情報処理学会論文誌　プログラミング　Vol.5　No.4　27–38 (Sep. 2012)

定理証明支援系 Coq への対話的修正機構の導入

森口 草介[1,a)]　渡部 卓雄[1,b)]

受付日 2012年2月14日, 採録日 2012年5月21日

概要：定理証明支援系 Coq に対する，対話的にプログラムの変更・修正を行うための手法を提案し，その手法を Coq に組み込んだ ECoq を実装する．定理証明支援系によるプログラムの検証は，プログラムがある性質を満たすという証明により行われる．この証明はプログラムの構造や記述と非常に強く結び付いているため，プログラムを変更した場合，証明もまたその変更に依存した変更を行わなければならない．しかし，プログラムと証明の一貫性を保つために必要な変更箇所は，たとえばコンストラクタの追加という簡単な変更に限った場合であっても，既存の証明支援系では見つけにくい場合がある．このような問題に対処するため，我々は Coq で検証を行ったプログラムと証明に対して，その一貫性を保ちつつ変更するための手法を提案する．本論文で提案する手法では，コンストラクタの追加を行い，変更が必要な箇所を利用者に提示する．このとき，コンストラクタを追加する型だけではなく，宣言時の状態やその型を利用する他の型などの情報を用いるため，本手法は Coq の内部に組み込むことを前提としている．この機能は，変更が必要な箇所を対話的に修正するものであるため，我々はこれを対話的修正機構と呼ぶ．本論文で紹介する ECoq は，コンストラクタの追加を行うコマンドを Coq に追加したものである．ECoq を用いることで，利用者はソースコードに直接コンストラクタを追加して得られるエラーメッセージより細かい粒度での情報が得られる．特に ECoq は，通常決してエラーが起こらないが，修正する可能性のある箇所を指摘することで，利用者が修正箇所を見つける補助を行う．本論文では，例題を通じてエラーメッセージが出ない箇所を ECoq が提示できることを確認する．

キーワード：Coq，対話的修正機構，プログラム拡張，プログラム検証

Implementation of an Interactive Correction Mechanism for Coq

Sosuke Moriguchi[1,a)]　Takuo Watanabe[1,b)]

Received: February 14, 2012, Accepted: May 21, 2012

Abstract: It is generally difficult to extend or modify an already-verified program while maintaining the consistency of the program itself and its accompanying proofs of certain desirable properties. In this paper, we propose a novel method to support the process of modifying verified programs by interactively correcting the program definitions and proofs developed with Coq proof assistant. For this method, we introduce ECoq, our extended version of Coq equipped with a component called interactive correction mechanism. The mechanism described in this paper deals with the addition of new constructors to existing inductive types and then tries to locate prospective correction points within the modified program and its proofs. Thanks to this mechanism, our method enables us to find such correction candidates more accurately than usual process guided by error messages of Coq. In particular, ECoq can point out some correction candidates that do not originate any errors.

Keywords: Coq, interactive correction mechanism, program extension, program verification

1. はじめに

対話的証明支援系は数学的な定理の証明のほか，言語の意味論や型システムの健全性，コンパイラの正当性の証明

[1] 東京工業大学大学院情報理工学研究科
Graduate School of Information Science and Engineering, Tokyo Institute of Technology, Meguro, Tokyo 152–8552, Japan
[a)] chiguri@psg.cs.titech.ac.jp
[b)] takuo@acm.org

などの形でプログラム検証 [1], [2] に利用されている．この検証は，プログラムの挙動や性質を，関数や述語，定理の形で記述し，証明することにより行われる．証明はプログラムを表現するデータ構造などに関する場合分けや帰納法によって行われるため，プログラムの記述と非常に強く結び付いている．

検証が行われたプログラムでは，記述した性質が成り立っていることから，バグが非常に少ない，場合によっては存在しないと考えられる．したがって，このプログラムに対するバグ修正はほとんど行われることがない．一方で，プログラムの機能強化や変更は別途発生する可能性がある．たとえばプログラムにおけるデータ型のコンストラクタを 1 つ増やすような単純な変更を考えた場合でも，プログラマは検証したプログラムにコンストラクタを追加し，プログラム全体の整合性を確認し，さらに証明を修正する必要がある．

対話的証明支援系におけるプログラムの変更は，簡単な変更の後，全体を処理系に読ませ，エラーが出た箇所を修正する，という流れで行われる．コンストラクタを追加する例では，パターンマッチの網羅性の検査によってエラーが報告される．しかし，このような変更すべき箇所は必ずしもエラーとならず，うまく発見できない箇所も存在する．これはパターンマッチのようなものだけでなく，定理の証明などに関しても同様である．変更すべき箇所が発見されなかったプログラムに対する証明は，仮に証明が完了していたとしても，検証の対象であるプログラムが意図どおりでないため，ある種のバグを残してしまう．このようなバグは規模が大きくなるに従い，他の変更箇所に紛れてしまい，プログラマに気付かれにくくなってしまう．同様の問題は一般のプログラミング言語全般において発生しうるが，新たに追加したデータに関するテストを行うことでバグを発見できる場合も多い．しかし，今回の問題では証明支援系による検証が行われていることにより，プログラマはテストを行わない可能性がある．

この方法の問題は修正箇所の検出を処理系のエラー検出に任せている点にある．プログラマは直接検証済みのプログラムを編集し，処理系に読ませているが，処理系は読み込んだプログラムが修正をしているものであるか，または新規のプログラムであるか判別がつかない．したがって，処理系の判断は全体として整合性がとれているか否かのみとなり，上記の問題が発生する．

プログラムや証明する定理が意図どおりかを機械的に判定することは本質的に不可能である．しかし，処理系が修正であると判別できることで，プログラマに対して情報を提供することが可能となる．例にあげたように単純な修正では，プログラマが意図や誤りに気付くために必要な情報はそれほど多くはない．一方，これらの情報がいっさい得られない場合，プログラマが誤りに気付くことは非常に困難である．我々の目標は，エラー検出より詳細な情報を処理系に提示させることで，プログラマが誤りに気付く可能性を高めることである．

本研究では，定理証明支援系 Coq に対してコンストラクタの追加を可能とするコマンドと，それにともない発生する変更箇所をプログラマに提示，修正する機能を追加した ECoq を実装する．このコマンドにより，処理系はプログラマが行っている変更を知ることができ，プログラム上では補完されてしまう箇所やエラーにならないが密接に関連する箇所もプログラマに提示することができる．しかし，このような情報をプログラム全体について 1 度にプログラマに提示した場合，誤りに気付くための情報がそのほかの情報に埋もれ，気付くことができない可能性がある．ECoq における修正は，プログラマへ提示する情報を修正対象に関連する部分に限定して提示し，対話的に進めることで，プログラマが修正に集中することができるようにしている．このように，プログラマと処理系が対話的にプログラムを変更・修正することから，我々は ECoq に実装した機構を**対話的修正機構**と呼んでいる．対話的修正機構は，Coq が用いる型システムや変更対象の宣言時の状態などを利用して修正箇所を検出するため，Coq の実装を拡張して導入することが必要である．本論文では，Coq に対する対話的修正機構の実現方法を提案し，また実装した ECoq により，通常の変更と読み込みではエラーとして検出されない，しかし修正すべき箇所を検知できることを確認する．

本論文の構成は以下のようになっている．2 章では例題を通してエラーによる修正方法とそれにより発生する検知漏れについて述べる．3 章では，本論文で提案する対話的修正機構の概要と例題に対する対話について説明したのち，修正箇所の検知手法やその妥当性について議論する．4 章では，本論文で提案する対話的修正機構の実装における制限や機構の特徴について述べる．5 章では関連研究との比較を行い，最後に 6 章で本論文のまとめおよび将来課題について検討を行う．

2. 例題

本章では，非常に簡単な手続き型言語である While 言語に対して意味論の定義および簡単な最適化器を作成し，この最適化器が元の意味論を保存することを証明する．その後，言語の構文を拡張し，その際の修正について述べる．

2.1 While 言語の意味論と最適化器

Coq を用いた While 言語の抽象構文は**図 1** のように与えられる[*1]．`Aexp` は算術式を表し，ここでは定数値，変数，加算のみとなっている．`Bexp` はブール式を表しているが，今回の例としては重要ではないため，省略している．

[*1] プログラム全体は web に公開している．http://www.psg.cs.titech.ac.jp/~chiguri/WhileLang_PRO88.v

```
Definition Val := Z.
Definition Env := Var->Val.

Inductive Aexp : Set :=
| Aval  : Val->Aexp
| Avar  : Var->Aexp
| Aadd  : Aexp->Aexp->Aexp.

Inductive Bexp : Set := ... (* omitted *)

Inductive Stm : Set :=
| Sass   : Var->Aexp->Stm
| Sskip  : Stm
| Sif    : Bexp->Stm->Stm->Stm
| Swhile : Bexp->Stm->Stm
| Scomp  : Stm->Stm->Stm
```

図 1　Coq による While 言語の抽象構文
Fig. 1　The abstract syntax of While language in Coq (part).

Stm は While 言語における文を表しており，変数への代入や条件分岐，繰返しが存在する．

While 言語の意味論は**図 2** のように記述される．算術式，ブール式，文それぞれの意味は Aeval，Beval，NS によってつけられる．これらはすべて big-step semantics として定義されているが，前 2 者は Coq の関数として定義されている一方，後者は述語として定義されている．これは，Coq が終了しない関数を記述できないため，無限ループを記述することができる While 言語の意味論を陽に記述することができないことによる．なお，NS は自然意味論であり，終了するプログラムに対してのみ意味づけすることができる．

次に，この言語に対する最適化器を導入する．最適化器の対象とするものは，状態に対して不変な要素の簡約・排除である．ここでは，算術式に関する最適化として，定数どうしの加算を事前に計算しておき，その結果と置き換えることを行う．また，文に関する最適化として，条件式にあたるブール式がつねに真（または偽）に定まるような条件分岐を対応する分岐先の文に置き換える，つねに偽に定まる繰返し文を skip 文に置き換える，ということを行う．このような最適化器を**図 3** のように記述した．なお，ブール式に関する最適化器は省略するが，基本的な内容は算術式のそれと同じである．

この最適化器の正しさを保証するために，算術式やブール式については最適化の前後で意味が変わらないことを，文については最適化前の結果が最適化後も保たれることを証明した．これらの定理は**図 4** のように記述される．今回は算術式に対する証明を与え，ブール式については全体を省略，文については証明の最初の分岐についてのみ言及している．算術式の最適化器の検証では式に対する構造帰納法を用い，変数や定数の場合の自明なケースを自動証明（intuition）に任せ，加算に関してのみ明示的に帰納法の仮定を利用して書き換えを行っている．また，文の最適化器の検証では意味論の導出に関する帰納法（induction）

```
Fixpoint Aeval (a : Aexp) (e : Env) := match a with
  | Aval z => z
  | Avar v => e v
  | Aadd a1 a2 => ((Aeval a1 e) + (Aeval a2 e))%Z
  end.

Fixpoint Beval (b : Bexp) (e : Env) := ...
  (* omitted *)

Definition ExtEnv (v : Var) (z : Z) (e : Env) :=
  fun x : Var => if (VarEq x v) then z else e x.

Inductive NS  : Stm->Env->Env->Prop :=
| NSass     : forall v a e,
   NS (Sass v a) e (ExtEnv v (Aeval a e) e)
| NSskip    : forall e, NS Sskip e e
| NSif_T    : forall b S1 S2 e1 e2,
   Beval b e1 = true  -> NS S1 e1 e2
                      -> NS (Sif b S1 S2) e1 e2
| NSif_F    : forall b S1 S2 e1 e2,
   Beval b e1 = false -> NS S2 e1 e2
                      -> NS (Sif b S1 S2) e1 e2
| NSwhile_T : forall b S e1 e2 e3,
   Beval b e1 = true -> NS S e1 e2
                     -> NS (Swhile b S) e2 e3
                     -> NS (Swhile b S) e1 e3
| NSwhile_F : forall b S e,
   Beval b e = false -> NS (Swhile b S) e e
| NScomp    : forall S1 S2 e1 e2 e3,
   NS S1 e1 e2 -> NS S2 e2 e3
               -> NS (Scomp S1 S2) e1 e3.
```

図 2　Coq による While 言語の意味論
Fig. 2　The semantics of While language in Coq.

```
Fixpoint optimize_A (a : Aexp) := match a with
  | Aadd a1 a2 =>
    match (optimize_A a1, optimize_A a2) with
    | (Aval z1, Aval z2) => Aval (z1+z2)%Z
    | (a1', a2') => Aadd a1' a2'
    end
  | _ => a
  end.

Fixpoint optimize_B (b : Bexp) := ... (* omitted *)

Fixpoint optimize (S : Stm) := match S with
  | Sass x a => Sass x (optimize_A a)
  | Sif b S1 S2 => match optimize_B b with
    | Btrue => optimize S1
    | Bfalse => optimize S2
    | b' => Sif b' (optimize S1) (optimize S2)
    end
  | Swhile b S' => match optimize_B b with
    | Bfalse => Sskip
    | b' => Swhile b' (optimize S')
    end
  | Scomp S1 S2 =>
    match (optimize S1, optimize S2) with
    | (Sskip, S2') => S2'
    | (S1', Sskip) => S1'
    | (S1', S2') => Scomp S1' S2'
    end
  | _ => S
end.
```

図 3　While 言語の最適化器
Fig. 3　The optimizer for While language.

```
Lemma optimize_A_keep_result :
  forall (a : Aexp) (e : Env),
    Aeval a e = Aeval (optimize_A a) e.
Proof.
 induction a; intuition; simpl in *;
  rewrite IHa1; rewrite IHa2;
   case (optimize_A a1); case (optimize_A a2); auto.
 Qed.

Lemma optimize_B_keep_result :
  forall (b : Bexp) (e : Env),
    Beval b e = Beval (optimize_B b) e.
 ... (* omitted for brevity *)
 Qed.

Theorem optimize_keep_semantics :
  forall (S : Stm) (e1 e2 : Env),
    NS S e1 e2 -> NS (optimize S) e1 e2.
Proof.
 intros S e1 e2 H; induction H;
  try (rewrite optimize_A_keep_result);
   try constructor; auto; simpl.
 ... (* omitted for brevity *)
 Qed.
```

図 4　最適化器の性質と証明（一部）

Fig. 4　The specifications for the optimizer and their proofs (abridged).

を用いている．

2.2　演算子および文の追加と修正

図 1 の構文は必要最小限度のものであるため，実際にはより多くの演算子や文があることが望ましい．そこで，算術演算子として乗算を，文として repeat-until 文を導入する．まず，構文を表す型に対応するコンストラクタを追加する．それぞれ，ここでは `Amult` および `Srepeat` とする．本来のプログラムの内容から，これらのコンストラクタを追加した後，いくつかの修正をしなければならない．修正箇所としては乗算の意味論，repeat-until 文の意味論，それぞれの最適化器や定理の証明がある．**図 5** に本来行われるべき修正を示す．プログラマは，これらの修正箇所を何らかの形で知る必要がある．

最も一般的な修正箇所の探索方法は，コンストラクタを追加したプログラムを Coq に読み込ませ，エラーが出た箇所を見ていくものである．たとえば，算術式の評価関数である `Aeval` はパターンマッチにおいて乗算に関するパターンが存在しないため，定義箇所において以下のようなエラーを出力する．

```
Error: Non exhaustive pattern-matching:
  no clause found for pattern Amult _ _
```

プログラマは，このエラーに基づいて乗算の意味論を追加する．

一方で，この手法における問題として，エラーが発生しない箇所の存在があげられる．まず，文の意味論を表している述語 `NS` は定義上 repeat-until 文に関する意味論を持っていないが，述語の定義は全域性の判定が行われないため，エラーが発生しない．述語の性質上このような記述をエラーとして検出することは不可能であり，プログラマが見つけなければならない．

```
Inductive Aexp : Set := ...
| Amult : Aexp->Aexp->Aexp.
Inductive Stm : Set := ...
| Srepeat : Stm->Bexp->Stm.

Fixpoint Aeval (a : Aexp) (e : Env) := match a with
  ...
  | Amult a1 a2 => ((Aeval a1 e) * (Aeval a2 e))%Z
  end.
Inductive NS  : Stm->Env->Env->Prop := ...
| NSrepeat_T : forall b S e1 e2,
   NS S e1 e2 -> Beval b e2 = true
              -> NS (Srepeat S b) e1 e2
| NSrepeat_F : forall b S e1 e2 e3,
   NS S e1 e2 -> Beval b e2 = false
              -> NS (Srepeat S b) e2 e3
              -> NS (Srepeat S b) e1 e3.

Fixpoint optimize_A (a : Aexp) := match a with
  ...
  | Amult a1 a2 =>
    match (optimize_A a1, optimize_A a2) with
    | (Aval z1, Aval z2) => Aval (z1*z2)%Z
    | (a1', a2') => Amult a1' a2'
    end
  | _ => a
  end.

Fixpoint optimize (S : Stm) := match S with
  ...
  | Srepeat S' b => match optimize_B b with
    | Btrue => optimize S'
    | b' => Srepeat (optimize S') b'
    end
  | _ => S
  end.
```

図 5　While 言語の追加と修正箇所．図 1 や図 2，図 3 と同じ箇所は ... で省略している

Fig. 5　Extensions and modifications of While language (abridged).

次に，算術式用の最適化器である `optimize_A` もエラーを発生しない．算術式の最適化器は定義上加算**以外のすべての場合で**元の値を返すため，乗算の際もそのままを返してしまう．これは最適化器の意図と必ずしも一致せず，むしろ最適化の漏れであると考えられる．しかしながら，記述上乗算の場合もパターンマッチにおけるワイルドカードパターン[*2]によって補完されてしまうため，エラーとして報告されず，やはりプログラマ自身が見つけなければならない．同様の理由により，`optimize` もエラーを発生しない．

最後に，証明に関してもエラーが発生しない場合がある．証明を短く記述しようとした場合，比較的自明な証明をまとめて記述することもあり，このような記述が変更を吸収してしまうことが考えられる．たとえば，

[*2] アンダースコアによるパターンマッチ．すべてのコンストラクタと合致する．

optimize_A_keep_result における証明は，加算の場合以外の証明は自動証明に任せている．最適化器は（上の問題により）乗算も元と同じ値を返すため，定数や変数の場合同様自明であり，この自動証明によって補完される．また，文に対する最適化器は，意味論中に追加した文が存在しないため，帰納法の場合分けに現れず，やはり証明は終了する．

一般に証明においてエラーが発生せずに終了することは問題ではないが，しかし他の問題と重なった場合に問題となる．証明は，関数などの定義でエラーが発生しなかった場合にプログラマが修正箇所に気付く機会となるが，しかしエラーが出なければプログラマが気付く機会は得られず，意図とは異なる結果のまま証明が完了してしまう．

以上の問題の多くは，記述方法による補完が原因であり，この問題が発生しないようにプログラムを記述することは十分に可能である．しかし，これらの機能そのものは非常に有用であり，使用しなければ証明が非常に大きくなる可能性もある．図 4 を例にとると，全体を提示した optimize_A_keep_result では，induction の後の intuition を Aexp のコンストラクタごとの証明で記述する必要がある．また，optimize_keep_semantics の証明ではブール式の最適化である optimize_B の結果によって分岐する際に，5 つのタクティクを連結させたものによって，6 つのサブゴールのうち 4 つのサブゴールが証明できた．このような分岐は証明時に 4 回出現したため，この連結したタクティクは証明に補完を用いないように記述すると，証明に 16 回出現した．我々の行った証明では，この 4 回の出現ごとに連結したタクティクによって証明したため，出現の回数を 4 回に抑えることができた．このように，補完を利用しない場合何度も同じようなタクティクの列が出現し，証明が長くなってしまう．

述語の定義に関しては，存在する記述が元々正しく，また網羅性の判定などが存在しないため，コンストラクタの追加による影響を受けにくい．したがって，述語の定義において型エラーが発生する可能性は少なく，エラーを頼りに修正箇所を見つけることはできない．

3. 対話的修正機構

前章における問題は，読み込んだプログラムが変更したものである点を処理系が考慮しないことで発生している．我々は，処理系を通して変更を行うことで，必要となる変更箇所を検出できると考えた．この手法では，変更とそれにともなう修正を対話的に行うため，支援系に組み込む機構を対話的修正機構と呼ぶ．また，Coq に対話的修正機構を組み込んだものを ECoq と呼ぶ．

本章では，例題を通して対話的修正がどのように行われるかを述べた後，修正箇所として提示される箇所をどのように検知するかについて述べる．また，この検出に関して，Coq の用いる predicative Calculus of Inductive Constructions（以下 pCIC）の型システムにおける妥当性について説明する．なお，Coq への実装における問題に関しては 4 章で議論する．

3.1 修正の流れ

対話的修正機構による修正は，以下の流れで行われる．

(1) プログラマはコマンドによって帰納型へ新たなコンストラクタを追加する．
(2) 対話的修正機構はコンストラクタを追加できるかを判定し，問題があればプログラマに通知する．
(3) プログラマはコンストラクタの追加を任意の回数行う．
(4) 対話的修正機構はコンストラクタの追加が終了した時点で，影響を受けた要素を列挙し，プログラマに提示する．
(5) プログラマは提示された箇所に対して必要となるコードを記述する．上の提示と修正を繰り返す．
(6) 提示される箇所がなくなると修正は完了となる．

(2) における追加の判定は，コンストラクタを追加した帰納型が pCIC の型として妥当であるかについて行う．また，そのほかにも帰納型が満たすべき性質が存在するが，この詳細は 3.3 節で述べる．

実際に ECoq において修正を行ったコードは**図 6** のようになる．ECoq は，追加のためのコマンドである Extend Inductive コマンド（1–2 行目，3–4 行目，5–12 行目）を備えており，プログラマはこのコマンドによってコンストラクタを追加する．なお，このコマンドの入力時点では関数や証明などで修正が行われるべき場所を出力することはない．

3–4 行目のコマンドでは文への repeat-until 文の追加を行っている．この結果，ECoq は以下のような出力を行う．

```
Stm is extended.
You may extend the following types : NS
```

```
1  Extend Inductive Aexp : Set :=
2  | Amult : Aexp->Aexp->Aexp.
3  Extend Inductive Stm : Set :=
4  | Srepeat : Stm->Bexp->Stm.
5  Extend Inductive NS : Stm->Env->Env->Prop :=
6  | NSrepeat_T : forall S b e1 e2,
7     NS S e1 e2 -> Beval b e2 = true
8                -> NS (Srepeat S b) e1 e2
9  | NSrepeat_F : forall S b e1 e2 e3,
10    NS S e1 e2 -> Beval b e2 = false
11               -> NS (Srepeat S b) e2 e3
12               -> NS (Srepeat S b) e1 e3.
13 Deploy.
14 (* Extensions for Aeval *)
15 exact ((Aeval a0 e) * (Aeval a1 e))%Z.
16 Defined.
17 (* Extensions for optimize_A *)
18 ...
```

図 6　ECoq を用いた構文拡張用のコード
Fig. 6　Syntactic extension code in ECoq.

2 行目の出力は，NS が Stm に依存しているため，プログラマに修正を提案している．この出力は提案にすぎないため，プログラマが必要ないと判断すれば無視して問題ない．今回は NS にも拡張が必要であるため，5 行目から 12 行目のように拡張した．

この拡張により，想定していたコンストラクタの追加が完了するので，Deploy コマンド（13 行目）により修正を展開する．この展開によって初めて，ECoq は行われた修正の影響を受けた箇所が存在するかを要素の定義順に確認する．今回の例では，まず Aeval が拡張した Aexp を展開しているため，この展開が検知される．この展開は以下のように表示される．

```
In Aeval (1 destructors)
fix Aeval (a : Aexp) (e : Env) {struct a} : Val :=
  match a with
  ...
  | Amult a0 a1 => _
    ^^^^^^^^^^^^^^^^
  end
```

このように，拡張する必要のある箇所を，その文脈とともに表示する．文脈は直接関係のある部分だけであり，他のパターンに関しては出力されない．これは，他のパターンが大きな項となっていたときに，修正箇所が一見して分からなくなってしまうためである．

ECoq が影響があると判断する箇所は，ワイルドカードパターンなどを考慮せず，拡張した型を展開するパターンマッチの存在する箇所すべてとなっている．そのため，2.2 節におけるエラーの検出結果と異なり，optimize_A や optimize などにおけるパターンマッチも同様に検知される．

修正箇所が検知されると，ECoq はその場所に入るべき値，すなわちパターンマッチの返す値を要求する．上の例では，Amult の場合に返す値（この場合は Val 型の値）である．Coq では，証明用の対話によって関数を構築することができるため，この要求に対して証明のように記述を行う．そのため，ECoq は上の出力に続き，以下の出力を行う．

```
1 subgoal

  Aeval : Aexp -> Env -> Val
  a : Aexp
  e : Env
  a0 : Aexp
  a1 : Aexp
  ============================(1/1)
   Val
```

図 6 では 15 行目の exact タクティクによって掛け算の評価方法を記述している．なお，証明による関数の構築はあまり一般的ではないが，図 6 で用いている exact や apply などのタクティクを利用することで通常の関数に近い記述が可能であるため，それほどプログラマへの負担は大きくない．

Aeval の補完が終了することで，次の修正箇所の検出と補完が行われる．すべての要素に対して処理が終了すると，プログラムと証明の修正が完了となる．

3.2　修正箇所検知手法

本論文で提案する対話的修正機構は，主にコンストラクタの追加を可能とする．コンストラクタの追加により，pCIC の型システムにおいていくつかの問題が発生するが，本質的にプログラマに処理をゆだねる必要がある点はパターンマッチのパターンが不足する点のみである．ここではどのような処理によって不足箇所を発見するかについてのみ説明し，検出箇所に関する妥当性については 3.3 節で議論する．

3.2.1　記述の定式化

Coq においてプログラマが記述する要素は大別すると，帰納型の宣言，項の宣言，証明に分かれる．この中で，証明は記述のうえではタクティクの列で構成されているが，タクティクは内部的に項を作り上げており，完了した証明は項として定義される[*3]．本手法では，証明に関してタクティクの列ではなく作成された項を用いる．したがって，以降の定式化では証明について陽に扱うことはせず，項と同一の扱いをする．

本手法では，項や帰納型に対して型検査を行う必要があるため，ある時点で宣言された項や帰納型などの環境に関する情報が必要である．したがって，定式化のために，大域変数や帰納型の宣言を環境の要素として定義する．まず，pCIC の項[*4]と帰納型の定義を以下のように表す．

$$
\begin{array}{lcll}
T & ::= & \texttt{Set} & \text{(Set ソート)} \\
 & | & \texttt{Prop} & \text{(Prop ソート)} \\
 & | & \texttt{Type} & \text{(Type ソート)} \\
 & | & x & \text{(変数)} \\
 & | & (T\ T) & \text{(関数適用)} \\
 & | & \texttt{fun}\ x : T \texttt{ => } T & \text{(関数抽象)} \\
 & | & \forall x : T,\ T & \text{(全称限量)} \\
 & | & \texttt{let}\ x := T\ \texttt{in}\ T & \text{(let 式)} \\
 & | & I & \text{(帰納型)} \\
 & | & C & \text{(コンストラクタ)} \\
 & | & \texttt{match}\ T\ \texttt{return}\ T\ \texttt{with} & \\
 & & \quad (\ |\ C\ x^* \texttt{ => } T\ )^*\ \texttt{end} & \\
 & & & \text{(パターンマッチ)} \\
\mathit{Ind} & ::= & I\ (x\ :\ T)^* : T := (\ |\ C\ (x : T)^* : T\ )^* & \\
 & & & \text{(帰納型の宣言)}
\end{array}
$$

なお，パターンマッチの return の次にある項はパターンマッチの返す型である．これまで記述した関数の中では

[*3] Coq の用いている Curry-Howard 対応による．
[*4] 再帰に用いられる fix を省略しているが，再帰のガードコンディション以外はほぼ関数抽象と同じ扱いであるため，ここでは省略している．

省略されていたが，Coq 内部ではこの型をつねに持っており，また ECoq における処理では非常に重要なため明示している．

宣言された大域変数とその型（および実体）の集合を Γ_{Var} とする．この各要素を $x : T$（実体のない変数の場合）または $x := T : T$（実体のある変数の場合）と表す．また，帰納型の宣言の集合を Γ_{Ind} で表し，その要素は *Ind* で表されるとする．同様に，帰納型の拡張の宣言の集合を Γ_{Ext} とし，その要素を帰納型の宣言同様 *Ind* で表す．なお，型や変数の宣言の順序が定まっていないが，適切な（プログラマが実際に宣言した）順序で宣言されているように扱い，拡張の宣言に現れる帰納型はすべて帰納型の宣言に現れており，パラメータなどの不整合は起こらないものとする．

3.2.2　述語の依存性

3.1 節の例では，文の拡張時に意味論の拡張を提案していた．経験的に，述語にパラメータとして存在する型は，その構造をある程度述語に反映していることが多い．たとえば `NS` は `Stm` のそれぞれのコンストラクタについて言及しており，非常に密接に関係している．このような述語はコンストラクタの追加時に必ずしも拡張されるわけではないが，しかし可能性は高いため，提案という形で情報を提示することにした．

実装では，拡張した型をパラメータに持つ帰納型をすべて提案することでこの機能を実現している．なお，ここでいうパラメータとはコンストラクタの持つパラメータではなく，型自体についているパラメータである．したがって，`Aexp` をパラメータに持っているコンストラクタを持つ `Stm` などは，`Aexp` の拡張時に検出されることはない．

より厳密には，型 I' を拡張した際に，Γ_{Ind} に含まれるすべての帰納型の宣言 $I\ (v_1 : T_1)\ \ldots\ (v_n : T_n) : T := \ldots$ について，T_1 から T_n のどれか，または T が I' を含んでいる場合（かつ I 自身が I' でないとき）に，I が I' の影響を受ける可能性があるとしてプログラマに提示する．この提示方法は非常にアドホックであるため，多くの実例を通して妥当性を検討し，また提示の漏れがあればより精度の良い提示手法を研究する必要がある．

3.2.3　項の修正箇所

コンストラクタの追加に対して拡張する必要がある箇所は，パターンマッチにおけるパターンである．したがって，パターンマッチが出現しうる箇所をすべて走査する必要がある．Coq では，変数の型などにパターンマッチが使用できるため，出現しうる位置は関数の宣言などに限定されない．

以上の点から，ECoq では環境 Γ_{Ind}，Γ_{Var}，Γ_{Ext} について，次の手続きによって検出対象を決定する．

- Γ_{Ind} に含まれるすべての帰納型の宣言
  $I\ (x_1 : T_1)\ \ldots\ (x_n : T_n) : T :=$
  $\quad \mid C_1\ (x_{C_1} : T_{C_1})\ \ldots\ (x_{C_m} : T_{C_m}) : T_1 \mid \ldots$ に対し，すべての項（$T_1, \ldots, T_n$，T，$T_{C_1}, \ldots, T_{C_m}$，$T_1, \ldots$）を対象として次に示す検出を行う．
- Γ_{Var} に含まれるすべての $x : T$ に対し，T を対象として次に示す検出を行う．
- Γ_{Var} に含まれるすべての $x := T_1 : T_2$ に対し，T_1 と T_2 を対象として次に示す検出を行う．

これらの検出対象に対し，項の構造に対して再帰的に検出を行う．なお，検出および修正は型と項を同時に検出対象とする場合，拡張した結果を用いて項を検出対象とする．これは依存型にパターンマッチが存在する場合，項の側が型をつけられない可能性があるためである．具体的な検出アルゴリズムは次のとおりである．

- T が変数，ソート，帰納型，コンストラクタそのものの場合，これらは拡張対象ではなく，また部分項もないので検出を終了する．
- T が関数適用 $(T_1\ T_2)$ の場合，関数抽象 $\mathtt{fun}\, x : T_1 \Rightarrow T_2$ の場合，全称限量 $\forall\, x : T_1, T_2$ の場合，および let 式 $\mathtt{let}\ x := T_1\ \mathtt{in}\ T_2$ の場合，T_1 と T_2 それぞれに対して再帰的に検出を行う
- T がパターンマッチ
  $\mathtt{match}\ T'\ \mathtt{return}\ T_r\ \mathtt{with}$
  $\quad C_1\ \ldots \Rightarrow T_1 \mid \ldots \mid C_n\ \ldots \Rightarrow T_n \mathtt{end}$ の場合，
  - T'，T_r，および T_1 から T_n のそれぞれに対し，検出を再帰的に行う．
  - T' の型が帰納型 I であり，I の宣言が Γ_{Ext} に入っている場合，追加されたコンストラクタを C' とすると，`with` 以降のパターンとして C' に関するパターンを追加し，その結果として **$\mathbf{T_r}$ を拡張した型**の値をプログラマに要求する．

このように，処理自体は非常に単純であり，実装も比較的容易である．この検出方法の問題点については 4.3 節で述べる．

3.3　検出箇所の妥当性

本論文で用いる対話的修正機構では，以下の 2 つを前提として考えることで，型システム上の問題を単純化している．

- 帰納型や関数などの型の不変性．追加されたコンストラクタを除き，すべての要素は同じ型がつけられる．
- 既存要素に対する関数の結果の不変性．追加されたコンストラクタを含まないデータに対し，すべての関数は変更前と同じ結果を返す．

この結果，追加されたコンストラクタ以外のすべてにおいて，既存の型付け規則とまったく同じ規則・構成によって型をつけることができる．一方，型付け規則においてコンストラクタが出現する場所では，既存の型付け規則と整合性がとれるかを確認する必要がある．

本論文における対話的修正機構では，コンストラクタの

追加およびパターンマッチにおけるパターンの追加を行う．したがって，前述の型の不変性の仮定から，多くの構造は拡張後も拡張前に用いた型付け規則と同じもので型付けを行うことができる．より形式的には，項に関して以下の性質が成り立つ．

性質 1　ある帰納型の環境 Γ_{Ind} と変数の環境 Γ_{Var} によって妥当な型 T_2 がつけられる項 T_1 に対し，Γ_{Ind} を拡張する Γ_{Ext} を用いて，3.2.3 項で定義したアルゴリズムを用いて T_1 を T_1' に拡張したとき，T_2 を拡張した T_2' が存在し，拡張した環境で T_1' に T_2' という型がつく．

上記の性質は項 T_1 の拡張前の型付けに関する帰納法を用いることで，帰納型とパターンマッチの型付けに関して以外は容易に証明できる．pCIC では，帰納型とパターンマッチを除くと項の構造と使用する型付け規則がほぼ対応するためである．一方，帰納型とパターンマッチについてはいくつかの例外的な規則があり，それらに対して制限を設ける必要がある．以降では，これらの型規則が上記の性質を満たすために必要な制限について述べる．これらの制限は検出アルゴリズムの制限ではなく，**コンストラクタの追加に関する制限**である．

3.3.1　パターンマッチの型

pCIC におけるパターンマッチの一般的な型付け規則は，通常の関数型言語における型システムと同様に，パターンマッチの展開する項が帰納型 I に属し，それぞれのコンストラクタのパターンで型 T を返すときに，パターンマッチ全体は型 T に属するというものである．したがって，追加したパターンの結果が型 T の値を返す限り，拡張後の型も T としてつけることができる．

一方で，pCIC では Prop に属する型（以下 Prop 型と呼ぶ）を展開するパターンマッチは Prop ヒエラルキの要素を返さなければならない，という制約が存在する．なお，Prop ヒエラルキに属する要素は，Prop 自身，Prop 型，およびそのような帰納型のコンストラクタである．この制約は Coq の関数を他のプログラミング言語の関数に変換（Extraction）するために存在する．

Coq からの変換時，Coq は Prop ヒエラルキに属する定理や証明といった要素を消去し，Set や Type に属する要素のみを残す．その結果，パターンマッチが除去されてしまい，結果が不定になってしまう場合がある．たとえば以下のコードを考える．

```
fun (t : or P Q) =>
  match t with
  | or_introl _ => 0
  | or_intror _ => 1
  end
```

ここで，or は Prop->Prop->Prop という型を持ち，P，Q はともに Prop 型の要素であるとする．変換後のプログラムでは，t が Prop ヒエラルキに属するため消去されるため，このパターンマッチ自体が消えてしまう．しかし，0 や 1 は nat 型に属し，これは Set に属するため，変換時に消去されない．結果として，変換後のプログラムは存在しない引数で結果を変えていることとなる．このようなプログラムは明らかに実現不可能であるため，変換を考慮している Coq のシステムでは，パターンマッチが Prop 型を展開する場合 Prop ヒエラルキに属する要素を返さなければならない．

この制約自体は，コンストラクタを追加することやパターンマッチのパターンを追加するだけが可能な対話的修正機構による影響を受けない．しかし，この制約には例外が存在し，こちらの影響が問題となる．

あるパターンマッチが Prop 型を展開する場合でも，その型が Empty definition や Singleton definition と呼ばれる定義である場合，Prop ヒエラルキに属しない要素を返すことが許される．Empty definition とはコンストラクタを持たない帰納型の定義であり，また Singleton definition とは，単一のコンストラクタを持ち，かつコンストラクタのパラメータがすべて Prop のヒエラルキに属する帰納型の定義である．これらの型を展開する場合では，パターンマッチにより現れるすべてのパラメータは変換時に無視され，結果としてそのパターンマッチによって生成された要素は変換後の結果に寄与しない．そのため，変換時にパターンマッチそのものを消去することで変換が可能となる．たとえば，eq は Singleton definition であるため，以下の関数は変換可能であり，記述が許される．

```
Definition f (t : eq x) : nat :=
  match t with
  | refl_equal => 0
  end.
```

型 eq の唯一のコンストラクタである refl_equal はパラメータを持たないため，t はパターンマッチの結果に寄与していない．このスクリプトを OCaml のプログラムへ変換すると以下のコードになる．

```
let f _ = 0
```

これらの例外はコンストラクタの数に依存しているため，コンストラクタを増やした場合に影響がある．たとえば Singleton definition である型を拡張した場合，その型が singleton でなくなってしまうため，結果として上にあげた例外的なパターンマッチが記述できない場合が考えられる．対話的修正機構では，このような問題を回避するために，パターンマッチの可能なヒエラルキが保存されているかを確認する．具体的には，Singleton definition である型の拡張を許さず，また Empty definition である型は Singleton definition となる範囲までに制限している[*5]．

3.3.2　帰納型とソート多相

帰納型はパラメータを受け取ると，結果としてソート

[*5] 実装では，パターンマッチの可能な範囲が各型に保存されているため，容易に確かめることができる．

（Set，Prop または Type）に属する．このソートを帰納型の結果のソートと呼ぶことにする．また，パラメータが残っている状態の型をアリティと呼ぶ．

コンストラクタの型については，対応する帰納型が妥当な型であれば帰納型がつく．また，帰納型については通常宣言時のアリティが対応する．しかし，帰納型についてはソート多相という特殊な規則がある．

ソート多相とは帰納型における結果のソートをできる限り小さくするものである．ソートは Prop，Set，Type の順に大きいと見なされる[*6]．小さなソートは conversion rule によってより大きなソートとして扱えるため，結果のソートは基本的に小さい方がより適用可能な範囲が広く，有用である．

Coq におけるソート多相は結果のソートが Type である帰納型にのみ適用される[*7]．このソート多相は以下の手順で決定される．この帰納型が，empty または singleton definition として見られる場合，この結果のソートを Prop と見なす．そうでない場合に，もしも Set として見られるほど小さい場合，結果のソートを Set と見なす．どちらにもあてはまらない場合は，定義どおりに Type と見なす．

ソート多相はパラメータによっても発生する．この点においても，Coq では Type 型および結果が Type 型である全称型のパラメータに限定している．この典型的な例としてはライブラリに存在する list があげられる．Coq の List モジュールにおける list の定義は以下のとおりである．

```
Inductive list (A : Type) : Type :=
| nil : list A
| cons : A -> list A -> list A.
```

この定義では，list は Type->Type として定義されている．しかし，パラメータ A には conversion rule によって Set や Prop の要素も与えられる．たとえば，list nat は正しい型であり，自然数のリストを意味する．

このとき，本来であれば nat が Type として扱われ，list nat は Type に属するはずである．しかし，ソート多相はこれを Set として扱うことを許す．これは list の定義においてパラメータ A を nat に置き換えたときに，list の大きさが Set の要素となる程度に小さいためである．

しかし，コンストラクタの追加によって，この sort polymorphism の挙動が変更される可能性がある．たとえば，次のコンストラクタを list に追加してみる．

```
ins : forall B : Type, B -> list A -> list A
```

このコンストラクタは，list に任意の型の要素を入れることを可能にする．しかし，このコンストラクタを list に追加した場合，たとえば list nat は自然数のリストだったが，コンストラクタ ins はあらゆる要素をリストに入れることを許しており，ins の型はつねに Type に属する．このため，ins のコンストラクタを追加した list nat は Type として扱われる．

議論の複雑化を避けるためにも型の保存は必要であり，したがってソート多相の挙動が変化する拡張は許容できない．そのため，コンストラクタの追加によってソート多相が保存されているかをつねに確かめなければならない．幸い，Coq 内部においてソート多相は，型に存在するどのパラメータに依存してソートが変わるかを保存しており，使用時はその中の最も小さいソートとするように実装されている．したがって，この依存性を拡張前の定義と比較することでソート多相が保存されているかを容易に確かめることができる．

4. 実装

前章の機構を，Coq8.3pl2 に対して実装した．本章では，前章において説明した手法における実装上の制約と問題について述べる．

4.1 帰納原理関数

Coq では，inductive type を宣言した際に，同時にいくつかの関数が作成される．たとえば図 1 の Aexp の場合，Aexp_ind，Aexp_rec，Aexp_rect という関数が作られる．ここでは，これらの関数を**帰納原理関数**[*8]と呼ぶ．これらの関数は，構造帰納法を表現するために用いられており，それぞれ Prop，Set，Type を返す関数である．Aexp_ind の型は**図 7** のようになっている[*9]．パラメータ P は証明する述語を表しており，その後に続く 3 つの引数は構造帰納法の証明を表す．これらはそれぞれ順に定数に対する証明，変数に対する証明，加算に対する証明となっている．

このことから分かるとおり，帰納原理関数の引数は，コンストラクタの数に依存している．追加された引数は，パターンマッチにおけるパターンと同様に補完を必要とする．しかし，より深刻な問題として，拡張によって型が保存されていない点があげられる．たとえば Aexp に乗算を表す Amult を導入したとき，Aexp_ind は**図 8** のようになる．図 7 と比べると 7–8 行目の引数が増えており，Aexp_ind

```
Aexp_ind :
 forall (P : Aexp -> Prop),
  (forall v : Val, P (Aval v)) ->
  (forall v : Var, P (Avar v)) ->
  (forall a : Aexp, P a ->
    forall a0 : Aexp, P a0 -> P (Aadd a a0)) ->
  forall a : Aexp, P a
```

図 7　Aexp の帰納原理関数の 1 つである Aexp_ind の型
Fig. 7　The type of Aexp_ind, one of induction schemes for Aexp.

[*6] Coq8.2 までは Prop と Set に大小関係は存在しなかったが，8.3 から Prop は Set より小さくなった．
[*7] pCIC では結果のソートが Set であってもソート多相の対象となりうるが，ここでは Coq の実装の影響を考えるために Coq 側の手法についてのみ考える．
[*8] Elimination principle，induction scheme などとも呼ばれる．
[*9] 他の 2 つは Prop を Set や Type に置き換えたものである．

```
Aexp_ind :
 forall (P : Aexp -> Prop),
  (forall v : Val, P (Aval v)) ->
  (forall v : Var, P (Avar v)) ->
  (forall a : Aexp, P a ->
    forall a0 : Aexp, P a0 -> P (Aadd a a0)) ->
  (forall a : Aexp, P a ->
    forall a0 : Aexp, P a0 -> P (Amult a a0)) ->
  forall a : Aexp, P a
```

図 8　Aexp 拡張後の Aexp_ind の型
Fig. 8　The type of Aexp_ind for extended Aexp.

の型が変わっていることが分かる．

この型の変化は，前章の手法で仮定した型の不変性に反するため，本研究ではこのような関数の使用方法を強く制限する．方針として，帰納原理関数をパターンマッチと同様に扱うことで，3.2 節で議論した手法をそのまま使うようにする．

具体的には，すべての帰納原理関数の出現について，必ず引数がすべて与えられている状態のみを許すことにする．この場合，必要な修正は追加されたコンストラクタのケースに相当する引数が不足するのみであり，したがってパターンマッチのパターンが不足することと同一視することができる．実際に，帰納法で用いる場合にはこの制限を満たしているため問題なく，またそのほかの帰納原理関数の利用はそれほど多くないため，広い範囲でこの制限は問題とならない．

これに従って 3.2.3 項で導入したアルゴリズムを拡張すると，次のようになる．

- 対象とする項 T が変数である場合で，かつそれが拡張した型の帰納原理関数である場合（I_ind，I_rec，I_rect のいずれかの名前），
  - 関数の引数の数と直接適用されている引数の数を比較し，既存のコンストラクタ用の引数がすべて渡されている場合に，追加したコンストラクタ用の引数を追加する．要求する型は拡張された帰納原理関数の型に合わせたものとなる．
  - 引数の数が足りない場合，拡張全体を中止する．
- そのほかの場合は，3.2.3 項におけるアルゴリズムと同様に動く．

このように，制限を満たせない場合は拡張そのものを中止するようにした．拡張そのものをアドホックに行うことで全体の整合性を保つことも考えられるが，機械的に修正箇所を見つけることが難しくなるため，ここでは考慮しない．

帰納原理関数（またはそれに相当する関数）はプログラマが独自に定義することもできる．しかし，上にあげたとおり，帰納原理関数において引数の追加はほぼ必須であり，またどの関数が帰納原理関数として用いられるかを機械的に判別することは難しい．さらに，現在の対話的修正機構が引数の追加をサポートしていないため，プログラマが明示的に拡張することも難しい．したがって，プログラマ定義の帰納原理関数については現在の対話的修正機構ではサポートしないが，引数の追加そのものが多くの拡張に見られるので，サポートは今後の課題とする．

4.2　手順の制約

3 章では順序や対象についての制限をいっさいあげなかった．これは pCIC がモジュールなどの概念を取り扱わないためである．しかし，Coq の実装ではこれらの概念が名前空間などの形で扱われている．

Coq では，名前空間ごとに同名の型をいくつか定義することができる．そのため，型名のみではどの型を追加しているのか判別が難しく，また場合によっては参照する型が本来のモジュール内で参照できない場合もある．このようなことを防ぐために，今回の実装では，現在編集しているモジュール（ない場合はトップレベル）の定義のみを修正可能とした．本手法は既存のモジュールの拡張を行うことを目標としているため，この制約は大きな障壁となるが，制約を解消した実装は将来の課題とする．

修正作業の順序の制約としては，修正作業中の新しい拡張の禁止があげられる．これは修正の順序が前後すると，修正が必要となる箇所の検知，および再構築する環境が複雑化していくためである．また，作業中の時点で完了していた修正作業を再構築時に利用するかという問題もある．ただし，3.2.2 項の方法で提示されるべき帰納型が提示されなかった場合や，提示されてもプログラマが無視した場合に，修正作業中に気付く場合もあるため，実装の複雑さとの兼ね合いを考慮し導入を検討する．

4.3　有効な対象と現状の問題

対話的修正機構は，コンストラクタを追加するという要求であり，かつ 4.1 節であげた問題が生じない限り，あらゆるプログラムに対して適用することができる．特に本論文の例題のように，現在の構成やパターンを保持したままのプログラムに対して，確実な修正を行うことが可能となる．

しかし，3.2 節における手法はそのシンプルさから問題を持つ．例として，Aexp の最適化器による結果の保存に関する，拡張前の証明を表す項の一部を**図 9** に記載する．2 行目において帰納原理関数が用いられているため，Amult が追加された際にはこの箇所が検知され，Amult の場合に関する証明を行う．一方，本来の証明では，何もする必要なく証明が完了するようになっている．これは証明の流れが Aadd の場合とまったく同様であるためであり，したがって本研究の手法では明らかに作業が増えることが分かる．

また，さらに大きな問題として，14 行目や 16 行目のパターンマッチにおいて Aexp の展開を行っている点があげ

```
fun a : Aexp =>
Aexp_ind
 (fun a0 : Aexp => forall e : Env, Aeval a0 e = Aeval (optimize_A a0) e)
 (fun (v : Val) (e : Env) => eq_refl (Aeval (optimize_A (Aval v)) e))
 (fun (v : Var) (e : Env) => eq_refl (Aeval (optimize_A (Avar v)) e))
 (fun (a1 : Aexp)
  (IHa1 : forall e : Env, Aeval a1 e = Aeval (optimize_A a1) e)
  (a2 : Aexp)
  (IHa2 : forall e : Env, Aeval a2 e = Aeval (optimize_A a2) e) (e : Env) =>
 eq_ind_r
  (fun v : Val =>
  (v + Aeval a2 e)%Z =
   Aeval
    match optimize_A a1 with
    | Aval z1 =>
      match optimize_A a2 with
      | Aval z2 => Aval (z1 + z2)%Z
      | Avar _ => Aadd (optimize_A a1) (optimize_A a2)
      | Aadd _ _ => Aadd (optimize_A a1) (optimize_A a2)
      end
    | Avar _ => Aadd (optimize_A a1) (optimize_A a2)
    | Aadd _ _ => Aadd (optimize_A a1) (optimize_A a2)
    end e)
 ... (* 80 lines left *)
```

図 9　optimize_A_keep_result の証明を表現する項
Fig. 9　The term denoting a proof of optimize_A_keep_result.

られる．このようなパターンマッチは本手法の検出対象だが，証明の項全体で 10 カ所以上存在するため，それらのすべての箇所で修正作業を強制される．Stm に対する最適化器でも同様に多くのパターンマッチがあり，作業量が現実的でなくなってしまう．

この問題の解決は原理上難しくない．本来の証明を記述したコードがあるという仮定の下では，拡張前と拡張後で証明を行い，拡張した影響により増加したゴールをすべて集めることで，既存の証明を最大限利用することができる．また，証明中にエラーが出た場合でも，そのエラーの出現した箇所を特定することで，コードを直接変更した場合に再利用がきかないような証明も利用することができる．しかし，この解決方法は実装に関しては容易ではないため，将来の課題とする．

5. 関連研究

本研究のように，証明支援系における証明後のプログラムを拡張するという試みは，我々の知る限り存在しない．適用可能範囲を限定することにより，証明を容易に，柔軟に記述するという研究は多く存在する [3], [4], [5] が，我々の研究とこれらは相反することはなく，互いに補完し合うものであると考えられる．たとえば，これらの研究により拡張前の検証を行い，我々の手法により拡張を行うといった利用が考えられる．

関数型言語として Coq をとらえた場合，対話的修正機構は型の定義や関数の挙動を変更する機構であり，アスペクト指向言語が持つ機能としてとらえられる．特に，本論文であげた対話的修正機構は，Aspectual Caml [7] における機能と非常に近い．Aspectual Caml は OCaml をアスペクト指向言語へと拡張した言語であり，データ型の拡張や関数の置き換えが可能である．

本研究と Aspectual Caml との類似点としては，可能な修正の範囲があげられる．本研究では，帰納型の拡張とパターンマッチの拡張を可能にしているが，Aspectual Caml では同様に型の拡張とポイントカットによるパターンマッチの拡張が可能である．Aspectual Caml ではこれ以外にも多くのポイントカットが指定でき，その意味では Aspectual Caml の方が柔軟であるといえる．一方で，Aspectual Caml においてパターンマッチの拡張を考えると，本論文であげた問題と同様に，追加したコンストラクタに関する適切なパターンの追加忘れが発生すると考えられる．対話的修正機構ではこの問題を解決するという点で異なっている．

コンストラクタの追加という観点では，type class や polymorphic variant [6] など，expression problem と呼ばれる問題の解決法があげられる．特に type class は Coq に 8.2 より導入されており [8]，すでに使用できる．これらは型の定義後にコンストラクタ（に相当するもの）を追加することが可能であり，その意味で本研究の手法と近い．しかし，これらは網羅性の保証が存在しないことや，帰納原理関数が定義できない（定義が自明でない）など，証明において望ましい性質が欠けているため，Coq に対しての導

入に困難がともない，プログラマへの負担も小さくない．本研究の手法は Coq の持つ機構を最大限利用することで，プログラマの負担をできる限り小さくしている．

6. まとめ

本論文では，型にコンストラクタを追加した際に，考えられる修正箇所を提示するシステムである対話的修正機構について，手法と実装方法を提示した．これにより，対話環境におけるエラーが発生しないような箇所であっても候補としてあげることが可能であることを示した．

将来の課題としては，4 章であげた制約の緩和，対話的修正機構の可能な修正の範囲の拡張があげられる．前者としては，たとえばソート多相に関して，今回は必ずソートが等しくなるように制限をしているが，実際には型のソートを `Set` から `Type` に拡張することでプログラム全体が許容される可能性がある．このような作業自体は比較的機械的にできるものであり，またプログラマの手が必要となる場合は対話的に進めることが可能であることから，対話的修正機構の有用な機能となると考えられる．一方で，この挙動が pCIC の型システム上で正しく表現されるものであるかを事前に考慮する必要がある．また，後者については非常に多くの修正が考えられ，たとえば関数や型へのパラメータの追加，関数の置き換えなどがあげられる．特に関数へのパラメータの追加は帰納原理関数においても行われていることであり，導入の必要性は高い．しかし，帰納原理関数においても非常に強い制約を用いているため，この緩和などを同時に行わなければならない．そのほかにも，証明の再利用など実装上の制約の緩和などが，将来の課題としてあげられる．

謝辞　査読者より有益な助言・コメントをいただいた．ここに感謝する．本研究の一部は科学研究費補助金（研究課題番号 24500033）の補助を受けている．

参考文献

[1] The CompCert project, available from ⟨http://compcert.inria.fr/⟩.
[2] Coqtail project, available from ⟨http://sourceforge.net/projects/coqtail/⟩.
[3] Aydemir, B., Charguéraud, A., Pierce, B.C., Pollack, R. and Weirich, S.: Engineering Formal Metatheory, *ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp.3–15 (2008).
[4] Chlipala, A.: Parametric higher-order abstract syntax for mechanized semantics, *Proc. 13th ACM SIGPLAN international conference on Functional programming*, pp.143–156 (2008).
[5] Chlipala, A.J., Malecha, J.G., Morrisett, G., Shinnar, A. and Wisnesky, R.: Effective interactive proofs for higher-order imperative programs, *ACM SIGPLAN international conference on Functional programming*, pp.79–90 (2009).
[6] Garrigue, J.: Programming with polymorphic variants, *ACM SIGPLAN Workshop on ML*, informal proceedings (1998).
[7] Masuhara, H., Tatsuzawa, H. and Yonezawa, A.: Aspectual Caml: an aspect-oriented functional language, *ACM SIGPLAN International Conference on Functional Programming*, pp.320–330 (2005).
[8] Sozeau, M. and Oury, N.: First-Class Type Classes, *21st International Conference Theorem Proving in Higher Order Logics*, LNCS, Vol.5170, pp.278–293 (2008).

森口 草介　（学生会員）

昭和 60 年生．平成 21 年東京工業大学大学院情報理工学研究科計算工学専攻博士前期課程修了．同年より東京工業大学大学院情報理工学研究科計算工学専攻博士後期課程．主な研究分野はアスペクト指向，定理証明支援系．日本ソフトウェア科学会会員．

渡部 卓雄　（正会員）

東京工業大学大学院理工学研究科情報科学専攻博士後期課程修了．理学博士（東京工業大学）．日本学術振興会特別研究員，北陸先端科学技術大学院大学勤務を経て，現在，東京工業大学大学院情報理工学研究科計算工学専攻准教授．プログラミング言語の設計と実装，形式手法，自己反映計算に興味を持つ．日本ソフトウェア科学会，ACM 各会員．