shift/reset による
モナドトランスフォーマの提案と実装

金子 ちひろ，浅井 健一

お茶の水女子大学

kaneko.chihiro@is.ocha.ac.jp

asai@is.ocha.ac.jp

概 要　近年、OchaCaml などの開発環境が整ってきており、shift/reset のさらなる応用が期待されている。Filinski 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。本論文では、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。本論文のアイデアは、直接形式で継続の戻り値（アンサータイプ）となっているモナドに着目し、トランスフォーマの機能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 OchaCaml による実装を行った。今後の課題として、提案手法の性質および正当性について議論する。

1　序論

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] は継続を扱うための命令であり、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。そのため、例外処理や入出力、非決定性プログラミングなど、さまざまな応用が考えられてきた。近年では、OchaCaml [7] 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。

Filinski [3] は、shift/reset を用いてモナド [8] が表現できることを示した。この表現を用いると、従来のモナド形式によるエフェクトを、直接形式で記述することが可能となる。

本論文では、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。図 1 のプログラムは、提案手法を用いて、二つの状態とエラー処理を組み合わせたものである。

Liang ら [6] の提案したモナドトランスフォーマは、既存のモナドを変換することで、徐々にエフェクトを拡張するための手法である。ただし、この手法はモナド形式を前提としているため、従来の変換を直接形式に「翻訳」するだけでは、効率が悪くなってしまう。そこで、本論文は継続の戻り値となっているモナドに着目することで、shift/reset による「アンサータイプの変換」を提案する。このアイデアに基づき、各種のトランスフォーマ（StateT, ErrorT, ContT, EnvT）について実装を行った。

一般に、モナドトランスフォーマは、Haskell のクラスを用いた実装（MTL：Monad Transformer Library）が知られている。本論文では、提案手法の実装（OchaCaml）と比較するために、OCaml のモジュールおよびファンクタを用いて表現する。

本論文の構成は、以下のようになる。まず 2 節では、shift/reset についての簡単な説明を行う。3 節から 5 節までは、既存の手法として、モナド、モナドトランスフォーマ、および Filinski の手法を紹介する。6 節で提案手法であるアンサータイプの変換を述べた後、7 節でそれに従った実装を示す。最後に、8 節では提案手法の性質について議論し、9 節で関連研究を、10 節で本論文のまとめを述べる。

```
(* エフェクトの合成 *)
module T1 = StateT(Int)(IdM)   (* T1 = int 型の状態を導入 *)
module T2 = ErrorT(T1.M)       (* T2 = エラー処理を導入 *)
module T3 = StateT(Int)(T2.M)  (* T3 = int 型の状態を導入 *)
module St1 = T3.LiftState(T2.LiftState(T1.Op))  (* T1 のエフェクトを T3 にリフト *)
module Err = T3.LiftError(T2.Op)                (* T2 のエフェクトを T3 にリフト *)
module St2 = T3.Op                              (* T3 のエフェクト *)
module Monad = T3.M                             (* T3 のモナド *)

(* 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) *)
```

図 1. 使用例

2 shift/reset

継続を扱う命令の多くは、実行時の継続を関数として扱うことができる。例えば、`1 + f () * 2` というプログラムにおいて、`f ()` の継続は「戻り値に 2 を掛けて 1 を足す」である。これを関数として表したものは `(fun a -> 1 + a * 2)` となる。

Danvy らが提案した `shift/reset` では、実行時の継続を切り取る `shift` に対して、`reset` で継続の範囲を限定することができる。OchaCaml の構文では、`shift (fun 変数 -> 式)` とすることで実行時の継続を束縛し、式を実行する。また、`reset (fun () -> 式)` は、式における継続を `reset` の内側に限定する。すると、例えば

```
1 + reset (fun () -> 2 * shift (fun k -> k 3 + 4))
```

というプログラムにおいて、`k` に束縛される継続は、`(fun a -> 2 * a)` となる。

二つの命令を関数として捉えた場合には、以下のような型を持つ。ここで、`answer` で示した継続の実行結果（戻り値）の型をアンサータイプと呼ぶ。

```
val shift : (('a -> answer) -> answer) -> 'a
val reset : (unit -> answer) -> answer
```

`shift` で切り取った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` の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。一方、`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、純粋な関数型言語のプログラムで、さまざまな「副作用」を抽象化する手法である。本論文では、モナドによる副作用をエフェクトと呼ぶ。

モナドはエフェクトを表すような型として定義することができる。本論文で扱うモナドの種類と、その定義を表 1 に示した。任意の計算（`'a`）に対して、`state`, `err` などのエフェクトが加えられている様子が分かる。以下では、このようなモナドの型を一般に `m` で表す。

表 1. モナドの種類

モナド	エフェクト	定義
Id モナド	なし	`'a`
State モナド	状態変数	`state -> state * 'a`
Error モナド	エラー処理	`'a err`
Cont モナド	継続渡し形式	`('a -> answer) -> answer`
Env モナド	環境変数	`env -> 'a`

```
(* Id モナド *)
module IdMonad : Monad = sig
  type 'a m = Id of 'a
  let return x = Id x
  let bind (Id x) k = k x
end

(* State モナド *)
module StateMonad : Monad = sig
  type state = int
  type 'a m = state -> state * 'a
  let return x = fun x -> (s, x)
  let bind m k = fun s0 -> let (s1, a) = m s0 in k a s1
end
```

図 2. モナドの実装（Monad）

3.1 実装

OCaml のシグネチャによる定義を示す。以降では、型であるモナド（m）とは別に、この実装（モジュール）をモナドと呼ぶことがある。

```
module type Monad = sig
  type 'a m
  val return : 'a -> 'a m
  val bind : 'a m -> ('a -> 'b m) -> 'b m
end
```

任意のモナドは、型（'a m）に対する操作として return と bind を提供する。さらに、この二つは Monad Laws と呼ばれる性質を満たす必要があるが、ここでは割愛する。例として、Id モナドと State モナドの実装を図 2 に示す。

3.2 モナド形式

プログラムでモナドを使用するには、関数を次のように記述する。

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

このようなプログラムの形式をモナド形式（monadic style）と呼ぶ。例えば、以下は $f(x, y) = g(x) + h(y)$ をモナド形式で記述したものである。

```
(* State エフェクト：get, put *)
val get : unit -> state m        (* 状態に値を書き込む *)
val put : state -> unit m        (* 状態の値を取り出す *)

(* Error エフェクト：error *)
val error : unit -> 'a m         (* エラーを発生させる *)

(* Cont エフェクト：callcc *)
val callcc : (('a -> 'b m) -> 'a m) -> 'a m

(* Env エフェクト：local, ask *)
val local : env -> 'a m -> 'a m  (* 環境に値を設定する *)
val ask : unit -> env m          (* 環境の値を取り出す *)
```

図 3. エフェクトに対するシグネチャ

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

このように、モナド形式では頻繁に bind を使用しなければならない（Haskell では bind を略記するための構文が用意されている）。そのため、エフェクトを導入する際に、プログラム全体をモナド形式に書き換えるのは大変な作業である。一方で、プログラムを一度モナド形式で記述すると、その後は使用するモナドを変更するだけで、さまざまなエフェクトを扱うことができる。

3.3 エフェクトの実装

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。Id モナドを除く各モナドは、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。各エフェクトに対するシグネチャを図 3 に示す。

4 モナドトランスフォーマ

前節で紹介したモナドは、いずれも単一のエフェクトを扱うものであった。ここでは、モナド形式でエフェクトを組み合わせる方法として、Liang のモナドトランスフォーマについて述べる。モナドトランスフォーマは、既存のモナドを変換することにより、新たなエフェクトで拡張することができる。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。本論文では、変換元のモナドをベースモナドと呼ぶ。

トランスフォーマを使用すると、モナドを繰り返し変換することで、必要なエフェクトを組み合わせることができる。例えば、Id モナドを StateT と ErrorT で変換したモナドは、状態とエラーの両方を扱うことができる。また、StateT を繰り返し用いることで、複数の状態を備えたモナドを生成することも可能である。

トランスフォーマは、モナドの型の変換として定義することができる。本論文で扱うトランスフォーマの種類と、変換後のモナドの定義を表 2 に示した。ここで、定義中の m は任意のベースモナドを表す。例えば、Id モナドを StateT, ErrorT の順で変換したモナドは `state -> (state * 'a err)` となる。また、ErrorT, StateT の順で変換したモナドは `state -> (state * 'a) err` となる。

表 2. モナドトランスフォーマの種類

トランスフォーマ	エフェクト	定義（変換後のモナド）
StateT	状態変数	`state -> (state * 'a) m`
ErrorT	エラー処理	`('a err) m`
ContT	継続渡し形式	`('a -> answer m) -> answer m`
EnvT	環境変数	`env -> 'a m`

4.1 実装

モナド（モジュール）に対するトランスフォーマは、ファンクタとして実装することができる。以下では、ベースモナドの実装を `Base`、変換後のモナドの実装を `M` で表す。

```
module type MonadT =
  functor (Base : Monad) ->  (* ベースモナドの実装 *)
sig
  module M : Monad           (* 変換後のモナドの実装 *)
  val lift : 'a Base.m -> 'a M.m
end
```

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。`lift` は、ベースモナドの計算（`'a Base.m`）を変換後のモナド（`'a M.m`）に埋め込む関数である。例として、StateT の実装を図 4 に示した。StateT では、上記の変換に加えて `get` と `put` を提供する。

モナドの実装に対する Monad Laws と同様に、`lift` が満たすべき性質として Monad Transformer Laws がある。以下の二つの等式は、`Base.return` と `Base.bind` を使用した任意の計算が、変換後のモナドで（その意味を変えずに）実現できることを示す。このように、ベースモナドの計算を、変換後のモナドに「持ち上げる」ことをリフトと呼ぶ。

```
lift (Base.return x)  =  M.return x
lift (Base.bind m k)  =  M.bind (lift m) (fun a -> lift (k a))
```

4.2 リフト

トランスフォーマでは、`Base.return` と `Base.bind` のほかに、ベースモナドのエフェクトをリフトする必要がある。一般に、ベースモナドに対する関数を `Base.op` で表すと、エフェクトのリフトを次のように実現することができる。

```
module LiftOp = struct
  let op arg = lift (Base.op arg)
end
```

モナド形式では、関数の戻り値がモナド（`Base.m`）になっているため、それを `lift` が変換後のモナド（`M.m`）に埋め込むことでリフトが実現できる。例外として `Base.callcc` と `Base.local` については、引数の部分にもモナドが現れているため、この手法では正しくリフトすることができない。このような関数に対しては、個別にリフトを定義する必要があるが、本論文では割愛する。

```
(* StateT : MonadT *)
module StateT (Base : Monad) = struct
  module M = struct
    type state = int
    type 'a m = state -> (state * 'a) Base.m
    let return x = fun s -> Base.return (s, x)
    let bind m k = fun s0 -> Base.bind (m s0) (fun (s1, a) -> k a s1)
  end
  let lift m = fun s -> Base.bind m (fun x -> Base.return (s, x))
  module Op = struct  (* State エフェクト *)
    let get () = fun s -> Base.return (s, s)
    let put s = fun _ -> Base.return (s, ())
  end
end
```

図 4. StateT の実装

5 Filinski の表現

これまでに紹介した手法は、いずれもモナド形式を前提としたものであった。本節では、モナドによるエフェクトを直接形式（direct style）で表現する方法として、Filinski の Represent について述べる。

5.1 直接形式

モナド形式では、関数呼び出しの際に `bind` に継続を渡していることから、一種の継続渡し形式（CPS：continuation passing style）と捉えることができる。直接形式とは、`shift/reset` などの継続命令を用いることで、継続（ここではモナド）を使用する関数を、通常の関数と同様に記述する手法である。例えば、3.2 節の $f(x, y) = g(x) + h(y)$ を直接形式で記述したものは、次のようになる（ここで、計算は左から実行されるとする）。

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

5.2 Represent

Represent は、モナド形式で実装されたモナド（`return` と `bind`）に対して、`reflect` と `reify` の二つの操作を定義する。`reflect` は実行時の継続を `bind` に渡す操作であり、`reify` は初期継続として `return` を与える操作である。

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

この二つの操作を用いると、エフェクトを直接形式の関数として実装することができる。例として、State モナドに対するエフェクトは次のように実装できる。

```
open Represent(StateMonad)  (* 直接形式の表現 *)
module StateOp = struct     (* エフェクトの実装 *)
  let get () = reflect (fun s -> (s, s))
  let put s = reflect (fun _ -> (s, ()))
end
let suc () = put (get () + 1)          (* 使用例 *)
let test = reify (fun () -> suc ()) 0  (* 実行 = (1, ()) *)
```

```
(* Id モナド *)
module IdM : DirectMonad = sig
  type 'a m = Id of 'a
  type ('a, 'z) run = 'a m
  val reflrect (Id x) = shift (fun k -> k x)
  val reify t = reset (fun () -> Id (t ()))
  val run t = reify t
end
```

図 5. モナドの実装（DirectMonad）

直接形式によるエフェクトは、reflect を用いて実装される。また、プログラムを実行する際には reify を使用する。

6 提案手法

本論文の目的は、直接形式のプログラムで、各種のエフェクトを組み合わせることである。そのための手法として、直接形式で実装されたモナドに対するトランスフォーマを提案する。

提案手法の実装には OchaCaml を用いた。OchaCaml は Caml Light の拡張であるため、トランスフォーマの表現に必要なファンクタ機能を備えていない。本論文では、OchaCaml における実装を、ファンクタとして表現しなおしたものを示す。

6.1 変換対象の定義

従来のモナドの実装（Monad）は、モナド形式の操作として return と bind を提供していた。提案手法が変換するモナドは、Filinski の表現に基づき、直接形式の操作として reflect と reify、および run を提供するものとする。このようなモナドに対するシグネチャとして、以下のような DirectMonad を定義する。ここで、run はプログラムを実行する際に、状態変数（State モナド）などの初期値を与えるための操作である。

```
module type DirectMonad = sig
  type 'a m           (* 従来と同じ型 *)
  type ('a, 'z) run  (* run の実行結果の型 *)
  val reflect : 'a m / 'z m -> 'a / 'z m
  val reify : (unit / 'a m -> 'a / 'a m) / 'z -> 'a m / 'z
  val run : (unit / 'a m -> 'a / 'a m) / 'z -> ('a, 'z) run / 'z
end
```

shift/reset の型システムにより、関数の型にはアンサータイプを指定する必要がある。ここで、reflect が bind に渡す継続を切り取ることや、reify が初期継続として return を与えることなどから、アンサータイプ（継続の戻り値）はモナドになることがわかる。'z で表したパラメータは、アンサータイプが多相であることを示す。この定義に従い、Id モナドを実装したものを図 5 に示した。提案手法では、このような DirectMonad から DirectMonad への変換を行う。

同様に、各エフェクトに対するシグネチャを図 6 に示す。モナド形式の関数と比較すると、直接形式の関数では戻り値からモナドを除く代わりに、アンサータイプをモナドに指定する。ここで、callcc や local のパラメータの変化については、関数の挙動（エフェクトの及ぶ継続の範囲）を考慮して指定したものである。

このように、直接形式で実装されたエフェクトでは、関数のアンサータイプがモナドとなる。すると、モナドの変換に対する操作として、アンサータイプの変換を考えることができる。

```
(* State エフェクト *)
val get : unit / 'z m -> state / 'z m
val put : state / 'z m -> unit / 'z m

(* Error エフェクト *)
val error : unit / 'z m -> 'a / 'z m

(* Cont エフェクト *)
val callcc : (('a / 'a m -> 'b / 'a m) / 'a m -> 'a / 'a m) / 'z m -> 'a / 'z m

(* Env エフェクト *)
val local : env / 'z m -> ((unit / 'a m -> 'a / 'a m) / 'z m -> 'a / 'z m) / 'z m
val ask : unit / 'z m -> env / 'z m
```

図 6. 直接形式のエフェクト

6.2 従来のリフト

アンサータイプの変換を述べる前に、従来のリフト手法（lift）を直接形式で用いることを考える。以下のトランスフォーマは、変換対象とするモナドの実装を、従来のモナド形式（Monad）から直接形式（DirectMonad）に変更したものである。

```
module type D_MonadT =
  functor (Base : DirectMonad) ->  (* ベースモナドの実装 *)
sig
  module M : DirectMonad           (* 変換後のモナドの実装 *)
  val lift : 'a Base.m -> 'a M.m   (* 従来のリフト操作 *)
end
```

モナド形式では、lift が変換するモナド（Base.m）は、関数の戻り値によって得られていた。一方、直接形式の関数に lift を用いるには、アンサータイプからモナドを取り出さなければならない。reify を使用すると、初期継続を実行することで、アンサータイプを戻り値として得ることができる。すなわち、ベースモナドに対する関数（Base.op）を以下のように実行する。

```
Base.reify (fun () -> Base.op arg)
```

したがって、この戻り値に対して従来のリフト手法を用いることができる。最後に、reflect を使用して再び直接形式にすることで、直接形式におけるリフトが次のように実現できる。

```
module LiftOp = struct
  let op arg = M.reflect (lift (Base.reify (fun () -> Base.op arg)))
end
```

この変換はほぼ自明であるが、モナドを戻り値にするなど、モナド形式を経由した変換と考えられる。また、そのために reify および reflect を使用しており、効率が悪く感じられる。

6.3 アンサータイプの変換

lift はモナド形式を前提としているため、直接形式で使用するには余分な処理が生じてしまう。そこで、提案手法では直接形式でリフトを実現する操作として、新たにアンサータイプに着目した direct_lift を提供する。提案手法に対するシグネチャを以下に示す。

```
module type DirectMonadT =
  functor (Base : DirectMonad) ->  (* ベースモナドの実装 *)
sig
  module M : DirectMonad           (* 変換後のモナドの実装 *)
  type 'a lift                     (* Base.m のパラメータ *)
  val direct_lift :                (* アンサータイプの変換 *)
    (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。また、この定義に伴って導入している 'a lift （従来の lift とは無関係）は、変換後のモナドにおいて Base.m のパラメータを表す型とする。direct_lift を用いたエフェクトのリフトは以下のように行う。

```
module LiftOp = struct
  let op arg = direct_lift (fun () -> Base.op arg)
end
```

注意点として、従来のリフト（lift）が関数の戻り値を Base.m から M.m に変換していたのに対し、direct_lift を用いたリフトでは、サンクを実行する間のアンサータイプを M.m から Base.m に変換する。すなわち、型の変換が従来とは逆方向になっている。direct_lift の詳しい動作については、次節の実装において述べる。

7 実装

提案手法（DirectMonadT）に従い、StateT, ErrorT, ContT, EnvT について実装を行った。ここではその一部を紹介し、shift/reset を用いたアンサータイプの変換について述べる。

7.1 モナドの変換

提案手法に対するシグネチャ（DirectMonadT）に従い、直接形式で実装されたモナドを変換する。変換後のモナドである M については、従来のトランスフォーマによる実装（モナド形式）をもとに、以下の方針で実装した。

- 型の変換には、従来と同じ定義を用いる
- reify では、初期継続となっている return を変換する
- reflect は、bind の変換における継続を shift/reset で表現する
- 新たに run に対する変換を定義する（reify の変換に類似）

StateT における M の実装を図 7 に示す。注意点として、StateT ではモナドの型（M.m）が関数となるため、この部分に対してアンサータイプの指定が必要となる。ここでは、モナド自身に対するアンサータイプは、戻り値と同じ型とする。

```
(* 変換後のモナド *)
module M = struct
  type 'a m = state / (state * 'a) Base.m
              -> (state * 'a) Base.m / (state * 'a) Base.m
  type ('a, 'z) run = state / 'z -> (state * 'a, 'z) Base.run
  let reflect m =
    shift (fun k -> fun s0 ->
      let base = reset (fun () -> m s0) in
      let (s1, a) = Base.reflect base in k a s1)
  let reify t =
    fun s -> Base.reify (fun () -> shift (fun return ->
      let return2 = fun s -> return (s, x)
      in reset (fun () -> return2 (t ())) s))
  let run t s = Base.run (fun () -> reify t s)
end
```

図 7. 変換後のモナド（StateT）

7.2 direct_lift の実装（StateT, ContT, EnvT）

アンサータイプの変換を実装するには、モナドの型の構造に注目する。

StateT, ContT, EnvT による変換後の型（M.m）は、いずれもベースモナドに引数を加える形で定義されている。そこで、ベースモナドのパラメータを 'a lift で表すと、変換を次のように表すことができる（アンサータイプは省略する）。

```
type 'a m = arg -> 'a lift Base.m  (* 左辺の m は M.m *)
```

この変換に対して、direct_lift を次のように定義する。

```
let direct_lift t = shift (fun k -> fun arg -> k (t ()) arg)
```

以下は、このプログラムに対する説明である。

実行するサンクが返す型を res と置くと、shift が切り取る k の型は res -> 'z M.m となっている。すると、k (t ()) が変換後のモナド（'z M.m）となるのに対して、引数を与えた k (t ()) arg はベースモナド（'z lift Base.m）になることがわかる。ここで、サンクを実行する間の継続は (fun x -> k x arg) であるから、その戻り値（アンサータイプ）はベースモナドとなる。

また、t () の実行後に k が呼び出されると、継続は再び k の内部に限定される（これは、shift/reset の性質による）。すなわち、サンクの実行後にアンサータイプは 'z M.m に戻される。

このように、direct_lift は継続の範囲を一時的に変更することで、アンサータイプの変換を実現することができる。

7.3 direct_lift の実装（ErrorT）

同様に、ErrorT ではモナドの変換が次のように表せる。

```
type 'a m = 'a lift Base.m  (* 'a lift = 'a err である *)
```

すると、この変換に対する direct_lift は次のように定義できる。

```
let direct_lift t = t ()
```

この定義で、direct_lift は単にサンクを実行するだけである。ErrorT では、変換後のモナド（'z M.m）がそのままベースモナド（'z lift Base.m）になっているため、アンサータイプを変換する必要がないのである。すなわち、ErrorT では direct_lift を用いたリフトは不要である。

7.4 その他の実装

各トランスフォーマでは、以上の定義に加えて、エフェクトおよびリフトを提供する。これらを含めた StateT の完全な実装を図 8 に示す。この実装では、`reflect` と `reify` の変換の一部を補助関数（`reflect_state`, `run_state`）として定義している。また、状態の型（`state`）を `State.t` として外部から受け取るようにした。受け取る型の例は、

```
module Int = struct type t = int end
```

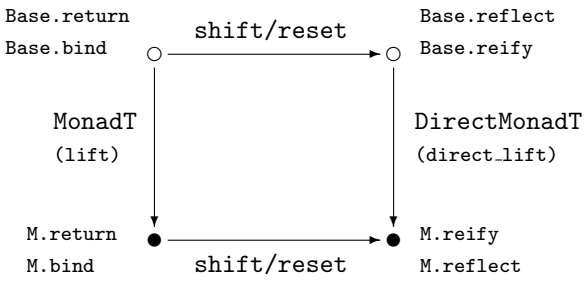
である。また、リフトの実装における `StateOp(Base).Type` などの表記は、ベースモナドに対するエフェクトのシグネチャを表す。なお、`Base.callcc` と `Base.local` に対しては、モナド形式と同様に、個別にリフトを実装した。

8 議論

8.1 正当性

正当性は今後の課題である。`direct_lift` による変換は、従来の手法とは大きく異なっているため、正しくリフトを実現していることを示すには証明が必要となる。また、実装した `direct_lift` は特定の型構造に着目して行ったものであり、任意のトランスフォーマに対して「アンサータイプの変換」を実装する手法は確かではない。

正当性を示す方法としては、提案手法（`DirectMonadT`）の位置付けが、以下となることの証明が考えられる。



上記の図において、変換の合流性があることを示す。すなわち、任意のベースモナドについて、従来のトランスフォーマで変換した後に `shift/reset` で表現したモナドと、`shift/reset` で表現したものを提案手法で変換したモナドが（何らかの意味で）等価であることを証明すればよい。

また、Liang は `lift` が満たす性質として Monad Transformer Laws を示しており、これを `direct_lift` が何らかの形で満たすことを示すべきである。`direct_lift` と `lift` の関係については、現時点で次のようなものを仮定している。

```
direct_lift t  =  M.reflect (lift (Base.reify t))
```

このような課題について、今後のアプローチとしては、以下による証明を考えている。

1. CPS 変換 [2]
2. Monad Laws による等価性
3. 部分評価、簡約
4. `shift/reset` の公理 [5]

```
module StateT (State : sig type t end) (Base : DirectMonad) =
struct
  type 'a lift = State.t * 'a

  let reflect_state f =
    shift (fun k -> fun s0 -> let (s1, a) = f s0 in k a s1)
  let run_state t s =
    shift (fun k -> reset (fun () ->
      let x = t () in fun s2 -> k (s2, x)) s)

  module M = struct
    type 'a m =
      State.t / 'a lift Base.m
      -> 'a lift Base.m / 'a lift Base.m
    type ('a, 'z) run = State.t / 'z -> ('a lift, 'z) Base.run
    let reflect m = reflect_state (fun s -> Base.reflect (reset (fun () -> m s))
    let reify t = fun s -> Base.reify (fun () -> run_state t s)
    let run t s = Base.run (fun () -> run_state t s)
  end

  module Op = struct
    type state = State.t
    let get () = reflect_state (fun s -> (s, s))
    let put s = reflect_state (fun _ -> (s, ()))
  end

  let direct_lift t = shift (fun k -> fun s -> k (t ()) s)

  module LiftState (Base : StateOp(Base).Type) = struct
    type state = Base.state
    let get () = direct_lift (fun () -> Base.get ())
    let put s = direct_lift (fun () -> Base.put s)
  end

  module LiftError (Base : ErrorOp(Base).Type) = struct
    let error () = direct_lift (fun () -> Base.error ())
  end

  module LiftCont (Base : ContOp(Base).Type) = struct
    let callcc f = reflect_state (fun s ->
      let f2 cont2 =
        let cont x = reflect_state (fun s2 ->
          cont2 (run_state (fun () -> x) s2))
        in run_state (fun () -> f cont) s
      in Base.callcc f2)
  end

  module LiftEnv (Base : EnvOp(Base).Type) = struct
    type env = Base.env
    let local t =
      reflect_state (fun s ->
        Base.local r (fun () -> run_state t s))
    let ask () = direct_lift (fun () -> Base.ask ())
  end
end
```

図 8. StateT の実装（提案手法）

8.2 直感的な理解

ここでは、以上とは別の角度から direct_lift の動作に対する直感的な理解を示す。

直接形式では、アンサータイプがモナドとなっている。また、shift/reset の型システムより、アンサータイプは対応する reset の型を表すものである。すると、直接形式の表現では、reset がモナドに対応すると考えられる。これを視覚的に表現するため、reset に囲まれたコンテキストを < ... > で表すと、あるモナドを用いて、適当なエフェクト f () を実行するコンテキストは以下のようになる。

```
< ... f () ... >
```

このコンテキストにおいて、f は shift を用いることで、reset までの継続（コンテキスト）を切り取ることが可能である。また、この動作により f は reset（すなわち、モナド）が表すエフェクトにアクセスすると考える。

これに対して、提案手法のトランスフォーマは、新たに内側に reset を導入する。

```
< ... < ... f () ... > ... >
```

すると、内側の reset が新しいエフェクトに、外側の reset が古い（ベースモナドの）エフェクトに対応する。ここで、f が古いエフェクトを使用することを考えると、shift が切り取る継続は内側の新たな reset に限られてしまい、外側の reset にアクセスすることができない。

すると、direct_lift の直感的な理解は、内側にある reset を抜けて、外側のエフェクトにアクセスするための手法である。すなわち、実行の様子は次のようになる。

```
    < ... < ... direct_lift f ... > ... >
→   < ... ... f () ... ... >  (* ここで f が shift を実行 *)
```

9 関連研究

直接形式でエフェクトを組み合わせる別の方法として、Filinski が Layered モナド [4] を提案している。これも shift/reset を用いてエフェクトを重ねる手法であるが、トランスフォーマとは合成方法が異なっており、各エフェクト（モナド）が独立に近い形で定義されている。

10 まとめ

本論文では、直接形式でエフェクトを組み合わせる手法として、モナドトランスフォーマによるアンサータイプの変換を提案した。実際に各種のトランスフォーマを実装し、動作する様子を確認している。今後の課題は、8 節で述べたような性質を証明することである。

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] Olivier Danvy and Andrzej Filinski. Representing Control, a Study of the CPS Transformation. *Mathematical Structures in Computer Science*, Vol. 2, No. 4, pp. 361–391, December 1992.

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

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

[5] Yukiyoshi Kameyama and Masahito Hasegawa. A Sound and Complete Axiomatization of Delimited Continuations. In *In Proc. of 8th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'03*, pp. 177–188. ACM Press, 2003.

[6] 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.

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

[8] Philip Wadler. The Essence of Functional Programming. In *Conference Record of the 19th Annual ACM symposium on Principles of programming languages*, pp. 1–14, January 1992.