修 士 論 文 の 和 文 要 旨

研究科・専攻	大学院　　情報理工学研究科　情報・ネットワーク工学専攻　博士前期課程		
氏　　名	山田 伊織	学籍番号	1631156
論 文 題 目	定理証明支援系 Coq における 手続き的証明から宣言的証明への変換		

要　　旨

定理証明支援系 Coq における証明は、一般に手続き的証明と呼ばれる形式で記述される。これは対話的証明を前提としており、自然言語による証明記述と大きく異なるため、可読性が高いものではない。この問題を解決するために Coq 用宣言的証明言語 C-zar が開発された。宣言的証明は可読性が高く、また外部ツールを導入し易い。しかし、C-zar は手続き的証明に対して記述量が多い上に柔軟性が低く、Coq ユーザに受け入れられなかった。本研究では、Coq の手続き的証明から C-zar の証明を生成することで、両者間の橋渡しを行う。一般に手続き的証明から宣言的証明への変換手法としては、証明項や証明木のような中間表現を経由する方法が考えられ、既に定理証明支援系 Matita では証明項を経由する手続き的証明から宣言的証明への変換が存在する。しかし、中間表現は元の証明と比べて詳細かつ巨大になり、元の手続き的証明 1 ステップに対して数百ステップの宣言的証明が生成されてしまう場合もある。一方で、C-zar は手続き的証明で用いられるタクティックと呼ばれるコマンドを利用することができ、これによって手続き的証明の 1 ステップは、多くの場合 C-zar の数ステップと対応させることができる。本研究では、元の手続き的証明と証明項の両方を用いて変換を行うことで、元の証明に近い粒度の宣言的証明の生成を実現する。

平成 29 年度修士論文

定理証明支援系 Coq における
手続き的証明から宣言的証明への変換

電気通信大学
大学院情報理工学研究科
情報・ネットワーク工学専攻
コンピュータサイエンスプログラム

学籍番号　：1631156
氏名　：山田 伊織
主任指導教員　：中野 圭介 准教授
指導教員　：中山 泰一 准教授
提出日　：2018 年 3 月 13 日

要旨

定理証明支援系 Coq における証明は、一般に手続き的証明と呼ばれる形式で記述される。これは対話的証明を前提としており、自然言語による証明記述と大きく異なるため、可読性が高いものではない。この問題を解決するために Coq 用宣言的証明言語 C-zar が開発された。宣言的証明は可読性が高く、また外部ツールを導入し易い。しかし、C-zar は手続き的証明に対して記述量が多い上に柔軟性が低く、Coq ユーザに受け入れられなかった。本研究では、Coq の手続き的証明から C-zar の証明を生成することで、両者間の橋渡しを行う。一般に手続き的証明から宣言的証明への変換手法としては、証明項や証明木のような中間表現を経由する方法が考えられ、既に定理証明支援系 Matita では証明項を経由する手続き的証明から宣言的証明への変換が存在する。しかし、中間表現は元の証明と比べて詳細かつ巨大になり、元の手続き的証明 1 ステップに対して数百ステップの宣言的証明が生成されてしまう場合もある。一方で、C-zar は手続き的証明で用いられるタクティックと呼ばれるコマンドを利用することができ、これによって手続き的証明の 1 ステップは、多くの場合 C-zar の数ステップと対応させることができる。本研究では、元の手続き的証明と証明項の両方を用いて変換を行うことで、元の証明に近い粒度の宣言的証明の生成を実現する。

目次

1 はじめに

定理証明支援系は、ユーザが記述した形式証明の正当性を機械的に検証するプログラムである。ケプラーの定理のような複雑な証明を形式化し、証明の正当性を保証するために用いられる [1] 。また、新たな理論を提唱するにあたり、その形式化を定理証明支援系上で行う例も存在する [2] 。多くの定理証明支援系はプログラムを記述・検証し、実行可能な形で出力する機能を持つため、オペレーティングシステム [3] や言語処理系 [4] のような基幹ソフトウェアの検証にも利用されている。

INRIA によって開発された Coq [5] は、広く利用されている定理証明支援系のひとつであり、四色定理の証明 [6] や奇数位数定理の証明 [7] の形式化に使われたことで知られている。また、プログラムに関する利用例としては検証を行った C コンパイラ [8] や LLVM クローン [9] の作成などが挙げられる。Why3/Frama-C [10] や Spoofax [11] のように、証明すべき命題が書かれた Coq ソースファイルを出力することで、検証機能を Coq に委託するようなソフトウェアも存在する。

Coq では、タクティックと呼ばれる命令を順次与えることで証明を記述する。タクティックは「等式によって書き換える」「2 つの命題に分解する」といった命題に対する操作を表し、この操作を繰り返して既存の定理や公理に帰着させることで証明を行う。ユーザは、Coq の対話環境を利用して、命題が変化していく様子を確認しながら証明を記述する。一般に、タクティック列による証明スタイルを手続き的証明と呼ぶ。

例として、$\forall x \forall y, (x+1)+y = x+(y+1)$ に対応する手続き的証明をソースコード 1.1 に示す。以降、Coq で記述された証明のことを証明スクリプトと呼ぶ。ソースコード 1.1 は、x に対する帰納法を用いた証明を表す証明スクリプトであり、4 行目が $x = 0$ の場合、5 行目から 7 行目が $x = S\ x'$ ($x'+1$ を意味する) の場合の証明を表している。各行が一つのタクティックに対応しており、各タクティックの働きは次の通りである。ここで、数字はソースコード 1.1 の行番号を表す。

4. `intros`: 命題から「`forall x y:nat,`」を取り除き、「`x:nat`」「`y:nat`」を仮定に追加する。
   命題は `(S x) + y = x + (S y)` になる。
5. `induction x`: `x` について帰納法を開始する。
   命題は `(S 0) + y = x + (S y)` と `(S (S x')) + y = (S x') + (S y)` に分岐する。
6. `reflexivity`: （`x = 0` の場合について）両辺を簡約し、等しいことを確認する。
   命題 `(S 0) + y = x + (S y)` の証明が完了する。
7. `simpl`: （`x = S x'` の場合について）両辺を簡約する。

ソースコード 1.1　Coq における証明

```
1  Goal forall x y : nat,
2    S x + y = x + S y.
3  Proof.
4    intros.
5    induction x.
6    reflexivity.
7    simpl.
8    f_equal.
9    apply IHx.
10 Qed.
```

ソースコード 1.2　変更された証明

```
1  Goal forall x y : nat, S x + y = x + S y.
2  Proof.
3    intros.
4    induction x as [| _ kinou].
5    all: swap 1 2.
6    simpl.
7    f_equal.
8    apply kinou.
9    reflexivity.
10 Qed.
```

命題は S (S (x' + y)) = S (x' + (S y)) になる。

8. f_equal: 前行で得られた命題の両辺から後者関数 S を取り除く。
   命題は S (x' + y) = x' + (S y) になる。
9. apply IHx: 前行で得られた命題に帰納法の仮定 (IHx) を適用する。
   命題 S (x' + y) = x' + (S y) の証明が完了する。

ここで、証明の記述者が証明を分かり易くするために変更を加えたいとする。例えば、やや不自然ではあるが、x が S x' の場合の証明を x が 0 の場合よりも先に行い、帰納法の仮定の名前が kinou になるように変更したいとする。ソースコード 1.1 にそのような変更を加えると、ソースコード 1.2 のような証明になる。これは、5 行目で 2 つの場合の証明順を交換することを表し、実際に 6 行目から 8 行目と 10 行目でそれぞれの場合の証明を行っている。また、4 行目で帰納法の仮定名の変更を行っている。このような変更は各タクティックにどのような機能が備わっているかを把握していなければ行うことができない。

ソースコード 1.1 を見てわかるように、手続き的証明は記述量が少ないスタイルである。Coq では 1 ステップずつタクティックを実行し、Coq が提示する情報を見ながら証明を考える、対話的証明が一般的である。対話的証明では状態を巻き戻して試行錯誤する場合もあるため、記述量が少ないことは重要である。

一方、完成した証明スクリプトには証明中に Coq が提示する情報が残らないため、証明スクリプトのみからどのように証明を行っているのか読み取ることは難しい。特に、induction タクティックのように場合分けを行う場合などには、証明スクリプトのどの部分が 1 つ目の場合に対応するかといった証明の構造が不明瞭である。そのため、変数名の変更など命題や証明に手直しを加えたい場合にも、変更箇所が明確ではない。また、証明を書いた本人以外が証明スクリプトの内容を理解するためには、証明スクリプトを Coq で実行して何が行われているか確認するか、タクティックの動作を覚えその様子を想像する必要がある。このような事態を避けるため、Coq で書かれた証明とは別に、その内容を説明する文書が作成される場合もある。

手続き的証明にも構造を分かり易くするような手法は存在する。例えば、ソースコード 1.3 は

ソースコード 1.3　Coq における証明

```
1  Goal forall x y : nat, S x + y = x + S y.
2  Proof.
3    intros x y.
4    induction x.
5    - reflexivity. (* x = 0 *)
6    - simpl. f_equal. apply H. (* x = S x' *)
7  Qed.
```

ソースコード 1.4　C-zar による証明

```
1  Goal forall x y : nat, S x + y = x + S y.
2  proof.
3    let x:nat, y:nat.
4    per induction on x.
5    suppose it is 0.
6      thus (0 + S y = 0 + S y) by Nat.add using reflexivity.
7    suppose it is (S x') and IHx:(S x' + y = x' + S y).
8      have (S (S (x' + y)) = S (x' + S y)) by IHx using f_equal.
9      hence (S (S x') + y = S x' + S y) using simpl.
10   end induction.
11 end proof.
12 Qed.
```

ソースコード 1.1 に対して構造が明確になるように手を加えたものである。3 行目では x と y を仮定に移動することを、5 行目と 6 行目では各タクティックが induction タクティックで生まれた 2 つの場合のどちらに対応しているかをそれぞれ示している。このような証明であれば、証明がどのように行われているのかがある程度理解でき、変更も多少容易になるものの、それでもソースコード 1.2 にする方法は自明ではない。また、6 行目で何をしているか理解するためには、やはりタクティックの動作について知っているか、実行する必要がある。

こういった問題を解決するために、Coq 用の証明言語 C-zar [12] が開発された。C-zar は Coq8.1 から標準プラグインとして組み込まれた。Coq8.3 以降は『数学的証明言語（Mathematical Proof Language)』と呼称を改めたが、本論文では分かり易さのため『C-zar』を用いる。C-zar は、命題を中心に記述する宣言的証明と呼ばれる証明スタイルを採用している。C-zar は読み易さや保守性に重きを置いており、構造の把握や証明順序の変更などが行い易い。例えば、ソースコード 1.4 はソースコード 1.1 と同等の証明である。

ソースコード 1.4 の証明における各行の働きを次に示す。なお、対応関係を明示するためにソースコード 1.4 では using を使ってタクティックを明示しているが、宣言的証明にとって本質的な記述ではなく、実際には明示する必要も無い。

ソースコード 1.5　変更された C-zar による証明

```
1  Goal forall x y : nat, S x + y = x + S y.
2  proof.
3    let x:nat, y:nat.
4    per induction on x.
5    suppose it is (S x') and kinou:(S x' + y = x' + S y).
6      have (S (S (x' + y)) = S (x' + S y)) by kinou using f_equal.
7      hence (S (S x') + y = S x' + S y) using simpl.
8    suppose it is 0.
9      thus (0 + S y = 0 + S y) by Nat.add using reflexivity.
10   end induction.
11 end proof.
12 Qed.
```

2. proof: C-zar による証明を開始する。
3. let x:nat, y:nat: 命題から「forall x y,」を取り除き、「x:nat」「y:nat」を仮定に追加する。
4. per induction on x: x について帰納法を開始する。
5. suppose it is 0: x = 0 の場合についての証明を開始する。
6. thus 〜 by Nat.add using reflexivity: 両辺が等しいと見なせるため証明を完了する。
7. suppose it is (S x'): x = S x' の場合について証明を開始する。
8. have 〜 by IHx using f_equal: 帰納法の仮定の両辺に S を適用する。
9. hence 〜 using simpl: 証明したい命題を簡約すると前行の命題になるため証明を完了する。
10. end induction: 帰納法を終了する。
11. end proof: C-zar による証明を終了する。

C-zar では原則として、新しく出現する仮定には必ずその名前を指定する、そのステップで示される命題を明示するなど、ステップ毎の動作を陽に示すように設計されている。このように、命題を中心に記述するのが宣言的証明の特徴である。C-zar では証明の構造に変更を加え易く、例えばソースコード 1.4 にソースコード 1.2 と同様の変更を加えると、ソースコード 1.5 のようになる。対応する行を入れ替え、IHx の出現箇所を kinou に書き換えるだけであるため、変更箇所が明確であり、機械的な書き換えも行い易い。

現状では定理証明支援系自体が一般的に普及しているとは言い難く、従って形式証明を扱うツールも豊富ではないものの、これが増えるに伴って宣言的証明はその重要度を増すと考えられる。Whiteside ら [13] [14] は、形式証明記述とプログラム開発の類似性から、形式証明にもプログラミングにおける IDE のような高機能の開発環境が必要であると主張し、形式証明の機械

的なリファクタリング手法を提案している。ソースコード 1.5 で見たように、宣言的証明では機械的に変更を加えることが容易であり、リファクタリング手法と相性が良い。

しかしながら、C-zar が Coq ユーザに使われることはほとんどなく、Coq8.7 において C-zar は保守者がいないことを理由に標準プラグインから削除されている。利用されなかった背景として、手続き的証明に熟練した既存ユーザには主なメリットである読み易さを感じにくく、記述が冗長である・対話的証明が行い難いといった宣言的証明の欠点が強調されてしまったことが一因ではないかと考えられる。特に「命題を可能な限り単純化する」といった自動証明タクティックは Coq でよく用いられるが、タクティックの実行結果を明示する必要がある宣言的証明とは相性が悪い。

特に、C-zar に限らず一般的に、宣言的証明は手続き的証明に比べ記述コストが高いことが大きな欠点として挙げられる。そのため、テンプレート化による記述の省略 [15] [16] や自動証明による宣言的証明生成 [17] といった手法が開発されており、C-zar にもこれらのように簡単に記述もしくは生成できる機能が必要であると考えられる。

本研究では、Coq における手続き的証明を C-zar による宣言的証明に変換することで、記述コストを抑えながら宣言的証明を生成する手法を提案する。具体的には、Coq のプラグインとして、指定した手続き的証明を宣言的証明に変換する機構を開発した。これによって、対話的証明によって宣言的証明を生成することが可能になる。

手続き的証明から宣言的証明への変換の既存研究として、定理証明支援系 Matita [18] では、証明項から変換可能な宣言的証明言語が Coen [19] によって実装されている。証明項は、Coq や Matita における証明の内部表現であり、証明を行うことで生成される。手続き的証明による証明項に対してこの変換を行うことで、宣言的証明が生成できる。

一方でこの変換は、宣言的証明を容易に記述することを主な目的としていない。Coen は、手続き的証明や証明項、もしくは証明木といった様々な証明の表現方法について、できる限り相互に変換可能であるべきだと主張しており、宣言的証明も表現方法の一つに過ぎない。加えて、宣言的証明から生成した証明項に対してこの変換を用いることで、宣言的証明の詳細化を行うことも、その目的としている。このような動機のため、この変換は証明の構造を保つことを目標としており、宣言的証明言語もそれを満たすように設計されている。具体的には、証明項から宣言的証明への変換と宣言的証明の実行が、一部の情報の明示化を除いて逆変換となるように作られている。

自動証明タクティックは、非常に巨大な証明項を生成し得る。Coen の変換では、逆変換が可能なように、証明項の 1 要素と宣言的証明の 1 要素が概ね対応するようになっている。このため、生成される宣言的証明も詳細かつ膨大なものとなり、証明全体の流れは把握し難い。例えば、場合分けすればそれぞれの場合では自明な命題について、手続き的証明では、1 つのタクティックによって証明することも、全ての場合について一つずつ証明することも可能である。一方、Coen の宣言的証明言語では、常に全ての場合について列挙する。手続き的証明から宣言的証明への変換を考えたとき、全てが詳細化されてしまうと、証明で重要な部分に注目できなく

なってしまう。

本研究では、証明項だけでなく、元の証明のタクティックも利用することで、変換前後の証明の粒度を近づけ、より可読性の高い証明を生成する。これは Coen の変換に似た証明項のみからの変換も含んでおり、ユーザは変換結果を元の手続き的証明に近い粒度の証明にするか詳細化された証明にするかを選ぶことができる。

本研究では特に、既存の証明に変更を加えたい場合や証明を読み易く書き換えたい場合に、その前段階として変換を行うことを想定している。宣言的証明への変換を行うことで、変更すべき箇所を明確にし、かつエディタでの一括置換などによる機械的な修正を容易にする。

本論文の構成は次のようになっている。2 章では、Coq において (手続き的な) 証明をどのように行うか、またそれが Coq 内部でどのように扱われているかを紹介する。3 章では、C-zar の構文とその特徴について説明する。4 章では、具体的な変換手法を 3 段階に分けて紹介する。5 章では実装した変換について、その機能と標準ライブラリに適用した結果を確認する。6 章で関連研究について紹介し、7 章では今後の課題について述べる。

2 Coq による証明

Coq [5] は INRIA によって 1984 年に開発された定理証明支援系である。その特徴として、カリー・ハワード同型対応 [20] を基礎とした証明、タクティック記述言語 $\mathcal{L}_{tac}$ による証明の自動化などが挙げられる。前章で述べたように、数学およびソフトウェア開発の両方で利用されている。

本章では、Coq において証明がどのような表現で扱われるかを説明し、その後 Coq でどのように証明を行うか述べる。

2.1 証明項と証明状態

Coq では、カリー・ハワード同型対応 [20] に基づき、プログラムによって命題や証明を記述する。例えば、Coq における記述「`forall P, P → P`」は「任意の命題 `P` について、`P` ならば `P`」を意味するが、同時に型引数 `P` と型 `P` を持つ項を引数にとり型 `P` の項を返す関数の型としての意味も持つ。Coq に置いてこの命題を証明することは、この型を持つ項を与えることである。Coq ではプログラムを記述することもあるため、証明を表す項を特に証明項と呼ぶ。

命題や証明の記述には、Coq 独自の関数型言語 Gallina を用いる。Gallina の特徴として、型も項として扱える点が挙げられる。例えば、`0` は自然数を表す型 `nat` の項であり、`nat` は集合を表す型 `Set` の項である。Gallina の項の構文を図 2.1 に示す。ここで x は型名やコンストラクタ名を含む変数名を表し、c はコンストラクタ名を表すものとする。

なお、以降で証明項を書くとき、その一部または全体に対して、型注釈を入れる場合がある。例えば、`fun (x:nat) ⇒ x + 1` と `fun x ⇒ x + 1 : nat → nat` はどちらも項 `fun x ⇒ x + 1` と同じ意味を持つ。この注釈はそれぞれ、変数 `x` が型 `nat` を持つこと、項全体の型が `nat → nat` であることを表す。

また、`fun x ⇒ (fun y ⇒ ...⇒ t)` は以降 `fun x y ...⇒ t` のように略記する。同様に、`forall x, forall y, ..., t` は `forall x y ..., t` と略記する。

依存関数型は関数型の一般化で、例えば `forall A, A → A` という型を持つ項は、型 `A` と型 `A` を持つ項を引数として受け取り、型 `A` を持つ項を返す 2 引数関数である。`fun A (x:A) ⇒ x` がこの型を持つ。

Coq では証明項を与えることで証明を行うが、一度に証明項全体を与える必要はなく、段階的に証明項を構築することができる。構築途中の証明項は、未構築の部分を表す実在変数 (existential variable)$?x$ を含む形で表される。以降、実在変数を含む可能性がある証明項を部分証明項と呼ぶ。

構築途中の証明の状態は部分証明項によって表すことができるが、通常 Coq ユーザに証明の

$t := x$ (変数)

| $t_0\ t_1\ \ldots\ t_n$ (関数適用)

| `fun` $x \Rightarrow t$ (無名関数)

| $t \to t$ (関数型)

| `forall` x , t (依存関数型)

| `let` $x := t_0$ `in` t_1 (局所変数定義)

| `match` t `with` (パターンマッチ)

  | $c_1\ t_{11}\ \ldots\ t_{1j} \Rightarrow t_1$

  ...

  | $c_i\ t_{i1}\ \ldots\ t_{ik} \Rightarrow t_i$ `end`

| `fix` $x_0 := t_0$ `with` ... `with` $x_i := t_i$ `for` t_j (無名再帰関数)

| `cofix` $x_0 := t_0$ `with` ... `with` $x_i := t_i$ `for` t_j (無名余再帰関数)

図 2.1　Gallina の構文定義

状態として部分証明項が示されることはない。Coq の対話環境である coqtop による証明の様子を図 2.2 に、標準の開発環境である CoqIDE による証明の様子を図 2.3 に示す。図 2.2 では二重線を挟んだ 2 行、図 2.3 では画面右側がその時点での証明の状態を表している。どちらも、「`y : nat`」という利用できる仮定の表示と、「`1 + y = 0 + S y`」という示すべき命題の表示で構成されている。この仮定と命題の組をゴールと呼ぶ。

例では 1 つのゴールに対して仮定は 1 つだけ存在しているが、一般に仮定は存在しないもしくは複数存在する場合がある。例えば、図 2.2 の中ほどにある「`x, y : nat`」という記述は、利用できる仮定が `x:nat` と `y:nat` の 2 つであることを示している。一方で、命題はゴールに対して常に 1 つだけ存在する。命題を複数に分解した場合には、分解によって生まれた命題それぞれに対してゴールが作られる。実際、図 2.2 ではゴールが 2 つ存在しており、`subgoal` から始まる 2 行が、今注目しているゴールの他に、もう 1 つゴールが存在することを示している (仮定を省略しているため、ゴールそのものではない)。一般に、証明の状態は 0 個以上のゴールによって表される。ゴールが 0 個の場合とは、示すべき命題が存在しないということであるため、証明が完了したことを表す。

ゴールは部分証明項における実在変数の一つに対応しており、ゴールの仮定が実在変数の位置における環境を、ゴールの命題が実在変数の型を示している。部分証明項には実在変数が複数含まれる可能性があり、ゴールも実在変数と同数存在するため、構築途中の証明は 1 つ以上のゴールの組に対応している。ゴールを任意の順番でならべたものを、ゴールリストと呼ぶ。Coq ではゴールリストの先頭を「現在注目しているゴール」として扱い、他のゴールより多くの情報が表示される他、ゴールに対する操作の多くは注目しているゴールに対してのみ実行される。Coq

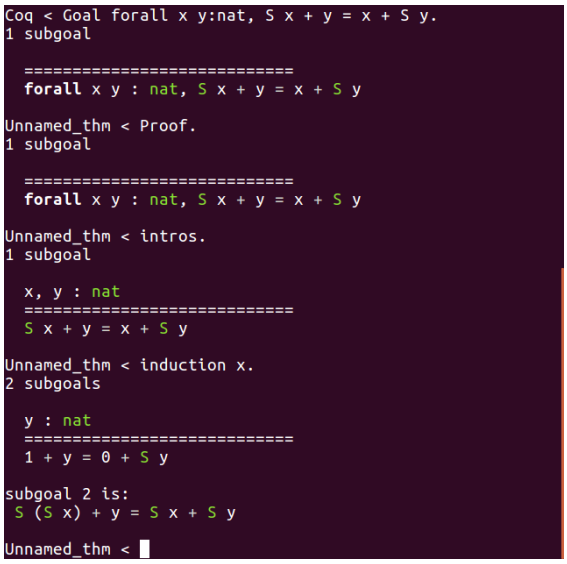

図 2.2　coqtop による証明実行例

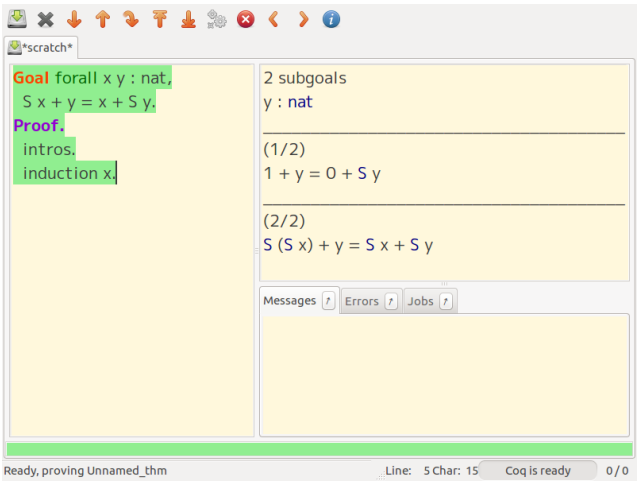

図 2.3　CoqIDE による証明実行例

における証明状態は、部分証明項とゴールリストの組であると考えると扱い易い。以下では仮定が H で命題が P であるようなゴールを $H \vdash P$ で示すことにする。

ここでは、ソースコード 1.1 で `induction x` を実行した時点での証明状態を例として示す。部分証明項は次のようになる。

```
(fun (x:nat) (y:nat) ⇒
  nat_ind (fun x':nat ⇒ S x' + y = x' + S y) ?g1
    (fun (x':nat) (IHx:S x' + y = x' + S y) ⇒ ?g2) x)
```

ここで、`nat_ind` は型 `nat` を持つ項に対する帰納法を行うような関数であり、引数の 1 つ目が示したい命題、2 つ目が `0` の場合の証明、3 つ目が `S x'` の場合の証明、4 つ目が帰納法の対象となる変数を表す。ここでは `nat_ind ... x` は `S x + y = x + S y` という型を持つ。無名関数は依存関数型 (返り値の型で引数を使わない場合には関数型) に対応するため、この部分証明項全体は `forall x y, S x + y = x + S y` という型になる。

また、実在変数?g1、?g2 に対応するゴール g1、g2 はそれぞれ次のようになる。ここで Γ は証明開始時の環境であるとする。?g2 は?g1 とは別の無名関数の中にあるため、?g1 よりも多くの仮定を持つ。

```
g1 = Γ, x:nat, y:nat ⊢ (S 0) + y = 0 + (S y)
g2 = Γ, x:nat, y:nat, x':nat, IHx:(S x' + y = x' + S y)
       ⊢ (S (S x')) + y = (S x') + (S y)
```

タクティックによって生まれた新たなゴールを、タクティック実行前のゴールまたはタクティックに対するサブゴールと呼ぶ。ゴール `Γ, x:nat, y:nat ⊢ (S x) + y = x + (S y)` に対して `induction x` を実行したときに g1 および g2 を得るため、この 2 つのゴールは `Γ, x:nat, y:nat ⊢ (S x) + y = x + (S y)` のサブゴール、または `induction x` のサブゴー

ソースコード 2.1　Coq での証明の例

```
1 Goal forall x y : nat, S x + y = x + S y.
2 Proof.
3   intros.
4   induction x.
5   reflexivity.
6   simpl.
7   f_equal.
8   apply IHx.
9 Qed.
```

ルなどと呼ばれる。

2.2　Coq による手続き的証明

Coq では、ユーザはタクティックの実行によって証明を進める。タクティックは、証明状態から証明状態への (部分) 関数である。タクティックを繰り返し適用し、ゴールが無くなった時、言い換えれば部分証明項が実在変数を含まない形になったとき、証明が完了する。

なお、Coq のタクティックはゴールリストに対し任意の変形を加えることができるが、本研究ではゴールリストの先頭にあるゴールにのみ実行され、サブゴールは任意の順番でゴールリストの先頭に追加されるものとする。また、タクティックによってはグローバル環境に定理を追加するものなども存在するが、本研究では対象外とする。

Coq による簡単な証明の例として、ソースコード 1.1 をソースコード 2.1 に再掲する。証明は、Coq への命令を表すコマンドと証明ステップを表すタクティックから成る。

Goal コマンドは証明したい命題を宣言するコマンドである。また、証明した定理に名前を付けたい場合、Goal コマンドの代わりに Lemma コマンドを用いる。Lemma コマンドでは、定理が引数をとることも可能である。例えば、Lemma lemma1 (x:nat) : x = x. を実行したとき、x:nat が仮定に追加された状態で命題 x = x の証明を開始し、証明終了後に lemma1 という名前で他の命題の証明に使用できるようにする。Proof コマンドは証明の始まりを示すが、実際には何も機能を持たない。省略しても問題無いが、Goal コマンドの命題が複数行に渡る場合などは証明の開始位置が分かり易くなるため有用である。証明の完了は Qed コマンドで示す。

証明で用いられる主なタクティックを以下に示す。

intros

ゴールの命題が $X \to Y$ もしくは forall x:X, Y の形であるならば、命題を Y に変え、仮定に x:X を追加する。ゴールの命題が $X \to Y$ のとき、名前は自動的に生成される。

`apply` H

仮定 H をゴールの命題に適用する。H が $X \to Y$ かつゴールの命題が Y のとき、ゴールの命題を X に変える。また、H がゴールと同じ命題を表すとき、そのゴールの証明を完了する。

`simpl`

ゴールの命題を簡約する。例えばゴールが`⊢ 1 + 1 = 2`のとき`simpl`を実行すると、サブゴール`⊢ 2 = 2`が生成される。2 つの命題は等価であるため、部分証明項は変化しないが、命題の形を変えることで、以降に実行するタクティックの挙動に影響を与える。

`rewrite` H

仮定の等式 H を使ってゴールの命題を書き換える。例えばゴールが`⊢ x = y`であり、`H`が`y = z`という命題を示すとき`rewrite H`を実行すると、サブゴール`⊢ x = z`が生成される。

`f_equal`

ゴールの命題が $f\,x = f\,y$ のとき、両辺から f を取り除き、$x = y$ にする。

`reflexivity`

ゴールの命題が $x = x$ のとき、または両辺を簡約すれば同じ形になるとき、そのゴールの証明を完了する。

`destruct` x

項 x について場合分けを行う。x の型のコンストラクタが n 個のとき、x をそれぞれの場合で置き換えた n 個のサブゴールを生成する。

`induction` x

項 x について構造的帰納法を行う。基本的には`destruct`タクティックと同様だが、ゴールの仮定に帰納法の仮定を追加する。

`trivial`

簡単なゴールについて自動的に証明を完了する。証明が完了できなかった場合にはゴールは変化しない。より強力な自動証明タクティックとして`auto`や`firstorder`なども存在する。

Coq では、タクティック記述言語 $\mathcal{L}_{tac}$ [21] を用いることで、既存のタクティックから新たなタクティックを作成できる。例えば、$\mathcal{L}_{tac}$ においてセミコロンは 2 つのタクティックを繋げる演算子 (タクティカル) である。`destruct x; auto`は、`x`について場合分けを行い、そのサブゴールそれぞれについて`auto`を行う 1 つのタクティックになる。Coq は $\mathcal{L}_{tac}$ で作ったタクティックに名前を付ける機能も備えているため、ユーザは自分で新しいタクティックを定義することができる。場合に応じて自動証明タクティックを作成することで、簡単に証明が行えるようになる。

複数のゴールが存在する時に証明の構造を明示する機能として、バレットと呼ばれる構文が

存在する。これはタクティックではなく一つのコマンドに相当する。ソースコード 1.3 の 5・6 行目行頭にある-記号がバレットである。バレットを実行すると、証明状態は変化しないが、バレットを実行した時点でのゴールリストの 2 番目以降のゴールに対する操作が行えなくなる。2 番目以降のサブゴールを証明するにはもう一度バレットを実行する必要があり、また 1 番目のゴールの証明が (そのサブゴールも含めて) 完了するまでバレットを実行することはできない。例えば、場合分けを行うタクティックを実行した直後にバレットを用いることで、証明スクリプトのどの範囲が一つの場合に対応しているかを明示することができる。ただし、場合分けを行う時点で既に複数のゴールが存在する場合には、それらのゴールも場合分けのサブゴールと同等に扱われてしまうなど、多少注意しないと直観に反する証明スクリプトになってしまう。

手続き的証明では、タクティックが「どのゴールに対して」「どのような変形を行ったか」を知るには、各タクティックの働きについて知っている必要がある。しかし、Coq ではタクティックはユーザ定義可能なものであり、Coq 標準のものだけに限ったとしてもその全てを網羅することは困難である。証明記述中は対話環境が各時点におけるゴールを表示し、それによってユーザはタクティックを実行した結果を把握できるため大きな障害にはならないが、補助なしで証明を読むためには多少 Coq の経験があっても難しい。

3　宣言的証明言語 C-zar

C-zar [12] は Corbineau によってプラグインとして開発された Coq の宣言的証明言語であり、後に Coq 本体に採用された。Coq での宣言的証明は既に MMode [22] で試みられていたが、C-zar はより高機能かつ自然な宣言的証明を実現した。C-zar は以下の 4 つを主な特徴としている。

- 可読性: 各ステップの働きがコマンドで示されていること。
- 自然言語らしさ: 自然言語での証明に似た構造を持つこと。
- 保守性: Coq のバージョンで振るまいが変わった時、その影響範囲が限定的であること。
- 独立性: Coq で実行しなくても十分理解できるだけの情報が明示されること。

本章では、まず宣言的証明一般について述べた後、C-zar の構文を紹介し、その特徴を挙げる。

3.1　宣言的証明

宣言的証明は命題を中心に記述する証明スタイルを指し、証明言語 Mizar [23] における証明をその源流に持つ。以降の宣言的証明言語は Mizar を基礎としており、また命題中心という特徴から、どの宣言的証明言語も基本的な構文が類似している。

宣言的証明では、基本的に各ステップで、そのステップにおけるゴールの命題を明示的に記述する。また、`induction` のように仮定が増えるステップでは、その命題と名前を記述する。この特徴のため、宣言的証明では各ステップにおけるゴールが明確である。

ただし、証明したい命題を示すだけでは、それが本当に証明可能であるか定理証明支援系が判定できるとは限らない。そのため、多くの宣言的証明言語は命題に加えて、その命題が成り立つ根拠も記述する必要がある。これを justification と呼ぶ。

手続き的証明は、ゴールをサブゴールに分解していく後向き証明 (backward proof) が一般的だが、C-zar を含めた多くの宣言的証明言語は既知の命題からボトムアップに最終的なゴールへ近付いていく前向き証明 (forward proof) を基本としている。言い換えれば、C-zar では殆どの場合、ゴールの変形ではなく、命題を仮定として追加していくことで証明を行う。よって、手続き的証明と宣言的証明の各ステップを対応付けるとき、その順番は凡そ逆向きになる。

ただし、前向き証明を主とする宣言的証明言語であっても、部分的には後向き証明を行う場合がある。例えば場合分けを行う場合、場合分けを行うことを宣言し、それぞれの場合について証明することが一般的である。これは後向き証明であり、タクティックによる手続き的証明と大差ない。同様に、手続き的証明でも部分的には前向き証明を取り入れている。

代表的な宣言的証明言語として、定理証明支援系 Isabelle [24] の宣言的証明言語 Isar [25] が

挙げられる。Isabelle は元々は手続き的証明しか備えていなかったが、現在では Isar を利用する証明が主流となっている。

3.2 C-zar の構文

C-zar の構文のうち、本研究で使用するものについて説明する。ここではピリオドで終わるひとつの命令を文と呼び、例えば `proof.` を「`proof` 文」と呼ぶことにする。

本研究では、数学で広く用いられる前向き証明を積極的に利用し、場合分けなど後向き証明の方が分かり易いと思われる場合のみ部分的に後向き証明を用いることで、より分かり易い証明の生成を目指す。そのため、紹介する構文は前向き証明が主である。

なお、C-zar では、証明ステップが正しく完了しなかった場合でも警告を出すのみで先に進められる場合がある。これは、不完全な証明ステップがあった場合にも、その先の証明について検証を可能にするための措置である。もちろん、そのまま証明を完了することはできず、`Qed` コマンドを実行した時点でエラーになる。

`proof.` 〜 `end proof.`

C-zar を使って証明を記述する範囲を示す。この内部でのみ C-zar を記述できる。

ただし、`proof` 文を実行した時点でゴールリストの先頭にあたるゴールのみを証明し、`end proof` 文を実行した時点でそのゴールの証明が完了するものとする。

`thus` P `by` ls `using` tac`.`

`hence` P `by` ls `using` tac`.`

どちらもゴールを証明する。P はこの文が証明する命題、つまりゴールの命題もしくはそれと等価な命題である。

by 以降は justification であり、タクティック tac によって、定理名リスト ls に含まれる定理・仮定のみを用いて証明できることを示す。なお、定理名リストには定理を表す名前を書くのが一般的ではあるが、任意の Gallina の項を書くことが可能である。`hence` 文の場合、直前の文で示した命題も ls に含まれるものとして扱う。

tac を実行した結果、P の証明が完了せずサブゴールが残ってしまった場合、`firstorder` タクティックを拡張した自動証明タクティックにより各サブゴールの証明を試み、ひとつでも証明が完了しなかった場合には警告を出す。

ls および tac はそれぞれ省略することもできる。ls が省略された場合は定理・仮定を使わずに (`hence` 文の場合は直前に示した命題のみを使って) 証明を行い、tac が省略された場合は自動証明タクティックのみで証明を行う。ソースコード 1.4 程度の簡単な証明であれば、tac を明示する必要は無い。

逆に、使用する定理を制限しない場合には ls として*を指定する。また、P でゴールの命題を示す特別な名前として `thesis` が用意されている。

have H : P by ls using tac.

then H : P by ls using tac.

thus 文/hence 文と同様に tac を使って、命題 P を証明する。ただし、P にはゴールではなく任意の命題を記述する。その後、P の証明に H という名前を付けて仮定に追加する。then 文は hence 文同様、直前に示した命題も ls に含まれるものとして扱う。H が省略された場合は証明した命題は次のステップでのみ (hence 文などによって) 使用できる。

let H : P.

ゴールの命題にある前提に H という名前を付けて仮定に移動する。P は H の型 (命題) を表す。手続き的証明におけるタクティック intro H と同様の意味を持つが、名前や型の省略は許されていない。

define x as t.

項 t に x という名前を付けて仮定に追加する。

claim H : P. 〜 end claim.

内部で命題 P を証明し、H という名前を付けて仮定に追加する。囲まれた部分の証明を部分証明と呼び、claim 文から end claim 文までの証明を claim 句と呼ぶ。

per cases on x. 〜 end cases.

per induction on x. 〜 end induction.

suppose it is (c t_1 ...t_n) and H_1:P_1 and ...and H_m:P_m.

項 x について場合分けを行う。cases 文から end cases 文の間でだけ suppose 文が利用できる。suppose 文はゴールを書き換え、x のコンストラクタが c である場合について証明を始める。cases 文の代わりに induction 文を使った場合には、suppose 文から始まる証明で帰納法の仮定 H_i:P_i を使用できる。

3.3 手続き的証明との比較

Harrison [26] は宣言的証明と手続き的証明の 2 つの証明スタイルを比較して、次のように述べている。

- 初心者には宣言的証明が親しみ易いが、手続き的証明はより柔軟な記述が可能である。
- 可読性は宣言的証明が勝り、外部ツールとの連携も行い易い。
- 実行 (証明検証) 効率は手続き的証明が勝る。
- 対話的な証明には手続き的証明が向いているが、大規模な証明を部分ごとに分割して行う場合には宣言的証明が向いている。

本研究では、宣言的証明が多フォーマットでの表示や外部ツールとの連携に優れている点に注目する。前者の例として、Coq に標準で含まれる coqdoc コマンドによる文書が挙げられる。coqdoc は Coq スクリプトから HTML や LaTeX ファイルを生成するコマンドであり、文芸的プログラミング [27] や論文執筆をその用途のひとつとしている。実際に coqdoc を用いて書かれた文書としては、Coq のライブラリドキュメントの他、書籍『Certified Programming with Dependent Types』 [28] や『Software Foundation』 [29] などが挙げられる。Coq による証明自体を文書として読ませたい場合、タクティックの働きを知らずとも大まかな証明の流れが分かる C-zar は適切であると考えられる。

後者の例としては、1 章で挙げたリファクタリングツールの他、Isabelle 用 IDE である jEdit が挙げられる。jEdit では、1 つの証明の複数箇所の検証を並行して進めることができる。これは、各ステップにおいて証明の成功失敗に関わらず、その結果が (明示されているため) 明確であるという宣言的証明の特長によって成り立っている。手続き的証明では各タクティックの実行結果は実行しないと分からず、前のステップから順番に実行することでしか検証できないため、一度に 1 箇所のエラーしか検知できない。対話的証明においてはこれは大きな問題にならないが、一度完成した証明を修正する場合には手間がかかる。C-zar では、エラーの代わりに警告を出すという手法で、複数エラー検知を可能にしている。

現状では証明の開発環境はプログラミングのそれと比較すると大きく劣っているが、周辺環境が充実すれば、C-zar は手続き的証明に対して大きな優位性を持ちうる。

一方、C-zar の欠点は大きく 2 つ挙げられる。まず、宣言的証明一般に見られる問題として、C-zar は手続き的証明に比べ明らかに記述量が多い。例えば、ソースコード 1.1 とソースコード 1.4 を行数で比較すると、C-zar は手続き的証明の約 1.5 倍である。更に C-zar は 1 行が長いため、文字数で比較すると約 3 倍になる。なお、ここで証明している命題は比較的短いので、記述量の差も少ないと考えられる。特に、関数の等価性などプログラムの性質に関する証明では、命題に含まれる関数定義の展開を行う場面がある。命題に関数定義がほぼそのまま現れることもあり、記述量は格段に増えることが予想される。これは単純に記述の手間が増えるだけでなく、1 ステップが数行に及び、ステップ間の差分や証明全体が見通しが分かり難くなるという問題にも繋がる。CoqIDE などで証明を実行すれば、命題が定位置に表示され、操作によってステップごとに表示が切り替わるため、変更箇所が明確になるが、同じことは手続き的証明でも可能である。証明する命題や辿る道筋によっては、宣言的証明にしたからといって必ずしも読み易くなるわけではない。

次に、C-zar では各ステップの結果を想定して書く必要がある。これは単純な場合であればそれほど問題にならないが、より高機能なタクティックを用いる場合に障害になりうる。手続き的証明であれば、証明冒頭でゴールを簡単にするような自動証明タクティックを実行し、その後残ったサブゴールを証明するという手法が使えるが、C-zar の場合には自動証明タクティックがどのように働くかを推測しなければ利用することができない。

これらの欠点により、C-zar では証明の道筋が明確でない場合に証明を記述することが難しい。そこで本研究では、手続き的証明によって証明を探索した後、変換によって C-zar による証明スクリプトを生成するという手法によって、両者の長所を活かす。

4　変換手順

本研究では、次の 3 段階で手続き的証明から宣言的証明への変換を実現する。なお、変換の対象はタクティックのみとし、Goal コマンドおよび Qed コマンドは含まない。

1. 手続き的証明において各タクティックが生成する部分証明項を取得する。
2. 各ステップのゴール間の依存関係を明確にした木構造 (ゴール木) を構築する。
3. ゴール木から宣言的証明に変換する。

3 の変換では、元の証明で用いられたタクティックを justification に含めることで、1 つのタクティックを数個の文に変換する。ただし、C-zar の仕様から、一部のタクティックは justification に用いても期待される働きをしない。そのようなタクティックについては宣言的証明で直接用いず、タクティックによって生成された証明項のみから宣言的証明への変換を行う。この変換は C-zar が対応している任意の証明に対して行えるが、生成される証明が冗長になってしまう。そのため、可能な限りタクティックを用いる変換を行うものとする。

本章では、4.1 節で 1 および 2 の変換を、4.2 節で 3 の変換のうちタクティックを用いるもの、4.3 節で 3 の変換のうちタクティックを用いないものを示す。ただし、4.2 節および 4.3 節では理解し易い単純な変換の範囲に止め、4.4 節で先読みなどを行うように 3 の変換を拡張することで、より可読性の高い証明を目指す。

4.1　証明スクリプトからゴール木への変換

1 の変換は、手続き的証明を入力とし、タクティックと部分証明項、ゴールリストの 3 つ組からなる列を返す。タクティックは元の手続き的証明のもの、部分証明項はそのタクティックが生成したもの、ゴールリストはそのタクティックのサブゴールである。ソースコード 1.1 に対して 1 を適用した結果を図 4.1 に、図 4.1 の構造を図 4.2 に示す。なお、これらの図では simpl の節点の子孫は省略した。

この変換は、手続き的証明の各タクティックにそれが生成する部分証明項およびサブゴールを紐付ける変換であり、各ステップでの証明状態の差分を取るだけでよい。例えば、*tac* というタクティックを実行したとき、証明状態の部分証明項が fun x ⇒ ?g1 から fun x ⇒ ?g2 x に変化したとすると、*tac* は?g1 を?g2 x で置換するタクティックであると言える。また、ゴールリストは [g1] から [g2] に変化しているため、このステップのサブゴールは g2 のみである。よって、このステップに対応する要素は (*tac*, ?g2 x, [g2]) という 3 つ組になる。

2 の変換では、各ステップがどのステップのサブゴールに対するものであるかという依存関係を明確にする。変換の出力であるゴール木は、次の構文で表される木構造である。

```
[
  (intros, fun x y ⇒ ?g1, [g1]),
  (induction x,
    nat_ind (fun x0 ⇒ S x0 + y = x0 + S y) ?g2 (fun x0 IHx ⇒ ?g3) x,
    [g2, g3]),
  (reflexivity, @eq_refl nat (0 + S y), []),
  (simpl, ?g4, [g4]),
  ...
]
```

図 4.1　1 の出力例

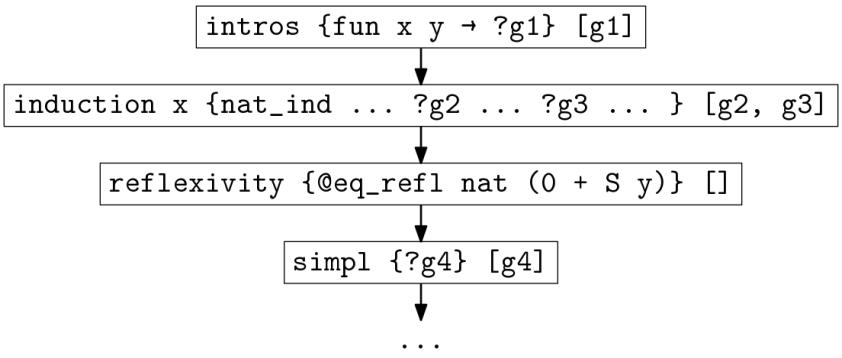

図 4.2　図 4.1 のイメージ

$T := (hyp,\ tac,\ diff,\ [x_1{:}T_1,\ \ldots,\ x_n{:}T_n])$

ゴール木の節点 1 つは元となる証明の 1 ステップ、つまりタクティック 1 つと対応する。*hyp* はこのステップで仮定が増えたかを表す真偽値、*tac* はこのステップで実行したタクティック、*diff* はこのステップで生成した部分証明項とする。また、最後のリストはこのステップにおける各サブゴールをそれぞれ部分木に対応させる。このリストを *subs* と呼ぶことにする。

図 4.1 を変換したゴール木を図 4.3 に示す。図 4.4 は図 4.3 の表すゴール木を図示したものである。角の丸い節点が仮定の増えたステップ、長方形の節点が仮定の増えていないステップを表す。

本研究では、タクティックが常にゴールリストの先頭に対してのみ実行されると仮定している。そのため、例えばあるステップに 2 つのサブゴールが存在するならば、その次のステップから 1 番目のサブゴールに対する証明が始まり、1 番目のサブゴールに対する証明が終わった次のステップから 2 番目のサブゴールに対する証明が始まると考えてよい。また、サブゴールが存在しないならば、対象のゴールの証明は終わったと考えられる。そのため、入力のリストを先頭か

```
(true, intros, fun x y ⇒ ?g1, [
  ?g1:(true, induction x,
    nat_ind (fun x0 ⇒ S x0 + y = x0 + S y) ?g2 (fun x0 IHx ⇒ ?g3) x)),
    [
      g2:(false, reflexivity, @eq_refl nat (0 + S y), []),
      g3:(false, simpl, ?g4, [g4: ...])
    ])
])
```

図 4.3　ゴール木の例

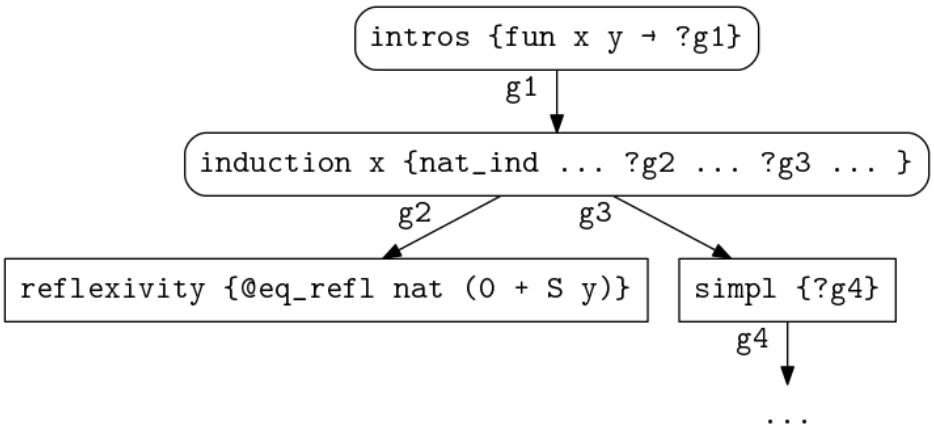

図 4.4　図 4.3 のイメージ

ら順に読み込みながら、複数のサブゴールを持つステップを分岐点として覚えておき、サブゴールの無いステップが来た時にそれ以降のステップを直近の分岐点の子として繋ぎ替える操作を繰り返すことで、変換が実現できる。

1 で出力されたリストからゴール木への変換関数 B の定義を図 4.5 に示す。Check(*diff*) は部分証明項 *diff* によって仮定が追加されたか、つまり *diff* に局所変数定義や無名関数が存在し、かつその部分項に実在変数が存在するかを true/false で表す。また、$x::l$ はリスト l の先頭に要素 x を追加したリストを表し、π_i は組の i 番目の要素を返す射影関数を表す。

分岐する際、i 番目の部分木がリストのどこまでに対応するか覚えておかなければ、$i+1$ 番目の部分木への変換の開始位置が分からない。そのため、実際の変換はゴール木とリストの残りの組を返す関数 B' で行い、B では第 1 射影 π_1 によってゴール木のみを取り出している。ここでは B' の引数として空リストが渡されることは想定していない。B' の引数として空リストが渡されるのは、証明が未完了の場合のみであり、本変換の対象外である。

$$B(l) := \pi_1(B'(l))$$

$$B'((tac,\ diff,\ [])::l) := ((\text{false},\ tac,\ diff,\ []),\ l)$$

$$B'((tac,\ diff,\ [g])::l) := ((\text{Check}(diff),\ tac,\ diff,\ [g{:}\pi_1(B'(l))]),\ \pi_2(B'(l)))$$

$$B'((tac,\ diff,\ [g_1,\ldots,g_n])::l) :=$$
$$(\text{Check}(diff),\ tac,\ diff,$$
$$[g_1{:}\pi_1(B'(l)),\ g_2{:}\pi_1(B'(\pi_2(B'(l)))),\ \ldots,\ g_n{:}\pi_1(B'(\underbrace{\pi_2(B'(\ldots\pi_2(B'}_{n-1}(l))\ldots)))),$$
$$\underbrace{\pi_2(B'(\ldots\pi_2(B'}_{n}(l))\ldots)))$$

図 4.5　2 の変換関数定義

ソースコード 4.1　後向き証明

```
1  Goal forall x y z:Prop,
2    (x → y) → (y → z) → (x → z).
3  Proof.
4    intros.
5    apply H0.
6    apply H.
7    apply H1.
8  Qed.
```

ソースコード 4.2　前向き証明

```
1  Goal forall x y z : Prop,
2    (x → y) → (y → z) → x → z.
3  proof.
4    let x:Prop, y:Prop, z:Prop,
5        H:(x → y), H0:(y → z), H1:x.
6    have x by H1 using (apply H1).
7    then y by H using (apply H).
8    hence z by H0 using (apply H0).
9  end proof.
10 Qed.
```

4.2　基本的な変換

3 の変換は基本的にゴール木の節点ごとに行う。このゴール木の根は、変換前の証明では先頭のタクティックに対応し、変換後の証明では多くの場合最後の文に対応する。変換は根から行うため、証明は末尾側から生成される。これは、タクティックによる手続き的証明がろ向き証明であり、C-zar の証明が前向き証明であることに起因している。

例えば、ソースコード 4.1 を変換するとソースコード 4.2 になる。このとき、ソースコード 4.2 の生成順は、4・5 行目、8 行目、7 行目、6 行目の順である。これはそれぞれソースコード 4.1 の 4 行目、5 行目、6 行目、7 行目に対応している。

$F(t)$:= `proof.` $F'(t,$ true$)$ `end proof.`

$F'(($false, *tac*, *diff*, []$)$, true$)$:= `thus` Typeof$($*diff*$)$ `by` Vars$($*diff*$)$ `using` *tac*`.`

$F'(($false, *tac*, *diff*, []$)$, false$)$:= `have` Typeof$($*diff*$)$ `by` Vars$($*diff*$)$ `using` *tac*`.`

$F'(($false, *tac*, *diff*, $[g{:}x]$$)$, true$)$:=
  $F'(x,$ false$)$ `hence` Typeof$($*diff*$)$ `by` Vars$($*diff*$)$ `using` *tac*`.`

$F'(($false, *tac*, *diff*, $[g{:}x]$$)$, false$)$:=
  $F'(x,$ false$)$ `then` Typeof$($*diff*$)$ `by` Vars$($*diff*$)$ `using` *tac*`.`

$F'(($false, *tac*, *diff*, $[g_1{:}(t_1, d_1, sg_1), \ldots, g_n{:}(t_n, d_n, sg_n)]$$)$, true$)$:= $(n \geq 2)$
  `claim` H_1:Typeof(d_1). $F'((t_1, d_1, sg_1),$ true$)$ `end claim.`
  ...
  `claim` H_n:Typeof(d_n). $F'((t_n, d_n, sg_n),$ true$)$ `end claim.`
  `thus` Typeof$($*diff*$)$ `by` Vars$($*diff*$)$, $H_1, \ldots, H_n$ `using` *tac*`.`

$F'(($false, *tac*, *diff*, $[g_1{:}(t_1, d_1, sg_1), \ldots, g_n{:}(t_n, d_n, sg_n)]$$)$, false$)$:= $(n \geq 2)$
  `claim` H_1:Typeof(d_1). $F'((t_1, d_1, sg_1),$ true$)$ `end claim.`
  ...
  `claim` H_n:Typeof(d_n). $F'((t_n, d_n, sg_n),$ true$)$ `end claim.`
  `have` Typeof$($*diff*$)$ `by` Vars$($*diff*$)$, $H_1, \ldots, H_n$ `using` *tac*`.`

$F'(($true, *tac*, *diff*, *subs*$)$, *goal*$)$:= $F''($*diff*, *subs*, *goal*$)$

図 4.6　変換関数の定義

変換はゴール木に関数 F を適用することによって行う。ただし、F は `proof` 文および `end proof` 文を生成するのみで、実質的な変換は下請けの関数 F' で行う。F および F' の定義を図 4.6 に示す。ここで、Typeof(t) は項 t の型、つまりその証明項が証明する命題を表し、Vars(t) は項 t 中の自由変数の集合を表す。また、各 H_i は環境に含まれない適当な名前を生成するものとする。F'' はタクティックを用いない変換関数であり、4.3 節で定義する。

F' の第 2 引数は、そのステップで示す命題がその時点における (つまり証明全体もしくは部分証明全体の) ゴールであるかを示し、主に `have` 文/`thus` 文の使い分けに用いられる。変換は、

hyp の値によって大きく 2 パターンに分けられる。*hyp* が偽、つまり仮定が増えていない場合、F' は *subs* のサイズと第 2 引数の真偽によって 6 通りに分かれる。*hyp* が真、つまり仮定が増えている場合は F'' によって、*tac* を使わずに変換する。

基本的な考え方として、任意の部分木 (*hyp*, *tac*, *diff*, *subs*) について、*subs* の各要素に対応する命題が $H_1, \ldots, H_n$ という名前で仮定に存在するとき、部分木全体に対応する命題 P は `thus` P `by` $H_1, \ldots, H_n$ `using` *tac* で表せる。なぜならば、P に対して *tac* を実行したときのサブゴールは $H_1, \ldots, H_n$ そのものであり、自動証明タクティックによって対応する仮定が自動的に使われるためである。つまり、ゴール木の根をそのように変換した後は、`thus` 文の代わりに `have` 文を用いて各サブゴールに適当な名前を付けるようにすればよい。

しかし、それだけではステップ数と同数の仮定名が生成されてしまう上、証明の構造が全く現れない。そのため、`then` 文や `hence` 文を用いて不要な仮定名を省いたり、`claim` 文/`end claim` 文を使うことで分岐先をそれぞれ一つの節として扱うようにしている。また、この方法では、対象ステップで増えた仮定をその子孫で使うことができず、常に $H_1, \ldots, H_n$ が証明できるとは限らないため、仮定が増えている場合には次節で説明する別の方法を使う必要がある。

変換規則を見れば分かるように、この変換では 1 つの節点 (タクティック) を、*subs* のサイズ (サブゴールの数) が 1 以下の時は 1 個の文に、2 以上の時は $2 *$ (サブゴール数) $+ 1$ 個の文に変換する。

4.3 タクティックを用いない変換

C-zar では、仮定の追加と命題の変形を同時に行うことは、ごく限られた場合にしか許されていない。つまり、仮定が増えている場合、1 ステップの手続き的証明に対応する 1 ステップの宣言的証明があるとは限らない。そのため、F'' ではタクティックは使わず、証明項 *diff* の各要素に対応する C-zar の構文に変換する。

F'' の定義を図 4.7 に示す。ここで Select(*subs*,?x) は、*subs* において実在変数?x に対応す部分木を表す。F'' で生成された `have` 文などは justification でタクティックを指定しないが、`by` 部で証明項がそのまま与えられているため、自動証明が可能である。

C-zar は同名の仮定が複数定義されることを許さないため、ここで現れる名前 x は、出現箇所での環境に含まれていないものであるとする。実際には、重複する名前が現れた場合、適当な名前を付け直すことは容易である。

他の多くの文と異なり、`let` 文は後向き証明ステップであり、`let` 文をそのまま出力してしまうと、その影響範囲が変換前と異なってしまう場合がある。*goal* が false の場合にそのような事態が発生するため、`claim` 句にすることでその影響範囲を限定している。

関数型および依存関数型は項の型もしくは命題を表す項であり、証明項ではないことが明らかであるため、`define` 文を用いることで以降の変換を省略する。なお、ここで t に実在変数が含まれている場合には不正な証明が生成されてしまうが、そのようになる可能性は低く、たとえ含

まれる場合であっても、その項を内部で展開するという対策が可能である。

なお、無名再帰関数および無名余再帰関数に関しては、対応する C-zar の構文が存在しないため、本研究では対象としない。

4.4 可読性の向上

ここまでで変換は行えるようになったが、このまま F を用いると、不必要に長大な証明が生成されてしまう。より人間が書いたものに近い証明を出力し可読性を高めるために、変換関数に対して以下の 5 つの拡張を施す。

4.4.1 claim 句から have 文への変更

例として、次の証明を考える。

```
1 Goal forall A B:Prop, A → B → A∧B.
2   intros. split. apply H. apply H0.
3 Qed.
```

これを F によって変換すると、次の証明になる。

```
1 Goal forall A B:Prop, A → B → A∧B. proof.
2   let A:Prop. let B:Prop. let H:A. let H0:B.
3   claim H1:A. thus A by H using apply A. end claim.
4   claim H2:B. thus B by H0 using apply B. end claim.
5   thus A∧B by H1, H2 using split.
6 end proof. Qed.
```

3 行目および 4 行目において claim 句は名前を付ける機能しか持っていないため、それぞれ一つの have 文に置き換えることができる。このように、claim 句の中で 1 ステップしか行わない場合、have 文に置き換えて簡潔にした方が分かり易い。ゴール数が増加するステップ (split) ではなく、その次のステップ (apply) で claim 文が必要かを判断するように F' および F'' を拡張すれば、無駄な claim 文の削減を実現できる。サブゴールを持つか、つまり $subs$ が空でないかを見れば、claim 句にすべきか have 文にすべきかはすぐに判定できる。

上の証明に対して claim 句から have 文への変更を適用すると、次のようになる。

```
1 Goal forall A B:Prop, A → B → A∧B. proof.
2   let A:Prop. let B:Prop. let H:A. let H0:B.
3   have H1:A by H using apply A.
4   have H2:B by H0 using apply B.
5   thus A∧B by H1, H2 using split.
6 end proof. Qed.
```

$F''(\texttt{?}x,\ subs,\ goal) := F'(\text{Select}(subs,\ \texttt{?}x),\ goal)$

$F''(x,\ subs,\ \text{true}) := \texttt{thus}\ \text{Typeof}(x)\ \texttt{by}\ x.$

$F''(x,\ subs,\ \text{false}) := \texttt{have}\ \text{Typeof}(x)\ \texttt{by}\ x.$

$F''(t_0\ \ldots\ t_n,\ subs,\ \text{true}) :=$
$\quad \texttt{claim}\ H_0 : \text{Typeof}(t_0).\ F''((t_0,\ \text{true})\ \texttt{end claim}.$
$\quad \ldots$
$\quad \texttt{claim}\ H_n : \text{Typeof}(t_n).\ F''((t_n,\ \text{true})\ \texttt{end claim}.$
$\quad \texttt{thus}\ \text{Typeof}(\mathit{diff})\ \texttt{by}\ H_0\ \ldots\ H_n.$

$F''(t_0\ \ldots\ t_n,\ subs,\ \text{false}) :=$
$\quad \texttt{claim}\ H_0 : \text{Typeof}(t_0).\ F''((t_0,\ \text{true})\ \texttt{end claim}.$
$\quad \ldots$
$\quad \texttt{claim}\ H_n : \text{Typeof}(t_n).\ F''((t_n,\ \text{true})\ \texttt{end claim}.$
$\quad \texttt{have}\ \text{Typeof}(\mathit{diff})\ \texttt{by}\ H_0\ \ldots\ H_n.$

$F''(\texttt{let}\ x := t_0\ \texttt{in}\ t_1,\ subs,\ goal) :=$
$\quad \texttt{claim}\ x : \text{Typeof}(t_0).\ F''(t_0,\ subs,\ \text{true})\ \texttt{end claim}.\ F''(t_1,\ subs,\ goal)$

$F''(\texttt{fun}\ x \Rightarrow t,\ subs,\ \text{true}) := \texttt{let}\ x : \text{Typeof}(x).\ F''(\text{t},\ subs,\ \text{true})$

$F''(\texttt{fun}\ x \Rightarrow t,\ subs,\ \text{false}) :=$
$\quad \texttt{claim}\ \text{Typeof}(\text{fun}\ x \Rightarrow t).\ F''(\text{fun}\ x \Rightarrow t,\ subs,\ \text{true})\ \texttt{end claim}.$

$F''(t_1 \to t_2,\ subs,\ \text{true}) := \texttt{thus thesis by}\ (t_1 \to t_2).$

$F''(t_1 \to t_2,\ subs,\ \text{false}) := \texttt{define}\ H\ \texttt{as}\ (t_1 \to t_2).$

$F''((\texttt{forall}\ x,\ t),\ subs,\ \text{true}) := \texttt{thus thesis by}\ (\texttt{forall}\ x,\ t).$

$F''((\texttt{forall}\ x,\ t),\ subs,\ \text{false}) := \texttt{define}\ H\ \texttt{as}\ (\texttt{forall}\ x,\ t).$

F''(match t with $c_1\ t_{11} \ldots t_{1j} \Rightarrow t_1 \mid \ldots \mid c_i\ t_{i1} \ldots t_{ik} \Rightarrow t_i$ end, *subs*, true) :=
  per cases on t.
  suppose it is t_{11} ... t_{1j}. $F''(t_1, subs, \text{true})$
  ...
  suppose it is t_{i1} ... t_{ik}. $F''(t_i, subs, \text{true})$
  end cases.

F''(match t with $c_1\ t_{11} \ldots t_{1j} \Rightarrow t_1 \mid \ldots \mid c_i\ t_{i1} \ldots t_{ik} \Rightarrow t_i$ end, *subs*, false) :=
  claim Typeof(t_1).
  F''(match t with $c_1\ t_{11} \ldots t_{1j} \Rightarrow t_1 \mid \ldots \mid c_i\ t_{i1} \ldots t_{ik} \Rightarrow t_i$ end, *subs*, true)
  end claim.

図 4.7　証明項のみによる変換関数の定義

4.4.2　仮定の重複回避

claim 句から have 文への変更を施した上記の例について、H1 や H2 が示す命題は H や H0 と同一であり、新たに証明する必要はない。H1 の証明項は H そのものであり、単に別名をつけたに過ぎないため、不要なステップである。つまり、次のように単純化できる。

```
1 Goal forall A B:Prop, A → B → A∧B. proof.
2   let A:Prop. let B:Prop. let H:A. let H0:B.
3   thus A∧B by H, H0 using split.
4 end proof. Qed.
```

このような単純化を行うには、ゴール数が増加するステップ、つまり子の数が 2 以上である節点の変換で子の *diff* が変数であるかどうかを確認する方法が最も簡単である。

4.4.3　証明項ではない項の扱いの改善

証明項の内部には、値を示す項がしばしば含まれるが、これを証明項として変換してしまうと不自然な証明が生成されてしまう。例えば、F''(x+2,[],false) は次のようになる。ここで、F'' はこれまでに説明した拡張を施しているものとする。

```
1 claim H:nat.
2   have H0:nat by S 0. thus nat by S H0.
3 end claim.
4 have nat by (x + H).
```

多くの場合、項が証明項であるかどうかは型を見れば分かるため、証明項でなかった場合には define 文を用いて直接定義することで、このような証明の生成を回避する。上の例であれば、define H as x + 2 という文に変換される。

4.4.4 justification の省略

Vars(*diff*) の結果として多くの仮定が出てきた場合、明示すると却って意図が分かりにくくなる。また、実際の Coq では暗黙の引数になっている項なども明示するようになっているため、by 部で指定する必要の無い場合もある。例えば、ソースコード 1.1 の 8 行目の f_equal タクティックを変換するとき、Vars(*diff*) は 6 個の仮定のリストになるが、これらは指定しなくともよい。

数が多い場合にはこのように不要な項が複数含まれている可能性が高いため。*で置き換えるようにする。

4.4.5 induction 文の利用

Coq では、induction タクティックはよく使われるタクティックである。induction タクティックは帰納法の仮定を追加するため、証明項を経由する変換になるが、induction タクティックの作る証明項は関数適用であり、帰納法を用いていることが分かり難い。例えば、次のような証明を考える。

```
1 Lemma example_ind x:nat, x = x.
2   induction x. auto. auto.
3 Qed.
```

ここでは自然数 x について帰納法を使い、x が 0 の場合と S x0 の場合それぞれを auto によって証明している。例を簡潔にするため、実際には必要の無い帰納法を使った。これを変換すると、次のようになる。

```
1 have H1: 0 = 0 using auto.
2 have H2: forall x0, x0 = x0 → (S x0) = (S x0) using auto.
3 thus x = x by (@nat_ind (fun x0 ⇒ x0 = x0 ) H1 H2 x).
```

この証明は証明項をそのまま用いているため、帰納法を用いていることが分かり難い。C-zar では帰納法に対応する induction 文が用意されているため、これを利用すると、次のように変換できる。

```
1 per induction on x.
2 suppose it is 0.
3   thus 0 = 0 using auto.
4 suppose it is (S x0) and IHx:(x0 = x0).
5   thus (S x0) = (S x0) using auto.
6 end induction.
```

induction タクティックは nat_ind のような、帰納法用の特殊な関数を利用する。これを検出して induction 文に変換することで、可読性を向上する。

5　実装と実験

本研究では、前章で示した変換について、実際に Coq のプラグインとして実装した。さらに、Coq 標準ライブラリの一部に対して本変換を適用し、変換対象となる証明の割合の測定および、既存手法に比べて元の証明に近い証明が生成されていることの確認を行った。

5.1　実装

プラグインにより、Coq に DProof コマンドおよび DEnd コマンドの 2 つを追加した。コマンドを加えた証明の例をソースコード 5.1 に示す。なお、Require コマンドは、プラグインを有効化するためのものである。DEnd コマンドが実行された時、DProof コマンドから DEnd コマンドまでの証明を C-zar に変換し、表示する。ソースコード 5.1 を実行した場合であれば、図 5.1 のように表示される。

この 2 つのコマンドは証明の開始時や終了時だけでなく、証明の任意の箇所に挿入することができる。証明の一部のみを DProof コマンドおよび DEnd コマンドで囲んだ場合、その部分のみを変換して表示する。ただし、DEnd コマンドよりも前の時点で DProof コマンド時点のゴールの証明が完了した場合にはそこまでの証明しか変換せず、DEnd コマンドの時点で証明が完了していない場合には証明すべき部分にコメントを挿入する。

例えば、ソースコード 5.2 を実行すると、7 行目から 9 行目のみが変換され、ソースコード 5.3 が変換結果として出力される。

ソースコード 5.1　プラグイン使用例

```
1  Require Import dp.DProof.
2  Goal forall x y : nat,
3    S x + y = x + S y.
4  DProof.
5    intros.
6    induction x.
7    reflexivity.
8    simpl.
9    f_equal.
10   apply IHx.
11 DEnd.
12 Qed.
```

```
  ============================
  S (S (x + y)) = S (x + S y)

Unnamed_thm < f_equal.
1 subgoal

  x, y : nat
  IHx : S x + y = x + S y
  ============================
  S (x + y) = x + S y

Unnamed_thm < apply IHx.
No more subgoals.

Unnamed_thm < DEnd.
(* not tactic, ignored: DEnd. *)
Goal forall x y : nat, S x + y = x + S y.
proof.
  let x:nat, y:nat.
  per induction on x.
  suppose it is 0.
    thus (1 + y = 0 + S y) by y using reflexivity.
  suppose it is (S x0) and IHx:(S x0 + y = x0 + S y).
    have (S (S (x0 + y)) = S (x0 + S y)) by IHx, x0, y using f_equal.
    hence (S (S x0) + y = S x0 + S y) using (simpl).
  end induction.
end proof.
Qed.


Unnamed_thm <
```

図 5.1　DEnd コマンド実行の様子

ソースコード 5.2　部分的な適用の例

```
1  Require Import dp.DProof.
2  Goal forall x y : nat,
3    S x + y = x + S y.
4  Proof.
5    intros.
6  DProof.
7    induction x.
8    reflexivity.
9    simpl.
10 DEnd.
11   f_equal.
12   apply IHx.
13 Qed.
```

ソースコード 5.3　ソースコード 5.2 の出力

```
1  Goal forall x y:nat, S x + y = x + S y.
2  proof.
3    let x:nat.
4    let y:nat.
5    per induction on x.
6    suppose it is 0.
7      thus (0 + S y = 0 + S y) by Nat.add, y
8                               using reflexivity.
9    suppose it is (S x0)
10     and IHx:(S x0 + y = x0 + S y).
11     (* write your proof *)
12     thus (S (S x0) + y = S x0 + S y) using simpl.
13   end induction.
14 end proof.
15 Qed.
```

部分的な変換を行い、その変換結果によって元の証明を置き換える場合は、単なる置換では 2 つの証明スタイルが混在することになる。宣言的証明の部分のみを補題として切り出すことで、混在を避けることが望ましい。そのため、C-zar による証明だけでなく、Goal コマンドおよび Qed コマンドも同時に出力することで、独立した証明として扱えるようにした。

この時、DProof コマンド実行時点での仮定 (ローカル環境) を反映してゴールおよび環境を書き換える必要がある。ここでは各仮定 $x\colon T$ に対してゴールに forall $x\colon T$ を、証明に let $x\colon T$ を書き加えることとした。

DProof コマンドが実行されると、その時点での証明状態を保存し、標準入力の記録を開始する。Coq では、入力されたタクティックの情報をプラグインが取得することはできないため、標準入力を取得し、構文解析を行う必要がある。このため、本プラグインは標準入力を用いない CoqIDE などでは利用することができない。

DEnd コマンドが実行されると、DProof コマンドで保存した証明状態を復元した後、記録したタクティックを順に実行し、各時点での証明状態を、部分証明項の差分をとりながら記録する。これは 4 章で説明した変換の第 1 段階にあたる。その後、続けて第 2 段階および第 3 段階の変換を行うことで、C-zar による証明を生成する。

5.2　実験

タクティックを用いた変換によってどの程度読みやすくなったかを確認するために、タクティックを用いない変換と本研究の手法での記述量を比較した。

比較対象として、Coen による手続き的証明から宣言的証明への変換が考えられるが、変換対象は Matita の証明言語であり、Coq のものとは異なるため、そのまま Coq で用いることはで

表 5.1　変換結果

	総行数	総文字数	総ステップ数
変換前	530	13,899	676
A で生成した証明	2,235	56,357	1,600
B で生成した証明	14,641	512,789	7,431

きない。Coen の変換は、証明項のみを元に宣言的証明を生成する。そこで、証明項のみから変換した証明の大きさは全て元の証明項の大きさに従うという仮定の下で、同じく証明項のみからの変換である F'' にその定理の証明項全体を与え代用とする。

以下、本研究の変換手法 (F によるステップ毎の変換) を A、比較対象である証明項の情報のみを用いた変換手法を B と呼ぶ。

形式証明の読み易さについて定量的な計測は困難であるため、ここでは証明の大きさに着目する。証明項のみからの変換では生成される証明が巨大なものとなってしまう問題が指摘されている [16] ため、より簡潔な証明が生成されていることを確認することで、読み易い証明となっていることを示す。

Coq の標準ライブラリに含まれる、Arith モジュールの定理 264 個に対して変換を行い、変換前後における行数・文字数・ステップ数を調べた。ステップ数とは、手続き的証明においてはタクティック数、宣言的証明においては `proof` 文および `end proof` 文以外の文の数である。また、行数は空行を、文字数は空白文字およびコメントを含まないものとして数えた。各変換によって生成された 265 個の証明について、行数・文字数・ステップ数をそれぞれ合計したものを表 5.1 に示す。

A では行数・文字数・ステップ数のいずれにおいても B に比べて大幅に少なく、タクティックの利用による冗長な部分の削減は効果があると考えられる。以下では、ステップ数について詳細に見る。

264 個の定理それぞれについて、A によるステップ数の増加率を図 5.2 に、B によるステップ数の増加率を図 5.3 に示す。なお、F の定義では 1 ステップのみからなる手続き証明は必ず 1 ステップのみからなる宣言的証明に変換されるはずであるが、図 5.2 では 2 ステップの証明がある。これは、`Lemma` コマンドの引数に対応するステップである。

A では変換後のステップ数が概ね変換前の 1 倍から 10 倍程度に収まっているのに対し、B では一部の証明が非常に大きなものになってしまっている。これは、たとえ変換前が 1 ステップのみの証明であったとしても、それが自動証明タクティックなどであれば証明項は大きなものになる場合があるためである。図 5.3 において最大のものでは 1 ステップの手続き的証明から 631 ステップの宣言的証明を生成している。

一方、B には変換後の方が変換前に比べステップ数が少ないものもある。これは、複数ステップかけて生成された証明項を、C-zar の証明中に直接記述することで 1 ステップとしている。ス

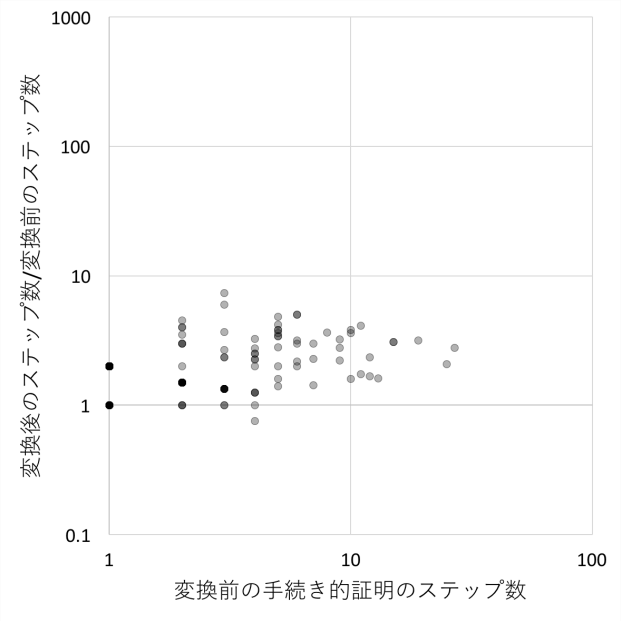

図 5.2　提案手法による変換 (A) のステップ数

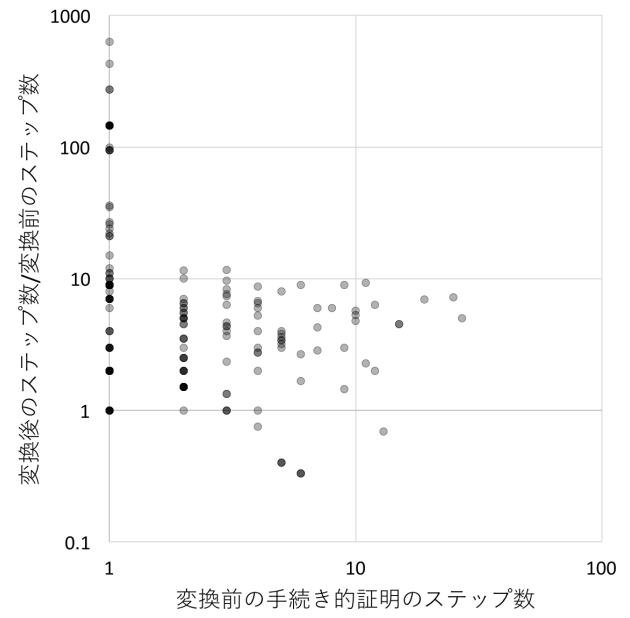

図 5.3　証明項のみによる変換 (B) のステップ数

ソースコード 5.4　変換前の証明

```
1 Lemma add_assoc n m p : n + (m + p) = n + m + p.
2   destruct n.
3   - trivial.
4   - apply add_assoc_pos.
5   - apply opp_inj. rewrite !opp_add_distr. simpl. apply add_assoc_pos.
6 Qed.
```

テップ数こそ減っているものの、証明項から証明の流れを読み取るのは困難である。

実際の変換例をソースコード 5.4 とソースコード 5.5、ソースコード 5.6 に示す。ソースコード 5.4 は整数の加算の結合則を示す手続き的証明であり、変換の入力とする。ソースコード 5.5 は A による変換結果、ソースコード 5.6 は B による変換結果である。

この証明は命題 $\forall n, (2 * n)/2 = n$ を証明している。ソースコード 5.4 では、induction によって n について帰納法を行い、n が 0 の場合は trivial タクティックによる自動証明で、n が S n0 の場合には等式変形を行い帰納法の仮定と同じ形にすることで証明を行っている。ソースコード 5.5 ではどこで n が 0 の場合の証明を行ったか、また等式がどのように変形されているかが明示されており、変換によって可読性が上がっていると言える。

ソースコード 5.6 でも同様の情報は示されているが、行数がおよそ倍になっており、証明全体の流れを掴むことは少し難しい。10 行目から 21 行目にかけては特に冗長で、実際にはこの 12 行のうち必要なのは (仮定名などに手を加える必要はあるものの)18 行目のみである。この 12 行は f_equal タクティックによって生成された証明項を変換したものであるが、f_equal は本来、

ソースコード 5.5　A で変換された証明

```
1  Goal forall (n:nat), div2 (2 * n) = n.
2  proof.
3    let n:nat.
4    per induction on n.
5    suppose it is 0.
6      thus (0 = 0) by (@Logic.eq_refl nat 0).
7    suppose it is (S n0) and IHn:(div2 (2 * n0) = n0).
8      have (S (div2 (n0 + (n0 + 0))) = S n0) by IHn, n0 using now f_equal.
9      then (div2 (S (S (n0 + (n0 + 0)))) = S n0) using (simpl).
10     then (div2 (S (n0 + S (n0 + 0))) = S n0) by n0 using (rewrite add_succ_r).
11     hence (div2 (2 * S n0) = S n0) using (simpl mul).
12   end induction.
13 end proof.
14 Qed.
```

ソースコード 5.6　B で変換された証明

```
1  Goal forall (n:nat), div2 (2 * n) = n.
2  proof.
3    let n:nat.
4    per induction on n.
5    suppose it is 0.
6      thus (0 = 0) by (@Logic.eq_refl nat 0).
7    suppose it is (S n0) and IHn:(div2 (2 * n0) = n0).
8      claim HLo:(S (Init.Nat.div2 (n0 + (n0 + 0))) = S n0).
9        have H: (div2 (n0 + (n0 + 0)) = n0) by IHn.
10       claim (S (Init.Nat.div2 (n0 + (n0 + 0))) = S n0).
11         claim HLo1:(S (Init.Nat.div2 (n0 + (n0 + 0))) =
12                     S (Init.Nat.div2 (n0 + (n0 + 0)))).
13           have HLo3:(S = S) by (@Logic.eq_refl (nat → nat) S).
14           hence (S (Init.Nat.div2 (n0 + (n0 + 0))) =
15                  S (Init.Nat.div2 (n0 + (n0 + 0)))) by (f_equal
16             (fun f : nat → nat ⇒ f (Init.Nat.div2 (n0 + (n0 + 0)))) HLo3).
17         end claim.
18         have HLo2:(S (Init.Nat.div2 (n0 + (n0 + 0))) = S n0) by (f_equal S H).
19         hence (S (Init.Nat.div2 (n0 + (n0 + 0))) = S n0) by (Logic.eq_trans HLo1 HLo2).
20       end claim.
21       hence thesis.
22     end claim.
23     have HLo0:(n0 + S (n0 + 0) = S (n0 + (n0 + 0))) by (add_succ_r n0
24       (n0 + 0)).
25     thus (div2 (S (n0 + S (n0 + 0))) = S n0) by (eq_ind_r
26       (fun n : nat ⇒ div2 (S n) = S n0) HLo HLo0).
27   end induction.
28 end proof.
29 Qed.
```

ソースコード 5.7　1 ステップの証明の例

```
1 Lemma even_plus_aux n m :
2     (odd (n + m) ↔ odd n ∧ even m ∨ even n ∧ odd m) ∧
3     (even (n + m) ↔ even n ∧ even m ∨ odd n ∧ odd m).
4 Proof.
5   rewrite ?even_equiv, ?odd_equiv, <- ?Nat.even_spec, <- ?Nat.odd_spec;
6   rewrite ?Nat.even_add, ?Nat.odd_add, ?Nat.even_mul, ?Nat.odd_mul;
7   unfold Nat.odd; do 2 destruct Nat.even; simpl; tauto.
8 Qed.
```

f x = g y の形の命題を証明するために f x = g x および g x = g y に命題を分割し、それぞれに自動証明を試みるタクティックである。この証明では f と g にあたるものが同じ S であるため、その大部分が意味の無いものになってしまっている。手続き的証明では型さえ合っていれば生成される証明項に注意が向けられることは少ないため、冗長な証明項を生成するタクティックは多く、B による証明が肥大化する大きな原因となっている。

図 5.3 では、変換によるステップ数の増加が 10 倍を超える場合のほとんどにおいて、元の証明のステップ数が 1 であることが確認できる。その理由としては、Arith モジュールの大部分、具体的には 264 個の証明のうち 156 個が 1 ステップの証明であること、巨大な証明項を生成するのは多くの場合自動証明タクティックのような強力なタクティックであるため 1 ステップで終わらせ易いことなどが考えられる。Arith モジュール中の 1 ステップの手続き的証明の例をソースコード 5.7 に示す。実際には 1 つのユーザ定義タクティックによって書かれているが、ここではその定義を展開した。

ソースコード 5.7 は多くのタクティックで構成された証明のように見えるが、セミコロンは複数のタクティックを結合して 1 つのタクティックにするため、全体で 1 ステップと数えられる。ソースコード 5.7 に B を使って変換を行うと、429 ステップの C-zar の証明が生成される。一方、A では 1 ステップの手続き的証明は必ず 2 ステップ以内の証明に変換可能である。そのため、1 ステップの証明の変換を含めて比較するのはあまり公平とはいえない。また、1 ステップの証明はサブゴールが存在しないため、C-zar への変換があまり意味を持たない。

Arith モジュールのうち、変換が実用的に適用できる 92 個の証明についての変換結果を表 5.2 に示す。ここでは、2 ステップ以上かつ Qed コマンドを用いている証明を対象とした。Arith モジュールには Qed コマンドの代わりに Defined というコマンドを用いている証明があるが、これは証明によって関数を構成しており、生成される証明項が重要である。変換後の証明による証明項は変換前の証明による証明項と異なったものとなるため、あまり現実的ではない。行数・文字数・ステップ数のいずれにおいても、表 5.1 ほど極端ではないものの、やはり A の方が簡潔な証明が生成できていることが確認できる。

また、各定理について A で変換した場合と B で変換した場合のステップ数の比を図 5.4 に示

表 5.2　一部の証明に関する変換結果

	総行数	総文字数	総ステップ数
変換前	295	8,381	427
A で生成した証明	1,425	37,396	1,060
B で生成した証明	2,841	72,641	2,107

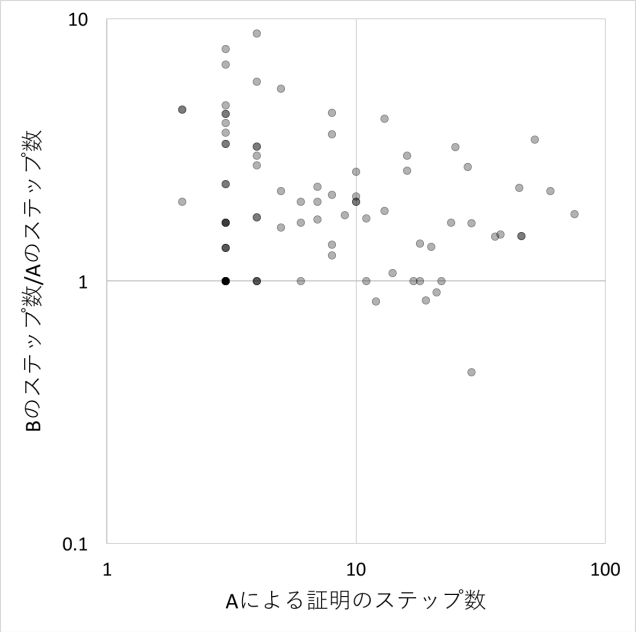

図 5.4　両変換のステップ数の比

す。B の方が少ないステップ数になっている例も若干見られるものの、基本的には A の方がステップ数が少ない。

全体として、A の方が B よりも元の手続き的証明に近い宣言的証明を生成できていると言える。

5.3　変換対象外の証明

Arith モジュールには、265 個の証明の他に本システムの対象外となるような証明も含まれており、これらは変換によって正しくない C-zar の証明を生成した。本節では対象外の証明の内訳と対処法について述べる。

正しく変換できない証明は 25 個あり、そのうち 13 個は依存パターンマッチによるものだった。依存パターンマッチとは、分岐先によって値の型が異なるパターンマッチである。例えば、次に示す項は、`x` が `0` の場合には `nat` 型の値 `1` であり、それ以外の場合は `bool` 型の値 `true` である。一見矛盾しているようだが、この項全体で `match x with 0 => nat | _ => bool end` 型の値であると考えることで正しく型付けされる。

```
match x as x0
  return match x0 with 0 ⇒ nat | _ ⇒ bool end
with
  | 0 ⇒ 1
  | _ ⇒ true
end
```

依存パターンマッチを証明項に含む証明の一部は、正しく変換することができない。claim 句にするなど適切な処置によって正しく変換することも可能と考えられるが、今回は対応していない。

2 番目に多かったのは証明項に無名再帰関数を含むもので、9 個あった。無名再帰関数を生成するステップは必ず仮定を追加するため、証明項を経由する変換となり、また C-zar に対応する構文が無いため、変換に失敗する。以降のステップを 1 つにまとめ、証明項を直接表示することで正しい証明は得られるものの、証明として読み易いものではないため行わなかった。

その他の失敗する証明としては、C-zar のバグが原因と考えられるものなどもあった。今回はタクティックが対象外の挙動をする証明は無かったが、状況によっては対象外の挙動を起こしうるタクティックは存在した。例えば auto タクティックは、生成する証明項が大きい場合、それを補題としてグローバル環境に追加するという機能を持つ。これは証明項の大きさを抑えるための処置だと考えられるが、同様の証明を C-zar で行うことは困難である。C-zar は部分的に手続き的証明に切り替える機能も持つため、これを利用することも考えられるが、そのためには対象外の挙動を検知する必要がある。検知にはグローバル環境の監視なども必要となり困難であるため、本研究では不正な証明が生成されるようになっている。

正しく変換できなかった証明は全体のおよそ 1 割と無視できない割合であり、対象外の証明であってもそれをユーザが判断することは容易ではない。変換対象の拡大および対象外の証明の検知は今後の課題である。

6 関連研究

類似した目的を持つ研究として、手続き的証明を宣言的証明に変換する研究に加え、Coq の手続き的証明を理解し易い形で表示する研究、宣言的証明の記述を補助する研究を挙げる。

既に述べたように、定理証明支援系 Matita では、証明項から変換可能な宣言的証明言語が実装されている。手続き的証明を実行して得た証明項に対してこの変換を行うことで、宣言的証明が生成できる。しかしこの変換は、手続き的証明から宣言的証明への変換のみならず、一部を省略して書かれた宣言的証明の補完も主な用途としている。そのため、この手法で生成される宣言的証明は、実験における変換 B のように詳細かつ長大なものとなり、証明全体の流れは把握し難い。本研究では、証明項だけでなく元の証明のタクティックも利用することで、変換前後の証明の粒度を近づけ、より可読性の高い証明を生成する。

手続き的証明を理解し易い形で表示する研究としては、Coq のウェブインタフェース ProofWeb [30] による証明木形式や Fitch 記法での表示機能 [31] 、coqdoc による HTML 文書に各ステップでの証明状態を表示する機能を付加する Proviola [32] などが挙げられる。これらは、特定の状況において理解し易い表示を行う試みであり、証明スクリプト自体の可読性および外部ツールとの親和性の向上を図る本研究とは、目的が若干異なる。可読性のみであれば、手続き的証明におけるタクティックの働きを説明するコメントを証明スクリプトに追加する Coqatoo [33] がある。タクティックの働きは「どのような命題をどのような操作を行うか」という形で示され、宣言的証明言語に近いコメントが生成される。しかし、Coqatoo は初学者の学習用ツールであるため、一部のタクティックにのみ対応しており、複雑な証明は対象としていない。

宣言的証明の記述を補助する研究は Coq には存在しないものの、他の定理証明支援系においては既存研究が存在する。miz3 [16] は、HOL Light における宣言的証明の入力補助システムである。タクティックに似たコマンドによって、宣言的証明のひな型を生成する。その際、入力している箇所に応じて命題などを自動補完するため、ステップ毎に手続き的証明から宣言的証明への変換を行っている状態に近い。ただし、タクティックそのものを使えるわけではないため、Coq に導入することは難しいと考えられる。また、宣言的タクティック [15] は、宣言的証明のテンプレートを定義できるシステムであり、強力な入力補助システムとなる。Isabelle/Isar では Sledgehammer と呼ばれる機構が証明を探索することによって、ユーザが証明を考え記述する場面を少なくすることで、宣言的証明の記述の煩わしさを大きく軽減する。

7 おわりに

本研究では、Coq の手続き的証明から C-zar による宣言的証明への変換を定義した。また、実際に変換を行い、タクティックを変換に用いることが、冗長な証明の生成を回避するために有効であることを確認した。本システムを用いることで、従来の書き易さを損なうことなく、可読性および保守性の高い証明を生成することができる。

一方で、必ずしも全ての証明が読み易く変換できるとは言えない。より可読性の高い証明を生成するためには、以下のような改良が考えられる。

- C-zar の多様な構文の利用
  C-zar は等式変形など一部の証明に関して特別な構文を持つ。`induction` 文もこの一種である。特化した構文であることから意図が伝わり易くなるため、これらの構文を利用することでより明快な証明を生成する。
- C-zar の部分命題証明の利用
  `thus` 文では、ゴール $P \wedge Q$ における P のような、ゴールの一部に相当する命題も証明することができる。これを用いることで、`claim` 文を減らし、多段ネストを抑制する。
- 結合されたタクティックの分解
  例えば `intros` タクティックと `auto` タクティックはそれぞれ 1 文に変換可能であるが、`intros; auto` は証明項のみから変換する必要があるため、生成される宣言的証明は膨大なものになり得る。前後のタクティックを別ステップとして扱うことで、生成される証明の長さを抑える。
- by 部の単純化
  `thus` 文などの by 部では *diff* で出現する全変数を指定しているが、必ずしもその全てが必要なわけではない。本当に必要なもののみを抽出することで、そのステップの意図を明確にする。
- 省略可能な部分の明示
  `have` 文などに備わる自動証明機能は強力であるため、変換後の証明においてタクティックの省略や複数ステップの統合、ステップ自体の省略が可能な場合が多い。これらの箇所をユーザに示すことで、より本質的な部分に注目し易くする。

本システムの最大の課題は、現時点での最新バージョンである Coq8.7 への対応だろう。8.7 では C-zar が標準プラグインから削除され、さらにプラグインは用意された API のみでしか Coq 内部にアクセスできないように変更された。そのため、本システムを Coq8.7 に対応させるには、まず C-zar の修正が必要になる。単なる修正にとどまらず、C-zar を改良することも考えられる。例えば、C-zar は余帰納法を使った証明を記述することができない。Coq の任意の手続

き的証明を C-zar へ変換可能にするには、言語仕様自体を拡張する必要がある。

本システムを有効に活用するには、C-zar に対するリファクタリングシステムが望まれる。本システムが生成する証明スクリプトには、まだ改善の余地が多い。そもそも手続き的証明と宣言的証明というスタイルの違いから、不自然な証明スクリプトが生成されることは避け難い。定理証明支援系の内部状態を取得できなくとも証明の情報が得られるという宣言的証明の利点を活かし、プログラミングにおける IDE に備わっているような半自動リファクタリングが可能になれば、不自然な証明の修正は容易になる。こういったツールに関しては Whiteside らの研究 [14] が存在するが、Coq での利用には至っていない。

リファクタリングシステムが存在しない現状では、命題に大きな変更が加わったなどで大幅な修正が必要になった場合、C-zar を修正するのではなく、手続き的証明を用いた方が有効な場合が多い。C-zar は手続き的証明を埋め込める `escape` 文を用意しているが、手続き的証明と宣言的証明の混在は保守性の面から望ましくない。このようなとき、C-zar から手続き的証明に変換できるシステムによって、2 つのスタイルを行き来することができれば有用である。

こうした周辺ツールが整うことで、宣言的証明言語はその真価を発揮すると期待される。

謝辞

本研究に際して、様々なご指導を頂きました中野圭介准教授、岩崎英哉教授に深謝いたします。また、研究の助言など様々な面で助けていただいた、中野研究室、岩崎研究室の皆様に心から御礼申し上げます。

参考文献

[1] Thomas Hales et al. “A Formal Proof of the Kepler Conjecture”. In: *Forum of Mathematics, Pi* 5 (2017). DOI: 10.1017/fmp.2017.1.

[2] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: https://homotopytypetheory.org/book, 2013.

[3] *seL4*. URL: https://sel4.systems/ (visited on 01/01/2018).

[4] *CakeML*. URL: https://cakeml.org/ (visited on 01/01/2018).

[5] *The Coq Proof Assistant*. URL: https://coq.inria.fr/ (visited on 01/01/2018).

[6] Georges Gonthier. “Formal proof—the four-color theorem”. In: *Notices Amer. Math. Soc.* 55.11 (2008), pp. 1382–1393. ISSN: 0002-9920.

[7] Georges Gonthier et al. “A Machine-Checked Proof of the Odd Order Theorem”. In: *ITP 2013, 4th Conference on Interactive Theorem Proving*. Ed. by Sandrine Blazy, Christine Paulin, and David Pichardie. Vol. 7998. LNCS. Rennes, France: Springer, July 2013, pp. 163–179. DOI: 10.1007/978-3-642-39634-2_14.

[8] *CompCert*. URL: http://compcert.inria.fr/ (visited on 01/01/2018).

[9] *VellVM: Verified LLVM*. URL: http://www.cis.upenn.edu/~stevez/vellvm/ (visited on 01/01/2018).

[10] *Frama-C*. URL: https://frama-c.com/ (visited on 01/01/2018).

[11] Eelco Visser et al. “A Language Designer’s Workbench: A One-Stop-Shop for Implementation and Verification of Language Designs”. In: *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2014. Portland, Oregon, USA: ACM, 2014, pp. 95–111. ISBN: 978-1-4503-3210-1. DOI: 10.1145/2661136.2661149. URL: http://doi.acm.org/10.1145/2661136.2661149.

[12] Pierre Corbineau. “A Declarative Language for the Coq Proof Assistant”. In: *Types for Proofs and Programs: International Conference, TYPES 2007, Cividale des Friuli, Italy, May 2-5, 2007 Revised Selected Papers*. Ed. by Marino Miculan, Ivan Scagnetto, and Furio Honsell. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 69–84. ISBN: 978-3-540-68103-8. DOI: 10.1007/978-3-540-68103-8_5. URL: https://doi.org/10.1007/978-3-540-68103-8_5.

[13] Iain Whiteside et al. “Towards Formal Proof Script Refactoring”. In: *Intelligent Computer Mathematics: 18th Symposium, Calculemus 2011, and 10th International Con-*

ference, MKM 2011, Bertinoro, Italy, July 18-23, 2011. Proceedings. Ed. by James H. Davenport et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 260–275. ISBN: 978-3-642-22673-1. DOI: 10.1007/978-3-642-22673-1_18. URL: https://doi.org/10.1007/978-3-642-22673-1_18.

[14] Dominik Dietrich, Iain Whiteside, and David Aspinall. "Polar: A Framework for Proof Refactoring". In: *Logic for Programming, Artificial Intelligence, and Reasoning: 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*. Ed. by Ken McMillan, Aart Middeldorp, and Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 776–791. ISBN: 978-3-642-45221-5. DOI: 10.1007/978-3-642-45221-5_52. URL: https://doi.org/10.1007/978-3-642-45221-5_52.

[15] Serge Autexier and Dominik Dietrich. "A Tactic Language for Declarative Proofs". In: *Proceedings of the First International Conference on Interactive Theorem Proving*. ITP'10. Edinburgh, UK: Springer-Verlag, 2010, pp. 99–114. ISBN: 3-642-14051-3, 978-3-642-14051-8. DOI: 10.1007/978-3-642-14052-5_9. URL: http://dx.doi.org/10.1007/978-3-642-14052-5_9.

[16] Freek Wiedijk. "A Synthesis of the Procedural and Declarative Styles of Interactive Theorem Proving". In: *Logical Methods in Computer Science* 8.1 (2012). DOI: 10.2168/LMCS-8(1:30)2012. URL: https://doi.org/10.2168/LMCS-8(1:30)2012.

[17] Sascha Böhme and Tobias Nipkow. "Sledgehammer: Judgement Day". In: *Automated Reasoning: 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*. Ed. by Jürgen Giesl and Reiner Hähnle. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 107–121. ISBN: 978-3-642-14203-1. DOI: 10.1007/978-3-642-14203-1_9. URL: https://doi.org/10.1007/978-3-642-14203-1_9.

[18] *Matita*. URL: http://matita.cs.unibo.it/ (visited on 01/01/2018).

[19] Claudio Sacerdoti Coen. "Declarative Representation of Proof Terms". In: *Journal of Automated Reasoning* 44.1 (June 2009), p. 25. ISSN: 1573-0670. DOI: 10.1007/s10817-009-9136-7. URL: https://doi.org/10.1007/s10817-009-9136-7.

[20] William A. Howard. "The formulas-as-types notion of construction". In: *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Ed. by J. P. Seldin and J. R. Hindley. Academic Press, 1980, pp. 479–490.

[21] David Delahaye. "A Tactic Language for the System Coq". In: *Logic for Programming and Automated Reasoning: 7th International Conference, LPAR 2000 Reunion Island, France, November 6–10, 2000 Proceedings*. Ed. by Michel Parigot and Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 85–95. ISBN: 978-3-

540-44404-6. DOI: 10.1007/3-540-44404-1_7. URL: https://doi.org/10.1007/3-540-44404-1_7.

[22] M Giero, F Wiedijk, and Mariusz Giero. "MMode, a Mizar Mode for the proof assistant Coq". In: *Technical Report NIII-R0333* (Jan. 2003).

[23] *Mizar*. URL: http://www.mizar.org/ (visited on 01/01/2018).

[24] *Isabelle*. URL: https://www.cl.cam.ac.uk/research/hvg/Isabelle/ (visited on 01/01/2018).

[25] Markus Wenzel. "Isar — A Generic Interpretative Approach to Readable Formal Proof Documents". In: *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs' 99 Nice, France, September 14–17, 1999 Proceedings*. Ed. by Yves Bertot et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 167–183. ISBN: 978-3-540-48256-7. DOI: 10.1007/3-540-48256-3_12. URL: https://doi.org/10.1007/3-540-48256-3_12.

[26] John Harrison. "Proof Style". In: *Types for Proofs and Programs: International Workshop TYPES'96*. Ed. by Eduardo Giménex and Christine Pausin-Mohring. Vol. 1512. Lecture Notes in Computer Science. Aussois, France: Springer-Verlag, 1996, pp. 154–172.

[27] Donald E. Knuth. "Literate Programming". In: *Comput. J.* 27.2 (May 1984), pp. 97–111. ISSN: 0010-4620. DOI: 10.1093/comjnl/27.2.97. URL: http://dx.doi.org/10.1093/comjnl/27.2.97.

[28] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013. ISBN: 0262026651, 9780262026659.

[29] Benjamin C. Pierce et al. *Software Foundations*. Electronic textbook, 2017. URL: http://www.cis.upenn.edu/~bcpierce/sf.

[30] *ProofWeb*. URL: http://proofweb.cs.ru.nl/ (visited on 01/01/2018).

[31] Cezary Kaliszyk and Freek Wiedijk. "Merging Procedural and Declarative Proof". In: *Types for Proofs and Programs: International Conference, TYPES 2008 Torino, Italy, March 26-29, 2008 Revised Selected Papers*. Ed. by Stefano Berardi, Ferruccio Damiani, and Ugo de'Liguoro. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 203–219. ISBN: 978-3-642-02444-3. DOI: 10.1007/978-3-642-02444-3_13. URL: https://doi.org/10.1007/978-3-642-02444-3_13.

[32] Carst Tankink et al. "Proviola: A Tool for Proof Re-animation". In: *Intelligent Computer Mathematics: 10th International Conference, AISC 2010, 17th Symposium, Calculemus 2010, and 9th International Conference, MKM 2010, Paris, France, July 5-10, 2010. Proceedings*. Ed. by Serge Autexier et al. Berlin, Heidelberg: Springer Berlin

Heidelberg, 2010, pp. 440–454. ISBN: 978-3-642-14128-7. DOI: 10.1007/978-3-642-14128-7_37. URL: https://doi.org/10.1007/978-3-642-14128-7_37.

[33] Andrew Bedford. *Coqatoo: Generating Natural Language Versions of Coq Proofs*. 4th International Workshop on Coq for Programming Languages (CoqPL 2018). 2018. URL: https://arxiv.org/pdf/1712.03894.pdf.

付録: 拡張変換関数定義

全ての拡張を施した変換関数を以下に示す。

F' および F'' に、示した命題の名前を表す引数 *name* を追加した。_は名前が無いことを示す。また、IsProof(x) は項 x が証明項もしくは実在変数を含む項のとき true、それ以外のとき false を表す。

Pattern(*var*, *thesis*, n) は、変数 x の型の n 番目の `suppose` 文の一部を表す。基本としては、x の型の n 番目のコンストラクタおよびその引数を並べた形のパターンである。引数の型が x の型自身である時、命題 *thesis* 内の x をパターンで置換したものを、新たな名前と `and` と共に加える。例えば、Pattern(`x`, `x = x`, 2) は `S x' and H:S x' = S x'` である。

None は空の証明スクリプト、つまり何も出力しないことを示す。

本文で挙げた定義を以下に再掲する。

- Typeof(t) : 項 t の型
- Vars(t) : 項 t 中の自由変数の集合 (ただし要素が多い場合には$*$)
- Select(*subs*, `?`x) : 対応関係 *subs* において実在変数`?`x に対応する節点

F'((false, *tac*, *diff*, []), false, _) と F'((false, *tac*, *diff*, []), false, *name*) のように重複のある定義については、引数が具体化されている方 (例の場合では前者) を優先するものとする。

$F(t)$:= `proof.` $F'(t,$ true, _) `end proof.`

F'((false, *tac*, *diff*, []), true, *name*) := `thus Typeof(`*diff*`) by` Vars(*diff*) `using` *tac*.

F'((false, *tac*, *diff*, []), false, _) := `have` Typeof(*diff*) `by` Vars(*diff*) `using` *tac*.

F'((false, *tac*, *diff*, []), false, *name*) := `have` *name*:Typeof(*diff*) `by` Vars(*diff*) `using` *tac*.

(IsProof(d) = true)

F'((false, *tac*, *diff*, [g:x]), true, *name*) :=
$F'(x,$ false, _) `hence` Typeof(*diff*) `by` Vars(*diff*) `using` *tac*.

F'((false, *tac*, *diff*, [g:x]), false, _) :=

$F'(x,$ false, _) have Typeof(*diff*) by Vars(*diff*) using *tac*.

$F'((\text{false}, tac, diff, [g{:}x]), \text{false}, name) :=$
$F'(x,$ false, _) then *name*:Typeof(*diff*) by Vars(*diff*) using *tac*.

(IsProof(d) = false)

$F'((\text{false}, tac, diff, [g{:}(h,(t,d,s),r,n)]), \text{true}, name) :=$
thus Typeof(*diff*) by Vars(*diff*), d using *tac*.

$F'((\text{false}, tac, diff, [g{:}(h,(t,d,s),r,n)]), \text{false}, \_) :=$
have Typeof(*diff*) by Vars(*diff*), d using *tac*.

$F'((\text{false}, tac, diff, [g{:}(h,(t,d,s),r,n)]), \text{false}, name) :=$
have *name*:Typeof(*diff*) by Vars(*diff*), d using *tac*.

$PV(t, p) := p$ (IsProof(t) = true)

$PV(t, p) :=$ None (IsProof(t) = false)

$JV(t, n) := n$ (IsProof(t) = true)

$JV(t, n) := t$ (IsProof(t) = false)

$F'((\text{false}, tac, diff, [g_1{:}(t_1, d_1, sg_1), \ldots g_n{:}(t_n, d_n, sg_n)]), \text{true}, name) :=$
$PV(d_1, F'((t_1, d_1, sg_1), \text{true}, H_1))$
$\ldots$
$PV(d_n, F'((t_n, d_n, sg_n), \text{true}, H_n))$
thus Typeof(*diff*) by Vars(*diff*), $JV(d_1, H_1), \ldots, JV(d_n, H_n)$ using *tac*.

$F'((\text{false}, tac, diff, [g_1{:}(t_1, d_1, sg_1), \ldots g_n{:}(t_n, d_n, sg_n)]), \text{false}, \_) :=$
$PV(d_1, F'((t_1, d_1, sg_1), \text{true}, H_1))$
$\ldots$
$PV(d_n, F'((t_n, d_n, sg_n), \text{true}, H_n))$
have Typeof(*diff*) by Vars(*diff*), $JV(d_1, H_1), \ldots, JV(d_n, H_n)$ using *tac*.

$F'((\text{false}, tac, diff, [g_1{:}(t_1, d_1, sg_1), \ldots g_n{:}(t_n, d_n, sg_n)]), \text{true}, name) :=$
$PV(d_1, F'((t_1, d_1, sg_1), \text{true}, H_1))$
$\ldots$
$PV(d_n, F'((t_n, d_n, sg_n), \text{true}, H_n))$
`have` $name{:}\text{Typeof}(diff)$ `by` $\text{Vars}(diff), JV(d_1, H_1), \ldots, JV(d_n, H_n)$ `using` tac.

$F'((\text{true}, tac, diff, subs), goal, name) := F''(diff, subs, goal, name)$

$F''(diff, subs, goal, name) :=$ `define` H `as` $diff$. $(\text{IsProof}(diff) = \text{false})$

$(\text{IsProof}(diff) = \text{true})$

$F''(?x, subs, goal, name) := F'(\text{Select}(subs, x), goal, name)$

$F''(x, subs, \text{true}, name) :=$ `thus` $\text{Typeof}(x)$ `by` x.

$F''(x, subs, \text{false}, \_) :=$ `have` $\text{Typeof}(x)$ `by` x,

$F''(x, subs, \text{false}, name) :=$ `have` $name{:}\text{Typeof}(x)$ `by` x,

$(\text{IsInd}(t_0) = \text{true})$

$F''((\text{true}, tac, t_0 \ldots x, [g_1{:}(t_1, d_1, sg_1), \ldots g_n{:}(t_n, d_n, sg_n)]), \text{true}, name) :=$
`per induction on` x.
`suppose it is` $\text{Pattern}(x, \text{Typeof}(diff), 1)$.
$F'((t_1, d_1, sg_1), \text{true})$
$\ldots$
`suppose it is` $\text{Pattern}(x, \text{Typeof}(diff), (\text{n}))$.
$F'((t_n, d_n, sg_n), \text{true})$
`end induction.`

$F''((\text{true}, tac, t_0 \ldots x, [g_1{:}(t_1, d_1, sg_1), \ldots g_n{:}(t_n, d_n, sg_n)]), \text{false}, name) :=$
`claim` $\text{Typeof}(diff)$.
`per induction on` x.
`suppose it is` $\text{Pattern}(x, \text{Typeof}(diff), 1)$.
$F'((t_1, d_1, sg_1), \text{true})$

...
suppose it is Pattern(x, Typeof(*diff*), (n)).
$F'((t_n, d_n, sg_n)$, true)
end induction.
end claim.

(IsInd(t_0) = false)

$F''(t_0 \ldots t_n, subs, \text{true}, name) :=$
$PV(t_1, F''(t_0, subs, \text{true}, H_0))$
...
$PV(t_n, F''(t_n, subs, \text{true}, H_n))$
thus Typeof(*diff*) by $JV(t_1, H_1), \ldots, JV(t_n, H_n)$.

$F''(t_0 \ldots t_n, subs, \text{false}, \_) :=$
$PV(t_1, F''(t_0, subs, \text{true}, H_0))$
...
$PV(t_n, F''(t_n, subs, \text{true}, H_n))$
have Typeof(*diff*) by $JV(t_1, H_1), \ldots, JV(t_n, H_n)$.

$F''(t_0 \ldots t_n, subs, \text{false}, name) :=$
$PV(t_1, F''(t_0, subs, \text{true}, H_0))$
...
$PV(t_n, F''(t_n, subs, \text{true}, H_n))$
have *name*:Typeof(*diff*) by $JV(t_1, H_1), \ldots, JV(t_n, H_n)$.

$F''(\texttt{let } x := t_0 \texttt{ in } t_1, subs, \text{true}, name) := F''(t_0, subs, \text{true}, x)\ F''(t_1, subs, goal, name)$

$F''(\texttt{let } x := t_0 \texttt{ in } t_1, subs, \text{false}, \_) :=$ claim Typeoft_0. $F''(\texttt{let } x := t_0 \texttt{ in } t_1, subs, \text{true}, \_)$ end claim.

$F''(\texttt{let } x := t_0 \texttt{ in } t_1, subs, \text{false}, name) :=$ claim *name*:Typeoft_0. $F''(\texttt{let } x := t_0 \texttt{ in } t_1, subs, \text{true}, \_)$ end claim.

$F''(\texttt{fun } x \Rightarrow t, subs, \text{true}, name) := \texttt{let } x$:Typeof($x$). $F''(\text{t}, subs, \text{true}, name)$

F''(`fun` $x \Rightarrow t$, *subs*, false, *name*) :=
  `claim` *name*:Typeof(`fun` $x \Rightarrow t$). F''(`fun` $x \Rightarrow t$, *subs*, true, _) `end claim`.

$F''(t_1 \to t_2$, *subs*, true, *name*$)$:= `thus thesis by` $(t_1 \to t_2)$.

$F''(t_1 \to t_2$, *subs*, false, _$)$:= None

$F''(t_1 \to t_2$, *subs*, false, *name*$)$:= `define` *name* `as` $(t_1 \to t_2)$.

F''((`forall` x, t), *subs*, true, *name*) := `thus thesis by` (`forall` x, t).

F''((`forall` x, t), *subs*, false, _) := `define` H `as` (`forall` x, t).

F''((`forall` x, t), *subs*, false, *name*) := `define` *name* `as` (`forall` x, t).

F''(`match` t `with` $c_1\ t_{11} \ldots t_{1j} \Rightarrow t_1 \mid \ldots \mid c_i\ t_{i1} \ldots t_{ik} \Rightarrow t_i$ `end`, *subs*, true, *name*) :=
  `per cases on` t.
  `suppose it is` $c_1\ t_{11} \ldots t_{1j}$. $F''(t_1$, *subs*, true)
  ...
  `suppose it is` $c_i\ t_{i1} \ldots t_{ik}$. $F''(t_i$, *subs*, true)
  `end cases`.

F''(`match` t `with` $c_1\ t_{11} \ldots t_{1j} \Rightarrow t_1 \mid \ldots \mid c_i\ t_{i1} \ldots t_{ik} \Rightarrow t_i$ `end`, *subs*, false, _) :=
  `claim` Typeof(t_1).
  F''(`match` t `with` $c_1\ t_{11} \ldots t_{1j} \Rightarrow t_1 \mid \ldots \mid c_i\ \mathrm{t}_{i1} \ldots \mathrm{t}_{ik} \Rightarrow t_i$ `end`, *subs*, true)
  `end claim`.

F''(`match` t `with` $c_1\ t_{11} \ldots t_{1j} \Rightarrow t_1 \mid \ldots \mid c_i\ t_{i1} \ldots t_{ik} \Rightarrow t_i$ `end`, *subs*, false, *name*) :=
  `claim` *name*, Typeof(t_1).
  F''(`match` t `with` $c_1\ t_{11} \ldots t_{1j} \Rightarrow t_1 \mid \ldots \mid c_i\ \mathrm{t}_{i1} \ldots \mathrm{t}_{ik} \Rightarrow t_i$ `end`, *subs*, true)
  `end claim`.