型付きDSLに対するプログラム変換の型安全なフレームワーク

高木 尚　亀山 幸義

領域特化言語（DSL）の設計においては，型による安全性が重要である．コンパイラやプログラム変換の記述は誤りが発生しやすく，出力を確認しデバックするのは現実的ではない．DSL を型付きで埋め込み，プログラム変換実装におけるミスを事前に取り除くことを目指す．型を保存する変換のみを許すことで，プログラム変換に安全性を与える試みがあるが，従来の手法では記述が困難な変換が存在する．自由変数を含むコードの解析や，MGU(最汎単一化子) の計算を行う変換は難しく，特にスーパーコンパイラという最適化は安全な実装ができるかわかっていない．本研究では，従来の高階抽象構文 (HOAS) で表現していたコードを，型付き de Bruijn インデックスを用いたコードの表現に変換することで，型安全性を保ったまま表現力の拡張を図ったフレームワークを提案する．フレームワークにおけるスーパーコンパイラの実装を通して，その表現力について検討を行った．

1　導入

1.1　DSL とタグレスファイナル法

領域特化言語 (DSL) は，特定の問題領域における処理の記述のために，広く用いられている．DSL の実現方法には，様々なものがあるが，多種多様な DSL が比較的短期間に設計・開発され改訂されていく，という状況のもと，開発・設計・最適化等を軽量にできる DSL 開発手法 (フレームワーク) に注目が集まっている．軽量に開発される DSL は，多くの場合，汎用プログラミング言語に埋め込まれるかたちで実現され，DSL のプログラムに対する最適化は，埋め込まれたコードに対するプログラム変換として実現される．

Kiselyov らによるタグレスファイナル法[3] は，静的に型付けられた DSL(以降では型付き DSL と呼ぶ) を汎用プログラミング言語に埋め込む手法である．埋め込み先の言語をホスト言語と呼ぶ．タグレスファイナル法の特徴は，DSL の構文だけでなく，その型付け規則もホスト言語に埋め込むことにより，DSL の項の型付けをホスト言語の型システムで行う点にある．この手法は，DSL のプログラムの型エラーを静的に発見できるだけでなく，DSL のプログラム変換が型付けを保存しないことも静的に発見できる．そのため，DSL に対する様々な領域固有の最適化を実装するプログラマにとっても利点がある．Kiselyov らは，SQL のクエリ最適化[8] や部分評価などの領域にタグレスファイナル法を適用している．

タグレスファイナル法によるプログラム変換の記述は，静的に型が付く保証が得られるという利点がある一方で，記述の容易さにおいて問題がある．その原因は，変数束縛の表現方法であり，通常のタグレスファイナル法は，高階抽象構文 (HOAS) を利用して変数束縛を表現するため，項の中の変数を直接取り扱う変換を表現するのが容易ではないといったことや，閉じた項のみが表現可能であるといった問題がある．この問題を回避するためには，(1)de Bruijn インデックスによる表現を導入する[4]，(2) 弱高階抽象構文 (WHOAS) による表現を導入する，(3) 依存型を持つ言語をホスト言語として型文脈を直接扱う [1], という方法がある．しかし，(1) は，DSL のプログラム記述が簡便ではなくなる，(2) は WHOAS を用い

The type safe framework for program transformations of typed DSL

Sho Takaki　Yukiyoshi Kameyama, 筑波大学コンピュータサイエンス専攻, Dept. of Computer Science, University of Tsukuba.

ても変数の束縛の構文においては HOAS と同様の扱いにくさがあるという点，(3) はホスト言語そのものが拡張されて煩雑になるという問題がある．一方で，この問題はタグレスファイナル法そのものに起因するのではなく，既存の枠組みの上に，変換を表現できる幅を広げる仕組みを構築することは可能であると我々は考えている．

1.2　提案方式

本研究では，これらの観察をもとに，タグレスファイナル法において，外部的・表層的には HOAS を用い，内部的には必要に応じて，型付き de Bruijn インデックス (以降 TDB と表記する) を用いるというハイブリット型プログラム変換フレームワークを提案する．2 つの表現間の変換は，個々の DSL に依存しないためフレームワークが提供を行う．TDB への変換および TDB 上での操作は一貫して型の保証が保たれるため，全体を通して型の保証は保たれる．図 1 は，提案方式による最適化の概略である．DSL の最適化の開発者は本フレームワーク上で，基本的には HOAS を用いて最適化を記述する．同時に，パターンマッチや変数操作を含む最適化では TDB を用いるということが可能になる．プログラム変換を記述するのに適切な表現形式を選択できるフレームワークを提供し，プログラム変換を記述する際の負担が軽減することが本研究の目的である．

1.3　本論文の構成

本論文は，本章を含め全 5 章から構成される．

第 2 章では，型付き DSL として対象言語の一例を提示し，それが HOAS と TDB の双方でどのようにメタ言語上で表現されているかについて述べる．第 3 章では，本研究の提示するフレームワークの実装方式，並びに使用方法について説明する．加えて，HOAS 上では，素直には実装が難しかった変換のプロセスについて例を挙げ，TDB 上ではどのように行うかについて述べる．第 4 章では，本フレームワークをもちいて，複雑な変数操作を必要とするプログラム変換を記述することに向けての課題を，スーパーコンパイラ [9] というプログラム最適化の手法の実装を通して報告する．第 5 章では，研究内容と成果についてまとめ，今後の研究課題について述べる．

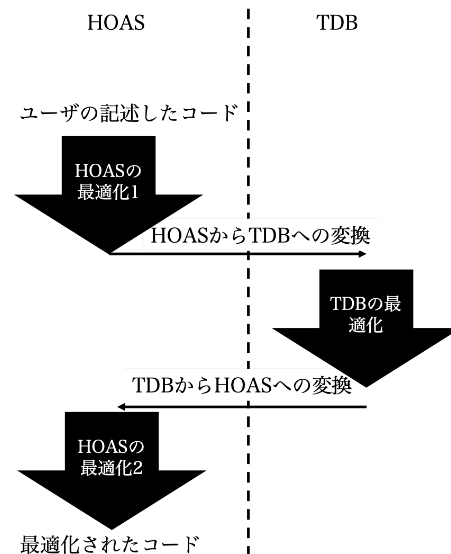

図 1　提案方式による最適化

2　埋め込み手法

2.1　対象言語の一例

この論文では，具体的な説明をするため、対象となる DSL を 1 つ固定して考える．これの構文を図 2 に示す．

型付け規則の一部として，関数と変数に関わる規則は図 3 の通りである．

2.2　高階抽象構文

高階抽象構文 (HOAS) は，ホスト言語の関数として対象言語のコードを表現する．加えて，関数の型を設定することで，DSL に型安全性を与える．関数の型の設定方法はホスト言語により異なるが，今回は OCaml をホスト言語としてモジュールのシグネチャに記述する．対象言語の一部について対応する関数の型を記述したシグネチャの一部をソースコード 1 に示す．シグネチャは，モジュールの型を示すものであり，`val` で宣言したものが，モジュールにおける関数の型に対応する．`type 'a exp` で宣言しているのは，抽象型といい，ここでは具体的な型を設定しない．シ

t ::=
 v (変数), $ft_1 \ldots t_n$ (関数適用)
 nil (空リスト), **cons**(t,t) (リスト構成子)
 case $t : \tau$ *list* **of** [] $\rightarrow$ t | ($v_h : v_{tl}$) $\rightarrow$ t (case 式)
 if v $=$ t **then** t **else** t (if 式)
f ::=
 $\lambda v_1 : \tau_1 . \ldots \lambda v_n : \tau_n . t$ (関数)
 fix(f : τ) (再帰関数)
 fix2($f : \tau$ **and** $f : \tau$) (相互再帰関数)
 fst(f) (左抽出), **snd**(f) (右抽出)
τ ::=
 char (文字型), *bool* (真偽値型)
 τ *list* (リスト型), (τ,τ) (組型), $\tau \rightarrow \tau$ (関数型)

図 2　対象言語

グネチャを実装したモジュールにおいては，この型を個別に具体化することができる．

ソースコード 1　HOAS による対象言語の埋め込み

```
module type THOAS = sig
  type 'a exp
  val tlam :
    'a typ
    -> ('a exp -> 'b exp)
    -> ('a -> 'b) exp
  val tapp :
    ('a -> 'b) exp
    -> 'a exp
    -> 'b exp
  ...
end
```

注目するべきは，ラムダ抽象の構造である．ラムダ抽象の第 2 引数の型は，`('a exp -> 'b exp)` であり，これはホスト言語上のラムダ式に対応する．例えば，$\lambda x : bool.$**cons**(x,**nil**) は，`tlam (Bool) (fun x -> cons x nil)` となる．この x のように，各変数の管理は，ホスト言語上に委譲されるため，束縛されていない変数や，誤った型での変数の扱いはホスト言語の型検査によって防止することができる．

一方で，自由変数を表現することができないため，全ての項は閉じていなければならない．加えて，項の構成子がホスト言語上の関数であることと，3.3.1 節で述べるように `tlam` の内部には，関数適用行わなければアクセスできないことから，パターンマッチや項同士の比較を正しく実装することは容易ではない．

2.3　型付き de Bruijn インデックス (TDB)

本研究では，TDB を，ホスト言語の一般化代数データ型を用いて埋め込むことにより表現する．ソースコード 2 は，OCaml をホスト言語として対象言語でコードを表すデータ構造の一部である．型 `index`, `tTerm` は共に型パラメータが 3 つ設定されており，1 つ目が型環境，3 つ目が表現型をさす．$\vdash$ の左の型環境を第 1 パラメータ，右の表現型を第 3 パラメータとみて，型規則と同様の形式をしていることがわかる．型パラメータの 2 つ目は，プログラム変換で一時的に用いるもので関数の型環境を表す．

変数は，通常の de Bruijn インデックスと同様に，変数束縛の位置をインデックスで示したものを変数の代わりに用いる．

GADT では，データの構造により設定される型が異なるため，DSL の型を埋め込むことができている．また，パターンマッチや項同士の比較も容易に実装ができる．一方で，変数がインデックスになるため，可読性が低く，プログラム変換の際にも変数を適切に操作することが求められる．

ソースコード 2　TDB による対象言語の埋め込み

```
type (_, _, _) index =
  | Z : ('a  -> 'g,  'd, 'a) index
  | S : ('g, 'd, 'b)  index
         -> ('a -> 'g, 'd,  'b)  index

type (_, _, _) tTerm =
  | TVar : ('g, 'd, 'a) index
     -> ('g, 'd, 'a) tTerm
  | TLam : 'a typ *
           ('a -> 'g, 'd, 'b) tTerm
     -> ('g, 'd, ('a -> 'b)) tTerm
  | TApp : ('g, 'd, 'a -> 'b) tTerm *
           ('g, 'd, 'a) tTerm
     -> ('g, 'd, 'b) tTerm
  ...
```

$$\frac{(v:\tau)\in\Gamma}{\Gamma\vdash v:\tau}(\text{T-Var}) \quad \frac{\Gamma,(v:\tau_1)\vdash t:\tau_2}{\Gamma\vdash\lambda v:\tau_1.t:\tau_1\to\tau_2}(\text{T-Lam}) \quad \frac{\Gamma\vdash t_1:\tau_1\to\tau_2\ \Gamma\vdash t_2:\tau_1}{\Gamma\vdash t_1\ t_2:\tau_2}(\text{T-App})$$

図 3　対象言語の型付け規則の一部

2.4　結合関数の例

2 つのリストを受け取り，結合させたリストを返す関数は，まず，対象言語の構文では，以下のように定義される．

$$\begin{gathered}\mathbf{fix}(\lambda f: \mathit{char\ list} \to \mathit{char\ list} \to \mathit{char\ list}.\\ \lambda x: \mathit{char\ list}.\lambda y: \mathit{char\ list}.\\ \mathbf{case}\ x: \mathit{char\ list}\ \mathbf{of}\ []\ \to\ y \mid (x_h : x_{tl})\ \to\\ \mathbf{cons}(x_h, f\ x_{tl}\ y):\\ \mathit{char\ list} \to \mathit{char\ list} \to \mathit{char\ list})\end{gathered}$$

この関数を HOAS を用いて埋め込むと次のようになる．

ソースコード 3　HOAS による結合関数

```
fix (Arrow(List(Char),
     Arrow(List(Char), List(Char)))
 (tlam (Arrow(List(Char),
        Arrow(List(Char), List(Char))))
            (fun f ->
   tlam (List(Char)) (fun x ->
    tlam (List(Char)) (fun y ->
     case (Char) x y (fun xh xtl ->
      cons (xh) (tapp (tapp f xtl) y)
     )
    )
   )
 )
)
```

TDB では次の通りである．

ソースコード 4　TDB による結合関数

```
Fix(Arrow(List(Char),
      Arrow(List(Char), List(Char))),
   TLam(Arrow(List(Char),
         Arrow(List(Char), List(Char))),
     TLam(List(Char), TLam(List(Char),
       Case(Char,
            TVar(S Z),
            TVar(Z),
            Cons(TVar(Z),
            TApp(TApp(TVar(S (S (S (S Z)
                ))), TVar(S Z)),
                 TVar(S (S Z)))))
      ))
   )
)
```

3　両表現を併用するフレームワーク

まず，両表現をタグレスファイナル上にハイブリットに構成する方法を述べ，開発者視点での使用法と，HOAS では正しく記述することが困難な操作についての，本フレームワークでの実装を説明する．

3.1　フレームワークとタグレスファイナル

タグレスファイナル法では，対象言語に対するプログラム変換を，ファンクタすなわちモジュール間の関数として実装する．ファンクタの内部では，プログラム変換の最中，一時的に，コードを GADT で保持することが可能である [5]．これに，TDB を表現する GADT を適用することにより，本フレームワークは設計されている．ソースコード 5 にあるのが，GADT として TDB 表現を採用したファンクタである．DBLib がファンクタ名で，F が引数のモジュール (今回は HOAS シグネチャを実装したもの) である．

ソースコード 5　TDB を採用するファンクタ

```
module DBLib(F : THOAS.THOAS) = struct
  module X = struct
    type 'a f = 'a F.exp
    type ('g, 'd) ctx =
      | CtxZ : (unit, 'd) ctx
      | CtxS : 'a typ * ('ctx, 'd) ctx
                -> ('a -> 'ctx, 'd) ctx

    type 'd funs = EMPTY : unit funs
    type 'a exp =
      {contents : 'c 'd.
        ('c, 'd) ctx ->
        'd funs ->
        ('c, 'd, 'a) tTerm}
    type 'a t =
      | Term : 'a exp -> 'a t
      | Ukn : 'a f -> 'a t

    ...

    let fwd : type a. a f -> a t =
      fun t -> Ukn t
```

```
  let bwd : type a. a t -> a f =
    function
    | Term({contents = c}) ->
       toHOAS TENil CtxZ TFNil (c CtxZ
            EMPTY)
    | Ukn t -> t
  end
  open X
  module IDelta = struct
    let tlam t f = (* to TLam  *)
    let tapp x y = (* to TApp  *)
    ...
  end
end
```

'a t は，このファンクタ内でのコードの表現を表している．ここに'a exp を使用可能にすることで，TDB をファンクタ内の主要な表現として採用する．TDB から HOAS への変換は，toHOAS に記述し，HOAS から TDB への変換は，各構文を構成する関数 (tlam, tapp 等) の関数定義として記述する．具体的な TDB・HOAS 相互変換は，[2] を元に実装を行ったため詳細は省くが，例をもとにアルゴリズムについて簡単に述べる．

tlam の実装として，$\lambda x : \tau_1.\lambda y : \tau_2.\mathbf{cons}(x,y)$ を例に考える．HOAS では，tlam t1 (fun x -> tlam t2 (fun y -> cons x y)) という関数を含む項である．これを TDB の表現すなわち TLam(t1, TLam(t2, Cons(TVar(S Z), TVar(Z)))) に変換する場合を考える．tlam の変換は，x や y に何を代入すればよいかという問題が主である．今回 HOAS の各変数は'a exp 型である．したがって，変数が登場する位置のコンテクスト ('c, 'd) ctx を参照することが可能である (これを j とおく)．tlam のコンテクスト (i とおく) を記録しておけば，適切なインデックスを両者の差分 $(j - i)$ により計算することができる．例えば，ここでの x は $1 - 0 = 1$，y は $1 - 1 = 0$ となる．

逆に，TLam(List(t), Case(t, TVar(Z), TVar(Z), TVar(S Z))) を HOAS に変換するプロセスは以下の通りである．

1. TLam(List(t), ...) を tlam (List(t)) (fun x -> ...) に変換し，x を先頭の環境に入れる．
2. Case(t, TVar(Z), TVar(Z), ...) の TVar(Z), は環境から引かれ x に置換される．
3. Case(..., TVar(S Z)) は，1, 2 と同様に case ... (fun h tl -> tl) に変換される．

TDB を用いるファンクタにはすべて，この相互変換を実装しなければならないが，本研究では，これを分離することに成功した．TDB を用いて最適化を記述する開発者は，本ライブラリを用いることで，記述したいプログラム変換のみに注目するコーディングを行うことができる．

3.2 使用法

ソースコード 6 にその使用例を示す．このファンクタでは，スーパーコンパイラの変換手続きを TDB 上で行う SC.sc という関数を，sc というタグのある位置から適用するという処理を記述している．DBLib ファンクタの中に実装された TDB・HOAS 間の変換を include 命令を使って展開する．HOAS の構造を意識することなく TDB での処理を書くことができている．

ソースコード 6　最適化実装の一例

```
module SC(F : THOAS.THOAS) = struct
  module Lib = LibTDB.DBLib(F)
  module X = struct
    include Lib.X
  end
  open X
  module IDelta = struct
    include Lib.IDelta
    let sc (Term({contents = body})) =
      let t =
       fun i funs ->
       let unifier =
         Shift.ctx_to_unifier i in
       let memo =
         SC.ctx_to_memo i
       in SC.sc
            unifier
            memo
            i
            (TDB.FNil)
            (body i funs)
      in Term({contents = t})
  end
end

module PSC(F : SYMOBS) = struct
  module OptM = SC(F)
  include SYMT(OptM.X)(F)
  include OptM.IDelta
```

```
end
```

なお，自由変数が増えてしまうような変換を記述した場合，ホスト言語の型検査により静的に発見することができる．例として，自由変数が1つだけ許されている環境において，その変数にインデックス1を代入するような変換を書く．以下のソースコード7は，その変換をscというタグのある位置から実行した例である．substitute0は，インデックス0に対して代入を行う関数である．

ソースコード 7　型エラーになる変換 (ctx_over.ml)

```
let sc body  =
  match body with
  | Term({contents = body}) ->
     let cnt : type c d.
                       (c, d) ctx ->
                       d funs ->
                       (c, d, 'a) tTerm =
       function
       | CtxS(_, _) as i->
          fun funs ->
          let term = body i funs in
(* 24 *)  substitute0 term (TVar(S Z))
       | i -> fun funs -> body i funs
     in
     Term({contents = cnt})
```

このコードは，以下のようにホスト言語の型検査を通過しない．インデックス1が現在の環境では許されていないからである．

ソースコード 8　型エラーメッセージ

```
File "over_ctx.ml", line 24, characters
    39-40:
Error: This expression has type ('a -> '
    b, 'c, 'a) TDB.index
       but an expression was expected of
            type ($1, d, $0) TDB.index
       Type 'a -> 'b is not compatible
           with type $1
```

3.3　TDBによる各種操作

3.3.1　構文的解析

TDBは一階のデータ構造のため，パターンマッチが可能である．例えば，$\lambda x : \tau_1 . \lambda y : \tau_2 . \mathbf{cons}(x,y)$ は，HOASでは，`tlam t1 (fun x -> tlam t2 (fun y -> (cons x y)))` であり，`tlam` の第2引数は関数である．$\lambda x : \tau_1$. より内部にどのような項があるかは，関数 `fun x -> ...` に適用を行わない限り調べることができない．一方，TDBにおいては，`TLam(t1, TLam(t2, Cons(TVar(S Z), TVar(Z))))` となり，ホスト言語の通常のパターンマッチにより構文を調べることができる．

3.3.2　変数の使用不使用判断

あるコードにおいて，各変数が使用されているかどうかを判定することが必要とされる場面がある．例えば，ソースコード9のループを含むコードについて考える．

ソースコード 9　変換前

```
for i in item do
  let x = 10! in
  job x i;
done
```

`x = 10!`は不変式であるからlet束縛を持ち上げ，ソースコード10のプログラムに変換したい．

ソースコード 10　変換後

```
let x = 10! in
for i in item do
  job x i;
done
```

そのためには，`x = 10!`に変数iが登場しないことを判定する必要がある．この判定は，TDBにおいて素直に実装が可能である．

出力として，次のusedというデータ構造をおく．

ソースコード 11　出力のデータ構造

```
type 'g used =
 | Empty : unit used
 | Used : 'a typ * 'g used
           -> ('a -> 'g) used
 | NUsed : 'a typ * 'g used
           -> ('a -> 'g) used
```

このデータ構造は，インデックスに対応したリストの構造をしている．外側が若いインデックスである．例えば，1,3が使用されている場合，これに対応する出力は `NUsed(t0, Used(t1, NUsed(t2, Used(t3, Empty))))` となる．型パラメータに型環境`'g`を取っており，インデックスの増減に対して，不適切な操作が行われた場合，ホスト言語の型検査により発見で

きる．

各インデックスの使用不使用を判定した結果を出力する関数 vsearch の一部は次のようになる．

ソースコード 12　使用不使用判定関数

```
let rec vsearch : type g a d. g used ->
    (g, d, a) tTerm -> g used =
  let rec write : type g a d. g used ->
      (g, d, a) index -> g used =
    fun u i ->
    match u, i with
    | NUsed(t, u), S i ->
        NUsed(t, write u i)
    | Used(t, u), S i ->
        Used(t, write u i)
    | NUsed(t, u), _ -> Used(t, u)
    | _, _ -> u
  in
  let pick : type g a. (a -> g) used ->
      g used =

    function
    | NUsed(_, u) -> u
    | Used(_, u) -> u
  in
  fun u term ->
  match term with
  | TVar(i) -> write u i
  | TApp(f, arg) ->
     let u' = vsearch u arg in
     vsearch u' f
  | TLam(t, body) ->
     pick (vsearch (NUsed(t, u)) body)
 ...
```

3.3.3　代入

代入は，HOAS においては想定することが難しい操作である．そもそも HOAS のコードは全て閉じていることが条件であるから，代入対象の変数を考えることがないからである．例えば，$\lambda y.\lambda x.\mathbf{cons}(x,y)$ に，$x = \mathbf{true}$ を代入するというのは不適切な操作である．一方，前述のようにスーパーコンパイラにおいては，開いた項を扱い，自由変数に対して代入を行う必要もある．以後の実装は，依存型をもつ言語による TDB の実装 [6] を元に構成したものである．

まず，代入する値を受け取り，インデックス 0 に代入を行う関数を生成する関数 (mapto0) の型は次のソースコード 13 である．

ソースコード 13　mapto0 の型

```
mapto0 : type a b g d.
           (b -> g, d, b) tTerm ->
           (b -> g, b -> g, d) inx_term
```

戻り値にある inx_term は，インデックスから項への関数を保持するデータ型である．次のように定義されている．

ソースコード 14　inx_term

```
type ('g1, 'g2, 'd) inx_term =
  {mapto : 'a. ('g1, 'd, 'a) index ->
               ('g2, 'd, 'a) tTerm}
```

次に，インデックス 1，インデックス 2，··· に代入するためにはこの戻り値 (inx_term) を持ちあげればよい．この持ち上げる関数の型は次の通りである．

ソースコード 15　exts の型

```
 exts : type g1 g2 b d.
   (g1, g2, d) inx_term
   -> b typ
   -> (b -> g1, b -> g2, d) inx_term
```

第 2 引数にあるように，持ち上げる型が判明しなければならない．TDB がチャーチスタイルを採用している理由の 1 つである．

4　スーパーコンパイラ

スーパーコンパイラ [9] は，既に判明している変数の値と，分岐などから得られる情報に基づくプログラムの最適化の技術である．得られた情報よりプログラムの記号的実行を進めて，もとのプログラムより高性能な残余プログラムを生成する．今回実装を行うスーパーコンパイラは [7] に則ったものである．一例としてソースコード 16 を考える．

ソースコード 16　変換前

```
case ss of
  [] -> false
  (s':ss') -> if s' = p
              then f ss
              else true
```

3 行目の ss は，$s' = p$ が真である分岐の中にあり，また，case 式における分解可能な場合に位置するため，より詳しい表現にすることができる．実際にこの式に対して，変換を行うと以下のソースコード 17 が

得られる．

ソースコード 17　変換後

```
case ss of
  [] -> false
  (s':ss') -> if s' = p
              then f cons(p, ss')
              else true
```

このようにして得られた情報を用いて，処理を進めて case 式や if 式を展開していく．また，関数があった場合は定義を展開する．再帰的な関数の場合は，プログラム変換が停止しなくなってしまうことをさけるため，folding という操作で展開を停止する．例えば，2 つのリストを結合する関数 *append* に $\mathbf{cons}(x{,}xs)$ と ys を適用した後，スーパーコンパイラに適用した場合を考える．*append* の関数定義が，2.4 節のものに従う場合，変換後は，ソースコード 18 になる．ただし，`let f x = t in f y` は，$\mathbf{fix}(\lambda f : \tau_1 \to \tau_2.\lambda x : \tau_1.t : \tau_1 \to \tau_2)\ y$ の糖衣構文である．

ソースコード 18　変換後の結合関数

```
let h1 xs ys = cons(x, h2 xs ys)
and
h2 xs ys =
  case xs of
    [] -> ys
    | (h:tl) -> cons(h, h2 tl ys)
in h1 xs ys
```

新たに，h1，h2 という関数を相互再帰的に定義し，元の再帰の構造を畳み込んでいる．この手続きにより，スーパーコンパイラは停止性を保っている．

4.1　実装にあたっての困難点

スーパーコンパイラの実装にあたっては、本フレームワークにおいても実装が困難な操作が存在した．

4.1.1　小さなインデックスへの代入

3.3.3 節にある実装では，大きいインデックスを小さいインデックスに置き換えるような代入を記述することはできない．例えば，$\lambda x : bool.y$ に対して，$y = x$ という代入を適用するのは，想定されていなかった．しかしながら，スーパーコンパイラにおいてはソースコード 17 のように，$ss = \mathbf{cons}(p{,}ss')$ を実行する必要が生じる．今回は部分的に負のインデックスを導入することでこの問題を解決した．一方で，負のインデックスは不適切に扱うとコード生成時にエラーを生じてしまう問題がある．

4.1.2　相互再帰関数定義

スーパーコンパイラにおいては，ソースコード 18 のように相互再帰の関数定義を動的に拡張する必要がある．関数定義の拡張は，現在プログラム変換中のスコープとは異なる場所に定義をしなければならないという点で難しい．また，引数の自由変数を発見して新しく定義する関数の引数にセットするという操作も同時に行う．これらの操作は HOAS から TDB に変換してもなお，困難なものであるが，次の生成方法に変更することで folding 操作を実装した．

再帰関数のスコープ　関数をコピーして変換途中のスコープの内部に配置する

HOAS における任意の相互再帰の関数　HOAS に戻す際に，相互再帰から入れ子の再帰関数に変更する

引数の自由変数を関数の引数にセット　引数に関わらず全ての自由変数を関数の引数にセットする

この生成方法に則ると *append* $\mathbf{cons}(x{,}xs)\ ys$ は，ソースコード 19 に変換される．

ソースコード 19　本方式の結合関数の変換例

```
 let h1 x xs ys = cons(x,
  (let h2 x xs ys =
   case xs of
    [] -> ys
    (h:tl) -> cons(h, h2 x tl ys)
  in h2 x xs ys))
in h1 x xs ys
```

4.2　実験の総括

実験の成果としては，変数操作を含むプログラム変換のプロセスを実際に TDB 上で実装することができることがわかったという点である．構文的解析や代入操作および，関数定義などは，HOAS 上では実装が難しいものであったが，TDB に変換することで，平易に実装することができた．本フレームワークで実装されたスーパーコンパイラは，前述の妥協点はあった

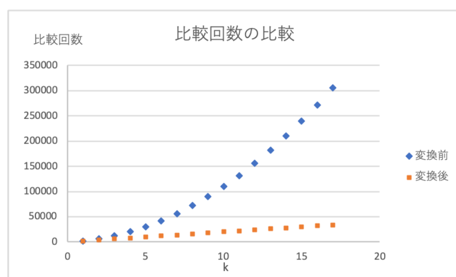

図 4　部分文字列照合の最適化 (比較回数)

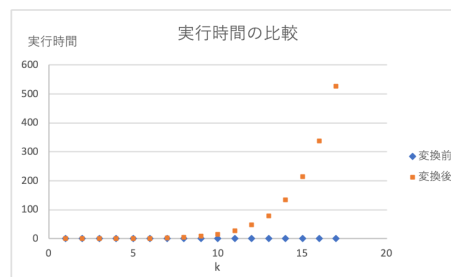

図 5　部分文字列照合の最適化 (実行時間)

ものの，最適化プロセスにおいて型を保存すること，スコープの誤りやインデックスの取り違えが発生しないことを保証できている．

最後に，実装したスーパーコンパイラについて，性能の検討を行ったものを付記する．スーパーコンパイラは，部分文字列照合のプログラムについて，照合したい文字列に特化したコードを生成することができる．特化したコードが KMP アルゴリズムに従うものになっているかは，スーパーコンパイラの性能を評価する上の指標となる．

照合文字列を $A^k B$，検索をかける文字列を $A^{1000k} B$ として，変換前と変換後で比較を行なった．

図 4 は文字の比較回数についての結果である．変換前の比較回数は非線形に増大しているが，変換後の比較回数は線形に推移していることが見て取れる．

比較回数で見ると変換による効率化が実現できていることが分かる．

一方で，全ての自由変数を引数にセットするように変更したことが影響し，現時点で実行速度の面では，望ましい性能には至っていない (図 5)．

5　まとめと今後の課題

本研究が提案するフレームワークは，タグレスファイナル法を拡張し，より変数の扱いが多様なプログラム変換をサポートできるようにすることを目指している．従来のタグレスファイナル法に則りつつも TDB を中間表現として用い，全体として型安全性を保持しつつ変換を記述できるライブラリを実装した．このフレームワークにより、HOAS では困難であったいくつかの操作を平易に実装できることを示した．フレームワークの記述力を調査するため，スーパーコンパイラを実装を試みた．実装に際して，変換のアルゴリズムを変更しなければならない点や，変換の安全性について一部妥協しなければならない点が生まれた．今後の課題としては，実装できたスーパーコンパイラについて，求められる性能が出ているか検討することと，WHOAS など他の埋め込み手法についても同様の方式で導入できないかといった点である．

参 考 文 献

[1] Asai, K.: Extracting a Call-by-Name Partial Evaluator from a Proof of Termination, *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM 2019, New York, NY, USA, Association for Computing Machinery, 2019, pp. 6167.

[2] Atkey, R., Lindley, S., and Yallop, J.: Unembedding domain-specific languages, *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, ACM, 2009, pp. 37–48.

[3] Carette, J., Kiselyov, O., and Shan, C.: Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages, *J. Funct. Program.*, Vol. 19, No. 5(2009), pp. 509–543.

[4] Kiselyov, O.: λto SKI, Semantically, *Functional and Logic Programming*, Gallagher, J. P. and Sulzmann, M.(eds.), Cham, Springer International Publishing, 2018, pp. 33–50.

[5] Kiselyov, O: Embedding and optimizing domain-specific languages in the typed final style, http://cufp.org/2015/t4-oleg-kiselyov-dsl-in-typed.html.

[6] Kokke, W., Siek, J. G., and Wadler, P.: Programming language foundations in Agda, *Science of Computer Programming*, Vol. 194(2020), pp. 102440.

[7] Srensen, M. H., Glck, R., and Jones, N. D.: A positive supercompiler, *Journal of Functional Programming*, Vol. 6, No. 6(1996), pp. 811838.

[8] Suzuki, K., Kiselyov, O., and Kameyama, Y.:

Finally, Safely-extensible and Efficient Language-integrated Query, *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '16, New York, NY, USA, ACM, 2016, pp. 37–48.

[9] Turchin, V. F.: The Concept of a Supercompiler, *ACM Trans. Program. Lang. Syst.*, Vol. 8, No. 3(1986), pp. 292325.