仕様記述を可能にした継続モナドにおける関数の内部証明

窪田 唯花 (指導教員：浅井 健一)

1 はじめに

限定継続を含む体系は、CPS 変換されたものについて意味論や正当性の研究が多くなされている。しかしその体系において、作成したプログラムの挙動についての証明を扱ったものは少ない。そこで、限定継続を含む体系での証明を実現することを目指した。CPS 変換されたプログラムとモナドと呼ばれる構造は類似点があり、モナドを用いた内部証明は継続の入らない別のモナドで実現されている [2]。そのため、継続を含む体系でも内部証明が可能な構造を、型システム [1] を参考にした上でモナドを用いて定義した。そして限定継続を用いた関数の一例について、定義と同時に内部証明をすることに成功した。

なお、本論文で扱うコードは定理証明系言語 Agda で記述されている。

2 限定継続

継続とは、「その後の計算」のことである。例えば `1 + 2 * 3 - 4` という式を計算するとき、最初にするべき計算は乗算の `2 * 3` であり、このときの継続は「計算結果に 1 を足して 4 を引く」、つまり `1 + [.] - 4` という計算になる。継続の中でも範囲が限定されているものを限定継続と呼ぶ。限定継続にはいくつか種類があるが、本論文では継続を取り出す関数として `shift`、継続の範囲を限定する関数として `reset` を用いる構造を扱う。

具体例として、`reset (1 + shift (λ k → 0) * 3) - 4` という計算を考える。`shift` は `reset` までの継続を取り出すため、`k` には `shift` 全体を `[.]` とした `reset` までの継続 `reset (1 + [.] * 3)` (関数の形で書くと `λ x → reset (1 + x * 3)`) が入る。その状態で `shift` の中身を実行すると、継続の情報を捨てた上で `0` が返る。最後に `reset` の外側にある `- 4` と `0` を計算し、全体では `-4` が答えとして返る。

このように、現在行なっている計算の周りの計算を自由に扱うことができるのが限定継続の大きな特徴である。

3 モナド

3.1 モナドとは

副作用をもつ計算を扱う構造をモナドと呼ぶ。モナドは `return` と `bind` という関数を中心に構成されている。`return` は要素をモナドに変換するもので、`bind` は式の実行順序を指定するものである。これらの関数の型は同一だが、その型自体の定義と関数の中身を変化させることで、さまざまな副作用をもつ計算を扱うことができる。

3.2 継続モナド

モナドのうち、副作用として継続を扱うものを継続モナドと呼ぶ。その構成は以下のようになる。

```
Monad : (A : Set) → Set
Monad A = (A → S) → S

return : ∀ {A : Set} (x : A) → Monad A
return x = λ cont → cont x

bind : ∀{A B : Set} (x : Monad A)
       → (f : A → Monad B) → Monad B
bind x f = λ cont →
       x (λ v → f v (λ v2 → cont v2))

reset : (x : Monad S) → Monad S
reset x = λ cont → cont (x (λ v → v))

shift : ∀{A : Set} (f : (A → Monad S)
       → Monad S) → Monad A
shift f = λ k → f (λ v →
       (λ cont → cont (k v))) (λ x → x)
```

`S` は答えの型で、本論文では自然数のリストとしている。`Monad` は任意の型 `A` を受け取るような型で、引数が `A` 型、返り値が `S` 型になる継続を受け取り、最後に `S` 型の要素を返す。`return` は `A` 型を持つ要素 `x` を受け取り、`A` 型のモナドにして返す。`bind` は最初の計算 `x` と、その値を受け取って続きの計算を行うような関数 `f` を受け取り、`x` を先に計算して結果に `v` と名前をつけ、`v` を使って残りの計算 `f` を行う。`reset` は引数として `S` 型のモナドを受け取り、その計算と周りの継続との間に恒等継続を入れる。`shift` は切り取ってくる継続 `k` から `S` 型のモナドを返すような関数 `f` を受け取り、`A` 型のモナドを返す。いずれの関数についても最初に継続を受け取っている点が継続モナドの特徴であり、CPS 変換との類似性として挙げられる。

本論文では、性質を証明したい関数として、自然数を要素として持つリストを受け取り要素の順を反転させたリストを返す関数 `rev` を、限定継続 `shift` と `reset` を用いて以下のように定義する。

```
rev : List N → Monad (List N)
rev [] = return []
rev (x :: xs) =
  shift (λ k → bind (rev xs)
              (λ l → bind (k l)
              (λ l' → return (x :: l'))))
```

引数のリストが空であった場合は、空のリストを `return` の引数として返す。リストが要素をひとつ以上持つ場合、まず `shift` で周りの継続を `k` として受け取り、引数の最初の要素を除いた残りのリストについて再帰する。再帰した関数から返ってきたリストを継続 `k` に渡して実行し、最後にそのリストの先頭に引数の最初の要素を結合して `return` の引数として返す。

このような関数 `rev` を実行する際、継続の範囲を示す `reset` は最初に 1 つだけあればよい。そのため、実行時は以下のように周りに `reset` をつけたものを使う。

```
rev2 : (l : S) → Monad S
rev2 l = reset (rev l)
```

例えば、この関数にリスト `1::2::[]` と初期継続 `(λ x → x)` を渡すと `2::1::[]` が返ってくる。

4 PCont モナド

4.1 定義

3 節における継続モナドの定義を、内部証明を可能な形に改良する。まず、もとのモナドの型 (A→S)→S のそれぞれを証明が可能な型に変更する。

```
Pre : (A : Set) → Set1
Pre A = A → Set

Post : Set1
Post = S → Set
```

Pre には関数の返り値自体が示す性質、Post にはそれを継続に渡した後の返り値が示す性質を記述する。Set は命題の型でもあるため、この部分に任意の命題を記述することができる。

その上で、モナドの型を以下のように変更する。

```
PCont : (A : Set) → Pre A →
        Post → Post → Set
PCont A pre post1 post2 =
   ((Σ [ a ∈ A ] pre a) →
    (Σ [ s ∈ S ] post1 s)) →
   (Σ [ s ∈ S ] post2 s)
```

基本は継続を受け取って値を返すものになっており、3 節のモナドの定義と似た構成である。

return、bind、reset、shift の定義は以下のようになる。

```
return : {A : Set} → (x : A) →
         {t : Pre A} → (t x) →
         {α : Post} → PCont A t α α
return x tx k = k (x , tx)

bind : {A B : Set} → {t1 : Pre A} →
       {t2 : Pre B} → {α β γ : Post} →
       (PCont A t1 β γ) →
       ((x : A) → t1 x → PCont B t2 α β)
       → PCont B t2 α γ
bind c1 c2 k =
 c1 (λ { (a , t1a) → c2 a t1a k})

reset : {t2 : Pre S} → {t α : Post} →
        PCont S t2 t2 t → PCont S t α α
reset x k = k (x (λ v → v))

shift : {A : Set} → {t : Pre A} →
      {α β γ δ : Post} →
      (f : ((x : A) → (t x) →
      PCont S α δ δ) → PCont S γ γ β)
      → PCont A t α β
shift f k = f (λ x tx cont →
           cont (k (x , tx))) (λ x → x)
```

PCont の引数として新たに増えた 3 箇所の対応関係は、型システムを参考に導出している。

4.2 rev の内部証明

上記の定義を用いて、3 節で定義した rev 関数について、「関数の実行前後でリストの長さが変化しない」という性質を証明する。PCont モナドは内部で証明を行うことができるため、rev の定義時に返り値となる PCont モナドの引数として性質を記述する。

```
rev : (l acc : List N) →
    PCont (List N)
    (λ l' → length' {N} [] ≡ length' l')
    (λ l' → length' acc ≡ length' l')
    (λ l' → length' l + length' acc
                      ≡ length' l')
rev [] acc = return [] refl
rev (x :: xs) acc =
  do' (λ p → p) (λ p → p)
  (λ p → rev-h xs acc p)
  (shift (λ k → bind (rev xs (x :: acc))
      (λ l1 → λ t1l1 → bind (k l1 t1l1)
      (λ l2 → λ t1l2 →
      return (x :: l2) (cong suc t1l2)))))
```

引数が空のリストの際は、3 節の定義とほぼ同一である。リストが要素をひとつ以上持つ場合も、shift から先は 3 節の定義とほぼ同様のものになっている。

ここで使用している do' は、PCont モナドの各種の関数から導かれる性質と rev で実際に証明したい性質の橋渡しをしている。do' の引数は、順に PCont の pre、post1、post2 についての関係を表している。

関数内部で引数として渡されている t1l1、t1l2 はそれぞれ、引数 l1、l2 が bind の第一引数で行った操作の性質を満たしていることの証明が入る。これにより、bind で行った最初の計算の性質を次の計算に引き継ぐことができている。

3 節の rev との大きな違いとして、引数に新たに acc を加えた点が挙げられる。前述の性質を証明する際には「一番最初に rev に渡したリストの長さ」という全体の情報が必要となる。しかし 3 節で定義した引数のみでは、関数が再帰したときにその情報を持ち歩くことができないため、「既に見終わった要素の数」を表す acc を引数として導入した。acc は関数内部では他の引数と干渉しないため、関数の挙動自体には関与していない。

5 まとめと今後の展望

型システムを基礎に限定継続と命題の入ったモナドを定義し、関数の内部証明を実現した。モナドは CPS 変換と近い形をしているため、CPS での証明や、最終的には直接形式で限定継続の入った体系の証明をするための足がかりになると考える。

しかし現状の課題として、証明部で書くべき命題に本来関数の実行には関与しない acc を導入する必要があり、まだ直感的な証明ができる状態になっていないことが挙げられる。また、今回は 1 つの例でのみ証明を行ったため、別の関数でも同様に内部証明ができるか検証することも必要と考えられる。

参考文献

[1] Olivier Danvy and Andrzej Filinski. A functional abstraction of typed contexts. BRICS 89/12, August 1989.

[2] Wouter Swierstra. A Hoare Logic for the State Monad. *International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009)*, pp. 440–451, August 2009.