余帰納法に基づく定理証明の自動化

四宮 誠一　海野 広志

本研究では、余帰納的に定義された述語に関する定理のための自動証明法を提案する。そのような述語を用いると、ストリーム・無限木といった無限データ構造や循環データ構造に関する性質を記述したり、プログラムの無限実行トレースの集合を表現したりすることが可能であるため、プログラムの検証にも応用できる。本研究では、余帰納的定理証明の自動化のため、証明システムを定式化し、SMT ソルバを用いた証明探索法を提案する。また、提案手法を実装し、関連手法との比較実験を行った。

1 はじめに

本研究は余帰納法と呼ばれる定理証明法を自動化するための手法を提案する。余帰納法とは、数学的帰納法に代表される帰納法と対照的な証明法である。帰納法は自然数全体の集合や有限長のリスト全体の集合などの、帰納的に定義された集合に関する定理証明に用いられる。一方で、余帰納法は無限長のリスト全体の集合や無限長のツリー全体の集合などの、余帰納的に定義された集合に関する定理証明に用いられる。余帰納的に定義された集合は、次のように定式化される。

定義 1.1 (余帰納的に定義された集合 νF) 台集合を L として、単調関数 $F : 2^L \to 2^L$ に関して $S \subseteq F(S)$ を満たす集合 S のうち最大のものを余帰納的に定義された集合と呼び、νF と書く。

例えば単調増加数列全体を集めた集合は $F(S) = \{\mathrm{Cons}(x1, \mathrm{Cons}(x2, xs)) \mid x1 < x2 \wedge \mathrm{Cons}(x2, xs) \in S\}$ に対応する集合 νF である。このとき台集合 L は無限リスト全体の集合である。

図 1 は余帰納法を用いて証明できる定理の例であり、本研究で実装した自動証明ツールへの入力例でもある。この述語定義の記法は L. Simon らの提唱する co-LP(co-Logic Programming) [1] の記法とほぼ同じであり、違いは大文字が関数を表し小文字が変数を表すという点だけである。co-LP の述語定義の記法はプログラミング言語 Prolog と全く同じだが、述語を余帰納的に解釈する点が異なる。すなわち、述語の定義文から一意に定まる関数 F を用いて、F に対応する集合 νF として述語を定義する。例えば図 1 の述語 lexOrder に対応する F は次のようになる。$F(S)(x1, x2) = \exists a, as, b, bs.(x1 = \mathrm{Cons}(a, as) \wedge x2 = \mathrm{Cons}(b, bs) \wedge a < b) \vee (x1 = \mathrm{Cons}(a, as) \wedge x2 = \mathrm{Cons}(b, bs) \wedge a = b \wedge S(as, bs))$ また、assert 文では上で定義した述語を用いて定理を記述する。$\models$ の左側を前提部と呼び、右側を帰結部と呼ぶ。前提部の中のコンマは論理積を意味する。帰結部は述語変数適用（$p(x, y)$ など）ただ 1 つでなければならない。assert 文の中の変数はすべて全称量化される。本研究の手法で証明に成功すると、証拠として証明手順を表す証明木が得られる。

本論文は 4 つの節で構成される。第 2 節では自動証明手法を解説する。第 3 節では実装したツールの実験結果を述べる。第 4 節では関連研究との比較を

---

* Automating Coinductive Theorem Prooving. This is an unrefereed paper. Copyrights belong to the Authors.

Seiichi Shinomiya, Hiroshi Unno, 筑波大学大学院 システム情報工学研究科, Graduate School of Systems and Information Engineering, University of Tsukuba.

```
lexOrder(Cons(a, as), Cons(b, bs)) :- a < b.
lexOrder(Cons(a, as), Cons(b, bs)) :- a = b, lexOrder(as, bs).
assert(lexOrder(x, y), lexOrder(y, z) |= lexOrder(x, z)).
```

図 1　無限リスト上の辞書式順序の推移律

行う。

2　自動証明手法

本研究の提案手法では与えられた述語定義文と assert 文を元に初期ジャッジメントを作り、ジャッジメントに対して図 2 の推論規則を適用していくことで証明を行う。ジャッジメントとは推論規則中の $D; A; \phi \vdash C$ のことを指す。D は述語の定義の集合、A は前提部の論理式に含まれる述語変数適用を集めた集合、ϕ は前提部の論理式から述語変数適用を除いた論理式、C は帰結部の論理式である。

ジャッジメント $D; A; \phi \vdash C$ は各述語変数を余帰納的に解釈したもとで $\bigwedge A \wedge \phi \Rightarrow C$ が妥当であることを意味する。余帰納的な解釈とは、co-LP と同様の方法 [1] で D の述語定義文から関数 F を抽出し、F に対応する集合 νF として各述語変数の意味を与えることを指す。

$$
\begin{array}{rcl}
D & ::= & \{d_1, ..., d_m\} \\
d & ::= & P(\widetilde{t}) \text{ :- } \phi_1, ..., \phi_m, P_1(\widetilde{t_1}), ..., P_k(\widetilde{t_k}). \\
t & ::= & x \mid n \mid \text{true} \mid \text{false} \mid f(\widetilde{t}) \mid t_1 + t_2 \mid t_1 * t_2 \\
\phi & ::= & t_1 < t_2 \mid t_1 = t_2 \mid \phi_1 \wedge \phi_2 \mid \top \\
A & ::= & \{P_1(\widetilde{t_1}), ..., P_m(\widetilde{t_m})\} \\
C & ::= & t_1 < t_2 \mid t_1 = t_2 \mid C_1 \wedge C_2 \mid C_1 \vee C_2 \\
 & & \mid P(\widetilde{t}) \mid \exists x.C
\end{array}
$$

ここで t は項を表すメタ変数であり、P は述語変数を表すメタ変数である。x は項の変数を表し、n は整数の定数を表す。また、f は Cons などの構成子を表すメタ変数である。ただし $t_1 + t_2$ と $t_1 * t_2$ と $t_1 < t_2$ は整数型の項にのみ定義される。

図 1 の assert 文の場合、初期ジャッジメントは $A = \{\text{lexOrder}(x, y), \text{lexOrder}(y, z)\}$, $\phi = \top$, $C = \text{lexOrder}(x, z)$ となる。

COINDUCT 規則は余帰納法の原理を一般化した定理 $n \in \{1, 2, 3, ...\} \wedge S \subseteq F^n(S) \Rightarrow S \subseteq \nu F$ をジャッジメントに適用する規則である。F^n は関数 F を n 回合成した関数を表す。余帰納法の原理とは次の定理のことである。

> **定理 2.1 (余帰納法の原理)** 台集合を L として、任意の単調関数 $F : 2^L \to 2^L$ と任意の集合 S に対して $S \subseteq F(S) \Rightarrow S \subseteq \nu F$ が成り立つ

例えば図 1 の assert 文の内容は $S(as, cs) = \exists bs.\text{lexOrder}(as, bs) \wedge \text{lexOrder}(bs, cs)$ とし、$\nu F = \text{lexOrder}$ とすれば $S \subseteq \nu F$ と同値である。そのため余帰納法の原理を適用して、$S \subseteq F(S)$ を意味するジャッジメントに帰着できる。

COINDUCT 規則で使われている記号の意味は次の通り。fvs は与えられた論理式中に出現する変数の集合を返す関数である。$\widetilde{x}$ は変数の列である。$\{\widetilde{x}\}$ は変数の列を変数の集合として扱うことを表す。$\setminus$ は集合の差の演算子である。X はジャッジメントの前提部にのみ現れる変数の集合である。$\exists X$ は変数の集合の各要素を存在量化変数として扱うことを表す。F は帰結部の述語 νF を定義するために用いられた関数を表す。

UNFOLD 規則はジャッジメントの前提部の述語変数適用を述語の定義にしたがって展開する推論規則である。lexOrder のように 2 つ以上の節で定義される述語を展開する場合は証明木が枝分かれする。

VALID 規則は、証明木の枝を閉じるための推論規則である。この規則が適用できるのは妥当性判定に成功したときに限られる。妥当性判定とはジャッジメントの前提部の論理式が真のとき帰結部の論理式も真であることの判定である。本研究では、ジャッジメント $D; A; \phi \vdash C$ を論理式 $\bigwedge A \wedge \phi \Rightarrow C$ に変換して、代表的な SMT ソルバの 1 つである Z3 [2] に解かせる

$$\frac{X = \mathrm{fvs}(\bigwedge A \wedge \phi) \setminus \{\widetilde{x}\} \qquad S = \lambda\widetilde{x}.\exists X.\bigwedge A \wedge \phi \qquad n \in \{1,2,3,...\} \qquad D; A; \phi \vdash F^n(S)(\widetilde{x})}{D; A; \phi \vdash \nu F(\widetilde{x})}\ (\text{COINDUCT})$$

$$\frac{P(\widetilde{t}) \in A \qquad \sigma = \{\widetilde{x} \mapsto \widetilde{t}\} \qquad D; A \cup \sigma A'; \phi \wedge \sigma\phi' \vdash C \qquad (\text{for each } (P(\widetilde{x}) \text{ :- } \bigwedge A' \wedge \phi') \in D)}{D; A; \phi \vdash C}\ (\text{UNFOLD})$$

$$\frac{\models \bigwedge A \wedge \phi \Rightarrow C}{D; A; \phi \vdash C}\ (\text{VALID})$$

図 2　提案手法の推論規則

ことで妥当性判定を行う。Z3 は代数的データ型と未解釈関数を扱う機能を持っている。ジャッジメントを論理式に変換する際に、述語は Bool 型の値を返す未解釈関数としてエンコードし、Cons などの関数記号は代数的データ型のコンストラクタとしてエンコードする。このエンコードに基づく妥当性判定は健全だが完全ではない。

2.1　推論規則の適用戦略

本研究では次の戦略に従って推論規則を適用することで定理を証明する。

1. k = 1 とする
2. 与えられた入力から作ったジャッジメントに COINDUCT 規則 (n = k) を適用する
3. もし妥当性判定の結果が成功なら VALID 規則を適用して証明木の枝を閉じる
4. もし妥当性判定の結果が失敗なら
   (a) もし UNFOLD 規則の適用回数が閾値を超えていたら k = k + 1 として手順 2 に戻って証明を再試行する
   (b) UNFOLD 規則を適用し、生成された各ジャッジメントに対して手順 3 から証明を行う

本来、UNFOLD 規則は無限に適用し続けることができる。しかし UNFOLD 規則を何回か適用すると、それ以上 UNFOLD しても意味がない述語変数適用、すなわち妥当性判定の結果に影響を及ぼさないような述語変数適用が現れる。そこで、UNFOLD 規則の適用回数が一定値を超えたら証明を一旦打ち切り、k を 1 増やして COINDUCT 規則を適用し直して証明を再試行する。

UNFOLD 規則を適用する述語変数適用は幅優先探索の順序で選ぶ。すなわち、未展開の述語変数適用はキューで管理し、先頭の述語変数適用を UNFOLD 対象とする。UNFOLD 規則によって新たに追加された述語変数適用はキューの末尾に入る。

上記の手法で lexOrder の推移律 (図 1) を証明すると、次のような証明木が生成される。

```
COINDUCT
| UNFOLD lexOrder(x, y)
| | UNFOLD lexOrder(y, z)
| | | VALID
| | | VALID
| | UNFOLD lexOrder(y, z)
| | | VALID
| | | VALID
```

3　実験結果

上記の自動証明手法を実装し、既存の余帰納的定理の証明ツールである Circ [3] のオンライン・デモ・ページに掲載されているサンプルコードの定理群を自動証明させる実験を行った。61 個の定理のうち、14 個の定理の証明に成功し、47 個の定理の証明に失敗した。成功率が低い原因を分析したところ、Circ のサンプルコードで用いられている再帰関数には自分自身だけでなく他の再帰関数も呼び出すような複雑なものが多く含まれており、提案手法ではそのような再帰関数にうまく対応できていないことが分かった。また、そのような再帰関数に関する定理を証明するには、COINDUCT 規則を適用する前に UNFOLD 規則を何度か適用するとうまくいく可能性があることが分かった。提案手法を改良し、そのような再帰関数

に対応できるようになると実験結果は大幅に改善できると考えられる。一方、サンプルコードを Circ に証明させたところ全ての定理の証明に成功した。これは Circ で証明できる定理だけをサンプルコードとして掲載したためだと考えられる。そのため、この定理群は各ツールの性能を比較するためのベンチマークセットとしては不適切と言える。

我々は余帰納法を用いる定理証明器のために広く使われているベンチマークセットを探したが、そのようなものは見つけることができなかったので、新たにベンチマークセットを作成して評価実験を行った（表1）。既存の証明ツールである Dafny [4] と Circ にも同じ定理を証明させて結果を比較した。表中の ✓ は証明成功を、× は証明失敗を表す。また、定理中の s と t は無限リスト型の全称量化変数を表す。

全部で 24 個の定理のうち証明に成功した定理の数は提案手法が 21 個、Dafny が 12 個、Circ が 14 個という結果になった。Dafny でも Circ でも証明できない lexOrder の推移律や $\mathrm{inc}(s) \Rightarrow \mathrm{inc}(\mathrm{even}(s))$ などの定理も提案手法では証明できた。inc は単調増加数列の述語、even は無限リストの偶数番目の要素 ([0], [2], [4], ...) を取り出す関数である。

証明に失敗した定理の例として $\mathrm{zip1}(\mathrm{even}(s), \mathrm{odd}(s)) = s$ がある。この定理は余帰納法の原理 $S \subseteq F(S) \Rightarrow S \subseteq \nu F$ を素朴に適用しても証明できず、前提部を適当な集合 S' との和集合に弱めてから余帰納法の原理 $(S \cup S') \subseteq F(S \cup S') \Rightarrow (S \cup S') \subseteq \nu F$ を適用すれば証明できることが分かっている。この定理の場合、集合 S' の内容 $S'(z, s) = \exists x, o, x', s', e.\mathrm{odd}(\mathrm{Cons}(x, s), o) \wedge s = \mathrm{Cons}(x', s') \wedge \mathrm{even}(s', e) \wedge \mathrm{zip1}(o, e, z)$ は元の定理の証明を試みる過程で機械的に特定可能である。そのように、S' を特定して前提部を弱める機能を提案手法に追加する拡張は将来の課題である。

4 関連研究

Dafny は余帰納法を帰納法に帰着して SMT ソルバに解かせるというアプローチを採用している自動証明器である [5]。実験の結果、Dafny は再帰パターンの異なる関数や述語が混在する定理の証明が苦手だと分かった。例えば $\mathrm{even}(\mathrm{zip2}(s, t)) = s$ の証明には

表 1　実験結果

定理	提案手法	Dafny	Circ
lexOrder の推移律	✓	×	×
lexOrder の反射律	✓	✓	×
lexOrder の反対称律	✓	✓	×
$\mathrm{inc}(s) \Rightarrow \mathrm{inc}(\mathrm{even}(s))$	✓	×	×
$\mathrm{zip1}(s, t) = \mathrm{zip2}(s, t)$	✓	×	✓
$\mathrm{zip1}(\mathrm{even}(s), \mathrm{odd}(s)) = s$	×	×	✓
$\mathrm{even}(\mathrm{zip1}(s, t)) = s$	✓	×	✓
$\mathrm{even}(\mathrm{zip2}(s, t)) = s$	✓	✓	✓
$\mathrm{concat}(s) \Rightarrow \mathrm{alt}(s)$	✓	✓	×
無限二分木の mirror の対称律	✓	✓	×
$\mathrm{f}(\mathrm{morse}) = \mathrm{morse}$	×	×	✓
$\mathrm{even}(\mathrm{morse}) = \mathrm{morse}$	×	×	✓
$\mathrm{nots} \circ \mathrm{f} = \mathrm{f} \circ \mathrm{nots}$	✓	×	✓
$\mathrm{rev2}(\mathrm{rev2}(s)) = s$	✓	×	✓
$\mathrm{countUp}(s) \Rightarrow \mathrm{inc}(s)$	✓	✓	×
adds の交換律	✓	✓	✓
adds の単位元	✓	✓	✓
adds の結合律	✓	✓	✓
$\mathrm{nots} \circ \mathrm{even} = \mathrm{even} \circ \mathrm{nots}$	✓	×	✓
$\mathrm{negs} \circ \mathrm{even} = \mathrm{even} \circ \mathrm{negs}$	✓	×	✓
merge の単調増加保存性	✓	✓	×
adds の単調増加保存性	✓	✓	×
adds の upDown 保存性	✓	×	×
$\mathrm{nots}(\mathrm{nots}(s)) = s$	✓	✓	✓

成功しているが $\mathrm{even}(\mathrm{zip1}(s, t)) = s$ の証明には失敗している。even と zip1 の再帰パターンが噛み合わなかったことが原因だと考えられる。

Circ は項書き換えと cyclic proof に基づく自動証明システムである。実験の結果、Circ は順序関係などの述語に関する定理の証明が苦手だとわかった。例えば lexOrder の推移律、反射律、反対称律はどれも Circ では証明に失敗した。一般に、項書き換えに基づく定理証明系は不等式や述語の定理証明が苦手という事実が指摘されている [6]。Circ も同様の理由でそれらの定理証明に失敗したと考えられる。

5 結論と今後の課題

余帰納的に定義された述語に関する定理のための自動証明法を提案し、それを自動証明ツールとして実装した。実験の結果、既存ツール Circ のサンプルコードに掲載されている定理はあまり証明できなかったものの、順序関係などの述語に関する定理証明では Circ や Dafny よりも優れた性能を発揮することが分かった。

実験結果の節で触れたように COINDUCT 規則を適用する前に UNFOLD 規則を何度か適用するように推論規則の適用戦略を改良するのは今後の課題である。また、$S \subseteq F(S)$ の証明に失敗した場合に $(S \cup S') \subseteq F(S \cup S')$ の証明を試みるようにアルゴリズムを改良できると考えられる。今後これを実装し、自動証明できる定理の範囲を広げる予定である。さらに、証明に失敗した定理の一部は原因がまだ分かっていないので原因を明らかにし、アルゴリズムの改善につなげる。

参 考 文 献

[1] Luke Simon, Ajay Bansal, Ajay Mallya, and Gopal Gupta. Co-logic programming: Extending logic programming with coinduction. In *ICALP'07*, pp. 472–483. Springer, 2007.

[2] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS'08*, pp. 337–340. Springer, 2008.

[3] Dorel Lucanu, Eugen-Ioan Goriac, Georgiana Caltais, and Grigore Roşu. Circ: A behavioral verification tool based on circular coinduction. In *CALCO'09*, pp. 433–442. Springer, 2009.

[4] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR'10*, pp. 348–370. Springer, 2010.

[5] K. Rustan M. Leino and Michał Moskal. Co-induction simply. In *ISFM'14*, pp. 382–398. Springer, 2014.

[6] Takahiro Nagao and Naoki Nishida. Proving inductive validity of constrained inequalities. In *PPDP'16*, pp. 50–61. ACM, 2016.