最適化を fold を使って形式化する。環境を自然な形で記述することができるようになったため、各変換も自然に形式化することが可能になっている。

現段階ではまだ fold によって形式化をしただけで、各最適化の正当性の証明などには手をつけられていない。しかし、形式化することにより fold/build 規則を用いた融合が可能になることなどの成果が得られており、それによって最適化の若干の速度向上が得られている。

以下、２節ではリストの fold/build 規則について述べる。３節でコンパイラの構文木とその fold を示す。それを使って４節でα変換、５節でβ変換を形式化する。形式化を行えた後に、これらを使い、融合変換を行う。合成するにあたって必要になる関数の細分化について６節で述べ、７節で構文木に対する fold/build 規則を示したのち、実際の合成は８節で行う。９節では環境の合成について述べ、１０節、１１節でη変換、定数伝播について述べる。１２節で関連研究を述べ、１３節でまとめる。

2　リストに対する fold/build 規則

再帰関数の一般的な形として fold を挙げる。リストに対する fold とは、標準的にリストを受け取ってリスト内の cons と nil を与えられたパラメータ（下のプログラムでは k と z）に置き換えるものである。関数型言語 OCaml で書いた fold の定義を次に示す。

```
(* fold : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b *)
let rec fold k z list = match list with
                          [] -> z
                        | first::rest -> k first (fold k z rest)
```

fold k z はリスト list を受け取ると、リストの要素がまだ存在する場合は始めの要素と再帰をかけている残りの要素に対して k を行い、リストの終わりまできたら z を返す。例として fold を使ってリストを受け取ったら各要素を加算するプログラムを示す。

```
(* plus : int list -> int *)
let plus list = fold (+) 0 list
```

これは cons を (+) に、nil を 0 に置き換えたものである。リストの要素がまだ存在するときは始めの要素と残りの要素を (+) し、リストの終わりまできたら 0 を返す。

```
# plus [1; 2; 3];;
- : int = 6
```

fold はリストを受け取ってコンストラクタの置き換えを行うことでリストを消費しているが、対照的にリストを作り上げる関数も存在する。例えば整数 m を受け取ったら、1 から m までの要素の（逆順の）リストを作る関数は以下のように書ける。

```
(* make_list : int -> int list *)
let make_list m =
  build (fun c n -> let rec f m = if m = 0 then n else c m (f (m-1)) in f m)
```

ここで build は次のように定義できる関数である。

```
(* build : (('a -> 'a list -> 'a list) -> 'b list -> 'c) -> 'c *)
let build g = g (fun x y -> x::y) []
```

build は引数として渡された g に対してコンストラクタである cons と nil を渡す。cons と nil を受け取ると、g はそれらを使ってリストを作り上げる。ここで cons や nil をわざわざ build を用いて、c、n に変えているのは次に fold との合成を行う為である。fold、build を繋げる規則は Gill らによって以下のように示されている。

定理 1 (Gill, Launchbury, and Peyton Jones [2]) ある型 A に対して、g の型が

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

となるならば

$$fold\ k\ z\ (build\ g)\ =\ g\ k\ z$$

が成立する。

関数 make_list 内に出てくる build の引数は m が int なら `((int -> 'a -> 'a) -> 'a -> 'a)` となるため定理の仮定を満たす。従って make_list と plus を合成した関数 sum は以下のようになる。

```
(* sum : int -> int *)
let sum m = plus (make_list m)
          =  fold (+) 0  (build (fun c n ->
               let rec f m = if m = 0 then n else c m (f (m-1)) in f m))
          = (fun c n -> let rec f m = if m = 0 then n else c m (f (m-1)) in f m) (+) 0
          = let rec f m = if m = 0 then 0 else (+) m (f (m-1)) in f m
```

（3行目から4行目に移るところで定理を使用している。）fold/build 規則を使う前は、一度、中間リスト [m,m-1,...,1,0] を作ってから合計を求めていたが、規則適用後はこの中間リストが削除されて自然に 0 から m までの和を求めるプログラムとなっていることがわかる。即ち fold/build 規則は2回のループを1回に融合し、中間リストを削除していることが分かる。

3　コンパイラの構文木

ここで扱うコンパイラは関数型言語に対するコンパイラで、住井 [7] によるコンパイラをベースとしている。このコンパイラは字句解析、構文解析を終えたのち、 k-正規形 [1] と呼ばれる内部表現に変換される[1]。その後 α 変換を行った後、 β 変換、η 変換、let の結合性変換、不要変数除去、定数伝播、関数展開といった最適化を周回し、クロージャ変換、レジスタ割り当て、コード生成を経て機械語に変わる。ここで対象としているのは各種最適化の部分で、特に本論文では α 変換、 β 変換、 η 変換、定数伝播を扱うこととする。このようにコンパイラの最適化部では多くの変換を行っている為、fold/build 規則によりループ回数と中間データの削除をできると考えられる。

コンパイラの構文木は単純化したものとして以下のものを対象とする。

$$
\begin{array}{rcll}
M, N & ::= & x & \text{(変数)} \\
 & | & \lambda x.M & \text{(関数抽象)} \\
 & | & M\ N & \text{(関数適用)} \\
 & | & \text{let } x\ =\ M \text{ in } N & \text{(局所変数定義)}
\end{array}
$$

次に OCaml 用に書いたものを示す。

```
type lambda_t = Var of string                              (* 変数 *)
              | Lam of string * lambda_t                   (* 関数抽象 *)
              | App of lambda_t * lambda_t                 (* 関数適用 *)
              | Let of string * lambda_t * lambda_t        (* 局所変数定義 *)
```

リストのときと同様にして、この構文木に対する fold を定義する。fold はコンストラクタの数だけ引数を受け取り、上で `lambda_t` と書かれた箇所の再帰を引き受ける。それによりできた関数は以下のようになる。

```
(* fold : (string -> 'a) ->                        (* 変数 *)
          (string -> 'a -> 'a) ->                  (* 関数抽象 *)
          ('a -> 'a -> 'a) ->                      (* 関数適用 *)
          (string -> 'a -> 'a -> 'a) ->            (* 局所変数定義 *)
```

[1] k-正規形とは全ての部分式に名前がついている構文であるが、ここでは単に let 文の加わったλ式と思って差し支えない。

```
          lambda_t -> 'a *)
let rec fold var lam app let1 tree = match tree with
    Var(str) -> var str
  | Lam(str, t) -> lam str (fold var lam app let1 t)
  | App(t1, t2) -> app (fold var lam app let1 t1) (fold var lam app let1 t2)
  | Let(str, t1, t2) -> let1 str (fold var lam app let1 t1) (fold var lam app let1 t2)
```

以下では、この fold を使って、コンパイラの各種最適化を表現していく。

4 α変換

はじめにα変換について考えてみる。α変換は、変数の置き換えを行って別の変数には別の名前を与えるような変換のことである。従って、新たに変数が導入される関数抽象と let 文のみで処理が行われる。α変換の定義は以下のようになる。

$$
\begin{array}{rcl}
\alpha \ : \ \text{式} \times \text{環境} & \rightarrow & \text{式} \\
\alpha[\![x]\!]\,\rho & = & \rho(x) \\
\alpha[\![\lambda x.\, M]\!]\,\rho & = & \lambda q.\, \alpha[\![M]\!]\,\rho[x \mapsto q] \\
 & & \text{ただし } q \text{ は他とはぶつからない新しい変数。} \\
\alpha[\![M\ N]\!]\,\rho & = & (\alpha[\![M]\!]\,\rho)\,(\alpha[\![N]\!]\,\rho) \\
\alpha[\![\text{let } x = M \text{ in } N]\!]\,\rho & = & \text{let } q = \alpha[\![M]\!]\,\rho \text{ in } \alpha[\![N]\!]\,\rho[x \mapsto q] \\
 & & \text{ただし } q \text{ は他とはぶつからない新しい変数。}
\end{array}
$$

これは通常、再帰関数で書くことができるが、ここではデータ構造に着目した fold で書くことを試みる。先程示したリストの例と同様にデータ構造で再帰している箇所は fold が受け持ち、実際に何を行うかは引数で受け取ることにする。これは、上記の定義に**環境が出てきていなければ**次のように書ける。

```
let alpha = fold (fun str -> Var(ρ(str)))
                 (fun str t -> let q = gensym str in Lam(q, t))
                 (fun t1 t2 -> App(t1, t2))
                 (fun str t1 t2 -> let q = gensym str in Let(q, t1, t2)
```

（ここで gensym は新しい変数を生成する関数である。）

しかし、実際には1行目の $\rho(str)$ のところで str をどの変数に置き換えるかがこのままでは不明である。この情報は構文木のより根本付近で定義されるため、何らかの方法でそれをここまで運んでくる必要がある。従ってα変換を fold と build で形式化する際に最も着目すべき点はこの環境の取り扱いである。

従来の fold/build 規則には環境というべきものが存在していなかった。そのため余計なパラメタがある再帰式をそのままの形で fold を使って表現するのは自明ではなかった。コンパイラの形式化をする際に必要な環境が、既存の fold/build 規則にどのように追加できるか考えてみる。

fold はコンストラクタの置き換えのみをおこなっているが、置き換える相手は任意なので、これを上のような単に構文木を作るものではなく「環境を受け取ったら構文木を作る」ような関数に変換することが考えられる。これは環境の型を `table_t` とすると fold の型に出ている型変数 'a を `table_t→lambda_t` というものに置き換えたことに相当する。この考え方に基づいてα変換を表現すると以下のようになる。

```
(* alpha : lambda_t -> table_t -> lambda_t *)
let alpha = fold (fun str table -> Var(search str table))
```

```
                (fun str t table -> let q = gensym str in Lam(q, (t (extend str q table))))
                (fun t1 t2 table -> App(t1 table, t2 table))
                (fun str t1 t2 table ->
                   let q = gensym str in Let(q, t1 table, t2 (extend str q table)))
```

ここで search は環境から値を取り出す関数、extend は環境を拡張する関数である。fold が受け持つ再帰の返り値は table_t→lambda_t 型なので fold に渡す関数中の t、t1、t2 は全てこの型をもつ。それらに対し適切な環境を渡すことで α 変換に必要な環境の受け渡しを行っている。例えば $let\ x\ =\ M\ in\ N$ を変換する場合、M の部分は ρ、N の部分は $\rho[x \mapsto q]$ で変換するが、それに対応して t1 には table が t2 には (extend str q table) が渡されている。このように fold への引数を環境を受け取る前に書くことで、従来の fold の定義を変更することなく自然に環境の受け渡しを表現することができた。

最後に、最初に示したリストと同様に Var、Lam、App、Let のコンストラクタを build を用いて抽象化することを考える。構文木に対する build は以下により定義できる。

```
(* build : ((string -> lambda_t) ->                                  (* 変数 *)
            (string -> lambda_t -> lambda_t) ->                     (* 関数抽象 *)
            (lambda_t -> lambda_t -> lambda_t) ->                   (* 関数適用 *)
            (string -> lambda_t -> lambda_t -> lambda_t) ->         (* 局所変数定義 *)
            'a) ->
           'a *)
let build g = g (fun str -> Var(str))
                (fun str t -> Lam(str, t))
                (fun t1 t2 -> App(t1, t2))
                (fun str t1 t2 -> Let(str, t1, t2))
```

これを使って α 変換を書き直すと、次のように書くことができる。

```
(* alpha : lambda_t -> table_t list -> lambda_t *)
let alpha = build (fun var lam app let1 ->
              fold (fun str table -> var (search str table))
                   (fun str t table -> let q = gensym str in lam q (t (extend str q table)))
                   (fun t1 t2 table -> app (t1 table) (t2 table))
                   (fun str t1 t2 table ->
                      let q = gensym str in let1 q (t1 table) (t2 (extend str q table))))
```

この定義は α 変換の定義をそのまま素直に変換したものになっている。以上で fold、build を用いて α 変換を形式化することができた。

5 β 変換

β 変換は無駄な変数の置き換えをなくす変換のことで、let 文で変数がまた別の変数に置き換わっている場合に行われる。例えば

```
let x = y in x+1
```

のように y を単純に x に置き換えているようなときは、β 変換後は

```
let x = y in y+1
```

のように x が全て y に置き換えられる。（この式は、その後不要変数除去を行うと y+1 になる。）β 変換の定義は以下のようになる。

$$
\begin{array}{rcl}
\beta \ : \ \text{式} \times \text{環境} & \rightarrow & \text{式} \\
\beta[\![x]\!]\,\rho & = & \rho(x) \\
\beta[\![\lambda x.\, M]\!]\,\rho & = & \lambda x.\, \beta[\![M]\!]\,\rho \\
\beta[\![M\ N]\!]\,\rho & = & (\beta[\![M]\!]\,\rho)\,(\beta[\![N]\!]\,\rho) \\
\beta[\![\text{let } x = y \text{ in } N]\!]\,\rho & = & \text{let } x = \rho(y) \text{ in } \beta[\![N]\!]\,\rho[x \mapsto \rho(y)] \\
\beta[\![\text{let } x = M \text{ in } N]\!]\,\rho & = & \text{let } x = \beta[\![M]\!]\,\rho \text{ in } \beta[\![N]\!]\,\rho
\end{array}
$$

ここで α 変換と異なるのは、全ての let 文に対して同じ処理を行うのではなく、let 文の中の第 2 引数が変数だった場合とそうでない場合とで処理が異なることである。この構文的な特徴を反映させるため、let 文の場合を二つに分けて fold を次のように変更する。

```
(* fold : (string -> 'a) ->                          (* 変数 *)
          (string -> 'a -> 'a) ->                    (* 関数抽象 *)
          ('a -> 'a -> 'a) ->                        (* 関数適用 *)
          (string -> string -> 'a -> 'a) ->          (* 局所変数定義 let1 *)
          (string -> 'a -> 'a -> 'a) ->              (* 局所変数定義 let2 *)
          lambda_t -> 'a *)
let rec fold var lam app let1 let2 tree = match tree with
    Var(str) -> var str
  | Lam(str, t) -> lam str (fold var lam app let1 let2 t)
  | App(t1, t2) -> app (fold var lam app let1 let2 t1) (fold var lam app let1 let2 t2)
  | Let(str, t1, t2) -> (match t1 with
             Var(str1) -> let1 str str1 (fold var lam app let1 let2 t2)
           | _ -> let2 str (fold var lam app let1 let2 t1) (fold var lam app let1 let2 t2))
```

また build も同様に let の場合分けを行う。

```
(* build : ((string -> lambda_t) ->                                  (* 変数 *)
            (string -> lambda_t -> lambda_t) ->                     (* 関数抽象 *)
            (lambda_t -> lambda_t -> lambda_t) ->                   (* 関数適用 *)
            (string -> string -> lambda_t -> lambda_t) ->           (* 局所変数定義 let1 *)
            (string -> lambda_t -> lambda_t -> lambda_t) ->         (* 局所変数定義 let2 *)
            'a) ->
           'a *)
let build g = g (fun str -> Var(str))
                (fun str t -> Lam(str, t))
                (fun t1 t2 -> App(t1, t2))
                (fun str1 str2 t -> Let(str1, Var(str2), t))
                (fun str t1 t2 -> Let(str, t1, t2))
```

let 文の第 2 引数が変数だった場合とそうでない場合に対応するため、fold への引数が let1 と let2 の二つに増えるている。これは let 文の構文を二つ用意したことに相当するが、ここでは最初に示したコンパイラの構文木の定義は変更せず、fold の場合分けを増やすことで対応している。

fold と build で形式化したβ変換は次のようになる。

```
(* beta : lambda_t -> table_t -> lambda_t *)
let beta = build (fun var lam app let1 let2->
            fold (fun str table -> var (search str table))
                 (fun str t table -> lam str (t table))
                 (fun t1 t2 table -> app (t1 table) (t2 table))
                 (fun str1 str2 t table ->
                      let1 str1 (search str2 table) (t (extend str1 (search str2 table) table)))
                 (fun str t1 t2 table -> let2 str (t1 table) (t2 table)))
```

この定義もα変換のときと同様に、β変換の定義をそのまま素直に表現したものになっている。これでβ変換も fold と build を用いて形式化することができた。

6　関数の細分化

α変換とβ変換が fold と build で形式化できたので、これらを fold/build 規則に従い合成を行いたい。しかし、β変換は let 文を二つの場合に分けているために構文がα変換とは合わなくなっている。そこでα変換の構文をβ変換の構文に合わせ、fold の定義を統一することを考える。α変換における let 文の処理は let 文の第２引数に関わらず

```
fun str t1 t2 table ->
  let q = gensym str in let1 q (t1 table) (t2 (extend str q table))
```

であった。新しい fold に対応するためには、これを第２引数が変数だった場合とそうでない場合に分離すればよい。第２引数が変数でない場合は上のまま行ってよい。一方、第２引数が変数 `Var(str)` だった場合は、`(t1 table)` の部分を実行すると、必ず `search str table` という変数になるなることがわかる。これらのことから新しい fold を使ったα変換は以下のように定義できる。

```
(* alpha : lambda_t -> table_t  -> lambda_t *)
let alpha = build (fun var lam app let1 let2 ->
              fold (fun str table -> var (search str table))
                   (fun str t table -> let q = gensym str in lam q (t (extend str q table)))
                   (fun t1 t2 table -> app (t1 table) (t2 table))
                   (fun str1 str2 t table -> let q = gensym str1 in
                      let1 q (search str2 table) (t (extend str1 q table)))
                   (fun str t1 t2 table -> let q = gensym str in
                      let2 q (t1 table) (t2 (extend str q table))))
```

fold の４つ目の引数である

```
fun str1 str2 t table ->
  let q = gensym str1 in let1 q (search str2 table) (t (extend str1 q table))
```

が新たに導入された第２引数が変数だった場合である。この変換は、現在は手で行なっているが、簡単な部分評価を行うことで自動的に行えるのではないか、検討中である。

7　構文木に対する fold/build 規則

リストに対する fold/build 規則と同様に、構文木に対しても fold/build 規則を作ることができる。

定理 2 g の型が

$$g : \forall\beta.(string \to \beta) \to (string \to \beta \to \beta) \to (\beta \to \beta \to \beta)$$
$$\to (string \to string \to \beta \to \beta) \to (string \to \beta \to \beta \to \beta) \to \beta$$

となるならば

$$fold\ var\ lam\ app\ let1\ let2\ (build\ g)\ =\ g\ var\ lam\ app\ let1\ let2$$

が成立する。

この定理は高野ら [8] の結果からすぐに導かれる。高野らは fold/build 規則が一般的なデータ構造に対して成り立つことを示しており、上の定理はそれを構文木に対して適用したものとなっている。β 変換の let 文の場合分けを反映して fold が受け取る引数を 1 つ増やしているが、let 文を分解する必要がないときはこの引数は不要である。また逆により多くの場合分けが必要ならば引数もそれに応じて増えることになる。しかし、いずれの場合も高野らの結果の特殊なケースになっており、対応する fold/build 規則を簡単に証明することができる。

8　2 つの合成

構文木に対する fold/build 規則を使って、ここでは α 変換と β 変換を合成する。コンパイラは α 変換を行った後で β 変換を行っているので、α 変換後に出力される構文木を β 変換の入力とする。

```
(* ab : lambda_t -> table_t -> table_t -> lambda_t *)
let ab tree tablea tableb = beta (alpha tree tablea) tableb
```

ここで tablea は α 変換用の環境、tableb は β 変換用の環境である。これを先程から書いている fold、build で展開すると次のようになる。

```
= build (fun var lam app let1 let2 -> (* beta の定義 *)
     fold (fun str table -> var (search str table))
          (fun str t table -> lam str (t table)
          (fun t1 t2 table -> app (t1 table) (t2 table))
          (fun str1 str2 t table ->
                    let1 str1 (search str2 table) (t (extend str1 (search str2 table) table)))
          (fun str t1 t2 table -> let2 str (t1 table) (t2 table) )

          (build (fun var lam app let1 let2 -> (* alpha の定義 *)
              fold (fun str table -> var (search str table))
                   (fun str t table -> let q=gensym str in lam q (t (extend str q table)))
                   (fun t1 t2 table -> app (t1 table) (t2 table))
                   (fun str1 str2 t table -> let q = gensym str1 in
                                             let1 q (search str2 table) (t (extend str1 q table)))
                   (fun str t1 t2 table -> let q = gensym str in
                                             let2 q (t1 table) (t2 (extend str q table)))
                   tree
```

```
                tablea))
        tableb)
```

これに fold/build 規則を適用すると、次のようになる。

```
(* ab : lambda_t -> table_t -> table_t -> lambda_t *)
let ab = build (fun var lam app let1 let2 ->
          fold (fun str tablea tableb -> var (search (search str tablea) tableb))
               (fun str t tablea tableb -> let q = gensym str in
                  lam q ((t (extend str q tablea)) tableb))
               (fun t1 t2 tablea tableb -> app ((t1 tablea) tableb) ((t2 tablea) tableb))
               (fun str1 str2 t tablea tableb -> let q = gensym str1 in
                  let p=(search (search str2 tablea) tableb) in
                  let1 q p ((r (extend str q tablea)) (extend q p tableb))
               (fun str t1 t2 tablea tableb -> let q = gensym str in
                  let2 q ((t1 tablea) tableb) ((t2 (extend str q tablea)) tableb)))
```

関数抽象と let 文で gensym を使って新たな変数を生成し、その変換情報を環境に追加していることからα変換が行われていることが分かる。また let 文の第 2 引数が変数だった場合には、その情報も環境に追加しているのでβ変換していることが分かる。以上のことからこの合成はα変換とβ変換を合成したものになっていることが分かる。

9 環境の合成

前節で 2 つの変換は 1 つに合成することができたが、それぞれの変換で使われていた環境である table は 2 つ存在している。例えば変数を扱う関数は

```
fun str tablea tableb -> var (search (search str tablea) tableb)
```

のようになっている。このようになっているとループの回数は減っているが、中間データはクロージャの形で残ってしまっている[2]。そこで次に table の合成も行い、実際に中間データを削除することを試みる。table を合成した関数は以下のようになる。

```
(* ab : lambda_t -> table_t -> lambda_t *)
let ab = build (fun var lam app let1 let2 ->
           fold (fun str table -> var (search str table))
                (fun str t table -> let q = gensym str in
                                    lam q (t (extend str q table)))
                (fun t1 t2 table -> app (t1 table) (t2 table))
                (fun str1 str2 t table -> let q = gensym str1 in
                   let p = (search str2 table) in let1 q p (t (extend str1 p table)))
                (fun str t1 t2 table -> let q = gensym str in
                   let2 q (t1 table) (t2 (extend str q table))))
```

[2]OCaml のように関数の適用環境時にクロージャを作らないように実装されていれば、クロージャは削除されている。しかし、環境は依然として二つ必要である。

変数 x が α 変換の環境で y になり、それがさらに β 変換の環境で z になるのであれば α β の環境では x をいきなり z にしている。これは環境が関数で表現されていれば関数合成、リストで表現されていればリストの融合変換をしていることに相当する。

なお、予備的な実験を行った結果、実行時間は α 変換と β 変換を順に行ったものと比べて、２つの変換を合成した場合には８４%、さらに環境を合成した場合には８０%に減少した。

10 η 変換

η 変換とは、入力プログラム中の `let x = M in x` を `M` にする変換である。定義は以下のようになる。

$$
\begin{aligned}
\eta \ : \text{式} &\rightarrow \text{式} \\
\eta[\![x]\!] &= x \\
\eta[\![\lambda x.\, M]\!] &= \lambda x.\, \eta[\![M]\!] \\
\eta[\![M\ N]\!] &= (\eta[\![M]\!])\,(\eta[\![N]\!]) \\
\eta[\![\text{let } x = M \text{ in } x]\!] &= \eta[\![M]\!] \\
\eta[\![\text{let } x = M \text{ in } N]\!] &= \text{let } x = \eta[\![M]\!] \text{ in } \eta[\![N]\!]
\end{aligned}
$$

η 変換では let 文の第３引数が変数のときに特別な処理を行うので let 文の第３引数が変数かどうかで場合分けを行う必要がある。それに伴って fold の受け取る引数の数も変わる。そうしてできた fold と η 変換は以下のようになる。

```
let rec fold var lam app let1 let2 tree = match tree with
    Var(str) -> var str
  | Lam(str, t) -> lam str (fold var lam app let1 let2 t)
  | App(t1, t2) -> app (fold var lam app let1 let2 t1) (fold var lam app let1 let2 t2)
  | Let(str, t1, t2) -> (match t2 with
             Var(str1) -> let1 str (fold var lam app let1 let2 t1) str1
           | _ -> let2 str (fold var lam app let1 let2 t1) (fold var lam app let1 let2 t2))

let build g = g (fun str -> Var(str))
                (fun str t -> Lam(str, t))
                (fun t1 t2 -> App(t1, t2))
                (fun str1 t str2 -> Let(str1, t, Var(str2)))
                (fun str t1 t2 -> Let(str, t1, t2))

(* eta : lambda_t -> lambda_t *)
let eta = build (fun var lam app let1 let2->
            fold (fun str -> var str)
                 (fun str t -> lam str t)
                 (fun t1 t2 -> app t1 t2)
                 (fun str1 t str2 -> if str1 = str2 then t else let1 str1 t str2)
                 (fun str t1 t2 -> let2 str t1 t2))
```

この fold、build で形式化した η 変換を α 変換や β 変換と合成するためには、再度、構文を合わせなくてはならない。η 変換では let 文の第３引数が変数かどうかで場合分けしている。従って、この場合を α 変換や β 変

換でも作る必要がある。β変換では第２引数によって場合分けしていたのでβ変換と合成しようと思えば合計４通りが必要になる。これらは全て６節に述べた通り、機械的に得ることができる。ここではこれ以上述べないが、細分化後は前節と同様にしてα変換やβ変換（あるいはそれらを合成した変換）などと合成できる。

11　定数伝播

定数伝播は、変数の値がすでに定義されている場合には変数の代わりにその値を代入した式を返すものである。この変換では、変数が（整数などの）定数になっているかによって動作を変えるので、まず、構文木に定数が入ってくる必要がある。（加えて、四則演算なども加えるのが普通である。）その上で、各変数が定数になっているかを環境を使用して覚えておき、それに従って変換を行なう。詳しくは述べないが、これまでの変換と同様にして fold と build を用いて形式化できる。その際、定数になっているかいないかで場合分けする必要があるため、それに応じて fold の定義も細分化される。細分化後は定数伝播も合成できるようになる。

12　関連研究

fold に余分なパラメタが来てもうまく処理できるように拡張する仕事としては、西村 [5, 6] や Voigtländer[9] による仕事があげられる。彼らは accumulator があるような関数の融合変換を扱っている。彼らの仕事は accumulator に計算結果が格納されるような関数でも扱えるようになっている一方で、結果にはサイクリックな依存関係が生じ、lazy な let 文が必要になっている。我々の仕事は、余計なパラメタ（環境）はあくまで補助的な役割をになっているに過ぎず、一般的な accumulator を使ったプログラムなどは扱えない。しかし、逆に fold の定義自体は以前のものと同じものが使えており、またサイクリックな依存関係も生じないですんでいる。

コンパイラを fold 等を使って形式化する研究としては、Meijer [3] によるものがあげられる。この仕事では、カテゴリ論的な考察をもとにいろいろな言語を形式化し、簡単な機械語を出力するところまでを考慮している。我々の仕事は、まだ最適化のみしか扱っていないが、形式化の際、環境を自然に表現する方法を採用しているため、実際にコンパイラに使われているような変換も自然に表現できている。

補助的な情報を引数として渡す方法は、例えばリストの最初のｎ個の要素をとってくる関数

```
let take n list = fold (fun x y m -> if m = 0 then [] else (x::(y (m-1))))
                       (fun m -> [])
                       list n
```

などが Meijer, Jeuring[4] によって紹介されている。本研究は、彼らの仕事とは独立に行われたが、彼らの手法をコンパイラ最適化において環境を渡すのに適用したと見ることもできる。

13　まとめ

本論文では、コンパイラ内部で行なわれる各種最適化を fold を使って形式化した。従来、扱いが自明でなかった環境を自然に扱えることを示し、実際にいくつかの変換を fold/build 関数で表現した。これにより、各変換の動きが簡潔に記述されるばかりでなく、fold/build 規則を使った融合変換を行なうことにより、中間データの削除を行なった。最適化によって必要な構文的分類が異なるため、fold の定義が細分化されるが、この細分化は機械的な操作で行なうことができる。

本稿で紹介した変換では、入力と出力はいずれも同じデータ構造だが、これらは同じである必要はない。コンパイラで行なわれている中間表現への変換なども同様に形式化できることが期待される。

現在のところ、まだα変換、β変換、η変換、定数伝播の４つの変換を形式化したばかりであるが、今後、コンパイラ内の他の変換についても形式化を進めていく予定である。また、本研究では形式化に着目し、各変換の正当性（変換の前後でプログラムの意味が不変であるか）については、まだ議論していない。しかし、形式化されたことにより、そういった議論もしやすくなっていくと期待している。

謝辞　国立情報学研究所の高野明彦氏、及び査読者の方々からは有益なコメントを頂きました。

参考文献

[1] Birkedal, L., M. Tofte, and M. Vejlstrup “From Region Inference to von Neimann Machines via Region Representation Inference,” *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pp. 171–183 (January 1996).

[2] Gill, A, J. Launchbury, and S. L. Peyton Jones “A Short Cut to Deforestation,” *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA’93)*, pp. 223–232 (June 1993).

[3] Meijer, H. J. M. *Calculating Compilers*, Ph.D. thesis, University of Nijmegen (February 1992).

[4] Meijer, E., and J. Jeuring “Merging Monads and Folds for Functional Programming,” *Advanced Functional Programming (LNCS 925)*, pp. 228–266 (May 1995).

[5] Nishimura, S. “Deforesting in Accumulating Parameters via Type-Directed Transformations,” *Proceedings of the Third Asian Workshop on Programming Languages and Systems (APLAS’02)* (December 2002).

[6] Nishimura, S. “Correctness of a Higher-Order Removal Transformation through a Relational Reasoning,” *Proceedings of the First Asian Symposium on Programming Languages and Systems (APLAS’03), LNCS 2895*, pp. 358–375 (November 2003).

[7] 住井英二郎、コンパイラ演習資料 `http://www.yl.is.s.u-tokyo.ac.jp/~sumii/pub/compiler-enshu/` (2002).

[8] Takano, A., and E. Meijer “Shortcut Deforestation in Calculational Form,” *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA’95)*, pp. 306–313 (June 1995).

[9] Voigtländer, J. “Using Circular Programs to Deforest in Accumulating Parameters,” *ACM SIGPLAN Asian Symposium on Partial Evaluation and Semantics-Based Program Manipulation (ASIA-PEPM ’02)*, pp. 126–137 (September 2002).