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	早川　恵太	学籍番号	1631119
論 文 題 目	定理証明支援系 Coq における証明木を操作可能なインタフェースの設計および実装		

要　旨

一般的に，数学の定理や論理学における命題の真偽を示すには，正しく行われた証明が必要不可欠である．しかし，証明を人間の手で行う場合は，論理の飛躍や定理等の書き間違いなどのヒューマンエラーの危険性がある．この問題を解決するアプローチとして，定理証明支援系と呼ばれるシステムを利用した，計算機上で証明を記述する手法が存在する．Coq はその 1 つであり，過去に人の手では書ききることが困難な証明を，Coq を用いて行った研究も存在する．Coq ではユーザによって入力されるタクティクと呼ばれるコマンドによって，対話的に証明が行われる．しかし，Coq 標準のインタフェースではユーザはタクティクの羅列でしか証明を確認できず，行った証明の論理展開等の構造を確認しにくいという欠点がある．これを解決するため，本研究ではユーザにとってわかりやすい形で証明を表示しながら証明を進めることを可能とする Coq のユーザインタフェースの設計ならびに実装を行った．具体的には，関数型言語 OCaml を用いて Coq 内部で処理されている証明の情報を取得し，証明木という形に変換した．証明木は XML 形式のデータに変換し，インタフェース側に証明木の情報を伝達できるようにした．その後，Java で実装したインタフェースが証明木を受け取り，その情報を整理した後にわかりやすい形にして画面に表示することで，Coq 標準のインタフェースよりも証明の構造を把握しやすくした．さらに，本インタフェース上で Coq に対する操作を行えるようにすることで，本インタフェースだけでユーザの操作が完結するように実装した．また，幾つかの機能を実装することで，特に Coq や論理学の初学者にとって証明の構造や推移がわかりやすい形で証明を扱うことができるように設計した．これらにより，証明の視認性の向上や初心者及び初学者に対する証明操作の手助けという本研究の目的に寄与することができた．

平成 29 年度修士論文

定理証明支援系 Coq における
証明木を操作可能な
インタフェースの設計および実装

電気通信大学
情報理工学研究科
情報・ネットワーク工学専攻

学籍番号　　: 1631119
氏名　　: 早川 恵太
主任指導教員 : 中野 圭介 准教授
指導教員　　: 村尾 裕一 准教授
提出日　　: 2018 年 1 月 29 日

要旨

数学の定理や論理学における命題の真偽の証明を人間の手で行う場合は，常にヒューマンエラーの危険性がある．定理証明支援系は計算機上で証明を記述できるシステムであり，Coq は代表的な定理証明支援系の 1 つである．しかし Coq には，ユーザが行った証明の構造を確認しにくいという欠点がある．これを解決するため，本研究ではユーザにとってわかりやすい形で証明を表示しながら証明を進めることを可能とする Coq のユーザインタフェースの設計ならびに実装を行った．具体的には，関数型言語 OCaml を用いて Coq 内部で処理されている証明の情報を取得し，証明木という形で出力させる．その後，Java で実装したインタフェースが証明木を受け取り，わかりやすい形にして画面に表示する．さらに，本インタフェース上で Coq に対する操作を行えるようにすることで，本インタフェースだけでユーザの操作が完結するように実装した．また，幾つかの機能を実装することで，特に Coq や論理学の初学者にとって証明の構造や推移がわかりやすい形で証明を扱うことができ，証明の視認性の向上や，初心者及び初学者に対する証明操作の手助けという本研究の目的に寄与することができた．

目次

1　序論

本章では，本研究の背景と目的，本論文の構成について述べる．

1.1　背景

一般的に，数学における定理が成り立つことや論理学における命題の真偽を示すためには，論理の展開やその根拠を明確に記した証明が必要となる．証明は人間の手によって行われるのが一般的である．しかし，人間が証明を行う場合変数や利用する補題の書き間違いなどの人為的ミスや，理解不足や論理の飛躍といった証明を行う者の能力不足に起因する誤りが起こりうる．正しく行われなかった証明には意味がないため，このような問題は極力回避すべきといえる．

確実に正しいといえる証明を実現するための方法として，人の手だけによらない環境を用いて証明を行うというアプローチがある．そのために開発されたのが，定理証明支援系と呼ばれる計算機によって証明を行うことを支援するシステムである．これによって，前述したヒューマンエラーの可能性を排除した証明が実現される．また，定理証明支援系ではプログラムの性質そのものの証明も可能である．支援系を用いて記述したプログラムが満たす性質の証明をその正当性を保ったまま他の言語へ抽出することで，その証明された性質を満たすプログラムとして利用することもできる．そのため，人の手では書ききることが難しい複雑な証明を正しさを保証しつつ行う場面だけでなく，プログラム開発においても定理証明支援系の活躍の場は存在している．Coq [1] は定理証明支援系のひとつで，これまで述べてきた特徴を備えている．

しかし，Coq にはユーザが実際に証明を行う上で証明の構造を把握しにくいという欠点がある．Coq において，ユーザはタクティクと呼ばれる証明を進めるためのコマンドを入力することで対話的に証明を進めていく．このとき，Coq で提供されているインタフェースでは，ユーザはタクティクの実行前後の状況を表示させることはできるが，証明全体の構造をわかりやすい形で表示する機能は提供されていない．そのため，ユーザは今自分が何を証明しようとしているのか，現状が証明全体の中のどの段階に位置しているのかを把握することが容易でない．特に Coq 初学者において，証明の構造を把握することができれば，適用したタクティクの効果や自分の辿ってきた証明の道筋の理解がしやすくなる効果が期待できる．

1.2　目的と方針

本研究では，証明の視認性の向上並びにインタフェース上での直感的な操作によって，特に初学者において定理証明支援系 Coq を利用しやすくすることを目的とする．

本研究の最終目標は，証明の構造を従来よりもわかりやすく表示し，かつ直感的な操作によっ

て視認性の向上や証明の補助を行える Coq の新しいユーザインタフェースを設計及び実装することである．これにより，特に Coq 初学者や論理学の初学者にとって，Coq における証明のやり方や証明の構造を理解しやすくなると期待できる．

実現に際しては，証明を木構造の一種である証明木を用いて表現し，ユーザにとって見やすく表示する．また，表示された証明木をユーザが操作することで，構造の把握や証明の補助ができるようにする．実装方法としては，Coq のプラグインとユーザインタフェースの 2 つに分離して実装する．本来の Coq に証明木を取り出してインタフェースに情報を伝達する機能を付加するプラグインを Coq に導入し，本研究で実装するインタフェースとの連携を行えるようにする．またインタフェース上で Coq にタクティクやその他の入力を行えるようにすることで，ユーザはインタフェースの操作のみで証明を進められるように設計する．Coq とインタフェースの間のデータの伝達には，マークアップ言語 XML を用いる．

1.3 本論文の構成

2 章では，本研究で扱う定理証明支援系である Coq について，概要と利用例，また拡張の方法について述べる．3 章では，本研究において実装するプラグインやインタフェースの設計方針や構造について述べる．4 章では，本研究において実装するプラグイン並びにインタフェースについて，それぞれの具体的な実装内容について述べる．5 章では，実装したインタフェースで証明を行った際の様子と，実装した機能についての考察や評価について述べる．6 章では，本研究に関連した既存研究について，本研究との共通点や差異を考察する．最後に 7 章では，本研究での成果物のまとめと，今後の課題や展望について述べる．

2　定理証明支援系 Coq

Coq は，フランス国立情報学自動制御研究所 (INRIA) によって開発された，定理証明支援系である．Coq は，2013 年に ACM Software System Award 及び ACM SIGPLAN Programming Languages Software Award を受賞した実績のあるソフトウェアである．Coq を用いて証明された定理の一例として，四色定理 [2] が挙げられる．また，Coq によってその性質や挙動が正しいことの証明を行いながら開発されたプログラムの一例として，C 言語コンパイラである CompCert [3] が存在する．

Coq などの定理証明支援系による証明の主な利点として，計算機の支援によって，人為的な誤りのない証明を行うことができるという点が挙げられる．Coq を利用する主な目的は，数学の定理証明やプログラムの安全性の証明，プログラムが満たすべき挙動の定義などである．

本章では，Coq を用いた証明の方法と Coq のプラグインによる拡張の方法，本研究で証明を扱う際に利用する証明木，そして Coq の証明とプログラムを関連付けるカリー・ハワード同型対応について述べる．

2.1　Coq による証明

Coq では，タクティクと呼ばれる証明を進めるためのコマンドを用いて，1 ステップずつ対話的に証明を進める．以下に挙げる例では，証明行うにあたって，Coq で提供される標準的な CUI である coqtop を使用した．簡単な例として，以下の性質を証明する．

$$\forall A\, \forall B, (A \to B) \to A \to A \wedge B.$$

これは，任意の A，B において，$(A \to B)$，A が成り立つという仮定のもとで，$A \wedge B$ が成り立つことを導くことができるという意味の命題である．この命題を実際に証明するには，Coq に対して以下のように入力する．ここで，`"Coq <"`は Coq のコマンドプロンプトであり，`"<"`以降が，ユーザによる入力である．

```
Coq< Theorem sample: forall A B:Prop, (A->B)->A->A/\B.
```

`Theorem sample` のように記述することで証明の名前を設定し，その後ろに続けて証明すべき命題を記述する．ここで `Prop` は，命題を表す型である．上のように入力すると，Coq は次のように現在の証明の様子を出力する．

```
Coq < Theorem sample: forall A B:Prop, (A->B)->A->A/\B.
1 subgoal

  ============================
  forall A B : Prop, (A -> B) -> A -> A /\ B
```

二重線の上部が仮定が表示される領域であり，下部が示すべき結論が表示される領域である．また，subgoal は証明しなければならない命題を表す．この例では，現在結論部分には 1 つしか命題がないのでサブゴールはひとつであり，1 subgoal と表示される．

これ以降に示す入力例においては，"Coq"の部分が現在証明している定理等の名前になる．また，これ以降のタクティクによる証明の進行のたびに Coq は現在の証明の様子を出力する．

ここから証明を始めるには，intros タクティクを使用する．このタクティクによって，「A が真という仮定のもとで B が証明されれば，$A \to B$ が言える」という含意の導出規則を適用することができる．この例では，「$A \to B$，A が真という仮定のもとで $A \wedge B$ が証明できれば，$(A \to B) \to A \to A \wedge B$ が言える」という形になる．

```
sample < intros.
1 subgoal

  A, B : Prop
  H : A -> B
  H0 : A
  ============================
  A /\ B
```

この結果，次に証明すべきは $A \wedge B$ であるという形になった．H，H0 は，導出によって得られた仮定に Coq が参照のためにつけた名前である．ここで証明すべき結論を見ると，これを証明するには A と B がどちらも真であるという必要があることがわかる．この場合は，タクティク split を適用する．その結果，証明すべき結論は，A を証明する方と B を証明する方のふたつに分かれる．

```
sample < split.
2 subgoals

  A, B : Prop
  H : A -> B
  H0 : A
  ============================
  A

subgoal 2 is:
 B
```

この局面においては，$A \to B$ と A の仮定のもとで A を証明する必要がある．仮定 $H0$ に A そのものがあるため，この仮定を適用することで，証明を進めることができる．このような場合は，タクティク `apply H0` を適用する．

```
sample < apply H0.
1 subgoal

  A, B : Prop
  H : A -> B
  H0 : A
  ============================
  B
```

今度は，$A \to B$ と A の仮定のもとで B を証明する必要がある．ここで，仮定 `H : A -> B` があることから，A が真であると言えれば B も真といえることがわかる．したがって，タクティク `apply H` で仮定を適用し，そのように結論を変形させる．

```
sample < apply H.
1 subgoal

  A, B : Prop
  H : A -> B
  H0 : A
  ============================
  A
```

最後に，さきほどと同じように apply H0 とすれば，このサブゴールも証明が完了する．

```
sample < apply H0.
No more subgoals.
```

この証明においては他に証明すべきサブゴールは存在しないため，これで証明終了となる．証明を終了するには Qed. と入力する．

```
sample < Qed.
(intros **).
split.
 (apply H0).

 (apply H).
 (apply H0).

Qed.
sample is defined
```

以上が簡単な証明の流れとなる．

また，Coq には auto タクティクが存在する．これを用いると，intros や apply 等のタクティクをある程度自動的に適用させることができる．この例では，はじめの intros の代わりに auto を利用するだけで証明を完了させることも可能である．

以上が Coq を用いた証明の流れの例である．

ここで，上の例を Coq のデフォルトの GUI である CoqIDE で行った画面を図 2.1 に示す．証明中では，例の中に現れた仮定と結論の表示と同様の表示が画面右上の枠内にされるため，今その段階でどのような操作を行えばよいか把握することはできる．しかし，証明を終えたのちに，証明全体がどのようにされていったのかをこの画面だけから推し量るのは容易ではない．したがって，このような表示ではなくもっと理解しやすい表示を行って証明を扱えるようにすることが本研究の目的となっている．

2.2 プラグインによる拡張

Coq はプラグインを用いて拡張することができる．拡張内容としては，Coq に入力するコマンドの追加や，処理の変更などが挙げられる．プラグインの著名な例として，SSReflect [4] という，タクティクの簡略化や軽い証明を自動的に行えるようにするなどの拡張が実現されている．前節で紹介した Coq による四色定理の証明も，SSReflect の拡張を利用して証明が行われている．また，Coq は関数型言語 OCaml [5] で実装されているため，Coq のプラグインについても OCaml 及びその関連ライブラリを用いて実装を行う．

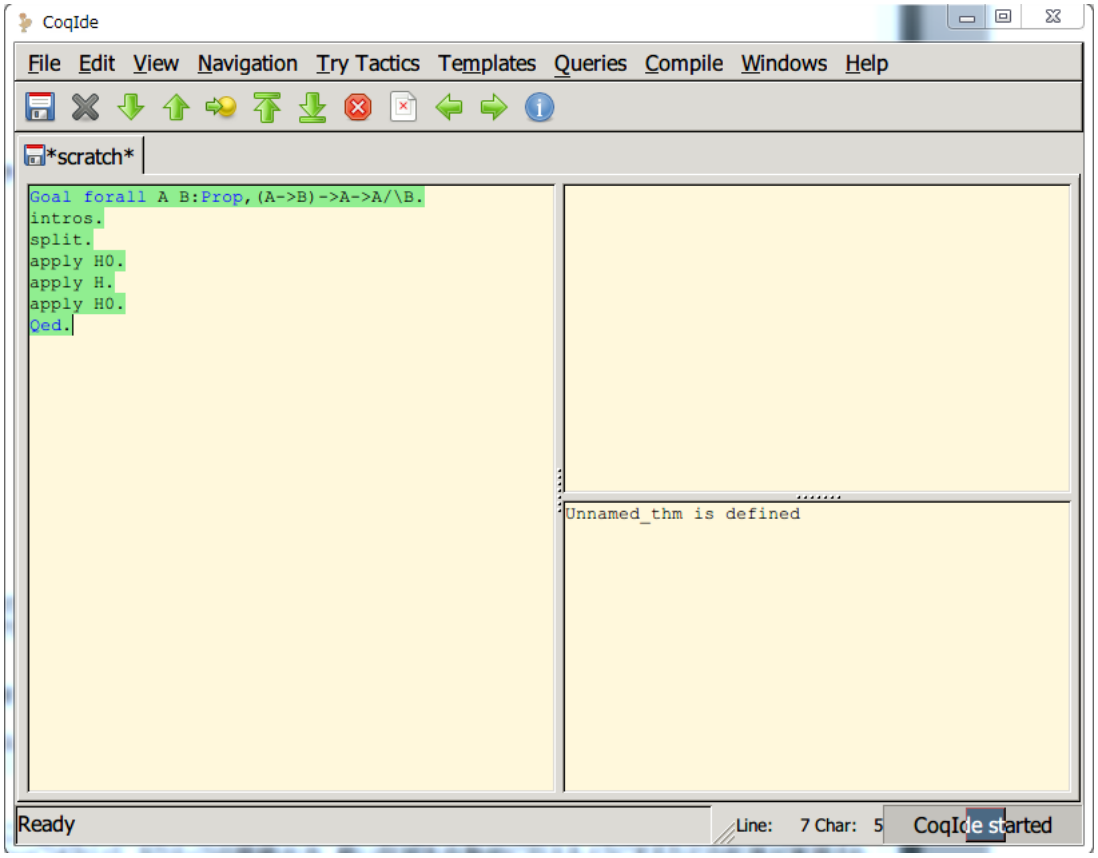

図 2.1　CoqIDE で $\forall A\, \forall B, (A \to B) \to A \to A \wedge B.$ の証明を行った例

Coq にプラグインを導入する際には，最低限の構成として以下のパッケージが導入されている必要がある．

- OCaml 処理系
  OCaml の開発環境
- CamlP4
  OCaml の文法を拡張するパッケージ

また，以下の 3 つのファイルを作成する必要がある．

- 文法の拡張と，その処理を記述する ml4 ファイル
- 使用するファイルを明示するライブラリである mllib ファイル
- モジュールとして読み込む宣言を記述する v ファイル

これらのファイルのそれぞれについて説明する．まず ml4 ファイルには実際に行わせる処理を記述する．例として，本研究で設計したプラグインの構成要素である showp.ml4 の内容を以下に示す．

```
(*i camlp4deps: "grammar/grammar.cma" i*)

DECLARE PLUGIN "showp"
VERNAC COMMAND EXTEND Showp CLASSIFIED AS QUERY

    ["Showp"] ->
    [Body_showp.showp_fun()]
END
```

この記述によって，showp という名前でコマンドを定義することを宣言する．次いで，[ ] 内に実際に入力させるコマンド名を記述する．今回は，Showp. と Coqtop に入力すれば，定義した処理が実行される．続いて，->以下の [ ] に，コマンドが入力された際に実行される処理を OCaml の文法に則って記述する．コマンドによる処理の最終的な返り値は unit 型，すなわち出力である必要がある．処理が特定の文字の出力などの簡単なものであれば，この欄に直接記述してもよいが，実際にはもっと複雑かつ長い処理を行うことがほとんどである．そのため，OCaml ソースコードである ml ファイルに別途処理を記述し，そこで定義した関数を呼び出して使用することになる．

次に，コンパイル時に使用する ml ファイルと ml4 ファイルの名前を，mllib ファイルに記述する．これにより，コンパイル時に OCaml コンパイラが記述したソースファイルを認識し，そこで定義した関数が利用できるようになる．また v ファイルには，定義したモジュールを Coq

に読み込ませるための記述を行う．

最後に，これらのファイルを実際にプラグインとして機能させる方法について述べる．同一ディレクトリ内に src，theories というディレクトリを作成し，src 以下に ml4 ファイルと mllib ファイル，処理内容を記述した ml ファイルを格納する．theories 以下には v ファイルを格納する．その後に，Makefile を作成するが，このとき `coq_makefile` というコマンドが使用できる．これにより，Make というファイルを事前に作成しておくことで Makefile を自動生成できる．

Make には使用するソースファイルのパスや，導入する Coq などの情報を記述しておく．これにより，使用している環境に合致した Makefile が生成される．最後に make し，make install を行えばプラグインの導入は完了する．

実際に Coq での証明のなかで，定義したコマンド等を利用したい場合は，モジュールとしてプラグインを読み込む必要がある．具体的には，今回の例では以下のように入力する．

```
Coq < Require Import Coq_custom.Showp.
```

上記の入力が正しく受け付けられれば，プラグインの導入が完了する．この状態で証明中に Coq に対して `Showp.` と入力すると，定義した処理が実行される．本研究では，このコマンドが入力されるとその時点での証明の構造を整形し外部ファイルに出力するという処理がなされる．処理の内容については，4 章にて後述する．以上が，本研究で行うプラグイン作成の手順である．

2.3 証明木

本節では，本研究での拡張の実装にあたり利用した証明木の概要と構造を述べる．証明木とは，論理学や自然言語学における推論の様子を，証明したい命題の上部にそれを証明するために必要な補題を記述するというシンプルな木構造で表したものである．本研究では，仮定と結論を分けて管理することが比較的容易であり，また木構造であるため XML 形式への変換などがスムーズに行えることから，この構造を利用して証明を扱うこととした．

2.3.1 証明木の構造

例として，命題 $\forall A \forall B, (A \to B) \to A \to A \wedge B$ の証明木を図 2.2 に示す．

以下，証明木の構造を説明する．各行について，$\vdash$の左側に変数と型の関係の集合である環境が配置される．右側にはその環境のもとで導かれるべき結論が配置される．まず，最終的に証明したい命題が最下部に位置する．その上部には，この命題を証明するための補題，すなわち仮定 $A \to B$ のもとで $A \to A \wedge B$ が成り立つという命題が入る．上部へ一段上がると，さらに仮定をとって，$A \wedge B$ を証明する形をとる．さらにその上部には，$A \wedge B$ が成り立つための補題 2 つ

$$\dfrac{\dfrac{\dfrac{\dfrac{}{H_0 : A \to B, H_1 : A \vdash A} \quad \dfrac{}{H_0 : A \to B, H_1 : A \vdash A \to B} \quad \dfrac{}{H_0 : A \to B, H_1 : A \vdash A}}{H_0 : A \to B, H_1 : A \vdash B}}{H_0 : A \to B, H_1 : A \vdash A \wedge B}}{H_0 : A \to B \vdash A \to A \wedge B}}{\vdash (A \to B) \to A \to A \wedge B}$$

図 2.2　$\forall A\, \forall B, (A \to B) \to A \to A \wedge B$ の導出を示す証明木

が横に並んで入る．それぞれ，仮定 $A \to B$ と A のもとで A が成り立つという命題，B が成り立つという命題である．B が結論となる段の上には，同様に B を証明するための補題が並ぶ．これらについては仮定部分に結論そのものが含まれるため，これらの上部に新たに補題が入ることはない．すべての枝分かれの最上部の命題すべての証明が終了した時点で，全体の証明が完了する．

ここで，証明木の構造を利用する利点について述べる．証明木には，前述した利点の他に証明の分岐が直観的に把握できるという長所がある．例えば場合分けなどで証明が枝分かれするような場面を考える．手書きや Coq での証明では分岐した証明を横に並べて書くことはできず，縦に並べて書き続けなければならない．そのため，似たような方法で証明を行うような場合でも分岐同士の対応関係を把握しにくいという問題が発生する．しかし，証明木を用いて証明の表示を行えば，類似する証明の構造がそのまま類似して横に並んで表示されるため，前者と比べて証明の構造の理解がしやすくなる．以上が他の証明の表示方法に対する証明木の利点である．

2.4　カリー・ハワード同型対応

ここで，Coq において証明の構造を扱う，ひいては木構造を生成する上で重要な要素となるカリー・ハワード同型対応 [6] について説明する．この理論は，プログラムが証明に，また型が命題にそれぞれ対応するというものである．Coq における証明はこの考え方に則って構成されている．例えば証明内で $A \to B$ という命題があった場合，それは $A \to B$ という関数の型に対応する．Coq においては，完了した証明を関数の形で表示させるタクティクが存在する．例として，さきほど説明した $\forall A\, \forall B, (A \to B) \to A \to A \wedge B$ の証明を関数の形で表示させた例を以下に示す．

```
(fun (A B : Prop) (H : A -> B) (H0 : A) => conj H0 (H H0))
```

このように，実際に Coq の証明は内部では例のような関数適用の形で表現されている．この証明において最終的に証明するのは，命題 $\forall A\, \forall B, (A \to B) \to A \to A \wedge B$ である．カリー・ハ

ワード同型対応に照らし合わせると，この証明は，$A \wedge B$ という型を返すような関数とみなすことができる．したがって，この証明を表す関数は，A 型である `H0` と，$A \to B$ 型の関数 `H` に A 型の `H0` を適用して得られる B 型との conjunction，すなわち $A \wedge B$ を返すものになっている．このように，命題と型は対応する関係となっている．

また，証明の内部構造は，証明項と呼ばれるラムダ式や関数適用といった証明の要素で構成されている．Coq における証明は，この証明項をタクティクによって変化させていくことに相当する．証明項の中に現れる命題も，プログラム中では型として扱われる．そのため，本研究で実装した証明項を操作させるような関数内では木構造を生成する際に必要となる情報である命題を型として操作している．

3　設計

本章では，本研究で実装するユーザインタフェース及び Coq を含むシステムの概観と，Coq-GUI 間の連携方法について述べる．

3.1　設計方針

本研究の目標は，1 章で述べたように Coq における証明をわかりやすく表示し，また証明を行いやすくするためのインタフェースの実装である．その実現に際して，以下のような設計方針を定めた．

- 想定するユーザは，Coq による証明に慣れていないような初学者とする．
- ユーザは，インタフェースのみの操作で Coq を利用した証明を行えるようにする．
- Coq における証明は，証明木の構造を利用して証明項を管理しやすい形で扱う．
- 木のノードに対する操作で証明の補助を行えるようにする．
- Coq での処理とインタフェースでの処理は完全に分離する．

以下，それぞれの方針について説明する．

まず，本インタフェースの対象とする初学者とは，学校で論理学や数学の証明を手書きで行ったことはあるが，適用した定理がどのように式を書き換えているかなどの具体的な論理展開やしくみについて詳しくないような者を指す．手書きの証明から論理の流れや仮定や定理などの適用の様子を追いきるのは現実的ではない．したがって，証明についてその構造や論理展開を把握したい状況においては，定理証明支援系で証明を記述し，証明の具体的な構造を解析することに大きな意義があると考えられる．証明木の形で証明の視認性を上げ，証明のステップが進むたびに，変数や命題の変遷など何が起こっているのかを追いやすくするのが，本研究の具体的な目標である．

次に，ユーザが操作するのは本研究で実装するユーザインタフェースのみで済むように，全体の設計を行う．証明を進めるたびにインタフェースと coqtop の画面を行き来するのは，ユーザにとっては行う操作の数が増加することになり，好ましくないためである．証明にかかわるすべての操作をインタフェース側で代替することで，操作によるユーザのストレスを少しでも緩和するのが，この方針を掲げる理由である．

続いて，Coq による証明を証明木の形に変換する理由について述べる．もともと，Coq での証明は証明項という形で表現されている（詳細は次節で述べる）．証明の情報を取得するだけならば，証明項を扱うだけで十分である．しかし証明項だけでは，証明のステップごとの仮定の内容やその時点でのサブゴールの形などの情報を得ることができない．本研究においては視認性向

上のためにステップごとの情報を表示する必要があるため，証明項を証明木の構造に変換することで，情報の不足がないように管理することとした．

次に，インタフェース上に表示した証明木に対してユーザが操作を行うことで，証明を見やすく，また行いやすくする機能を導入する．具体的には，枝分かれがあるような証明に対してすでに証明が完了した部分を折りたたむことで現在取り組んでいる部分を注視しやすくすることや，証明中の任意の状態での仮定やサブゴールの様子を簡単に参照できるようにすることが該当する．これらによって，Coq の扱いに慣れていない初学者にとっての証明の視認性の向上を図る．

最後に，Coq 側での証明項から証明木を生成する一連の処理とインタフェース側で証明木を表示したり木に対して操作を行う処理は完全に分離して実装する．OCaml にはグラフィカルユーザインタフェースの作成が可能なプラグインが存在しており，それを用いて OCaml でインタフェースを実装すれば，証明木をさらに変換して別のプログラミング言語で実装されたインタフェースに送信する必要はなくなる．また，OCaml のデータとして保持されている証明項をそのまま利用できるため，各機能の実装が比較的容易になるというメリットがある．しかし，OCaml でインタフェースを実装した場合，証明の処理と表示の処理の依存が深くなってしまい，拡張性に乏しくなるという欠点がある．そのため，広く利用されている Java などのプログラミング言語を利用してインタフェースの部分を独立して実装し，特定のプロトコルを利用してデータの送受信を管理することで，Coq 本来の信頼性を確保した証明とインタフェースの拡張性を両立することを図る．加えて，Coq とインタフェースを完全に分離することで，Coq のバージョンに依存しない実装が可能となる．Coq での証明の内部表現 (証明項) の構造が変化しない限り，本システムを利用することが可能である．なお，証明木のデータ送信のプロトコルとしては XML というマークアップ言語を利用する．その詳細は次章で述べる．

3.2 システム全体の概観

本研究で実装するシステム全体の模式図を図 3.1 に示す．本研究で実際に設計および実装するのは，Coq プラグインと GUI の部分となる．本システムで想定する処理の流れは以下のようになる．図中の 1〜8 の箇所が，以下の 1〜8 の項目に該当する部分である．

1. ユーザが，証明を行いたい命題や定理を入力する．
2. Coq 処理系にその入力を読み込ませて，証明を行う．
3. その時点での証明項をプラグインを用いて証明木に変換する．
4. 変換された証明木を XML 形式に変換し，外部ファイルに出力する．
5. GUI が XML ファイルを読み取り，証明木に復元する．
6. 証明木を表示する．
7. ユーザが表示された証明木を見て，証明を進めるタクティクを入力する．
8. GUI が受け取った入力を Coq に通達し，新たな証明項を生成する．

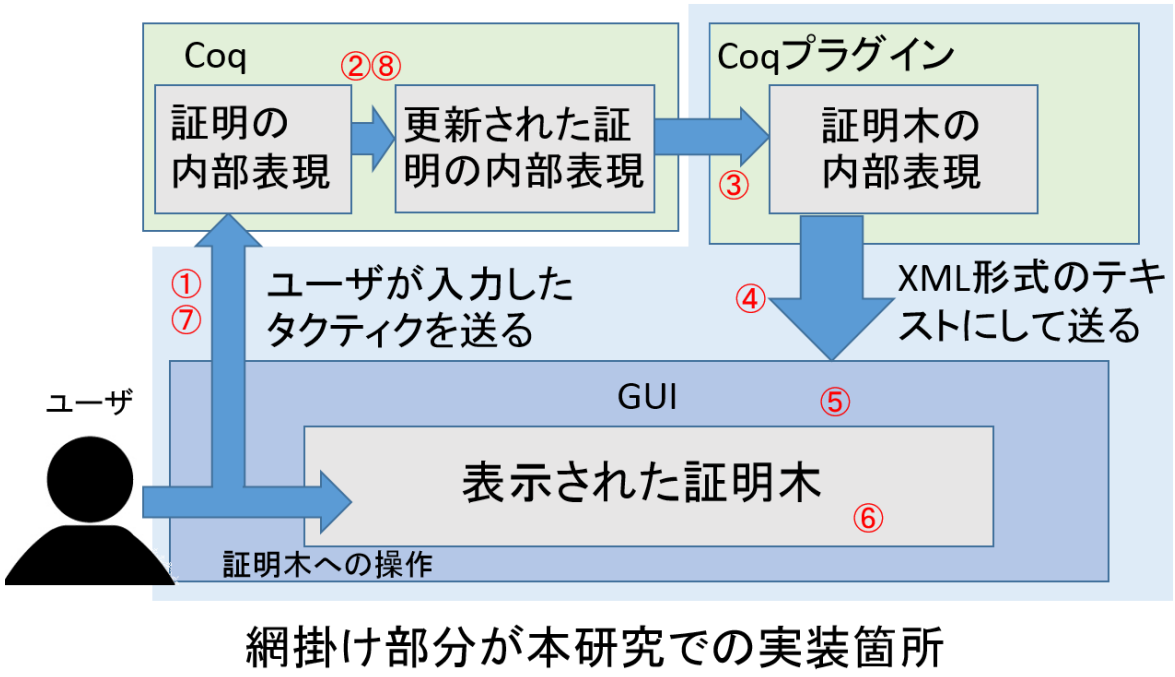

図 3.1　システム全体の模式図

9. 3 に戻る

以上のループを繰り返すことで，証明を進めていく．また，Coq プラグインの実装は OCaml で行い，GUI の実装は Java の統合開発環境である Eclipse [7] を用いて行う．これらは異なるプログラミング言語で実装するため，そのままでは Coq と GUI 間でのやり取りを行うことはできない．そこで，Coq から GUI へ情報を伝達する手段として XML [8] を採用した．XML は比較的軽量なマークアップ言語である．HTML タグのように`<foo>`，`</foo>`で囲った内部に要素を記述していくが，このとき入れ子の構造をとることも可能である．また，配列や構造体などの特殊な構造でも囲いを並べるだけで取り扱えるため，表現できるデータの幅が広いという特徴を持っている．そのため，木構造を持つデータの表現も容易に行うことができる．本研究においては，証明木のデータを GUI に送る際のプロトコルとして利用している．

3.3　GUI の満たすべき性質

GUI を設計するにあたって，証明の視認性を向上させるために，以下の 2 点に注意して設計を行った．

- 仮定と結論は分けて表示する
- 未解決のゴールについて，そこに収まるべき命題を表示する

まず 1 つ目の項目について説明する．GUI 上の証明木の各ノードには，そのとき証明すべきサブゴールが表示されるように設計する．そのとき，例えば

```
1 subgoal
  A, B : Prop
  H : A -> B
  H0 : A
  ============================
  A /\ B
```

という状態を表すノードの表示は，仮定と結論を分けて，専用のスペースを設けて

```
  A, B : Prop
  H : A -> B
  H0 : A
 --------------
  A /\ B
```

と表示させ，表示した証明木のノードには，$A \wedge B$ とだけ表示させるようにする．これは，仮定と結論を分けずに

```
  A,B: Prop, H: A -> B, H0: A |-  A /\ B
```

と表示させてしまうと，ノード 1 つ 1 つの表示が長くなってしまい，全体として見にくくなることが予想されるからである．例は単純なのでこの程度の行数の表示となる．しかし，より長い証明を扱う場合などは当然仮定の数も大きくなるため，木全体の表示が横に伸びていってしまい，これでは「証明の視認性を向上させる」という目標にそぐわない．したがって，本研究の実装では仮定と結論は分割て表示させ，証明木としての視認性を損なわないような表示となるように設計している．

続いて，2 つ目の項目について述べる．前章の証明木の項でも説明したとおり，証明を完遂させるにはすべての未解決の項について証明を完了させる必要がある．インタフェース上の証明木にも未解決の項は現れるが，その際にその項が本来持つべき型，すなわち証明すべき命題が表示されていないと，特に初学者にとっては次に何をすればよいのかわからなくなってしまう可能性がある．そのため，未解決の項でも空欄にしておくのではなく，この型を持つという結果になるように証明を進めるという指標として，対応する命題を表示させるようにする．例えば，未解決の項があり，本来ここには $A \to B$ という型が入るという場合，空欄ではなく

$$(A \to B)(notyetfinished)$$

と表示するようにする．ユーザはこの表示によって $A \to B$ という命題を証明するという状況を容易に把握できるようになる．これによって，未解決の項がある状態でも次に行うべき操作の予測を立てやすくなり，結果として初学者の補助となることが期待できる．

4　実装

本章では，本研究において定義した関数や作成した構造などについて，Coq プラグインの実装と GUI の実装に分けて具体的な内容を述べる．

4.1　Coq プラグイン

本研究で実装する Coq プラグインの機能は，Coq 内部で管理されている証明項を参照し，それに対応する証明木を生成し，さらに GUI での表示に必要な情報を付与し XML 形式のファイルとして外部に出力することである．具体的な内部処理のプロセスは以下のようになる．

1. 証明項のデータを取得する．
2. 証明項の内部の de Bruijn Index [9] を変数の形に変換する．
3. 変換された証明項から，対応する木構造を生成する．
4. 木構造を，XML 形式に変換し，外部に出力する．

次節以降で，上記のプロセスのそれぞれについての具体的な処理を説明する．

4.1.1　証明項と証明木の定義

それぞれの過程の具体的な説明に先立ち，証明項の具体的な構造と本研究にて定義した証明木の構造の説明を行う．まず，Coq で定義されている証明項は複数のデータ構成子を持ち，その内容は以下の通りである．

- `Rel i` ・・・de Bruijn Index を表す．引数が Index の番号を示す．
- `Var v` ・・・変数を表す．引数が変数名を示す．
- `Meta v`・・・メタ変数を表す．引数が変数名を示す．
- `Evar (k, a)`・・・実在変数を表す．実在変数は，未完成の証明項を示す変数である．
- `Sort v`・・・証明項の種類を表す．`Set`(関数)，`Type`(型)，`Prop`(命題) がある．
- `Cast (v, ck, tp)`・・・キャストを表す．変数 `v` が `tp` 型にキャストされることを示す．`ck` はキャストの種類を示す．
- `Prod (v, t1, t2)`・・・直積型を表す．名前が `v`，型が t1->t2 という形に対応する．
- `Lambda (v, t1, t2)`・・・ラムダ式を表す．λ `v:t1, t2` という式に対応する．
- `LetIn(x, c1, t1, c2)`・・・Let in 式を表す．`Let x = c1:t1 in c2` という構造を示す．
- `App(c, ca)`・・・関数適用を表す．`c` が関数，`ca` が引数の配列である．Coq 内部では ∧ や ¬ の表現が App で行われる．

- Const v・・・定数を表す.
- Ind pi・・・Inductive で定義した規則等を表す.
- Construct pc・・・Ind に何番目のものかという情報を付加したものである.
- Case (ci, t1, t2, ca)・・・Case 文を表す．実態はパターンマッチであり，t1 が返り値の型，t2 がパターンマッチ対象の型，ca がこの項の子となる証明項の配列である.
- Fix (a, (n, tp, t1))・・・再帰関数を表す.
- CoFix (a, (n, tp, t1))・・・余再帰関数を表す.
- Proj (p, v)・・・レコード型を定義した際にできる射影関数を表す.

ここで，本研究で扱うデータ構成子の範囲について述べる．本研究はユーザを Coq や論理学の初学者と想定している．そのため，単純な証明や基本的な構造において証明項中に表れないデータ構成子については，以降の説明では割愛する．Fix や CoFix などがそれに相当する.

次に，証明項を proofterm として，これらの構造の BNF 表記を用いた記述を図 4.1 に示す．なお図中では本研究で実装した変換関数に関わるデータ構成子についてのみ記述し，他の構成子は割愛した.

ここで，<dBIndex>は de Bruijn Index 記法で表された番号，<Id>は変数名，<key>は Evar において本来型としてあるべき変数の名前を示し，<Anonymous>は名前を持たない変数，すなわち関数適用を表す．また<CaseInfo>はどの変数におけるパターンマッチであるかなどの情報を表す．いずれも本研究で実装した変換の範囲ではこれ以上詳しく参照することがないため，終端記号とした.

次に，本研究で定義した証明木の構造を BNF 表記で記述したものを図 4.2 に示す．ここで，

```
<proofterm>::='Rel' <dBIndex>|
              'Var' {<Id>|<Anonymous>}|
              'Evar' <key> {<proofterm>}*|
              'Prod' {<Id>|<Anonymous>}<proofterm><proofterm>|
              'Lambda' {<Id>|<Anonymous>}<type><proofterm>|
              'LetIn' {<Id>|<Anonymous>}<proofterm><type><proofterm>|
              'App' <proofterm>{<proofterm>}*
              'Case' <CaseInfo><proofterm><proofterm>{<proofterm>}*
             . . .
<type>::=<proofterm>
```

図 4.1　証明項の BNF 表記 (抜粋)

```
<prooftree>::= 'Not_yet' <env><Prop><type>|
               'Node' <env><Prop>{<prooftree>}*
<env>::= {<Prop><type>}
```

図 4.2　証明木の BNF 表記

Not_yet は未解決の証明，Node は証明における 1 つの状態を表す．<env>は環境で，そのノードにおける仮定や変数を表す．具体的には，その仮定や変数の名前とその型のタプルを変数の数だけ保持する．<Prop>はそのノードにおける命題，すなわち型を表す．また，Node は子を持たない場合，子として証明木を 1 つないし複数持つ場合がある．子を持たない場合，そのノードが証明木における葉となる．

4.1.2　変換の定式化

Coq プラグインに実装する証明項から証明木への変換は，証明項の論理的な正しさを崩すことなく対応する証明木へ変換する必要がある．本節では実装する変換の具体的な内容について述べる．

まず，証明項におけるそれぞれの構成子について，証明木への変換の式を図 4.3 に示す．なお，Rel は後述する de Bruijn Index の変換関数によって Var に置換されるため，証明木への変換においては表れない．また，Prod は証明項中の型に相当する部分にのみ現れるため，証明木への変換は定義していない．以降，それぞれのデータ構成子について，より具体的な変換の内訳を述べる．

Var

Var で表される証明項は変数を表す．未解決な証明ではないため，Node への変換となる．後述する関数群を用いてこの項の型と環境を取得し，Node の要素として保持させる．また，変数単体の項にそれ以降の証明が続くことはないため，この項に対応する証明木には子となる証明木は存在しない．

Evar

Evar は未解決の証明に相当するため，変換先は Not_yet である．3.3 節で述べたように，未解決の証明について対応すべき型を明示する必要があるため，<key>の示す変数をこのノードの型とし Not_yet の要素として保持させる．<proofterm>* の部分は，この証明項を導く関数適用にあたる情報が格納されているため，証明木に変換する際には不要となる．

```
'Var' <Id> ->
  'Node' <env><Prop>
'Evar' <key>{<proofterm>}* ->
  'Not_yet' <env><Prop><type>
'Lambda' {<Id>|<Anonymous>}<type><proofterm> ->
  'Node' <env><Prop><prooftree>
'LetIn' {<Id>|<Anonymous>}<proofterm><type><proofterm> ->
  'Node' <env><Prop><prooftree>
'App' <proofterm>{<proofterm>}* ->
  'Node' <env><Prop>{<prooftree>}*
'Case' <CaseInfo><proofterm><proofterm>{<proofterm>}* ->
  'Node' <env><Prop>{<prooftree>}*
```

図 4.3　証明項から証明木への変換

Lambda

Lambda はラムダ式に相当し，新たに仮定に変数や命題を加える際に現れる．変換先は Node となる．新たに追加される項は<Id>の内容にあたるため，<Prop>が示すこの項の型を決定する際には，Id が指す変数を加えた環境の上で決定する必要がある．<Anonymous>である場合は環境に影響を与えないため，特殊な処理はせずに型を決定する．この項は変数や仮定の追加だけを示すため，証明が枝分かれすることはない．したがって，このノードの子となる<prooftree>は 1 つだけになる．

LetIn

LetIn は局所関数定義に相当する．定義された局所関数は仮定と同じように扱うことになるため，処理は Lambda とほぼ同様となる．

App

App は関数適用を表す．最初の<proofterm>は関数を示す．2 つ目以降の<proofterm>は引数を表す．この項の型には関数の適用結果があたるが，このときすべての引数を関数に順に適用した結果を返せるように処理を行う．<env>と<Prop>を決定したのちに，Node に変換する．

Case

Case はパターンマッチに相当する．CaseInfo の内容は証明木に保持させる必要はない．この項の型の決定は，Case の 2 つ目の<proofterm>による．この<proofterm>は Lambda であり，この項の 2 つ目の<proofterm>が Case の項としての型に相当する．したがって，<Prop>を決定する際にはこのような処理を行い，正しく型を得られるようにする．この項からは証明が枝分かれする可能性があるため，このノードの子となる<prooftree>は複数存在しうる．

4.1.3 証明項の取得

本節以降，前節で述べた変換を実現するために実装した関数についての具体的な内容を述べる．まず変換処理のはじめに，現在行っている証明の証明項を取得する必要がある．Coq には現在進行中の証明の情報を取得する give_me_the_proof 関数と，証明の中で未定の項を含む証明項を取得する partial_proof 関数がすでに定義されている．これらの関数を用いて取得した証明の情報から，証明木を生成するのに必要な情報をパターンマッチを利用して参照し証明木の生成関数の引数として利用している．以下に，実際に呼び出される関数 showp_fun のコードを示す．

```
let showp_fun () =
  let p = try Proof_global.give_me_the_proof () with
            Not_found -> failwith "give_me_the_proof:not found" in
  let pprf = try Proof.partial_proof p with
               Not_found -> failwith "partial_proof:not found" in
  match Proof.proof p with (_,_,_,_,em) ->
    gen_file ( make_xml em (List.hd ((List.rev_map
                gen_ptree [] em)  (List.rev_map (rel2var [] )  pprf)))))
```

ここで，pprf は，Term.constr list 型を持つ．Coq 内での証明項は Term.constr 型で定義されており，この pprf は証明項のリストを意味する．pprf のそれぞれの項に対して，de Bruijn Index (後述) を変数に置換する関数 rel2var を適用する．その返り値のリストのそれぞれの項について，証明木の生成関数 gen_ptree を適用することで，証明木のデータが生成される．それを make_xml 関数が受け取り XML 形式の文字列に変換し，最後に gen_file 関数で外部ファイルへ出力するという流れになっている．各種関数の具体的な内容は後述する．

4.1.4 de Bruijn Index の置換関数

次に，de Bruijn Index を変数に変換する関数 `rel2var` の内部処理について述べる．de Bruijn Index とは，ラムダ計算等で利用される変数の表現法のひとつである．この記法では，変数についてその変数を宣言する λ が何層外側に位置するかを表す自然数を用いて表現する．

例として

$$\lambda\, y.(\,\lambda\, x.\, x\; y)$$

というラムダ式を考える．x が 1 に，y が 2 に対応しているため，これを de Bruijn Index の表現を用いて表すと

$$\lambda\,\lambda\, 1\; 2$$

となる．

ここで，実際に Coq 内部で扱われている証明項の一部を例として以下に示す．

```
Lambda(Name "A", Sort(Prop Pos),
        Lambda(Name "B", Sort(Prop Pos),
                Lambda(Name "H", Rel 2,
                        Lambda(Name "H0", Prod(Anonymous, Rel 3, Rel 3),
                                App(Rel 1, [|Rel 2|])))))
```

これをラムダ式と de Bruijn Index の表現法で記述すると，

$$\lambda\, A.(\,\lambda\, B.(\,\lambda\, H.(\,\lambda H0.H0\; H))) \qquad (\text{ラムダ式})$$
$$\lambda\,\lambda\,\lambda\,\lambda\, 1\; 2 \qquad (deBruijnIndex)$$

となる．このように，Coq 内部では de Bruijn Index を用いて変数の管理が行われているため，このままでは証明木を作る際に変数の表示が変数の名前ではなくただの数字となってしまう．数字による変数の参照はユーザにとって非常にわかりにくいため，証明木として構造を生成する前にこれらの変数指定をすべて実際の変数名に置換する必要がある．本節で述べる `rel2var` 関数は，変数名のリストと置換の対象となる証明項を引数として，内部の de Bruijn Index を対応する変数の名前で置換する関数である．

まず，証明項の型である `Term.constr` 型は，本研究で定義したインタフェース関数 `kind` によって，`(t, t) kind_of_term` 型に変換される．この型は，変数やラムダ式等の証明の構成要素に対応したデータ構成子によって形成されており，それらによるパターンマッチでそれぞれの

構成子に応じた処理を定義していく．

それぞれの構成子で Rel i の形を持つ証明項が含まれる可能性のある引数に対して rel2var を再帰的に呼び出し，証明項の内部に含まれる Rel をすべて置換する．Lambda や LetIn など，新しい変数を定義するような証明項の場合は，変数名をリストとして保持し変換時の変数表に追加していきながら処理を進めることで，正しい対応関係を保持できるようにする．Rel が現れた場合，変数名のリストを利用して対応する Var に置き換える．最終的に，すべての Rel の部分が対応する Var に置換された証明項がこの関数の返り値となる．

具体的な処理の例として，de Bruijn Index を変数に置換する処理のコードを rel2var より一部抜粋して以下に示す．

```
let rec rel2var (vs: Name.t list)(t: Term.constr): Term.constr =
match kind t with
|Rel i          -> begin try match (List.nth vs (i-1)) with
                        |Name v     -> mkVar v
                        |Anonymous ->
                                failwith "Anonymous : not supported" with
                   Failure "nth" -> (print_int i);failwith"Rel:not found"
                   end
    . . .
|Lambda (v,t1,t2)    -> mkLambda(v,rel2var vs t1, rel2var (v::vs) t2)
    . . .
```

Rel がマッチした場合，すなわち de Bruijn Index で表された項に対する処理について述べる．Coq での de Bruijn Index は 1 から始まるため，i 番目の Index に対応するのは変数名のリストの i-1 番目である．それが Name t，すなわち名前を示すものの場合はその名前を持つ Var を生成している．そうでない場合は，エラーを返す．

Lambda がマッチした場合，前述したように変数定義 (コード中では v) が新たに追加されることになる．そのため，Name.t のリスト，すなわち変数名のリストである vs に変数を追加し，そのリストを引数にして rel2var を呼び出すようにする．

また，実際のコードでは App や Case などのパターンマッチが続けて記述されている．Rel が表れない項についてはそのまま返り値とし，Rel を含む可能性のある項に対しては再帰的に rel2var を呼び出す．これにより，すべての証明項を走査して変換を行っている．

4.1.5　証明木の生成関数

次に，rel2var の返り値，すなわち de Bruijn Index が変数に置換された証明項から，証明木を生成する関数 gen_ptree の説明を行う．この関数では，証明が完了しているか否かにかかわらず，本処理が行われた時点での証明項を用いて対応する木構造を生成する．タクティクの適用によって証明が進むごとにその証明項に対応する証明木は変化し，一般に大きくなっていく．以下，図 4.4 に証明項から証明木を生成するイメージを示す．このように，1 つの証明項からは 1 つの証明木が生成される．証明を進行するにあたって証明項が変化するたびに証明木を生成することで，証明の 1 つ 1 つの状態に応じた情報を取得できるようにしている．

定義した証明木の構造を表す型として，以下の ptree 型を定義した．

```
type env = (Id.t * constr) list

type prop = constr

type ptree =
  |Not_yet of env * (prop * types)
  |Node of env * prop * ptree list
```

図 4.4　証明項と木構造の対応

env は変数名と型のタプルをリストとして保持する型で，証明中での環境を表す．ここには，例えば (A,"Prop") のように，証明中で宣言された変数の名前とその型が保持されていく．prop は命題を示す. 次に，ptree 型の各データ構成子の概要と役割を順に述べる．Not_yet は未解決の証明を表し，環境と未解決の項およびその型を情報として保持する．Node は一番多く証明木の要素として現れるもので，環境と証明項と自身の子となる証明木のリストを保持する．証明木のリストが空の場合，そのノード以降に証明が続かない，すなわちそのノードで証明が完了していることを示す．以下，それぞれのデータ構成子について証明木の生成処理の内容を説明する．

以下に，Var，Lambda から木構造を生成するコードを抜粋して示す．

```
let rec gen_ptree env em e : ptree =
        match kind e with
        . . .

        |Var v               -> Node(env,(type_of env em e),[])
        |Lambda (v,t1,t2)    -> let l_env = match v with
                                        |Name i    -> ((i,t1)::env)
                                        |Anonymous -> env in
                                  Node(env,(type_of l_env em e),
                                   [gen_ptree l_env em  t2])
        . . .
```

まず，Var の項をルートとする木構造を生成する場合について説明する．この項は変数そのものであるため，あとに続く証明は存在しない．したがって子となる証明木は存在しないため，子の証明木を入れるリストは空となる．

次に，Lambda について木を生成する場合は，まず v が変数の形をしているかどうかを調べる．変数である場合，すなわち v が Name i の形をしている場合，その型 t1 とのタプルを env に追加してから，Node の生成を行う．これにより，この項を表すノードの環境にこの項自身で宣言した変数も含めることができる．t2 には，証明木において子の部分にあたる証明項が対応する．そのため，t2 を gen_ptree に適用させた結果，すなわちこの証明項に続く証明を示す証明木を，子の証明木のリストとして保持させる．原則的に，Lambda で表される証明項では証明の分岐 (複数のサブゴールの出現) はないため，このリストは単項リストとなる．LetIn においては，局所関数定義は変数と同様に扱えるため Lambda の場合とほぼ同じ処理を行っている．

4.1.6 証明項の型取得関数

続いて，gen_ptree の補助関数として定義した type_of 関数について説明する．この関数は，型のタプルのリスト，ローカルな環境，証明項を受け取って，その証明項の型を返す．この型は，カリー・ハワード同型対応により証明中における命題と 1 対 1 で対応する．したがって，この関数は証明項に対してその項が示す命題を返す関数とみなすことができる．ほぼすべての構成子に対して処理を記述しているが，特に複雑な処理を施した部分だけを説明する．本拡張に合わせた処理を定義しなくてよいデータ構成子に対しては，Coq のコードで既に定義された型取得関数を用いて型情報を取得している．

Lambda や LetIn，App といった，変数の宣言や関数適用を含むような構成子の場合には，既存の型取得関数では望ましい結果が得られないため，新たに独自の処理を記述する必要がある．以下に，type_of 関数の Lambda，App に対する処理の部分を抜粋して，そのコードを示す．LetIn については，Lambda の処理とほぼ同一であるため割愛した．

```
let rec type_of (ts: (Id.t * constr) list) em (t:Term.constr): constr =
match kind t with

 |Lambda (v,t1,t2)   ->let ts = match v with
                              |Name i    -> ((i,unkind t1)::ts)
                              |Anonymous -> ts           in
                            let t_of_t2 = type_of ts em t2 in
                            let v = match v with
                              |Name i when has_id i t_of_t2 -> v
                              |_ -> Anonymous        in
                            mkProd (v,unkind t1,t_of_t2)

 |App (f,a) ->let al = Array.to_list a in
                 let rec t_app ls  tf al =
                  match (tf,al) with
                   |(Prod (v,_,t2),(arg::args)) -> t_app ((v,arg)::ls)  t2 args
                   |(_,(_::_))        ->
                                         (failwith ("App:non_prod was detected"))
                   |(_,[])           -> (List.fold_left  subst (unkind tf) ls) in
                 t_app [] (kind(type_of ts em f)) al
```

まず Lambda に対する処理を説明する．has_id 関数は，変数名と型を受け取って型の中にその変数が含まれているかどうかを調べる関数である．Lambda の処理では，最終的に Prod にして型を返すが，その際に変数が型の中に含まれている場合は $A \to B$ のような形，含まれていない

場合は $\forall A$ のような形となる．前者ならば (変数名：型) のリストに新たに追加しなければならず，後者ならば逆に追加してはならないため，パターンマッチを用いてリストに追加するかどうかの判断を行っている．その後，変数が含まれているかを確認してから Prod の構成子の形にして返す．

次に，App の処理を説明する．例として，$f\ a\ b\ c$ という関数適用を考える．この式の型は a, b, c すべてを適用させた後の返り値の型になる．そのため，すべての関数適用を完了した状態にならなければ正しい型を得ることができない．したがって，すべての引数を格納したリスト (al) を用意し，すべての引数に対して引数の名前と型を保持させた．コード中の ls が，引数名と型のタプルのリストに相当する．その後，subst 関数で型内部の変数をすべて対応する型に変換し，その結果を用いて左から畳み込みを行うことで，部分適用の結果の型を順に得ながら引数すべてを適用した結果の型を計算できるように記述した．

subst 関数は，証明項と (置換したい変数：対応する型) のタプルを受け取って，証明項内のすべての該当する変数を置換する関数である．変数は Var 型として現れるため，Var が現れる可能性のある要素に対して再帰的に呼び出すことで，証明項内の走査を行っている．これらによって，正しく関数適用の処理を行いその結果が型として取得される．

最後に，Case についての処理を説明する．前節で述べたように，Case 構成子で表される項の型には t1 の型が相当する．t1 の実態は Lambda(v,tx1,tx2) となっており，この v は Anonymous，すなわち変数宣言ではない．そのため，この項の型は tx2 がそのまま該当する．したがって，パターンマッチを用いてこの要素を参照し，Case の項全体としての型としている．

ここで，証明項を利用して証明の情報を得ることの利点について述べる．証明の情報を取得する方法として，ユーザが入力したタクティクを取得して証明をやり直すことで，ユーザが定義した処理を行う関数中で扱えるようなデータ構造を持つ証明を再構築するというアプローチがある．しかし，本研究においてはこの手法は採用しなかった．この手法で証明の情報を得ようとすると，ユーザが自動証明を行うようなタクティクを適用して証明を行った場合に，自動で行われた部分の証明の過程については「自動で証明が行われた」という情報以外に何も得られなくなってしまい，具体的な証明の流れを把握することができなくなるためである．それに対して，証明項は適用したタクティクに依存せずに，その構造を決定している．したがって，auto．などの自動的に証明を行うタクティクを適用したとしても，それらを使用せずに手動で 1 つずつタクティクを適用していった場合と同じ証明項を得ることができる．特に本研究は，初学者を対象にしているため，具体的にタクティクがわからなくても証明の構造を見たい，という場合に効果を発揮することが期待できる．そのため，本研究では，証明項を経由して証明の情報を得るように設計した．

4.1.7　証明木の受け渡し

本節では，Coq プラグイン側の処理の最終部分である，生成した証明木の XML 形式への変換ならびに外部ファイルとしての出力を行う関数について述べる．

証明木を表す XML

まず, 本研究で定義する証明木を表す XML の例を, 図 4.5 に示す．この例は, $\forall A \forall B \forall C, (A \to B) \vee (A \to C) \to A \to B \vee C$ の証明を，本プラグインを用いて XML 形式の証明木に変換したテキストの一部である．

本研究で定義する XML の形式は，以下のような要素を持つ．

```
<?xml version="1.0" encoding="UTF-8"?>
<prooftree>
 <node>
  <index>0</index>
  <term>forall A:Prop, forall B:Prop, forall C:Prop,
        ((A -> B) \/ (A -> C)->(A->B \/ C))</term>
  <env>
  </env>
  <isdone>true</isdone>
  <focused>false</focused>
  <isfinished>false</isfinished>
  <children>
   <node>
   <index>1</index>
   <term>forall B:Prop, forall C:Prop, ((A -> B) \/ (A -> C)->(A->B \/ C))</term>
   <env>
    <item>A:Prop</item>
   </env>
   <isdone>true</isdone>
   <focused>false</focused>
   <isfinished>false</isfinished>
   <children>
    <node>
     <index>2</index>
     <term>forall C:Prop, ((A -> B) \/ (A -> C)->(A->B \/ C))</term>
. . .
```

図 4.5　証明木を表す XML テキストの一部

- prooftree ・・・ルートとなる要素．XML は 1 つのルートを持つ必要がある．
- node・・・ノードを表す．各ノードの要素を子として持つ．
- index・・・node の子で，各ノードの通し番号を表す．拡張性のために定義した．
- term・・・node の子で，各ノードにおける命題を表す．証明において，その時々で証明すべきサブゴールである．
- env・・・node の子で，各ノードにおける環境を表す．環境とは，定義されている変数名や仮定を表す．
- item・・・env の子で，環境内の各要素を表す．仮定の数に応じて，item の数は変化する．
- isdone・・・node の子で，そのノードにおける証明が完結しているかを boolean で表す．
- focused・・・node の子で，そのノードに対応する項が Coq において現在タクティクが適用されるゴールとして設定されているかどうかを boolean で表す．
- isfinished・・・node の子で，このノードが含まれる証明全体において証明が完結しているかどうかを boolean で表す．1 つの prooftree に含まれる isfinished はすべて同じ値となる．
- children・・・node の子で，そのノードの子となる node を子に持つ．

XML テキストの生成

次に，証明木を受け取って，それに対応した XML 形式のテキストを生成する make_xml 関数の処理について述べる．全体のおおまかな処理としては，ヘッダ部分と，最初の<prooftree>，最後の</prooftree>を除いた部分を，下請け関数である genxml 関数に生成させている．genxml 内では，以下のローカル関数が定義されている．

- check_isdone・・・isdone が true か false かを決定する．具体的な決定方法は後述する．
- env_xml・・・env 部分を担当する．変数名と型を受け取って，それをもとに<item>，</item>で囲われたテキストを生成する．
- count_children・・・証明木を引数にとり，子の証明木がいくつあるかを返す．子に対する index の決定に利用される．

ここで，check_isdone の決定方法について述べる．引数となった木の構成子が子を持たない Node ならば，それ以上証明が続いておらず，かつ未解決でもないため true とする．Not_yet であれば未解決であるため false，子を持つノードであれば，子の木のルートの isdone が全て true であれば true，そうでなければ false としている．これにより，証明が全て完了したときのみ，ルートのノードの isdone が true となる．

これらの関数により生成されたテキスト群を結合し，完全な XML 形式にしたテキストが，make_xml 関数の返り値として外部ファイルへの出力関数に渡される．

外部ファイルへの出力

最後に，完成した XML テキストを GUI が参照できるよう，外部ファイルへ出力する gen_file 関数の処理について述べる．外部ファイルへの出力は，OCaml 標準ライブラリで提供されている入出力チャネルを利用して行う．指定したパスに存在するファイルを開き，そこへ生成した XML テキストを書き込む．このとき，指定するパスはインタフェース側から参照できる範囲にある必要がある．最後に出力チャネルを閉じることで，処理が終了する．これにより，Coq でコマンドを入力するとその時点での証明項から証明木を生成し，その情報を記述した XML テキストが外部ファイルへ出力される (本研究では output.xml という形で出力される) という一連の処理がアトミックに行われる．

4.2 GUI

次に，実際にユーザが操作するインタフェース側の実装の説明を行う．本インタフェースによって，本研究の目的の 1 つであった，証明木を用いた，証明のわかりやすい表示が提供される．本インタフェースで実現した機能は，以下の通りである．

- 証明を証明木の形で表示する．
- 証明中の環境を，証明木中ではなく，別の場所に分けて表示する．
- 証明木の部分的な折りたたみを行う．
- インタフェース上で，Coq に対するタクティクなどの入力を行う．
- Coq を用いて記述した証明を，インタフェースに読み込む．
- インタフェースを用いて記述した証明を，Coq で扱えるように出力する．

本節では実装したユーザインタフェースの機能を説明し，その後に，クラス構造や実装した機能の具体的な処理の内容を述べる．

4.2.1 実装した機能

まず，実際のインタフェースの画面表示を図 4.6 に示す．

まず，画面中央に Coq から取得した証明を表す証明木が表示される．各ノードに表示されている文字列が，Coq での証明におけるサブゴールを表す．また，ノードをクリックすることでそのノードに関する詳細な状況が画面左側に表示される．

画面左側のテキストエリアは，クリックで選択したノードの情報が表示される部分である．具体的には，そのノードにおける環境が表示される．言い換えれば，そのサブゴールを証明する際

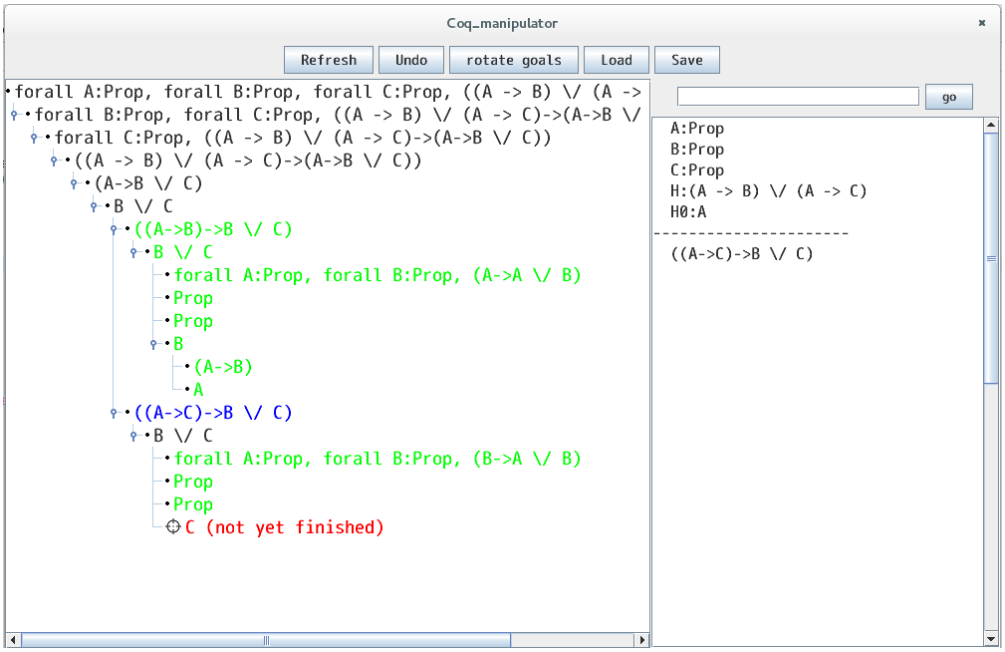

図 4.6　インタフェースの画面表示

に利用可能である変数や仮定がサブゴールとともに表示される．

次に，証明木の折りたたみ機能について説明する．子を持つようなノードのアイコン部分をクリックすることで，そのノードをルートとする部分を折りたたむことができる．実際に，図 4.6 の $(A \to B) \to B \vee C$ の部分から折りたたんだ画面を図 4.7 に示す．

この機能により，ユーザは証明木の任意の箇所を折りたたみ，表示する量を削減することができる．特に枝分かれが多く証明木の表示が縦に長くなるような場合に本機能を用いることで，比較的長い証明でも全体の構造を把握しやすく表示することができる．これらにより，結果として証明の視認性の向上が期待できる．

画面右側のテキストフィールドに，現在の証明を進めるためのタクティクを入力し go ボタンを押下することで，入力されたタクティクが Coq 側へ送られ，そのタクティクにより処理が行われた新しい証明木が中央部分に表示される．これにより，ユーザはいちいちインタフェースの画面と Coq の画面を行き来しなくとも Coq を用いた証明が可能となる．

画面上部のボタンはそれぞれ証明木の再読み込み，アンドゥ，証明するゴールの変更，Coq で読み込める証明ファイルの読み込み，Coq で読み込める証明ファイルへの保存を行うためのものである．読み込みを行った場合そのファイルで行われていた証明が画面に表示される．保存を行うと，それまでにユーザが入力したタクティクが保存されたファイルが生成される．これにより，もともと Coq で記述していた証明を本インタフェースで確認することや，逆に本インタフェースで記述していた証明を Coq 本来の表示で見ることも可能である．

本インタフェースで提供する機能は以上である．

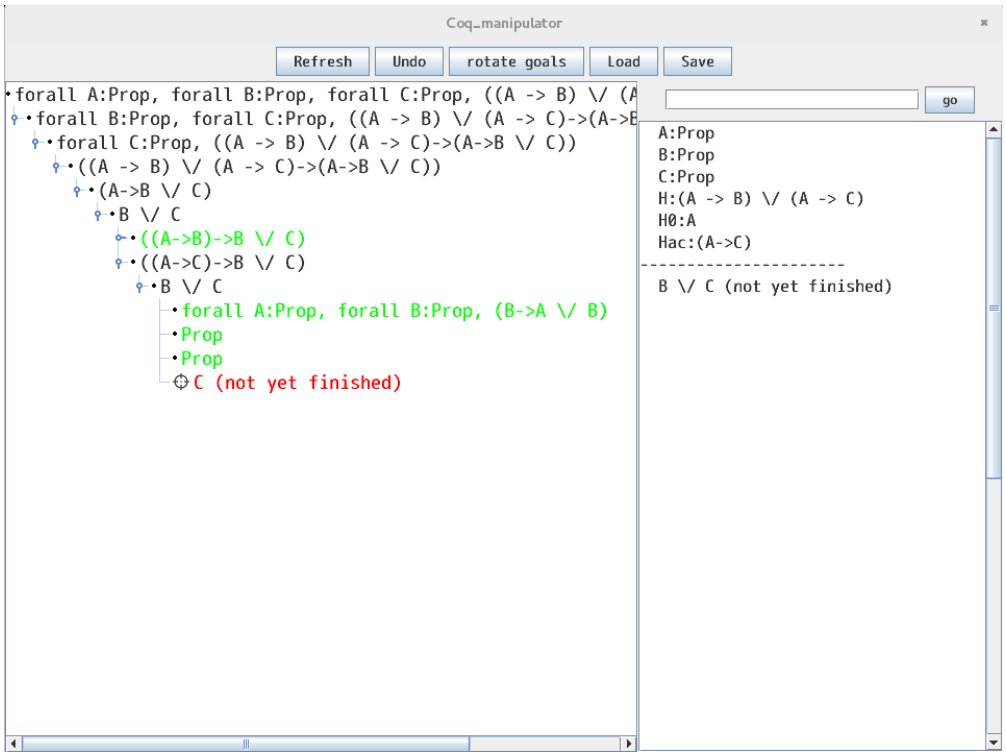

図 4.7　インタフェースの画面表示

4.2.2　実装したクラス

本インタフェースを実装するにあたって用意したクラスについて，それぞれの役割を述べる．実装したクラスは以下の 6 つである．

- MainPanel クラス
- ProofTree クラス
- Reader クラス
- InputManager クラス
- ButtonManager クラス
- MyTreeCellRenderer クラス

MainPanel クラスでは，実際にユーザが閲覧したり操作したりするウィンドウやウィンドウ内の領域についての定義を行っている．本システムを起動した際の処理もここで行う．ProofTree クラスでは，証明木の構造の定義を行っている．Reader クラスでは，前節で説明した Coq プラグインによって生成された XML ファイルを読み取り，そこから証明木のインスタンスを作成する処理を行う．InputManager クラスでは，インタフェース上の入力フォームから入力されたユーザによる証明操作を Coq 側へ伝達するための処理を定義している．ここで証明操作とは，タクティクの入力や証明したい定理の入力などを指す．次に，ButtonManager クラスでは実装し

た機能を呼び出すボタンがクリックされた際の処理を定義している．最後に，MyTreeRenderer クラスでは証明木の表示について，各ノードの状況に応じた表示方法の定義を行っている．

次節以降，6 つのクラスについて具体的な処理の様子を説明していく．

4.2.3 MainPanel クラス

MainPanel クラスでは，フィールドとして以下の要素を持つ．

- 証明木や環境の表示，ボタンの配置に使用するパネル
- 表示や入力に使用するテキストフィールド
- 証明木
- バックグラウンドで稼働する Coq のプロセスハンドラ
- ユーザの入力した文字列を管理するリスト

プログラムとして実行するのに必要な Main 関数も，このクラスのメソッドとして定義している．Main 関数では，初期処理としてウィンドウ全体のインスタンス化や各ボタンの配置，イベント処理においてボタンの区別のために利用するアクションコマンドの定義などを行う．この際，バックグラウンドで Coq のコマンドインタフェースである Coqtop を呼び出し，そのプロセスをインスタンスとして保持するようにした．以降はこれを引数として利用することで，Coqtop への入力処理を定義できる．

また，画面上の証明木をクリックすることで起動するイベント処理もこのクラス内で行った．本インタフェースでは，インタフェース上の証明木のノードをクリックすることでそのノードにおける環境を表示する機能が実装されている．ProofTree 型のデータは JTree というライブラリで提供される構造で定義しており，各 get 関数を用いてどのノードがクリックされたかを知ることができる．ノードがクリックされていることが判明した場合，ProofTree クラスで定義している `makeEnv` 関数で，表示するための環境を文字列の形で生成し env として保持する．これにより，インタフェースの証明木上でクリックしたノードに対して，そのノードの状態における仮定や変数，サブゴールといった情報を表示できるようにしている．

4.2.4 ProofTree クラス

次に，ProofTree クラスについて説明する．ProofTree クラスは証明木の構造を定義したクラスで，メンバーとして以下の要素を持つ．

- index
- term
- env
- isdone

- children

それぞれ，4.1 節で示した XML 形式への変換で用いたタグと同様の内容を要素として持つ．このクラスで定義したメソッドは 2 つあり，証明木の項を受け取ってそれをルートとした木構造を生成する `makeTree` 関数と，前節で述べた環境を文字列として生成する `makeEnv` 関数である．`makeTree` 関数は，すでに index や term などの各情報が格納された `ProofTree` を引数にとり，そのルートノードを表示用の形式にして返す．まず自身を登録し，次に自身の子の ProofTree に対して再帰的に `makeTree` を適用し，その返り値を子として登録するという手順になる．また，`makeEnv` は引数にとった `ProofTree` の env の内容を読み取り，文字列に直して返す．

4.2.5 Reader クラス

Reader クラスでは，外部の XML ファイルから内容を読み取り，対応する証明木を生成する処理を定義している．メンバーはなく，メソッドだけを定義している．

このクラスで定義しているメソッドは，`treeParse` 関数と，`mtXML` 関数である．

`treeParse` 関数は証明木の構造を持つ XML ファイルを読み込み，内容をパースしてその証明木のルートノードを返す．Java での外部ファイルの読み込みは，`DocumentBuilder` パッケージを利用して行った．OCaml プラグイン側によって生成された `output.xml` からテキストデータをインスタンスとして取得し，そこから最初の`<node>`の内容，すなわちルートノードの情報を `node` として保持する．最後に，下請け関数である `mtXML` 関数に渡し，その結果を返すという処理になっている．

ここで，`mtXML` 関数は，ノードを受け取ってそれに対応する ProofTree を返す．実際の処理中はルートノードに対してこの処理を行う．ルートノードは証明木全体を表すため，これを引数として与えることで結果として証明木全体に対すて処理を行える．また，Java で XML を扱うにあたって SAX というライブラリを利用した．SAX は XML 文書を扱うための API で，少ないメモリで大きな XML ファイルも読み込めるのが特徴である．`mtXML` 関数では，前節で述べた証明木の要素について XML から一項目ずつ抜き出し，それぞれ対応した要素として `ProofTree` に保持させる処理を行う．このように，このクラスでは，OCaml 側から得た XML 形式の証明木から Java で扱える形の証明木への変換を行っている．

4.2.6 InputManager クラス

InputManager クラスでは，インタフェースで入力されたユーザのタクティクを Coq 側に伝達する処理を行う．メンバーとしては，以下の要素を持つ．

- 入力ストリーム
- 出力ストリーム

- バッファリーダ
- Coqtop のプロセス
- プリントストリーム

ここで，Java と外部プロセスとの入出力のやり取りはストリームと呼ばれる入出力用のチャネルを介して行う．Java における入力ストリームには外部プロセスからの出力が流れ込み，Java の出力ストリームから送られたデータは外部プロセスへの入力になる．プリントストリームは出力ストリームを引数にとって生成し，ここに外部プロセスへの具体的な入力が入る．本研究では，外部プロセスは Coqtop が稼働しているプロセスが相当する．

このクラスでは，起動時に呼び出されてバックグラウンドで Coqtop を起動する `execute` 関数と，ユーザからの入力を Coqtop へ伝達する `throw_to_coq` 関数が定義されている．外部プロセスの取得には，`ProcessBuilder` パッケージを利用した．`execute` 関数では，起動した Coqtop の入出力ストリームを取得し，最初にプラグインを導入する宣言として `Require Import Coq_custom.Showp.` と Coqtop に入力させる．これにより，バックグラウンドで起動した Coqtop に前節で述べたプラグインを導入する処理を行う．入力が終わったらプリントストリームをフラッシュし，出力ストリームを閉鎖することで起動処理が完了する．

次に `throw_to_coq` 関数について述べる．この関数では，ユーザによるタクティクなどの入力を 1 行の文字列として引数にとり，末尾に`" Showp."`をつけて Coqtop への入力として流す．この関数が呼ばれるのはタクティクや定理が入力された場合なので，そのたびにプラグインで実装した機能を呼び出すコマンドを入力させることで，証明木の更新を行うようにしている．また，現在の証明木のルートの`<isfinished>`が True，すなわちすべての証明が終わっている場合について，入力が「Qed.」であった場合，Coq 側では証明が完了し，新たな命題の入力を待つ状態に遷移する．そのため，インタフェース側の対応として，証明木を起動時のものに上書きすることで次の証明を開始できるようにしている．以上が InputManager クラスの説明となる．

4.2.7 ButtonManager クラス

次に，ButtonManager クラスについて説明を行う．このクラスは MainPanel クラスで生成したインスタンスをコンストラクタの引数としてとり，そこに配置されたボタンのイベント処理を担っている．以降，それぞれのボタンが押下された際の処理を説明する．

Reload ボタン

証明木の再表示を行う．証明木の更新が行われた際に，何らかの理由で XML ファイルの生成に時間がかかることがある．その際に再表示処理が先に終わってしまうと，画面に表示されるのは 1 手前の証明木になり，画面と Coqtop の状態に齟齬が出てしまい，正しい証明の進行を妨げる要因となりうる．そういった場合にこのボタンを押下することで，現状の証明木を再び読み込

んで表示し，Coq 側とインタフェース側の状態の不一致を解消することができる．

Undo ボタン

Coq で証明を行う際に，タクティクの適用には成功したが証明は正しく進行しないという場面や，複数のタクティクを用意して証明が進行するかの試行を行う場面が存在する．そういった場合にスムーズに作業を行うには，1 つ前の状態に容易に戻る手段が必要となる．このボタンを押下すると，最後に入力したタクティクを撤回し証明がひとつ前の状態へ戻る．

Rotate goals ボタン

Rotate goals ボタンを押下すると，証明が完了していないノードが同じレベルに複数個ある場合に限り，証明を進めるノードを変更できる．また，ノードを変更したという記録も同時に取るように実装を行ったため，後述する Save ボタンで書き込まれるファイルにもユーザの操作と同じ操作をサブゴールに対して行うコマンドが記録される．

Load ボタン

Load ボタンを押下するとファイル選択ウィンドウが現れ，ユーザは読み込む証明のファイルを選択する．ファイルが決定したら，そのファイルに記述されている定理の記述やタクティクを Coqtop へ伝達し，ファイルの記述の続きから証明を行える状態にする．まず `inputlist` を初期化し，選択したファイルから一行ずつ読み込んだ内容を追加していく．同時に，Coqtop へ同じ内容を入力として与えることで，もとの証明ファイルで行われていた証明と同様の状態を作るように実装した．また，`inputlist` にはこのファイルに記述してあった入力と同じ順序で同じ内容が格納されるため，後述する Save 機能を利用すれば，もともと途中まで進んでいた証明を本インタフェース上で進め，その内容を上書きすることも可能である．

Save ボタン

最後に，Save ボタンが押下された際の処理について述べる．ここでは，ボタンが押下された時点での `inputlist` の内容を Coq で読み込めるファイル形式で保存する処理を行う．ユーザが入力したタクティクなどが `inputlist` の要素として保持されているため，それらを 1 つずつ受け取って，ファイルに書き込んで改行する処理を繰り返す．書き込みが終わって出力されたファイルはそのまま Coq 本体に読み込ませることが可能な形式になっており，Coq 本体で証明の続きを行うことも可能である．

また，それぞれのボタン押下処理の後に，証明木の再描画処理を行っている．これにより，ボタンによる処理の結果を画面に反映させ，Coq 側とインタフェース側の状態を同期させている．

4.2.8 MyTreeRenderer クラス

最後に，MyTreeRenderer クラスについて述べる．このクラスでは，証明木の各ノードにおける `isdone`，`focused`，`isfinished` の情報から，ノードを表示する際の文字色やアイコンを決定する処理を定義している．具体的には，入力したタクティクが適用されるゴールに対応するノードにはその旨を表すアイコンを表示させた．続いて，現在詳細情報を表示しているノードの文字色は青，未解決の項は赤色，証明が完了している部分木のノードは緑色に表示させ，証明の進捗の様子を確認しやすくした．

5　考察

本章では，実装したシステムについて，現状で存在する Coq の GUI である CoqIDE との比較を行う．加えて，実装に行き着かなかった機能についても述べる．

5.1　CoqGUI との比較

まず，2 つのインタフェースについて同じ命題の証明を行った際の画面を，図 5.1，図 5.2 に示す．図で行っている証明は，$\forall A\, \forall B\, \forall C, (A \to B) \vee (A \to C) \to A \to B \vee C.$ の証明である．

まず，ユーザが目にする部分の比較を行う．CoqIDE では，画面左側に入力したタクティクの羅列，右上側に現在のサブゴールと環境が表示される．1 章でも述べたとおり，CoqIDE ではタクティクの羅列だけを読んで証明を追わなくてはならない．しかし，本インタフェースでの表示であれば証明の全体の形を一目で把握することができる．加えて，前述した畳み込み機能も用いれば視認性はさらに向上する．色やアイコンによって各ノードの状態の把握もしやすくなる．以

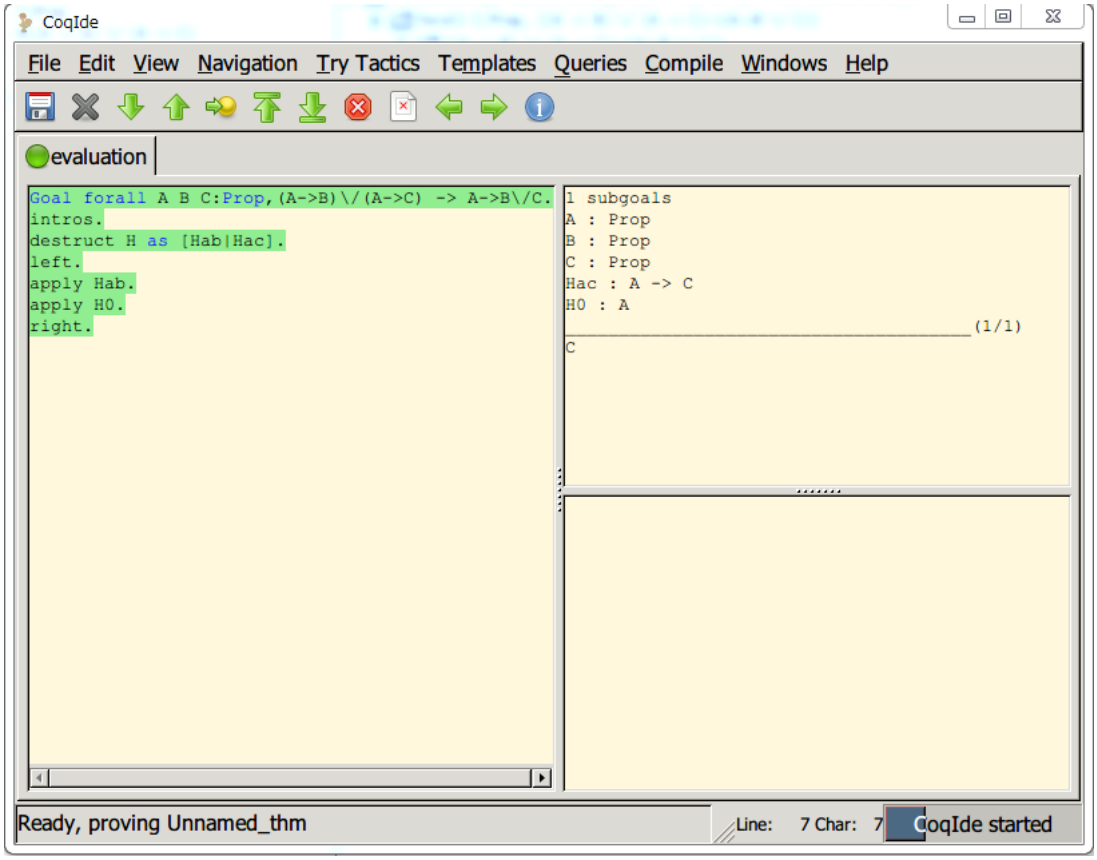

図 5.1　CoqIDE のインタフェースの画面表示

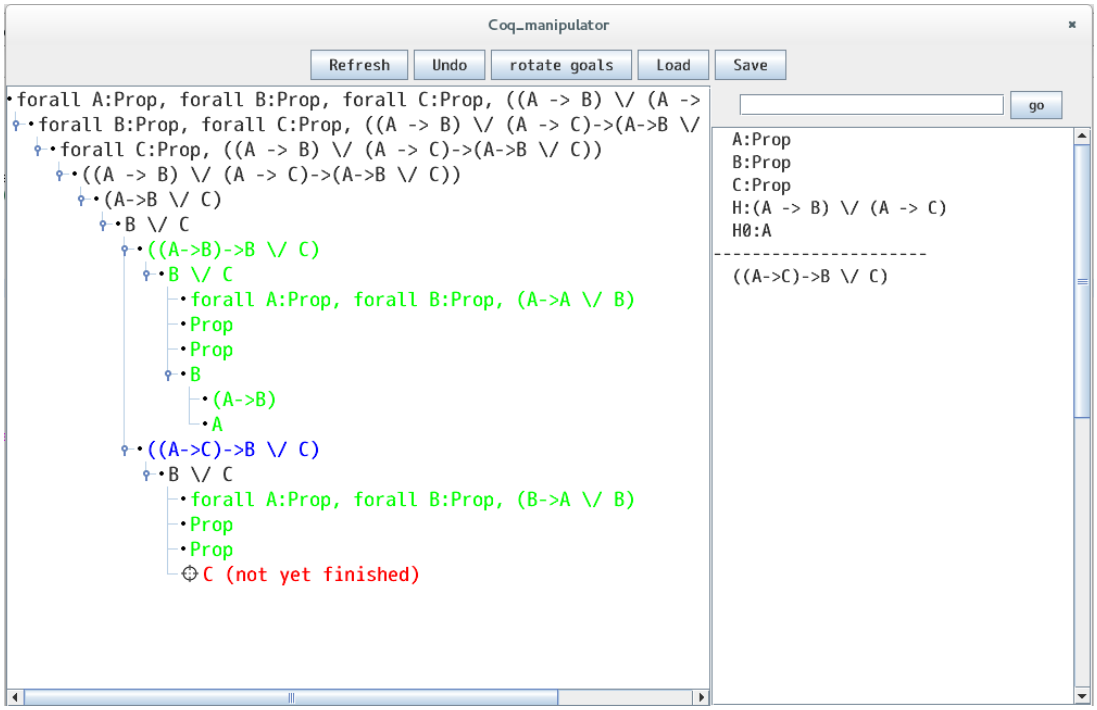

図 5.2　本研究で実装したインタフェースの画面表示

上の理由から，本インタフェースは証明の視認性を直感的な操作で向上させるという本研究の目的に寄与しているといえる．

次に，本インタフェースでは，CoqIDE で提供しているファイル保存・読み込みやアンドゥといった標準的な機能も搭載することができた．初学者に不便のない形で，CoqIDE だけではなく本システムを利用してもらう動機付けをすることができた．

続けて，本システムの拡張性について述べる．CoqIDE は単体で完結しているインタフェースであり，OCaml プラグインによって新たなタクティクや関数ライブラリの追加などの機能を追加することが可能である．しかし，Coq 本体の内部を改造して機能の追加や変更を行う場合，その機能により正しい証明が行われなくなる可能性が生じ，本来 Coq が持っていた証明の信頼性が保証されなくなるという欠点が存在する．それに対して，本システムでは Coq での証明に関する処理とインタフェースでの表示に関わる処理は XML を介して連携しているだけであり，Coq の証明を扱う部分は一切変更を施さずに新しいインタフェースの実装を行っている．したがって，Coq 本来の信頼性を損なうことなく機能を追加することが可能である．また，Coq のバージョンに依存しない設計になっているため，証明項の内部表現が変わらない限りは本システムを利用可能である．前述した機能追加の際にも，Coq の実装状況を考慮することなく，独自の機能を実装できる．以上の理由から，本システムは Coq 単体にはない拡張性を実現することができた．

最後に，対応可能な証明の範囲について述べる．CoqIDE においては，Coq を用いて記述できるすべての証明が表示可能である．しかし，本システムにおいては，表示可能な証明が一部限定

されている．これは，4 章で述べた証明項の構成子のうち，App の要素の構造に起因する．初学者にとって多く触れるであろう比較的単純な証明においては，App の要素の構造は同じ形をしており，4 章で説明した関数群はそれに対応できるように実装されている．しかし，複雑な関数適用になると，App の要素の引数の数が変化することを確認している．App で適用される関数のパターンは膨大で，それらすべてに対応できるようにするのは現実的ではなかった．対象を初学者と限定していることもあり，それら全てに対応する必要性は高くないと考え，本研究では対応を行わなかった．今後，本システムの性能を引き上げ，本システムの対象となるユーザの範囲を拡大する場合は，まずこの部分の対応を進める必要がある．

5.2 定量的評価

前節では，CoqIDE と本システムに対して主観的な比較や評価を行った．本システムの効果を客観的に評価するためには，実際に第三者にシステムを利用してもらうことが不可欠である．被験者を集めて本システムを利用した実験を行うことができた場合，UEQ [10] を用いて，ユーザの体験を定量的に表すことができる．UEQ は，プログラムやアプリケーションにおけるユーザの体験，すなわち使いやすさや効率性などをカテゴライズし，点数として表現できるアンケート方式の評価方法である．被験者の実験に対する意欲や心理状態などで評価が変動することが考えられるが，経験を定量的に評価する手段として有用性がある．今後の課題として，本システムに対してこのような定量的な評価を行うことが挙げられる．

5.3 実装できなかった機能

最後に，本研究の範囲で実装できなかった機能について述べる．

5.3.1 タクティクのサジェスト機能

初学者にとって，今どのタクティクが有効であるか，またどのタクティクをどのように適用すれば望んだ通りに証明を進められるかを考えるのは困難を伴う．そういった場合にシステム側で有効なタクティクを提示することができれば，初学者への支援という目的に沿う機能となる．有効なタクティクの候補を提示するだけなら，メインで稼動している Coqtop とは別に検証用の Coqtop を複製し，そこにすべてのタクティクを 1 つずつ入力し，正しく証明が進んだものだけを表示するという手段が考えられる．しかし，タクティクの総数はかなり多く，しかも引数を伴うものに関してはそれだけパターン数が増す．そのため，すべてのタクティクを試すというのは現実的ではない．apply や reflexivity などの頻出するものだけを選んで試行することでパターン数を削減するという手法も考えられるが，サジェスト機能としての精度が格段に落ちてしまう．したがって，この機能を実装するためには，証明中におけるタクティクごとの証明への登場

頻度や正しくタクティクが適用されたパターンなどを記録したデータベースと連携し，現在進行中の証明における有力な候補を絞り込むといった手段をとる必要がある．本機能を実現するためには，まずこのようなデータベースをどのように作成するか，またデータベースとの連携をどう実装するかといった方針を決める必要がある．

5.3.2　証明のループの検出機能

初学者が陥りやすい証明のミスのひとつに，タクティクの適用によって証明が進まずにループしてしまうというものがある．従来の CoqIDE などでは，このループを視覚的に発見できるような機能は提供されていない．本システムの表示方法であれば，ループに陥っている部分はほぼ同じ形状の木として現れることが推察されるため，それを検知してユーザに知らせる機構を実装すれば，このようなミスを防ぐあるいは早期に発見することが可能になる．本機能を実装するためには，類似したあるいは同等の構造を持つ部分木の存在を検知するアルゴリズムが必要となる．実装するためには，そのアルゴリズムの確立とどの程度の大きさのループまで検知するようにするかなどの種々のパラメータの吟味や定義が求められる．

6 関連研究

本章では，本研究に関連する既存の研究について本研究との共通点と差異を中心に考察する．

ProofGeneral [11] は，Emacs 上でいくつかの定理証明支援系を動作させるためのツールである．本研究で対象とした Coq も ProofGeneral 上で動作させることができる．GUI としての見た目は CoqIDE とほぼ同一であるが，Coq を外部で稼働させるのではなく ProofGeneral のシステムの一部として内包している．そのため，Coq 内部で扱われているメタ情報などを抜き出して表示でき，この点が本システムとの大きな差異である．ただ，証明自体の整形出力機能はデフォルトでは搭載されていないため，証明を証明木やその他の見やすい形で表示させるためにはそのための拡張が別途必要となる．

Web ブラウザ上で Coq を動作させるためのインタフェース [12] も存在する．この研究では，Web 上に CoqIDE とほぼ同等の機能をもつインタフェースを用意し，Coq 本体はサーバ上で稼働させている．インタフェースと Coq 本体とのデータのやりとりは HTTP を用いて行われており，ユーザからの入力や Coq からの出力の管理は JavaScript で行っている．Coq のインタフェースの設計ならびに実装という点で本研究と目的を同じくするが，その実現方法に大きな差がある．また，この研究では十分普及したインターネット上で Coq を手軽に利用するということが第一目標となっているが，本研究の目的は Coq を扱いやすいように改良するというものであり，その点が大きな差異となっている．さらに，本研究ではプラグインを用いて機能を実装しているため，他のプラグインを追加して導入することで機能の拡張が可能である．Web サーバ上の Coq にユーザの手元にあるプラグインを導入することは容易ではないため，この点でも差が存在している．

The Incredible Proof Machine [13] は，Web 上で証明を学ぶための Web コンテンツである．ユーザは含意の導出規則や $\wedge$，$\vee$ などを表すブロックを配置し，それらを線でつなぐことによって証明をグラフィカルに構築できる．基礎的な証明の問題が複数用意されており，ユーザはこれらを解いていくことで証明の学習を行えるように設計されている．このコンテンツは主に教育を目的としてデザインされており，証明の初学者に対するアプローチを行うという点に関しては本研究と類似する箇所が存在する．この研究では，証明の構築にあたって特定の定理証明支援系などには依存せず独自で導出規則や除去規則などを定義し，それを利用して証明の正誤などを判定している．証明の扱いを Coq に依存している本研究とは，その点で違いが存在する．

Miki β [14] は，ユーザが記述した証明木を描画する GUI である．このインタフェースは単純型付きラムダ計算に対応しており，OCaml と，OCaml で GUI 開発を行うためのライブラリである lablgtk を利用して開発されている．ユーザは証明木を作りたい命題を記述し，規則の適用などを GUI 上で行っていくことで証明の進行に対応する証明木を閲覧することができる．証明木の構成に必要な導出・除去規則などはすべてこのシステム内で定義されており，Coq に規則

の扱いを任せている本研究とはこの点で異なっている．この研究と本研究のわかりやすい共通点は，証明木を用いて証明をわかりやすく表示させようとしているという点である．また，論文中で著者は以下のように述べている．

> Actually, growing a proof tree can be regarded as decomposing and proving goals in theorem proving. A theorem prover typically implements various kinds of automation, such as tactics found in Coq [1]. It is an interesting challenge to incorporate such automation in Miki β.

Miki β では Coq 等の定理証明支援系との連携は行っておらず，特に自動証明に関して，Miki β との組み合わせについて言及している．本研究で利用している Coq には自動で証明を行えるタクティクが存在しており，本システムでそのようなタクティクを適用した場合，自動で処理された証明の過程もすべて飛ばさずに表示することが可能である．

7　結論

本章では，本研究で得られた成果と，これからの展望や残った課題について述べる．

7.1　成果

本研究では，Coq や論理学の初学者を対象として証明をわかりやすく表示し，かつ証明の手助けを行えるようなユーザインタフェースの設計ならびに実装を行った．実現に際しては，Coq 側の処理を行う機構とユーザが操作するインタフェースに分けて設計を行った．OCaml で Coq プラグインを設計し，証明の情報を証明項と呼ばれる構造を利用して取得した．それを証明木の構造を用いて整形し，インタフェースに伝達できるようにした．インタフェース側では，受け取った証明木をわかりやすい形で表示することで，ユーザにとっての証明の視認性の向上を実現することができた．また，証明の流れを視認しやすくすることで，初学者にとっての証明の理解の手助けをするという目標も達成できた．

7.2　展望

5.3 節で述べた，証明をやりやすい部分から始める機能やタクティクのサジェスト機能を実装することができれば，本システムの実用性，汎用性はさらに高まることが期待される．また，内部処理を改良し扱うことのできる証明の範囲をより大きくしていけば，初学者のみならずある程度 Coq に熟練したユーザにとっても実用的であるようなツールとなれると考えられる．Coq のユーザビリティ向上という目的を真に達成するためには，まだ改善の余地がある．

謝辞

本研究を行うに当たりまして，御指導御鞭撻を賜りました中野圭介准教授，岩崎英哉教授に深く感謝申し上げます．また，本研究に対して多大なご助言を頂きました中野研究室および岩崎研究室の皆様に心から感謝いたします．

参考文献

[1] The Coq Proof Assistant. https://coq.inria.fr/

[2] Georges Gonthier. *A computer-checked proof of the Four Colour Theorem*, 2004. http://research.microsoft.com/en-US/people/gonthier/4colproof.pdf

[3] Xavier Leroy. *Formal certification of a compiler back-end, or: programming a compiler with a proof assistant*, 2006. http://pauillac.inria.fr/~xleroy/publi/compiler-certif.pdf

[4] SSReflect. http://ssr.msr-inria.inria.fr/

[5] OCaml. https://ocaml.org/

[6] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, pp.108-110, 2002.

[7] Eclipse. https://www.eclipse.org/

[8] XML. https://www.w3.org/XML/

[9] Nicolaas Govert de Bruijn. *Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem*, Indag.Math., 34(5), pp.381-392, 1972.

[10] Bettina Laugwitz, Theo Held, Martin Schrepp. *Construction and Evaluation of a User Experience Questionnaire*, HCI and Usability for Education andWork, pp.63-76, 2008.

[11] David Aspinal. *Proof General: A generic tool for proof development*,Tools and Algorithms for the Construction and Analysis of Systems, pp.38-43, 2000.

[12] Cezary Kaliszyk. *Web Interfaces for Proof Assistants*,Electronic Notes in Theoretical Computer Science, pp.49-61, 2007.

[13] Joachim Breitner. *Visual theorem proving with the Incredible Proof Machine*, International Conference on Interactive Theorem Proving, Interactive Theorem Proving, pp.123-139, 2016.

[14] Kanako Sakurai, Kennichi Asai. *MikiBeta: A General GUI Library for Visualizing Proof Trees* , 20th International Symposium on Logic-Based Program Synthesis and Transformation, pp.184-193, 2010.