対話的修正と対象プログラムとの合成手法

森口 草介　高橋 和子

本論文では、Coq により検証した系と、それに対して対話的修正機構を用いて作成した修正部分を、合成して一つの系として得る手法を与える。Coq により行った証明に対して、修正をコマンドを通じて対話的に行う機構、対話的修正機構を提案した。この対話の結果として記述したコマンドの列を対話的修正と呼ぶ。

対話的修正は、明示的に修正箇所を指示するコマンドの他に、それらの影響を受ける箇所を機構が提示し、その場所に挿入する項を表現するコマンドから構成される。これは、修正対象の系の記述から分離されているので、管理が容易となるものの、系が単体で完結しないなど、可読性の観点からは望ましくない。これを改善するために、対象の系と対話的修正を合成することが考えられる。しかし、型の宣言や関数における修正箇所は比較的容易に発見できるものの、証明に関しては証明を表す項から発見していたため、コマンド列として記述される証明において、適切な修正箇所を発見することは難しかった。

本論文では、対象としている系の証明を、合成が容易となるように変形することで、元の証明を利用して記述した対話的修正との合成を可能とする。変形の結果、元の証明に比べ記述量が増大してしまうが、これは自動化された箇所が一部展開されることに起因する。変形は元の証明の流れに合わせて行われるため、メンテナンス性や速度の観点では問題とならない。ここでは、小さな例を用い、合成時に発生しうる問題や変形方法について述べる。

1　はじめに

Coq をはじめとした対話的証明支援系による数学的体系やプログラムの検証は、厳密な記述により確かさが保証される。他方、この厳密さにより、通常の手による検証に比べ、記述の冗長さ、証明の煩雑さが発生する。

この厳密さによる問題は、単純な拡張に対する再検証においても発生する。数学的体系における（表現可能な）定数の追加やプログラミング言語における糖衣構文の追加は、追加前に確かめた性質をほぼそのまま満たす。この場合、自然言語による証明では、拡張後の性質について「同様に」という言葉で記述することも少なくない。しかし、Coq を用いた検証において、「同様に」という記述は難しく、自動化を試す、類似した証明を読ませる、などを試すにとどまる。一方で、自動化の手法によっては、一見検証を変更することなく解決したようにみえる場合がある。ただし、その検証が意図した通りに行われたかを判断することは難しく、実際に検証しようとした仕様が意図を満たさない場合もある。

対話的修正機構 [6] は、このような修正時における、影響の範囲を網羅的に調べながら対象の修正を行うための機構である。この機構は、コマンドを通じた対話により、一度記述した型や関数、証明に関する修正を行う。修正の内容を機構が受け取ることで、影響を受ける範囲を調べ、提示し、修正を求める。この対話によって見落としやすい修正箇所などを確実に提示することが可能となる。

以下、対話的修正機構を用いて修正する対象を**被修正系**、作った修正の記述を**対話的修正**と呼ぶ。対話的修正は、コマンドの列として記述され、被修正系からは分離されているため、修正内容の着脱が容易であるという利点を持つものの、どの箇所がどのように変更されたかを確認することが難しいという問題を持つ。

第一著者はこの問題に対して、対話的修正と被修正系を合成することで問題の解決を図った [5] が、しか

A method for weaving an interactive extension into its target
Sosuke MORIGUCHI and Kazuko TAKAHASHI, 関西学院大学, Kwansei Gakuin University.

しこの時点では二つの問題によって全面的な解決に至らなかった。

1. 修正する箇所が表面的な記述と無関係に行われるため、その位置を正確に計ることが困難だった。
2. 証明の記述を複雑化できるため、修正の適用方法が容易には決定できなかった。

前者は、特に証明を表すスクリプト（以下**証明スクリプト**）において顕著である。これは当初の対話的修正機構が、型と項の構造を元に修正箇所を決定し、証明スクリプトを利用することなく行っていたためである。そのため、本来の証明の流れでは発生しない箇所に修正箇所が存在する、単一の tactic が生成した項の一部を修正箇所として検知する、といったことが発生した。これにより、証明スクリプトのどの箇所から発生したかを決定することが難しく、また決定できたとしても元の tactic がどのように振る舞うのかを見積もることが難しかった。

この問題は、第一著者の提案した、記述された元の証明スクリプトを再実行することで修正箇所を減らす技法[3] を用いることで解決できる。再実行では、その証明スクリプトの流れに沿った修正箇所を得られ、またそのときに得られる修正箇所は tactic から実際に出現するものであるため、これらの問題が回避される。

後者の問題は、この手法でも未解決の問題である。これは例えば以下のスクリプトを考える。

```
inversion H; destruct n; auto.
```

修正によって、inversionの結果現れるゴールが変わり、autoによって証明できないものが現れうる。しかし、この場合で存在しない、などの理由により destructが失敗すると、再実行に基づく修正箇所の検出ではその位置から修正を開始する。この修正結果を、上のスクリプトと合成しようとした場合、destruct nに対して位置、内容を決定しなければならない。しかし、これらが repeatによって繰り返されるように記述されていた場合、destruct nの失敗は修正によって増えたゴール以外でも発生している可能性がある。

この場合に修正をどこに挿入するかは、次の二つがありうる。

- repeat内部に挿入する。この場合、繰り返しの中で他の証明に影響を与えないために、可能な限り挿入内容を修正箇所以外で実行されないように保護しなければならない。
- repeatの直後に挿入する。この場合、他の証明がこの修正により影響を受けないよう、修正箇所にのみ適用されるようにその箇所に現れるゴール数を決定し、分岐を生成する必要がある。

どちらの問題も、単一の repeatであれば解決も難しくないが、Coq では repeatをネストして使用できる。これは、汎用に用いることを考えたとき、先ほど挙げた問題のどちらも常に考慮しなければならないことを意味する。

本論文では、この問題に対し、元の証明を再実行する際の動きを元に証明スクリプトを変形、展開し、挿入する箇所を一意に特定することで解決する。その際に、repeatなどの構文をできる限り変形しないままに合成をすることで、自動化を維持することを目標とする。

本論文の構成は以下の通りである。

- 第 2 節では、対話的修正機構による修正の流れとその結果得られる対話的修正について述べた後、被修正系と対話的修正の合成において発生する問題を説明する。
- 第 3 節では、被修正系の証明スクリプトにおいて、修正の記述時に開始した点である**挿入点**について議論、分析をする。
- 第 4 節では、被修正系の証明スクリプトを、挿入点の近辺において展開を行う。この作業により、合成を容易な手順とする。
- 第 5 節では関連研究に関して述べ、最後に第 6 節においてまとめる。

2 対話的修正と合成の問題

対話的修正単体については、特にメンテナンス性に関する問題がいくつか存在する。ここでは、対話的修正機構による修正の流れと、問題点について明らかにする。

2.1 対話的修正機構

対話的修正機構とは、被修正系に対する修正を確実に行うために提案した機構である。そのため、ここでは単純な被修正系と修正について、例を用いて説明する。なお、本論文で扱う対話的修正機構は[3]において提案したものである。したがって対話的修正機構の挙動に関する形式的定義、詳細はそちらを参照されたい。

被修正系の例として、簡単な変数付きの算術式について説明する。内容を箇条書きにしたものが以下の通りである。

- 算術式には定数、変数、加算がある。
- 評価は変数の値を持つ環境を引数に、算術式の構造に対して再帰的に評価することで行われる。
- 定数同士の加算を折りたたむ最適化がある。
- 算術式が変数を含む、という述語がある。
- 変数を含む算術式は、最適化しても定数にならないという証明がある。

これを Coq のコードとしたものが図 1 である。算術式や述語の定義が型として定義されており、また評価と最適化が関数として記述されている。また、証明は述語に関する帰納法を用いている。

この証明は、図 1 以外に図 2 や図 3 のようにも書ける。各証明では、`induction`によって述語に関する帰納法を行った際に三つのゴールに分離している。元のコードにおける証明では、一旦ピリオドで区切ることによりそれぞれの証明を順次行っていたが、図 2 のコードでは `simpl`の直後にゴールが三つ残っており、それぞれに対して`|`で区切ったそれぞれを適用することを明示している。この場合、述語の定義の変化などによりゴールの数が変化すればそこで証明が止まるため、信頼性の向上に役立つと考えられる。

他方、図 3 ではそれぞれのゴールに一つずつ適用することと異なり、三種類の`||`を使って区切った証明を**順次**試すように記述されている。この場合、加算に関しての証明である二番目と三番目が、例えば乗算を組み込んだ場合の証明としても利用できる。このように、証明としてはある程度先の修正を見越した記述もありうる。

ここで、算術式の演算子の追加を考える。ここで考えるものは、単項の負号である。対話的修正機構では、コマンドにより型の拡張を行うと、その後関数や証明に対し修正を指示され、それをコマンドによって修正するという流れとなっている。例えば図 1 を対象とした場合、図 4 の対話的修正が記述できる。 被修正系の証明を図 2 や図 3 に入れ替えた場合、表面的に記述する対話的修正は変化しない。しかし、これは原因が異なっている。

- 図 1 では `induction`によって生成されたゴールがそのまま残ったため、そのゴールについて証明を

```
Inductive exp : Set :=
| val : Z -> exp
| var : V -> exp
| add : exp -> exp -> exp.

Fixpoint eval (e : exp) (env : V -> Z) :=
match e with
| val z => z
| var v => env v
| add e1 e2 => eval e1 env + eval e2 env
end.

Fixpoint opt (e : exp) :=
match e with
| add e1 e2 =>
   match opt e1 with
   | val z1 =>
      match opt e2 with
      | val z2 => val (z1 + z2)
      | e2' => add (val z1) e2'
      end
   | e1' => add e1' (opt e2)
   end
| _ => e
end.

Inductive var_include : exp -> Prop :=
| var_v : forall v, var_include (var v)
| var_al : forall e1 e2, var_include e1 ->
     var_include (add e1 e2)
| var_ar : forall e1 e2, var_include e2 ->
     var_include (add e1 e2).

Goal forall e, var_include e ->
                forall z, opt e <> val z.
intros e H.
induction H; intros z; simpl.
 intro vi; simpl in vi; inversion vi.
 case_eq (opt e1); intros; intro H1;
    try (inversion H1; fail).
  apply (IHvar_include z0); auto.
 case_eq (opt e2); case (opt e1); intros;
    intro H1; try (inversion H1; fail).
  apply (IHvar_include z1); auto.
Qed.
```

図 1　Coq による単純な算術式についての検証。

```
Goal forall e, var_include e ->
                forall z, opt e <> val z.
intros e H.
induction H; intros z; simpl; [
 intro vi; simpl in vi; inversion vi |
 case_eq (opt e1); intros; intro H1;
    try (inversion H1; fail);
  apply (IHvar_include z0); auto |
 case_eq (opt e2); case (opt e1); intros;
    intro H1; try (inversion H1; fail);
  apply (IHvar_include z1); auto].
Qed.
```

図 2　証明の別解 1。

```
Goal forall e, var_include e ->
                forall z, opt e <> val z.
intros e H.
induction H; intros z; simpl;
(intro vi; simpl in vi; inversion vi;
    fail) ||
(case_eq (opt e1); intros; intro H1;
    try (inversion H1; fail);
  apply (IHvar_include z0); auto; fail) ||
(case_eq (opt e2); case (opt e1); intros;
    intro H1; try (inversion H1; fail);
  apply (IHvar_include z1); auto).
Qed.
```

図 3　証明の別解 2。

求めている。

- 図 2 では適用するゴールの数が修正結果生成されるゴールについて無視する必要があったため、追加生成されたゴールを別途証明を求めている。
- 図 3 ではそれぞれ||でつながれた証明を試行し、結果的に失敗したためそこまでの状態で別途証明を求めている。

2.2　対話的修正の問題

対話的修正機構では、被修正系に対して網羅的な修正を求めることで、意図せぬ修正漏れを起こしづらくする。また、図 1 と図 4 のように、対話的修正の記述が被修正系と分離されているため、被修正系への適用と分離が容易に行えるという利点がある。

一方で、分離されてるために型の宣言などの全体を把握することが難しくなる。これは被修正系と分離されていることとのトレードオフとなっているが、修正の内容を特に分離する必要がない場合、単純なデ

```
Extend Inductive exp : Set :=
| neg : exp -> exp.

Extend Inductive var_include : exp -> Prop
     :=
| var_n : forall e, var_include e ->
     var_include (neg e).

Deploy.
(* eval *)
exact -(eval e0 env).
Defined.
(* opt-outside *)
exact (match opt e0 with
  | val z1 => val (-z1)
  | e' => neg e'
  end).
(* opt-inside-add1 *)
exact (add (neg e0) (opt e2)).
(* opt-inside-add2 *)
exact (add (val z1) (neg e0)).
Defined.
(* proof *)
case_eq (opt e); intros; intro H1;
   try (inversion H1; fail).
 apply (IHvar_include z0); auto.
Qed.
```

図 4　負号に関する対話的修正。

メリットとなる。

本論文における目標は、対話的修正と被修正系を合成し、修正後の体系を表すソースコードを生成することである。この場合に発生しうる問題としては、以下のものがある。

1. 修正時に作成した記述を埋め込む場所の発見が難しい。
2. 発見した箇所に埋め込んだ場合に他の証明に影響を与える可能性がある。
3. 可読性や自動化をある程度保った状態にすることが難しい。

順に説明する。

まず、発見の難しさは、対話的修正機構の修正箇所の発見手法に依存する。当初、対話的修正機構による証明内の修正箇所は、証明を表す項を関数同様に調べることで行っていた。これは網羅性や確実性、既存のシステムの再利用、理論的基礎という視点では優れているが、項における修正箇所が証明スクリプトの修正に比べ過剰に検出されることから、ユーザビリティの低下を招いていた。また、修正箇所がどのような流れによって発生したかが分かりづらく、本研究のように

証明スクリプトと修正箇所の対応付けが必要となる際の障害となっていた。

現在では、再実行に基づく修正箇所の発見[3]を利用することで、証明スクリプトに利用されたtacticの単位で修正箇所を判定できる。そのため、本論文では修正そのものがこの手法に基づいて行われた前提で議論を進める†1。

次に、他の証明への影響を考える。Coqにおいて証明に用いられる言語Ltacは`repeat`による繰り返しやtacticの連結、複数の特定数のゴールへの適用などの機能がある。修正によって`repeat`の中で新しいゴールが生成されると、その直後に何らかの形で修正の内容を配置する必要がある。しかし、その修正は他の繰り返しにおいて影響を与えうる。一方、特定数のゴールへの適用は数が変化しているため、必ず変更しなければならない。

最後に可読性や自動化について考える。上の二点について解決しうる最も単純な手法は、証明に使われている記法を全て展開し、単一のtacticを必要なだけ配置するものである。例えば、上の`repeat`について、それが3回繰り返したのであれば、3回その中身を記述する、`try`などについては成功した場合中身を置き、失敗した場合は消去する、などの手法で達成できる。しかし、この場合「類似の証明に対して自動的に証明する」などの工夫を施した記述が全て展開されており、それ以降対話的修正機構が対応できないような修正を行った場合に再利用することが困難となる。

本論文において述べる手法は、元の記述の構成を極力保持したままで修正を合成する。この手法では、まず証明スクリプトを一部のみ展開し、挿入すべき箇所を決定可能にする。展開は汎用的なものでなく、対象とする対話的修正についてのスクリプトの挙動を元に行う。これにより、最小限度の展開によって合成が可能となることが期待できる。

3 挿入点

展開のために、ここでは各ゴールがどのようなスクリプト内のパスを通過するかを考える。以下では、

$$
\begin{aligned}
T \quad ::= \quad & \texttt{atomic\_tac} \\
| \quad & T;T \\
| \quad & T \,||\, T \\
| \quad & T;[T|T|...|T] \\
| \quad & \texttt{try}(T) \\
| \quad & \texttt{repeat}(T)
\end{aligned}
$$

図5　本論文で扱うLtac（のサブセット）。

図5の文法をLtacの文法として用い、証明スクリプトを要素Tの0個以上のリストとして扱う。

ここで表す文法は以下のような分類である。

- `atomic_tac`はこれ以上分解できないtacticを表し、`T1;T2`は`T1`に続けて`T2`を実行することを表す。
- `T1 ||T2`は`T1`を実行し、失敗した場合に`T2`を実行する。
- `try(T)`は`T`を実行し、失敗した場合は何も実行しなかったものとする。
- `repeat(T)`は`T`を、失敗するまで繰り返し実行する。

なお、この中に含まれていないLtacの文法は全て`atomic_tac`として扱う。

対話的修正において、修正を行う起点を以下では**挿入点**と呼ぶ。挿入点は対話的修正機構がどのような点を開始点にするかによって異なる。

現在提案している対話的修正機構の再実行機能は、連結の途中における失敗、特定数のゴールへの適用またはtacticの終端まで再実行を行う。この結果、多くの場合は`repeat`や`try`の内部に挿入点は存在しない。これは、`repeat`は内部で失敗するまで終了しないが、失敗した場合にエラーにならずに終了するため、再実行の手法の関係上以降のtacticの実行を続けようとするためである。

例外として、特定数のゴールへの適用時に中止する場合がある。したがって、以下のようなコードを記述した場合、挿入点が`repeat`の内部に存在する場合が発生する。

```
repeat (inversion H; [ .. | .. ])
```

`H`の型が帰納的であり、かつコンストラクタを追加した型である場合、生成されるゴールが増える可能性

†1　前述の通り、2.1節の説明はこの機能を用いている。

がある。しかし、このコードは可読性やメンテナンス性に関して疑問がある。なぜならば、繰り返し同じものを展開し、同じ処理をすることはほとんどないためである。例えば destructのように同じ変数名で異なる変数が割り当てられるようになる場合もあるが、repeatで繰り返すほど同じ処理として記述することは難しく、省略した分岐のそれぞれが比較的複雑な記述をする必要がある。このような場合、可読性に関しての問題の他に、対話的修正機構の範囲に限らず修正一般に対して非常にもろい構造となる。

本論文では、挿入点が構造的に内側に入るものを展開する。この構造とは、repeatや tryの他、||でつながれたものや特定数のゴールへの適用を意味する。これらはゴールが増加した際に成功や失敗について様々な影響が考えられるため、展開することで調整可能とする。

4 証明スクリプトの展開

前節の分析から、次の展開を考える。

1. 挿入点が repeat、tryの内部である場合、または||によって連結したものの左側の内部である場合（右側の内部に挿入点がある場合は除く）、以下の方針で展開する。まず、複数のゴールで実行されないように直前で tactic の連結を切り、適用されるゴールの数だけ複製する。その後、
   - repeatは被修正系や対話的修正における繰り返しの数だけ、生成されるゴールの数に合わせて繰り返しリストに挿入する。
   - tryの内部の tactic について、成功失敗により中身をそのまま置くか除去するかを決定する。
   - ||の左側が成功する場合は左側のみを、右側について試す場合は右側を残す。
2. 挿入点が特定数のゴールへの適用直前の場合、後続の tactic の列を内部に組み込む。これは各々のゴールへ適用する tactic に対し、証明がその分岐内で終わっていれば何もせず、終わっていなければ後ろの tactic の列をそのまま連結する。その後、特定数のゴールへの適用に増加したゴールの数だけ idtacを適用するように追加する。修正内容はその分岐が終了した後に増加したゴールを証明する箇所に配置する。
3. 挿入点が特定数のゴールへの適用内部の場合、後続の tactic の列を上と同様にして内部に組み込み、その後外側の適用を展開する。具体的には、各ゴールに適用する tactic の列をその順序に合わせて配置する。

この展開は挿入点以外について行われないため、前節において説明した対話的修正機構の特性上、特定数のゴールへの適用が含まれない自動化はそのまま残る。この展開を再帰的に行うことで、挿入点は次の三点のどれかとなる。

- 二つの tactic の間。この場合は単純にその間に修正結果を挿入すればよい。
- 二つの tactic を連結した間。後続の tactic 全体をtryで囲い、tactic 終了後に修正内容を配置することで合成できる。
- ||の右側の内部。これは直上の基準同様に連結した後続の tactic に tryを用いることで合成できる。

なお、この挿入では tryの追加が発生するので、そのほかの挿入点がこれによる影響を受ける場合、再度上に挙げた展開を行う必要がある。

展開を図 1 の証明スクリプトに適用すると、全く変形されない。これは、挿入点が最後の **Qed**の直前であり、展開する必要がないためである。挿入後の証明スクリプトは以下の通り。

```
Goal forall e, var_include e ->
                forall z, opt e <> val z.
intros e H.
induction H; intros z; simpl.
 intro vi; simpl in vi; inversion vi.
 case_eq (opt e1); intros; intro H1;
    try (inversion H1; fail).
  apply (IHvar_include z0); auto.
 case_eq (opt e2); case (opt e1); intros;
    intro H1; try (inversion H1; fail).
  apply (IHvar_include z1); auto.
 case_eq (opt e); intros; intro H1;
     try (inversion H1; fail).
  apply (IHvar_include z0); auto.
Qed.
```

次に図 2 に対して行った場合に記す。

```
Goal forall e, var_include e ->
                forall z, opt e <> val z.
intros e H.
induction H; intros z; simpl; [
 intro vi; simpl in vi; inversion vi |
 case_eq (opt e1); intros; intro H1;
    try (inversion H1; fail);
  apply (IHvar_include z0); auto |
 case_eq (opt e2); case (opt e1); intros;
    intro H1; try (inversion H1; fail);
  apply (IHvar_include z1); auto |
  idtac ].
(* insert the script here *)
```

この場合の挿入点は [の直前であり、上で述べた展開の手順の 2 に相当する状況である。そのため、エラーを回避するために idtacを追加することでゴールの数を合わせている。また、後続の tactic の列が存在しないため、それぞれの記述はそのままである。

最後に図 3 に対して展開を行った場合について述べる。この場合の挿入点は二つ目の||の直後であり、再実行は case_eq (opt e2)の変数名が存在しないというエラーによって止まっている。これは展開に相当する状況ではない（||の内部だが、右側なので例外的に許される)。したがって、展開せずに、一部変形を行って挿入を行う。挿入の方法を元に生成される証明スクリプトは以下の通り。

```
Goal forall e, var_include e ->
                forall z, opt e <> val z.
intros e H.
induction H; intros z; simpl.
(intro vi; simpl in vi; inversion vi;
    fail) ||
(case_eq (opt e1); intros; intro H1;
    try (inversion H1; fail);
  apply (IHvar_include z0); auto; fail) ||
(* try is inserted *)
(try (case_eq (opt e2); case (opt e1);
    intros;
    intro H1; try (inversion H1; fail);
  apply (IHvar_include z1); auto)).
(* inserted script *)
 case_eq (opt e); intros; intro H1;
    try (inversion H1; fail).
  apply (IHvar_include z0); auto.
Qed.
```

このように、順次試行することによる自動化は残すことができており、これに乗算を追加した場合でも証明が自動的に行われる。しかし、tryが追加されたことにより、他の修正点が try内部に存在した場合の問題が発生しうる。この場合、スクリプトが複雑化したり、自動化が展開される可能性がある。

5 関連研究

Coq における証明のように、tactic を用いた証明に関するリファクタリングは、Whiteside らが行った研究がある [4]。この研究におけるリファクタリングは、名前の変更などの比較的局所的なものと、証明をゴール指向から演繹的証明に変更するなどの大域的なものがあり、可読性に重点を置いている。しかし、本研究で述べたように修正や挿入を意識した変形や展開ではないため、これらのリファクタリング技法をそのまま用いることはできない。今後、このリファクタリング技法による証明スクリプトの変形が、どのように本論文で提案した手法に影響を与えるかを分析することで、本論文の手法の拡張として導入を考慮したい。

本研究では、tactic の個々の挙動をほぼ考慮せず、意味については一部の文法のみを考慮するに留めているが、各 tactic の挙動について考慮することで、その他の展開の基準を得ることも考えられる。Ltac の意味論としては、Jedynak らの研究 [1] がある。今後はこれを用いたより詳細な分析、展開について考慮した手法の提案や、今回の展開が Ltac における意味論に関して意味を保存していることの証明を行う。

分離したプログラムの断片を一つのコードとして表現することは、アスペクト指向 [2] における織り込みやメタプログラミングとして行われる。しかし、多くのアスペクト指向の織り込みやメタプログラミングはコンパイル時や実行時において想定した動作を実現するものであり、それらの機能を組み込んだソースコードを生成することはほとんどない。これは、機能、非機能にかかわらず、これらの記述を内部に組み込んだ記述が必ずしも元の言語における記述と対応せず、また対応する場合でも可読性の確保は難しいためである。本研究の基盤である対話的修正機構は、そもそも保守的な変更に限っているため、元のソースコードの構造を保持しつつ、可読性の確保が可能となった。

6 まとめ

本論文では、対話的修正機構によって行った修正を被修正系に組み込むための手法として、被修正系の証明スクリプトを展開を経て行う手法を提案した。この

手法では、単純な合成ではうまく挿入できない修正の記述を、被修正系の記述を展開することで可能とした。繰り返しなどの自動化については挿入しなければならない箇所に限り展開することで、少なくとも簡単な例では残すことに成功している。

今後の課題としては、挿入点の拡張が挙げられる。本論文における手法は、対話的修正機構の再実行機能がエラーを起こす箇所まで実行することを利用した。しかし、この再実行機能は必ずしも外側まで実行せず、修正によって発生したゴールが生成された tactic で実行を止めた方がよい場合もある。これは、生成後から外側に脱出するまでに、不可逆な tactic が実行された場合などがある。極端な例として、`clear`という tactic は、環境で使用出来る変数を消去するものであり、一般には以降使用しない変数に対して用いることで環境を整理する目的に使用される。しかし、変数が見えなくなるためこれを実行した後で証明ができなくなることもある。

このように、可能な限り tactic を実行するという戦略は、場合によっては問題を起こしうる。調査した範囲ではこのように過剰な操作が行われることはなかったが、想定することは可能であり、そのための対策を用意すべきである。現在、実装する上で出現箇所に戻るコマンドの提供を考えている。この場合には、挿入点が任意の箇所に入ることになり、現在の仮定が崩れる。

これをサポートするには`repeat`内など、全ての通過した箇所や通過時の状態を計算し、展開方法をさらに拡張する必要がある。しかし、これによって自動化を全て展開しなければならない可能性もあるため、今後は自動化の残り方や柔軟な展開手法について考察する。

参考文献

[1] Jedynak, W., Biernacka, M., and Biernacki, D.: An operational foundation for the tactic language of Coq, *PPDP*, Peña, R. and Schrijvers, T.(eds.), ACM, 2013, pp. 25–36.

[2] Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J.: Aspect-Oriented Programming, *11th European Conference on Object-Oriented Programming*, LNCS, Vol. 1241, June 1997, pp. 220–242.

[3] Moriguchi, S. and Watanabe, T.: An interactive extension mechanism for reusing verified programs, *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, New York, NY, USA, ACM, 2013, pp. 1236–1243.

[4] Whiteside, I., Aspinall, D., Dixon, L., and Grov, G.: Towards Formal Proof Script Refactoring, *Proceedings of the 18th Calculemus and 10th International Conference on Intelligent Computer Mathematics*, MKM'11, Berlin, Heidelberg, Springer-Verlag, 2011, pp. 260–275.

[5] 森口草介, 渡部卓雄: Coq のための対話的修正機構を用いた変更の織り込み手法, 日本ソフトウェア科学会 第 *28* 回大会, September 2011.

[6] 森口草介, 渡部卓雄: 定理証明支援系 Coq への対話的修正機構の導入, 情報処理学会論文誌プログラミング（*PRO*）, Vol. 5, No. 4(2012), pp. 27–38.