情報処理学会論文誌　プログラミング　Vol.11　No.4　1–12 (Dec. 2018)

定理証明支援系 Coq における不等式変形記法

村田 康佑[1,a)]　江本 健斗[1,b)]

受付日 2018年4月20日, 採録日 2018年8月8日

概要：数学定理やプログラムの性質の形式的証明では，自然数上の不等式についての証明が頻出する．しかし，定理証明支援系 Coq での不等式の形式的証明は，非形式的証明とは異なる記法で記述されるため，数学的な直観がそのまま使えないことも多い．たとえば，非形式的証明では，不等式 $L \leq R$ を証明するために，しばしば $L = M_1 \leq M_2 = M_3 \leq \cdots \leq M_n = R$ のように項を不等号で「鎖状」につなげて示す宣言的な記法が用いられる．こうした記法は数学の教科書等でよく馴染んだ記法であり，直観的に理解・記述することが可能である．一方，Coq にはそうした宣言的な記法は標準では用意されていないため，証明の理解・記述が困難になっている．本論文では，Coq 上で，自然数上の不等式変形を，非形式的証明のように「鎖状」に記述する手法を提案する．本手法の特徴は，タクティックライブラリによって「鎖状」記法が実現されることにあり，それゆえ，提案記法はライブラリをモジュールとして読み込むだけで既存記法とあわせて使うことができる．また，このタクティックライブラリを用いて，Ackermann 関数の性質についての不等式の証明を試みる．その結果，標準的な数学の教科書と近い記法で形式的証明を記述できることを確認する．

キーワード：定理証明支援系，Coq，宣言的証明，タクティックライブラリ，不等式

A Tactic Library for Transforming Inequalities in Coq

Kosuke Murata[1,a)]　Kento Emoto[1,b)]

Received: April 20, 2018, Accepted: August 8, 2018

Abstract: Formal proofs of inequalities on natural numbers is important for formal proofs of mathematical theorems and properties of programs. However, Coq's notation of formal proofs of inequalities is different from that of informal proofs. For instance, when we write an informal proof of inequality $L \leq R$, we usually use a declarative notation like a "chain" such that $L = M_1 \leq M_2 = M_3 \leq \cdots \leq M_n = R$. Such notation is common in textbooks, and thus it enables us to understand proofs intuitively. On the other hand, standard Coq does not support such a declarative notation, so that we cannot understand proofs in Coq intuitively. In this paper, we propose a novel approach to enable us to write formal proofs in the "chain" notation. One of main features of our approach is that the chain notation is realized as a tactic library, so that we can use it easily by only loading it as a module, and in conjunction with the conventional notation. We also try writing formal proofs for properties of Ackermann function with our tactic library. The result shows that we can write the formal proofs like informal proofs in a textbook.

Keywords: proof assistant, Coq, declarative proof, tactic library, inequality

1. はじめに

プログラム検証や数学の定理証明検証といった文脈で，Coq [1] や Isabelle/HOL [2] 等の定理証明支援系が有名である．これらのツールは，ユーザが形式的証明を記述するのを補助し，また型検査器によって証明を検証するものである．典型的な成果としては，四色定理の証明の検証 [3] や C コンパイラ [4] の検証が有名である．また最近，Coq およびその拡張である SSReflect/MathComp についての日本語の教科書 [5] が出版される等，国内でも注目されている．

[1] 九州工業大学
Kyushu Institute of Technology, Iizuka, Fukuoka 820–8502, Japan
[a)] murata@pl.ai.kyutech.ac.jp
[b)] emoto@ai.kyutech.ac.jp

しかし，定理証明支援系 Coq での形式的証明は，鉛筆（あるいはボールペン，万年筆，...）でノート上に行う非形式的証明とは異なる記法や構造で記述されるため，数学的な直観がそのまま使えないことも多く，不便である．

本論文が目指す究極の目的は，証明を書く・証明を読むという 2 つの場面において，非形式と形式の間にある大きな溝を埋めることにある．より具体的な目標は，以下の 2 点について，Coq における形式的証明と非形式的証明の溝を埋めることである．

- **記述性**：鉛筆等で紙上に証明を書くのと同じ感覚で，Coq による形式的証明が作れるようになることを目指す．
- **可読性**：非形式的証明がそうであるように，完成した Coq スクリプトが Coq と対話することなく読めるようになることを目指す．

もとより，すべての証明において非形的証明と Coq の形式的証明の差を埋めることは難しい．非形式的証明には，分野ごとに特有の記法や常識があるからである．それゆえ，まずはある特定の分野に絞って，非形式的証明と形式的証明の差異について整理することが必要になってくる．

本論文が対象とするのは，自然数上の不等式である．数学定理やプログラムの性質の形式的証明では，自然数上の不等式についての証明が頻出する．また，自然数や不等式は数学においてごく基本的な対象であるから，主定理が自然数上の不等式ではなかったとしても，その主定理を示すための補題として自然数上の不等式が現れることもある．身近な例としては，計算量の評価といった文脈で不等式上の自然数が重要になることがある [6]．それゆえ，自然数上の不等式について考察することは有用である．

では，自然数上の不等式の証明において，非形式的証明と Coq による形式的証明はどういった違いを孕んでいるだろうか．例題を通して考察する．ここでは，不等式

$$\forall\, n : \mathbf{nat},\ 5 \le n \to 2^{n+1} \le n! \qquad (1)$$

の証明を考える．この不等式の証明は様々な方法がありうるが，非形式的な証明では**図 1** のような証明，Coq での形式的証明では**図 2** のスクリプトによる証明が，それぞれ典型的な例だと考えられる．ただし，図 2 のスクリプトに現れる `n!` は，`Notation` コマンドを使って定義したものである．

図 1 と図 2 を見比べてみると，Coq スクリプトによる形式的証明は，非形式的証明に対して以下のような短所をもっていることに気づく．

- 形式的証明は，非形式的証明に比してより多くの変数が現れており，しかもそれぞれの変数がどういう型を持っているのかがスクリプト上からは分からない．このことを定理証明という文脈でいい直せば，証明中に仮定の名前がたくさん現れるが，それぞれの仮定が具体的に何かは分からないということになる．たとえば，図 2 の形式的証明では `n` のほかにも，`H` といった変数（仮定）が現れているが，Coq との対話なしで `H` が何をさしているのか理解するのは難しい．これはスクリプトを証明として読む場合の可読性を大きく下げる原因となっている．
- 非形式的証明では，「左辺を変形して右辺にする」という方針で証明がなされている．そのため，不等式 $L \le R$ を証明するために，$L = M_1 \le M_2 = M_3 \le \cdots \le M_n = R$ のように項を等号や不等号で鎖状につ

Proof.　n についての帰納法による．
(Base case) $n = 5$ のとき，

$$\begin{aligned} & \text{(左辺)} \\ = \ & 2^{5+1} \\ = \ & 64 \\ \le \ & 120 \\ = \ & 5! \\ = \ & \text{(右辺)}. \end{aligned}$$

(Induction step) n の場合を仮定する．すると，$n+1$ の場合も

$$\begin{aligned} & \text{(左辺)} \\ = \ & 2^{(n+1)+1} \\ = \ & 2 \cdot 2^{n+1} \\ \le \ & 2 \cdot n! && \text{(帰納法の仮定より)} \\ \le \ & (n+1) \cdot n! && (2 \le n+1 \text{ より}) \\ = \ & (n+1)! \\ = \ & \text{(右辺)} \end{aligned}$$

より成り立つ．　□

図 1　不等式 $\forall\, n : \mathbf{nat},\ 5 \le n \to 2^{n+1} \le n!$ の典型的な非形式的証明

Fig. 1　A typical informal proof for inequality $\forall\, n : \mathbf{nat},\ 5 \le n \to 2^{n+1} \le n!$.

```
1  Lemma sampl_ineq_prop :
2      forall (n : nat), 5 <= n -> 2^(n+1) <= n!.
3  Proof.
4    intros n H.
5    induction H.
6    - simpl; omega.
7    - simpl.
8      apply Nat.add_le_mono; auto.
9      rewrite Nat.add_0_r.
10     rewrite <- Nat.mul_1_l at 1.
11     apply Nat.mul_le_mono; auto.
12     omega.
13 Qed.
```

図 2　不等式 $\forall\, n : \mathbf{nat},\ 5 \le n \to 2^{n+1} \le n!$ を証明する典型的な Coq スクリプト

Fig. 2　A typical Coq script for proving a inequality $\forall\, n : \mathbf{nat},\ 5 \le n \to 2^{n+1} \le n!$.

なげて示す記法が用いられている（以下では，この記法を**鎖状記法**という）．ここでは，$M_1, \ldots, M_n$ といった項に注目をして逐次的変形が進められることによって証明が完成している．一方，形式的証明では，ゴールの不等式全体に注目して，それを「自明な不等式に還元する」ことで証明を完成させている．たとえば，図 2 のスクリプトの 8 行目に現れている `apply` タクティックと `auto` タクティックの連続は，`Nat.add_le_mono` ($\forall\, m\, m\, p\, q : \mathbf{nat},\ n \le m \to p \le q \to n + p \le m + q$) を用いて，現在のサブゴール

$$\Gamma \vdash 2^{m+1} + (2^{m+1} + 0) \le m! + m \cdot m!$$

を，より簡単な不等式

$$\Gamma \vdash 2^{m+1} + 0 \le m \cdot m!$$

へと書き換えている．もとより，非形式的証明においてもこうした「自明な不等式に還元する」方法が使われないわけではなく，「左辺を変形して右辺にする」方法と併用されているというべきであろう．しかし，Coq の非形式的証明においては鎖状記法はそもそもサポートされていないため，2 つの記法が併用できる非形式的証明に比して不自由な思考を強いてしまっているといえる．

以上，Coq における形式的証明の短所を 2 点指摘した．このうち前者については，適切にコメントやアノテーションを施すことによってある程度緩和することができる．たとえば `H` という変数が何を指しているか分からなければ，スクリプト上に `H` が何を指しているのかメモしておけばよいということである．

一方，後者はより深刻な問題である．「左辺を変形して右辺にする」という証明の方針をサポートするためには，ぜひとも鎖状記法をサポートするべきである．

本論文では，不等式の鎖状記法をサポートするためのタクティックライブラリを設計し，タクティック記述言語 Ltac を用いて実装した．このライブラリの使用例として，不等式 (1) の証明を記述したスクリプトを**図 3** に示す．このスクリプトを例にして，本論文で実装したタクティックライブラリの特徴について説明する．このスクリプトの証明では帰納法が用いられているが，base case と induction step のそれぞれについて，左辺から右辺へ至る項の変形の過程が明示的に書かれていることが分かる．特に，図 2 の非形式的証明と同様に，

- ステップごとの変形の結果と，
- ステップごとの変形ができる理由（ただし自明な場合は省略できる）

が明示的に表れている点に注目されたい．たとえば，スクリプトの 18 行目から 19 行目は，`IHle` を理由にして，`(2 * (S m + 1)) <= (2 * m !)` なる不等式変形ができることを示している．また，アノテーションによって，現在のサブゴールや重要な変数の型がスクリプト中に明示的に現れていることにも注目されたい．たとえば，スクリプトの 8，14，15 行目には，現在のサブゴールや変数 `IHle` の型の情報が明示的に書かれている．このアノテーションは単なるコメントではなく，Coq の型チェックを呼び出すタクティックとして設計されており，現在のコンテクストと整合しない記述は型チェックによって弾かれるようになっている．これらの機序によって，Coq を知らないユーザであっても，十分にこのスクリプトは理解できると考えられる．

```
1  Lemma sampl_ineq_prop :
2    forall (n : nat), 5 <= n -> 2^(n+1) <= n!.
3  Proof.
4    intros n H.
5    @ H : (5 <= n).
6    induction H.
7    (* base case : n = 5 *)
8    - @ goal : (2 ^ (5 + 1) <= 5!).
9      Left
10     = 64.
11     <= 120   { omega }.
12     = Right.
13   (* induction step *)
14   - @ goal : (2 ^ (S m + 1) <= (S m)!).
15     @ IHle : (2 ^ (m + 1) <= m !).
16     Left
17     = (2 ^ (S m + 1)).
18     = (2 * 2 ^ (m + 1)).
19     <= (2 * m !)     { by IHle }.
20     <= ((S m) * m !)
21                 { because (2 <= (S m)) by omega }.
22     = ((S m)!).
23     = Right.
24 Qed.
```

図 3　提案するタクティックライブラリを用いて不等式 $\forall\, n : \mathbf{nat}$, $5 \le n \to 2^{n+1} \le n!$ を証明するスクリプト

Fig. 3　A script for proving $\forall\ n : \mathbf{nat},\ 5 \le n \to 2^{n+1} \le n!$ using novel tactic library.

また本論文では，提案する手法の評価として，実装したタクティックライブラリを用いて，Ackermann 関数の諸性質の形式的証明を，非形式的な証明に似た可読性の高い記法で記述できることを示す．

なお，本論文を読むうえで，以下の点に注意されたい．

- 我々は，本論文で提案する手法を Coq 8.7.0 で実装した．また，その実装が，少なくとも本論文で紹介した例の範囲では Coq 8.7.1 でも正しく動作することを確かめた．
- 本論文中にはいくつか Coq のスクリプトが現れるが，典型的なモジュールのインポートや notation の定義は

省略している．
- 本論文で述べる手法の実装および，その実装の評価のために作成した証明は，Web 上の以下の URL で公開している：https://github.com/MurataKosuke/Inequality

これ以降の本論文の構成は以下のとおりである．2 章では，関連研究の紹介と，本論文との比較を行う．3 章では，タクティックライブラリの設計と実装の基本方針について述べる．4 章では，3 章で述べた方針を受けて，タクティックライブラリをさらに使いやすいものにするためのアイデアとその実装について述べる．5 章では，本論文で提案するタクティックライブラリを用いた有意義な命題の例として，Ackermann 関数の性質についての証明について述べる．最後に，6 章で本論文のまとめと将来の課題について検討を行う．

2. 関連研究

本章では，本論文の関連研究について述べる．我々の知る限りでは，不等式の証明や不等式変形の記述に特化したライブラリは先に例がない．しかし，等式変形や宣言的証明といった観点から探してみると，いくつか関連研究を見つけることができる．

2.1 等式変形について

等式変形の記述についての既存研究として，Tesson らのプログラム運算のためのタクティックライブラリ [7] があげられる．プログラム運算とは，プログラムの代数的性質を利用して意味を保存したままプログラムを変形し，高速なプログラムを得る技法であり，Bird-Meertens formalism（BMF）といわれる体系が有名である．Tesson らは，BMF に基づく等式運算を，Bird によるプログラム運算のレクチャーノート [8] に出てくるような形で書くためのタクティックライブラリを与えている．プログラム運算という領域に特化したタクティックライブラリではあるが，等式変形を記述するための枠組みとして使用することもできる．しかし，本論文でサポートするような不等式変形までは具体的にはサポートしていない．

本論文は，Tesson らのアイデアが，等式変形のみならず自然数上の不等式変形にも部分的に適用可能であることを示したものであり，また自然数上の不等式変形で生じる特有の問題について考察を加えたものである．また，本論文で実装したタクティックライブラリは，Tesson らのタクティックライブラリの実装を参考にしつつ，そのアイデアを自然数上の不等式変形へと拡張したものである．

2.2 宣言的証明について

各ステップにおけるゴールを明示的に記述するような証明の記法を，宣言的記法という．定理証明支援系において宣言的記法を実現する試みはいくつかある．Isabelle/HOL をベースにした宣言的証明記述言語の設計として Isar [9] が有名であるが，Coq でも Corbineau による新たな証明記述言語の実装 [10] がある．この研究は，全体にわたって自然言語に近い形で証明を記述できるように言語をデザインするものである．そのなかには，等式変形を鎖状に近い記法で記述するためのシンタクスもあり，その点では本論文と同じアプローチによる実装が含まれているといえる．

Corbineau による手法 [10] では，Coq をベースにしつつも新たな言語を設計しなおしていた．それゆえ，処理系を得るためには既存の Coq 処理系を再コンパイルすることが必要になる．一方，我々の手法では，新たな記法が Ltac の範疇で実装したタクティックライブラリとして実現されており，既存の Coq 処理系でタクティックライブラリを読み込むだけで使用できるという点に特徴を持つ．

3. 鎖状記法の設計と実装

本章では，提案する手法の概要について述べる．

3.1 タクティックライブラリの設計の概要

本手法で提案するタクティックライブラリの設計について，概要を述べる．本手法は，あとで述べるように，少し注意すれば，
- 右辺から左辺に至る項の変形
- 等号 $=$ および広義の不等号 $\leq$ のみならず狭義の不等号 $<$ の現れる不等式変形

へ拡張することができるが，まずは「左辺から右辺に至る，等号 $=$ および広義の不等号 $\leq$ が現れるような変形」を記述する手法に絞って解説する．

本論文で提案する手法の核心は，以下の 4 つの `Tactic Notation`[*1]（以下，単にタクティックという）からなる．
- **タクティック 1** `Left =` ⟨*term*⟩ `{` ⟨*tactic*⟩ `}`
- **タクティック 2** `=` ⟨*term*⟩ `{` ⟨*tactic*⟩ `}`
- **タクティック 3** `<=` ⟨*term*⟩ `{` ⟨*tactic*⟩ `}`
- **タクティック 4** `= Right`

これらのタクティックを用いて，$\Gamma \vdash t = s$ や $\Gamma \vdash t \leq s$ の形をしたサブゴールを示すことができる．具体的には，**タクティック 1** からはじめて，**タクティック 2**，**3** を繰り返し用い，**タクティック 4** で証明を終える．これらのタクティックを並べると，あたかも項の変形の過程が鎖状に記されているかのように見えるのが重要な点である．**図 4** は，論理式 $\forall\, x\ y\ z : \mathbf{nat},\ x = y \to y \leq z \to x \leq z$ を示すスクリプトであり，このスクリプトの 5–9 行目では非形式的証明のような不等式変形が書かれているように見えるが，よく見ると上の 4 つのタクティックを並べているだけ

[*1] Coq では，`Tactic Notation` 機能を用いると，タクティック記述言語 Ltac を用いて，新たなタクティックとそのための記法を柔軟に設計することができる．

```
1  Proposition example :
2    forall x y z : nat, x = y -> y <= z -> x <= z.
3  Proof.
4    intros x y z H0 H1.
5    Left
6    = x   { reflexivity }.
7    = y   { exact H0 }.
8    <= z  { exact H1 }.
9    = Right.
10 Qed.
```

図 4　核心となるタクティックのみを用いた単純なスクリプトの例
Fig. 4　An example of simple script using important tactics.

であることが分かる．

これらのタクティックは，具体的には次のような動作をする．

- タクティック 1　`Left =` ⟨*term*⟩ `{` ⟨*tactic*⟩ `}`
  現在のサブゴール $\Gamma \vdash x \leq y$ を，$\Gamma \vdash \langle term \rangle \leq y$ に書き換える．ただし，その書き換えを行うため，$x = \langle term \rangle$ をタクティック ⟨*tactic*⟩ を用いて示し，それを使って現在のサブゴールに現れる左辺の x を ⟨*term*⟩ へと rewrite する．つまり，以下のスクリプトと同じである．

  ```
  let Hre := fresh "H" in (
    assert (x = ⟨term⟩) as Hre by ⟨tactic⟩;
    rewrite Hre at 1; clear Hre).
  ```

  なお，`let Hre := fresh "H" in` の部分は，一時的におく仮定である $x = \langle term \rangle$ にフレッシュな変数を割り当てるために必要な記述である．
- タクティック 2　`=` ⟨*term*⟩ `{` ⟨*tactic*⟩ `}`
  タクティック 1 と同様である．
- タクティック 3　`<=` ⟨*term*⟩ `{` ⟨*tactic*⟩ `}`
  現在のサブゴール $\Gamma \vdash x \leq y$ を $\Gamma \vdash \langle term \rangle \leq y$ に書き換える．ただし，その書き換えを行うため，$x \leq \langle term \rangle$ をタクティック ⟨*tactic*⟩ を用いて示し，それと $\leq$ の推移律から $\langle term \rangle \leq y$ を導く．つまり，以下のスクリプトと同じである．

  ```
  let Hre := fresh "H" in (
    assert (x <= ⟨term⟩) as Hre by ⟨tactic⟩;
    transitivity (⟨term⟩) ;
    [ apply Hre | idtac ];
    rewrite Hre; clear Hre).
  ```

- タクティック 4　`= Right`
  `reflexivity` のいい換えである．

これらのタクティックは，いずれも Ltac を用いて容易に実装することができる．というのも，Ltac には強力なパターンマッチの機能が用意されており，現在のサブゴールに現れる部分項を取得したり，その部分項を使って既存のタクティックを動かすことができるからである．

Step	Script buffers	Gola window
0	intros x y z H0 H1. Left = x { reflexivity }. = y { exact H0 }. <= z { exact H1 }. = Right.	========== forall x y z : nat, x = y -> y <= z -> x <= z
1	intros x y z H0 H1. Left = x { reflexivity }. = y { exact H0 }. <= z { exact H1 }. = Right.	x, y, z : nat H0 : x = y H1 : y <= z ========== x <= z
2	intros x y z H0 H1. Left = x { reflexivity }. = y { exact H0 }. <= z { exact H1 }. = Right.	x, y, z : nat H0 : x = y H1 : y <= z ========== x <= z
3	intros x y z H0 H1. Left = x { reflexivity }. = y { exact H0 }. <= z { exact H1 }. = Right.	x, y, z : nat H0 : x = y H1 : y <= z ========== y <= z
4	intros x y z H0 H1. Left = x { reflexivity }. = y { exact H0 }. <= z { exact H1 }. = Right.	x, y, z : nat H0 : x = y H1 : y <= z ========== z <= z
5	intros x y z H0 H1. Left = x { reflexivity }. = y { exact H0 }. <= z { exact H1 }. = Right.	

図 5　CoqIDE 上で図 4 を読み込んだときの対話の様子
Fig. 5　Progress of interactive proof in Fig. 4 on CoqIDE.

再び図 4 のスクリプトを例にして，各タクティックの動きについて説明する．このスクリプトを，CoqIDE 上でタクティックごとに読み込んだ様子が**図 5** である．

- Step 1 は，Coq 標準の `intros` タクティックを用いているだけであるから問題ないだろう．
- Step 2 では，サブゴールは何も変化していないが，実際には `assert (x = x) as H2 by reflexivity` によって `H2 : x = x` なる項を型コンテキストに追加し，`rewrite H2` で書き換えを行ったあと，`clear H2` で `H2` を削除している．
- Step 3 では，`assert (x = y) as H2 by (exact H0)` によって `H2 : x = y` なる項を型コンテキストに追加し，`rewrite H2` で書き換えを行ったあと，`clear H2` で `H2` を削除している．
- Step 4 では，`assert (y <= z) as* H2 by (exact H1)` によって `H2 : y <= z` なる項を型コンテキストに追加し，`transitivity z` によって，2 つのサブゴール “`y <= z`” と “`z <= z`” を得ている．前者は `exact H0` によって証明を完了し，後者は `idtac` により何もせずそのままにしておいている．それゆえ，Step 4 終了時のゴールとしては，`z <= z` が残る．
- 最後に Step 5 で，`reflexivity` を呼び出して，証明

を終了している．

なお，こうした素朴な実装では，**タクティック 1** よりも前に**タクティック 2** や**タクティック 3** が呼び出せてしまったり，式変形中に**タクティック 1** が複数回呼び出せてしまうという難点がある．もちろんそうしても正しい形式的証明は得られるが，鎖状記法による証明として人間が読むには意味の分からないものになってしまう．それゆえ，ぜひともこうした（スクリプトを人間が読むうえでは）意味の分からないタクティックの適用順を排除する機序が欲しい．実は，これは次節で述べる方法と同じ方法で実現できる．

3.2　右辺から左辺に至る記法への対応

前節では，左辺から右辺に至る方向（以下この変形の方向を rightwards という）への変形を記述するタクティックについて説明した．これとほぼ同様の手法で，右辺から左辺に至る方向（以下この変形の方向を leftwards という）への変形を記述するためのタクティックを実装することができる．具体的には，**タクティック 1**，**タクティック 3**，**タクティック 4** の代わりに以下の 3 つのタクティックを実装し，**タクティック 2** を左辺でなく右辺を書き換えるようにすればよい．

- **タクティック 5** `Right =` ⟨*term*⟩ `{` ⟨*tactic*⟩ `}`
  現在のサブゴールが $\Gamma \vdash x \leq y$ のとき，$y =$ ⟨*term*⟩ を ⟨*tactic*⟩ により示し，右辺 y を ⟨*term*⟩ へ書き換える．
- **タクティック 6** `>=` ⟨*term*⟩ `{` ⟨*tactic*⟩ `}`
  現在のサブゴールが $\Gamma \vdash x \leq y$ のとき，$y \leq$ ⟨*term*⟩ を ⟨*tactic*⟩ により示し，$\leq$ の推移律を用いて右辺 y を ⟨*term*⟩ へ書き換える．
- **タクティック 7** `= Left`
  `reflexivity` と同様．

ところが，問題となるのは rightwards な変形と leftwards な変形を同時にサポートしたい場合である．というのも，**タクティック 2** を呼び出した時点での現在の変形の方向が，leftwards か rightwards かどちらだったか覚えていないと，左辺と右辺のどちらを書き換えたら良いか分からなくなるからである．

タクティックを正しい順序で呼び出している限り，rightwards な変形の最初は**タクティック 1** で始まるし，leftwards な変形の最初は**タクティック 5** で始まる．それゆえ，**タクティック 1** および**タクティック 5** が呼び出された時点で，変形の向きが rightwards か leftwards かを表すダミー項を追加しておけば，書き換えの方向を覚えておくことができる．そのダミー項のための型として，2 つのデータ型

```
Inductive state : Type :=
  Rightwards : state
| Leftwards : state.
```

および

```
Inductive memo (cs : state) : Prop :=
  s : memo cs.
```

を定義しておく．そのうえで，**タクティック 1** が呼び出されたら，タクティック

```
pose ( memo Rightwards ).
```

を呼び出すようにすれば，型コンテクストに

```
P := memo Rightwards : Prop
```

という仮定が追加される．**タクティック 5** が呼び出された場合も同様に `memo Leftwards` なる項を型コンテキストに追加する．そうしておけば，Ltac のゴールのパターンマッチを用いて，型コンテクストに `memo Rightwards : Prop` があるか `memo Leftwards : Prop` があるか調べることで現在の書き換えの向きが分かるわけである．それゆえ，**タクティック 2** を変形の向きに応じて書き換える向きを変えたり，**タクティック 3** と**タクティック 4** を rightwards な書き換えのときにしか呼び出せないようにするといったことが可能になる．

3.3　狭義の不等号

これまで，広義の不等号（すなわち，$\leq$ と $\geq$）における鎖状記法について議論した．しかし，ゴールが $L < R$ のような狭義の不等号（すなわち，$<$ と $>$）による不等式になっている場合について考えると，これまでのタクティックだけでは対応できない．

ひとまず rightwards な変形のみを考える．狭義の不等号による不等式を鎖状記法により証明するために，以下の 2 点を拡張することが必要である．

(1) **タクティック 2** を，$<$ にも対応できるように拡張する．具体的には，**タクティック 2** について，さらに以下のルールを追加すればよい：現在のサブゴールが $\Gamma \vdash x < y$ を $\Gamma \vdash$ ⟨*term*⟩ $< y$ に書き換えるようにする．ただし，その書き換えを行うため，$x <$ ⟨*term*⟩ をタクティック ⟨*tactic*⟩ を用いて示し，それと $<$ の推移律から ⟨*term*⟩ $< y$ をゴールにする．

(2) 新たなタクティック
  `<` ⟨*term*⟩ `{`⟨*tactic*⟩`}`
  を実装する．このタクティックは，以下のような動作をする：現在のサブゴール $\Gamma \vdash x < y$ を $\Gamma \vdash$ ⟨*term*⟩ $\leq y$ に書き換えるようにする．ただし，その書き換えを行うため，$x <$ ⟨*term*⟩ をタクティック ⟨*tactic*⟩ を用いて示し，それと，

$$x < \langle term \rangle \to \langle term \rangle \leq y \to x < y \tag{2}$$

  が成り立つことから，⟨*term*⟩ $\leq y$ をゴールにする．

上の 2 点のうち 2 点目について，「現在のサブゴール $\Gamma \vdash x < y$ を，（$\Gamma \vdash$ ⟨*term*⟩ $< y$ ではなく）$\Gamma \vdash$ ⟨*term*⟩ $\leq y$

にする」という点が重要である．つまり，このタクティックが呼び出されたおかげで，ゴールに現れる不等号を $<$ から $\leq$ へと変えるわけである．というのも，広義の不等号は `reflexivity` が成り立たないので，$<$ のままでは**タクティック 4**（`= Right.`）で証明を終わることができないからである．幸いにして，式 (2) はつねに成り立つので，いつでもこのような書き換えを行うことができる．

3.4　自明なタクティックの省略

図 4 のスクリプトの 5–6 行目において，`x = x` を示すためのタクティックとして `reflexivity` タクティックを指定しているが，この記述は冗長である．なぜならば，`x = x` を示すために `reflexivity` を用いるのは当たり前だからである．このように適用するタクティックが自明な場合には，{*tactic*}部を省略できるようにしたい．

そのためには，**タクティック 1** に加えて，新たに

- **タクティック 1′** `Left =` ⟨*term*⟩

を実装すればよい．この**タクティック 1′** は，ほとんど**タクティック 1** と同様だが，$x = \langle term \rangle$ の証明を ⟨*tactic*⟩ で行うのではなく，たとえば

```
(simpl;reflexivity) || easy || auto
```

で行うようにすればよい．これは，`simpl;reflexivity`，`easy`，`auto` を順番に試すものである．これで自明なタクティックは省略できるようになる．

また，**タクティック 1** の ⟨*tactic*⟩ の部分には，以下のパターンもよく現れる．

- `rewrite` H
- `assert` t `as` H `by` ⟨*tactic*⟩`;`
  `rewrite` H`; clear` H

これらについても，**タクティック 1** の実装の ⟨*tactic*⟩ 部分を上記に置き換えた以下のようなタクティックを用意しておくと便利である．

- **タクティック 1″** `Left =` ⟨*term*⟩`{ by` H `}`
- **タクティック 1‴**
  `Left =` ⟨*term*⟩`{ because` t `by` H `}`

タクティック 1 のみならず，**タクティック 2**，**タクティック 3**，**タクティック 6** についても，同様に ⟨*tactic*⟩ 部を省略したタクティックを作ることができる．

3.5　強力な自動証明タクティックとの併用

本論文で提案するライブラリは，強力な自動証明タクティックとあわせて使用することによって，変形の 1 ステップを人間が理解しやすい程度に大きなものにできるため，より分かりやすい証明を記述できるようになることが期待される．具体的には，以下のようなタクティックと併用すると良い．

omega タクティック　`omega` タクティックは，加法のみからなる（したがって乗法を含まない）自然数上の等式や不等式[*2]を自動証明するためのタクティックである．

ring タクティック　`ring` タクティックは，環（ring）や半環（semiring）上の多項式の等式を自動的に証明するタクティックである．Coq での自然数型 `nat` は半環であるため，自然数上の多項式についての等式は `ring` タクティックを用いて自動的に証明できる．

3.6　アノテーション

1 章で指摘したように，スクリプトの可読性を向上するために，現在のゴールや重要な変数の型等をスクリプト中に明示しておくためのアノテーションが導入されるべきである．そこで，以下の 2 つのアノテーションを用意した．

- `@` H `:` t
- `@ goal :` t

上は変数 H の型が t であることを表し，下は現在のサブゴールが t であることを表す．このアノテーションも Ltac による `tactic notation` として実装することができる．具体的には，**図 6** のようにすればよい．この実装では，パターンマッチによってアノテーションの記述が現在のコンテクストにあっているかどうかを検査し，あっていなければ適切なエラーメッセージを表示して失敗するようになっている．

4.　Generalized Rewriting の利用

本章では，Generalized Rewriting [11] を用いると，鎖状記法を用いた証明記述がより直観的になる場合があることを説明する．

4.1　問題提起

突然だが，**図 7** のスクリプトは，

$$\forall\, x\ y : \mathbf{nat},\\ x \leq y \to S\,(S\,(S\,(x^2))) \leq S\,(S\,(S\,(y^2)))$$

の証明である．しかし，12 行目の `(* ??? *)` の部分が欠けていて不完全である．この部分には，$H : x \leq y$ を仮定として

$$S\,(S\,(S\,(x^2))) \leq S\,(S\,(S\,(y^2))) \tag{3}$$

を示すタクティックを記述することになるが，何を記述するのが良いだろうか？　もちろん，乗算を含む不等式なので `omega` タクティックは使うことができない．

通常，式 (3) のような不等式を示すとき，多くの Coq ユーザは，

[*2] より厳密には，Presburger 算術の論理式として表せる範囲のものということになる．

```
Tactic Notation (at level 2)
       "@" ident(H1) ":" constr(t) :=
  match goal with
  | [ H : t |- _ ] =>
    match H with
    | H1 => idtac "Type Annotation:" ; idtac
    ↪  "OK, "H":"t"is in current environment."
    | _  => fail 2 "Type Annotation Error: No
    ↪  such identifier"H1"that is typed"t
    end
  | _ => fail 2 "Type Annotation Error: No such
  ↪  identifiers that is typed"t
  end.

Tactic Notation (at level 2)
       "@" "goal" ":" constr(t) :=
  match goal with
  | [ |- t ] => idtac "OK, current goal is"t
  | [ |- ?t1 ] => fail 2 "Current goal is
  ↪  not"t", but is"t1
  end.
```

図 6　アノテーションの実装
Fig. 6　Implementation of annotations.

```
Proposition example :
  forall (x y : nat) ,
    x <= y ->
    S (S (S (x^2))) <= S (S (S (y^2))).
Proof.
  intros x y H.
  @ goal :
    (S (S (S (x^2))) <= S (S (S (y^2)))).
  @ H : (x <= y).
  Left
  =  (S (S (S (x^2)))).
  <= (S (S (S (y^2)))) { (* ??? *) }.
  = Right.
Qed.
```

図 7　例題：$\forall\ x\ y : \mathbf{nat},\ x \le y \to S\ (S\ (S\ (x^2))) \le S\ (S\ (S\ (y^2)))$ を証明するスクリプト（ただし一部欠けていて不完全である）
Fig. 7　Example: a script of proving $\forall\ x\ y : \mathbf{nat},\ x \le y \to S\ (S\ (S\ (x^2))) \le S\ (S\ (S\ (y^2)))$ (Note that it is incomplete script, because of including a blank).

- $\forall\, x\ y : \mathbf{nat},\ x \le y \to S\ x \le S\ y$
- $\forall\, x\ y : \mathbf{nat},\ x \le y \to x^2 \le y^2$

のような命題の `apply` を繰り返して，仮定 $H : x \le y$ へと帰結させるのではないだろうか．この方針で証明を書くとすれば，仮に上の 2 つの命題を `S_monotone` と `sq_monotone` として，`(* ??? *)` の部分に以下のようなタクティックを書くことになる．

```
repeat apply S_monotone; apply sq_monotone
; exact H
```

しかし，これはまさに式 (3) の証明を構築するスクリプトそのものであり，もとの鎖状記法の中にこのスクリプトを書くのは不自然である．

直観的には，2 乗 $(-)^2$ も後者関数 S も $\le$ に関して単調であるのだから，そのまま x を y に書き換えたい．実は，そうした直観的な証明を Coq 上で書く方法がある．端的には generalized rewriting を用いればよいのだが，次節以降でそれを詳しく説明する．

4.2　Generalized Rewriting

Coq の `rewrite` タクティックは，Coq のデフォルトの等号 `=` が Leibniz equality であることを利用して，`t = s` であるときゴールに現れる `t` を `s` に書き換えるものである．実はこの `rewrite` タクティックは，Coq 標準ライブラリにある `Setoid` モジュールを読み込むことで，等号のみならず様々な二項関係へ一般化して使うことができる．様々な二項関係といっても，同値関係へと一般化して使用するのが普通であるが，実際には推移的な関係であれば同値関係である必要はないし，本論文でも $\le$ といった同値関係でない関係に拡張する．

Generalized Rewriting のアイデアについて，本論文に関係する範囲で簡単に説明する．そのために，まずいくつか用語の定義をする．A，B を型とする．A 上の 2 項関係 R および $m : A$ に対して，m が R の morphism あるいは proper element であるとは，$R\ m\ m$ が成り立つことである．また，A 上の 2 項関係 R および B 上の 2 項関係 R' に対して，$A \to B$ 上の 2 項関係 $R \mathrel{+\!\!+\!\!\to} R'$ を以下のように定義する：

$$\lambda\ (f\ g : A \to B),\ \forall\ (x\ y : A),\ R\ x\ y \to R'\ (f\ x)\ (g\ y).$$

特に，関数 $f : A \to B$ が，$A \to B$ 上の 2 項関係 $R \mathrel{+\!\!+\!\!\to} R'$ の morphism であるとき，

$$\forall\, x\ y,\ R\ x\ y \to R'\ (f\ x)\ (f\ y)$$

が成り立つことに注意されたい．たとえば，後者 S は $\le \mathrel{+\!\!+\!\!\to} \le$ の morphism である．なぜならば，

$$\forall\ (x\ y : \mathbf{nat}),\ x \le y \to S\ x \le S\ y$$

が成り立つからである．

A を型とし，R を推移的な二項関係とする．ひとたび関数 $f : A \to A$ が $R \mathrel{+\!\!+\!\!\to} R$ の morphism だと示されると，以下の 2 つの推論ができるようになる：

$$\frac{\Gamma,\ R\ x\ y \vdash R\ (f\ y)\ t}{\Gamma,\ R\ x\ y \vdash R\ (f\ x)\ t} \qquad \frac{\Gamma,\ R\ x\ y \vdash R\ t\ (f\ x)}{\Gamma,\ R\ x\ y \vdash R\ t\ (f\ y)}$$

この推論が妥当であることは容易に示せる．重要な点は，上の 2 つの推論について，左の推論は $R\,x\,y$ を理由にして $R\,(f\,y)\,t$ を $R\,(f\,x)\,t$ に書き換えることができるといっていると読めるし，右の推論は $R\,x\,y$ を理由にして $R\,t\,(f\,x)$ を $R\,t\,(f\,y)$ に書き換えることができるといっていると読めるという点である．これが Generalized Rewriting のアイデアである．

上の規則について，分かりやすい例をあげておく．$\leq$ は推移的な二項関係であるし，後者 S は $\leq \mathrel{+\!\!\!\rightarrow} \leq$ の morphism であったから，

$$\frac{\Gamma, x \leq y \vdash S\,y \leq t}{\Gamma, x \leq y \vdash S\,x \leq t} \qquad \frac{\Gamma, x \leq y \vdash t \leq S\,x}{\Gamma, x \leq y \vdash t \leq S\,y} \tag{4}$$

なる書き換えが可能である．この書き換えの妥当性は直観的に明らかだろう．

4.3 Coq における Gneralized Rewriting

Coq では，標準ライブラリの `Coq.Classes.Morphisms` 上に，morphism（proper element）と $\mathrel{+\!\!\!\rightarrow}$ に対応する概念として，それぞれ `Proper` 型クラスと，`++>` 演算子が定義されている．それゆえ，たとえば，後者 S が $\leq \mathrel{+\!\!\!\rightarrow} \leq$ の morphism であることを，`Proper (le ++> le) S` のインスタンスをつくる形で宣言することができる[*3]．具体的には，以下のスクリプトのようにして宣言できる．

```
Program Instance morphism_S_le :
  Proper (le ++> le) S.
Next Obligation.
Proof.
  simpl_relation.
Qed.
```

このインスタンスを作成した後では，`rewrite` タクティックを用いて，式 (4) のような書き換えが可能になる．たとえば，サブゴールが

```
  x, y : nat
  H : x <= y
  ============================
  S (S (S x)) <= S (S (S y))
```

という状態で `rewrite H` を読み込むと，このサブゴールは

```
  x, y : nat
  H : x <= y
  ============================
  S (S (S y)) <= S (S (S y))
```

へと変化する．すなわち，`x` が `y` に書き換わっているのである．なお，この例から分かるように，morphism が複数回適用されていても rewrite することが可能である．

4.1 節の最初の質問に戻る．図 7 の `(* ??? *)` に当てはまるタクティックは何がよいかという問題であったが，以下の 2 つのインスタンスが作成されていれば，“`compute; rewrite H`” と書くだけで済ませることができる．

- `Proper (le ++> le) S`
- `Proper (le ++> le) (fun x => x^2)`

なお，`compute` タクティックはゴールをできるだけ簡約するタクティックである．上のインスタンスにおける `(fun x => x^2)` をゴール中の `x^2` にパターンマッチさせるために，`compute` タクティックを用いて簡約する必要がある．`compute` タクティックを用いなければならないのはユーザにとって冗長であるが，3.4 節で述べたのと同様な方法で，タクティックライブラリ側で適切な省略記法を定義して隠蔽すれば，大した問題ではない．

4.4 提案タクティックライブラリでの利用

本論文で実装したタクティックライブラリには，自然数上の不等式の証明でよく使う morphism について，あらかじめインスタンスを作成している．それゆえ，1 章で提示した図 3 のスクリプトにおいて，18–19 行目の変形および 20–22 行目の変形を簡潔に記述することができている（なお，該当箇所に現れている `by···` や `because···by···` については，3.4 節を参照）．

5. 応用例：Ackermann 関数の性質

本章では，情報科学において基本的な命題の証明を通して，本手法の有用性を検証する．具体的には，提案したタクティックライブラリを用いて Ackermann 関数の性質をいくつか示した．結果として，標準的な教科書（たとえば文献 [12]）に現れる証明と似た記法で書くことができた．

5.1 Ackermann 関数とその性質

Ackermann 関数は，

$$\begin{cases} ack\ 0\ y & = & S\ y \\ ack\ (S\ x)\ 0 & = & ack\ x\ 1 \\ ack\ (S\ x)\ (S\ y) & = & ack\ x\ (ack\ (S\ x)\ y) \end{cases} \tag{5}$$

で定義される関数 $ack : \mathbf{nat} \to \mathbf{nat} \to \mathbf{nat}$ のことであり，計算可能であるが原始再帰的でない関数の例として有名である[*4]．ack が原始再帰的でないことを示す方法はいくつかあるが，最も有名なものは，ack が任意の原始再帰的関数よりも増加が早いことをいうものであり，そのなかでは，

[*3] $\leq$ は，Coq 上では関係 `le` として定義されている．

[*4] ここでいう「原始再帰的」な関数とは，1 階の（すなわち高階でない）関数 $f : \mathbf{nat} \to \mathbf{nat}$ であり，定数関数，後者関数，射影関数から，1 階の関数の関数合成および 1 階の関数の原始再帰法を繰り返し適用することによって得られるものを指す．厳密な定義については，たとえば教科書 [12] を参照．

補題として ack が満たす以下の等式や不等式を証明することになる．

補題 1　$\forall\, y : \mathbf{nat},\ ack\ 1\ y = S\ (S\ y)$.

補題 2　$\forall\, x\ y : \mathbf{nat},\ S\ (S\ y) \leq ack\ (S\ x)\ y$

補題 3　$\forall\, x\ y : \mathbf{nat},\ y < ack\ x\ y$

補題 4　$\forall\, x\ y : \mathbf{nat},\ ack\ x\ y < ack\ x\ (S\ y)$

補題 5　$\forall\, x\ y : \mathbf{nat},\ ack\ x\ (S\ y) \leq ack\ (S\ x)\ y$

補題 6　$\forall\, x\ y : \mathbf{nat},\ ack\ x\ y < ack\ (S\ x)\ y$

補題 7　$\forall\, c_1\ c_2 : \mathbf{nat}, \exists\, c_3 : \mathbf{nat}\ , \forall\, x : \mathbf{nat},$
$ack\ c_1\ (ack\ c_2\ x) \leq ack\ c_3\ x$

我々は，本論文で作成したタクティックライブラリで，有用な命題について可読性の高い形式証明を書くことができることを確かめるため，上にあげた 7 つの等式・不等式の証明を形式化した．以下では，その結果を述べる．

5.2　形式化

我々は，前節であげた 7 つの等式・不等式の形式的証明を記述するため，まずは以下のように Ackermann 関数を定義した．

```
Fixpoint ack (x y : nat) : nat :=
  match x with
  | 0 => S y
  | S x' => let fix ackx (y : nat) :=
                match y with
                | 0 => ack x' 1
                | S y' => ack x' (ackx y')
                end
            in ackx y
  end.
```

この定義には `Fixpoint` コマンドを使っているが，このコマンドには停止性の明らかな再帰関数しか定義できないという制約がある[*5]．その制約により，この `ack` の定義は，式 (5) で示した非形式的定義とは一見異なるものになっている．しかし，この定義は，`ack` の定義中で，`ackx = ack (S x')` なる関数を局所的に定義して，現れる `ack (S x')` を `ackx` に置き換えただけで，実際には非形式的定義と同じものである．実際，以下の 3 つの補題が成り立つことが容易に示せる．

- `forall y, ack 0 y = S y`
- `forall x y, ack (S x) 0 = ack x 1`
- `forall x y,`
  `ack (S x) (S y) = ack x (ack (S x) y)`

この定義のうえで，先にあげた 7 つの補題のうち，**補題 1** および**補題 2** を示したスクリプトを，それぞれ**図 8** および**図 9** に示す．

[*5] 具体的には，再帰呼び出しごとに引数が減少するような関数しか定義できない．

```
1  Proposition ack_1_y_eq_SSy :
2    forall (y : nat), ack 1 y = S (S y).
3  Proof.
4    induction y.
5    - @ goal : (ack 1 0 = 2).
6      Left
7      = 2.
8      = Right.
9    - @ goal : (ack 1 (S y) = S (S (S y))).
10     @ IHy  : (ack 1 y = S (S y)).
11     Left
12     = (ack 1 (S y)).
13     = (ack 0 (ack 1 y)).
14     = (ack 0 (S (S y)))      { by IHy }.
15     = (S (S (S y))).
16     = Right.
17 Qed.
```

図 8　$\forall\, y : \mathbf{nat},\ ack\ 1\ y = S\ (S\ y)$ を証明するスクリプト
Fig. 8　A script for proving $\forall\, y : \mathbf{nat},\ ack\ 1\ y = S\ (S\ y)$.

補題 2 は，先にあげた 7 つの不等式のなかで唯一，二重帰納法を用いる証明であり最も複雑なものであると考えられるが，図 9 のスクリプトは，以下の点のおかげで，人間が読んでも十分に理解可能なものになっている．

- 証明自体が宣言的な記法で書かれており，変形のステップを明確に理解することができる．また，アノテーションによって重要な変数とその型（今回の例であれば帰納法の仮定）が明示されており，どういう仮定を用いたのかきわめて理解しやすいものになっている．
- アノテーションによって，帰納法のゴールごとにサブゴールを明示しており，人間が証明を理解する助けになっている．

また，図 9 のスクリプトについて，次の点を指摘しておきたい．この証明において，y についての帰納法の induction step（21 行目から 33 行目）では，右辺から左辺に至る方向（leftwards）の式変形により証明が進められているが，このことが証明全体を分かりやすいものにしていると考えられる．というのも，leftwards な変形にしたことにより，最初の変形（28 行目から 29 行目）が，ack の定義を用いた書き換えというごく自然なものになっているからである．もし反対に rightwards な変形で証明を書こうとしたら，最初の変形は 32 行目から 31 行目を導く $(S\ (S\ (S\ y)) \leq S\ (S\ (S\ (S\ y))))$ という非常に「思いつきにくい」変形から始めなくてはならない．このように，leftwards な書き換えと rightwards な書き換えの両方をサポートすることは，証明の記述や理解という点において本質的なサポートとなることがある．

ここでは誌面の都合で不等式（**補題 2**）のみスクリプトを乗せたが，7 つのすべての等式・不等式において，こう

```
Proposition ack_SSy_le_ack_Sx_y :
  forall (x y : nat), S (S y) <= ack (S x) y.
Proof.
  induction x.
  - intros y.
    @ goal : (S (S y) <= ack 1 y).
    Left
    = (S (S y)).
    = (ack 1 y)   { by ack_1_y_eq_SSy }.
    = Right.
  - induction y.
    + @ goal : (2 <= ack (S (S x)) 0).
      @ IHx :
        (forall y : nat, S (S y) <= ack (S x) y).
      Right
      =  (ack (S (S x)) 0).
      =  (ack (S x) 1).
      >= (S (S 1))          { by IHx }.
      >= 2                  { omega }.
      = Left.
    + @ goal :
        (S (S (S y)) <= ack (S (S x)) (S y)).
      @ IHx  :
        (forall y : nat, S (S y) <= ack (S x) y).
      @ IHy  :
        (S (S y) <= ack (S (S x)) y).
      Right
      =  (ack (S (S x)) (S y)).
      =  (ack (S x) (ack (S (S x)) y)).
      >= (S (S (ack (S (S x)) y))) { by IHx }.
      >= (S (S (S (S y))))         { by IHy }.
      >= (S (S (S y)))             { omega }.
      = Left.
Qed.
```

図 9　$\forall\, x\ y : \mathbf{nat},\ S\ (S\ y) \leq ack\ (S\ x)\ y$ を証明するスクリプト
Fig. 9　A script for proving $\forall\, x\ y : \mathbf{nat},\ S\ (S\ y) \leq ack\ (S\ x)\ y$.

した可読性に優れたスクリプトを得ることができた．

6. おわりに

本章では，本論文のまとめと，今後の課題について述べる．

6.1 まとめ

本論文では，Coq 上で自然数上の不等式を形式的に証明する際に，数学の教科書に現れるような記法で Coq スクリプト記述するための手法を提案した．具体的には，等式や不等式の変形をそのステップごとに変形の結果と変形ができる理由を明記しながら鎖状に記す記法や，重要な項や現在のサブゴールをアノテーションとしてスクリプト中に記す記法が，タクティックとして実現できることを示し，実際に Ltac を用いて容易に実装できることを提示した．これにより，ユーザはタクティックライブラリを読み込むだけで，不等式変形を「鎖状」に記述することができるようになることを示した．さらに，本論文で提案するタクティックライブラリを用いて，Ackermann 関数の諸性質という情報科学において基本的な命題の証明を，スクリプトとしての可読性の高い形で記述することができることを示した．

6.2 今後の課題

今後の課題として，以下の 4 点をあげておく．

- 本論文では自然数 **nat** 上の不等式に限定して議論したが，本論文で述べた手法の大部分は，そのまま他の型上の不等式にも拡張可能であると考えられる．しかし，自然数以外の型上の等式や不等式では，`omega` タクティックや `ring` タクティックといった強力な自動証明タクティックが動かないこともある．その場合，証明が複雑になってしまいスクリプトの記述性および可読性を下げる可能性が高い．それゆえ，自然数以外の型でも非形式的証明と同じような感覚で有用な命題を示せるようにするためには，その対象となる型ごとに特有の知見が必要になってくると考えられる．特に，実数上の不等式は応用上重要であるにもかかわらず，様々な演算が入り組んだ複雑な式が現れるため難しい．このような，自然数以外の型への対応は重要な今後の課題である．
- 本論文では，現在のサブゴールや重要な変数の型をメモするためのアノテーションを提案した．しかし，このアノテーションは，たとえば CoqIDE を用いてスクリプトを対話的に構築するといった場面では，単に冗長なだけのものである．というのも，現在のサブゴールや重要な変数の型は Goal window に表示されているのだから，わざわざスクリプト上に入力する必要がない．こうした無駄な入力を省略するために，IDE による自動補完を充実させることが重要である．こうした IDE 上の補完機能の設計は今後の課題である．
- 本論文で提案する記法によって，不等式の証明は，「自明な不等式に還元する」方針と「左辺から右辺を導く（あるいは右辺から左辺を導く）」という 2 つの方針が並立することになった．形式的証明において，これら 2 つの方針の使い分けの明確な指針があるわけではない．そうした指針の模索は今後の課題である．
- 本論文では，不等式変形の記法を Coq 上に実装する方法を提案した．この成果が，Agda 等の Coq 以外の定理証明支援系にも適用可能かどうかを調べるのも，今後の課題である．

謝辞　原稿を注意深くお読みいただき，適切な助言をいただきました査読者および編集委員に感謝します．本研究は JSPS 科研費 JP15K15974 の助成を受けたものです．

参考文献

[1] The Coq Development Team: The Coq Proof Assistant (Version 8.8.1) (2018), available from ⟨http://coq.inria.fr⟩.

[2] Nipkow, T., Wenzel, M. and Paulson, L.C.: *Isabelle/HOL: A Proof Assistant for Higher-order Logic*, Springer-Verlag (2002).

[3] Gonthier, G.: Formal Proof—The Four-Color Theorem, *Notices of the American Mathematical Society*, Vol.55, No.11, pp.1382–1393 (2008).

[4] Leroy, X.: Formal Verification of a Realistic Compiler, *Comm. ACM*, Vol.52, No.7, pp.107–115 (2009).

[5] 萩原　学，アフェルト・レナルド：Coq/SSReflect/MathComp による定理証明，森北出版 (2018).

[6] Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C.: *Introduction to Algorithms, 3rd Edition*, The MIT Press, 3rd edition (2009).

[7] Tesson, J., Hashimoto, H., Hu, Z., Loulergue, F. and Takeichi, M.: Program Calculation in Coq, *Algebraic Methodology and Software Technology*, pp.163–179, Springer Berlin Heidelberg (2011).

[8] Bird, R.S.: An Introduction to the Theory of Lists, *Proc. NATO Advanced Study Institute on Logic of Programming and Calculi of Discrete Design*, pp.5–42, Springer-Verlag New York, Inc. (1987).

[9] Bauer, G. and Wenzel, M.: Calculational Reasoning Revisited, *Proc. 14th International Conference on Theorem Proving in Higher Order Logics*, pp.75–90, Springer-Verlag (2001).

[10] Corbineau, P.: A Declarative Language for the Coq Proof Assistant, *Types for Proofs and Programs*, pp.69–84, Springer Berlin Heidelberg (2008).

[11] Sozeau, M.: A New Look at Generalized Rewriting in Type Theory, *Journal of Formalized Reasoning*, Vol.2, No.1, pp.41–62 (2010).

[12] 有川節夫，宮野　悟：オートマトンと計算可能性，培風館 (1986).

村田 康佑

1995 年生．2018 年九州工業大学情報工学部知能情報工学科卒業．現在，同大学大学院情報工学府博士前期課程在籍．論理や型によるプログラムの静的解析等に興味を持つ．

江本 健斗　（正会員）

2015 年 4 月より九州工業大学准教授，博士（情報理工学）．高水準並列プログラミング手法，アルゴリズム導出，それらの定理証明支援系による形式的証明等に興味を持つ．日本ソフトウェア科学会，ACM 各会員．