推薦論文● PPL2015

レコードの拡張を許す対話的修正機構

森口 草介　高橋 和子

我々は対話的証明支援系 Coq に対して，対話的修正機構という，既存の証明済みの体系に対する拡張をサポートする機構の提案を行った．提案した機構では，宣言した帰納型に対する新しい構成子の追加を可能としていた．しかし，この機構では既存の構成子や型，関数に対する新たなパラメータの追加は不可能であった．

本論文では，レコードに対して新たなフィールドを追加する手法と，その場合の Coq における型システムの制限について分析する．この手法は，レコードだけでなく，構築子や関数に対するパラメータの追加も可能とするが，その場合の問題点についても述べる．

We introduced an interactive extension mechanism, which allows us to extend programs and systems verified with Coq. In the mechanism, only adding new constructors into existing inductive types are allowed, but adding parameters into functions or constructors are not.

In this paper, we introduce a novel mechanism, which allows us to add new fields into record types. We also show the limitations of the mechanism arising from the type system of Coq. The method used in the mechanism is also applicable for adding parameters to existing constructors and functions. We explain problems occurring when we apply the method into them.

1　はじめに

対話的証明支援系を用いた研究は，数学的体系やアルゴリズムなど，様々な分野に応用されている．ソフトウェアに対する検証も，CompCert Compiler [2] を始めとした比較的規模の大きなものが行われている．これらの体系やソフトウェアは，対話的証明支援系の持つデータ型や述語，関係，モジュールなどを用いて記述される．例えば Coq では帰納型，レコード型，モジュール機構，そして型クラス [15] を用いることができる．

体系の形式化やプログラムの記述は漸進的に行われることもある．つまり，体系のコアの部分の記述を先に行い，その後拡張として他の機能を追加していく．このときの追加は必ずしも大きな変更ではなく，単にデータ型へ新しい構築子を追加する，レコードのフィールドを追加する，証明すべき性質を追加する，などの単純なものも多い．この場合，コア部分の記述を拡張しなければならないが，単純な拡張であってもソースコードを直接編集しなければ不可能な拡張も多い．しかし，直接編集を行う場合，修正すべき場所を見逃し，バグを残してしまう可能性がある．

直接の編集を回避するために，我々は Coq における帰納型を拡張するための機構である対話的修正機構 [18] [11] を提案した．この機構では，定義済みの帰納型にあらたな構築子を追加したり，それに対応するパターンマッチの分岐を追加することが可能である．全体の拡張を対話によって行うことで，影響を受けうる型を提示したり，パターンマッチを網羅的に拡張できる．

しかしながら，この機構ではパラメータを追加することができず，例えば以下の拡張ができない．

- 関数や型への引数の追加

An interactive Extension Mechanism extending Record Types.

Sosuke Moriguchi, Kazuko Takahashi, 関西学院大学, Kwansei Gakuin University.

コンピュータソフトウェア, Vol.33, No.2 (2016), pp.125–138.

[研究論文] 2015 年 7 月 31 日受付.

- レコード型へのフィールドの追加

特に，Coq におけるレコード型 (その応用としての型クラス) では，フィールドにおいて他のフィールドに対する性質の記述が行われる．レコード型では，部分構造の記述や型クラスとしての親クラスの指定が可能であり，これらによってレコード型の拡張を表現できる．しかし，これら拡張の記述はインスタンスの網羅性を持たず，いくつかの要素に対応する拡張を記述し損ねる可能性がある．他方，直接編集を行うと，網羅性については確保されるものの，関数によってレコードのインスタンスを生成する場合，性質も含めて全て項として記述しなければならないという問題がある．

本論文では，Coq に対する対話的修正機構を拡張し，既存のレコード型に対して新たなフィールドを追加する手法を提案する．対話的修正機構では，元の記述と拡張の記述が分離されており，関数によって生成されるインスタンスへの拡張も容易に記述可能である．手で修正した場合などの問題点について述べた後，対話的修正機構を用いた記述について説明する．なお，本論文で提案している対話的修正機構は実装予定のものであり，貢献は主として手法の応用可能範囲についての分析である．

本論文の構成は以下の通りである．

- 第 2 節では，まず例題を用いて手での修正による問題点について述べる．ここで用いる例は expression problem [16] と呼ばれる問題のインスタンスである．
- 第 3 節では，適用する拡張と，そのための対話的修正機構との対話について述べる．
- 第 4 節では，この機構における制限などを Coq の用いている型検査の規則を元に説明する．
- 第 5 節で関連研究について述べ，最後に第 6 節でまとめる．

なお，本論文は静的型付き関数型言語と基本的な Coq の文法についての事前知識を想定している．

2 Coq における Expression Problem

本節では，例題を用いて Coq におけるレコード型の拡張について考える．本節で用いる例題は expression problem のインスタンスであり，既に関数型言語やオブジェクト指向言語において様々な解決方法が提案されているが，Coq において実現されているものはそれほど多くない．ここでは，まず帰納型を用いて expression problem について説明し，その 1 つの解決方法である型クラスによる解法を述べる．その後，解決方法についての解析を行う．

2.1 帰納型による式の定義

Expression problem では，算術式の定義，式上の操作の定義をまず行い，これら 2 つの側面に対する拡張性を議論する．ここでは，算術式を帰納型を用いて定義し，操作を関数として定義する．

2.1.1 初期のプログラム

Coq における帰納型を用いて算術式を定義すると以下のようになる．ただし，ここでは自然数と加算のみを定義している．

```
Inductive iexp : Set :=
| Iconst : nat -> iexp
| Iadd   : iexp -> iexp -> iexp.
```

次に，関数として算術式を計算した結果を返す関数 ievalを定義する．これは算術式の構造に関する再帰関数を用いている．

```
Fixpoint ieval e :=
  match e with
  | Iconst n   => n
  | Iadd e1 e2 =>
     (ieval e1) + (ieval e2)
  end.
```

2.1.2 操作の追加

まずこのプログラムに対して，操作の追加を行う．ここでは，単純なスタックマシンの命令へのコンパイルを記述する．

```
Fixpoint icompile e :=
  match e with
  | Iconst n   => (Ipush n :: nil)
  | Iadd e1 e2 => (icompile e1)
                  ++ (icompile e2)
                  ++ (Iplus :: nil)
  end.
```

この定義では次のスタックマシン命令の定義が必要となる．

```
Inductive inst : Set :=
| Ipush : nat -> inst
| Iplus : inst.
```

それぞれの命令はスタックへの自然数のプッシュおよびスタック上の 2 つの自然数をポップしてそれらを足し合わせたものをプッシュする命令である．命令を解釈して実行するスタックマシンは次の通りである．

```
Fixpoint execute il st :=
  match il with
  | Ipush n :: il' =>
     execute il' (n :: st)
  | Iplus :: il' =>
     match st with
     | n2 :: n1 :: st' =>
        execute il' (n1 + n2 :: st')
     | _ => None
     end
  | nil =>
     match st with
     | n :: nil => Some n
     | _ => None
     end
  end.
```

この追加については特に問題なく実現できる．また，この追加に関して，コンパイルが正確に動いていることを示すこともできる．

```
Theorem compile_correct :
  forall e : iexp,
    execute (icompile e) nil
    = Some (ieval e).
```

この証明は非常に簡単なため，ここでは省略する．

2.1.3 式の種類の追加

次に，式の種類の追加として，乗算を記述できるようにする．直観的には，以下のように iexpへ構築子を追加すればよい．

```
Inductive iexp : Set :=
 ... (* the same as before *)
| Imult : iexp -> iexp -> iexp.
```

しかし，この変更を行った後に Coq に読み込ませると，パターンマッチにおける網羅性検査によってエラーが検出される．これを回避するには，パターンマッチで iexpの要素を展開している箇所にパターンを追加すれば良い．例えば評価関数である ievalへの追加は以下の通りである．

```
Fixpoint ieval e :=
  match e with
  ...
  | Imult e1 e2 =>
     (ieval e1) * (ieval e2)
  end.
```

また，もし前節の操作 (コンパイル) を既に追加していた場合は，icompileの修正が必要となり，また命令の追加が必要であれば instの拡張も必要となる．命令が追加されると，式と同様にパターンマッチの網羅性検査によって executeの修正が必要となる．さらに，もしコンパイルの正当性について示していれば，その修正も必要となる †1．

2.1.4 帰納型における拡張の問題点

ここではプログラムに対する二種の拡張を見たが，これらの拡張をソースコードの編集や再コンパイルなしに可能かについて述べた問題を expression problem と呼ぶ．本節のように帰納型 (または代数的データ型) を用いて式を表現した場合，操作の追加は非常に簡単に行えるが，乗算の例のように式の種類の追加には対応できない．逆に，オブジェクト指向言語において visitor パターンを用いると逆の現象が発生する．

Expression problem は，プログラムの拡張性と編集の容易性，そして網羅性による安全性を全て担保することが難しいことを表している．これらの特徴はプログラムを徐々に発展させ，機能を追加する場合などに有用である．全ての特徴を満たすデータ構造は複数提案されており，例えば OCaml における多相ヴァリアント [5] があるが，Coq には採用されていない．

†1 今回のように非常に単純な証明であれば，修正を見越した自動化によって変更無しに証明ができる場合もある．しかし，一般には修正内容の予測も含め難しい．

2.2 型クラスを用いた表現

一般には，型クラスを用いることによって，expression problem における 2 つの拡張性を両立させることができる．Coq では，型クラスは 8.2 より導入されている [15]．ここでは，Coq の型クラスを用いた記述を説明する．

式ごとの操作を 1 つの型クラスのインスタンスとして表現し，それぞれの式をインスタンスとして宣言することで全体を構成する．関数や性質についてはそのフィールドとして表現され，また再帰関数については他のインスタンスに対する関数呼び出しとして記述される．型クラスを用いた初期プログラムは以下の通りである．

```
Class exp : Set := { eval : nat }.
Instance Econst (n : nat) : exp :=
  { eval := n }.
Instance Eadd (e1 e2 : exp) : exp :=
  { eval := @eval e1 + @eval e2 }.
```

ここで，@は関数の暗黙の引数 (implicit parameter) を明示するために用いられる．

新しい関数を追加するには，型クラスを継承することで記述できる．ここでは expWithCompileとして宣言している．

```
Class expWithCompile `(e : exp)
     : Set :=
  { compile : list inst }.
Instance ECconst (n : nat)
     : expWithCompile (Econst n) :=
  { compile := Ipush n :: nil }.
Instance ECadd
          (e1 e2 : exp)
          (e1' : expWithCompile e1)
          (e2' : expWithCompile e2)
     : expWithCompile (Eadd e1 e2) :=
  { compile := @compile e1 e1'
               ++ @compile e2 e2'
               ++ (Iplus :: nil) }.
```

ここでは関数 (コンパイル) のみ定義しているが，性質についても同様に追加できる．

もう一方の追加である式の種類の追加は，expのインスタンスを宣言するのみで良い．

```
Instance Emult (e1 e2 : exp) : exp :=
  { eval := @eval e1 * @eval e2 }.
```

これにより，コンパイルの関数と乗算の追加ができた．しかし，実際には乗算のコンパイルについて議論がされていないという点で不完全な拡張となっている．

2.3 型クラスを用いた場合の問題点

型クラスはその性質上，継承に対する網羅性が検査されない．これは本来型クラスでは一部のインスタンスが継承する性質を持っていないなどの理由により継承先のインスタンスとならない場合があるためである †2．しかし，Coq の型クラスでは，性質の追加だけではなく，関数の追加時にも継承を用いる必要がある．結果として，expression problem の解法として見た場合，関数の追加時に網羅性検査をしないという点で不完全な対応であり，確認をプログラマ自身が行う必要がある．

網羅性検査を行うには，例えば帰納型から型クラスのインスタンスへの対応関係を記述するという手がある．しかし，これは帰納型が直接の編集を除いて拡張できない，という点で妥当でない．したがって，Coq における型クラスは expression problem の解法としては不十分である．

拡張するという観点にのみ基づき，expression problem の制約から離れて考えた場合には，型クラスの定義を直接書き換えることで解決できる．

```
Class exp : Set :=
 { eval : nat; compile : list inst }.
```

編集後 Coq に再読込させると，Coq は追加したフィールドをそれぞれのインスタンスに対して求める．今回の場合は，compileを定数，加算の場合について求める．Coq は，Instanceを用いた記述に対しては証明モードによって不足した値を要求し，用いない記述に対してはフィールドが足りない旨のエラーを出力する．

†2 型クラスが広く用いられている Haskell では，型クラスへ処理を追加する場合に継承を必要としないため，この問題は発生しない．

Instanceを用いないインスタンスの記述としては，**Definition**による定数の宣言の他に，関数がインスタンスを生成する記述がある．例えば，次のコードは帰納型で定義した式を型クラスの式へ変換する関数である．

```
Fixpoint i2e (e : iexp) : exp :=
  match e with
  | Iconst n => {| eval := n |}
  | Iadd e1 e2 =>
     {| eval := @eval (i2e e1)
                 + @eval (i2e e2) |}
  end.
```

これは Econstや Eaddに相当するインスタンスを生成する関数である．厳密には Eaddは 2 つのインスタンスから新たなインスタンスを生成する関数であるため異なるが，5 行目と 6 行目に対応する Eadd (i2e e1)(i2e e2)は同じインスタンスを生成する．このような関数は型クラスのインスタンスとして考えるとあまり起こらないが，レコードではよく発生する．レコードでは継承が使用できないが，部分構造として他のレコードのインスタンスを取ることができるため，似た構造が記述できる．

ここで式にコンパイルの関数とその正当性を導入した場合，関数の返すインスタンスそれぞれに対して正当性の**証明**を記述しなければならない．証明を手で書くのは可能だが，非常に困難なため，可能であれば避けるべきである．また，証明モードによって関数を記述し，refineというタクティク (証明を進めるためのコマンド) を用いることで上に近い記述が可能となるが，書き換えが必要となり，また必要なフィールドを随時明記する必要があるなど，対応方法は単純とは言いがたい．

以上のように，Coq におけるレコード型や型クラスでは，拡張時に網羅性の点で問題が発生するか，または直接ソースコードを書き換える必要がある．ソースコードを書き換える場合には記述性に難があることもあり，またソースコードを直接編集できる必要がある．本論文で提案する対話的修正機構は，定義を修正するコマンドを用いることで網羅性を，対話によって修正をすることで記述性を向上させている．

3 対話的修正機構による拡張

本節では対話的修正機構によってどのようにレコード型を拡張するかについて述べる．ここで説明する機能は，以前の研究[11] において導入した，帰納型に対して構築子の追加をする拡張とは直交する機能であり，基本的には独立している．本節では拡張時の対話について重点的に述べ，機能としての制限は次節にて述べる．本節で拡張する対象は主にレコード型だが，レコード型は本質的に単一の構築子をもつ帰納型と同一であるため，本節では帰納型の構築子の拡張についても述べる．

なお，本節での入出力は対話的修正機構を組み込んだ Coq との対話を想定したものであり，記述されるコマンドは必要な情報を表すための疑似コマンドである．必ずしも入出力をこの通りに行うわけではないが，疑似コマンド中で指定されているパラメータは全て必要なものである．

3.1 証明の取扱

対話について説明する前に，まず対話的修正機構における証明の取扱について述べる．Coq では，タクティクと呼ばれる証明用のコマンドを用いて証明を行う．タクティクは提供されているものに限らず，Ltac という言語によってタクティクを定義することもでき，さらには Coq の実装言語である OCaml のコードを呼び出すことも可能である．これらの点で，タクティク毎に解釈を与えるなど，直接タクティクについて取り扱うことは難しい．

本論文で提案する対話的修正機構では，証明するタクティクの列の代わりに，証明を表す証明項 (証明する定理を型として持つ項) を証明の修正対象として用いる．Coq はカリーハワード同型を用いているため，全ての証明は項として表現されている．これにより，タクティクの取扱を考えることなく，項に関する手法によって全体の修正を行うことができる．

以前の研究[11] では，証明項を取り扱うことによって，修正する対象の数がタクティクの列を直接編集する場合に編集する箇所に比べ非常に多くなったため，元の証明を用いることで対象を減らす手法を用いた．

それに対して，我々の知る限りフィールドの追加などによって機構との対話が発生しうる箇所は (非常に複雑な自動証明タクティクなどを作成し用いない限り) ほぼ証明の修正と変わらない．これは，パターンマッチなどの展開におけるフィールドの追加の影響は，本来それらを必要としないため単に無視するだけで解決することによる．つまり，フィールドが追加されることで発生する対話は「新たな項を構築する際にどのような計算を行うか」であり，それは本来の証明においても記述されるものである．このことから，本論文で用いる対話的修正機構において証明項を用いることの欠点はそれほど大きくない．

なお，対話的修正機構による修正は，機構を組み込んだ Coq によってのみ解釈できるが，修正を読み込んだ後に内部に作られる修正後の表現は通常の Coq と完全に同じであるため，これを出力することで通常の Coq によって扱える記述を得られる．ただし，この場合証明がタクティクでなく項として得られるため，単体での可読性にやや問題がある．タクティクを考慮した合成は，タクティク記述言語である Ltac による制御文や構造の変化に対応する必要があり，非常に難しい．この問題を解消するには証明の記述に強く制約を加える必要があるが，それらに関する考察は将来課題とする．

3.2 レコード型へのフィールドの追加

レコード型の実体は単一の構築子を持った帰納型である †3．この構築子のパラメータはレコード型の各フィールドと同じであり，レコード型にフィールドを追加することは構築子にパラメータを追加することとほぼ同義である．なお，レコード型の場合は，定義時にフィールド名と同名の関数 (射影関数) が追加される点が異なっている．対話的修正機構はフィールドの追加時にこれらの関数を合わせて更新する．

対話的修正機構に対し，対話によるレコードや型クラスの拡張を命令するコマンドは以下の通りである．

```
Extend Record record_name
    : record_arity :=
  { extra_field : field_type ; ... }.
Extend Class class_name
    : class_arity :=
  { extra_field : field_type ; ... }.
```

なお，定義時には既存のフィールドは明示せず，追加するフィールドのみを記述する．例えば，第 2.2 節において挙げたものと同様に expに compileを追加する場合は以下のようになる．

```
Extend Class exp : Set :=
  { compile : list inst }.
```

このコマンドによって，対話的修正機構は expを新しいフィールド compileを追加したものとして再定義する．ただし，この時点ではインスタンスについては特に言及しない．他のクラスを拡張するなどした後，型の拡張の終了を言及するために **Deploy**コマンドを用いる．この **Deploy**コマンドを実行すると，対話的修正機構は生成されるインスタンスについて言及する．今回の拡張の場合，対話的修正機構は以下のメッセージを出力する．

```
In Econst (1 constructors)
Econst =
  fun n : nat =>
    {| eval := n; compile = _ |}
                  ^^^^^^^^^^^
```

まず，機構は Econstによって生成されるインスタンスにおける compileの値を要求する．Econstが関数になっているのは，Instanceコマンドが実際には引数からインスタンスを生成する関数だからである．機構は証明モードを使って値を求める．

```
1 subgoal

  n : nat
  ============================(1/1)
   list inst
```

ここでは，exact ([push n])のように記述することで，値を容易に記述できる．また，追加されたものが性質である場合も通常の証明として記述できる．

Econstの値を記述すると，次に Eaddによって生成されるインスタンスにおける compileの値を要求される．

†3 R というレコード型に対して構築子を指定しない場合，自動的に Build_R という名前の構築子が作成される．

```
In Eadd (1 constructors)
Eadd = fun e1 e2 : exp =>
       {| eval := eval + eval ; compile = _ |}
                                  ^^^^^^^^^^^
```

この対話は全てのインスタンス (が生成される箇所) における compileの値が定義されるまで続く．例えば 2.3 節で挙げた i2eを定義していた場合，パターンマッチの結果生成される二つのインスタンスに対しても値を要求される．なお，これらの検出される順序は定義された順である．

これらの対話が終わると，レコード型とインスタンス全てに対して拡張された状態となる．この例では，expは以下のように定義された状態となる．

```
Class exp : Set :=
 { eval : nat; compile : list inst }.
```

3.3 構築子のパラメータの追加

前述の通り，レコード型のフィールドの追加と構築子のパラメータの追加は非常に似ているが，異なる点として，構築子のパラメータは多くが無名であることが挙げられる．これは，既存のパラメータについて言及する際に問題を起こす．

この問題はパラメータについての性質を記述する際に発生する．例えば，次の型の定義は，自然数を 1 つ以上含むようにし，含まれる要素が全て 10 未満であることを保証したリスト状の型の定義である．

```
Inductive nlist : Set :=
| last :
   forall n : nat, n < 10 -> nlist
| next : forall n : nat, n < 10
                         -> nlist
                         -> nlist.
```

一方，この制約が記述されていない型を考えると以下の定義になる．

```
Inductive nlist : Set :=
| last : nat -> nlist
| next : nat -> nlist -> nlist.
```

ここで，制限をレコード型のフィールドの追加と同様に記述しようとした場合，元々の定義における nが何かを表すことができない．

これらの変数を明示するために，構築子のパラメータを全て記述し，以下のようにコマンドを定義する．

```
Extend Constructor constructor_name
    : constructor_type.
```

この記述では，全てのパラメータを**元の記述順**に明記することを強制し，また追加のパラメータはその末尾に追加することを強制する．これは二つ以上の同じ型のパラメータを持つ場合や，追加するパラメータが同じ型であるような場合に，対応する元のパラメータを判別することが不可能であるためである．パラメータが元の記述順に並んでいるかは，コマンドの実行時に機構によって確かめられる．

例えば，上に挙げた nlistへの拡張は以下のように記述する．

```
Extend Constructor last :
  forall n : nat, n < 10 -> nlist.
Extend Constructor next :
  forall n : nat, nlist
                  -> n < 10
                  -> nlist.
```

1 つ目のコマンドでは，nat->nlistが **forall** n:nat, nlistの糖衣構文であるため，容易にパラメータの順序の確認が行われる [†4]．これにより，構築子 lastに n<10という制限が追加されている．

2 つ目のコマンドにおける宣言は最初に想定した順序と異なる引数となっているが，これは機構がパラメータの順序を強制しているためである．しかし，どちらの順序であっても意味上はほぼ等価であるため，ここでは入れ替えなどについて考慮しない．

コマンドによる処理以降の対話についてはほぼレコード型の拡張と同様であるため，ここでは対話については省略する．

4 拡張に対する制限

本節では，前節で紹介した対話的修正機構における制限について述べる．この機構は保守的な拡張のみを

†4 無名パラメータに n という名前は付いていないが，アルファ等価な項である．

許す．つまり，本節で述べる制限を満たせば，拡張後のプログラムは Coq において well-typed となっているように制限をつける．方針としては，型付け規則に対する影響を可能な限り局所化することで，全体の型について整合性を保つ．

今回の拡張では，実際に拡張する構築子と，拡張する構築子を持つ型に影響を与える．したがって，これらの型付け規則のみに影響を留めることで全体の拡張を行うことを目標とする．なお，各節の記述に関する厳密な定義は Coq の Reference manual[3] 4.5 節を参照されたい．

また，本節の終わりに構築子だけでなく関数一般に適用する場合の問題について分析を行う．

4.1 陽性条件

帰納型の定義では，全ての構築子が陽性条件 (positivity condition) を満たす必要がある．直観的には，この制約は定義時に関数の引数などとして定義する型自身が出現してはならない，というものである．

陽性条件の判定には他に 2 つの概念が出現する．厳密陽性出現 (strictly positively occuring) と入れ子陽性条件 (nested positivity condition) である．

本研究では，帰納型の定義を変更するため，それ自身の影響の他，他の定義が出現する箇所に影響がある．これらの条件のうち，厳密陽性出現および入れ子陽性条件では帰納型およびその構築子を対象とした条件が存在する．レコード型へのフィールドの追加や，構築子へのパラメータの追加により，拡張する型だけでなく，その型を利用した他の型が影響を受ける場合がある．

例として，次のレコード型を考える．

```
Record X (A : Set) : Set :=
 { elem : list A; len : nat }.
```

要素の長さを保存したレコード型を意図している．このレコード型を用い，新しい帰納型を定義する．

```
Inductive W : Set := w : X W -> W.
```

この定義は Coq で問題なく記述できる．このレコード型はリストの長さとして自然数を保持していたので，そのことを示す根拠を追加した定義を考える．

```
Record X (A : Set) : Set :=
  { elem : list A; len : nat;
    _ : length elem = len }.
```

このレコード型の定義は可能だが，しかし再度 `W`を定義しようとすると，Coq は以下のエラーメッセージを出力する．

```
Error: Non strictly positive occurrence of
       "W" in "X W -> W".
```

これは，等価性を表す=が帰納型であり，その中で `W`が (`length`の暗黙の引数として) 出現していることが，入れ子陽性条件に反することを Coq が検出したものである．この例は，対話的修正によるフィールドの追加という形で性質を追加すると，他の定義が不可能となる場合があることを意味している．

この問題を避けるために，拡張時には，その型を**利用する型が陽性条件を満たすかを確認**し，満たさない型が存在する場合には拡張を拒否する．拡張がこの制限によって拒否される例は少なくないが，拒否される拡張の結果は本来 Coq において記述不可能なものとなるため，制限そのものを緩めることはできない．

4.2 構築子の出現

機構では新たなパラメータやフィールドの追加を行っているが，出現位置における型を保つために，構築子のパラメータは全て出現位置で与えられているものとする．したがって，例えば以下の記述がある場合，`Iadd`へのパラメータの追加は許されない．

```
Definition ie_add1 : iexp -> iexp :=
  Iadd (Iconst 1).
```

本定義中では，`ie_add1`が 2 つの構築子 `Iadd`と`Iconst`を用いている．しかし，`Iadd`は 2 つのパラメータを取るが，ここでは 1 つしか与えられていない．もしもここで `Iadd`にパラメータを追加すると，`ie_add1`の型が追加したパラメータを余計に必要とする関数となり，型が変わってしまう．`ie_add1`の型を変化させ，その出現位置全てに関して変更を要求することも考えられるが，それぞれにおいて型付けや要求する内容をどのようにすべきかは明らかではない．

このような変更の波及を避けるために，必ず構築子出現位置において追加するパラメータの値を要求するものとする．

この制限はレコード型についても同様である．レコードの記述では，型クラスの Instanceコマンドを用いた場合をのぞき，常に全てのフィールドが与えられている．また，Instanceコマンドもフィールドの一部を省略できるが，その直後に省略したフィールドの値を要求されることから，実際には出現位置で与えられている．したがって，レコードや型クラスの記法を用いる限り，この制限について問題となることはない．

なお，以下のように書くことで制限を回避し，ie_add1を記述しながら Iaddを拡張することはできる．

```
Definition ie_add1 (e : iexp)
    : iexp :=
  Iadd (Iconst 1) e.
```

この場合には，追加のパラメータを eを用いて記述することになる．このような書き換えは機械的に行うことも可能だが，今回はその書き換えは導入しない．

4.3 パターンマッチの型

Coq では，帰納型の構築子は構築時の他，パターンマッチにおける展開において出現する．本節で説明する制限はこのパターンマッチの結果返す型に関するものである．

原則として，Coq における **Prop**型 (**Prop**に属する帰納型) の展開では **Prop**型自身や **Prop**型の要素を返す必要がある．これは Coq がもつプログラム抽出の機能による制限である．どのようにプログラム抽出に影響するかについては省略するが，この原則には例外があり，その例外への制約を考える必要がある．

Prop型の展開を行う場合であっても，その型が**空の定義** (*empty definition*) であったり**単一要素の定義** (*singleton definition*) である場合には他の値を返してよいことになっている．ここでいう空の定義とは構築子を持たない型であり，例えば矛盾を表す Falseがある．また，単一要素の定義とは，ただ一つの構築子を持つ型であり，かつ構築子のパラメータが全て **Prop**型やその要素である場合を指す．単一要素の定義としては例えば Trueや論理積を表す andなどがある．

空の定義については今回影響はないが，レコード型は単一要素の定義となりうるので，**Prop**に属するレコードに関してパターンマッチを行い，自然数を返すような記述がある場合，そのレコードは **Prop**型以外の要素を含んではならない．この場合，構築子へのパラメータの追加によって自然数を追加すると例外に当てはまらなくなる．そのため，機構では**単一要素の定義に対する拡張は *Prop*型などに限定し，例外を逸脱しない範囲に制限**する．一般に単一要素の定義は特殊な場合が多く，通常の性質にはあまり出現しないため，この制限自体が影響を与えるとは考えにくい．

4.4 ソート多相

帰納型は，パラメータを与えると必ずソート (**Set**, **Prop**, **Type**のどれか) に属する．ここでは，宣言に記述されたソートを結果のソートと呼ぶことにする．また，パラメータや結果のソートを含めた帰納型の宣言における型をアリティと呼ぶ．

ソート多相は，結果のソートを可能な限り小さくしようとする機能である．ソート同士には大小関係があり，**Prop**，**Set**，**Type**の順に大きくなる．Coq の型付け規則では大きいソートへの変換規則があるため，より小さなソートとみなしておく方が利用範囲が広がる．

ソート多相は，アリティに含まれるパラメータに応じて，実際に置き換えた場合にどのソートとしてみなせるかを判定する．典型的な例は listであり，その宣言は以下のようになっている．

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

この中で，listのアリティは **Type->Type**である．ここで変換規則を用いて自然数の型 natが **Type**型に含まれると見なせることから，list natが記述できる．もし listの定義で Aを natに置き換えると，結果のソートを **Type**より小さい **Set**として宣言することができる．この場合，ソート多相によって list natの

結果のソートは **Set**として扱われる．

しかし，構築子へパラメータを追加すると，この振る舞いが変化する可能性がある．例えば consを次のように拡張する．

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

この構築子によって，Aの要素の直後に任意の型の要素を挿入することができるようになる．しかし，この結果 list natはもはや **Set**とは見なされなくなる．

一方で，次のように拡張する場合は特にソート多相の振る舞いを変えない．

```
cons : A -> list A -> list A
                   -> list A
```

このとき，consは 2 つのリストをとることで，連結したかのような表現が可能となる．しかし，この場合特に型として表現できる範囲が大きくなったわけではなく，list natは **Set**のままである．

対話的修正機構における拡張は保守的であるため，このようにソート多相の振る舞いを変化させるような拡張は許すことができない．そのため，機構は構築子を拡張しようとする際に，ソート多相の振る舞いが保たれているかを確かめる．幸い，Coq では内部においてソート多相がどのように振る舞うかを定義時に計算しているため，その様子が変化しているかを調べるだけで容易に確認できる．もし変化していた場合，機構はその拡張を拒否する．

ただし，この制限は非常に大きな問題を含んでいる．レコードでは，新たなフィールドを追加する際に，それまで全く関係のなかった型の要素を含めることが多々ある．これは本論文で用いている expの場合にもあり，compileを追加した後はその型の一部である instに関係を持っている．これは，もしも instが **Type**の要素であり，かつ **Set**と見なせない場合，拡張後は expの型が **Set**と見なせないことを意味している．今回の例ではそれぞれが **Set**であるため特に問題ないが，この拡張による依存関係による影響がどの程度問題となるかについて今後調査する必要がある．

4.5 帰納原理関数

Coq において帰納型を定義すると，自動的に帰納法のための関数 (以下帰納原理関数) が定義される．例えば iexpを定義した場合には，iexp_ind，iexp_rec，iexp_rectの 3 つが定義される．それぞれの関数は **Prop**，**Set**，**Type**をそれぞれ返すための構造的帰納法を表している．例えば iexp_indの型は以下のように表される †5．

```
iexp_ind :
 forall P : iexp -> Prop,
   (forall n : nat, P (Iconst n)) ->
   (forall i : iexp, P i
      -> forall i0 : iexp, P i0
                 -> P (Iadd i i0)) ->
   forall i : iexp, P i
```

ここで Pは帰納法で示す命題を表している．また，2 つ目と 3 つ目のパラメータはそれぞれ Pにおける定数の場合の証明と加算の場合の証明である．

それぞれの証明では，nや iのようにパラメータが存在する．これらは構築子のパラメータや，パラメータに対する帰納法の仮定を表している．したがって，構築子の拡張を行うと，帰納原理関数の型も同様に変化する．例えば Iconstに boolの要素を追加すると，iexp_indは以下のように変化する．

```
iexp_ind :
 forall P : iexp -> Prop,
   (forall (n : nat) (b : bool),
     P (Iconst n b)) ->
   (forall i : iexp, P i
      -> forall i0 : iexp, P i0
                 -> P (Iadd i i0)) ->
   forall i : iexp, P i
```

2 つ目のパラメータである定数に対する証明が bを多く取るようになっている．このような場合，既存の証明の変更としては追加のパラメータを受け取り，無視すれば良い．しかし，構築子の出現と同様に，もしもこの引数を渡さずに高階関数として扱った場合，型の変化が起こり，それが波及してしまう．これを避ける

†5 他の 2 つは Prop を Set や Type に置き換えたものである．

ために，帰納原理関数は全ての引数を与えることを要求する．

通常の帰納型における帰納原理関数はまれにこの制限に影響を受けるが，レコード型では再帰的な構造がなく，帰納原理関数を用いることはほとんどない．そのため，構築子に関する問題と異なりほぼ影響はない．

4.6 関数のパラメータへの追加

最後に関数のパラメータを追加することを考える．ここまでに，構築子へのパラメータについて考えたが，同様に考えることで関数へのパラメータの追加を考えることもありうる．幸い，関数であれば同様のアイディアによって追加は可能である．ただし，制約に関しても同様とする必要があるため，関数には全ての引数を与える必要があり，高階関数としての使用について非常に強い制限となる．例えば，拡張する関数は`map`関数の引数として使用できない．

高階関数については制限が強すぎるため，この制限を緩める方法を考える必要がある．型の変化を許さない緩め方としては，拡張した引数のみを，高階関数として出現した箇所で与えることで，多くの場合に問題を回避できる．しかし，拡張した引数は必ず最後に出現するため，これらのみを与えるには，例えば η 拡張によって関数抽象の形へ変換した上で与える必要がある．η 拡張によって得られた関数は外延的等価ではあるが，Coq で広く用いられる Leibniz 等価ではないため，証明が拡張前と同様となるかは定かではない．よって，他の規則からこれらを正当な変換と判定できるか，またはどのような制限下で同様の証明となるかについて議論する必要がある．また，型の変化を許す場合には，その波及がどのようになるかについて全体を分析する必要がある．このような手法についての検討，構築は将来課題とする．

5 関連研究

5.1 フィールドの追加

拡張可能なレコード型については多くの研究が成されている．レコード型は部分型とともに用いることで，多くのフィールドを持つインスタンスを少ないフィールドだけのレコード型としてみなすことができる．また，多相レコード[13] では，各フィールドに対する射影関数が多相になっており，より広いレコードを扱うことができる．Extensible Records[7] では，ラベルを重複して含むことが可能となっているなど，インスタンスがより柔軟に扱われている．

本論文で提案した対話的修正機構の主な目的はレコード型自体を拡張することであり，レコード型やそのインスタンスに柔軟さを与えることではない．上に挙げたレコードの拡張とは主に実行時に発生するインスタンスの変化であり，本研究の目的は記述したプログラム全体の変化を可能とするという点で大きく異なっている．

5.2 アスペクト指向

この機構の機能は，OCaml にアスペクト指向拡張を施した Aspectual Caml[9] の影響を受けている．Aspectual Caml はポイントカット・アドバイスモデルに基づくアスペクト指向を採用している．パラメータの追加やフィールドの追加は Aspectual Caml において open class として実現されており，これらの要素やインスタンスについての操作はアドバイスやデフォルト引数によって記述する．その他にも関数の本体を置き換えるなどの処理を実現している．しかし，アドバイスを記述するには適切な箇所へのポイントカットを記述する必要があり，本機構のように処理系から提示することはない．

アスペクト指向として考えた場合，対話的修正機構はアスペクトを構成するためのシステムと見られる．この視点により，一連の拡張をモジュールとして見ることができる．このように変更や拡張をモジュールとして適用することを主としたアスペクト指向のフレームワークとして，コンパイラフレームワークの JastAdd[6] がある．JastAdd では，コンパイルする対象言語の新しい文法のサポートや性質の追加をアスペクトを通じて行っている．

JastAdd の視点でモジュールとしての対話的修正機構との対話を見た場合，2 つの変更に対する合成を考え，それをサポートすることが考えられる．以前の研究である帰納型への構築子の追加に対して，拡張す

るコードと元の系との合成を考えた[19]が，パラメータの追加については考えていないため，これについても考察したい．

5.3 Expression problem

Delaware らの提案した MTC[4] は，Coq におけるデータ型や証明の部品化および結合手法である．MTC ではデータ型の一部とその部分に関する関数や性質をまとめた部品を型クラスのインスタンスとして作成し，それらの結合を自動化することで，全体の構築を半自動的に行う．この手法は既存の Coq の機能を用いて実現されており，本研究のように Coq へ新たな機能を追加する必要はない．また，部品同士の結合や部品の拡張は柔軟にでき，expression problem における両方向への拡張が可能となっている．一方，部品の記述には結合のためのパラメータや型を要するため，記述しようとしているシステムに比べて各要素が複雑になる．これは，Coq における元の記述に可能な限り近づけることでプログラマの負担を減らす対話的修正機構の考え方と対照的である．また，MTC ではパラメータの追加については対応できない．

他の言語における expression problem への解法は言語に対する新たな機能として対応することが多い．例えば OCaml では多相ヴァリアント[5]が使われている．多相ヴァリアントでは構築子とパターンマッチが通常のヴァリアントとほぼ同様に利用できる．しかし，それぞれの構築子自体は多相であるが，一度記述した関数について多相ではなく，パラメータの追加などは難しい．

Open data types と open functions[8] は Haskell における解法の 1 つである．多相ヴァリアント同様，これらはほぼ通常の関数とパターンマッチと同様の記法を利用しているが，型クラスと同様，網羅性を議論することが難しい．実際，open functions における一般的な網羅性の判定はプログラム全体が決定されるビルド時に行っており，Coq のように全体を決定できない場合には応用できない．

これらの構造は型や関数を常に開いており，いつでも変更，拡張が可能となっている．しかし，型が開いている限り，網羅性の検査は難しく，一方閉じた構造では変化が難しい．本研究では，対話を通じてレコード型や帰納型の定義を開き，変更し，そして対話を終わらせることで閉じている．閉じた後は再度開かない限り拡張できないので，その時点での列挙は容易であり，それゆえに網羅性検査も可能となっている．

5.4 型の整合性

パラメータの一部を要求するという点では，Agda[12] において用いられるホールが近い働きをする．Agda のホールは，その箇所に何かが入るとした上で全体の型検査を行う．これにより，記述に不足がある場合にも全体の検査が可能であり，ホールを項で置き換えていくことで順次全体の構成を行える．対話的修正機構による対話は，パラメータの追加のように変更をこちらが指示するものの他に，不足する箇所，影響を受ける箇所を順次記述するものがある．影響を受ける箇所の記述は，ホールを置き換える項の記述と対応する．

対話的修正機構では，Agda のホールと異なり，追加した要素に関するもののみを，本来宣言されている箇所の外部に記述する形式となっている．ホールのように定義する際には用いることはできないが，変更を別途記述することで，編集できない箇所 (既存のコードだけではなく，手元にコードがないライブラリなど) に対しても適用できるという利点がある．本研究では Coq を対象としているため，実際には項を作っている証明モードを利用することで外部に記述する箇所を特別に扱っている．外部に記述する要素はやはりある種のホールとして扱えるため，Agda において対話的修正機構を実現する際には，外部に記述可能なホールを自動生成することが妥当と考えている．

また，ソースコードの型の整合性をとるという観点では型デバッグが関係する．型デバッグでは，型エラーが発生した際に，プログラムのどの箇所が利用者の意図と異なるかを調査するために型の依存関係を計算したり[10]，利用者と対話することで異なる箇所を特定する型デバッガなどの研究が行われている[1]．特に型デバッガでは，コンパイラの機能を利用する形で実現するものもあり[17]，処理系と同じ挙動を利用するという動機は対話的修正機構を組込みとして実

現する理由と近い．

対話的修正機構では，拡張前の型は整合しているという仮定の下で，拡張時におこる型の不整合を検出する仕組みであるため，型デバッガと異なり，利用者に型が意図通りの箇所と意図通りでない箇所を尋ねるなどの作業が必要ない．一方で，型デバッガは引数の数が合わない場合以外にも様々な型エラーに対して適用可能だが，対話的修正機構のとる手法は可能な変更，不整合の解決方法が強く制限されている．より柔軟な変更への対応として，例えば 4.6 節における問題の解決のために型や項そのものの変更を許す場合に，型デバッガにおける対話 (アルゴリズミックデバッギング [14] の応用) を適用することで，利用者の負担を最小限にして修正箇所の特定ができると考えている．

6 まとめと今後の課題

本論文では Coq に対する対話的修正機構の新しい機能として，レコード型や構築子に新しいフィールド，パラメータを追加する手法を提案した．この機構ではレコードや構築子への追加のためのコマンドや，修正すべき箇所を検出する方法が組み込まれている．これにより，既存の記述の書き換えなしに，レコード型の定義とその全てのインスタンスについて追加したフィールドを持っていることを保証できるようになった．

対話的修正機構は，Coq に対するアスペクト指向拡張と捉えることもできる．つまり，対話的修正機構を用いて拡張を施すことで，元の記述を変更しないため，容易に拡張部分を切り離すことができる．

現状，レコード型や構築子に対する拡張では，2 つの拡張方法を織り交ぜている．1 つはコマンドによる型の拡張であり，もう一方は対話による値の記述である．後者における追加する場所は，残念ながら機構のみが知っているため，対話の結果残る記述には全く現れない．その結果，例えば元の体系の記述の変更後に拡張の記述を用いようとすると，そのまま用いられる箇所とそうでない箇所が分からないなど，再利用性が著しく低いことが分かる．また，2 つの拡張を 1 つの体系に同時に適用することも難しい．拡張箇所の明示や拡張の合成については将来課題とする．

参 考 文 献

[1] Chitil, O.: Compositional Explanation of Types and Algorithmic Debugging of Type Errors, in *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, ICFP '01, New York, NY, USA, ACM, 2001, pp. 193–204.

[2] The CompCert project, http://compcert.inria.fr/.

[3] The Coq Proof Assistant Reference Manual Version8.4, http://coq.inria.fr/distrib/current/files/Reference-Manual.pdf.

[4] Delaware, B., Oliveira, B. C. d. S. and Schrijvers, T.: Meta-theory à la carte, in *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Giacobazzi, R. and Cousot, R.(eds.), ACM, 2013, pp. 207–218.

[5] Garrigue, J.: Programming with polymorphic variants, in *ACM SIGPLAN Workshop on ML* , informal proceedings, 1998.

[6] Hedin, G. and Magnusson, E.: JastAdd - a Java-based system for implementing front ends, *Erectric Notes in Computer Science*, Vol. 44, No. 2(2001) pp. 59–78.

[7] Leijen, D.: Extensible records with scoped labels, *Trends in Functional Programming*, van Eekelen, M. C. J. D.(ed.), Vol. 6, Intellect, 2005, pp. 179–194.

[8] Löh, A. and Hinze, R.: Open data types and open functions, in *ACM SIGPLAN international conference on Principles and practice of declarative programming*, 2006, pp. 133–144.

[9] Masuhara, H., Tatsuzawa, H. and Yonezawa, A.: Aspectual Caml: an aspect-oriented functional language, in *ACM SIGPLAN International Conference on Functional Programming*, 2005, pp. 320–330.

[10] McAdam, B. J.: Generalising Techniques for Type Debugging, in *Selected Papers from the 1st Scottish Functional Programming Workshop* (*SFP99*), SFP '99, Exeter, UK, UK, Intellect Books, 2000, pp. 50–58.

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

[12] Norell, U.: Towards a practical programming language based on dependent type theory, PhD Thesis, Chalmers University of Technology, 2007.

[13] Ohori, A.: A Polymorphic Record Calculus and Its Compilation, *ACM Trans. Program. Lang. Syst.*, Vol. 17, No. 6(1995), pp. 844–895.

[14] Shapiro, E.: *Algorithmic Program Debugging*, MIT Press,, 1983.

[15] Sozeau, M. and Oury, N.: First-Class Type Classes, in *21st International Conference Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science, Vol. 5170, 2008, pp. 278–293.

[16] Torgersen, M.: The Expression Problem Revisited, in *Proceedings of European Conference on Object-Oriented Programming* (*ECOOP'04*), Lecture Notes in Computer Science, Vol. 3086, Springer Verlag, 2004, pp. 123–146.
[17] 対馬かなえ, 浅井健一: コンパイラの型推論を使用した型デバッガの提案, コンピュータソフトウェア, Vol. 30, No. 1(2013), pp. 180–186.
[18] 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.
[19] 森口草介, 高橋和子: 対話的修正と被修正系との合成手法, 日本ソフトウェア科学会 第 31 回大会, 2014.

森口草介

2013 年 3 月東京工業大学博士後期課程修了，博士 (工学)．同年神奈川大学外部資金雇用研究員．2014 年関西学院大学博士研究員．2015 年関西学院大学特別研究員，現在に至る．プログラム記述の技法とシステムの検証技術を元とした，検証のソフトウェア工学への応用に興味を持つ．日本ソフトウェア科学会，情報処理学会，ACM，各会員．

高橋和子

1982 年京都大学理学部卒．同年三菱電機株式会社入社．同社中央研究所等を経て 2000 年 4 月関西学院大学理学部助教授．2006 年 4 月関西学院大学理工学部教授，現在に至る．1997 年から 1999 年 ATR 音声翻訳通信研究所研究員．1994 年京都大学博士 (工学)．証明支援系とシステム検証技術，時空間に関する知識表現と推論機構に興味を持つ．日本ソフトウェア科学会，情報処理学会，電子情報通信学会，人工知能学会，各会員．