限定継続命令 shift/reset を用いた モナドトランスフォーマおよび探索アルゴリズムの実装

理学専攻・情報科学コース　金子ちひろ

1　はじめに

継続とは，プログラムの実行時における「残りの計算」を表す概念である．shift/reset [1] は限定継続を扱う命令の一つであり，さらに型システムを備えていることから，プログラムの実行順序に対して強力な制御能力を持つ．近年では OchaCaml [5] などの開発環境が整ってきたこともあり，shift/reset のさらなる応用が期待されている．本研究では，新たに実用性の高い応用として，モナドトランスフォーマ [4] および探索アルゴリズムの実装を行った．両者は汎用的なライブラリとして，ユーザ（プログラマ）に提供できる．このうち，本稿では前者について述べる．

Filinski [2, 3] はモナドを shift/reset で表現することで，従来のモナド形式によるエフェクトが，直接形式のプログラムで扱えることを示した．本稿は，直接形式のプログラムでエフェクトを組み合わせる手法として，Filinski の表現に対するモナドトランスフォーマを提案する．提案手法のアイデアは，継続の戻り値（アンサータイプ）となっているモナドに着目することで，トランスフォーマをアンサータイプの変換として表現することである．このアイデアに基づき，各種のトランスフォーマについて実装を行った．

2　shift/reset

shift/reset は限定継続を扱う命令である．実行時の継続を切り取る shift に対して，reset は継続の範囲を限定することができる．二つの命令を関数として捉えた場合には，次のような型を持つ．

```
val shift : ((’a -> ans) -> ans) -> ’a
val reset : (unit -> ans) -> ans
```

ans で示した型をアンサータイプと呼び，shift で切り取る継続のアンサータイプは，対応する reset の型に等しくなる．この制限を表すため，shift/reset に対応する型システムでは，関数の型を次のように表す．

```
t1 / ans1 -> t2 / ans2
```

t1 と t2 は，通常の引数と戻り値の型を表す．ans1 と ans2 は，関数が呼び出される前後でのアンサータイプの変化を表す．

3　モナド

モナドは，関数型言語において各種の副作用（エフェクト）を抽象化する手法である．一般に，モナドの実装に対するシグネチャは以下のように与えられる．

```
module type Monad = sig
  type ’a m  (* モナドを表現する型 *)
  val return : ’a -> ’a m
  val bind : ’a m -> (’a -> ’b m) -> ’b m
end
```

本稿では，モジュールであるモナドとは別に，抽象型 m（また，その値）をモナドと呼ぶことがある．

3.1　モナド形式

モナドを使用するには，return と bind の操作を用いて，関数を以下の形式（モナド形式）で記述する．

1. 通常の戻り値に対して return を用いる
2. 関数呼び出しの際，bind に継続を渡す

例として，次のプログラムは $f(x,y) = g(x) + h(y)$ をモナド形式で記述したものである．

```
let f x y = bind (g x) (fun a ->
            bind (h y) (fun b ->
            return (a + b)))
```

3.2　エフェクトの種類

return と bind だけでは，実際にエフェクトを使用することはできない．各モナドは，エフェクトを扱うための関数（operation）を提供しており，モナド形式で用いることができる．代表的なモナドの種類を表 1 に示す．

モナド	エフェクト	関数
Id	なし	–
State	状態変数	get, put
Error	エラー処理	error
Cont	継続渡し形式	callcc
Env	環境変数	local, ask

表 1: モナドの種類

4　モナドトランスフォーマ

表 1 のモナドは，いずれも単一のエフェクトを扱うモナドであった．モナドトランスフォーマは，既存のモナドを拡張することで，エフェクトを組み合わせる手法である．例えば，ErrorT は任意のモナドにエラー処理を加えるトランスフォーマである．同様に，各エフェクトに対応した StateT, ContT などがある．

4.1　モナドの変換

トランスフォーマの実装に対するシグネチャを以下に示す．任意のトランスフォーマは，モナドの変換に対する操作として lift を提供する．本稿では，Base で示した変換元のモナドをベースモナドと呼ぶ．

```
module type MonadT =
  functor (Base : Monad) ->
sig
  module M : Monad  (* 変換後のモナド *)
  val lift : ’a Base.m -> ’a M.m
end
```

4.2　リフト

モナドを変換する際に，ベースモナドのエフェクトを，変換後のモナドに「持ち上げる」ための手法をリフトと呼ぶ．一般的に，リフトは関数の戻り値に lift を用いることで実現できる．例えば，ベースモナドが

State モナドである場合，その関数 get と put に対するリフトは次のように行う．

```
let get () = lift (Base.get ())
let put s = lift (Base.put s)
```

5 Filinski の表現

これまでに紹介した実装は，いずれもモナド形式を前提とするものであった．一方，エフェクトの実装に shift/reset を用いると，関数のプログラムを直接形式で記述することができる．例えば，3.1 節の $f(x, y)$ に対するプログラムは次のようになる．

```
let f x y = g x + h y
```

Filinski は，エフェクトを直接形式で実装するための操作として，reflect と reify を定義した．

```
module Represent (M : Monad) = struct
  let reflect m =
    shift (fun k -> M.bind m k)
  let reify t =
    reset (fun () -> M.return (t ()))
end
```

reflect は shift を用いて，bind に実行時の継続を渡す操作である．また，reify は reset を用いて，初期継続に return を与える操作である．

6 提案手法

直接形式で実装されたエフェクト（モナド）に対するトランスフォーマを，以下の方針で実装した．

6.1 アンサータイプ

提案手法における変換のアイデアとして，直接形式の関数では，モナドの型（m）がアンサータイプとなることに着目する．表 2 は，モナド形式と直接形式で関数の型を比較したものである．モナド形式では戻り値の型がモナドになっているのに対して，直接形式ではアンサータイプとして現れている様子がわかる．

関数	型（上：モナド形式，下：直接形式）
get	unit -> state m unit / 'z m -> state / 'z m
put	state -> unit m state / 'z m -> unit / 'z m
error	unit -> 'a m unit / 'z m -> 'a / 'z m

表 2: 関数の型（比較）

6.2 変換対象の定義

Filinski の手法をもとに，変換対象とするモナドの実装を，次のように定義した．

```
module type DirectMonad = sig
  type 'a m  (* モナドを表現する型 *)
  type ('a, 'z) run
  val reflect : 'a m / 'z m -> 'a / 'z m
  val reify : (unit / 'a m ->
               'a / 'a m) / 'z -> 'a / 'z
  val run : (unit / 'a m -> 'a / 'a m) / 'z
            -> ('a, 'z) run / 'z
end
```

6.3 提案手法の定義

モナド形式のトランスフォーマが提供する lift は，関数の戻り値を変換するものであった．提案手法では，これに対する直接形式の操作として，アンサータイプを変換する direct_lift を提供する．

```
module type DirectMonadT =
  functor (Base : DirectMonad) ->
sig
  module M : DirectMonad
  type 'a lift  (* Base.m のパラメータ *)
  val direct_lift :
    (unit / 'z lift Base.m
     -> 'a / 'z lift Base.m) / 'z M.m
    -> 'a / 'z M.m
end
```

direct_lift は，ベースモナドの関数（サンク）を受け取り，アンサータイプを変換して実行する．direct_lift を使用したリフトは，次のように行う．

```
let get () =
  direct_lift (fun () -> Base.get ())
```

7 実行例

提案手法に従い，各種のトランスフォーマを実装した．以下のプログラムは，Id モナドをベースとして，StateT, ErrorT, StateT の順に変換したものである．

```
(* モナドの変換 *)
module T1 = StateT (Int) (IdMonad)
module T2 = ErrorT (T1.M)
module T3 = StateT (Int) (T2.M)
module St1 = T3.LiftState(T2.LiftState(T1.Op))
module Err = T3.LiftError(T2.Op)
module St2 = T3.Op
module Monad = T3.M
(* 使用例 *)
let f () = if St1.get () = St2.get ()
           then St2.put (St2.get () + 1)
           else Err.error ()
(* 実行 *)
let test1 = Monad.run f 1 1 (* (1, Ok (1, 0)) *)
let test2 = Monad.run f 1 0 (* (0, Error) *)
```

8 まとめと今後の課題

本研究では，直接形式でエフェクトを組み合わせる方法として，shift/reset によるモナドトランスフォーマを提案した．実装した direct_lift は，従来とは異なる手法（アンサータイプの変換）を用いており，正しくリフトを実現することを証明する必要がある．

参考文献

[1] Olivier Danvy and Andrzej Filinski. Abstracting Control. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pp. 151–160, June 1990.

[2] Andrzej Filinski. Representing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT synposium on Pronciples of programming languages*, pp. 446–457, 1994.

[3] Andrzej Filinski. Representing layered monads. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 175–188, 1999.

[4] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 333–343, January 1995.

[5] Masuko Moe and Kenichi Asai. Caml light + shift/reset = caml shift. In *Theory and Practice of Delimited Continuations (TDPC 2011)*, pp. 33–56, May 2011.