Vol.2019-5 No.4-4
2020/2/9

Coqによるビットストリームの余帰納的な形式化

金山遼馬[†1]　鈴木大郎[†1]

無限長のリストであるストリームは余帰納型として定義され、ストリームの具体例として、Rutten によって定義された二進数を表すビットストリームがある。本研究ではビットストリームを定理証明系 Coq を用いて形式化した。具体的には、ビットストリーム上の算術演算を定義し、ビットストリームがその演算上で可換環を成すことを証明した。本論文では Coq でのビットストリームの形式化の手法について述べる。

1. はじめに

ミーリーオートマトン[5] は古くから計算のモデルとして用いられているが、近年では、デジタルとアナログを統一的に扱うハイブリッドシステム[2] を宣言的に記述できる関数リアクティブプログラミング（FRP）[6),9)] の操作的意味論を与えるものとして注目されている。実際に、FRP に基づく言語である Yampa は、ミーリーオートマトンに基づくオートマトンアロー[11] を使って実装されている。

ミーリーオートマトンは FRP のモデルと考えられるので、FRP に基づくプログラムの検証には、ミーリーオートマトンの定理証明支援系による形式化が有効である。定理証明支援系とは、人間の証明をコンピュータによって補助しながら、その証明の正当性を検査してくれるシステムである。その中でも、Coq[14] は最も普及している定理証明支援系の一つである。

†1 会津大学
The University of Aizu

Rutten はビットストリームと呼ばれるデータ型の上での算術演算を定義し、それが可換環を成すことを示すことで、ビットストリームからミーリーオートマトンを代数的に構成できることを示した[8),13)]。ビットストリームは 0 と 1 からなる無限リストである。Coq は無限データ型を扱うために、余帰納的なデータ型を提供している。しかし Coq の余帰納的な定義は、制約が厳しく扱いづらいという問題点がある。

本研究では、Rutten によるビットストリームの算術演算の定義を余帰納を用いて Coq で形式化する。さらに、ビットストリームがその演算に関して可換環であることを Coq で証明する。Coq では可換環の代数構造に関する自動証明タクティクがあるため、ビットストリームが可換環を成すことを証明することで、ビットストリームからのミーリーオートマトンの構成に関する性質の証明で自動証明が利用できるようになることが期待される。

本論文の構成は以下の通りである。2 章では定理証明支援系 Coq の基本的な手法について述べ、3 章では Coq での余帰納的な定義と証明の手法について、元の概念を交えながら説明する。4 章では、まずビットストリームの概要とその上の算術演算について説明した後、Coq によるビットストリームの形式化と、ビットストリーム上の演算に関してビットストリームが可換環であることの証明について述べる。最後に 5 章で本研究のまとめについて述べる。

2. Coq

Coq[14] とは、INRIA（フランス国立情報学自動制御研究所）によって開発されている定理証明支援系である。Coq では、Gallina と呼ばれる関数型プログラミング言語を用いることによって、型や関数を定義することができる。更に、それに関する性質をタクティクと呼ばれるキーワードを用いて対話的に推論することによって証明できる。詳しくは Coq のチュートリアル[1),4),16)] やリファレンスマニュアル[15] を参照のこと。

2.1 帰納型の定義

Coq で用いる多くのデータ型は帰納的なデータ型である。Coq では、帰納的なデータ型をキーワード `Inductive` で定義する。その例として、自然数を表す `nat` 型の定義を示す。

```
Inductive nat: Type :=
  | O: nat
  | S: nat -> nat.
```

上記の `O` と `S` は構成子と呼ばれ、`nat` 型の値を構成するために用いられる。型としては、構

成子 O は nat 型の定数、構成子 S は nat から nat への一引数関数である。O は自然数 0、S n は自然数 n に 1 加えた自然数を表す。

型の引数をパラメータとして受け取って、その型に関する帰納的なデータ型を定義することもできる。その例として、任意の A 型のリストを表す list A 型の定義を以下に示す。

```
Inductive list (A: Type): Type :=
  | Nil: list A
  | Cons: A -> list A -> list A.
```

Nil は要素を持たない空リスト、Cons x xs はリスト xs の先頭に x を加えたリストを表す。二引数関数 Cons を演算子として表記可能にするためには、キーワード Infix もしくは Notation を用いる。

```
Infix "::" := Cons (at level 60, right associativity).
```

2.2 再帰関数の定義

再帰関数を定義するためにはキーワード Fixpoint を用いる。以下は二つの list A 型の値を結合させる関数 app である。

```
Fixpoint app {A: Type} (l m: list A): list A :=
  match l with
  | [] => m
  | x::l2 => x::(app l2 m)
  end.
```

関数 app は引数として list A 型の値を二つ受け取り、list A 型の値を返す。関数の引数は組として同時に受け取るのではなく、一つずつ受け取る。Fixpoint で定義される再帰関数は必ず停止性が保証されていなければならない。Coq によって再帰関数の停止性を自動的に推論可能にするためには、再帰呼び出しされる関数の引数が、その関数を再帰呼び出しする関数の引数より構成子が減っていることを明確にしなければならない。

再帰を含まない関数を定義したい場合はキーワード Definition を用いて定義する。Definition による関数の定義は再帰を含まないことを除けば Fixpoint による関数の定義と同様である。

2.3 Coq による証明

Coq では、命題を与えてそれを対話的に証明することができる。以下は関数 app によって右から空リスト [] を結合してもリストが変化しないことを表す定理 app_nil である。

```
Theorem app_nil : forall (A : Type) (l : list A), app l [] = l.
Proof.
intros.
induction l; simpl.
- reflexivity.
- rewrite IHl.
  reflexivity.
Qed.
```

図 1 定理 app_nil の帰納法による証明スクリプト

```
  Theorem app_nil: forall (A: Type) (l: list A), app l [] = l.
```

Coq では命題を記述すると証明モードと呼ばれる状態になる。証明モードに入ると Coq は以下を表示する。

```
  ______________________________________(1/1)
  forall (A : Type) (l : list A), app l [] = l
```

証明モードでは、ユーザーがタクティクと呼ばれるコマンドを入力することで推論を進めることができる。例えば、束縛変数を自由変数にするためのタクティク intros を入力すると、サブゴールは以下の通りに変化する。

```
  A : Type
  l : list A
  ______________________________________(1/1)
  app l [] = l
```

このように、証明モードで対話的に推論を進め、最終的に正しいことが自明な形に持っていけば証明が完了できる。定理 app_nil は**図 1** で示されたコマンドを入力することで証明できる。

3. Coq の余帰納的概念

本章では、Coq が提供する余帰納的な概念について、代表的な余帰納データ型であるス

トリームを例にとり述べる。特に、Coq での余帰納を扱いづらくする原因であるガード制約について述べる。

3.1 余帰納型の定義

余帰納型は無限に続くデータ型を定義するために用いられる。余帰納型の代表例は無限の要素を持つリストを表すストリームである。型 A の要素を持つストリームの集合 A^ω は $A^\omega = \{s \mid s : \mathcal{N} \to A\}$ で定義される。すなわち、ストリーム s は自然数の集合 $\mathcal{N}$（0 を含む）から A への関数である。これを $s = (s(0), s(1), s(2), \ldots)$ と表す。また、$s' = (s(1), s(2), s(3), \ldots)$ と表す。$s(0)$ と s' をそれぞれ、ストリーム s の頭部と尾部と呼ぶ。

Coq で余帰納的なデータ型を定義するには、キーワード `CoInductive` を用いる。帰納型は、あるデータ型から構成子によって新しいデータ型を構成するように定義するが、余帰納型では、あるデータ型を構成子によって別のデータ型に分解するように定義される。`A` 型の要素を持つストリームを表す `Stream A` 型の定義を以下に示す。

```
CoInductive Stream (A: Type): Type :=
  | SCons: A -> Stream A -> Stream A.
Infix ":::" := SCons (at level 60, right associativity).
```

`Stream A` 型は唯一の構成子 `SCons` によって構成されるデータ型である。構成子 `SCons` は、引数として `A` 型の値と `Stream A` 型の値を受け取り、`Stream A` 型の値を返す関数である。`SCons` は `list A` 型の構成子 `Cons` と同じ役割を果たすが、`Stream A` 型には `Nil` のような基礎となる構成子が存在しないため、必ず無限個の構成子を持つ値が構成される。

`Stream A` 型に関する関数の定義や命題の証明では、ストリームをその頭部と尾部に分割することが重要である。そのために以下の関数 `head, tail` を定義する。

```
Definition head {A} (s: Stream A): A := match s with x ::: _ => x end.
Definition tail {A} (s: Stream A): Stream A :=
  match s with _ ::: s => s end.
```

関数 `head, tail` を用いることによって、任意の `Stream A` 型の値 `s` を `s = s(0) ::: s‘` のように頭部と尾部に分割できる（`s(0)` と `s‘`は、それぞれ `head s` と `tail s` を表すために `Notation` によって導入された記法である）。

3.2 余再帰関数の定義

余帰納的なデータ型を持つ値は無限のデータ構造なので、再帰関数を使って間接的に定義しなければならない。しかし、余帰納的なデータ型は無限個の構成子によって構成されるために、無限に再帰呼び出しを行うことになり、関数の停止性が満たされない。そのため、余帰納的なデータ型を返す再帰関数は `Fixpoint` では定義できない。Coq では、無限に再帰を繰り返す関数を余再帰関数と呼び、キーワード `CoFixpoint` を用いて定義する。例として、すべての要素が等しいストリームを返す再帰関数 `repeat` の定義を以下に示す。

```
CoFixpoint repeat {A} (x: A): Stream A := x ::: repeat x.
```

`CoFixpoint` では、停止性の代わりに、生産性と呼ばれる制約が満たされることを保証しなければならない。生産性とは、どの余再帰呼び出しでも、余再帰呼び出しする関数の返り値が、その関数によって余再帰呼び出しされる関数の戻り値より構成子の数が増えなければならないという性質である。生産性が保証されていることを Coq によって推論できるようにするためには、ガード制約と呼ばれる構文的制約を満たさなければならない。`Stream A` 型では、ガード制約を満たすために以下の 3 つの条件が保証されなければならない。

(1) 定義中に構成子 `SCons` が一つ以上現れる
(2) 定義中の構成子 `SCons` より外側に `SCons` 以外の関数があってはならない
(3) 余再帰呼び出しは構成子 `SCons` の第二引数の一番外側で行われる

以下の関数 `exFail1` から `exFail3` はそれぞれ (1) から (3) を満たさない例である。

```
Fail CoFixpoint exFail1: Stream nat := exFail1.
Fail CoFixpoint exFail2: Stream nat := tail (1 ::: exFail2).
Fail CoFixpoint exFail3: Stream nat := 1 ::: tail exFail3.
```

どの場合も明らかに余再帰呼び出しを行っても構成子の数が増えない、すなわち生産性が満たされないことが分かる。ただし、Coq では生産性が保証される場合でもガード制約が満たすとは限らないという問題点がある。そのような場合に、生産性を満たすことを証明によって定義するためのキーワードは存在しない。したがって、そのような関数を、ガード制約を満たす補助関数を用いることによって、工夫して定義しなければならない。4 章では、本研究で行ったそのような手法について述べる。

3.3 ストリームの等価性の定義

3.2 節で述べたように、`Stream A` 型の値は、構成子だけで直接表すことができず、余再

帰的な関数を用いて間接的に表される。Coq が提供する等価性=では、その両辺が同じ値に書き換えられるときだけ等価になる。これは、Stream A 型の等価性を表すには適切ではないので、Stream A 型の等価性を新たに定義する必要がある。Stream A 型の等価性を表す 2 通りの方法を以下に示す[10]。

(1) 識別子 SCons の引数がそれぞれ等価
(2) Stream A 型の値の要素ごとの等価

(1) による等価性を表す関係 streamEq の定義を以下に示す。

```
CoInductive streamEq {A} (s t: Stream A): Prop :=
  make_streamEq: s(0) = t(0) -> s‘ ≡ t‘ -> s ≡ t
  where "s ≡ t" := (@streamEq _ s t).
```

streamEq は構成子 make_streamEq によって構成される余帰納的なデータ型として定義される。make_streamEq は、s(0) = t(0) と s‘ ≡ t‘の証明から s ≡ t の証明を構成する。

(2) の等価性は以下のように表せる。

```
forall (A: Type) (i: nat) (s t: stream A), s !! i = t !! i
```

ただし、s !! i は、Stream A 型の値 s の i 番目の要素を表し、その定義は以下の関数 elt により与えられる。

```
Fixpoint elt {A: Type} (s: Stream A) (i: nat): A :=
  match i with
  | O => s(0)
  | S i => s‘ !! i
end
where "s !! i" := (elt s i).
```

関数 elt は、停止性が保証されているので、キーワード Fixpoint を用いて再帰関数として定義できる。上記の (1) と (2) が等価な関係であることを示す以下の補題 streamEq_elt は、帰納法と 3.4 節で述べる余帰納法を用いて証明できる。

```
Lemma streamEq_elt s t:
  s ≡ t <-> forall (i: nat), s !! i = t !! i
```

補題 streamEq_elt に Stream A 型の引数 s, t を与えることによって、全称限量子 forall を省略できる。

3.4 ストリームの等価性の証明

ストリームの等価性を示すためには、3.3 節で定義した等価性を用いた 2 通りの証明手法が可能である[10]。

(1) cofix で余帰納法の仮定をおき、make_streamEq を適用して証明
(2) 補題 streamEq_elt を用いて、要素のインデックスに関する帰納法によって証明

(1) は余帰納法と呼ばれる証明手法である。Coq は証明とプログラムを同一視するカリー・ハワード同型対応にしたがうので、証明によって作られるプログラムがガード制約を満たすように証明を行わなければならない。しかし、タクティクによる証明の過程で、証明に対応するプログラムがガード制約を満たすことに注意しながら証明を行うことは困難なことが多い。それに対し、(2) の証明手法は帰納法による証明であるため、ガード制約が破れる危険性がない。したがって、本研究では (2) の方法でストリームの等価性証明を行う。

3.5 Proper 宣言

Coq では、等式で表された命題や仮定を使った書き換えを行うための rewrite というタクティクが用意されている。rewrite を使うと証明が簡潔になるという利点がある。しかし、ストリームに関する等価性は関係 ≡ により表されるため、そのままでは rewrite による書き換えが行えない。

≡ についても rewrite による書き換えを行えるようにするには、まず ≡ が等価関係であることを宣言する必要がある。そのためには、等価関係を満たすことを表すクラス Equivalence のインスタンスに ≡ がなっていることを宣言し、≡ が反射律、対称律、推移律を満たすことを証明すればよい。それらの性質は余帰納法により容易に証明できる。

しかし、この時点では stream A 型の引数を取る関数 f の引数については ≡ による rewrite を行えない。f を型 A の値を引数に取る関数、≈ を A 上の等価関係とするとき、等価関係 ≈ による rewrite が f の引数に適用できるなら、関数 f は Proper であるという。

Coq では、関数 f の引数に対して = 以外の等価関係 ≈ による rewrite を行いたいときは、f が ≈ に関して Proper であることを宣言する必要がある[3]。この宣言を Proper 宣言と呼ぶ。例えば、任意の A 型の値 a に対し、関数 SCons a が ≡ に関して Proper であることを以下のように宣言する。

```
Instance SCons_proper (a: A): Proper (streamEq ==> streamEq) (SCons a).
```

この宣言では、以下が成り立つことを証明する必要がある。

```
forall x y: Stream A, x ≡ y -> a ::: x ≡ a ::: y
```

`SCons a` が ≡ に関して Proper であることを証明すると、`SCons` の引数についても、≡ による関係を `rewrite` による書き換えに用いることが可能になる。

4. ビットストリームの形式化

本研究で行ったビットストリームの形式化について述べる。

4.1 ビットストリームの概要

本節では、Rutten によって定義されたビットストリームとその演算[13] について述べる。ビットストリームは 0 と 1 を要素に持つストリームであり、2 進数（2-adic number）とみなすことができる。具体的には、分母が奇数である有理数の集合を $\mathbb{Q}_{odd}$ とすると、任意の $q \in \mathbb{Q}_{odd}$ の 2 進表現 $B(q)$ を、以下の等式を満たす一意なビットストリームとして与えることができる。

$$B(q)(0) = odd(q) \qquad B(q)' = B(q - odd(q)/2)$$

ただし、m が奇数である既約分数 n/m に対して、$odd(n/m) = n \bmod 2$ である。上の 2 つの等式のように、ストリームの頭部の値と、尾部の値を余再帰によって定義する等式をストリーム微分方程式と呼ぶ[12]。ビットストリームによる 2 進表現の例をいくつか示す。

$$B(7) = (1,1,1,0,0,0,...) = 111(0)^\omega$$
$$B(-3) = (1,0,1,1,1,1,...) = 10(1)^\omega$$
$$B(1/3) = (1,1,0,1,0,1,0,...) = 1(10)^\omega$$

記号 $(s)^\omega$ は有限列 s が無限に繰り返されることを表す。整数を表すビットストリームは 2 の補数表現による二進数を通常とは逆向きに並べたものとみなせるが、整数ではない有理数は、2 の補数表現による二進小数とは異なる表現になる。

ビットストリーム上の算術演算もストリーム微分方程式によって定義できる。$\wedge$ と $\oplus$ をそれぞれビット上の論理積と排他的論理和、$[\_]$ を、任意のビット b に対して、

$$[b](0) = b \qquad [b]' = [0] \tag{1}$$

と定義される演算子とする。このとき、s, t を任意のビットストリームとすると、ビットストリーム上の加算、符号反転、乗算、逆数は以下のように定義される。

$$(s+t)(0) = s(0) \oplus t(0) \qquad (s+t)' = (s'+t') + [s(0) \wedge t(0)] \tag{2}$$
$$(-s)(0) = s(0) \qquad (-s)' = -(s' + [s(0)]) \tag{3}$$
$$(s \times t)(0) = s(0) \wedge t(0) \qquad (s \times t)' = (s' \times t) + ([s(0)] \times t') \tag{4}$$
$$(1/s)(0) = 1 \qquad (1/s)' = -(s' \times (1/s)) \tag{5}$$

減算 $s - t$ は $s + (-t)$ で、除算 s/t は $s \times (1/t)$ で表される。なお、s の逆数 $1/s$ は $s(0) = 1$ のときだけ定義される。

本研究では上記の演算を Coq で形式化する。このうち、符号反転と減算は上の定義をそのまま Coq で表してもガード制約が満たされるので、次節以降ではそれ以外の演算の形式化について述べる。

4.2 Coq によるビットストリームの定義

はじめにビットを表すデータ型 `bit` を定義する。Coq には、証明で用いられる命題の型を表す `Prop` の他に真理値を表す `bool` 型を扱うことができ、Coq のライブラリでは多数の `bool` 型に関する性質が証明されている。そのため、`bool` 型の性質を利用できるように、`bool` 型と同様の型として以下の通りに `bit` を定義する。

```
Definition bit : Type := bool.
Notation "1" := true.
Notation "0" := false.
```

`bit` 型は `bool` 型と同等であるが、明示的にビットを扱っていることを示すために `bit` 型を定義した。また、`bit` の値を分かりやすくするために `true` を `1`、`false` を `0` と表す。

ビット上の論理積 `and`、論理和 `or`、否定 `not`、排他的論理和 `xor` の各関数を、それぞれ `bool` 型上の論理積 `andb`、論理和 `orb`、否定 `negb`、排他的論理和 `xorb` として定義する。さらに、`Infix` を用いて、関数 `and`, `or`, `xor`, `not` を二項演算子`&&`, `||`, ⊕、前置演算子`!`と定義する。

定義された `bit` 型を用いて、ビットストリームの型を `Stream bit` と表せる。`Stream bit` 型の具体的な値として、以下の `Stream bit` 型の値を返す関数 `unit` を定義する。

```
CoFixpoint unit (x: bit) : Stream bit := x ::: [0]
  where "[ x ]" := (unit x).
```

`unit` は 4.1 節のストリーム微分方程式 (1) を Coq で定義したものであり、`[0]` と `[1]` はそ

れぞれ、整数 0 と 1 を表すビットストリームである。これらは、それぞれ加算、乗算についての単位元である。

4.3 Coq によるビットストリーム上の加算の定義

ビットストリーム上の加算は 4.1 節のストリーム微分方程式 (2) によって定義される。その定義に基づいて、Coq で定義すると以下の通りとなる。

```
CoFixpoint add (s t: Stream bit) : Stream bit :=
  (s(0) ⊕ t(0)) ::: add (add (s‘) (t‘)) [s(0) && t(0)].
```

しかし、この余再帰関数 add はガード制約を満たしていないため、ガード制約が保証されている補助関数を用いて add を再定義する。その後、それがストリーム微分方程式で表される等式を満たすことを証明する。

論理回路で行われる二進数の加算では、最下位の桁である 0 ビット目以外は繰り上がりを入力として受け取って計算を行う。それと同様に、繰り上がりを引数に持つビットストリームの加算を補助関数 addc として以下のように定義する。

```
CoFixpoint addc (s t: Stream bit) (c: bit) : Stream bit :=
  (s(0) ⊕ t(0) ⊕ c) ::: addc (s‘) (t‘) (carry (s(0)) (t(0)) c).
```

addc の定義はガード制約を満たしているため Coq で定義可能である。addc の定義で用いている carry は、3 つのビットを合わせたときに繰り上がりがあるならば 1、なければ 0 を返す関数であり、以下のように定義される。

```
Definition carry (x y c: bit) : bit :=
  match x with
  | 0 => y && c
  | 1 => y || c
  end.
```

定義したい関数 add の一番最初の評価では繰り上がりを考慮しなくてよいので、add は補助関数 addc を用いて以下のように定義できる。

```
Definition add (s t: Stream bit) : Stream bit := addc s t 0.
```

add は二項演算子+として定義する。

Stream bit 型の値を引数に取る関数 add を新たに定義したことに伴う、add が ≡ に関して Proper であることの宣言は容易に行うことができるので省略する。

上記で定義した add の定義がストリーム微分方程式 (2) で定義される演算の定義と等価であることを示すためには、以下の定理 add_head と add_tail を証明すればよい。

```
Theorem add_head s t: (s + t)(0) = s(0) ⊕ t(0).
Theorem add_tail s t: (s + t)‘ ≡ (s‘ + t‘) + [s(0) && t(0)].
```

これらは、cofix を用いた余帰納法による証明でも、補題 streamEq_elt を用いた証明でも示すことができる。

4.4 Coq によるビットストリーム上の乗算の定義

ビットストリーム上の乗算も、4.1 節のストリーム微分方程式 (4) にしたがって Coq で定義するとガード制約を満たさなくなってしまう。そのため、加算のときと同様に補助関数を用いて定義する。

論理回路における二進数の乗算の考え方を用いて、乗算を定義する。論理回路におけるビット列 s と t の乗算では、s の各ビットと t との乗算を s の最下位から順に行い、それらの和を蓄積していく。この方法にしたがうと、s の下位 n ビット目までの計算を築盛したものから下位 n ビットを除いて得られるビットストリーム acc_n の値を以下の漸化式のように定義できる。

$$acc_0 = [s(0)] \times t \tag{6}$$

$$acc_{n+1} = (acc_n)' + [s(n+1)] \times t \tag{7}$$

acc_n は乗算器による下位 n ビット目までの計算結果から下位 n ビットを除いたものなので、$(s \times t)(n)$ の値は

$$(s \times t)(n) = (acc_n)(0) \tag{8}$$

で表せる。

上記の方針に基づいて、ビットストリームの乗算を Coq で定義する。まず、式 (6) を用いて acc_0 を求めるために、bit 型の値 b とビットストリーム s が与えられたとき、[b]×s を表す関数 mulOne を定義する。

```
CoFixpoint mulOne (b : bit) (s : Stream bit) : Stream bit :=
  b && s(0) ::: (mulOne b (s‘)).
```

次に acc_n を用いた補助関数 mulc を以下のように定義する。

```
CoFixpoint mulc (s t acc : Stream bit) : Stream bit :=
```

```
  acc(0) ::: mulc (s‘) t (acc‘ + mulOne (s(0)) t).
```

mulc の定義はガード制約を満たしている。

式 (7)、(8) より、s から下位 $n+1$ ビットを除いたものと t、および acc_n を mulc に与えると、$s \times t$ から下位 n ビットを除いたビットストリームが得られることが容易にわかる。したがって、乗算を表す関数 mul は、補助関数 mulc を用いて以下のように定義できる。

```
Definition mul (s t: Stream bit) : Stream bit :=
  mulc (s‘) t (mulOne (s(0)) t).
```

mul は二項演算子 × として定義する。

Stream bit 型の値を引数に取る関数 mul を新たに定義したので、mul が ≡ に関して Proper であることを宣言する必要がある。宣言により必要となる証明は、add の場合と同様に容易である。

最後に、以下の定理を証明することで、mul の定義が 4.1 節のストリーム微分方程式 (4) で定義される乗算と等価であることを示す。

```
Theorem mul_head s t : (s × t)(0) = s(0) && t(0).
Theorem mul_tail s t : (s × t)‘ ≡ s‘ × t + [s(0)] × t‘.
```

これらの定理は補題 streamEq_elt を用いて証明できる。しかし、cofix を用いる余帰納法では証明を完了させるのは困難である。cofix を用いた証明中に、ストリームの等価性に関するタクティク rewrite による書き換えを行ってしまうと、その証明によって作られるプログラムがガード制約を満たさなくなってしまうという問題点がある。したがって、cofix を用いる余帰納法の証明では、ストリームの等価性に関する書き換えを行わずに推論しなければならない。例えば、乗算の証明内で加算の定義を用いるために、定理 add_head や add_tail による書き換えを行ってしまうと、その時点でガード制約を満たさなくなってしまう。それに対し、streamEq_elt による証明ではガード制約が破れることがないので、ストリームの等価性に関する書き換えを行っても特に問題がない。

ただ、streamEq_elt の証明が cofix を用いる証明より複雑になってしまうことがある。例えば、以下の補題 add_cancel_r を用いて証明を行うことを考える。

```
Lemma add_cancel_r s t : s ≡ t -> s + u ≡ t + u.
```

補題 add_cancel_r は add が Proper であることから直ちに証明できる。streamEq に関する等価性の証明では補題 add_cancel_r を用いることができるが、演算子!!で表された等価性の証明ではその補題を用いることができない。代わりに以下の補題 add_cancel_r_elt を用いる必要がある。

```
Lemma add_cancel_r_elt u s t i :
  (forall j, j <= i -> s !! j = t !! j) -> (s + u) !! i = (t + u) !! i.
```

補題 add_cancel_r_elt の証明は、自然数 i に関する累積帰納法を行わなければならず、またこの補題を用いて等式変形を行う証明でも、累積帰納法を用いらなければならない。したがって、補題 add_cancel_r を用いる証明に比べて補題 add_cancel_r_elt を用いる証明の方が複雑になってしまう。

4.5 Coq によるビットストリーム上の逆数と除算の定義

ストリーム s の逆数は s の先頭のビットが 1 であるときだけ定義されるので、そのようなビットストリームの型を bitOdd 型で表すことにする。これを用いて、4.1 節で示したストリーム微分方程式 (5) によって定義される逆数をそのまま Coq で定義しようとすると、ガード制約を満たさないので失敗する。

```
Fail CoFixpoint inv (t : bitOdd) := 1 ::: - (t‘ × inv t).
```

これは、尾部の一番外側の関数が inv ではなく符号反転演算子-であるためである。

ガード制約を満たす定義を与えるために、除算を定義してから逆数を定義する。ビットストリーム上の除算 (s/t) について、その頭部と尾部に関する等式を、Rutten による加算、乗算、除算の定義を用いて以下のように導く。

$$(s/t)(0) = (s \times (1/t))(0) = s(0) \wedge (1/t)(0) = s(0) \tag{9}$$

$$(s/t)' = (s \times (1/t))' = (s' \times (1/t)) + ([s(0)] \times (1/t)') \tag{10}$$

等式 (9) の最後の式の導出では、$(1/t)(0) = 1$ でなければならないという事実を用いている。

等式 (10) の最後の式の最も外側はまだ除算ではない。そこで、分配法則 $s \times u + t \times u = (s+t) \times u$、× の結合法則 $(s \times t) \times u = s \times (t \times u)$ と、$s \times (-t) = -(s \times t)$ が成り立つと仮定すると、これらと除算の定義より

$$\begin{aligned}(s/t)' &= (s' \times (1/t)) + ([s(0)] \times (1/t)') \\ &= (s' \times (1/t)) + ([s(0)] \times -(t' \times (1/t))) \\ &= (s' - [s(0)] \times t')/t \end{aligned} \tag{11}$$

が導ける。等式 (9) と (11) から関数 div を定義する。このとき、(11) の最後の式の最も

外側は除算なので、div はガード制約を満たす。

```
CoFixpoint div (s : Stream bit) (t : bitOdd) :=
  s(0) ::: div (s‘ - [s(0)] × t‘) t.
```

div は二項演算子/として定義する。

次に、関数 div の定義に現れる bitOdd 型の定義を与える。bitOdd 型は先頭の要素が 1 であるビットストリームを表すので、bitOdd 型は Stream bit 型の部分集合（部分型）と考えられる。Coq には任意の型の部分集合を表す subset 型が提供されているので、それを用いて型 bitOdd を以下のように定義する。

```
Definition bitOdd : Type := { s : Stream bit | s(0) = 1 }.
```

この定義の右辺で subset 型を用いている。

しかし、これまでに行った関数 div の定義を Coq で読み込むと型エラーが起きる。これは、div の引数 t の型が bitOdd 型なのに対し、div の右辺に現れる t‘では t が Stream A 型の値であることを仮定しているためである。このような場合への対処法として、Coq では、特定の型を暗黙的に別の型として識別させるためのコアーションと呼ばれる機能が提供されている。まず、bitOdd 型を Stream bit 型に変換する関数 bitOdd_coer を以下のように定義する。

```
Definition bitOdd_coer (s : bitOdd) : Stream bit := proj1_sig s.
```

proj1_sig は、bitOdd 型の値を Stream bit 型の値として扱えるようにする関数である。次にキーワード Coercion を用いて以下のように記述する。

```
Coercion bitOdd_coer : bitOdd >-> Stream.
```

これによって、bitOdd 型の値が、関数の引数の型と合わない場合、暗黙的に Stream bit 型の値として認識されるようになる。これによって、上記の関数 div が定義できるようになる。

関数 div が定義できたので、div が ≡ に関して Proper であることを以下のように宣言する。

```
Instance div_proper :
  Proper (streamEq ==> streamEq ==> streamEq) div.
```

しかしこの宣言は Coq により拒否されてしまう。これは、streamEq の引数は Stream A 型なので、Proper であることを示す関数の引数の型はすべて Stream A 型でなければならないのに対し、div の型は Stream bit -> bitOdd -> Stream bit であるためである。コアーションは関数の引数の型を暗黙的に別の型だと認識することができない。

これに対処するには、関数 div の第 2 引数の型を Stream bit 型にして、div を部分関数として定義するか、bitOdd 型の等価性を表す関係を streamEq とは別に定義するという方法が考えられる。本研究では後者の手法を用いて div が Proper であることを宣言した。まず bitOdd 型の等価性を表す関係 bitOddEq を以下の通りに定義する。

```
Definition bitOddEq (s t : bitOdd) : Prop := s ≡ t.
Infix "≡odd" := bitOddEq (at level 70).
```

次に、bitOdd 型を用いて Proper 宣言ができるようにするために、bitOdd が等価関係であることを宣言する。この宣言で必要になる証明は、streamEq が等価関係であることから容易に示せる。その後、bitOddEq と streamEq を変換可能にするために、以下の補題を証明する。

```
Lemma streamEq_bitOddEq (s t: bitOdd): s ≡ t -> s ≡odd t.
Lemma bitOddEq_streamEq (s t: bitOdd): s ≡odd t -> s ≡ t.
```

なお、上の補題では、bitOdd 型の s と t に対して Stream A 型を引数に取る ≡ を適用しているが、コアーションが定義されているので型エラーにはならない。

以上より、div が Proper であることを以下の記述によって宣言できる。

```
Instance div_proper :
  Proper (streamEq ==> bitOddEq ==> streamEq) div.
```

上の 2 つの補題は、既に ≡ に関して Proper であることを宣言した他の算術演算について、それらが bitOddEq に関しても Proper であることを余計な証明なしに保証するのにも使われる。

関数 div を用いれば逆数 inv はただちに定義できる。

```
Definition inv (t: bitOdd): Stream bit := div ([1]) t.
```

inv が ≡ に関して Proper であることの宣言はただちに終了する。

次に、以下の定理を証明することで、関数 inv が 4.1 節のストリーム微分方程式 (5) で

定義される逆数の演算と等価であることを示す。

```
Theorem inv_head t: (inv t)(0) = 1.
Theorem inv_tail t: (inv t)‘ ≡ - (t‘ × (inv t)).
```

これらの定理は補題 streamEq_elt を用いて証明できる。

さらに、関数 div が 4.1 節で示した除算の定義と一致することを以下の定理で示す。

```
Theorem div_eq_mul_inv s t: div s t ≡ s × (inv t).
```

この定理も補題 streamEq_elt を用いて証明できる。

4.6 ビットストリームの可換環の証明

Rutten は、4.1 節で定義したビットストリーム上の演算に関して、ビットストリームが可換環を成すということを述べている[8),13)]。本節では、実際にそれが成り立つことを Coq で示す。

Coq でビットストリームが可換環を成すという命題を記述するためにキーワード ring_theory を用いる。これによって記述された以下の定理 bitstream_ring_theory を証明する。

```
Theorem bitstream_ring_theory :
  ring_theory [0] [1] add mul sub minus streamEq.
```

この定理を示すためには **図 2** に記されている 9 つの等式を証明する必要がある。ただし、図 2 中の変数 x, y, z は任意の Stream bit 型の値である。(8) は sub の定義から即座に証明でき、(8) 以外は補題 streamEq_elt を用いて証明できる。

定理 bitstream_ring_theory を証明すると、Coq 上で Stream bit 型がその演算に関して可換環を成すことを宣言できるようになる。その宣言は以下の記述によって行われる。

```
Add Ring bitstream : bitstream_ring_theory.
```

この宣言を行うと、Stream bit 型に関する等式を証明するときに、その等式が環に関する演算のみで表されているならば、その等式が成り立つことをタクティク ring によって即座に証明できる。例えば、以下の補題 mul_add_distr_l はタクティク ring によって直ちに示せる。

```
Lemma mul_add_distr_l s t u: s × (t + u) ≡ s × t + s × u.
```

```
(1) [0] + x ≡ x
(2) x + y ≡ y + x
(3) x + (y + z) ≡ (x + y) + z
(4) [1] × x ≡ x
(5) x × y ≡ y × x
(6) x × (y × z) ≡ (x × y) × z
(7) (x + y) × z ≡ x × z + y × z
(8) x - y ≡ x + - y
(9) x + - x ≡ [0]
```

図 2 bitstream_ring_theory を示すための命題

ただし、演算 inv や div を含むビットストリームの等価性はタクティク ring によって証明できない。その場合は以下の補題 mul_inj_r と div_mul_l_red を用いて、両辺を inv と div を含まない式に変形すればよい。

```
Lemma mul_inj_r (u: bitOdd) s t: s × u ≡ t × u -> s ≡ t.
Lemma div_mul_l_red s (t: bitOdd): (s × t) / t ≡ s.
```

補題 mul_inj_r は両辺に同等の数を乗じるために、補題 div_mul_l_red は約分を行うために、それぞれ用いられる。両辺から inv と div が除去できたら、その等式はタクティク ring によってただちに証明できる。

5. ま と め

本研究では、既存の Coq での余帰納的な形式化を手法を用いて、Rutten が定義したビットストリームの形式化を行った。ビットストリーム上の演算を定義する際に注意しなければならないガード制約については、本論文で述べた手法を用いて対処した。また、Coq で提供されている自動証明タクティク ring を用いるためにビットストリームが可換環を成すことの証明を行った。結果として、ビットストリームの等価性の証明をほとんど自動的に行えるようになった。

今後、本研究で形式化したビットストリーム上の関数からミーリーオートマトンへの構成を形式化するにあたって、Coq の拡張である Ssreflect[7),16)] が利用できると考えられる。Ssreflect は証明のコマンドを簡潔にかけるだけではなく、有限型に関するライブラリが豊

富に提供されている。ビットストリーム上の特定の関数から、有限状態数のミーリーオートマトンを構成することが分かっており、その形式化には Ssreflect の有限型に関するライブラリが役に立つことが期待できる。

参 考 文 献

1) Bertot, Y. and Castéran, P.: Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions，Springer Science & Business Media (2013).
2) Branicky, M.: Introduction to Hybrid Systems, *Handbook of Networked and Embedded Control Systems*, Birkhäuser Boston, pp.91–116 (2005).
3) Castéran, P. and Sozeau, M.: A Gentle Introduction to Type Classes and Relations in Coq, `https://www.labri.fr/perso/casteran/CoqArt/TypeClassesTut/typeclassestut.pdf` (2016).
4) Chlipala, A.: *Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant*, MIT Press (2013).
5) Eilenberg, S.: *Automata, Languages,and Machines, Vol. A*, Academic Press (1974).
6) Elliott, C. and Hudak, P.: Functional Reactive Animation, *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, ACM, pp.263–273 (1997).
7) Gonthier, G., Mahboubi, A. and Tassi, E.: A Small Scale Reflection Extension for the Coq system, Research Report RR-6455, Inria Saclay Ile de France (2016).
8) Hansen, H. and Rutten, J.: Symbolic Synthesis of Mealy Machines from Arithmetic Bitstream Functions, *Sci. Ann. Comp. Sci.*, Vol.20, pp.97–130 (2010).
9) Hudak, P., Courtney, A., Nilsson, H. and Peterson, J.: Arrows, Robots, and Functional Reactive Programming., *Advanced Functional Programming*, Lecture Notes in Computer Science, Vol.2638, Springer, pp.159–187 (2002).
10) Krebbers, R., Parlant, L. and Silva, A.: Moessner's Theorem: An Exercise in Coinductive Reasoning in Coq, *Theory and Practice of Formal Methods*, Springer, pp. 309–324 (2016).
11) Paterson, R.: Arrows and Computation, *The Fun of Programming* (Gibbons, J. and de Moor, O., eds.), Palgrave, pp.201–222 (2003).
12) Rutten, J.J.: Behavioural differential equations: a coinductive calculus of streams, automata, and power series, *Theoretical Computer Science*, Vol. 308, No. 1-3, pp. 1–53 (2003).
13) Rutten, J.J.: Algebraic specification and coalgebraic synthesis of mealy automata, *Electronic Notes in Theoretical Computer Science*, Vol.160, pp.305–319 (2006).
14) The Coq Development Team: The Coq Proof Assistant, `https://coq.inria.fr/`.
15) The Coq Development Team: Library. Coq.Lists.Streams, `https://coq.inria.fr/library/Coq.Lists.Streams.html`.
16) 萩原学，レナルドアフェルト：Coq/SSReflect/MathComp による定理証明: フリーソフトではじめる数学の形式化，森北出版 (2018).