Delay 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 Delay 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、Delay 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、Haskell をはじめとした様々な関数型プログラミング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]。

1.1　Monae

Monae[ANS19] は、定理証明支援系 Coq 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、Mathcomp/ssreflect 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 ライブラリ [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。

1.2 構造的でない再帰関数の扱い

しかし、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。しかし、非決定性を持つ quicksort などのプログラムでは、引数が減少することを証明することはできず、[ANS19] では dependently-typed assertion と呼ばれる方法を用いる工夫が行われている。また、停止性の未解決なコラッツ関数などは定義することができない。

1.3 CoInductive type 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、余帰納的定義を用いる手法がある。余帰納的定義は無限長のリストであるストリームなど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。Coq では、生産性条件を満たす限り、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、戻り値が余帰納的データとなるが、停止性の証明なしに定義することができる。
例えば、[XZH^{+}19] では、coinductive type 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。
Delay 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、Delay 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。

1.4 Complete elgot monad

Delay モナドのインターフェイスを定義するにあたって、complete elgot monad[AMV10] を参考にした。complete elogot monad 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。

$$\begin{tikzcd}
X \arrow[r, "f^\dagger"] \arrow[d, "f"] & MA \\
M(A+X) \arrow[r, "{M[\eta_A, f^\dagger]}"] & M^2A \arrow[u, "\mu_A"]
\end{tikzcd}$$

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。
Delay 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 complete elgot monad となることが知られている [UV17]。

1.5 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。

- Delay 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 setoid ライブラリを用いた generalized rewrite tactics の利用により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 Delay 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、4 節で setoid ライブラリを用いた generalized rewrite tactics について、5 節で検証の具体例について説明する。また 6 節で関連研究について、7 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 Delay モナドの Monae における実装

2.1 Delay モナドの定義

Delay 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。

Delay モナドを構成する関手 Delay は、(A:Type) → (cofixpoint of X = A + X) 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.
```

Delay 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、Delay 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 complete elgot monad の † 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).
```

2.2 計算的同値性

さて、ここで等式変形による検証は、Delay 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 Delay 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、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 上で示したかについて説明する。
まず、余帰納法の原理とは次のような主張である。

定理 1. 余帰納法の原理
X,A,B,C を集合,F を $\mathcal{P}(X)$ から $\mathcal{P}(X)$ への単調な関数とする。
ただし、F が単調であるとは、$A \subset B \implies F(A) \subset F(B)$ が成り立つことである。
この時、$A \subset F(A) \implies A \subset (F$ の最大不動点$)$ が成り立つ。

このもとで、`wBisim A` の定義は、集合 X について、

$$F(X) = \{(\texttt{d1}, \texttt{d2}) | \exists \texttt{a} \in A, \texttt{Terminate d1 a} \wedge \texttt{Terminate d1 a}\}$$
$$\cup \{(\texttt{DLater d1}, \texttt{DLater d2}) | (\texttt{d1}, \texttt{d2}) \in X\}$$

で定義される F の最大不動点として定義される。
Coq では、`cofix`タクティックを用いて最大不動点を取ることで余帰納法を行うことができる。
例えば、余帰納法を用いて `wBisim (DLater d) d` を示した。
F は定義より、明らかに単調である。
従って、余帰納法の原理より、集合 $Y = \{(\texttt{DLater d}, \texttt{d}) | \texttt{d} \in \texttt{Delay A}\}$ とおき、$Y \subset F(Y)$ が成り立つを示せばよい。Coq では、この最大不動点 Y に対応する命題を cofix で宣言することができる。
任意に Y の要素 `d` を取ると、定義より、$\texttt{d} = (\texttt{DLater (DNow a)}, \texttt{DNow a})$ または、$\texttt{d} = (\texttt{DLater (DLater d')}, (\texttt{DLater d'}))$ の二つの場合がある。
一つ目の場合は、`Terminate (DNow a) a`, `Terminate (DLater(DNow a)) a` が成り立つため、$d \in F(Y)$ である。
二つ目の場合は、Y の定義より、$(\texttt{DLater d'}, \texttt{d'}) \in Y$ である。従って $\texttt{d} \in Y$ である。
よって、$Y \subset F(Y)$ であることがわかった。
したがって、Coq では、次のように証明される。

```
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 Delay モナドのインターフェイス

以上のことを踏まえて、Delay 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、Delay 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

表 **1.** The operator and relation

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

表 **2.** The laws of the wBisim

`wBisim_refl`	`a ≈ a`
`wBisim_sym`	`a ≈ b -> b ≈ a`
`wBisim_trans`	`a ≈ b -> b ≈ c -> a ≈ c`

等さを表す関係である、`wBisim` $\approx$ と、それが同値関係であるという規則を導入する。
また、`while` オペレーターに対する 6 つの規則を定義した。初めの 3 つの規則は、Complete elgot monad 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。
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 4 節で説明する。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。

3　型付きストアモナドとの組み合わせ

Delay 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、Delay 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、[AGS23] 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 delay_typed_storemonad を定義した。

表 3. The laws of while

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`
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`

3.1 型付きストアモナド

型付きストアモナドは、Coqgen[GS22] により変換したコードの、reference を表すために導入された。
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 operator 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 Delay モナドの構造を保つことを示すことで、delay_typed_storemonad を定義した。

`MS (seq binding) option_monad` $\Rightarrow$ `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、delay モナドが stateT で変換後、delay 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、delay モナドが exceptT で変換後 delay 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 定義できる関数の例

M : delay_typed_storemonad を用いることで、参照に関する操作と 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 while 文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)
      end.
Let factdts_aux n r := while (factdts_aux_body r) n.
Let factdts n := do r <- cnew ml_int 1;
                 do _ <- factdts_aux n r ;
                 do v <- cget r; Ret v.
```

4 setoid rewrite

Setoid ライブラリは、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 tactic の使用を可能にする。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。

```
Add Parametric Relation A : (M A) (@wBisim M A)
  reflexivity proved by (@wBisim_refl M A)
  symmetry proved by (@wBisim_sym M A)
  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
(`pointwise_relation A (@wBisim M B)`) とは関数の関係であって、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。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 tactics を中心とした証明が可能になった。

```
H1 : d1 ≈ d2                                       H1 : d1 ≈ d2
H2 : forall a, f ≈ g                               H2 : forall a, f a ≈ g a
                          ==================>
.........................  rewrite H1 H2.           .........................
d1 >>= f ≈  ...                                    d2 >>= g ≈  ...
```

図 1. rewrite with bind

```
H : forall a, f a ≈ g a.                           H : forall a, f a ≈ g a.
                          ==================>
.........................  rewrite H.               .........................
while f b ≈  ...                                   while g b ≈  ...
```

図 2. rewrite with while

5 検証の具体例

ここでは、Delay モナドのインターフェイスを用いた検証の具体例を紹介する。

5.1 Monad の満たす等式

モナドは、bind と return を持ち、表 4 の等式を満たす型クラスとして特徴づけられる。
Monae では、各 Monad をインスタンス化した際にこれらの等式と関連する補題を用いることができるようになる。特にここでは、表 4 の等式を用いて検証する。

5.2 マッカーシーの 91 関数に対する検証

マッカーシーの 91 関数 mc91 は、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関数である。

```
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、停止性が Coq では直ちに判定できない関数である。
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。残っている再帰の深さ n、m が計算している値である。

表 4. The laws of the monad

`bindretf`	`Ret a >>= f = f a`
`bindmret`	`m >>= Ret = Ret`
`bindA`	`(m >>= f) >>= g = m >>= (fun x => f x >>= g)`

```
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、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 1 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. proof of mc91

5.3 状態を用いた factorial

3.3 節で定義した 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。

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

ここでは、factdts の while 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. proof of factn if n = 0

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 3 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. proof of factn if n = n' + 1

6 関連研究

6.1 一般再帰関数に対する Coq 上での検証

[XZH+19] は、coinductive type 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。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 [ZHHZ20] では、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。

6.2 Elgot モナドに関する理論

Complete elgot monad に関する理論的研究としては、Sergey と Piróg らがが、Compete elogit monad の構造がいくつかの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], [PG14]。Simpson らは、は iteration theory の公理が完全であることを示している [SP00]。

7 まとめと今後の課題

本研究により、Delay 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。等価関係に基づく変形による一般再帰関数の検証の利点はとしては、まず coinduction による証明をする必要がないことである。Coq では coinductive type を用いて直接一般再帰関数を表すことが可能ではあるが、検証には余帰納法が不可欠となる。その際、Coq ではガード制約というコンストラクタと余再帰呼び出しに関する構文的制約を満たす必要があり、証明が難しくなる。一方、等価関係に基づく変形では、そのような難しさは生じず、直感的で簡潔な証明が可能になる。次に、他の副作用との組み合わせが容易であることもわかった。本研究で取り上げたように、再帰構造をモナドを用いて表現することで、モナドトランスフォーマーによる型付きストアモナドとの組み合わせを容易に行うことが可能である。さらに、インターフェイスにオペレーターや規則を追加することで他の副作用とも組み合わせることができると期待している。また、これまでの Monae での検証では等式変形による検証のみ扱っていたが、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もこれまでと同様に rewrite タクティクスを中心とした証明により行うことが可能であることがわかった。
今後の課題としては以下である。

1. 他の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しか取り扱っていないが、モナディック等式変形では、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、複数の副作用を持つプログラムに関する検証を可能にしている。特にモナドトランスフォーマーによる組み合わせが難しい確率モナドや非決定性モナドなどとの組み合わせが可能かどうかが課題である。
2. 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。一方で一般的な双模倣性に関する等価性変形を行うことで、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。そのような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どのように定義するか、またその健全性をどのように示すかが課題である。
3. 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。

参考文献

[AGS23] 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.

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

[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.

[PG14] Maciej Piróg and Jeremy Gibbons. The coinductive resumption monad. Electronic Notes in Theoretical Computer Science, 308:273–288, 2014. Proceedings of the 30th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXX).

[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.

[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.