修 士 論 文

# 手続きを含む命令型プログラムを検証するための証明戦術

令和 2年度 修了
三重大学大学院　工学研究科
博士前期課程　情報工学専攻
コンピュータソフトウェア研究室

**小島　裕登**

## 概 要

近年，プログラムの大規模化・複雑化に伴い，プログラムの安全性は様々な分野でますます重要となっている．プログラムの安全性を検証する方法には，動的テストやモデル検査，定理証明がある．動的テストやモデル検査と比べて定理証明は数学的帰納法を用いて，全ての入力について検査できる強みがある．しかし，実際のプログラムの検証には定理証明はあまり用いられていない．なぜなら，定理証明を用いて大規模なプログラムの正しさを検証する場合，人の手による証明だけでは膨大な時間コストが必要なためである．そのため，定理証明の実用化には自動証明が必要不可欠である．しかし，大規模で複雑なプログラムの検証の完全な自動化はほぼ不可能であり，人の手による証明は避けられない．そこで，人の手による証明の負担をできる限り削減する部分的な自動証明が期待される．

定理証明を用いて命令型言語のプログラムを検証する研究がされている．論理体系として Hoare 論理などが用いられる．Hoare 論理を用いたプログラムの検証では，事前条件と事後条件，ループ不変条件から代入文のような部分的プログラムの検証条件を求めることで証明を行う．その検証条件を自動生成する場合，最弱事前条件を用いるのが一般的である．しかし，手続き呼び出し文を含む場合，最弱事前条件で手続き呼び出し文の検証条件を生成するのは容易ではない．

そこで本研究では，Hoare 論理を用いて手続きを含む命令型プログラムの検証するための証明戦術を提案する．さらに，定理証明支援器 Coq を用いて自動証明を行う戦術を開発する．本手法は，非再帰手続きのプログラムの正当性を証明する場合は，最弱事前条件を用いて検証条件を自動的に生成する．また，再帰手続きのプログラムの正当性を証明する場合は，最弱事前条件と最強事後条件を組み合わせる事で検証条件を自動的に生成する．本手法の戦術を用いることで，元のプログラムの事前条件と事後条件，ループ不変条件から，検証条件を自動的に生成できる．自動生成する際に検証条件に関する表明の正当性を追加で検証する必要がある．しかし，表明の正当性については Coq で用意されている定理を用いることができるので容易に証明できる．本手法は，Hoare 論理によるプログラムの正当性の問題を，検証条件に関する表明の正当性の問題に帰着させる．また，検証条件に関する表明についても簡易的な正当性の検証を自動化するタクティクを開発した．

# 目 次

# 第1章　はじめに

## 1.1　研究背景

近年，ソフトウェアの大規模化・複雑化に伴い，ソフトウェアの正しさはますます重要となっている．その中でソフトウェアの正しさを証明する方法の 1 つに定理証明がある．定理証明は，数学や論理に基づいた技術である．定理証明は，数学的な定理の証明の他，コンパイラの正当性や型システムの健全性のようなプログラムの検証など多くの研究で利用されている．

Coq [1] は，関数型言語の特徴を持つ定理証明支援器の 1 つである．関数型言語の特徴を有するが，命令型言語の構文と意味論を形式化することで，命令型プログラムを扱える．また，命令型プログラムを検証する証明体系として，Hoare 論理 [2] がある．この論理体系を Coq で形式化することで，命令型プログラムも検証できる．

定理証明における検証は，ソフトウェアシステムが複雑になるほど，安全性の証明の手間が掛かる．Klein らの研究 [3] では，約 8,000 行の C コードを検証するのに 20 人年以上で 20 万行の Isabelle 証明スクリプトを要したという結果がある．この研究結果からも，定理証明を用いた複雑なシステムの検証は多大な手間が掛かることがわかる．人手による証明には限界があるため，自動証明が必要不可欠である．しかし，完全な自動証明は不可能である．そこで，証明の負担をできる限り軽減する部分的な自動証明が期待される．

## 1.2　研究目的

定理証明を用いて，プログラム検証のための証明の自動化に関する研究がされている．Cao らの研究 [4] では，Hoare 論理をヒープ領域に言及できるように拡張した分離論理を用いて，C 言語のサブセットを対象にしたプログラムの安全性を検証するための証明戦術を提案している．特に，リスト操作が関わるプログラムの検証を自動で証明できる証明戦術を提案している．また，検証条件の自動生成についての証明戦術も提案している．この研究では C 言語のサブセットでも `while` 文や `if` 文など基本的な文のみが対象となっており，手続き呼び出しは扱われていない．現実的なプログラムを扱う上で，手続き呼び出しは高頻度で現れるため，手続き呼び出しにも対応した自動証明が必要である．

そこで本研究では，手続き呼び出しを含む命令型プログラムを検証するための Coq の証明戦術を提案する．本研究では，Hoare 論理を論理体系として使う．さらに，手続き呼び出しを扱うために，最強事後条件と最弱事前条件を組み合わせることで自動戦術を開発する．本手法を用いることで，Hoare 論理によるプログラムの正当性の問題を，表明に関する正当性の問題に帰着させる．

## 1.3 本論文の構成

本論文の構成は以下の通りである．第 2 章は本研究で用いた定理証明支援器 Coq について解説する．第 3 章は本研究で対象とする命令型言語の構文や意味論についてと Hoare 論理について論じる．第 4 章は関連研究について論じる．第 5 章は手続きを含む命令型プログラムを検証する証明戦術について手法について詳しく論じる．第 6 章では第 5 章で述べた提案手法を実際の例に適用した場合について論じる．第 7 章はまとめである．

# 第2章　定理証明支援器Coq

本章では定理証明支援器 Coq の概要について論じる．また Curry-Howard 同型対応についてや Coq を用いた関数定義，命題の証明，関数の性質の証明について解説する．

## 2.1　Coqの概要

Coq [1] 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 は，型付きラムダ計算の Calculus of Constructions (CoC) に対し帰納的 (Inductive) 型を導入した 表現力の高い型付きラムダ計算である Calculus of Inductive Constructions と呼ばれる計算体系に基づいている．型や関数を定義することで，関数型プログラミング言語として用いられる．また，Curry-Howard 対応により，型に対応づけられる命題を証明することにより、型を持つ項を構成できる． Coq の型システムは依存型をもち，関数などの項をパラメータとして取る型を定義できる．証明する際には tactic と呼ばれる戦術を用いて対話的に証明する．

## 2.2　Curry-Howard同型対応

Coq は Curry-Howard 同型対応に基づいて，定理証明を行う．

Curry-Howard 同型対応とは，プログラム (ラムダ計算の項) と証明を, 型と命題を対応させる対応のことをいう．これにより，ある論理式の命題の証明が，命題に対応する型と，その型をもつプログラムの構成に帰着される．Curry-Howard 同型対応では，表 2.1 に示すような対応関係がある．

表 2.1: Curry-Howard 同型対応

| 論理 | コンピュータプログラム |
|---|---|
| 命題 | 型 |
| 証明 | プログラム |

また Coq の型システムは，依存型で表現力の高い型を扱える体系であるため，量化や述語を含む論理式も型で表せる．以下の表 2.2 は論理式と型の対応関係を表す.

## 2.3　関数定義

関数を定義するには `Definition` で，再帰関数を定義する場合は `Fixpoint` を用いる．以下の図 2.1 は `Definition` を用いて 2 つ値を引数にとり，それらを足し算する関数 `plus` を定義する例である．

表 2.2: 論理式と型の対応関係

| 論理 | Coq の世界 |
|---|---|
| 含意 (⇒) | 関数型 (→) |
| 連言 (∧) | 直積型 (and) |
| 選言 (∨) | 直和型 (or) |
| 全称量化 (∀) | 依存積型 (forall _ : _, _) |
| 存在量化 (∃) | 依存和型 (ex) |
| 真 | True 型 |
| 偽 | False 型 |

```
1 Definition plus (n m : nat) : nat :=
2    n + m.
```

図 2.1: 関数 plus

また，以下の図 2.2 は Fixpoint を用いて引数にとった値の階乗を求める関数を定義する例である．

```
1 Fixpoint real_fact (n : nat) : nat :=
2  match n with
3  | O → 1
4  | S m → n * real_fact m
5  end.
```

図 2.2: 関数 real_fact

それぞれ Definition と Fixpoint の後にきているのが関数名で，その後が仮引数とその型，その次が結果の型を表している．Fixpoint 中にある，match 構文によって，引数に関するパターンマッチを行っている．すなわち，引数が O の場合は 1 を返し，S m (n を m + 1 で表せる) の場合は再帰呼び出しの結果と掛け合わせる関数である．

Fixpoint は停止性が明らかな関数のみ定義できる．関数が停止するかどうかは Coq が判断する．停止性が明らかでないが停止する関数を定義する場合は Function などを用いて定義する．またその場合，関数の停止性について追加で証明をする必要がある．

## 2.4 命題の表現

以下の図 2.3 は命題論理における証明の例である．

2 行目の Proof が証明の始まり，7 行目の Qed が証明の終わりを表す．3〜6 行目の間にある intro や apply などが tactic と呼ばれる戦術である．この tactic を用いて証明を行っていく．Coq を用いた証明が，実際には何をしているかについて説明する．以下の図 2.4 は先ほどの図 2.3 を証明図で表したものである．

```
1 Theorem Test (P Q : Prop) : (P → Q) → P → Q.
2 Proof.
3    intro H.
4    intro H1.
5    apply H.
6    exact H1.
7 Qed.
```

図 2.3: 命題論理の証明例

$$\dfrac{\dfrac{\dfrac{P \to Q \quad P}{Q} \Rightarrow \text{apply H}}{P \to Q} \Rightarrow \text{intro H1}}{(P \to Q) \to P \to Q} \Rightarrow \text{intro H}$$

図 2.4: 証明図

以下の図 2.5 は `fun` $H\ H1 \Rightarrow H\ H1$ というプログラムの型が $(P \to Q) \to P \to Q$ であることを示している．

$$\dfrac{\dfrac{\dfrac{H : P \to Q \quad H1 : P}{H\ H1 : Q}}{\texttt{fun}\ H1 \Rightarrow H\ H1 : P \to Q}}{\texttt{fun}\ H\ H1 \Rightarrow H\ H1 : (P \to Q) \to P \to Q}$$

図 2.5: 推論木

図 2.5 からプログラム部分を除くと図 2.4 の論理式の証明が得られる．Coq の証明が型システムを基盤にしているのは，この発想に基づいている．つまり Coq では，論理式の証明を，プログラムの構成に帰着させている．このプログラムを証明とみなす考え方が 2.1 節で説明した Curry-Howard 同型対応である．

## 2.5 関数の性質の証明

以下の図 2.6 は 2 つの自然数の最大公約数を求める関数 `gcd` についての正当性を証明している例である．これはユークリッドの互除法を用いる時に重要となる最大公約数の性質について証明している．先ほどの命題論理と違い，このようにプログラムの正当性の証明も Coq で行える．

## 2.6 Ltac

また Coq には証明に用いる tactic を自作できる，Ltac [5] という言語が備わっている．本研究では，この Ltac を用いて自動戦術を実装する．図 2.7 に Ltac の実装例を示す．これは

```
1  Lemma gcd_neqz (m n : nat) : n <> 0 -> gcd m n = gcd n (m mod n).
2  Proof.
3    move=> H.
4    induction n.
5    -unfold not in H.
6     contradiction H.
7     reflexivity.
8    -simpl.
9     reflexivity.
10 Qed.
```

図 2.6: 最大公約数の性質の証明

先ほどの図 2.6 の一部の tactic をまとめたものである．例のように tactic を複数繋げて 1 つの自作 tactic で複数の tactic を実行するようにできる．また，パターンマッチング機能も備わっており，証明すべき目標の形に合わせて処理を分岐させることもできる．先ほどの図 2.6 の証明スクリプトが図 2.8 のように短く書ける．これをより一般化した自動戦術にすることで多くの証明の場面で手助けを行える．

```
1  Ltac my_tac x H :=
2    induction x;
3    [unfold not in H; contradiction H; reflexivity|
4     simpl; reflexivity].
```

図 2.7: Ltac の例

```
1  Lemma gcd_neqz (m n : nat) : n <> 0 -> gcd m n = gcd n (m mod n).
2  Proof.
3    move=> H.
4    my_tac n H.
5  Qed.
```

図 2.8: 最大公約数の性質の証明の自動化

# 第3章　命令型言語

本章では，本研究で扱う命令型言語の構文とその意味論について論じる．定理証明では構文と意味論を形式化することで命令型言語に関する推論が行える．また，Hoare 論理の概要と本研究で扱う命令文の Hoare 論理の公理と推論規則と表明の構文について論じる．

## 3.1　構文

まず，本研究で扱う命令型言語の構文について論じる．図 3.1 に本研究で対象とする命令型言語の構文を示す．$v$ は自然数の定数値を表す．$x$ はプログラム変数，$i$ は仕様変数を表す．プログラム変数とは，読み書き可能な変数であり，仕様変数は読出しのみ可能な変数である．$a$ は定数値，プログラム変数，仕様変数，加算などの一般的な算術式を表す．$b$ はブール式を表す．文 $s$ は，`skip` 文，代入文，ループ文，複合文，条件文，引数無し手続き呼び出しのいずれかを表す．ループ文には，ループ不変条件 $I$ が注釈付きとなっている．Coq の教科書に当たる [6] を参考に構文や意味論の形式化を行った．

$$
\begin{array}{rcl}
a & ::= & v \mid x \mid i \mid a + a \mid a * a \mid a - a \mid ... \\
b & ::= & \texttt{true} \mid \texttt{false} \mid a == a \mid a \leq a \mid b \wedge b \mid ... \\
s & ::= & \texttt{skip} \mid x ::= a \mid \{I\}\texttt{while}\ b\ \texttt{do}\ s \mid s\ ;;\ s \\
 & \mid & \texttt{ifb}\ b\ \texttt{then}\ s\ \texttt{else}\ s \mid \texttt{call}\ f
\end{array}
$$

図 3.1: 命令型言語の構文

Coq を用いて階乗計算を行う関数 fact の実装例を図 3.2，3.3 に示す．これらのプログラムは Coq 内に構文をデータ構造として形式化しただけなのでこのプログラム単体で実行はできない．図 3.2 の 4 行目にある `st` は，変数から自然数への関数を表す．図 3.3 の 6 行目の `fact` は，関数 `call_fact` を示す．

本研究では，引数無し手続きのみを扱うため，全ての変数をグローバル変数として扱うことで対処している．そのため，手続き呼び出しを行う前に本来引数として渡すはずの変数の前処理を行う必要がある．例えば，図 3.3 の `call fact` の前文の `x ::= x - 1` がそれに当たる．引数有りの手続きの場合の `call fact(x - 1)` の引数の処理に対応する．また本研究では，手続き呼び出し文は一度のみ現れるとし，フィボナッチ関数などは扱えない．相互再帰についても未対応である．また制限として，手続き本体に `while` 文が現れないものとする．これらについては今後の課題とする．

```
1 Definition loop_fact :=
2     z ::= x;;
3     y ::= 1;;
4     {{(st y) * fact (st z) =
          fact n}}
5     while ~(z == 0) do
6     y ::= y * z;;
7     z ::= z - 1
```

図 3.2: ループを用いた階乗関数

```
1 Definition call_fact :=
2    ifb (x == 0) then
3      y ::= 1
4    else
5      x ::= x - 1;;
6      call fact;;
7      x ::= x + 1;;
8      y ::= y * x
```

図 3.3: 再帰呼び出しを用いた階乗関数

## 3.2 意味論

次に，本研究で扱うプログラム意味論について論じる．算術式やブール式は表示的意味論を用いて意味を与える．表示的意味論は領域理論を基礎とし，プログラムの構成要素に，数学的な構造を対応させて意味を記述する方法である．[6] と同様に，再帰的な意味関数を与えることで式を評価する．

次に，文に意味を与える．文には操作的意味論を用いて意味を与える．操作的意味論とは，抽象的な計算機を定義して，プログラムの意味を抽象機械の状態の変化として意味を記述する方法である．つまり，プログラムの意味を「文はどのように振る舞うのか」といった操作とその結果を通してプログラムの意味を与える方法である．操作的意味論には，Plotkin によって提案された構造的操作的意味論 [7] と Kahn によって提案された自然意味論 [8] の 2 つある．両者とも一長一短であり使い分ける必要がある．本研究では，手続き文の自然意味論が広く知られているため，自然意味論を用いて意味を与える．自然意味論は，実行の初期状態と最終状態の遷移関係で意味を与える方法である．

本研究で扱う文に対する自然意味論の定義を図 3.4 に示す．手続きに関する操作的意味論は Nipkow の研究 [9] を参考にした．記法は [10] に倣って記述した．表記 $< s, \sigma >$ は，$s$ が文を表し，$\sigma$ が変数の状態を表し，変数から自然数への関数である．$< s, \sigma > \rightarrow \sigma'$ は，初期状態 $\sigma$ で文 $s$ を実行して，最終状態 $\sigma'$ で停止することを意味する．$\sigma[m/x]$ は，状態の更新を意味し，変数 $x$ の値が $m$ に変化した新たな状態を表す．$body$ は手続き名から関数本体プログラムへの関数を表す．

$$< \texttt{skip}, \sigma > \rightarrow \sigma$$

$$\frac{< a, \sigma > \rightarrow m}{< x := a, \sigma > \rightarrow \sigma[m/x]}$$

$$\frac{< s_1, \sigma > \rightarrow \sigma' \qquad < s_2, \sigma' > \rightarrow \sigma''}{< s_1; ; s_2, \sigma > \rightarrow \sigma''}$$

$$\frac{< b, \sigma > \rightarrow \texttt{true} \qquad < s_1, \sigma > \rightarrow \sigma'}{< \texttt{ifb}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2, \sigma > \rightarrow \sigma'}$$

$$\frac{< b, \sigma > \rightarrow \texttt{false} \qquad < s_2, s > \rightarrow \sigma'}{< \texttt{ifb}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2, \sigma > \rightarrow \sigma'}$$

$$\frac{< b, \sigma > \rightarrow \texttt{true} \qquad < s, \sigma > \rightarrow \sigma' \qquad < \{I\}\texttt{while}\ b\ \texttt{do}\ s, \sigma' > \rightarrow \sigma''}{< \{I\}\texttt{while}\ b\ \texttt{do}\ s, \sigma > \rightarrow \sigma''}$$

$$\frac{< b, \sigma > \rightarrow \texttt{false}}{< \{I\}\texttt{while}\ b\ \texttt{do}\ s, \sigma > \rightarrow \sigma}$$

$$\frac{< body\ f, \sigma > \rightarrow \sigma'}{< \texttt{call}\ f, \sigma > \rightarrow \sigma'}$$

図 3.4: 自然意味論

## 3.3 Hoare 論理

Hoare 論理 [2] は，命令型言語の正当性について推論するための形式的な理論である．Hoare 論理では，プログラムの性質を Hoare の三つ組 $\{P\}s\{Q\}$ で表し，$P$ と $Q$ を表明と呼び，$s$ はプログラムである．また，プログラム $s$ の実行前の表明 $P$ を事前条件，実行後の表明 $Q$ を事後条件と呼ぶ．この三つ組は,「事前条件 $P$ が成り立ち，プログラム $s$ を実行して停止するならば，その時，事後条件 $Q$ が必ず成り立つ」を意味する．図 3.5 に表明の構文を示す．$a'$ は 3.1 節で与えた算術式 $a$ を拡張したもので，Coq 内で定義された関数なども使えるように拡張した．

本論文では Hoare 論理の三つ組の具体例を用いながら提案手法を説明する．そのときに，$\{P\} \Rightarrow \{P'\}s\{Q\}$ や $\{P\}s\{Q'\} \Rightarrow \{Q\}$ のような記法を用いる．これらは，それぞれ事前条件を強める，事後条件を弱めることを意味する．

図 3.6 に，実際に Coq 内で階乗関数の正当性の問題を形式化した例を示す．n は仕様変数であり，関数 real_fact は Coq 内の関数型言語で定義され，正当性が検証された関数である．先ほど算術式 $a$ を拡張したのは，このように Coq 内の関数を扱えるようにするためである．

また，本研究で扱う Hoare 論理の規則を図 3.7 に示す．$C$ は事前条件と文と事後条件の組を要素とする集合を表す．$f$ は手続き名である．図 3.3 の 2 行目から 8 行目までが $body\ f$ に当たる．Coq の教科書 [11] を参考に Hoare 論理を Coq 内で形式化した．手続き文の Hoare 論理の規則は，理論として Hoare の研究 [12] を参考に，形式化として Nipkow の研究 [9] を参考に Coq 内で定式化をした．

$$
\begin{aligned}
P \quad ::= \quad & \texttt{true} \mid \texttt{false} \mid a' = a' \mid a' \leq a' \mid \neg P \mid P \wedge P \mid P \vee P \\
\mid \quad & P \Rightarrow P \mid \forall i.\ P \mid \exists i.\ P
\end{aligned}
$$

図 3.5: 表明の構文

```
1  Theorem loop_fact_correct (n : nat) :=
2     {(st x) = n}
3      z ::= x;;
4      y ::= 1;;
5      {{(st y) * real_fact (st z) = real_fact n}}
6      while ~(z == 0) do
7      y ::= y * z;;
8      z ::= z - 1
9      {(st y) = real_fact n}
```

図 3.6: 階乗関数の正当性

$$\frac{}{C \vdash \{P\}\texttt{skip}\{P\}}\ \text{hoare\_skip}$$

$$\frac{}{C \vdash \{Q[a/x]\}x ::= a\{Q\}}\ \text{hoare\_asgn}$$

$$\frac{C \vdash \{P\}s_1\{R\} \qquad C \vdash \{R\}s_2\{Q\}}{C \vdash \{P\}s_1;;s_2\{Q\}}\ \text{hoare\_seq}$$

$$\frac{C \vdash \{P \wedge b\}s_1\{Q\} \qquad C \vdash \{P \wedge \neg b\}s_2\{Q\}}{C \vdash \{P\}\texttt{ifb}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\{Q\}}\ \text{hoare\_if}$$

$$\frac{C \vdash \{I \wedge b\}s\{I\}}{C \vdash \{I\}(\{I\}\texttt{while}\ b\ \texttt{do}\ s)\{I \wedge \neg b\}}\ \text{hoare\_while}$$

$$\frac{P \Rightarrow P' \qquad C \vdash \{P'\}c\{Q'\} \qquad Q' \Rightarrow Q}{C \vdash \{P\}c\{Q\}}\ \text{hoare\_consequence}$$

$$\frac{C \cup \{(P, \text{call}\ f, Q)\} \vdash \{P\}body\ f\{Q\}}{C \vdash \{P\}\texttt{call}\ f\{Q\}}\ \text{hoare\_call}$$

$$\frac{(P, \text{call}\ f, Q) \in C}{C \vdash \{P\}\texttt{call}\ f\{Q\}}\ \text{hoare\_call2}$$

図 3.7: Hoare 論理

# 第4章　関連研究

## 4.1　Cao らの研究

Cao らの研究 [4] では，Hoare 論理をヒープ領域に言及できるように拡張した分離論理を論理体系として，C 言語のサブセットを対象にしたプログラムの安全性を検証するための証明戦術を提案している．この証明戦術は，表明に関する自動戦術と検証条件を自動生成する戦術の 2 種類に大きく大別される．表明に関する自動戦術では，特にリスト操作を行うようなプログラムを対象にして，自動証明の戦術を提案している．双方向リストに関しては未対応で今後の課題としている．

検証条件を生成する戦術では，1 つの検証条件生成サイクルを繰り返し実行する．こうすることで，途中で証明が失敗したり，検証条件が生成できない場合など戦術が失敗した場合に失敗したところまで生成される．これは，デバックする際に役立つ．失敗した時点まで検証条件の生成等を行うため，どういった理由で失敗したのかが判断しやすい．本研究でも，このアイデアを借り 1 ステップずつ繰り返し行うように実装した．

彼らの対象としている C 言語のサブセットはかなり基本的な文のみが対応している．手続きに関しては対象外となっている．

## 4.2　Appel らの研究

Appel らの研究 [13] では，論理体系としては分離論理を対象としている．こちらは，C 言語を検証するためのツールであり，検証済みツールとなっている．また，分離論理の正当性の証明では CompCert の意味論を用いている．

彼らの対象言語である C のサブセットはかなり広く，相互再帰や返り値など幅広く扱える．また証明戦術についても提案をしている．しかし，提案されている証明戦術自体はほとんど規則を適用するのみで自動化はされていない．また手続き文に対する証明においても，規則の適用の際に引数を指定する必要があり，多くの場面で人手による証明が必要となる．

以上から手続き呼び出しを含む命令型プログラムを対象として，検証条件を自動生成でき，表明の正当性についてもできる限り自動証明できるような証明戦術が必要である．

# 第5章　提案手法

本章では，手続き呼び出しを含むプログラムに対して検証条件を自動生成する方法について述べる．

## 5.1　証明戦術の流れ

提案手法の全体の流れについて説明する．以下の手順に従って検証条件の自動生成を実装した．

1. プログラムが非再帰手続きか再帰手続きかを確認する．再帰手続きの場合は 3.3 節で述べた，規則 hoare_call を適用する．非再帰手続きの場合は手順 5 の処理を行う．
2. 次に再帰手続きの停止条件に関する処理がくるため，Hoare 論理の `if` 文の規則を適用する．停止条件に対して処理を行う部分プログラムには，手順 5 の処理を行う．再帰手続きの停止条件に関する処理が無くなるまで以上を繰り返す．
3. 手続き呼び出し文が来るまで，Hoare 論理の複合文の規則を用いてプログラムを分解しながら，代入文の最強事後条件 [14] を用いて検証条件を計算する．
4. 事前条件と事後条件を適した条件に帰結の規則を用いて変化させて，規則 hoare_call2 を適用する．
5. 複合文の規則を用いて分解したのち，代入文の最弱事前条件 [15] を用いて検証条件を計算する．

以上の 5 つの手順を Coq の戦術として実装して用いて，検証条件を自動生成する．また，以上の手順の計算の擬似コードをアルゴリズム 1 に示す．

**アルゴリズム 1**

**Input:** $\{P\}s\{Q\}$

**Output:** 検証条件に関する表明の含意の集合 $Assn$

1: $Assn \leftarrow \emptyset$
   // 手順 *1*
2: **if** ($s \neq$ `call` $f$) **then**
3:  **return** $Assn \cup$ 最弱事前条件による検証条件生成 ($\{P\}s\{Q\}$)
4: **end if**
5: $\{P\}s\{Q\} \leftarrow$ hoare_call($\{P\}s\{Q\}$)
   // 手順 *2*
6: **while** $s =$ `ifb` $b$ `then` $s_1$ `else` $s_2$ **do**
7:  $\{P'\}s_1\{Q\}, \{P''\}s_2\{Q\} \leftarrow$`if` 文の規則 ($\{P\}s\{Q\}$)
8:  **if** ($s_1$ に `call` $f$ が含まれる) **then**
9:  $\{P\}s\{Q\} \leftarrow \{P'\}s_1\{Q\}$
10:  $Assn \leftarrow Assn \cup$ 最弱事前条件による検証条件生成 ($\{P''\}s_2\{Q\}$)
11:  **end if**
12:  **if** ($s_2$ に `call` $f$ が含まれる) **then**
13:  $\{P\}s\{Q\} \leftarrow \{P''\}s_2\{Q\}$
14:  $Assn \leftarrow Assn \cup$ 最弱事前条件による検証条件生成 ($\{P'\}s_1\{Q\}$)
15:  **end if**
16: **end while**
   // 手順 *3*
17: **while** $s \neq$ `call` $f;; s_1$ **do**
18:  $\{P\}s'\{R\}, \{R\}s''\{Q\} \leftarrow$ 複文の規則 ($\{P\}s\{Q\}$)
19:  $R \leftarrow$ 最強事後条件 ($s', P$)
20:  $\{P\}s\{Q\} \leftarrow \{R\}s''\{Q\}$
21: **end while**
22: $\{P\}$`call` $f\{R\}, \{R\}s_1\{Q\} \leftarrow$ 複文の規則 ($\{P\}s\{Q\}$)
23: $R \leftarrow$ 表明 $P$ において仕様変数がある項に変化したものを $Q$ に反映
   // 手順 *4*
24: $\{P'\}$`call` $f\{R'\} \leftarrow$ 帰結の規則 ($\{P\}$`call` $f\{R\}$)
25: $Assn \leftarrow Assn \cup \{P \Rightarrow P', R' \Rightarrow R\}$
26: $Assn \leftarrow Assn \cup$ hoare_call2($\{P'\}$`call` $f\{R'\}$)
   // 手順 *5*
27: $Assn \leftarrow Assn \cup$ 最弱事前条件による検証条件生成 ($\{R\}s_1\{Q\}$)
28: **return** $Assn$

## 5.2 提案手法の詳細

本節では各手順について詳細に説明する．その際に，以下の階乗を再帰手続きで計算するプログラムを例として用いる．

$$\{\} \vdash \{x = n\}\texttt{call fact}\{x = n \land y = \text{real\_fact } n\}$$

手続き fact の中身は以下である．

```
1    ifb (x == 0) then
2      y ::= 1
3    else
4      x ::= x - 1;;
5      call fact;;
6      x ::= x + 1;;
7      y ::= y * x
```

図 5.1: 手続き fact の中身

### 5.2.1 手順 1

プログラムが再帰手続きの場合は，hoare_call 規則を適用する．非再帰手続きの場合は，手順 5 のやり方で検証条件を求める．

#### 例

以下の規則を適用すると，

$$\frac{C \cup \{(P, \texttt{call } f, Q)\} \vdash \{P\}body\ f\{Q\}}{C \vdash \{P\}\texttt{call } f\{Q\}}\ \text{hoare\_call}$$

以下のようになる．

$$\frac{\{(x = n, \texttt{call fact}, x = n \land y = \text{real\_fact } n)\} \vdash \{x = n\}body\ \text{fact}\{x = n \land y = \text{real\_fact } n\}}{\{\} \vdash \{x = n\}\texttt{call fact}\{x = n \land y = \text{real\_fact } n\}}$$

*body* fact は図 5.1 に示されたプログラムなので展開する．これ以降の手順では以下の例を用いて説明を行う．導入した仮定について手続き文を証明する際に用いる．

```
1  {x = n}
2     ifb (x == 0) then
3       y ::= 1
4     else
5       x ::= x - 1;;
6       call fact;;
7       x ::= x + 1;;
8       y ::= y * x
9  {x = n ∧ y = real_fact n}
```

### 5.2.2 手順 2

手続き本体には再帰手続きの停止条件に関する条件分岐の処理が含まれている．そのため，Hoare 論理の `if` 文の規則を適用する．適用すると以下のように 2 つの部分プログラムの正当性に分けられる．

$$\frac{\{P \wedge b\} s_1 \{Q\} \qquad \{P \wedge \neg b\} s_2 \{Q\}}{\{P\}\texttt{ifb } b \texttt{ then } s_1 \texttt{ else } s_2\{Q\}}$$

一方は，停止した場合に対する処理をする部分プログラムの正当性．もう一方は，手続き呼び出し文を含む部分プログラムの正当性である．停止した場合に対する部分プログラムの検証条件の生成は手順 5 で求める．もう一方の部分プログラムに関しては，停止条件の処理が無くなるまで同様の処理を繰り返す．

#### 例

if 文の規則を適用すると以下のようになる．今回の場合，`if` 文の部分プログラムは停止に関する部分プログラムなので手順 5 の対象である．`else` の部分プログラムは手続き呼び出し文を含むため手順 3 の処理を行う．

```
1  {x = n}
2     ifb (x == 0) then
3  {x = n ∧ x = 0}
4        y ::= 1
5  {x = n ∧ y = real_fact n}
6     else
7  {x = n ∧ x <> 0}
8        x ::= x - 1 ;;
9        call fact;;
10       x ::= x + 1;;
11       y ::= y * x
12 {x = n ∧ y = real_fact n}
```

### 5.2.3 手順 3

#### 手続き呼び出し文の事前条件の計算

もしも手続き呼び出し文が無ければ手順 5 の最弱事前条件だけを用いて検証条件を生成できる．しかし，手続き呼び出し文が含まれる場合，手続き呼び出し文の最弱事前条件を求めるのは簡単ではない．Dijkstra の研究 [15] で提案された最弱事前条件は，プログラムと事後条件から事前条件を計算する方法である．手続き呼び出しの前で引数に関する処理を行う場合，最弱事前条件では引数に関する情報を得られず正しい検証条件が生成できない．

そこで，Floyd の研究 [14] で提案された代入文の最強事後条件を用いて，最初に手続き呼び出し文の事前条件を計算する．その後，求めた事前条件から手続き呼び出し文の事後条件を求める．最強事後条件とは，事前条件とプログラムから事後条件を計算する方法である．以下は，代入文の最強事後条件を表す．

$$\{A\}x := t\{\exists y.(x = t[y/x] \wedge A[y/x])\}$$

例えば，以下のように事前条件と代入文から事後条件 $R$ を求めることができる．

$$\{x = n\}x ::= x + 2\{R\}$$

$$\downarrow$$

$$\{x = n\}x :: x + 2\{x = n + 2\}$$

複合文の規則と代入文の最強事後条件を用いて，部分プログラムを前向きに推論していく．こうして，手続き呼び出し文の事前条件が自動的に求まる．

**例**

else 内の部分プログラムについて考える．現在以下の Hoare の三つ組となっている．ここから手続き呼び出し文の事前条件を求める．以下を複合文の規則により

$$\{x = n \wedge x \neq 0\} \quad x ::= x - 1;;\texttt{call fact};;x ::= x + 1;;y ::= y * x\{x = n \wedge y = \text{real\_fact } n\}$$

$$\downarrow$$

$$\{x = n \wedge x \neq 0\}x ::= x - 1\{R\}$$

$$\{R\}\texttt{call } fact;;x ::= x + 1;;y ::= y * x\{x = n \wedge y = real\_fact\ n\}$$

と 2 つの部分プログラムの正当性に分解する．さらに代入文の最強事後条件により

$$\{x = n\}x ::= x - 1\{R\}$$

$$\{R\}\texttt{call fact};;x ::= x + 1;;y ::= y * x\{x = n \wedge y = \text{real\_fact } n\}$$

$$\downarrow$$

$$\{x = n \wedge x \neq 0\}x ::= x - 1\{x = n - 1 \wedge n \neq 0\}$$

$$\{x = n - 1\}\texttt{call fact};;x ::= x + 1;;y ::= y * x\{x = n \wedge y = \text{real\_fact } n\}$$

事前条件 $\{x = n \wedge x \neq 0\}$ とプログラム $x ::= x - 1$ から，事後条件 $R$ が $\{x = n - 1 \wedge n \neq 0\}$ であることが求まる．求めた事後条件 $\{x = n - 1 \wedge n \neq 0\}$ が，手続き呼び出し文 `call fact` の事前条件となる．まとめると以下の 9 行目の検証条件が得られる．

```
1  {x = n}
2     ifb (x == 0) then
3  {x = n ∧ x = 0}
4        y ::= 1
5  {x = n ∧ y = real_fact n}
6     else
7  {x = n ∧ x <> 0}
8        x ::= x - 1 ;;
9  {x = n-1 ∧ n <> 0}
10       call fact;;
11       x ::= x + 1;;
12       y ::= y * x
13 {x = n ∧ y = real_fact n}
```

### 手続き呼び出し文の事後条件の計算

次に事後条件を計算する．計算する方法として，まず最強事後条件により，事前条件を変化させて事後条件を求めた．変化した部分は．事前条件の仕様変数がプログラム変数を含まない式に変化している点である．以下に例を示す．$x$ がプログラム変数，$n$ が仕様変数である．事前条件の仕様変数 $n$ が事後条件で式 $n-1$ に変化していることがわかる．

$$\{x = n\}x ::= x - 1\{R\}$$

$$\downarrow$$

$$\{x = n\}x ::= x - 1\{x = n - 1\}$$

この仕様変数からプログラム変数を含まない式への変化を，現在の事後条件にも反映させることで手続き呼び出し文の事後条件を求められる．

### 例

先ほど求めた手続き呼び出し文の事前条件 $\{x = n-1 \land x \neq 0\}$ は，事前条件 $\{x = n \land n \neq 0\}$ と代入文 $x ::= n - 1$ から求めた事後条件である．仕様変数 $n$ がプログラム変数を含まない式 $n-1$ に変化していることがわかる．現在の事後条件の $n$ を全て $n-1$ に置き換えれば良い．よって以下のように，手続き呼び出し文の事後条件を求めることができる．

$$\{x = n - 1 \land n \neq 0\} \quad \texttt{call}\ f;;x ::= x + 1;;y ::= y * x\{x = n \land y = \text{real\_fact}\ n\}$$

$$\downarrow$$

$$\{x = n - 1 \land n \neq 0\}\texttt{call}\ f\{x = n - 1 \land y = \text{real\_fact}(n-1) \land n \neq 0\}$$

$$\{x = n - 1 \land y = \text{real\_fact}(n-1) \land n \neq 0\}x ::= x + 1;;$$

$$y ::= y * x\{x = n \land y = \text{real\_fact}\ n\}$$

また，`if` 文の規則を適用した際にできた $\{n \neq 0\}$ についても事後条件に追加する．今までをまとめると以下の 11 行目の検証条件が得られる．

```
1  {x = n}
2     ifb (x == 0) then
3  {x = n ∧ x = 0}
4        y ::= 1
5  {x = n ∧ y = real_fact n}
6     else
7  {x = n ∧ x <> 0}
8        x ::= x - 1 ;;
9  {x = n-1 ∧ n <> 0}
10       call fact;;
11 {x = n-1 ∧ y = real_fact (n-1) ∧ n <> 0}
12       x ::= x + 1;;
13       y ::= y * x
14 {x = n ∧ y = real_fact n}
```

以上のことを推論規則として表すと以下のように表現できる．引数処理が複数ある場合は同様の処理を繰り返す．$n$ は仕様変数であり，$t$ はプログラム変数を含まない式．$[t/n]$ は変数 $n$ に項 $t$ を代入するという意味である．

$$\frac{\{P[t_0/n_0]...[t_i/n_i]\}\texttt{call } f\{Q[t_0/n_0]...[t_i/n_i]\} \qquad \{Q[t_0/n_0]...[t_i/n_i]\}s\{Q\}}{\{P[t_0/n_0]...[t_i/n_i]\}\texttt{call } f;;s\{Q\}}$$

以上により手続き呼び出し文の事前条件，事後条件が計算できる．

### 5.2.4 手順 4

手順 3 で手続き呼び出し文の事前条件と事後条件を求めたが，現在のままではまだ証明できない．

$$\{(P, \texttt{call } f, Q)\} \vdash \{P[t/n]\}\texttt{call } f\{Q[t/n]\}$$

現在の事前，事後条件が仮定の事前，事後条件の仕様変数 $n$ をプログラム変数を含まない式 $t$ に変化した表明のときに，手続き呼び出し文の正当性が証明できる．しかし，if 文の規則を用いた際に追加された再帰手続きの停止条件に関する仕様が表明に含まれており，現在は以下のような三つ組になっている．$C$ は停止条件に関する仕様を示す．

$$\{(P, \texttt{call } f, Q)\} \vdash \{P[t/n] \wedge C\}\texttt{call } f\{Q[t/n] \wedge C\}$$

ここで，帰結の規則を用いて $C$ を取り除いた表明に変化させる．そして，hoare_call2 の規則を適用することで証明ができる．

#### 例

今回の場合，

$$\{(x = n, \texttt{call fact}, x = n \wedge y = \text{real\_fact } n)\}$$

$$\vdash \{x = n - 1 \wedge n \neq 0\}\texttt{call fact}\{x = n - 1 \wedge y = \text{real\_fact } n \wedge n \neq 0\}$$

なので，$n \neq 0$ が $C$ に当たる．Hoare 論理の帰結の規則は，現在の事前条件を強め，事後条件を弱めることができる．この際に，表明の含意の正当性を追加で証明する必要がある．帰結の規則を適用することで以下の 10 行目，12 行目が得られる．

```
1  {x = n}
2     ifb (x == 0) then
3  {x = n ∧ x = 0}
4        y ::= 1
5  {x = n ∧ y = real_fact n}
6     else
7  {x = n ∧ x <> 0}
8        x ::= x - 1 ;;
9  {x = n-1 ∧ n <> 0}
10 ⇒{x = n-1}
11       call fact;;
12 {x = n-1 ∧ y = real_fact (n-1)}
13 ⇒{x = n-1 ∧y = real_fact (n-1) ∧ n <> 0}
14       x ::= x + 1;;
15       y ::= y * x
16 {x = n ∧ y = real_fact n}
```

### 5.2.5 手順 5

手続き呼び出し文が含まれていない部分プログラムに関しては，Dijkstra の研究 [15] で提案された，最弱事前条件を用いることで検証条件を計算できる．最弱事前条件とは，プログラムと事後条件から事前条件を求めるものである．複合文の規則を適用して代入文の最弱事前条件を後向きに推論していくことで事前条件が計算できる．代入文の最弱事前条件を以下に示す．

$$\{Q[t/x]\}x ::= t\{Q\}$$

例えば，以下のように事後条件と代入文から事前条件 $R$ を求められる．

$$\{R\}x ::= x * 2\{x = n\}$$

$$\downarrow$$

$$\{x * 2 = n\}x :: x + 2\{x = n\}$$

これを用いることで，残りの検証条件を生成できる．

#### 例

最弱事前条件を用いて残りの検証条件を計算すると以下の 4 行目，15 行目，17 行目が得られる．

```
1  {x = n}
2     ifb (x == 0) then
3  {x = n ∧ x = 0}
4  ⇒ {x = n ∧1 = real_fact n}
5        y ::= 1
6  {x = n ∧ y = real_fact n}
7     else
8  {x = n ∧ x <> 0}
9        x ::= x - 1 ;;
10 {x = n-1 ∧ n <> 0}
11 ⇒{x = n-1}
12       call fact;;
13 {x = n-1 ∧ y = real_fact (n-1)}
14 ⇒{x = n-1 ∧y = real_fact (n-1) ∧ n <> 0}
15 ⇒{x + 1 = n ∧y * (x + 1) = real_fact n}
16       x ::= x + 1;;
17 {x = n ∧ y * x = real_fact n}
18       y ::= y * x
19 {x = n ∧ y = real_fact n}
```

以上で手続き文の検証条件を生成ができ，Hoare 論理による階乗を計算するプログラムの正当性を証明できた．残るのは，以下の検証条件に関する表明の含意の正当性である．これらを全て証明できれば，正当性の証明は終わりである．

$$\{x = n \land x = 0\} \Rightarrow \{x = n \land 1 = \text{real\_fact } n\}$$

$$\{x = n - 1 \land n \neq 0\} \Rightarrow \{x = n - 1\}$$

$$\{x = n - 1 \land y = \text{real\_fact}(n - 1)\} \Rightarrow \{x = n - 1 \land y = \text{real\_fact}(n - 1) \land n \neq 0\}$$

$$\{x = n - 1 \land y = \text{real\_fact}(n - 1) \land n \neq 0\} \Rightarrow \{x + 1 = n \land y * (x + 1) = \text{real\_fact } n\}$$

## 5.3 まとめ

説明してきた手順を実装することで手続き文の検証条件を生成ができ，Hoare 論理によるプログラムの正当性の証明ができた．しかし検証条件を弱めたり，強めたりしているためそれらの表明の含意の正当性について，追加で証明する必要がある．本研究で提案する証明戦術は，Hoare 論理による正当性の問題を，表明の含意に関する正当性の問題に帰着させる．

また，表明の含意の正当性の証明についても，簡単な自動戦術を実装した．Coq で標準的に備わっている自動戦術 omega を利用したものになっている．omega はプレスバーガー算術と呼ばれる決定手続きを実装している．これは，加算や減算，定数の乗算，大小比較などを自動で証明してくれる．また，ヒントデータベース機能が Coq には備わっており，よく用いる補題などをデータベースに登録しておくことで，自動戦術を実行する際にそれらの補題を用いて証明を試みる．これらにより，表明に関する正当性の証明に関しても証明の手間を削減した．

# 第6章　実験

本章では，実際に再帰手続きで実装した図 6.1 の階乗プログラムの正当性を証明した際の，自動化前と後の証明スクリプトを比較する．

```
1    ifb (x == 0) then
2      y ::= 1
3    else
4      x ::= x - 1;;
5      call fact;;
6      x ::= x + 1;;
7      y ::= y * x
```

図 6.1: 再帰的な階乗プログラム

図 6.2 は，自動化前の証明スクリプトを示す．図 6.1 の 7 行ぐらいのプログラムの正当性の証明でも，50 行近くの証明スクリプトが必要になる．このことからも，大規模システムの検証には人手による証明だけでは厳しいことがわかる．1〜4 行目の fact_spec は手続き呼び出し文が満たすべき性質の定義である．手続き文の規則は，手続き文が成り立つと仮定して証明を行う．そのため，仮定として導入する際にこの定義を用いる．7 行目にある body は，手続き名から手続き本体への関数である．26 行目や 30 行目のように，証明者自身が適切な検証条件を適宜与えながら証明を行う．また，表明の正当性に関する証明部分は，19〜22 行目や，43〜54 行目に当たるが，同じ処理をする tactic が与えらるていることがわかる．

```
1  Definition fact_spec n :=
2       ((fun st => asnat(st x) = n),
3        (call f),
4         (fun st => asnat (st x) = n /\ asnat(st y) = real_fact n)).
5
6  Theorem callrecfact_correct (X : nat) :
7      body; Empty_set _ |- {{fun st =>  asnat(st x) = X}}
8        call f
9        {{fun st => asnat (st x) = X /\ asnat(st y) = real_fact X}}.
10 Proof.
11   apply: (hoare_call body _ _ _ _ (BIGCUP X fact_spec) _).
12   apply: In_BIGCUP; try omega.
13   now rewrite /body.
14   rewrite t_update_eq /fact.
15   apply: hoare_if.
16   apply: hoare_consequence_pre.
17   apply: hoare_asgn.
18   move=> st.
19   rewrite /assn_sub /bassn /t_update //=.
20   move=> [H1 H2].
21   rewrite -> eqb_eq in H1.
22   rewrite -H2 H1 //=.
```

```
23    apply: hoare_seq.
24    apply: hoare_asgn_fwd.
25    simpl.
26    apply: (hoare_seq _ _ (fun st : state => (X=? 0) <> true /\ asnat
          (st x) = (asnat(eval (x !-> Nat X; st) (x - 1))) /\ asnat (st
          y) = real_fact (asnat(eval (x !-> Nat X; st) (x - 1))))).
27    simpl.
28    apply: hoare_conseq.
29    move=> st [H0 H1].
30    exists (fun st => asnat (st x) = X - 1), (fun st =>  asnat (st x)
          = X - 1 /\ asnat (st y) = real_fact (X - 1)).
31    split.
32    apply: hoare_call2.
33    apply: Union_intror.
34    apply: In_BIGCUP; try omega.
35    split.
36    - apply: H1.
37    - move=> ? [? ?].
38      tauto.
39    apply: hoare_consequence_pre.
40    apply: hoare_seq.
41    apply: hoare_asgn.
42    apply: hoare_asgn.
43    rewrite /assn_sub /bassn /t_update //=.
44    move=> st [a [H0 H1]].
45    rewrite -> not_true_iff_false in a.
46    rewrite -> eqb_neq in a.
47    split.
48    rewrite H0.
49    rewrite sub_add; try omega.
50    rewrite H0 H1 sub_add.
51    rewrite mult_comm.
52    apply: fact_lemma.
53    omega.
54    omega.
55  Qed.
```

図 6.2: 自動化前：再帰手続き階乗関数の証明スクリプト

図 6.3 は，5 章で説明した自動戦術を用いた場合の証明スクリプトである．50 行近くあった証明スクリプトを 4 行まで短縮できた．6 行目の hoare_tactic が検証条件を自動生成する戦術で，assn_simp が表明の正当性に関して自動証明を行う戦術である．hoare_tactic には，引数として手続き呼び出し文が満たすべき性質の定義を与える．証明すべき表明の含意の正当性は 5.2.5 節で示した 4 つであったが，assn_simp により 3 つは自動的に証明され 1 つのみ人手により証明し．他にもいくつか証明を行い，証明コストが削減できていることを確認した．

```
1  Theorem callrecfact_correct (X : nat):
2      body; Empty_set _ |- {{fun st =>  asnat(st x) = X}}
3        call f'
4        {{fun st => asnat (st x) = X /\ asnat(st y) = real_fact X}}.
5  Proof.
6    (hoare_tactic fact_spec);assn_simp.
7    rewrite mult_comm.
8    rewrite sub_add; try omega.
9    apply: fact_lemma; omega.
10 Qed.
```

図 6.3: 自動化後：再帰手続き階乗関数の証明スクリプト

# 第7章　結論

本研究では，再帰手続きを含む命令型プログラムの正当性の検証において，検証条件を自動的に生成する証明戦術を提案した．再帰呼び出しを含む場合，最弱事前条件のみでは，引数の処理を無視してしまう問題があり，検証条件を自動生成することが難しいため，最強事後条件と最弱事前条件を組み合わせることで解決した．本手法により，Hoare 論理によるプログラムの正当性の検証の問題を，表明に関する正当性の検証の問題に帰着させた．また，表明の含意に関する正当性の自動証明する戦術についても実装した．

今後の課題は，現在は変数を全てグローバル変数として扱っているため，引数を持つ手続きが扱えていないので，ローカル変数を扱えるようにすることである．これは変数の状態環境をグローバルとローカルの 2 つに増やし意味論を拡張することで解決できると考えている．また，相互再帰についても扱えていないので，相互再帰を扱えるようにすることも今後の課題である．

# 謝辞

本研究を進めるにあたり, 日頃からご指導して頂いた山田俊行講師, 河内亮周教授, 森本尚之講師, 多くの助言や意見をくださったコンピュータソフトウェア研究室の皆様, 研究活動における様々な場面でお世話になりました落合美子事務員に感謝いたします.

# 参考文献


[1] The Coq Development Team: The Coq Proof Assistant, (online), `https://coq.inria.fr/` (accessed 2020-12-28).

[2] Hoare, C. A. R.: "An Axiomatic Basis for Computer Programmin", *Commun. ACM*, Vol. 12, No. 10, pp. 576–580 (1969).

[3] Klein, G. *et al.*: "SeL4: Formal Verification of an OS Kernel", *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, Association for Computing Machinery, pp. 207–220 (2009).

[4] Cao, J., Fu, M. and Feng, X.: "Practical Tactics for Verifying C Programs in Coq", *Proceedings of the 2015 Conference on Certified Programs and Proofs*, pp. 97–108 (2015).

[5] Delahaye, D.: "A Tactic Language for the System Coq", *Proceedings of the 7th International Conference on Logic for Programming and Automated Reasoning*, pp. 85–95 (2000).

[6] Pierce, B. C. *et al.*: *Logical Foundations*, Software Foundations, Vol. 1, Electronic textbook (2020). Version 5.8, `http://softwarefoundations.cis.upenn.edu`.

[7] Plotkin, G. D.: "A structural approach to operational semantics", *Journal of Logic and Algebraic Programming*, pp. 17–139 (2004).

[8] Kahn, G.: "Natural Semantics", *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science*, Springer-Verlag, pp. 22–39 (1987).

[9] Niplow, T.: "Hoare Logics for Recursive Procedures and Unbounded Nondeterminism", *Proceedings of the 16th International Workshop and 11th Annual Conference of the EACSL on Computer Science Logic*, Berlin, Heidelberg, Springer-Verlag, pp. 103–119 (2002).

[10] Winskel, G.: *The Formal Semantics of Programming Languages: An Introduction*, MIT Press (1993).

[11] Pierce, B. C. *et al.*: *Programming Language Foundations*, Software Foundations, Vol. 2, Electronic textbook (2020). Version 5.8, `http://softwarefoundations.cis.upenn.edu`.

[12] Hoare, C. A. R.: "Procedures and Parameters: An Axiomatic Approach", *Proceedings of Symposium on the Semantics of Algorithmic Languages* (Engeler, E., ed.), Springer-Verlag, pp. 102–116 (1971).

[13] Appel, A. W. *et al.*: *Program Logics for Certified Compilers*, Cambridge University Press, USA (2014).

[14] Floyd, R. W.: "Assigning meanings to programs", *Proceedings of Symposium on Applied Mathematics*, Vol. 19, pp. 19–32 (1967).

[15] Dijkstra, E. W.: *A Discipline of Programming*, Prentice-Hall (1976).