遅延モナドを用いた一般再帰関数に対する等式変形による検証

川上　竜司, Jacques Garrigue, 才川　隆文

名古屋大学多元数理学研究科
{ryuji.kawakami.c3, garrigue}@math.nagoya-u.ac.jp, tscompor@gmail.com

概 要　Coq のライブラリ Monae は、モナディック等式変形を用いてプログラムの計算効果に関する検証を可能にする。現在 Monae では、状態モナドや確率モナドなど、様々なモナドをサポートすることで多様なプログラムを扱うことができるが、構造的でない再帰関数の扱いが難しい。一方で、余帰納的に定義される遅延モナドを用いると一般再帰関数を表現できることが知られている。本研究では、遅延モナドに対する while を用いた適切なインターフェイスを定義し、その健全性を形式的に証明することで、Monae を用いた一般再帰関数に対する検証を可能にした。また、モナドトランスフォーマーを用いて他のモナドと組み合わせることで、計算効果を含みうるより一般的なプログラムに対する Monae を用いた検証を可能にする。

1　初めに

純粋関数型プログラムはその参照等価性としての性質から、等式変形による検証に適している。さらには、モナドと呼ばれる構造を用いることで、計算効果を表すことができ、関数型プログラミング言語 Haskell をはじめとした様々な言語において採用されている。Gibbons らは、モナドの持つ代数的な性質に着目し、それぞれのモナドのインターフェイスを、等式の集まりとして定義することで、計算効果の持つプログラムに関する等式変形による検証、モナディック等式変形を提案した [GH11]。

Monae　定理証明支援系 Coq[The24] でモナディック等式変形を用いた検証を可能にするツールは Monae[ANS19] である。Monae は、モナディック等式変形を行うための等式の集まりであるインターフェイスと、その健全性を保証するモデルから構成される。Coq を用いることで、検証の正しさを保証し、Coq の数学ライブラリ MathComp/SSReflect[GM10] を用いた簡潔な証明が可能になる。Monae では、モナディック等式変形で必要となるインターフェイスの階層を Hierarchy Builder[CST20] を用いて実装することで、複数のモナドの組み合わせや、再利用可能な構造的な証明を可能にする。

構造的でない再帰関数の扱い　定理証明支援系では無矛盾性の保証のため、停止しない関数を定義できない。Coq の `Fixpoint`コマンドでは、引数の持つ整礎な順序関係が構文的に自明な構造的再帰関数については定義できる。そうでないときは、`Equation` コマンドなどを用いて引数が整礎な順序関係を持ち減少する値であることの証明とともに定義する必要がある。
例えば、複雑な再帰を行うマッカーシーの 91 関数は Coq では直ちに定義することはできない。

```
1  let rec mc91 m = if 100 < m then m - 10 else mc91 (mc91 (m+11))
```

さらには、停止性の未解決なコラッツ関数などは再帰関数として定義することはできない。

余帰納型を用いた一般再帰関数の定義　定理証明支援系で構造的でない再帰関数を扱う他の方法として、無限個のコンストラクターを持つデータを定義する際に用いられる余帰納的定義を用いる手法がある。Coq では、ガード制約を満たす限り、無限にコンストラクターを適用することができるため、それを用いて無条件な再帰呼び出しを行い、構造的でない再帰関数を扱うことができる。それらの関数は、戻り値が余帰納型データとなるが、停止性の証明なしに定義することができる。
Capretta らの提案した遅延モナド [Cap05] を用いるとそういった余再帰的な関数呼び出しを行うプログラムをモナディックプログラムとして表すことができる。したがって、本研究では、遅延モナドに対する適切なインターフェイスを定義することで Monae を用いた停止性の保証できない一般再帰関数に対する検証を試みた。

完全エルゴートモナド　遅延モナドのインターフェイスを定義するにあたって、完全エルゴートモナド [AMV10] を参考にした。完全エルゴートモナドは、代数的に再帰構造を扱う Iteration theory[BÉ93] に対応しており、イテレーションと呼ばれる、各 $f : X \to M(Y+X)$ を $f^\dagger : X \to MY$ に対応させるオペレーター $(\_)^\dagger$ に関する 4 つの公理を満たすモナドとして定義される。
例えば、等式 fixpoint は次の可換図式で表される。

$$\begin{array}{ccc} X & \xrightarrow{\quad f^\dagger \quad} & MA \\ \downarrow{\scriptstyle f} & & \uparrow{\scriptstyle \mu_A} \\ M(A+X) & \xrightarrow{M[\eta_A, f^\dagger]} & M^2A \end{array}$$

この規則は関数 f が A の値を返すまで繰り返し計算を行った結果が、$f^\dagger$ に等しいことを表している。
遅延モナドは有限回の計算ステップを無視する同一視を行うことで完全エルゴートモナドとなることが知られている [UV17]。

モナドトランスフォーマーを用いた計算効果の合成　遅延モナドを用いることで、一般再帰関数を扱えるが純粋な関数に限られる。例えば、参照と while 文を用いて階乗を計算する `factorial` は遅延モナドだけでは表現できない。

```
let factorial n =
  let r = ref 1 in
  let l = ref n in
  while !l != 0 do
  r := !r * !l;
  l := !l - 1;
  done;
```

そこで、本研究では、複数のモナドを変換により組み合わせることができるモナドトランスフォーマーを用いて、例外モナドや、ocaml の参照を表現するために導入された型付きストアモナド [AGS23a] と組み合わせることで、複数の計算効果をもつプログラムに対する Monae を用いた検証を可能にした。

本稿の貢献と構成　本稿の貢献は以下のようにまとめられる。

- 遅延モナドの計算的同値性に関する規則を含むインターフェイスを定義し、その健全性を形式的に示すことで、一般再帰関数に関する検証を Monae で行うことを可能にした。
- モナドトランスフォーマーを用いた他のモナドとの組み合わせや一般項書き換えを可能にする setoid ライブラリを用いることで一般再帰関数に関する Monae による検証の有用性を高めた。

以下、2 節で遅延モナドのインターフェイスの詳細ついて、3 節で型付きストアモナドとの組み合わせについて、説明する。また 4 節で関連研究について、5 節でまとめと課題について論じる。なお、本研究のコードは以下の ulr から確認することができる。

`https://github.com/Ryuji-Kawakami/monae/tree/delaypull`

2　遅延モナドの Monae における実装

2.1　遅延モナドの定義

遅延モナドは、余帰納的に定義されることで停止しない関数の計算を表現することが可能である。
遅延モナドを構成する関手 Delay は、(A:Type) → (X = A + X の最大不動点) という型

を持つ関数である。Coq では、CoInductiveコマンドと、各最大不動点への埋め込みを表すコンストラクター DNow, DLater を用いて定義することができる。

```
CoInductive Delay (A : Type) : Type :=
  |DNow : A -> Delay A
  |DLater : Delay A -> Delay A.
```

遅延モナドの return オペレーターは、コンストラクタ DNow であり、bind オペレーターはCoFixpointコマンドを用いて定義される。

```
Let ret (a:A) := DNow a
CoFixpoint bind (m: Delay A) (f: A -> Delay B ) :=
  match m with
  |DNow a => f a
  |DLater d => DLater (bind d f)
  end.
```

さて、遅延モナドに付随するオペレーターとして、繰り返し処理を行う while を定義する。while オペレーターは完全エルゴートモナドの $(\_)^\dagger$ オペレーターに相当する。
CoFixpointコマンドを用いて定義され、右埋め込みの値 `inr a` が値 a での繰り返しの継続、左埋め込みの値 `inl b` が値 b を戻り値とする繰り返しの終了を表す。

```
CoFixpoint while {A B} (body: A -> M (B + A)) : A -> M B :=
  fun a => (body a) >>= (fun ab => match ab with
                                   |inr a => DLater (while body a)
                                   |inl b => DNow b end).
```

また、while オペレーターを用いて不動点オペレーターの定義が行えることが知られている。例えば、Filinski は第一級継続を用いて、while オペレーターから不動点オペレーターを定義している [Fil94]。

2.2 計算的同値性

さて、ここで等式変形による検証は、遅延モナドを用いて表したプログラムに対しては適さない。例えば、階乗を計算する関数 fact を遅延モナドを用いて定義した場合、`fact 3` は明示的な計算ステップ DLater を 3 つ含むため、`DNow 6` と一致しないためである。したがって、計算的な同値性を表す関係が必要である。そこで、有限個の DLater を除いて等しい場合、またはどちらも DLater が無限個続く場合に計算的に等しいとみなす関係 wBisim を [Cap05] と同様に次のように導入した。
まず、計算がある値で停止する性質を Terminate という帰納的な関係で定義する。
`Terminate d a` とは、d が計算の結果、値 a を返すことである。

```
Inductive Terminates A : Delay A -> A -> Prop :=
  |TDNow a : Terminates (DNow a) a
  |TDLater d a : Terminates d a -> Terminates (DLater d) a.
```

次に、この述語を用いて、wBisim を余帰納的関係として定義する。つまり、d1,d2 が有限個の DLater を除いて等しい場合 wBTerminate d1 d2 a が成り立ち、d1,d2 がどちらも DLater が無限個続く場合は、wBLater により余帰納法を用いることで、wBisim d1 d2 を示すことができる。

```
CoInductive wBisim A : Delay A -> Delay A -> Prop :=
  |wBTerminate d1 d2 a :
    Terminates d1 a -> Terminates d2 a -> wBisim d1 d2
  |wBLater d1 d2 : wBisim d1 d2 -> wBisim (DLater d1) (DLater d2).
```

2.3 余帰納法を用いた等式の証明

ここでは、wBisim_DLater を例にどのように余帰納法の原理を用いて、wBisim に関する性質を Coq 上で示したかを説明する。

```
  Lemma wBisim_DLater A : forall (d : M A), wBisim (DLater d) d.
```

Coq では、帰納法で fixオペレーターを用いて証明項を構成するのと同様に、余帰納法では cofixオペレータを用いて証明項を構成する。つまり、この例では、
cofix CIH (d : M A) : Oeq (DLater d) d := ··· から始まる項を構成すればよい。
さて、cofixに関する型規則は以下である [Gim95]。

$$\frac{\Gamma \vdash B : Set \qquad \Gamma, f : B \vdash N : B \qquad \mathcal{C}\{f, N\}}{\Gamma \vdash \mathrm{CoFix}\, f : B := N : B}\ \mathrm{CoFix}$$

ただし、CoI を余帰納的に定義された型、P を CoI のパラメーター、$B := \Pi_{z:Z}(\mathrm{CoI}\, P\, (z))$ $\mathcal{C}\{f, N\}$ を f と N に関する条件とする。この条件は構文的ガード制約と呼ばれ、N に関して帰納的に定義される。特に次のような性質を満たす

- N はコンストラクターを含まなければならない。
- 余再帰呼び出し f はコンストラクターの内側で行われ、その外側では、場合わけとラムダ抽象しか行われていない。

さて、このことに注意して、次のように証明を行った。
cofix タクティックで、余再帰呼び出しである apply CIH は、コンストラクター wBLater の内側で行われており、その前には d に関する場合わけしか行っていないため、確かに構文的ガード制約を満たす。

```
Lemma wBisim_DLater A : forall (d : M A), wBisim (DLater d) d.
Proof.
cofix CIH => d.
case: d => [a|d'].
- apply : wBTerminate.
  + by apply/TDLater/TDNow.
  + by apply TDNow.
- apply wBLater.
  by apply CIH.
Qed.
```

2.4 遅延モナドのインターフェイス

以上のことを踏まえて、遅延モナドのインターフェイスを表 1,2,3 のように定義した。
まず、遅延モナドのオペレイターは、繰り返し処理を行う while である。また、計算の等さを表す関係である、wBisim ≈ を導入した。表 2 は wBisim に関する規則である。前半の 3 つの規則は、wBisim が同値関係であることを表している。後半の 3 つの規則は、検証上有用であると考え導入した。bindmwB は bind の引数が同じ計算結果を表しているのならば、f に渡した結果も同じ計算結果になることを表している。
bindfwB は同じ f と g が同じ計算をするならば、bind で同じ引数を渡した結果も同じ計算結果になることを表す。
whilewB は、while 文でその都度繰り返す処理が、計算的に等しいならば、while 文全体として等しいことを表す。
後半の 3 つの規則は、while と bind が同値類を保つことを表している。したがって、パラメトリックモルフィズムとして登録することで、等式変形に近い証明を可能にした。詳細については 2.5 節で説明する。
表 3 は while オペレーターに関する規則である。完全エルゴートモナドの定義を参考にした。
fixpointE は、while によって、繰り返す処理をすることができることを表す規則である。
naturalityE は、while 文により行った計算結果を、次の処理に渡す場合、それを一つの while 文により記述できることを表す。
codiagonalE は、連続して入れ子になった while 文を一つの while 文にすることができることを表す。
これらの規則を用いた等式変形では、モデルにおいて、健全性を示した時のような余帰納法を用いた証明をする必要がなく、簡潔で直感的な検証が可能になる。

表 1. オペレーターと関係

`while : (A -> M(B + A)) -> A -> M B`
`wBisim : M A -> M A -> Prop`

表 2. wBisim に関する規則

wBisim_refl	`a ≈ a`
wBisim_sym	`a ≈ b -> b ≈ a`
wBisim_trans	`a ≈ b -> b ≈ c -> a ≈ c`
bindmwB	`d1 ≈ d2 -> d1 >>= f ≈ d2>>= f`
bindfwB	`(forall a, f a ≈ g a) -> d >>= f ≈ d >>= g`
whilewB	`(forall a, f a ≈ g a) -> while f a ≈ while g a`

表 3. while に関する規則

fixpointE	`while f a ≈ (f a) >>= (sum_rect Ret (while f))`

naturalityE	`while f a >>= g ≈`
	`while (fun y => (f y) >>=`
	`(sum_rect (M # inl o g) (M # inr o Ret))) a`

codiagonalE	`while ((M # ((sum_rect idfun inr))) o f ) a ≈` `while (while f) a`

2.5 一般書き換え

ユーザーが独自に定義したパラメトリック同値関係に対する rewrite タクティックスの使用を Setoid ライブラリは可能にする。一般再帰関数に対する検証では、等式変形による検証ではなく、計算的な同値性である wBisim に関する等価性変形を行う必要がある。したがって、wBisim をパラメトリック同値関係としてインスタンス化した。

```
1  Add Parametric Relation A : (M A) (@wBisim M A)
2    reflexivity proved by (@wBisim_refl M A)
3    symmetry proved by (@wBisim_sym M A)
4    transitivity proved by (@wBisim_trans M A) as wBisim_rel.
```

インターフェイスの規則である bindmwB と bindfwB、whilewB がそれぞれ関数 bind、while が同値関係 wBisim を保つことを表している。したがって、bind、while を Parametric morphism としてインスタンス化した。定義の中に現れる
任意の値に対して関数が同じ同値類の値を返すことを表す
(pointwise_relation A (@wBisim M B)) を用いて関数の間の関係を定義する。これに

より、図 1,2 のような書き換えが可能になる。さらに、setoid_rewriteを用いることで、束縛変数を含む項に関する書き換えが可能である。このこととついては、5.3 節で具体的に説明する。

```
Add Parametric Morphism A B : bind
  with signature (@wBisim M A) ==>
    (pointwise_relation A (@wBisim M B)) ==> (@wBisim M B) as bindmor.

  Add Parametric Morphism A B : while
  with signature (pointwise_relation A (@wBisim M (B + A))) ==>
    @eq A ==> (@wBisim M B ) as whilemor.
```

これにより、Monae を用いた他の検証と同様な rewrite タクティックスを中心とした証明が可能になった。

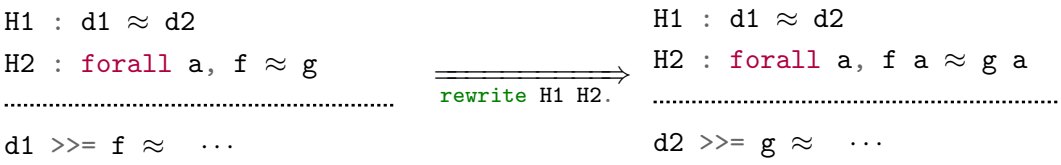

図 1. rewrite with bind

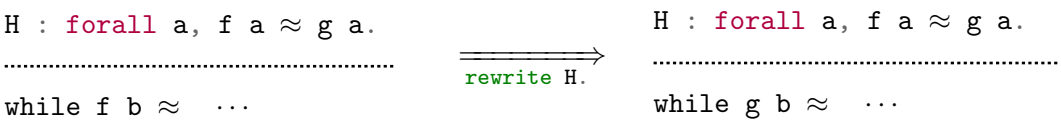

図 2. rewrite with while

2.6 マッカーシーの 91 関数に対する検証

遅延モナドのインターフェイスを用いた検証例として、再帰的に定義される関数であり、$m \leq 101$ の時、必ず 91 を返すという性質をもつマッカーシーの 91 関数 mc91 を扱う。

```
let rec mc91 m = if 100 < m then m - 10 else mc91 (mc91 (m+11))
```

$m \leq 100$ の時、11 足した値で二重に再帰するため、構造的ではない再帰関数である。したがって、mc91 関数を delay モナドを用いて表し、自然数 $m \leq 100$ について 91 が戻り値となることを示した。
まず、while オペレーターを用いて mc91 関数を用いて表す。n が残っている再帰の深さ、m が計算している値である。

```
Let mc91_body nm :=
    match nm with (n, m) =>
    if n==0 then ret (inl m)
            else if m > 100
                 then ret (inr(n.-1, m - 10))
                 else ret (inr(n.+1, m + 11))
    end.
Let mc91 n m := while mc91_body (n.+1, m).
```

mc91 関数が 91 を返すことを示す際、本質的な性質は、次の補題 mc91succE である。この補題と、mc91 n 101 ≈ Ret 91 であることと、$k = 90 - m$ に関する帰納法により従う。

```
Lemma mc91succE n m : 90 <= m < 101 -> mc91 n m ≈ mc91 n (m.+1).
```

さて、この補題を図 3 のように示した。

```
mc91 n m
 ⟦ rewrite /mc91. (*definition of mc91*) ⟧
  ≈  while mc91_body (n.+1, m)
 ⟦ rewrite fixpointE. ⟧
  ≈  (if 100 < m then Ret (inr (n, m - 10))
              else Ret (inr (n.+2, m + 11))) >>=
                   sum_rect Ret (while mc91_body)
 ⟦ rewrite ifN //. (* m ≤ 100 *)⟧
  ≈ Ret (inr (n.+2, m + 11)) >>= sum_rect Ret (while mc91_body)
 ⟦ rewrite bindretf /= fixpointE /=. ⟧
  ≈ while mc91_body (n.+2, m + 1)
 ⟦ rewrite bindretf fixpointE /=. ⟧
  ≈ (if 100 < m + 11
      then Ret (inr (n.+1, m + 11 - 10))
      else Ret (inr (n.+3, m + 11 + 11))) >>=
          sum_rect Ret (while mc91_body)
  ⟦ rewrite ltn_add2r ifT //. (* 90 ≤ m ⟹ 100 < m + 11 *) ⟧
  Ret (inr (n.+1, m + 11 - 10)) >>= sum_rect Ret (while mc91_body)
 ⟦ rewrite  bindretf fixpointE /= fixpointE. ⟧
  ≈  while mc91_body (n.+1, m + 11 - 10) = mc91 n (m+1)
```

図 3. mc91 関数に対する証明

3　型付きストアモナドとの組み合わせ

遅延モナドのインターフェイスを定義することで、一般再帰関数に対する検証が可能になった。この手法をより多くの関数へ適用するためには、遅延モナドを他のモナドと組み合わせることにより複雑な副作用を表現する必要がある。そこで、[AGS23b] において導入された型付きストアモナドとの組み合わせた遅延型付きストアモナドを定義した。

3.1　型付きストアモナド

Coqgen[GS22] により変換したコードの、参照を表すために導入されたモナドが型付きストアモナドである。
型と値のレコード binding のリストを状態として使うことで、型付きストアを表している。また参照を扱うためのオペレーターである `cnew`、 `cget`、 `cput` を持つ。

```
Record binding :=
  mkbind { bind_type : ml_type; bind_val : coq_type bind_type }.
```

本研究では、型付きストアモナドを、例外モナドトランスフォーマー `MX` と状態モナドトランスフォーマー `MS` の合成で定義された型付きストアモナドトランスフォーマーに変更し、それぞれのモナドトランスフォーマーが遅延モナドの構造を保つことを示すことで、遅延型付きモナドを定義した。

```
MS (seq binding) option_monad ⇒ MS (seq binding) (MX unit M)
```

3.2　状態モナドトランスフォーマーとの組み合わせ

状態モナドトランスフォーマー stateT は次のように定義される。
S が状態、M が変換前の関手、として `MS`,`retS`,`bindS` がそれぞれ変換後の関手、return, bind である。

```
Let MS := fun A => S -> M (A * S).
Let retS := fun A => curry Ret.
Let bindS m f := fun s => m s >>= uncurry f.
Let liftS m := fun s => m >>= (fun x => Ret (x, s)).
```

さて、stateT により状態モナドと組み合わせるためには、遅延モナドが stateT で変換後、遅延モナドであることを示す必要がある。そこで、[PG13] を参考に次のように `whileDS` と `wBisimDS` を定義した。関数 `dist1` は、分配法則を表し、型を合わせるために定義した。

```
M:delayMonad
Let DS := MS M
Let whileDS (body : X -> DS (Y + X)) :=
  curry (while (M # dist1 o uncurry body)).

Let wBisimDS (ds1 ds2 : DS A) : Prop :=
  forall s : S, wBisim (ds1 s) (ds2 s).
```

3.2.1 例外モナドトランスフォーマーとの組み合わせ

例外モナドトランスフォーマー exceptT は次のように定義される。
Z が例外の集合、M が変換前の関手、として MX,retX,bindX がそれぞれ変換後の関手、return, bind である。

```
Let MX := fun X => M (Z + X).
Let retX : idfun --> MX := fun X x => Ret (inr x).
Let bindX t f :=
  t >>= fun c => match c with inl z => Ret (inl z) | inr x => f x end.
```

stateT の場合と同様に、遅延モナドが exceptT で変換後遅延モナドであることを示す必要がある。そこで、次のように定義した。関数 DEA を合成することでにより、エラーが発生した際 inl (inl u) を返すことで、繰り返しを終了する。

```
M: delayMonad
Let DE := MX unit M.
Let whileDE (body : A -> DE (B + A)) : DE B := while (DEA o body)
Let wBisimDE (d1 d2 : DE A) := wBisim d1 d2.
```

3.3 検証の具合例

M : delay_typed_storemonad を用いることで、参照に関する計算と while を用いた繰り返しを含むプログラムを表現可能である。
例えば、参照と繰り返しを用いて階乗を計算するプログラム factdts を以下のように定義できる。

```
Let factdts_aux_body r n : M (unit + nat) :=
  do v <- cget r;
     match n with
     |O => do _ <- cput r v; Ret (inl tt)
     |S m => do _ <- cput r (n * v); Ret (inr m)
```

```
6        end.
7   Let factdts_aux n r := while (factdts_aux_body r) n.
8   Let factdts n := do r <- cnew ml_int 1;
9                    do _ <- factdts_aux n r ;
10                   do v <- cget r; Ret v.
```

factdts について、実際に階乗を計算していることを検証する。つまり次 factn と計算として一致することを示した。

```
1   Fixpoint fact n := match n with |0 => 1 |m.+1 => n * fact m end.
2   Let factn n := do r <- cnew ml_int (fact n);
3                  do v <- cget r; @ret M _ v.
```

ここでは、factdts の while オペレーターを用いた部分 fact_aux が fact を用いた形に書き換えられることについて取り上げる。この書き換えと、型付きストアモナドの等式を用いることで、factn に一致することが従う。証明は n に関する帰納法により図 4,5 のように行った。
underタクティックを用いることで、bind に関する束縛変数を含む項に関する等式変形を行っている。

```
while (factdts_aux_body r) 0
⟦ rewrite fixpointE/= !bindA. ⟧
 ≈  cget r >>= (fun s => (cput r s >> Ret (inl tt)) >>=
                      sum_rect Ret (while (factdts_aux_body r)))
⟦ under eq_bind => s. (* rewrite under binder *) ⟧
'Under[ (cput r s >> Ret (inl tt)) >>=
         sum_rect Ret (while (factdts_aux_body r)) ]
⟦ rewrite bindA bindretf/= -{1}(mul1n s). ⟧
'Under[ cput r (1 * s) >> Ret tt ]
⟦ over. ⟧
cget r >>= (fun s => cput r (1 * s) >> Ret tt)
= cget r >>= (fun s => cput r (fact 0 * s) >> Ret tt)
```

図 4. n = 0 の場合

n = n' + 1 の時の証明の際、setoid_rewrite IH の部分では、束縛変数を含む項に対する wBisim に関する等価性変形を行っている。これは、第 2.5 節で説明したように、bind をパラメトリックモルフィズムとして、インスタンス化しているため行うことができる。

```
while (factdts_aux_body r) n'.+1
⟦ rewrite fixpointE/= !bindA. ⟧
 ≈ cget r >>= (fun s => (cput r (n'.+1 * s) >> Ret (inr n')) >>=
                     sum_rect Ret (while (factdts_aux_body r)))
⟦ under eq_bind => s (* rewrite under binder *) ⟧
'Under[ (cput r (n'.+1 * s) >> Ret (inr n')) >>=
     sum_rect Ret (while (factdts_aux_body r)) ]
⟦ rewrite bindA bindretf/=. ⟧
'Under[ cput r (n'.+1 * s) >> while (factdts_aux_body r) n' ]
⟦ over. (* over *) ⟧
cget r >>= (fun s => cput r (n'.+1 * s) >>
           while (factdts_aux_body r) n')
⟦ setoid_rewrite  IH (*rewrite using induction hypothese IH*)⟧
 ≈  cget r >>= (fun s => cput r (n'.+1 * s) >>
           (cget r >>= (fun s0 => cput r (fact n' * s0) >> Ret tt)))
⟦ under eq_bind => s (* rewrite under binder *) ⟧
'Under[ cput r (n'.+1 * s) >>
       (cget r >>= (fun s0 => cput r (fact n' * s0) >> Ret tt)) ]
⟦ rewrite cputget -bindA cputput. (*laws for cput and cget *) ⟧
  'Under[ cput r (fact n' * (n'.+1 * x)) >> Ret tt ]
⟦ over. (* over *) ⟧
cget r >>= (fun s => cput r (n'.+1 * fact n' * s) >> Ret tt)
= cget r >>= (fun s => cput r (fact n'.+1 * s) >> Ret tt)
```

図 5. n = n' + 1 の場合

4 関連研究

4.1 一般再帰関数に対する Coq 上での検証

Xia らは、余帰納的型を用いて定義されたデータ構造 ITree を用いてインタラクションのある一般再帰関数を表現している [XZH+19]。この手法では、それぞれの副作用をイベントとして ITree に埋め込み、ITree の間の弱双模倣性を満たす等価関係を定義することでプログラムの検証を可能にしている。またそれに関連して Zakowski らはライブラリ pako を拡張した gpaco を用いることで弱双模倣性に関する等式論理を Coq で実装している [ZHHZ20]。

4.2 Elgot モナドに関する理論

完全エルゴートモナドに関する理論的研究としては、Sergey と Piróg らが、Compete elogit monad の構造が余帰納的一般再開モナドトランスフォーマーにより保たれることを示している [GRS15]。また、Simpson らは、は iteration theory の公理が完全であることを示している [SP00]。

5 まとめと今後の課題

本研究により、遅延モナドを通じて Monae で一般再帰関数を扱うことができるようになった。次に、モナドトランスフォーマーを用いて、状態変化や例外処理といった計算効果を含む一般再帰関数を Monae で扱うことができるようになった。最後に Setoid ライブラリを用いることで、等価関係に基づく変形による検証を rewrite タクティクスを中心とした証明により行うことができた。
今後の課題としては以下である。

プログラムの部分正当性の検証　現在、プログラムの実行後の状態について検証するには、factauxE の証明で行ったように整礎関係を見つけ出し、その関係に関する帰納法を行う必要がある。つまり、実質的にプログラムの停止性を示す必要があり、ホーア論理で行うようなループ不変条件を用いた検証を行うことができない。したがって、ループ不変条件を用いた部分正当性の検証をどのように等式変形の枠組みで導入するかが課題である。

また、今後の展望としては以下である。

確率モナドとの組み合わせ　今回の研究では、モナドトランスフォーマーによるモナドの組み合わせを行った。一方、確率モナドはモナドトランスフォーマーによる組み合わせが難しいことが知られている。Monae では、インターフェイスに異なるモナドのオペレーターとその間の等式を追加することで、確率モナドと非決定性モナドの組み合わせを行っている。遅延モナドに対しても同様の手法で組み合わせることで多様な検証が可能になると考えている。

一般的な双模倣性に関する Monae での検証　今回の研究で用いた計算的同値性は、Delay A 上の弱双模倣性を満たす関係である。一方、一般的な双模倣性に関する性質を持つモナドのインターフェイスを定義し、その健全性を示すことで、並行プログラムや Stream などの無限の長さも持ちうるデータを扱うプログラムに関する検証が期待できる。

参考文献

[AGS23a] Reynald Affeldt, Jacques Garrigue, and Takafumi Saikawa. A practical formalization of monadic equational reasoning in dependent-type theory, 2023.

[AGS23b] Reynald Affeldt, Jacques Garrigue, and Takafumi Saikawa. A practical formalization of monadic equational reasoning in dependent-type theory, 2023.

[AMV10] Jiří Adámek, Stefan Milius, and Jiří Velebil. Equational properties of iterative monads. Information and Computation, 208(12):1306–1348, 2010. Special Issue: International Workshop on Coalgebraic Methods in Computer Science (CMCS 2008).

[ANS19] Reynald Affeldt, David Nowak, and Takafumi Saikawa. A hierarchy of monadic effects for program verification using equational reasoning. In Graham Hutton, editor, Mathematics of Program Construction, pages 226–254, Cham, 2019. Springer International Publishing.

[BÉ93] Stephen L. Bloom and Zoltán Ésik. Iteration Theories, pages 159–213. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993.

[Cap05] Venanzio Capretta. General recursion via coinductive types. Logical Methods in Computer Science, Volume 1, Issue 2, July 2005.

[CST20] Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. Hierarchy builder: algebraic hierarchies made easy in coq with elpi. In FSCD 2020 - 5th International Conference on Formal Structures for Computation and Deduction, number 167, pages 34:1–34:21, Paris, France, June 2020.

[Fil94] Andrzej Filinski. Recursion from iteration. LISP and Symbolic Computation, 7:11–37, 1994.

[GH11] Jeremy Gibbons and Ralf Hinze. Just do it: simple monadic equational reasoning. SIGPLAN Not., 46(9):2–14, September 2011.

[Gim95] Eduarde Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nordström, and Jan Smith, editors, Types for Proofs and Programs, pages 39–59, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[GM10] Georges Gonthier and Assia Mahboubi. An introduction to small scale reflection in coq. Journal of Formalized Reasoning, 3(2):95–152, Jan. 2010.

[GRS15] Sergey Goncharov, Christoph Rauch, and Lutz Schröder. Unguarded recursion on coinductive resumptions. Electronic Notes in Theoretical Computer Science, 319:183–198, 2015. The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI).

[GS22] Jacques Garrigue and Takafumi Saikawa. Validating ocaml soundness by translation into coq. 2022.

[PG13] Maciej Piróg and Jeremy Gibbons. Monads for behaviour. Electronic Notes in Theoretical Computer Science, 298:309–324, 2013. Proceedings of the Twenty-ninth Conference on the Mathematical Foundations of Programming Semantics, MFPS XXIX.

[SP00] Alex Simpson and Gordon Plotkin. Complete axioms for categorical fixed-point operators. In Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science, LICS '00, page 30, USA, 2000. IEEE Computer Society.

[The24] The Coq Development Team. The Coq Proof Assistant Reference Manual, 8.20.0 edition, 2024.

[UV17] Tarmo Uustalu and Niccolò Veltri. The delay monad and restriction categories. In Dang Van Hung and Deepak Kapur, editors, Theoretical Aspects of Computing – ICTAC 2017, pages 32–50, Cham, 2017. Springer International Publishing.

[XZH+19] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in coq. Proc. ACM Program. Lang., 4(POPL), December 2019.

[ZHHZ20] Yannick Zakowski, Paul He, Chung-Kil Hur, and Steve Zdancewic. An equational theory for weak bisimulation via generalized parameterized coinduction. In Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, page 71–84, New York, NY, USA, 2020. Association for Computing Machinery.