Coq と Isabelle を用いたユークリッド原論の定理証明

岩間詞也 [a], 高橋正 [b]

[a] 甲南大学大学院 自然科学研究科 知能情報学専攻
神戸市東灘区岡本８－９－１, 658-8501
[b] 甲南大学知能情報学部知能情報学科
神戸市東灘区岡本８－９－１, 658-8501

概 要

ソフトウェアを用いて数学の定理の証明を行うことについては，多くのソフトウェアが開発されている。本研究では，定理証明ソフトウェア Coq と Isabelle を用いてユークリッド原論の定理証明を行った。この活動によってプログラミング技術と数学の本質についての理解を深めることが支援できる。Coq および Isabelle を用いて同じユークリッド原論の定理証明を行い, それぞれの特徴と差異を調べた。

キーワード： Coq，Isabelle/HOL，ユークリッド原論，定理証明

1 はじめに

Coq と Isabelle は，定理証明支援ソフトウェア（プルーフアシスタントソフトウェア）である [1-3]。我々は，Coq と Isabelle を用いて，ユークリッド原論の定理を証明する活動を提案する。その目的は，数学教育におけるプログラミングの再興である。かつて数学教育における探究的な活動として，プログラミングを利用した活動が試みられた。しかし，その活動は成功しなかった。その理由は，どのような学習目的のために，どのようなプログラミング技術を用い，それが自らのどのような思考と対応させることができるかという，最も基本的かつ重要な議論ができていなかったからである。

Coq と Isabelle は数式を形式言語で表わせ，論理的演算でそれらの手法を証明するための，提供された道具（ソフトウェア）であり，Coq と Isabelle を用いてユークリッド幾何学の定理を証明することが可能である。我々は，上記の数学教育における探究的な活動の再興を目指し，定理証明支援ソフトウェアを用いて，この課題に挑戦する。数学の学習は，数学的な考察を必要とする，主に精神的な活動である。Coq と Isabelle を用いることは，関係する概念だけでなく手順への洞察を必要とし，Coq と Isabelle の使用は手順に対処する学習者の数学的な概念に影響を及ぼす。

2　ユークリッド原論

現代数学の特徴は，公理と呼ばれるいくつかの命題を前提にして他の命題を証明し，「演繹的体系」として構成することである。この意味での数学の始まりは，古代ギリシャにおける幾何学である。「図形の論証」はユークリッドの『原論』に始まる。エジプトの測量術に端を発しタレスが理論的に研究したものが幾何学の始まりであるといわれている。それを書物としてまとめたものがユークリッドの『原論』である。「演繹的体系」を構成するうえで証明なしに使われる基本的な命題である「公理」として，ユークリッドの『原論』では，幾何学的性質である 5 つの「公準」と一般的な 8 つの「公理」が挙げられている [4]。

3　Coq による証明

Coq [1,2] を用いての証明過程を以下に示す。

3.1　Coq の起動

Coq を起動すると，右図のように 3 つの領域に分かれた画面が立ち上がる。番号を付けた 3 つの領域の役割は以下のようになっている。

1. script buffer (スクリプトバッファ)
   : 入力用
2. goal window (ゴールウィンドウ)
   : 証明済みの過程や証明すべき結論 (サブゴール) 等の表示
3. メッセージやエラーの表示

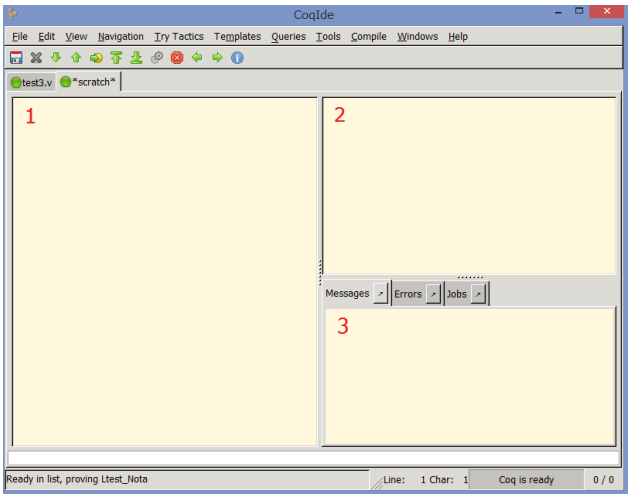


図 1: Coq の起動

プログラムや証明をスクリプトバッファへ打ち込み, それらを Navigation コマンド (画面上枠の緑矢印コマンド) によって読み込ませることでスクリプトが進行する。途中にエラー箇所があった場合，以降に書かれたスクリプトは実行されなくなる。

3.2　証明の流れ

Coq における証明の基本的な流れを示す。始めに，証明に必要な型として，点と線分の宣言を行う。

```
Definition Point := nat.
Definition Line := Point * Point : Set.
```

Point は自然数一つの型，Line は 2 つの Point からなる集合という形で定義される。Line A（線分 A）を定義したとき，“fst A”および “snd A”とすると, 線分 A を構成するそれぞれの点を指定できる。

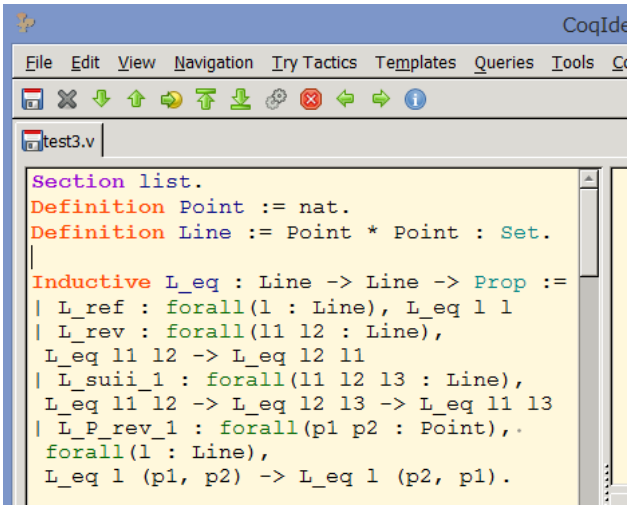


図 2: 関数の宣言

続けて，新たに定義した型に関する関数を宣言する。右図では，関数定義に使われるコマンドの一つ，Inductive を用いて新たな関数 “L_eq” と，それを結論に含む 4 つの関数を定義している。Inductive の特徴として，始めに指定した関数を結論に含む関数しか定義できないというものがある。

4 つの関数を簡便に表記すると, 以下のようになる。以下では，点 A と点 B をつなぐ線分 AB を AB と表記する。

- L_ref : AB = AB (反射律)
- L_rev : AB = CD ならば CD = AB (対称律)
- L_suii_1 : AB = CD かつ CD = EF ならば AB = EF (推移律)
- L_p_rev_1 : AB = CD ならば AB = DC

これらを用いた例題を解く過程を以下に示す。

例題 :
3 つの線分 s1, s2, s3 があるとき,
s1 = s2 かつ s2 = s3 ならば s3 = s1 である。

Variable : 補題や定理に使用する変数の宣言。ここでは 3 つの線分 s1, s2, s3 を宣言している。

Coq の証明は各コマンドを一つずつ読み込ませることで進行する。また，読み込みが一つ進むたびに，その時点での証明状況がゴールウィンドウに表示される。

```
Variable s1 s2 s3:Line.
Lemma Ltest1 :
L_eq s1 s2 -> L_eq s2 s3
-> L_eq s3 s1.
Proof. intros.
apply L_rev.
specialize (L_suii_1 s1 s2 s3). intro.
specialize (H1 H).
specialize (H1 H0).
trivial.
Qed.
```

図 3: 線分 s1, s2, s3 の宣言

証明開始を示す Proof. と，各前提と結論の分解を行う intros. まで証明を進めた状態が，次の図 4 である。

s1 = s2 および s2 = s3 を前提として，証明すべき結論 s3 = s1 を表示している。このときの H, H0 は各前提に自動的につけられる名前である。

まず，既存の定義 L_rev (対称律) を適用させ，結論の書き換えを行う。

apply L_rev を適用した結果が図 5 である。

```
Lemma Ltest1 : L_eq s1 s2 -> L_eq s2 s3
-> L_eq s3 s1.
Proof.
intros.
apply L_rev.
specialize (L_suii_1 s1 s2 s3). intro.
specialize (H1 H).
specialize (H1 H0).
trivial.
Qed.
```

```
1 subgoal
s1, s2, s3 : Line
H : L_eq s1 s2
H0 : L_eq s2 s3
_____________________________________(1/1)
L_eq s3 s1
```

図 4: 結論の分解を行うまで証明を進めた状態

結論 : L_eq s3 s1 が L_eq s1 s3 に変化した。

この結論と前提とを繋ぐべく，既存の定義 L_suii_1 を，証明中の問題に合わせた形で前提に加える。

```
1 subgoal
s1, s2, s3 : Line
H : L_eq s1 s2
H0 : L_eq s2 s3

_____________________________________
L_eq s1 s3
```

図 5: apply L_rev を適用した結果

specialize (“既存の定義，補題および前提” “引数”) とすることで，指定した定義，補題および前提に，引数を適用させられる。ここでは forall で構築することで 3 つの引数を受け付けられる状態にある定義 L_suii_1 に対して，変数 s1, s2, s3 を指定している。

spcialize (L_suii_1 s1 s2 s3). および intros. を適用した結果が図 6 である。

新たな前提 H1 という形で，L_suii_1 の変数が s1, s2, s3 に置き換えられたものが追加された。

Coq に元々存在する組み込み関数により，“A -> B” と “A” という前提があれば，“B”という前提を導くことができる。

ここからは，この前提を使った前提の書き換えを用いて，結論と同じものを作り出すことを目指す。使用するのは先ほども利用した specialize コマンドである。

```
1 subgoal
s1, s2, s3 : Line
H : L_eq s1 s2
H0 : L_eq s2 s3
H1 : L_eq s1 s2 ->
     L_eq s2 s3 -> L_eq s1 s3
______________________________________
L_eq s1 s3
```

図 6: intros. を適用した結果

```
spcialize (H1 H).
spcialize (H1 H0).
```

それぞれ前提 H1 を H, および H0 で書き換える。それぞれを順に適用した結果が図 7 である。

前提 H1 から “->”が除去され，結論に等しい前提が現れた。

ここで, trivial. を適用することで副題の証明完了を指定する。

```
1 subgoal
s1, s2, s3 : Line
H : L_eq s1 s2
H0 : L_eq s2 s3
H1 : L_eq s2 s3 -> L_eq s1 s3
______________________________________
L_eq s1 s3
```

```
1 subgoal
s1, s2, s3 : Line
H : L_eq s1 s2
H0 : L_eq s2 s3
H1 : L_eq s1 s3
______________________________________
L_eq s1 s3
```

図 7: 前提 H1 を H および H0 で書き換えた結果

全ての副題 (今回は 1 つ) の証明が完了したことで，図 8 のような表示がされる。

```
No more subgoals.
```

図 8: 証明の完了

最後に証明終了を示す Qed. を適用して証明モードを終了する。

3.3 略記

証明中に幾度も登場する関数を略記で表現させるコマンドが存在する。

以下は，“L_eq x y”を “x [@] y”と表示させるようにする命令である。

```
Definition Point := nat.
Notation ”( x [@] y )” := (L_eq x y).
```

このコマンドを利用し，前節と同じ問題の図 6 まで証明を進めた様子が図 9 である。

複雑な証明ほど，その証明ステップが長大となり, 同時に証明過程を示すゴールウィンドウの表示が煩雑となる。Notation コマンドを活用することで，表示される文字数を減らし，証明の煩雑化を緩和することができる。

証明中に略記の内容を確認する場合は，Print コマンドを使用する。“(x [@] y)”の内容を確認する場合は以下のような命令となる。

```
Print ”( x [@] y )”.
```

”(x [@] y)”の内容，即ち “L_eq”の内容が画面右下の枠内に表示される。

```
Notation "( x [@] y )" := (L_eq x y).

Lemma Ltest_Nota : (s1 [@] s2)
-> (s2 [@] s3) -> (s3 [@] s1).
Proof.
intros.
apply L_rev.
specialize L_suii_1. intro.
specialize (H1 s1 s2 s3).
specialize (H1 H).
specialize (H1 H0).
trivial.
Qed.
```

```
1 subgoal
s1, s2, s3 : Line
H : (s1 [@] s2)
H0 : (s2 [@] s3)
H1 : (s1 [@] s2) ->
     (s2 [@] s3) -> (s1 [@] s3)
______________________________________
(s1 [@] s3)
```

図 9: 略記コマンドを利用して証明を進めた様子

3.4 Coq による命題の証明

Coq によるユークリッド原論の証明に入る。ここでは，原論 Book1 の命題 1_2 :「与えられた点において，与えられた線分に等しい線分をつくること」を対象とする (命題 1_2 の証明には，本来命題 1_1 の証明と，円と円周上の点に関する定義が必要だが，ここでは省略する)。

先ずは前節と同様に，必要な関数を定義する。

LPL は線分+線分の型である。

各関数を簡便に表記すると，以下のようになる (以下, 線分 AB+線分 CD を AB+CD と表記する)。

```
Definition LPL := Line * Line : Set.

Inductive LPL_eq : LPL -> LPL -> Prop :=
| LPL_ref : forall(lp : LPL), LPL_eq lp lp
| LPL_rev : forall(lp1 lp2 : LPL),
 LPL_eq lp1 lp2 -> LPL_eq lp2 lp1
| LPL_suii_1 : forall(lp1 lp2 lp3 : LPL),
 LPL_eq lp1 lp2 -> LPL_eq lp2 lp3
 -> LPL_eq lp1 lp3
| LPL_trans_1 : forall(l1 l2 l3 l4 : Line),
 L_eq l1 l2 -> L_eq l3 l4 ->
 LPL_eq (l1,l3) (l2,l4).
```

図 10: 必要な関数の定義

- LPL_ref : AB+CD = AB+CD (反射律)。
- LPL_rev : AB+CD = EF+GH ならば EF+GH = AB+CD (対称律)。
- LPL_suii_1 : AB+CD = EF+GH かつ EF+GH = IJ+KL ならば AB+CD = IJ+KL (推移律)。
- LPL_trans_1 : AB = CD かつ EF = GH ならば AB+EF = CD+GH。
  原論の Book1 公理 2「等しいものに等しいものが加えられれば, 全体は等しい」より

さらに追加で，図 11 のような定義を行う。LPL の定義と同時に行わないのは，Inductive の仕様 (結論に指定した関数が含まれなければならない) に沿わない為である。

右は，“AB+CD = EF+GH かつ AB = EF ならば CD = GH.”を意味する。

(原論の Book1 公理 3「等しいものから等しいものがひかれれば，残りは等しい」より)

```
Definition LPL_trans_2
 : forall(l1 l2 l3 l4 : Line),
 LPL_eq (l1,l2) (l3,l4) ->
 L_eq l1 l3 -> L_eq l2 l4.
Admitted.
```

図 11: 追加の定義

図 12 のように，Notation コマンドで LPL_eq を略記できるようにしておく。

```
Notation "( x [@@] y )"
  := (LPL_eq x y).
```

図 12: Notation コマンドでの LPL_eq の略記

ここで，命題 1_2 の証明に必要となる，命題 1_1 :「与えられた線分の上に正三角形をつくること」(与えられた線分に，その 2 点それぞれと繋ぐことで正三角形となるような点をつくる) を定義として宣言する (図 13)。

```
Definition def_Prop1_1
(l : Line)(p : Point): Prop :=
L_eq l (fst l ,p) /\ L_eq l (snd l ,p).
```

図 13: 必要な定義の宣言

“線分 AB と点 P が命題 1_1 の関係にある とは，AB = AP かつ AB = BP である” という内容になっている (今回の証明に必要な箇所のみを抜き出した形である)。

ここから証明に入る。まずは，改めて命題 1_2 の内容を示す。

命題 1_2. 与えられた点において与えられた線分に等しい線分をつくること。

・原論における証明手順

- 仮定

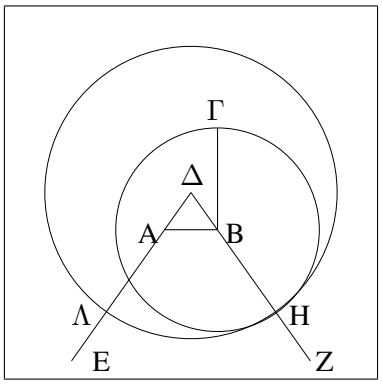
a1. 与えられた点を A，線分を BΓ とする。

a2. 線分 AB の上に等辺三角形 ΔAB をつくる (命題 1)。

a3. ΔA を延長して線分 AE，ΔB を延長して線分 BZ をそれぞれ描く。

a4. 中心 B，半径 BΓ をもって円を描き，BZ との交点を H とする。

a5. 中心 Δ，半径 ΔH をもって円を描き，AE との交点を Λ とする。



- 証明

P1. a5 より，ΔΛ = ΔH (ΔA+AΛ = ΔB+BH)。

P2. a4 より，BH = BΓ。よって ΔB+BH = ΔB+BΓ。

P3. P1 P2 より，ΔA+AΛ = ΔB+BΓ。

P4. a2 より，ΔA = ΔB。

S1. P3 P4 より，AΛ = BΓ。

・ S1 より，点 A において線分 BΓ に等しい線分 AΛ がつくられている。

これらの仮定および結論を Coq で表現したものが次のようになる (図 14)。

前提 1. AB の上に等辺三角形 ΔAB となる Δ がつくられている.

前提 2. BΓ = BH.

前提 3. ΔH = ΔΛ.

前提 4. AΔ+AΛ = BΔ+BH.

結論. BΓ = AΛ.

(ゴールウィンドウは intros. を実行した状態)

```
Variable A B Δ Γ H Λ : Point.
Proposition Prop1_2:
def_Prop1_1 (A,B) Δ ->
L_eq (B,Γ) (B,H) ->
L_eq (Δ,H) (Δ,Λ) ->
LPL_eq ((A,Δ),(A,Λ)) ((B,Δ),(B,H)) ->
L_eq (B,Γ) (A,Λ).
```

```
1 subgoal
s1, s2, s3 : Line
p1, A, B, Δ, Γ, H, Λ : Point
H0 : def_Prop1_1 (A, B) Δ
H1 : ((B, Γ) [@] (B, H))
H2 : ((Δ, H) [@] (Δ, Λ))
H3 : ((A, Δ, (A, Λ))
     [@@] (B, Δ, (B, H)))
______________________________________
((B, Γ) [@] (A, Λ))
```

図 14: 仮定および結論を Coq で表現したもの

H0 は論理積を含む前提である。このような前提は intros. では分離されないため，以下のコマンドを使用する。

destruct H0.

図 15 のように，前提 H0 が 2 つの新たな前提 H0，H4 に分解された。次に，後の書き換えの為に前提 H0 の両辺を入れ替えた前提を生成する。

specialize (L_rev (A, B) (fst (A, B), Δ)).
intro.
specialize (H5 H0).

```
H0 : ((A, B) [@] (fst (A, B), Δ))
H4 : ((A, B) [@] (snd (A, B), Δ))
H1 : ((B, Γ) [@] (B, H))
H2 : ((Δ, H) [@] (Δ, Λ))
H3 : ((A, Δ, (A, Λ))
     [@@] (B, Δ, (B, H)))
______________________________________
((B, Γ) [@] (A, Λ))
```

図 15: 前提の両辺を入れ替えたものの生成

対称律にあたる定義関数 L_rev に前提 H0 の引数 “(A, B)”と “(fst (A, B), Δ))”を指定し，追加された前提 H5 を H0 で書き換える。

定義 “l1 = l2 → l2 = l1”，前提 “AB = AΔ”より，前提 H5 “AΔ = AB”が生成された。

次に，前提 H5 と H4 を Line の推移律に当てはめることを考える。

specialize (L_suii_1 (A, Δ) (A, B) (B, Δ)).
intro.
specialize (H6 H5).
specialize (H6 H4).

```
H0 : ((A, B) [@] (fst (A, B), Δ))
H4 : ((A, B) [@] (snd (A, B), Δ))
H1 : ((B, Γ) [@] (B, H))
H2 : ((Δ, H) [@] (Δ, Λ))
H3 : ((A, Δ, (A, Λ)) [@@] (B, Δ, (B, H)))
H5 : ((fst (A, B), Δ) [@] (A, B))
______________________________________(1/1)
((B, Γ) [@] (A, Λ))
```

図 16: 追加された前提を書き換える

定義関数 L_suii_1 に H5 と H4 の引数を指定，追加された前提 H6 をそれぞれを指定して書き換える。

定義 “l1 = l2 → l2 = l3 → l1 = l3”,
前提 “AΔ = AB”, “AB = BΔ”より，
新たな前提 H6 “AΔ = BΔ” が生成された。

次に，前提 H3 と H6 を LPL の定義関数に当てはめる。

```
H0 : ((A, B) [@] (fst (A, B), Δ))
H4 : ((A, B) [@] (snd (A, B), Δ))
H1 : ((B, Γ) [@] (B, H))
H2 : ((Δ, H) [@] (Δ, Λ))
H3 : ((A, Δ, (A, Λ)) [@@] (B, Δ, (B, H)))
H5 : ((fst (A, B), Δ) [@] (A, B))
H6 : ((A, Δ) [@] (B, Δ))
_____________________________________(1/1)
((B, Γ) [@] (A, Λ))
```

図 17: 新たな前提の生成

```
specialize (LPL_trans_2 (A, Δ) (A, Λ) (B, Δ) (B, H)).
intro.
specialize (H7 H3).
specialize (H7 H6).
```

定義関数 LPL_trans_2 に H3 の引数を指定，追加された前提 H7 を H3 と H6 で書き換える。

定義 “l1+l2 = l3+l4 → l1 = l3 → l2 = l4”,
前提 “AΔ+AΛ = BΔ+BH”, “AΔ = BΔ” より，新たな前提 H7 “AΛ = BH” が生成された。次に，後の書き換えの為，前提 H1 の両辺を入れ替える。

```
H0 : ((A, B) [@] (fst (A, B), Δ))
H4 : ((A, B) [@] (snd (A, B), Δ))
H1 : ((B, Γ) [@] (B, H))
H2 : ((Δ, H) [@] (Δ, Λ))
H3 : ((A, Δ, (A, Λ)) [@@] (B, Δ, (B, H)))
H5 : ((fst (A, B), Δ) [@] (A, B))
H6 : ((A, Δ) [@] (B, Δ))
H7 : ((A, Λ) [@] (B, H))
_____________________________________(1/1)
((B, Γ) [@] (A, Λ))
```

図 18: 新たな前提と両辺の入れ替え

次に，前提 H7 と H8 を Line の推移律 (L_suii_1) に当てはめる。

```
H0 : ((A, B) [@] (fst (A, B), Δ))
H4 : ((A, B) [@] (snd (A, B), Δ))
H1 : ((B, Γ) [@] (B, H))
H2 : ((Δ, H) [@] (Δ, Λ))
H3 : ((A, Δ, (A, Λ)) [@@] (B, Δ, (B, H)))
H5 : ((fst (A, B), Δ) [@] (A, B))
H6 : ((A, Δ) [@] (B, Δ))
H7 : ((A, Λ) [@] (B, H))
H8 : ((B, H) [@] (B, Γ))
_____________________________________(1/1)
((B, Γ) [@] (A, Λ))
```

図 19: 推移律に当てはめる

最後に，結論を前提 H9 に合わせた形に書き換える。

```
apply (L_rev (A, Λ) (B, Γ)).
```

```
H0 : ((A, B) [@] (fst (A, B), Δ))
H4 : ((A, B) [@] (snd (A, B), Δ))
H1 : ((B, Γ) [@] (B, H))
H2 : ((Δ, H) [@] (Δ, Λ))
H3 : ((A, Δ, (A, Λ)) [@@] (B, Δ, (B, H)))
H5 : ((fst (A, B), Δ) [@] (A, B))
H6 : ((A, Δ) [@] (B, Δ))
H7 : ((A, Λ) [@] (B, H))
H8 : ((B, H) [@] (B, Γ))
H9 : ((A, Λ) [@] (B, Γ))
______________________________________(1/1)
((B, Γ) [@] (A, Λ))
```

図 20: 結論を前提に合わせた形に書き換える

前提の中に結論と合致するものが現れたので trivial で副題証明を終える。全ての副題が証明されたので，Qed で証明を終了する。

最後に，証明の全文を示す。

```
Variable A B Δ Γ H Λ : Point.
Proposition Prop1_2:
def_Prop1_1 (A, B) Δ ->
L_eq (B, Γ) (B, H) ->
L_eq (Δ, H) (Δ, Λ) ->
LPL_eq ((A, Δ), (A, Λ)) ((B, Δ), (B, H)) ->
L_eq (B, Γ) (A, Λ).
Proof. intros.
destruct H0.
specialize (L_rev (A, B) (fst (A,  B), Δ)). intro.
specialize (H5 H0).
specialize (L_suii_1 (A, Δ) (A, B) (B, Δ)). intro.
specialize (H6 H5).
specialize (H6 H4).
specialize (LPL_trans_2 (A, Δ) (A, Λ) (B, Δ) (B, H)). intro.
specialize (H7 H3).
specialize (H7 H6).
specialize (L_rev (B, Γ) (B, H)). intro.
specialize (H8 H1).
specialize (L_suii_1 (A, Λ) (B, H) (B, Γ)). intro.
specialize (H9 H7).
specialize (H9 H8).
apply (L_rev (A, Λ) (B, Γ)).
trivial.
Qed.
```

```
H0 : ((A, B) [@] (fst (A, B), Δ))
H4 : ((A, B) [@] (snd (A, B), Δ))
H1 : ((B, Γ) [@] (B, H))
H2 : ((Δ, H) [@] (Δ, Λ))
H3 : ((A, Δ, (A, Λ)) [@@] (B, Δ, (B, H)))
H5 : ((fst (A, B), Δ) [@] (A, B))
H6 : ((A, Δ) [@] (B, Δ))
H7 : ((A, Λ) [@] (B, H))
H8 : ((B, H) [@] (B, Γ))
H9 : ((A, Λ) [@] (B, Γ))
______________________________________(1/1)
((A, Λ) [@] (B, Γ))
```

図 21: 証明の終了

4 Isabelle による証明

前節に於いて Coq で行った証明を，同じ定理証明支援ソフトウェアである Isabelle [3] を用いて実行する方法を示す。

図 22 は，Coq と同様に，点，線分，線分+線分の型宣言と，線分および線分+線分の定義関数の宣言と略記，そして原論の命題 1_2 の証明に使用する命題 1_1 の定義をそれぞれ Isabelle で行う様子である。

この時点で見られる差異としては，Isabelle の定義関数宣言に使われる locale には，Inductive にあるような制限が無く，また Coq では定義に一般性を与えるために行なっていた全称限指定の必要が無いこと等が挙げられる。

```
theory Isabelle_Euclidean imports Main begin

datatype Point = "char"
datatype Line = Se "Point" "Point"
datatype LPL = Sel "Line" "Line"

locale Line_def =
  fixes L_eq :: "Line ⇒ Line ⇒ bool" (infixl "[@]" 50)
    and LPL_eq :: "LPL ⇒ LPL ⇒ bool" (infixl "[@@]" 50)
  assumes L_ref [simp,intro] : "s1 [@] s1"
    and L_rev : "⟦s1 [@] s2⟧ ⟹ s2 [@] s1"
    and L_suii_1 : "⟦s1 [@] s2; s2 [@] s3⟧ ⟹ s1 [@] s3"

    and LPL_ref [simp,intro] : "sl [@@] sl"
    and LPL_rev : "⟦sl1 [@@] sl2⟧ ⟹ sl2 [@@] sl1"
    and LPL_suii_1 : "⟦sl1 [@@] sl2; sl2 [@@] sl3⟧
       ⟹ sl1 [@@] sl3"
    and LPL_trans_1  : "⟦s1 [@] s2; s3 [@] s4⟧
       ⟹ Sel s1 s3 [@@] Sel s2 s4"
    and LPL_trans_2  : "⟦Sel s1 s2 [@@] Sel s3 s4; s1 [@] s3⟧
       ⟹ s2 [@] s4"

locale L_Proposition1_1 = Line_def +
  fixes L_Prop1_1 :: "Line ⇒ Point ⇒ bool" ("[p1-1] _,_")
  assumes Prop1_1 : "⟦[p1-1] Se p1 p2 , pn⟧
    ⟹ Se p1 p2 [@] Se p1 pn ∧ Se p1 p2 [@] Se p2 pn"
```

図 22: 定義関数の宣言と略記を Isabelle で行う様子

3.2 節の例題を Isabelle で証明する様子は，図 23 のようになる。

fixes が Coq の Variable に，assumes が前提，shows が結論にそれぞれあたる。Coq では Line の推移律や対称律を使うために，都度前提の追加と書き換えを必要としたが，Isabelle では二つの定義関数を指定することのみで証明が完了している。

ただし，Coq のゴールウィンドウで表示されていたような証明過程を逐一示すような機能は Isabelle には存在しない。代わりに Isabelle では，副題の証明状況 (文法にエラーがある場合はその旨) が画面下枠に表示される。

```
lemma (in Line_def) Ltest1:
  fixes s1 s2 s3 :: Line
  assumes "s1 [@] s2" "s2 [@] s3"
  shows "s3 [@] s1"
proof -
  from assms show "s3 [@] s1"
    by (blast intro : L_suii_1 L_rev)
qed
```

図 23: 3.2 節の例題を Isabelle で証明する様子

図 24 は “proof - ”以下のコマンドを入力する前と後の, その画面を並べたものである.

```
Failed to finish proof:
goal (1 subgoal):
 1. s3 [@] s1
```

```
show s3 [@] s1
Successful attempt to solve goal by exported rule:
  s3 [@] s1
```

図 24: “proof - ”以下のコマンドを入力する前と後

3.4 節と同様に，命題 1_2 の証明を以下に示す。

命題 1_2 の仮定と結論を Isabelle で表現したものを図 25 に示す。

その他の前提は Coq のそれと同じである。

```
theorem (in L_Proposition1_1) Proposition1_2:
  fixes A B Γ Δ Λ H:: Point
    and AΔ AΛ BΔ BΓ AB BH :: Line
  assumes
    "AΔ = Se A Δ" "AΛ = Se A Λ" "BΔ = Se B Δ"
    "BΓ = Se B Γ" "AB = Se A B" "BH = Se B H"
    "[p1-1] AB,Δ" "BΓ [@] BH" "ΔH [@] ΔΛ"
    "Sel AΔ AΛ [@@] Sel BΔ BH"
  shows "BΓ [@] AΛ"
```

図 25: 仮定と結論を Isabelle で表現したもの

Coq では入力したコマンドを一行ずつ読ませることで処理が進行するが，Isabelle は入力次第実行する。途中にエラーがあった場合，Isabelle にはエラー箇所を放置してその先の文の実行を試みるという特徴がある。これを利用することで Isabelle では，結論から逆向きに証明を完成させることが可能である。

結論である “BΓ [@] AΛ”の証明には，“AΔ+AΛ = BΔ+BΓ”と “AΔ = BΔ”を導く必要がある。そこで, 図 26 のような入力を行う。

赤く表示されている部分がエラー箇所である。すなわち，追加された仮定 P1, P2 は未証明であるが，それらを用いれば結論 “BΓ [@] AΛ”を証明できると分かる。

```
proof -
  from assms have P1 : "AΔ [@] BΔ"
    by (blast intro : )
  from assms have P2 : "Sel AΔ AΛ [@@] Sel BΔ BΓ"
    by (blast intro : )
  from assms P1 P2 show "BΓ [@] AΛ"
    by (blast intro : LPL_trans_2 L_rev)
qed
```

図 26: 結論の導出

P1 の証明は命題 1_1 の定義に Line の推移律と対称律を適用すれば完了する。

```
from assms have P3 :
      ”AB [@] AΔ ∧ AB [@] BΔ”
      by (simp add : Prop1_1)
from assms P3 have P1 : ”AΔ [@] BΔ”
      by (blast intro : L_suii_1 L_rev)
```

```
proof -
  from assms have P3 : "AB [@] AΔ ∧ AB [@] BΔ"
    by (simp add : Prop1_1)
  from assms P3 have P1 : "AΔ [@] BΔ"
    by (blast intro : L_suii_1 L_rev)
  from assms have P2 : "Sel AΔ AΛ [@@] Sel BΔ BΓ"
    by (blast intro : )
  from assms P1 P2 show "BΓ [@] AΛ"
    by (blast intro : LPL_trans_2 L_rev)
qed
```

図 27: P1 の証明の完了

Coq では論理積を含んだ前提を分解する必要があったが，Isabelle ではそのまま利用可能である。命題 1_1 を表す新たな仮定 P3 (証明は命題 1_1 の定義を指定する形になる) と，Line に関する二つの定義関数を指定することで，P1 の証明が完了する。

P2 の証明は，“BΔ+BH = BΔ+BΓ”があれば，LPL の推移律で証明できる。

```
from assms have P4 :
      ”Sel BΔ BH [@@] Sel BΔ BΓ”
      by (blast intro : LPL_trans_1 L_rev)
```

P4 の証明は，前提に存在する “BΓ = BH”, ”AΔ+AΛ = BΔ+BH”より導ける。よって，あとは適切な定義関数を指定すればよい。

```
proof -
  from assms have P3 : "AB [@] AΔ ∧ AB [@] BΔ"
    by (simp add : Prop1_1)
  from assms P3 have P1 : "AΔ [@] BΔ"
    by (blast intro : L_suii_1 L_rev)
  from assms have P4 : "Sel BΔ BH [@@] Sel BΔ BΓ"
    by (blast intro : )
  from assms P4 have P2 : "Sel AΔ AΛ [@@] Sel BΔ BΓ"
    by (blast intro : LPL_suii_1)
  from assms P1 P2 show "BΓ [@] AΛ"
    by (blast intro : LPL_trans_2 L_rev)
qed
```

図 28: 適切な定義関数の指定

```
from assms P4 have P2 :
      ”Sel AΔ AΛ [@@] Sel BΔ BΓ”
      by (blast intro : LPL_suii_1)
```

エラー箇所が無くなったことで証明完了となる。

```
proof -
  from assms have P3 : "AB [@] AΔ ∧ AB [@] BΔ"
    by (simp add : Prop1_1)
  from assms P3 have P1 : "AΔ [@] BΔ"
    by (blast intro : L_suii_1 L_rev)
  from assms have P4 : "Sel BΔ BH [@@] Sel BΔ BΓ"
    by (blast intro : LPL_trans_1 L_rev)
  from assms P4 have P2 : "Sel AΔ AΛ [@@] Sel BΔ BΓ"
    by (blast intro : LPL_suii_1)
  from assms P1 P2 show "BΓ [@] AΛ"
    by (blast intro : LPL_trans_2 L_rev)
qed
```

図 29: Isabelle での証明完了

最後に，証明の全文を示す。Coq での証明に比べ，行数即ち工程数に大きな差があることが分かる。

```
theorem (in L_Proposition1_1) Proposition1_2:
  fixes A B Γ Δ Λ H:: Point
    and AΔ AΛ BΔ BΓ AB BH :: Line
  assumes
    "AΔ = Se A Δ" "AΛ = Se A Λ" "BΔ = Se B Δ"
    "BΓ = Se B Γ" "AB = Se A B" "BH = Se B H"
    "[p1-1] AB,Δ" "BΓ [@] BH" "ΔH [@] ΔΛ"
    "Sel AΔ AΛ [@@] Sel BΔ BH"
  shows "BΓ [@] AΛ"
proof -
  from assms have P3 : "AB [@] AΔ ∧ AB [@] BΔ"
    by (simp add : Prop1_1)
  from assms P3 have P1 : "AΔ [@] BΔ"
    by (blast intro : L_suii_1 L_rev)
  from assms have P4 : "Sel BΔ BH [@@] Sel BΔ BΓ"
    by (blast intro : LPL_trans_1 L_rev)
  from assms P4 have P2 : "Sel AΔ AΛ [@@] Sel BΔ BΓ"
    by (blast intro : LPL_suii_1)
  from assms P1 P2 show "BΓ [@] AΛ"
    by (blast intro : LPL_trans_2 L_rev)
qed
```

5 考察

以下に，Coq と Isabelle の長所と短所を示す。

- Coq 長所

- 証明の過程を細かく見ることができる。
- 入力したコマンドの実行が手動で行われるため，エラー発生時の問題の所在が分かり易い。
- (Isabelle の証明と比較して) 前提や結論の書き換えが 1 つずつ実行されるため，ループが起こり難い。

- Coq 短所

- 証明の道筋が長く，冗長になり易い。
- 型や定義の宣言など，やや文法上の制約が多い。

- Isabelle 長所

- 逆からの証明完成を試みることができる。

- 証明の簡略，簡潔化が容易。
- 宣言に関する制約が少ない。

- Isabelle 短所

- 証明過程の確認が難しい。
- (Coq と比較して) 証明中にループが発生することが多い。

定理証明支援ソフトウェアとしてどちらが優れていると安易に評価することは出来ないが，これらの点に留意して，用いるソフトを選択するべきである。

参考文献

[1] “Welcome! — The Coq Proof Assistant”, https://coq.inria.fr/

[2] “Coq - Wikipedia”, https://ja.wikipedia.org/wiki/Coq

[3] “Isabelle”, https://isabelle.in.tum.de

[4] Fitzpatrick R., 2007, “EUCLID’S ELEMENTS OF GEOMETRY”, Lulu

[5] 髙橋 真, Coq による定理証明入門, http://herb.h.kobe-u.ac.jp/snap/coq.pdf

[6] 今井 宣洋, 2018,「Coq で学ぶ証明プログラミング！ テストだけでなく『証明』で安全性を保証する」, https://employment.en-japan.com/engineerhub/entry/2018/08/10/110000