shift/reset を含む関数の篩型を使った内部検証

理学専攻・情報科学コース　窪田 唯花

1　はじめに

限定継続制御演算子を含む体系では、実行している式の外側にある計算を操作することができる。そのため、限定継続制御演算子を含む関数を定義する際には実行時の文脈を正確に把握し記述する必要がある。これにより、再帰などを含む複雑なプログラムを記述する場合には作成したプログラムの挙動が期待されたものであるか直感的に理解することが難しい場合がある。

本研究では、限定継続制御演算子 shift/reset を含む関数についてその性質を内部検証することを目的とする。先行研究では、高階プログラム論理を拡張した体系で証明を実現したもの [3] や、SMT ソルバを用いて自動証明を実装したもの [2] が挙げられる。本研究は CPS 変換と実行順序を規定する継続モナドに性質を記述できる篩型を使用することで、shift/reset の型システム [1] を利用した性質の伝播を実現した。

なお、本研究で扱うプログラムは定理証明支援系言語 Agda で記述されている。

2　限定継続

継続とは、プログラム中のある式を実行している時点での残りの計算のことである。$1+2*3-4$ という式を実行する場合、最初に実行する乗算の周りに残されている計算 $1+[.]-4$ が現在の継続となる。限定継続は、継続の中でも範囲が限定されたものを指す。

継続の記述方法は、実行時環境を hole [.] で表す場合と継続そのものを関数として表す場合の大きく 2 通りが挙げられる。上記の例の継続を関数の形で表すと $\lambda x.\, 1+x-4$ となる。

shift/reset は、継続を関数の形で束縛して扱うことができる限定継続制御演算子の一種である。reset 演算 $\langle\rangle$ は、継続の範囲を限定する。shift 演算 $\mathcal{S}k.\, e$ は、その時点での reset までの継続を関数の形で切り取り変数 k に束縛した上で本体部分 e を実行する。本体部分 e の内部では束縛した継続 k を関数として扱うことができるため、種々の複雑なプログラムを比較的簡潔に記述できる。継続を本体内部で使用せず捨てることでエラー処理を実現したり、複数回使用することでバックトラックを実現することができる。また、継続の外側に別の計算を記述することで計算の順序を入れ替えることもできる。

shift/reset は継続を切り取る際や切り取った継続を呼び出す際に、次に shift を実行したときの継続が変化しないよう設計されている。そのため他の限定継続制御演算子に比べて型理論や健全性・完全性に加え多くの公理が証明されている。

3　CPS 変換と継続モナド

3.1　CPS 変換

shift/reset について議論する際は、一般に継続渡し形式 (Continuation-passing style; CPS) に変換して行われる。CPS は継続を明示的に扱う形式のことで、継続を関数の形で受け渡すように記述するため shift/reset が実装されていない言語においても同等の表現力を実現できる。しかし、特に shift と reset に対する CPS 変換 $[\![\ ]\!]$ は以下のように複雑であり、関数を記述する際に直接 CPS で記述することは適切でない。

$$[\![\mathcal{S}k.\, e]\!] \Rightarrow \lambda k.\, [\![e]\!]\,(\lambda v.\, \lambda k_1.\, k_1\,(k\,v))$$

$$[\![\langle e\rangle]\!] \Rightarrow \lambda k.\, k\,([\![e]\!]\,(\lambda x.\, x))$$

3.2　継続モナド

モナドは副作用を持つ計算を扱う構造のことである。同一の型についてモナドの型定義と関数の本体部分を変更することで種々の副作用計算を記述できる。モナド一般に定義される関数として、値をモナドの形に持ち上げる `return` と計算順序を規定する `bind` が存在する。

継続モナドは、CPS 変換を主な副作用として持つモナドを指す。継続モナドの構造を以下に示す。

```
1  Monad : (A B C : Set) → Set
2  Monad A B C = (A → B) → C
3
4  return : {A B : Set} (x : A) →
5    Monad A B B
6  return x = λ cont → cont x
7
8  bind : {A B C D E : Set} →
9    (Monad A B C) → (A → Monad D E B) →
10   Monad D E C
11 bind x f = λ cont → x (λ v →
12               f v (λ v2 -> cont v2))
13
14 reset : {A B C : Set} → Monad A A B→
15            Monad B C C
16 reset x = λ cont → cont (x (λ v → v))
17
18 shift : {A B C D E : Set} →
19           ((A → Monad B E E) →
20             Monad D D C) → Monad A B C
21 shift f = λ cont → f (λ v → λ k →
22             k (cont v)) (λ x → x)
```

`Monad` は継続モナドの型を表す。CPS は `A → B` 型の継続を受け取り `C` 型の値を返すため、`Monad` の型も同様に定義されている。`return` は値に対する CPS 変換、`bind` は let による実行順序の決定（k-正規形変換）を CPS 変換したものを表している。`reset`、`shift` は 3.1 節の CPS 変換と同一の変換を定義している。

また各関数の型は CPS 変換式から容易に導かれる。

4　篩型を用いた関数表現

篩型は通常の型を命題で修飾したような型であり、プログラムの性質を記述する際に用いることができる。基本型にのみ命題が追加され、関数型はその引数と返り値にそれぞれ命題がつく。

4.1　定義

Agda 上で篩型を定義したものを以下に示す。

```
  data Ty : Set₁ where
    Int : (p : ℕ → Set) → Ty
    IntList : (p : List ℕ → Set) → Ty
    _⇒_ : Ty → Ty → Ty

  ⟦ _ ⟧ : (A : Ty) → Set
  ⟦ Int p ⟧ = Σ[ n ∈ ℕ ] p n
  ⟦ IntList p ⟧ = Σ[ l ∈ List ℕ ] p l
  ⟦ A ⇒ B ⟧ = ⟦ A ⟧ → ⟦ B ⟧
```

Ty は篩型で受け取る引数の種類を示す。本研究では基本型として自然数と自然数のリストを扱うため、それぞれの要素について命題を記述できるようコンストラクタを用意する。_ ⇒ _ は関数に命題を付加したい場合に用いる。この場合は関数に対する命題ではなく、引数と返り値のそれぞれに命題を受け取る。

⟦_⟧ は、Ty 型を用いて篩型を記述したものである。これは Σ 型を用いて定義されている。Σ 型は二つ目の要素が一つ目の要素に依存できるような組を表す。ここでは、篩型を「基本型と、その基本型の値に対する命題の組」として定義している。

4.2 継続モナドと篩型

⟦_⟧ は Set 型を持っていた。Set は型そのものの型であり、自然数型 ℕ やリスト型 List A の型も Set である。Set 型を持つ値はプログラムの型定義（つまり x:A の A の部分）に記述することができる。プログラムの型に性質を追加したい場合は、通常の型部分を ⟦_⟧ で始まる型に変更することで実現できる。

継続モナドは CPS 変換を担うため、shift/reset 演算子が実装されていない言語においても直接形式と同様に使用することができた。そこで、継続モナドを用いて記述されたプログラムの型部分を適切に篩型へ変更することで、shift/reset を含むプログラムについて性質の証明が可能になると考えられる。

5 内部検証の例

以下のプログラムについて、内部検証の例を示す。

```
reset (1 + shift (λ k → 2 * k 3)) - 4
```

shift は reset までの継続 $1 + [.]$（関数 λ x → 1 + x）を k に束縛する。k は shift の本体部分 2 * k 3 で使用され、全体の結果として 4 が返ることになる。

そこで、上記のプログラムの結果が 4 になることを内部的に検証する。上記は直接形式で記述されており Agda ではそのまま記述することができないので、まず継続モナドを用いた形に変更する。

```
test : {A : Set} → Monad ℕ A A
test =
 bind (reset (bind (shift (λ k →
  bind (k 3) (λ x →
  return (2 * x))) (λ y →
  return (1 + y))))) (λ z →
  return (z - 4))
```

1 行目の自然数型を篩型に変更することにより、bind により部分式に付けられた新しい束縛変数 x,y,z と return の引数は自然数から篩型を表す組に変化する。篩型に変更することにより新たに記述が必要となる証明部分は {!!} で表す。

```
test' : {A : Set} →
    Cont ⟦ Int (λ n → n ≡ 12) ⟧ A A
test' =
 bind (reset (bind (shift (λ k →
  bind (k (3 , {!!})) (λ x →
  return (2 * proj₁ x , {!!})))) (λ y →
  return (1 + proj₁ y , {!!})))) (λ z →
  return (proj₁ z + 4 , {!!}))
```

{!!} 部分を証明で埋めることでプログラムが完成する。8 行目の証明は 2 行目の型部分から「proj₁ z + 4 が 12 と等しい」ことを記述する必要があることがわかる。つまり、z の第二要素は「proj₁ z が 8 と等しい」という証明が入っていることが考えられる。関数外部に補題 test'-lem を立てることで、2 つの命題を繋げて証明を埋めることができる。

以上のように証明を進めていくことで、以下の完成したプログラムが得られる。

```
test' : {A : Set} →
    Cont ⟦ Int (λ n → n ≡ 12) ⟧ A A
test' =
 bind (reset (bind (shift (λ k →
  bind (k (3 , refl)) (λ x →
  return (2 * proj₁ x ,
    test'-lem2 (proj₂ x))))) (λ y →
  return (1 + proj₁ y ,
    cong suc (proj₂ y))))) (λ z →
  return (proj₁ z + 4 ,
    test'-lem (proj₂ z)))
```

6 まとめと今後の課題

CPS 変換を行う継続モナドにおいて型部分を適切に篩型へ変更することで、shift/reset 演算子を含む関数の内部検証を実現した。

答えの型が変化するリストの結合関数や内部で shift を何度も実行するリストの反転などの再帰関数についても内部検証が可能となったが、現在は shift で束縛した継続を複数回使用して別の引数を渡すような関数は内部検証が成功していない。その原因として、継続の表現力が不足していることが挙げられる。篩型を用いて定義する関数は任意の述語を記述できるが、継続には外部から伝播した性質以外の内容が記述できない。また継続は束縛された後に何度も使用される可能性があるため一般化された性質を記述する必要があるが、関数定義の際に使用した非明示的な引数などの拡張を行えない。そのため、最初に継続を使用した箇所のみに成り立つような局所的な性質が適用されてしまう。

参考文献

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

[2] Fuga Kawamata, Hiroshi Unno, Taro Sekiyama, and Tachio Terauchi. Answer refinement modification: Refinement type system for algebraic effects and handlers. *Proc. ACM Program. Lang.*, Vol. 8, No. POPL, January 2024.

[3] 佐藤惇. 限定継続のための高階プログラム論理. 修士論文, 京都大学大学院情報学研究科, 2023.