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・通信工学　専攻　博士前期課程		
氏　　名	木下　大輔	学籍番号	1531033
論 文 題 目	部分的なプログラムの変更を考慮した証明支援		

要　旨

プログラムの任意の性質を保証し，バグのないプログラムを開発するために，Coq などの証明支援系と呼ばれるソフトウェアを用いる方法がある．Coq を用いて性質を保証したプログラムを開発する方法には，2 通りの方法がある．1 つ目は Coq 上にプログラムを直接記述し，証明を行い，他言語へ変換する方法．2 つ目は他言語で記述されたプログラムを Coq プログラムへ変換し，証明を行う方法である．しかし，どちらの方法であっても単方向の変換のみ行うため，前者の方法では他言語を直接変更できないため仕様が決定できず，後者の方法では証明を行いやすいように Coq 上でプログラムへ変更を加えることができない，という問題点がある．

本研究では，この問題を解決するため，OCaml プログラムと Coq プログラムの双方向の変換を可能とするシステムを提案する．これにより，OCaml プログラムの変更を Coq プログラムへ反映することで，OCaml 側で仕様が決定できると同時に，証明を行いやすくするために Coq 側でプログラムに加えた変更を，OCaml プログラムへ反映することができる．さらに，変更前の Coq プログラムを利用し，差分の小さい Coq プログラムを生成することで，以前の証明を部分的に再利用できることが期待される．また，OCaml の List モジュールを本システムで変換することで，Coq で扱うことのできる構文に変換される OCaml プログラムに対し，対応していることを示した．

平成 28 年度修士論文

部分的なプログラムの変更を考慮した証明支援

電気通信大学
情報理工学研究科 情報・通信工学専攻
コンピュータサイエンスコース

学籍番号　　: 1531033
氏名　　: 木下 大輔
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　　: 2017/1/30

要旨

プログラムの任意の性質を保証し，バグのないプログラムを開発するために，Coq などの証明支援系と呼ばれるソフトウェアを用いる方法がある．Coq を用いて性質を保証したプログラムを開発する方法には，2 通りの方法がある．1 つ目は Coq 上にプログラムを直接記述し，証明を行い，他言語へ変換する方法．2 つ目は他言語で記述されたプログラムを Coq プログラムへ変換し，証明を行う方法である．しかし，どちらの方法であっても単方向の変換のみ行うため，前者の方法では他言語を直接変更できないため仕様が決定できず，後者の方法では証明を行いやすいように Coq 上でプログラムへ変更を加えることができない，という問題点がある．

本研究では，この問題を解決するため，OCaml プログラムと Coq プログラムの双方向の変換を可能とするシステムを提案する．これにより，OCaml プログラムの変更を Coq プログラムへ反映することで，OCaml 側で仕様が決定できると同時に，証明を行いやすくするために Coq 側でプログラムに加えた変更を，OCaml プログラムへ反映することができる．さらに，変更前の Coq プログラムを利用し，差分の小さい Coq プログラムを生成することで，以前の証明を部分的に再利用できることが期待される．また，OCaml の `List` モジュールを本システムで変換することで，Coq で扱うことのできる構文に変換される OCaml プログラムに対応していることを示した．

目次

1　はじめに

1.1　背景

プログラミングにおいて，バグがないことが保証された安全なプログラムを作ることは，プログラムの品質の向上という面において重要である．それに対し，Coq [1] などの証明支援系と呼ばれるソフトウェアを用いて，プログラムの任意の性質を証明することで，安全性を示すという方法がある．証明支援系を用いると，ヒューマンエラーを防ぐ，全ての入力を網羅した保証を行うことができる，テスト工程を削減できるといった利点もある．

Coq は，関数型言語に似たプログラムとそのプログラムが満たすべき性質を記述し，性質について人間が対話的に証明を行うものである．今まで，変換にバグが含まれないことを保証した C コンパイラである CompCert の開発 [2]，D-Bus メッセージと JSON の相互変換における正当性を保証した iZE Smart Desktop の開発 [3]，Java Card のセキュリティの検証 [4] といった成果を挙げている．

Coq を用いることで，任意の性質について保証した証明付きのプログラムを開発することができ，その開発方法には次の 2 通りがある．1 つ目は，Coq 上にプログラムを記述し証明を行い，Coq が提供する他言語への変換機能を用いてそのプログラムを変換する方法．2 つ目は，他言語でプログラムを記述し，それを Coq のプログラムに変換して証明する方法である．

Coq を用いたプログラムの開発では，仕様変更により他言語プログラムに変更を加える場合と，証明を行いやすいように Coq 上でプログラムに変更を加える場合が想定される．しかし，前述のどちらの方法であっても単方向の変換しか考慮していないため，一方では他言語プログラムを直接変更できず，もう一方では Coq 上でプログラムへ変更を加えることができないという問題点がある．そこで本研究では，この問題を解決する手法を提案する．

1.2　目的と方針

本研究は，部分的な変更を考慮した，他言語プログラムに対する Coq における証明の支援を目的とする．ここで他言語は，Coq やファイル同期ツール Unison など，広い分野で実用的に利用されている言語である OCaml とする．

方針として，OCaml プログラムと Coq プログラムの双方向の変換を可能にする．これにより，OCaml のプログラムへの変更を Coq プログラムへ反映することで，OCaml 側でプログラムの仕様を決定でき，また，Coq 側で証明を行いやすくするために関数に変更を加えた際，それを OCaml プログラムへ反映することができる．

さらに，変更前の Coq プログラムを利用し，差分の小さい Coq プログラムを生成することで，

以前の証明を部分的に再利用できるようにする．

1.3 本論文の構成

2 章では，Coq を用いた証明付きプログラムの開発の方法と，現状の問題点について述べる．3 章では，双方向プログラミング，及び，本システムで利用する双方向変換プログラミングに基づいたプログラムについて述べる．4 章では，提案するシステムの設計，動作例及び対応している OCaml と Coq のサブセットについて述べる．5 章では，提案システムの実装について述べる．6 章は評価として，提案システムの記述力および，既存の類似ツールとの変換における速度比較について述べる．7 章では，関連研究について述べる．最後に 8 章では，本論文のまとめと今後の課題について述べる．

2　Coq を用いた証明付きプログラムの開発

Coq を用いて，OCaml プログラムの任意の性質を保証する方法は，以下の 2 通りが考えられる．

方法 1　Coq でプログラムを記述，証明し，OCaml へ変換する方法
方法 2　OCaml プログラムを Coq へ変換し証明する方法

以降では，Coq におけるプログラムの性質の証明の流れについて述べた後，それぞれの方法について，利点と問題点を議論する．

2.1　Coq によるプログラムの性質の証明

Coq でプログラムの証明を行う場合，まず以下のように証明を行うプログラムを Coq 上に記述する．

```
1 Fixpoint reverse (l:list A) :=
2   match l with
3   | nil => nil
4   | cons x xl => (reverse xl) ++ [x]
5   end.
```

reverse はリストを反転させる関数である．次に，reverse が満たすべき性質を Coq 上に記述する．

```
1 Fixpoint reverse (l:list A) :=
2   match l with
3   | nil => nil
4   | cons x xl => (reverse xl) ++ [x]
5   end.
6
7 Theorem rev_rev :
8   forall (l:list A), reverse (reverse l) = l.
```

ここでは，「reverse を二回適用すると，元のリストとなる」という性質 rev_rev を記述した．ユーザはこの性質を，タクティクと呼ばれるコマンドを用いて，Coq 上で対話的に証明を行う．

```
1 Fixpoint reverse (l:list A) :=
2   match l with
3   | nil => nil
4   | cons x xl => (reverse xl) ++ [x]
5   end.
6
7 Theorem rev_rev :
```

```
 8   forall (l:list A), reverse (reverse l) = l.
 9 Proof.
10   induction l; simpl.
11   - reflexivity.
12   - assert (forall (l:list A) a,
13               reverse (l ++ [a]) = a :: (reverse l)).
14     { induction l0; intros; simpl;
15       [| rewrite IHl0]; reflexivity.
16     }
17     rewrite H, IHl.
18     reflexivity.
19 Qed.
```

プログラム中の induction や reflexivity などがタクティクである．一般的に証明は，Proof. と Qed. の間に記述する．

このように，Coq 上にプログラムとそのプログラムが満たすべき性質を記述し，その性質を満たすことを証明することで，性質を保証したプログラムを開発することができる．

2.2 Coq から OCaml に変換する方法

方法 1 は，Coq から OCaml へ変換する方法である．この方法では，証明したいプログラムを直接 Coq 上に実装する．そして，実装したプログラムについて任意の性質を証明した後，Coq に標準的に備わっている他言語変換機能 Extraction を用いて，OCaml プログラムへ変換することで，その性質を保証した OCaml プログラムを得る．この方法は，Coq を用いる際の一般的な

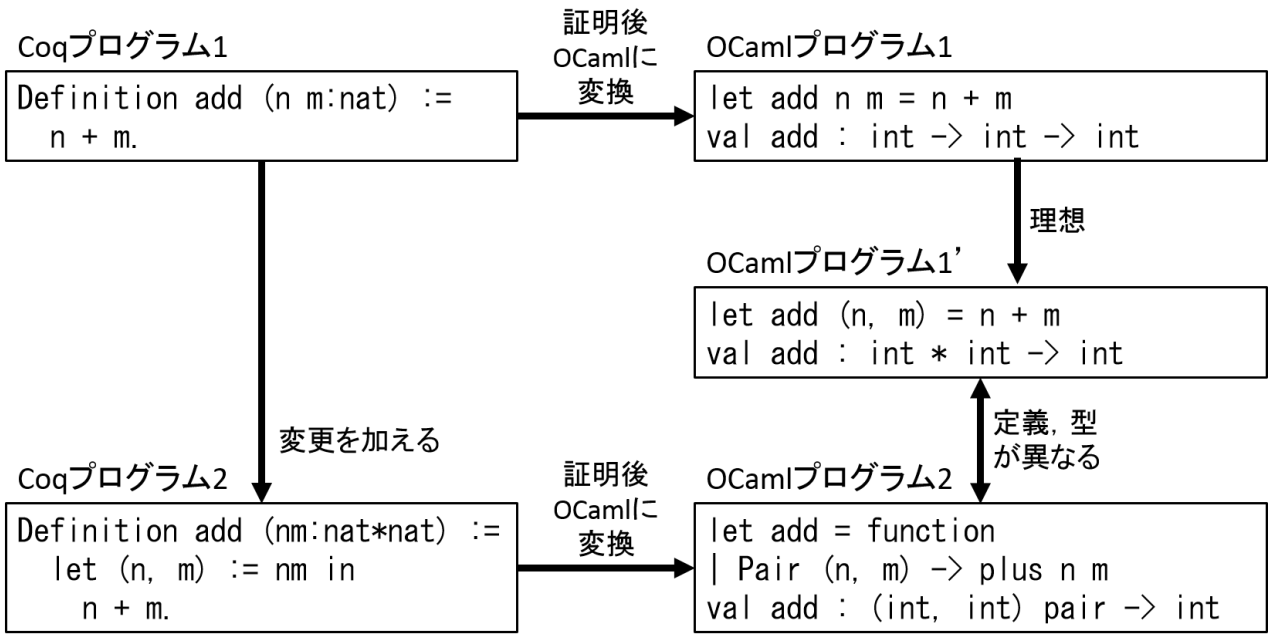

図 2.1　Coq から OCaml に変換する際の問題点

方法である．

この方法では，証明を行いやすいように Coq 上でプログラムに変更を加えることができ，証明が完了したプログラムに仕様変更がある場合には，最低限の変更を Coq 上で反映すれば良いため，既存の証明を部分的に再利用することができる．

しかし，方法 1 では次のような問題点が挙げられる．図 2.1 では，二つの自然数の和を返す Coq プログラム 1 を変換し，OCaml プログラム 1 を得ている．ここで，OCaml プログラム 1 を OCaml プログラム 1' の様に，引数を組で受け取り，その和を返す形式に変更したい場合を考える．この方法では，元々が Coq プログラムのため，OCaml プログラム 1 を直接変更してしまうと証明した性質が保証されなくなってしまう．したがって，Coq プログラム 1 を変更し，改めて OCaml プログラムへ変換する必要がある．しかし，Coq では仮引数の定義において，組の要素に変数を束縛することができない．そのため，自然数の組を受け取ってその和を返す定義とするには，Coq プログラム 2 の様に関数の内部で組の読み出しを行い，組の各要素に変数を束縛する必要がある．ところが，この Coq プログラム 2 を OCaml プログラムへ変換すると，理想としていた OCaml プログラム 1' とは型の異なる OCaml プログラム 2 に変換されてしまう．これは，Coq の内部で組は構文ではなく `pair` という型として定義されているためである．ただし，この例では Coq の `ExtrOcamlBasic` ライブラリを利用することで型が異なるという問題は解決することができる．

このように方法 1 では，OCaml プログラムに変更を加えたい場合に，まず Coq プログラムに変更を加え，それを再度 OCaml プログラムへ変換するため，OCaml の関数を直接変更することができない．さらに，変換して得られた OCaml プログラムが，必ずしもユーザの望んでいる定義や型になっているとは限らないため，OCaml 側で仕様を決めることができない．

2.3 OCaml から Coq に変換する方法

方法 2 は，OCaml から Coq へ変換する方法である．この方法では，証明したいプログラムを OCaml で記述し，それを Coq of OCaml [5] 等の既存の変換システムを用いて Coq プログラムに変換する．そして，得られた Coq プログラムを証明することで，元の OCaml プログラムの任意の性質を保証する．

この方法では，OCaml で記述したプログラムを Coq プログラムへ変換するため，性質を保証したいプログラムの仕様を OCaml 側で決めることができる．また，既存の OCaml プログラムを，Coq 上に再実装する手間を省くことができる，という利点もある．

しかし，方法 2 では次のような問題点が挙げられる．図 2.2 は，Coq of OCaml によって OCaml プログラムを Coq プログラムへ変換した例である．階乗を求める右の OCaml プログラム 3 を変換した結果，左の Coq プログラム 3 が得られる．OCaml プログラム 3 は，`int` 型（整数型）を受け取り，`int` 型を返す関数である．変換によって得られた Coq プログラム 3 は，`Z` 型（整数型）を受け取り `M [ Counter; NonTermination ] Z` 型を返す関数となっている．`M [`

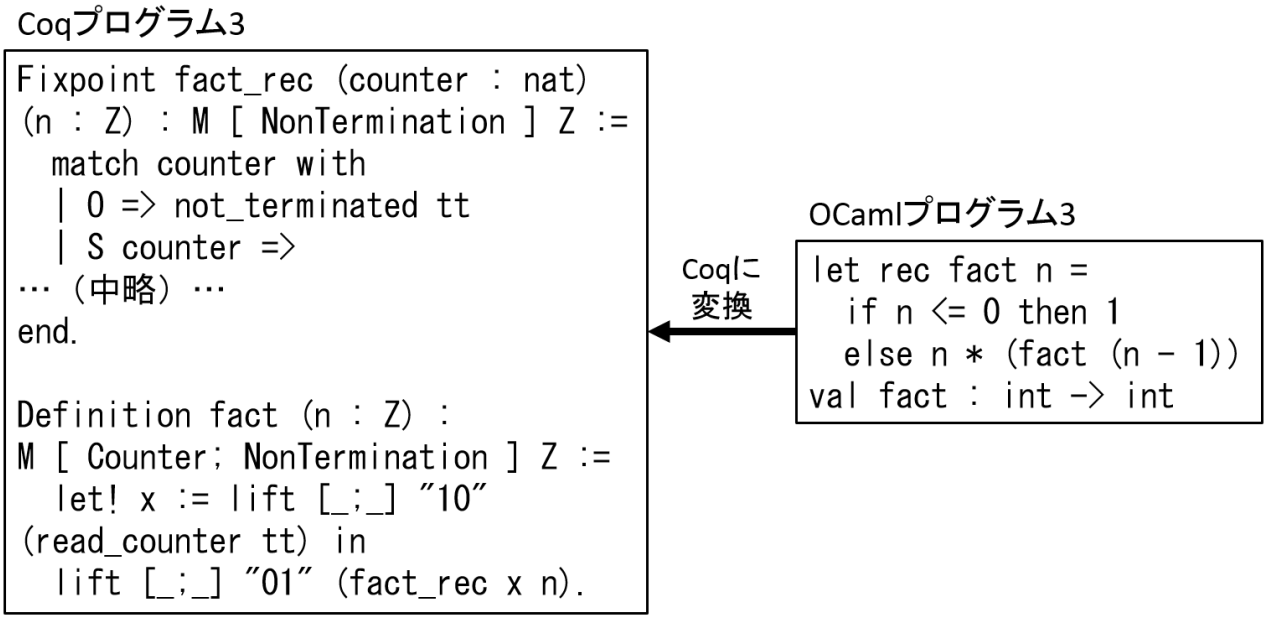

図 2.2　OCaml から Coq に変換する際の問題点

Counter; NonTermination] Z は，返り値は整数型だが，停止するかどうか分からないことを示している．これは，本来 Coq で扱えない停止しない可能性のある関数を Coq 上で記述できるようにするため，このような型の変化が発生したが，返り値の型が元の OCaml プログラム 3 とは大きく異なっている．また，プログラムの構造も複雑になっており，このように変換された Coq プログラムの性質を証明するのは簡単ではない．そして，証明しやすいように Coq 上で変更を加えると，OCaml プログラム 3 と Coq プログラム 3 の対応関係が崩れてしまう．したがって，この方法で得られた Coq プログラムを Coq 上で変更することは意味がないのでできない．

このように方法 2 では，構造の大きく異なるプログラムに変換されてしまうことがある．また，元々が OCaml プログラムのため，証明を行いやすいようにプログラムに変更を加える場合には，元の OCaml プログラムへ変更を加える必要があり，直接 Coq 上でプログラムを変更することができない．

3　双方向変換プログラミング

本章では，双方向変換プログラミングの概要について述べた後，本システムで双方向変換を行うのに用いたシステムについて述べる．

3.1　双方向変換プログラミングの概要

双方向変換は，元々はデータベースの分野で用いられていた概念である．ソースデータ（ソース）と，ソースから必要な情報を抜き出したターゲットデータ（ビュー）があった時，ビューに対して行った変更を，ソースに対して正しく反映させるような枠組みである．この枠組みをプログラミングによって実現したものが双方向変換プログラミングである．

ソースから情報を抜き出し，ビューを作る変換を get，ビューの情報をソースへ埋め込む変換を put と呼ぶ．ソースを s，ビューを v としたとき，これらの変換はそれぞれ，$get\ s = v$，$put\ s\ v = s'$ という関数で表すことができる．ここで，s' は変更後のソースである．この get と put は，以下に示す二つの性質を満たす必要がある．

$$put\ s\ (get\ s) = s \tag{GETPUT}$$

$$get\ (put\ s\ v) = v \tag{PUTGET}$$

GETPUT は，ビューを変更せずにソースへ反映すると，元と同じソースが得られるというものであり，ソースに対し意図しない変更が行われないという性質である．PUTGET は，ビューの情報を埋め込んだソースからは同じビューが得られるというものであり，ビューに対する変更を全てソースに埋め込むという性質である．これらを満たす get と put の組があって初めて，正しい双方向変換を行うことができる．

双方向変換プログラミングを行う場合，get と put にあたる変換関数をそれぞれ記述することが，一番単純な実装である．しかしその場合，それらの変換関数が GETPUT と PUTGET の両方を満たす必要がある．そして，一方の変換関数に対して変更を加えた場合，GETPUT と PUTGET をともに満たすように，他方の変換関数にも変換を加える必要がある．

また，上述の手間を省くために，get を記述することで，GETPUT と PUTGET を満たす put を自動で生成するような研究がされた [6]. しかし，get に対する put は一意に定めることができない場合があるため，get をベースとした双方向変換では，ユーザが期待する put を得られない可能性がある．

これに対し，put に対する get は一意に定めることができる [7] ことから，put をベースとした双方向変換言語 BiGUL [8] が開発された．

3.2 BiGUL

BiGUL [8] は，Ko らによって開発された，Haskell を基礎とする双方向変換プログラミングを行うための言語である．BiGUL は *put* をベースとした双方向変換が行えるようになっており，ユーザが *put* を記述すると，自動的に *get* が生成される．また，BiGUL における *put* と *get* が前述の GETPUT と PUTGET を満たしていることは，証明支援系 Agda [9] によって証明されている．

BiGUL における双方向変換の型は $BiGUL\ s\ v$ と表される．ここで，s はソースの型，v はビューの型である．また，*get* の型は $BiGUL\ s\ v \to s \to Maybe\ v$，*put* の型は $BiGUL\ s\ v \to s \to v \to Maybe\ s$ と表される．$Maybe$ 型とは，値があるかどうかわからないことを表す型である．何か値 x がある場合には `Just` x を返し，何も値がない場合には `Nothing` を返す．したがって，*get* や *put* において，成功した場合には `Just` によって値を返し，失敗した場合には `Nothing` を返す．以降では，提案システムを実装する上で利用した，BiGUL の用意している関数について説明する．

3.2.1 Skip

`Skip` は，ビューによってソースを書き換える際に，変更されたくない箇所に使用する．`Skip` の型は $(s \to v) \to BiGUL\ s\ v$ であり，$s \to v$ 型の任意の関数を一つ引数にとる．`Skip` は，引数の関数によってビューが決定される場合には値を返し，決定できない場合には `Nothing` を返す．以下の例を考える．

```
1 put (Skip add1) 1 2
2 > Just 1
```

ここで，`add1` は引数に整数を一つとり，1 加算した値を返す関数とする．この例では，ソースが `1`，ビューが `2` であり，ビューが `add1 1` によって決定されるため値を返している．しかし，次の例を考える．

```
1 put (Skip add1) 1 10
2 > Nothing
```

この例では，ビューの値が `add1 1` によって決定できないため値が返されていない．したがって，`Skip` を使うことで，ビューの変更を禁止することができる．

また，*get* を行った場合には，次のような結果を返す．

```
1 get (Skip add1) 1
2 > Just 2
```

したがって，`Skip` f に対して *get* を行うと，単にソースに対して f を適用した結果が返される．

3.2.2 Replace

Replace は，ビューをそのまま使ってソースを書き換える際に使用する．Replace の型は，$BiGUL\ s\ s$ となっている．以下に例を示す．

```
put Replace 1 2
> Just 2
```

Replace は，例外なく値を変更し，その結果を返すため，Nothing を返すことはない．また，Replace に対して get を行うと次のようになる．

```
get Replace 1
> Just 1
```

このように，Replace に対して get を行うと，ソースの値がそのまま返される．

3.2.3 Prod

Prod は，ソースの組 (s_1, s_2)，及びビューの組 (v_1, v_2) があったとき，v_1 の値を s_1 に，v_2 の値を s_2 に put するというような，複数の値を同時に扱う場合に用いる．Prod の型は，$BiGUL\ s_1\ v_1 \rightarrow BiGUL\ s_2\ v_2 \rightarrow BiGUL\ (s_1, s_2)\ (v_1, v_2)$ となっており，双方向変換を行う 2 つの関数を引数にとる．以下に例を示す．

```
put (Prod (Skip add1) Replace) (1, 2) (2, 4)
> Just (1, 4)
```

この例では，1 は Skip add1 によって変更を行わず，2 は Replace によって 4 で置き換えるというものである．なお，Prod で繋がれた双方向変換の中で，変換に失敗し Nothing を返すものが含まれていた場合は，Prod で繋がれた双方向変換全体の結果として Nothing が返る．

```
put (Prod (Skip add1) Replace) (1, 2) (10, 4)
> Nothing
```

Prod で繋がれた式に対して get を行うと，繋がれている双方向変換それぞれに対して get を行った結果が返る．

```
get (Prod (Skip add1) Replace) (1, 2)
> Just (2, 2)
```

また，Prod には Template-Haskell [10] のクォート式を利用した構文糖衣が用意されており，その定義は以下の通りである．

```
$(update [p| ソースのパターン |] [p| ビューのパターン |] [d| 変換ルール |])
```

この構文糖衣を用いて，先の例を再度記述すると，以下のようになる．

```
$(update [p| (x,y) |] [p| (x,y) |]
         [d| x=(Skip add1); y=Replace) |])
```

これは，(x,y) という形式のソースとビューに対して，x には (Skip add1) を行い，y には Replace を行うという命令となっている．

ここで，((1, 2), ((3, 4), 5)) というソースと，((10, 2), ((30, 4), 50) というビューがあるとする．ソースの 1，3，5 は，ビューの 10，30，50 で置き換え，2，4 は変更しないような双方向変換を構文糖衣を用いずに記述すると次のようになる．

```
(Prod (Prod Replace (Skip (const 2)))
   (Prod (Prod Replace (Skip (const 4))) Replace))
```

また，同様の関数を構文糖衣を用いて記述すると，以下のようになる．

```
$(update [p| ((v, w), ((x, y), z)) |] [p| ((v, w), ((x, y), z)) |]
         [d| v=Replace; w=(Skip (const 2)); x=Replace;
             y=(Skip (const 4)); z=Replace |])
```

構文糖衣を用いない場合では，上のようにソースやビューの形式が複雑になると括弧の数が多くなり，各双方向変換を Prod で繋ぐ位置が分かりにくくなっている．一方で構文糖衣を用いると，各要素にどのような双方向変換を適用するのかを，より直感的に記述することができている．

3.2.4 rearrS/rearrV

rearrS 及び rearrV は，ソースとビューの形式が異なる場合に用いられる．例えば，((1, 2), 3) というソースと (10, 20) というビューがあり，2 に対して 10 を，3 に対して 20 を put して，ソースを ((1, 10) 20) にしたいとする．この場合，ソースの形式を (2, 3) に変形するか，ビューを (((), 10), 20) のように変形するなどして，それぞれの形式をそろえる必要がある．

まず，ソースの形式を変形する場合を考える．ソースの形式を変形する場合には rearrS を用いる．rearrS は，$(rearrS [| $s_1 \to s_2$ 型のラムダ抽象 |]) の形式で記述され，[| と |] の間に記述されたラムダ抽象にしたがってソースを変形する．また，その型は $BiGUL\ s_2\ v \to BiGUL\ s_1\ v$ である．これを用いた例を以下に示す．

```
put ($(rearrS [| \((x0, x1), x2) -> (x1, x2) |]) Replace)
      ((1, 2), 3) (10, 20)
> Just ((1, 10), 20)
```

これは，((s0, s1), s2) という形式のソースを受け取り，一つ目の要素 s0 を除外し，ソースを (s1, s2) という組になるように，rearrS によってソースを変形している．

次に，ビューの形式を変形する場合を考える．ビューの形式を変形する場合には，rearrV を用いる．rearrV は，$(rearrV [| $v_1 \to v_2$ 型のラムダ抽象 |]) の形式で記述され，rearrS

同様に，記述されたラムダ抽象に従ってビューの形式を変形する．この型は $BiGUL\ s\ v_2 \rightarrow BiGUL\ s\ v_1$ であり，これを用いて先と同様の変換を行う例を以下に示す．

```
put ($(rearrV [| \(y1, y2) -> ((1, y1), y2) |])$
      ((Skip (const 1)) `Prod` Replace) `Prod` Replace)
      ((1, 2), 3) (10, 20)
> Just ((1, 10), 20)
```

これは，(v1, v2) という形式のビューを受け取り，先頭に 1 という要素を追加し，ソースと形式を揃える．ここで，ビューの先頭の要素は Skip (const 1) によって，ソースの先頭の要素から決定される要素である．したがって，Skip によってソースの最初の要素が変更されず，残りの要素が順に Replace によって変更され，rearrS を用いた場合と同じ結果になる．

この rearrS/rearrV を用いると，リスト構造 (x:xs) に対し，

```
$(rearrS [| \(x:xs) -> (x, xs) |])
```

とすることで，リストの先頭要素と，それ以外の要素に別々の処理を行うことも可能である．

3.2.5 Case

Case は，双方向変換における条件分岐を記述する際に用いる．Case の構文は以下のようになっている．

```
Case [
  $(normal [| 進入時条件1 |] [| 終了時条件1 |])
    ==> 双方向変換1,
  $(adaptive [| 進入時条件 |])
    ==> ソースを変換する関数1,
  ...,
]
```

進入時条件とは，その分岐に入る際の条件であり，型は $s \rightarrow v \rightarrow Bool$ となっている．ここには，ソースとビューに関する条件を記述する．終了時条件とは，その分岐における処理が終了したときに満たすべき条件であり，型は $s \rightarrow Bool$ となっている．ここには，双方向変換によって，変換されたソースの満たすべき状態を記述する．

Case では，上から順に分岐の進入時条件が確認される．normal というキーワードで記述された分岐へ入る場合，ソースとビューに対し，記述された双方向変換が適用される．双方向変換の後に，終了時条件を満たしていない場合は，その双方向変換を取り消し，他の分岐の進入時条件が改めて確認される．また，終了時条件は，同時に複数の分岐で真になってはならない．

adaptive というキーワードで記述された分岐へ入る場合，ソースに対して，記述された変換関数が適用され，改めて上から順に分岐の条件を確認する．ここで，ソースを変換する関数の型は $s \rightarrow v \rightarrow s$ となっており，この変換では，もう一度分岐を初めから確認した際，次に normal の分岐へ入れるように，ソースを変換しなければいけない．したがって，連続して adaptive の

分岐へ入る場合は失敗となり，Nothing を返す．

以下に，Case を用いた双方向変換の例を示す．

```
1  allReplace :: BiGUL [Int] [Int]
2  allReplace =
3    Case [
4      $(normal [| \s v -> null s && null v |] [| \s -> null s |])
5        ==> $(update [p| [] |] [p| [] |] [d| |]),
6      $(adaptive [| \s v -> length s > 0 && null v |])
7        ==> \_ _ -> [],
8      $(normal [| \s v -> length s > 0 && length v > 0 |]
9               [| \s -> length s > 0 |])
10       ==> $(update [p| x:xs |] [p| x:xs |]
11                   [d| x=Replace; xs=allReplace |]),
12     $(adaptive [| \s v -> null s && length v > 0 |])
13       ==> \_ _ -> [undefined]
14   ]
```

まず，4 行目の分岐の進入時条件は，ソースとビューが共に空リストの場合である．この場合は何も変換を行わないため，normal の分岐となっており，ソースに変更を加えないため，ソースが空リストであることを終了時条件としている．次に，6 行目の分岐への進入時条件は，ソースは空リストではないが，ビューが空リストの場合である．この場合，埋め込む情報が存在しないため，adaptive の分岐となっており，ソースの残りの情報を捨て，空リストに変換している．8 行目の分岐の進入条件は，ソースとビューが共に空リストではない場合である．この場合，双方向変換が行われるため normal の分岐となっている．ここではリストの先頭要素をビューの値で置き換え，残りの要素はこの関数を再帰している．最後に 12 行目は，ソースが空リストで，ビューが空リストでない場合である．この場合，埋め込むべきビューの情報はあるが，埋め込む対象が存在しないため，adaptive の分岐となっている．何か一つ要素を持つリストへソースを変換することにより，ビューの情報を埋め込む対象を生成している．

この関数を用いた結果を以下に示す．

```
1  put allReplace [1,2,3] [4,5,6,7]
2  > Just [4,5,6,7]
3
4  put allReplace [1,2,3] []
5  > Just []
6
7  put allReplace [] [4,5,6,7]
8  > Just [4,5,6,7]
```

3.2.6 emb

基本的に BiGUL では，*put* を記述することで *get* を自動的に得ることができる．しかし emb は，*get* と *put* を両方とも自分で記述したい場合に利用する．ただし，GETPUT 及び PUTGET

を満たさないような *get* と *put* を記述した場合は，必ず Nothing を返す．以下に例を示す．

```
dif :: BiGUL (Int, Int) Int
dif = emb g p
  where g (s1,s2) =
          if s1 < s2
          then s2 - s1
          else s1 - s2
        p (s1,s2) v =
          (v + s2, s2)
```

dif に対し，*get* をすると，ソースの二つの整数値の差を求め，*put* をすると，差がビューの値となるようにソースに情報を埋め込む関数である．この関数の実行結果は以下の通りとなる．

```
get dif (30, 40)
> Just 10

put dif (30, 40) 50
> Just (90,40)

get dif (90, 40)
> Just 50
```

この様に，emb を用いることで，より直感的に双方向の変換を記述することができる．

3.3 BiYacc

BiYacc [11] は，Zhu らによって開発された，パーサとプリンタを同時に開発することができる領域特化言語である．BiYacc は，ソースコードをソース，構文木をビューとして双方向変換を行っており，パーサが *get*，プリンタが *put* に対応している．また，BiYacc も *put* ベースの双方向変換であり，プリンタの定義を記述することで，パーサを自動的に得ることができる．特徴として，ソースコードへ構文木のデータを *put* する際，できるだけ元のソースコードの書式を残すため，インデントやコメントが保持されるという点が挙げられる．

BiYacc では，構文木の定義，ソースコードの構文の定義，及び構文木からソースコードを更新するルールの定義という，3 つの定義を記述する．以降では，算術式に対する，BiYacc プログラムを例として挙げる．ここで扱う算術式は，整数に対する四則演算に加え，剰余演算，べき演算及び括弧を扱うことができるものとする．

3.3.1 構文木の定義

BiYacc における構文木の定義は，関数型プログラミング言語における代数的データ型と同じ方法で記述する．扱う算術式の構文木を以下に示す．

```
Abstract

```

```
 3 data Arith = Add Arith Arith
 4              | Sub Arith Arith
 5              | Mul Arith Arith
 6              | Div Arith Arith
 7              | Mod Arith Arith
 8              | Pow Arith Arith
 9              | Num Int
10              deriving (Show, Eq, Read)
```

初めの Abstract は，以降に構文木の定義を記述することを示すキーワードである．Add は加算，Sub は減算，Mul は乗算，Div は除算，Mod は剰余，Pow はべき乗，Num は数値を表す．Num 以外は二項演算であるため，各コンストラクタが二つの式 Arith を持っている．Num は整数値であるため，Int 型の値を一つ持つ．deriving (Show, Eq, Read) は，BiYacc が処理を行う際に必要となるものであり，構文木の定義をする際には必ず記述する．

3.3.2 ソースコードの構文の定義

ソースコードの定義は，文脈自由文法によって記述する．扱う算術式の定義を以下に示す．

```
 1 Concrete
 2
 3 Arith1 -> Arith1 '+' Arith2
 4         | Arith1 '-' Arith2
 5         | Arith2
 6         ;
 7
 8 Arith2 -> Arith2 '*' Arith3
 9         | Arith2 '/' Arith3
10         | Arith2 '%' Arith3
11         | Arith3
12         ;
13
14 Arith3 -> Arith3 '^' Arith4
15         | Arith4
16         ;
17
18 Arith4 -> Int
19         | '(' Arith1 ')'
20         ;
```

Concrete は，以降にソースコードの構文の定義を記述することを示すキーワードである．BiYacc では，+ のような終端記号は，' ' で括って記述し，Arith1 のような非終端記号は，一文字目は必ず大文字で記述されなければならない．この例では，加算と減算，乗算と除算，べき乗をそれぞれ別の非終端記号で定義することで，結合の優先順位に対応している．

3.3.3 ソースコードを更新するルールの定義

ソースコードを更新するルールは，BiYacc 特有の文法で記述する．扱う算術式における定義を以下に示す．

```
 1 Actions
 2
 3 Arith +> Arith1
 4 Add x y +> (x +> Arith1) '+' (y +> Arith2);
 5 Sub x y +> (x +> Arith1) '-' (y +> Arith2);
 6 t       +> (t +> Arith2);
 7
 8 Arith +> Arith2
 9 Mul x y +> (x +> Arith2) '*' (y +> Arith3);
10 Div x y +> (x +> Arith2) '/' (y +> Arith3);
11 Mod x y +> (x +> Arith2) '%' (y +> Arith3);
12 t       +> (t +> Arith3);
13
14 Arith +> Arith3
15 Pow x y +> (x +> Arith3) '^' (y +> Arith4);
16 t       +> (t +> Arith4);
17
18 Arith +> Arith4
19 Num i   +> (i +> Int);
20 t        +> '(' (t +> Arith1) ')';
```

Actions は，以降にソースコードを更新するルールを記述することを示すキーワードである．3 行目は，構文木 Arith で，非終端記号 Arith1 に書かれた構文を更新するルールを記述することを表す．4 行目，5 行目は，+>の左辺の構文木で，右辺の算術式を更新することを表す．なお 4 行目では，コンストラクタ Add の二つの引数を x と y に束縛し，x で'+' の左の式を，y で右の式を更新することを記述しており，x と y はそれぞれ，再帰的にこの更新するルールをたどっていく．また，各ルールは，文末にセミコロンをつける必要がある．

3.3.4 実行例

BiYacc における，ソースコードの更新の流れは以下の通りである．

1. 構文木とソースコードを入力として与える．
2. 入力に対応した Actions のグループが選択される．
3. グループ内の更新のルールについて，構文木とソースコードを同時にパターンマッチする．
4. パターンマッチが成功した場合には，指定された更新を行う．
5. グループ内の全てのパターンマッチが失敗した場合，構文木がマッチするルールを選択し，更新する．

ここで，ここまでに例に挙げた定義をもとに，BiYacc によって作られるプログラムの実行例を示す．1 + 2 ^ 3 * 4 という式に対し，次の構文木の情報を埋め込むことを考える．

```
1 Add (Num 2)
2     (Div (Num 3)
3          (Num 4))
```

まず，構文木と算術式の形式から，3.3.3 節で定義した Actions の 3 行目，Arith +> Arith1 のグループが選択される．そして，構文木と算術式を同時にパターンマッチを行うと，4 行目のルールがマッチする．4 行目では，Add の一つ目の式で'+' の左辺を，Add の二つ目の式で '+' の右辺を更新するという定義があるため，(Num 2) で 1 を，(Div (Num 3) (Num 4)) で 2 ^ 3 * 4 をそれぞれ更新していく．前者は，6 行目，12 行目，16 行目を経て，Arith +> Arith4 グループへ移動し，19 行目にマッチするため，1 を (Num 2) の 2 で更新する．後者は，まず Arith +> Arith2 グループでパターンマッチを行う．ここでは全てのパターンに失敗するため，構文木 (Div (Num 3) (Num 4)) のみを参照し，改めてパターンマッチを行う．すると，10 行目にマッチするため，2 ^ 3 * 4 という情報を破棄し，新しく算術式を生成する．10 行目では，4 行目同様に，Div の二つの引数について再帰的にルールを適用する．すると，それぞれ 19 行目にマッチし，終了する．結果として，2 + 3 / 4 という式を得る．

4　設計

本章では，提案するシステムに求められる要件をまとめた後，全体像とその動作例，及び本システムが対応している OCaml と Coq のサブセット，本システムを利用する際の制限について説明する．

4.1　要件定義

提案システムの設計における要件を以下にまとめる．

要件 1　プログラムの仕様を OCaml 側で決定する．
要件 2　Coq 上でのプログラムの変更を，OCaml プログラムへ反映させる．
要件 3　OCaml プログラムを変更しても，以前との差分が小さい Coq プログラムを生成する．
要件 4　言語間の情報の差異を，できる限り吸収する．

要件 1 は，OCaml プログラムの開発に必要不可欠な要件である．Coq 上でプログラムの証明は行うが，最終的に使用するのは OCaml プログラムである．特に，OCaml プログラムで用いる一部の関数を Coq で証明する場合，OCaml 側で仕様が決定できないと，Coq で証明するまで証明する関数を扱う箇所の仕様が決定できない．そのため，Coq 側ではなく，OCaml 側での仕様決定が求められる．

要件 2 は，Coq 上で円滑な証明を行うために必要な要件である．OCaml 側でプログラムの仕様は決定するが，その際，証明で扱いにくい，もしくは証明できない形式となっていることがある．その際に，OCaml プログラムを変更し，再度 Coq プログラムへ変換を行っても，証明の行いやすい形式に変換されるかどうかはわからない．そのため，Coq 上で直接変更を加え，その変更を OCaml プログラムへ反映させることは，証明の支援に繋がる．

要件 3 は，既存の証明を再利用する際に求められる要件である．証明を行った OCaml プログラムに対し，仕様変更などで変更を加えた場合には，改めて証明を行う必要がある．その際，以前の Coq プログラムとの差分が小さい Coq プログラムを生成することで，既存の証明が再利用でき，証明の手間が軽減されることが期待される．

要件 4 は，Coq 及び OCaml でプログラムを記述する際に，その記述力を高めるために必要な要件である．OCaml プログラムと Coq プログラムでは，一方にしか存在しない情報というものがある．例えば，OCaml ではアンカリー化した引数を定義する際に，仮引数の段階で引数の組の要素それぞれに変数を束縛することができるが，Coq では変数を束縛することができない．また Coq では，同じ型の仮引数はまとめて型注釈を行うことができるが，OCaml では，変数一つずつにしか型注釈ができない，といったことが挙げられる．言語間で共通の情報のみを扱うので

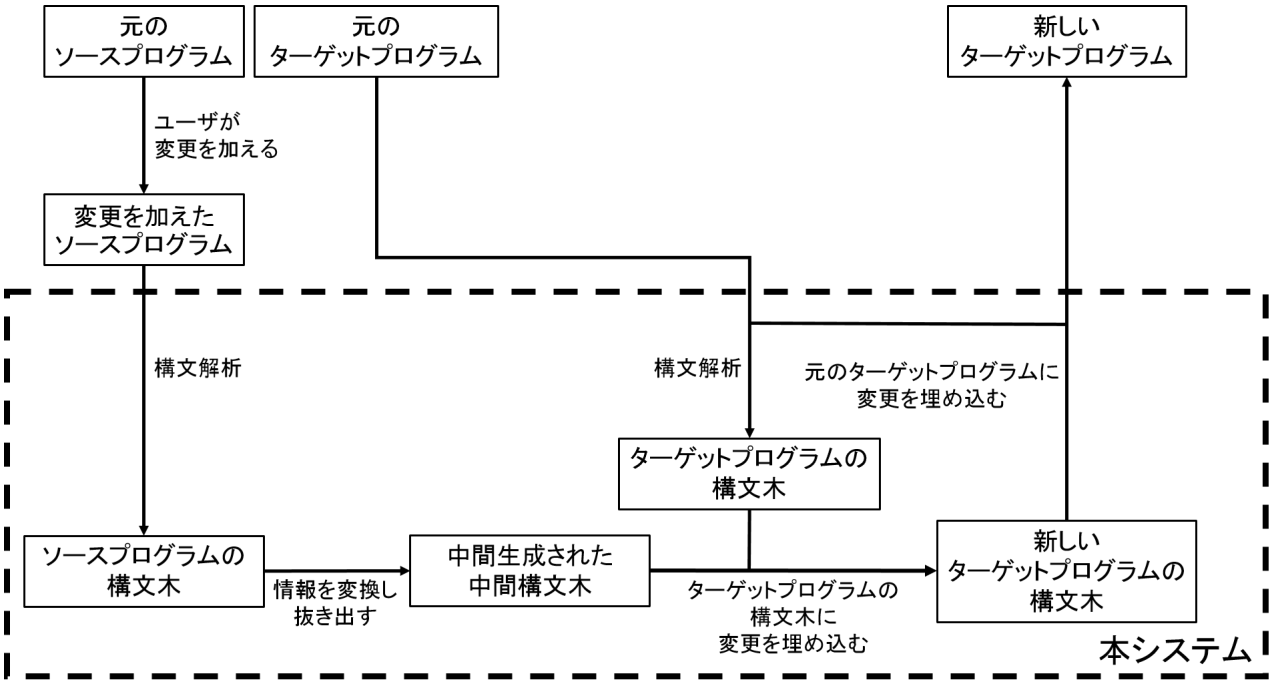

図 4.1　提案システムの全体像

はなく，このような言語間の差異を吸収し等価なプログラムへと変換できるようにすることで，本システムにおいて，より多様な記述が可能となる．

4.2　提案システムの全体像

提案システムの全体像を図 4.1 に示す．本システムは，前提として Coq のプログラムと OCaml のプログラムがあり，OCaml がソースプログラムの場合は Coq がターゲットプログラムとなり，Coq がソースプログラムの場合は OCaml がターゲットプログラムとなる．まず，ユーザは変更を加えたソースプログラムと，その変更を埋め込みたいターゲットプログラムを本システムに入力として与える．すると，本システムはソースプログラムを構文解析し，ソースプログラムの構文木を生成する．次に，生成された構文木から，必要な情報を変換しつつ抜き出し，OCaml と Coq の中間となるような構文木を生成する．この構文木は，OCaml と Coq どちらがソースプログラムになった場合でも同じデータ構造となるようにしており，中間構文木へ変換する際に言語間の差異を吸収する．そして，ソースプログラム同様にターゲットプログラムも構文解析してターゲットプログラムの構文木を生成し，その構文木に抜き出した構文木から，変更すべき情報を埋め込む．最後に，変更する情報が埋め込まれた新しいターゲットプログラムの構文木から，元のターゲットプログラムへ変更を埋め込むことで，ソースプログラムへ加えられた変更を反映した，新しいターゲットプログラムが生成される．なお，ソースコードと構文木間の変換には BiYacc を，構文木間の変換には BiGUL を利用している．

ターゲットプログラムを 1 から生成するのではなく，必要な情報を抜き出し，BiYacc によっ

て元のターゲットプログラムへ埋め込む方式をとることで，コメントやインデントなどの情報をできる限り維持した，差分の少ないターゲットプログラムを生成することができる．

また，ソースプログラムのみを本システムに入力として与えた場合には，1 からソースプログラムに対応するターゲットプログラムを生成する．

ここで，中間構文木は OCaml の構文木と Coq の構文木の共通要素のみとしている．その理由は，以下の通りである．本システムでは，ソースプログラムの構文木を *get* することによって必要な情報だけを抜き出し，中間構文木を生成している．そして，中間構文木の持つ情報を，ターゲットプログラムの構文木へ *put* することにより埋め込んでいる．この方法で生成される中間構文木では，あらかじめ OCaml と Coq の差を吸収した構文木を生成するため，共通要素のみで十分である．また，中間構文木を OCaml の構文木と Coq の構文木，それぞれの要素を全て網羅するような設計にすることも考えられる．その場合，ソースプログラムの構文木から *put* することにより中間構文木へ情報を埋め込み，それを *get* することで，変更された情報を持つ，新しいターゲットプログラムの構文木を生成することとなる．しかしこの方法では，元のターゲットプログラムの構文木へ情報を埋め込むのではなく，*get* によって新しく生成している．そのため，元の情報に変更を埋め込むという本システムの目的に背いてしまう．したがって，本システムでは，前者の方法を採用した．

4.3　本システムの典型的な利用シナリオ

本システムの典型的な利用シナリオを図 4.2 に示す．

ユーザは初めに OCaml プログラム p を記述し，(1) 本システムを用いて Coq プログラム q へ変換して証明を行う．その際，(2) 証明を行いやすくするために，Coq プログラム q に変更を加

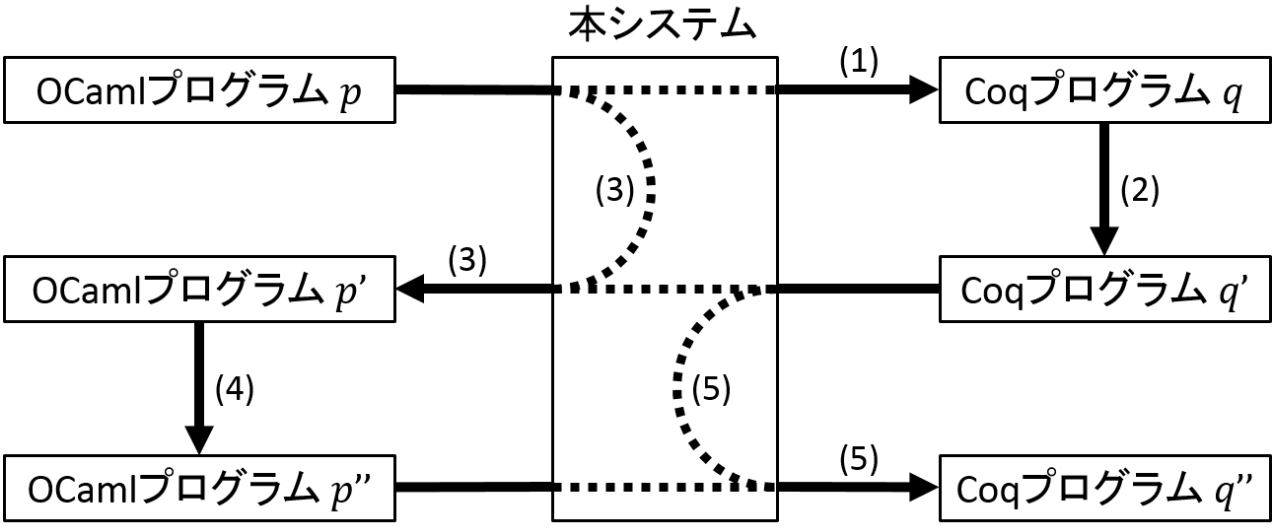

図 4.2　本システムの典型的な利用シナリオ

えて Coq プログラム q' となったとする．ユーザは Coq プログラム q' を証明した後，(3) Coq プログラム q へ加えた変更を本システムを用いて OCaml プログラム p へ反映し，OCaml プログラム p' を得る．その後 (4) 仕様変更により，OCaml プログラム p' に変更を加えて OCaml プログラム p" としたとする．そして (5) OCaml プログラム p' へ加えた変更を Coq プログラム q' へ反映し，Coq プログラム q" とする．最後に，Coq プログラム q" を証明することで，OCaml プログラム p" に対する任意の性質を示せる．以降はこの繰り返しとなる．

以上のように，本システムでは OCaml プログラムと Coq プログラムそれぞれに対し，繰り返し部分的な変更を加える．そしてそれを他方へ反映させることで，任意の性質を保証したプログラムを開発するといった利用方法を想定している．

4.4 本システムの利用例

まず，以下のような OCaml プログラムをユーザが記述したとする．

```
1 type 'a binary_tree =
2 | Leaf of 'a
3 | Tree of 'a binary_tree * 'a binary_tree
4
5 let rec size (bt : int binary_tree) : int =
6   match bt with
7   | Leaf i -> i + 1
8   | Tree (bt1, bt2) -> size bt1 + size bt2
```

これは，二分木を表すデータ型の `binary_tree` と，整数を要素に持つ二分木の各要素に 1 を加えたものの和を求める関数 `size` である．この OCaml プログラムを本システムに与えると，以下の Coq プログラムが得られる．

```
1 Inductive binary_tree ( A : Type ) : Type :=
2 | Leaf : A -> binary_tree A
3 | Tree : binary_tree A * binary_tree A -> binary_tree A.
4
5 Fixpoint size ( bt : binary_tree nat ) : nat :=
6   match bt with
7   | Leaf i => i + 1
8   | Tree (bt1, bt2) => size bt1 + size bt2
9 end.
```

ここでは，二分木の要素には正の整数のみが与えれられると仮定する．そして「各要素の和が 0 より大きい」という性質について証明を行い，以下のような Coq プログラムになったとする．

```
1 Require Import Omega.
2 Set Implicit Arguments.
3
4 Inductive binary_tree ( A : Type ) : Type :=
5 | Leaf : A -> binary_tree A
6 | Tree : binary_tree A * binary_tree A -> binary_tree A.
```

```
 7
 8 Fixpoint size (bt : binary_tree nat ) : nat :=
 9   match bt with
10   | Leaf i => i + 1
11   | Tree (bt1, bt2) => size bt1 + size bt2
12 end.
13
14 Functional Scheme size_ind := Induction for size Sort Prop.
15
16 Theorem size_gt_zero : forall bt, 0 < size bt.
17 Proof.
18   apply (size_ind (fun bt s => 0 < s)); intros; omega.
19 Qed.
```

この証明が書かれた Coq プログラムを，本システムを用いて OCaml プログラムへ再変換すると以下のプログラムとなる．

```
1 type 'a binary_tree =
2 | Leaf of 'a
3 | Tree of 'a binary_tree * 'a binary_tree
4
5 let rec size (bt : int binary_tree) : int =
6   match bt with
7   | Leaf i -> i + 1
8   | Tree (bt1, bt2) -> size bt1 + size bt2
```

今はプログラム部分を変更していないため，OCaml プログラムへ再変換しても元と同じ OCaml プログラムとなる．その後，元は要素に 1 加えたものの総和をサイズとしていたが，要素数をサイズとするような仕様変更があり，OCaml プログラムに変更を加えたとする．

```
1 type 'a binary_tree =
2 | Leaf of 'a
3 | Tree of 'a binary_tree * 'a binary_tree
4
5 let rec size (bt : int binary_tree) : int =
6   match bt with
7   | Leaf i -> 1
8   | Tree (bt1, bt2) -> size bt1 + size bt2
```

変更後の `size` は，二分木の要素数を数えるプログラムである．OCaml プログラムに変更を加えたため，改めて Coq でこの OCaml プログラムが満たすべき性質について証明をするため，本システムを用いて OCaml プログラムへ行った変更を元の Coq プログラムへ反映させると以下のような Coq プログラムが得られる．

```
1 Require Import Omega.
2 Set Implicit Arguments.
3
4 Inductive binary_tree ( A : Type ) : Type :=
5 | Leaf : A -> binary_tree A
```

```
 6 | Tree : binary_tree A * binary_tree A -> binary_tree A.
 7 |
 8 | Fixpoint size (bt : binary_tree nat ) : nat :=
 9 |   match bt with
10 |   | Leaf i => 1
11 |   | Tree (bt1, bt2) => size bt1 + size bt2
12 | end.
13 |
14 | Functional Scheme size_ind := Induction for size Sort Prop.
15 |
16 | Theorem size_gt_zero : forall bt, 0 < size bt.
17 | Proof.
18 | apply (size_ind (fun bt s => 0 < s)); intros; omega.
19 | Qed.
```

この Coq プログラムでは，OCaml プログラムへ行った変更が，関数名や関数の内部に反映されており，それ以外の部分や証明は元の情報が保持されている．この部分的な変更を施した Coq プログラムに対し，任意の証明を行う．一般的には証明の変更も必要だが，この場合については制約検査器の omega が汎用的であるため，証明の変更の必要がない．

以上のプロセスを繰り返すことで，差分の小さな Coq プログラムを用いて証明を行うことができ，部分的な変更を施した OCaml プログラムの任意の性質を保証することができる．

4.5 対応しているサブセット

本システムが対応している OCaml の構文と型を図 4.3 に，Coq の構文と型を図 4.4 示す．

OCaml 及び Coq の二項演算では，論理和，論理積，算術演算及び比較演算を扱うことができる．

OCaml の再帰関数定義及び局所再帰関数定義では，`let rec f ... and g ...` のように定義を `and` で繋ぐことで，相互再帰関数を定義することができる．Coq では `and` の代わりに `with` を用いることで定義できる．

OCaml の `function` $(e$ `->` $e)$`+` は，ラムダ抽象と条件分岐を組み合わせたものであり，引数を一つ受け取り，その値について直ちに条件分岐を行う構文である．また，`begin` e `end` は，式の繋がりや切れ目を明示する際に記述するものであり，(e) と同義である．

Coq には型の型が存在しており，`Type` 型と `Set` 型がそれに当たる．

Coq の変数定義及び再帰関数定義における仮引数の定義では，複数の引数に対してまとめて型注釈が行える．

OCaml では，アンカリー化された引数を受け取る関数を定義する場合，仮引数の段階でパターンマッチにより組の要素に名前を付けることができる．

また，本システムにおいて記述するプログラムにはいくつかの制限がある．OCaml と Coq で共通していることとして，負の数を記述する際には，`(-1)` のように，括弧をつけて記述しなけ

declaration	::=	`let` $id\ arg$ `=` e	変数定義
	|	`let rec` $id\ arg$ `=` e	再帰関数定義
	|	`type` $arg\ id$ `=` $(id$ `of` $t)$`+`	型定義
expression e	::=	`()`	ユニット
	|	c	定数
	|	x	変数参照
	|	`[]`	空リスト
	|	$e\ op\ e$	二項演算
	|	`(` e_1 ... e_n `)`	組
	|	`if` e `then` e `else` e	条件分岐
	|	`match` e `with` $(e$ `->` $e)$`+`	条件分岐
	|	`let` $id\ arg$ `=` e `in` e	局所変数定義
	|	`let rec` $id\ arg$ `=` e `in` e	局所再帰関数定義
	|	`let (` x_1 ... x_n `)` `=` e `in` e	組の読み出し
	|	`fun` arg `->` e	ラムダ抽象
	|	`function` $(e$ `->` $e)$`+`	ラムダ抽象及び条件分岐
	|	$e\ e_1$ ... e_n	関数適用
	|	`begin` e `end`	式の結合
type t	::=	`unit`	ユニット型
	|	`int`	整数型
	|	`bool`	真偽値型
	|	`string`	文字列型
	|	$t\ *\ t$	組型
	|	$t\ \rightarrow\ t$	関数型
	|	*user type*	ユーザ定義型

図 4.3　対応している OCaml の構文と型

ればいけない．さらに，Coq 上で新たに型を定義する場合，コンストラクタ名を大文字で始めなければならない．

declaration	::=	`Definition` $id\ arg$ `:=` e	変数定義
	|	`Fixpoint` $id\ arg$ `:=` e	再帰関数定義
	|	`Function` $id\ arg$ `:=` e	再帰関数定義
	|	`Inductive` $id\ arg$ `:=` $(id$ `:` $t)$`+`	型定義
expression e	::=	`tt`	ユニット
	|	c	定数
	|	x	変数参照
	|	$e\ op\ e$	二項演算
	|	`(` e_1 ... e_n `)`	組
	|	`if` e `then` e `else` e	条件分岐
	|	`match` e `with` $(e$ `=>` $e)$`+` `end`	条件分岐
	|	`let` $id\ arg$ `:=` e `in` e	局所変数定義
	|	`fix` $id\ arg$ `:=` e	局所再帰関数定義
	|	`let ‘(` x_1 ... x_n `)` `:=` e `in` e	組の読み出し
	|	`fun` arg `=>` e	ラムダ抽象
	|	$e\ e_1$ ... e_n	関数適用
type t	::=	`unit`	ユニット型
	|	`nat`	自然数型
	|	`Z`	整数型
	|	`bool`	真偽値型
	|	`string`	文字列型
	|	`Type`	`Type` 型
	|	`Set`	`Set` 型
	|	$t\ *\ t$	組型
	|	$t\ \rightarrow\ t$	関数型
	|	*user type*	ユーザ定義型

図 4.4　対応している Coq の構文と型

5 実装

本システムでは，OCaml のソースコードと OCaml の構文木間，OCaml の構文木と中間構文木間，中間構文木と Coq の構文木間，Coq の構文木と Coq のソースコード間の四つの双方向変換を行っている．ソースコードと構文木間の変換では BiYacc を利用し，構文木同士の変換では BiGUL を利用している．ここで，中間構文木は図 5.1 に示す構文に対応した構文木である．中間構文木の対応する構文と OCaml の構文の差異は，空リストや `function` 文，式の結合が存在せず，アンカリー化された引数を受け取る関数を定義する際に，仮引数の段階でパターンマッチにより組の要素に名前を付けることができないという点が挙げられる．また，中間構文木の対応

declaration	::=	`let` $id\ arg$ `=` e	変数定義
	|	`let rec` $id\ arg$ `=` e	再帰関数定義
	|	`type` $arg\ id$ `=` $(id$ `of` $t)$`+`	型定義
expression e	::=	`()`	ユニット
	|	c	定数
	|	x	変数参照
	|	$e\ op\ e$	二項演算
	|	`(` e_1 ... e_n `)`	組
	|	`if` e `then` e `else` e	条件分岐
	|	`match` e `with` $(e$ `->` $e)$`+`	条件分岐
	|	`let` $id\ arg$ `=` e `in` e	局所変数定義
	|	`let rec` $id\ arg$ `=` e `in` e	局所再帰関数定義
	|	`let (` x_1 ... x_n `)` `=` e `in` e	組の読み出し
	|	`fun` arg `->` e	ラムダ抽象
	|	$e\ e_1$ ... e_n	関数適用
type t	::=	`unit`	ユニット型
	|	`int`	整数型
	|	`bool`	真偽値型
	|	`string`	文字列型
	|	$t\ *\ t$	組型
	|	$t\ \rightarrow\ t$	関数型
	|	$user\ type$	ユーザ定義型

図 5.1　中間構文木に対応した構文と型

する構文と Coq の構文の差異は，自然数型や Type 型，Set 型が存在せず，局所再帰関数定義において後続の式を持つ，引数にまとめて型注釈をすることができないという点が挙げられる．

以降では，Coq と OCaml の構文の差異を埋める変換について述べるが，これらの差異は中間構文木に変換する際に吸収される．

5.1 OCaml のリストに対する構文解析

OCaml でのリストの表現方法には，[] や::のような特殊な記号を用いる表現と，[1; 2; 3] のように四角括弧を用いる表現の 2 通りがある．さらに，Coq とは異なり，コンストラクタを用いた形式で記述されることはない．そこで，[] と::については，それぞれ特殊文字として，構文解析の段階で Nil 及び Cons というコンストラクタに変換することにした．したがって，h :: t というリスト構造は Cons (h, t) と同義とみなして構文解析を行う．また，[1; 2; 3] という表現は，OCaml における一つの構文として扱い，構文解析の結果は，前者と同じように Cons と Nil の組み合わせで表現されるようにした．

5.2 型の変換

本システムでは，Coq にしか存在しないいくつかの型に対応している．そのため，OCaml へ変換する際に，その情報をなくしたり，別の型として変換している．

まず，Coq では数値を表す型として，自然数を表す nat 型と，整数を表す Z 型に対応しているが，OCaml では整数を表す int 型にのみ対応している．そのため，Coq から OCaml へ変換する場合には，nat 型と Z 型はともに int 型へと変換する．OCaml から Coq へ変換する場合には，Coq 側で Z 型を用いている箇所は int 型を Z 型に変換し，それ以外の箇所では nat 型に変換する．

次に，Coq では型の型が存在しており，それを表すのが Type 型と Set 型である．これらは Coq で型を定義するときなどに用いられるが，OCaml では必要のない情報である．そのため，Coq から OCaml へ変換する場合には，これらの型の情報は全て失われる．OCaml から Coq へ変換する際，OCaml において型の情報が無く，Coq で Type 型や Set 型を用いている場合には，OCaml ではそれらの型の情報が失われていたと判断し，それらの型の情報を付加して変換する．

5.3 function 文の変換

本システムでは，OCaml 特有の構文として，function 文を採用している．これは，fun x -> match x with と同義である．ここで，function 文を用いた次のような関数を考える．

```
1 let rec fact = function
```

```
2 | 0 -> 1
3 | n -> n * fact (n - 1)
```

fact は階乗を求める関数である．この関数は 1 引数関数だが，関数の宣言では仮引数を記述しておらず，function 文によるラムダ抽象によって引数を取っている．しかし Coq では，仮引数を宣言しない再帰関数を記述することができない．したがって，function を fun x => match x with という形に単純に置き換えた次のような関数では，仮引数がないため Coq 上で定義することができないという問題が発生する．

```
1 Fixpoint fact := fun Fun_Var => match Fun_Var with
2 | 0 => 1
3 | n => n * fact (n - 1)
4 end.
```

そのため本システムでは，OCaml の再帰関数定義において，関数本体が function で始まる場合には，中間構文木に変換する際，関数に Fun_Var という仮引数を追加し，function 文ではなく match Fun_Var with によるパターンマッチとすることでこの問題を回避している．これにより，変換後の Coq のプログラムは以下のようになる．

```
1 Fixpoint fact Fun_Var := match Fun_Var with
2 | 0 => 1
3 | n => n * fact (n - 1)
4 end.
```

また，OCaml では大文字から始まる名前の引数は定義できないため，プログラム中で元々記述してあった変数の名前が，仮引数 Fun_Var と重複することは避けられる．ただし，コンストラクタとして Fun_Var が定義されており，それを関数内で使っている場合などは，名前の衝突が発生する．なお，上記以外の場所で function 文が現れた際には，単に fun Fun_Var => match Fun_Var with となるように変換している．さらに，中間構文木から OCaml の構文木へ変換する際，再帰関数定義の引数が Fun_Var であった場合には，引数から Fun_Var を削除し，関数本体を function 文に変更するようにした．

5.4 型注釈のある引数の変換

Coq では，関数の引数に型注釈をつける際，同じ型の引数はまとめて（$x\ y\ z$:$type$）と記述することができる．しかし OCaml では，同じ型であっても変数ごとに型注釈をつける必要がある．

Coq の構文木から情報を抜き出し，中間構文木へ変える際には，型の情報を複製して引数ごとにその型情報を与え，最終的に OCaml で（x:$type$）（y:$type$）（z:$type$）となるようにした．

また，中間構文木から Coq の構文木に型の情報を埋め込む際には，それぞれの変数の型に依存し結果が変わる． Coq の引数（$x\ y\ z$:$type$）に対し，中間構文木の情報（x:$type$1）（y:$type$2）（z:$type$3）を埋め込む場合を例に，説明する．本システムでは，まず

Coq 及び中間構文木の変数定義を x から順にみていき，(1) Coq において，複数の引数に対してまとめて型注釈が行われているか確認する．まとめて型注釈をしていないなら，x の型を更新し，残りの引数について (1) に戻り処理を繰り返す．まとめて型注釈をしているなら (2) 中間構文木において，x の型と，その次の y の型が同じ型であるか確認する．等しくない場合には，Coq の定義を（x:$type$）（y z:$type$）に変形し，x の型を更新し，残りの引数について (1) に戻り処理を繰り返す．x と y の型が等しい場合には，y 以降の変数について，(2) に戻って処理を繰り返す．

例えば，Coq で（x y z:nat）という定義があった時，OCaml で仕様変更があり，(x:int)（y:int）(z:string) という中間構文木が得られたとする．この構文木の引数の情報を，元の Coq の型の情報へ埋め込むと，(x y:nat)（z:string）となる．また，(x:int)（y:string）(z:string) という中間構文木が得られた場合，これを埋め込むと (x:nat)（y z:string) となる．

5.5 アンカリー化された引数の変換

OCaml では，関数の定義をする際に，以下のように引数をアンカリー化された形式で記述することができる．

```
1 let add (n, m) = n + m
```

一方 Coq では，アンカリー化された引数を受け取る関数を定義することはできるが，OCaml のように，仮引数の段階でパターンマッチにより組の要素を変数に束縛することはできない．したがって，等価な関数を定義する場合，以下のように組の読み出しをはさむ必要がある．

```
1 Definition add n_m :=
2   let '(n, m) := n_m in
3     n + m
```

はじめに，OCaml から Coq へ変換する場合について述べる．まず，アンカリー化された引数 (x, y, z) をそれぞれ '_' でつなぎ，x_y_z という一つの引数で置き換える．次に，元々の引数 (x, y, z) と新しい引数 x_y_z による組の読み出しの式を生成し，元々の関数本体の前に挿入する．これらの操作により，組の要素に名前をつけた引数がなくなり，元々必要とされていた組の要素の名前は，組の読み出しによって取得するという Coq と同じ形式に変換される．

次に，Coq から OCaml へ変換する場合について述べる．まず，情報を埋め込む OCaml の関数に，アンカリー化された引数があるか，また，Coq の関数が組の読み出しで始まるかどうかを調べる．これらを満たす場合は，Coq の関数における組の読み出しがアンカリー化された引数の組に名前をつけているのだと判断し，組の読み出しを取り除き，組の読み出し以降の式を再帰的に処理を行う．また，情報を埋め込む OCaml の関数にアンカリー化された引数がない場合には，単に組の読み出しとして変換を行う．

以下に例を示す．次のようなアンカリー化された引数を持つ OCaml プログラムを考える．

```
1 let f (a,b) = a + b
```

この関数を本システムにより，Coq プログラムへ変換すると以下のようになる．

```
1 Definition f a_b :=
2   let '(a,b) := a_b in
3     a + b
```

この関数に対し，次のような変更をしたとする．

```
1 Definition f x_y c d_e :=
2   let '(x,y) := x_y in
3     let '(d, e) := d_e in
4       x + y + c + d + e
```

本システムを用いて，この変更を元の OCaml プログラムへ埋め込むと次のようになる．

```
1 let f (x,y) c d_e =
2   let (d,e) = d_e in
3     x + y + c + d + e
```

この例では，元々の OCaml プログラムに，アンカリー化された引数が一つしかなかったため，`d_e` は組の読み出しとして変換されている．

5.6 局所再帰関数定義の変換

OCaml の局所再帰関数定義の構文 `let rec` $id\ arg\ =\ e$ `in` e と異なり，Coq における局所再帰関数定義の構文は `fix` $id\ arg\ :=\ e$ となっている．`fix` 式には後続の式が存在しないなどの違いがあるが，本システムでは，これらが等価なプログラムとなるように変換を行っている．

まず，Coq の `fix` 式には，(`fix` $id\ arg\ :=\ e$) e とすることで，定義に対して直接関数適用を行えるという特徴がある．そのため，OCaml の `let rec` 文において，Coq 同様に定義に対して直接関数適用が行えるようにするには，`let rec` $id\ arg\ =\ e$ `in` id というように，後続の式で関数呼出を行うような記述をする必要がある．

また，OCaml の局所再帰関数定義では，定義した再帰関数を後続の式で使うことができる．しかし，Coq では後続の式を持たず，再帰関数を定義するだけの構文となっている．そのため，OCaml と同じように，定義した再帰関数を後続の式で扱うためには，局所変数定義 `let` を用いた，`let` $id\ :=$ `fix` $id\ arg\ :=\ e$ `in` e といった記述をする必要がある．

本システムでは，これらの構文の差を吸収し，双方向の変換を実現している．まず，Coq のプログラムから OCaml のプログラムへ変換する場合について述べる．`fix` 式が単体で記述されている場合は，単に `let rec` $id\ arg\ =\ e$ `in` id となるように変換している．`let` 式が出てきた場合は，まず `let` 式が引数を取るか確認する．引数を取らない場合には，`let` 式の直下に `fix` 文が出現するかどうかをさらに確認する．もしも `fix` 式が出現した場合は，それは局所定義した再

帰関数を後続の式で使いたいのだと判断し，let rec 式となるように変換する．let 式が引数を取る場合や，let 式の直下に fix 式が無い場合には，それは局所変数定義もしくは局所関数定義だと判断し，let 式になるように変換する．

次に OCaml のプログラムを Coq のプログラムへ変換する場合について述べる．OCaml のプログラムにおける let 式は，全て Coq でも let 式となるように変換している．OCaml のプログラムが let rec 式の場合には，後続の式が let rec 式によって束縛された名前による関数呼出のみであるか確認する．その場合には，再帰関数を局所的に定義しただけであると判断し，単体の fix 式になるように変換する．後続の式において，呼び出される関数名が異なっていたり，関数呼出以外の処理が記述されていた場合には，後続の式を用いる再帰関数であると判断し，let id := fix id ... というように，let rec で定義されていた再帰関数を fix 式に変換し，さらにその外側に let 式を用いて，let rec で束縛されていた名前に束縛する．これにより，Coq のプログラムでも，後続の式で局所定義した再帰関数を呼び出すことができる．

以下に，本システムを用いて OCaml から Coq へ変換する例を示す．

```
1 let last_element l default =
2   let rec last l =
3     match l with
4     | [] -> default
5     | x :: [] -> x
6     | _ :: l' -> last l'
7   in last l
```

last_element は，リストの最後の要素を取り出す関数である．この関数を本システムによって Coq プログラムへ変換すると次のプログラムが得られる．

```
1 Definition last_element l default :=
2   let last :=
3     fix last l :=
4       match l with
5       | Nil => default
6       | Cons x Nil => x
7       | Cons _ l' => last l'
8       end
9   in last l .
```

この例では，後続の式を用いた局所的な再帰関数の定義をする OCaml のプログラムであるため，Coq では let 式を用いて，後続の式を扱えるプログラムとして変換されている．

次に，本システムを用いて Coq から OCaml へ変換する場合の例を示す．

```
1 Definition last_element l default :=
2   (fix last l :=
3      match l with
4      | Nil => default
5      | Cons x Nil => x
6      | Cons _ l' => last l'
7      end) l.
```

この last_element もリストの最後の要素を取り出す関数であるが，定義を一部変更してある．この関数を本システムによって OCaml プログラムへ変換すると次のプログラムが得られる．

```
let last_element l default =
  (let rec last l =
     match l with
     | [] -> default
     | x :: [] -> x
     | _ :: l' -> last l'
   in last) l
```

この例では，fix 式を単体で使っている Coq のプログラムを変換しているため，OCaml のプログラムでは，let rec の後続の式では，局所定義した last という再帰関数を呼び出すという定義になっており，その定義に対して引数のリスト l を適用している．

上記の二つの例は，それぞれ逆向きに変換すると，元のプログラムを得ることができる．

5.7 型定義におけるコンストラクタの引数の変換

Coq における型定義の例を以下に示す．

```
Inductive binary_tree1 (A B : Type) : Type :=
| Leaf : A * B -> binary_tree1 A B
| Tree : binary_tree1 A B * binary_tree1 A B -> binary_tree1 A B.

Inductive binary_tree2 (A B : Type) : Type :=
| Leaf : A -> B -> binary_tree2 A B
| Tree : binary_tree2 A B -> binary_tree2 A B -> binary_tree2 A B.
```

binary_tree1 も binary_tree2 もどちらも二分木構造を表す型の定義である．Coq では，コンストラクタが受け取る型と返す型を記述している．binary_tree1 の定義を見ると，Leaf の引数は A * B -> binary_tree1 A B となっており，これは，A * B を受け取り，binary_tree1 A B という型を構成するということである．binary_tree2 の定義では，A -> B -> binary_tree2 A B のように，カリー化して定義されているため，A と B を受け取り，binary_tree2 A B を構成する．このように，Coq では，コンストラクタの引数を定義するとき，引数を組とするか，カリー化するかを選ぶことができる．次に，同じ二分木構造を OCaml で定義すると以下のようになる．

```
type ('a,'b) binary_tree =
| Leaf of 'a * 'b
| Tree of ('a,'b) binary_tree * ('a,'b) binary_tree
```

OCaml は Coq と異なり，コンストラクタが受け取る型だけを記述する．また，コンストラクタの引数は組でしか定義することができないという違いがある．したがって，コンストラクタを変換する際には，必要に応じて引数の形を変える必要がある．

Coq で記述したコンストラクタの引数を，OCaml のコンストラクタの引数へ変換する場合は単純である．引数を取らないコンストラクタは，そのまま引数なしとすればよい．引数を取る場合，定義が $Constr$: A -> B -> C であれば，C は引数ではないのでこれを除外し，$Constr$ of（A * B）という組を受け取るように変換する．

OCaml で記述したコンストラクタの引数を，Coq のコンストラクタの引数へ変換する場合は，変換対象のコンストラクタの定義が，Coq でどのように定義されているかに依存する．Coq で binary_tree2 のように，コンストラクタの引数がカリー化された定義がされている場合には，それに合わせ，カリー化された定義へと変換する．Coq において，情報を埋め込む対象のコンストラクタが，binary_tree1 のように，コンストラクタの引数を組で表現している場合や，Coq でそのコンストラクタが定義されていない場合には，引数は組として変換する．また，OCaml ではコンストラクタが構成する型を記述しないが，Coq ではそれを記述する必要があるため，コンストラクタが定義されている型の名前と，その型の型変数の情報を，コンストラクタの型を変換する関数へ与えている．前述の binary_tree であれば，binary_tree という名前と，'a 及び'b という型変数がこれにあたる．引数を取らないコンストラクタであれば，この型の名前と型変数を Coq へ変換した binary_tree A B を型情報として与える．引数を取るコンストラクタであれば，コンストラクタの引数の最後に-> binary_tree A B という型情報を付加する．これにより，OCaml で記述していない情報を復元でき，加えて，元々 Coq 上でカリー化した引数であれば，カリー化した形で変換することができる．

5.8 式におけるコンストラクタの引数の変換

以下に Coq において，式にコンストラクタが出現した場合について，上述の binary_tree1 と binary_tree2 を用いた例を示す．

```
 1 Fixpoint allsum1 (bt:binary_tree1 nat nat) : nat :=
 2 match bt with
 3 | Leaf (a, b) => a + b
 4 | Tree (bt1, bt2) => allsum1 bt1 + allsum1 bt2
 5 end.
 6
 7 Fixpoint allsum2 (bt:binary_tree2 nat nat) : nat :=
 8 match bt with
 9 | Leaf a b => a + b
10 | Tree bt1 bt2 => allsum2 bt1 + allsum2 bt2
11 end.
```

それぞれ，自然数を要素として持つ二分木の全ての要素を足し合わせる関数である．これら二つは，パターンマッチにおける左辺の記述方法が異なる．allsum1 は，コンストラクタの引数が組の場合の定義，allsum2 は，コンストラクタの引数がカリー化形式の場合の定義である．一方，前節で述べたように，OCaml ではコンストラクタの引数は組でしか与えることができない．し

たがって，型定義だけでなく，式においてもコンストラクタへの実引数の与え方を変える必要がある．

本システムでは，コンストラクタも関数の適用も，どちらも関数適用として構文解析をしている．したがって，$id\ expr_1\ ...\ expr_n$ という式が現れた際，id がコンストラクタなのか関数なのかを区別する必要がある．そこで，コンストラクタ名が全て大文字で記述されるという，本システムの制限を利用する．

はじめに，Coq から OCaml へ変換する場合について述べる．まず，関数適用があった場合には，適用する関数名を調べる．関数名が小文字で始まる場合には，それは関数の適用であると判断し，引数をそのまま変換する．大文字で始める場合，それはコンストラクタだと判断する．引数が 1 つの場合には何もせず，引数が二つ以上の場合には，それらを全て一つの組として括り，変換を行う．

OCaml から Coq へ変換する場合には，埋め込む対象の Coq のプログラムによって変化する．まず，OCaml の関数の関数名が大文字で始まるかどうかを調べる．小文字で始まる場合は，引数をそのまま変換する．大文字で始まる場合には，コンストラクタであると判断し，さらに情報を埋め込む Coq の関数適用の引数が 2 つ以上であるかを調べる．2 つ以上であれば，そのコンストラクタはカリー化された定義であると判断し，OCaml のコンストラクタの引数の組を，カリー化された形式へ変換する．1 つ以下の場合，OCaml のコンストラクタの引数の情報で Coq のコンストラクタの引数を上書きする．これにより，元の Coq のプログラムにおけるコンストラクタの引数の形式を保った変換を行うことができる．

6 評価

6.1 記述力

本システムの記述力を，OCaml のリストを扱う関数を集めた `List` モジュールを用いて評価する．OCaml の `List` モジュールには，48 個の関数が定義されている．これらの関数を本システムを用いて Coq プログラムへ変換した．変換結果を表 6.1 に示す．

変換時にエラーとなった関数が 4 個あった．まず一つは，特殊な記号を用いた式である．

```
1 let append = (@)
```

これは，中置演算子`@`の処理を `append` という名前に束縛する式である．本システムでは，特殊な記号を扱う式に対応していないため，変換エラーとなった．残りの三つは，いずれも逐次実行を行っている関数である．例を一つ示す．

```
1 let rec iter f = function
2   []    -> ()
3 | a::l -> f a; iter f l
```

上のプログラムのように，OCaml では式をセミコロンで繋ぐことで，繋がれた式を逐次実行することができる．しかし，Coq にはこのような構文がないため，本システムでは対応しておらず，変換エラーとなった．

次に，変換後 Coq での実行時にエラーとなるものが 14 個あった．これらはいずれもエラー処理を行っている関数であり，`invalid_arg`，`raise Not_found`，`assert false` のいずれかが使われていた．本システムでは，これらを関数適用と区別していなかったため，変換時にはエラーが出なかった．しかし，Coq ではエラー処理を行う関数を記述することができないため，Coq での実行時にエラーとなった．

第三に，停止性の証明を求められる関数が 3 個あった．これらは，パターンマッチの結果によって減少する引数が異なる関数や，相互再帰関数を内部に持つような関数であり，停止するこ

表 6.1　変換結果

エラーの種類	個数
変換時エラー	4
Coq での実行時エラー	14
停止性の証明が必要	3
利用可能	27
合計	48

とが容易に判断できない関数である．そのためこの 3 個の関数は，ユーザによる停止性の証明を必要とする．

最後に，残りの 27 個の関数はいずれも多相型の関数であるため，Coq ではその型を明示する必要があった．しかし，型注釈を行うことで，27 個の関数は全て利用することが可能であった．

以上より，本システムでエラーとなる関数は，もとより Coq 上で実装ができない構文を用いているものが殆どだった．したがって本システムは，Coq で利用できる構文に変換される OCaml プログラムについて対応しているといえる．

ただし，BiYacc では，ソースの情報がない場合や，ソースとビューの情報が大きく異なる場合，一部または全体を，ビューの情報のみを使ってソースを生成する．この方法で生成されたソースにはインデントなどの情報が無く，全て一行で出力されるという仕様がある．そのため，本システムを用いて，初めて OCaml プログラムを Coq プログラムへ変換した場合，Coq プログラムが一行で出力されるという問題点が挙げられる．しかし適切な位置に空白が挿入されているため，プログラムの動作には影響しない．したがって，ソースを整形するプログラムを挟むことで解決することができると考えられる．

6.2 変換速度

本システムの変換速度について評価する．実験環境は，CPU が Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz，メモリ 4GB の Ubuntu 15.04 上で行った．

まず測定 1 として，OCaml から Coq への変換速度を，関連研究である Coq of OCaml [5] 及び CFML [12] と比較した．変換対象は，OCaml の `List` モジュールの中で，本システムで変換することのできた 30 個の関数とした．これらを 100 回変換するのに要する時間を 50 回測定し，平均した結果を表 6.2 に示す．

また測定 2 として，Coq から OCaml への変換速度を Coq の Extraction と比較した．変換対象は，本システムで OCaml の `List` モジュールを変換して得られた 30 個の関数のうち，停止性の証明が必要な関数を除いた 27 個の関数とした．同様に，これらを 100 回変換するのに要する時間を 50 回測定し，平均した結果を表 6.3 に示す．

次に，本システムにおけるオーバーヘッドについて述べる．表 6.2 及び表 6.3 より，本システムは，OCaml から Coq への変換では，Coq of OCaml の約 12.5 倍，CFML の約 67.8 倍，Coq

表 6.2　測定 1 の結果

システム名	実行時間 [秒]
本システム	73.948
Coq of OCaml	5.931
CFML	1.090

表 6.3　測定 2 の結果

システム名	実行時間 [秒]
本システム	42.819
Coq	15.253

表 6.4　測定 1' の結果

変換種別	変換内容			実行時間 [秒]
get	OCaml プログラム	→	OCaml の構文木	19.014
	OCaml の構文	→	中間構文木	6.057
put	中間構文木	→	Coq の構文木	9.999
	Coq の構文木	→	Coq プログラム	37.114

表 6.5　測定 2' の結果

変換種別	変換内容			実行時間 [秒]
get	Coq プログラム	→	Coq の構文木	19.754
	Coq の構文	→	中間構文木	4.211
put	中間構文木	→	OCaml の構文木	6.448
	OCaml の構文木	→	OCaml プログラム	12.758

から OCaml への変換では，Coq の Extraction の約 2.8 倍の時間を要している．これは，本システムは比較対象のシステムと異なり，単方向の変換ではなく，双方向変換により，一方に他方の情報を埋め込む変換を行っているためであると考えられる．本システムでは，OCaml プログラムを Coq プログラムへ変換する際，OCaml プログラムだけでなく Coq プログラムも入力として受け取り，二つのプログラムを精査している．したがって，比較対象のシステムに比べて変換におけるオーバーヘッドが大きいのだと考えられる．

ここで，本システムで OCaml プログラムを Coq プログラムへ変換する場合，OCaml プログラムから OCaml の構文木へ変換，OCaml の構文木から中間構文木へ変換，中間構文木から Coq の構文木へ変換，Coq の構文木から Coq プログラムへ変換，という四つの変換を行う．これらの変換それぞれのオーバーヘッドについて考察する．測定 1' として，測定 1 と同じ条件で各変換にかかった時間を計測し，その結果を表 6.4 に示す．表 6.4 では，プログラムと構文木間の変換において，非常に時間がかかっている．この変換には BiYacc を利用しているが，BiYacc における構文の定義が適当でなく，無駄な精査をしているためではないかと考えられる．また，プログラムと構文木間の変換同士，構文木間の変換同士で比較をすると，ともに *put* に当たる変換の方が時間がかかっている．これは，*put* では，ビューの情報をソースへ埋め込むためにソースとビューの二つの情報を同時に精査するが，*get* はソースのみを精査する．そのため，*put* に当たる変換の方が時間がかかっていると考えられる．

同様に測定 2' として，測定 2 と同じ条件で計測した結果を，表 6.5 に示す．表 6.5 においても，全体としてはプログラムと構文木間の変換に時間がかかっており，構文木間の変換では *put* に当たる変換の方が時間がかかっている．これらの理由は上と同じであると考えられる．しかし表 6.5 のプログラムと構文木間の変換は，*get* に当たる変換の方が時間がかかっている．これは，

get と *put* の差以上に，Coq に対するプログラムと構文木を変換する BiYacc プログラムに問題があると考えられる．

以上のことから，本システムのオーバーヘッドの多くはプログラムと構文木の変換時に発生しており，特に Coq のプログラムと構文木の変換にかかる時間が大きい．したがって，この変換を行っている BiYacc プログラムの定義をより洗練することにより，オーバーヘッドを減らすことができると考えられる．

7 関連研究

7.1 Coq of OCaml

Coq of OCaml [5] は，OCaml によって実装された，OCaml の関数を Coq の関数に変換する変換系である．本来 Coq 上では記述できないエラー処理などの，副作用のあるプログラムも変換することができるという特徴がある．

Coq of OCaml では，OCaml プログラムを与えると，モナドを用いた，このシステム特有の型を持つ Coq プログラムへ変換する．そのため，元の OCaml プログラムとは型や構造が大きく変わってしまう．しかし，この変換によって得られる Coq プログラムを，Extraction によって再度 OCaml プログラムへ変換を行うと，元の OCaml プログラムとは構造が大きく異なるが，入出力に関して同じ振る舞いをするプログラムへと変換される．そのため，引数と返り値の関係を保つことで，プログラムの構造の変化を許容している．

このシステムの仕様は，一番初めに既存の OCaml プログラムを Coq プログラムへ変換し，それ以降の作業は全て Coq 上で行うことが前提となっている．したがって，一度 Coq プログラムへ変換した後は，OCaml プログラムは直接記述せず，変換によって得られた Coq プログラムを編集し，Extraction によってその都度 OCaml プログラムを得ることとなる．そのため，直接 OCaml プログラムを変更することが考慮されておらず，仕様の決定も Coq 側となっている．

Coq of OCaml が本システムと異なる点は二つある．一つ目は，本システムでは，できるだけ型や構造を保った変換を行っているという点である．二つ目は，本システムでは，OCaml を基点とした開発が行われることを想定している．そのため，OCaml プログラムの変更を考慮しており，OCaml 側で仕様を決定することができる．

7.2 CFML

CFML [12] は，OCaml によって実装された，OCaml の関数を Coq の公理に変換する変換系である．CFML では，OCaml プログラムを与えると，CFML の開発者の考案した Characteristic Formulae と呼ばれる関係式を基にした公理へ変換する．`ref` 型のような破壊的代入を含むプログラムも変換することができ，証明において有用なタクティクを数多く用意しているという特徴がある．

このシステムの仕様は，OCaml プログラムを Coq の公理へ変換し，それを証明することによって，元の OCaml プログラムの正しさを保証するというものである．また，このシステムによって得られるものは関数ではなく公理であるため，得られた Coq プログラムを Extraction によって OCaml プログラムへ変換しても，元の関数は得られない．そのため，Coq 側から証明を

しやすいようにプログラムへ変更を加えることができない．

本システムでは，Coq プログラムへの変更を OCaml プログラムへ埋め込むことができるため，証明を行いやすいように Coq 側からプログラムへ変更を加えることができるという点で，CFML とは異なっている．

8 おわりに

本研究では，部分的な変更を考慮した，OCaml プログラムに対する Coq における証明の支援を目的として，OCaml プログラムと Coq プログラムの双方向の変換を行うシステムを設計し，実装を行った．提案システムでは，OCaml プログラムへの変更を Coq プログラムへ反映できるため，OCaml 側でプログラムの仕様を決定できる．また，証明を容易にするために Coq プログラムへ加えた変更を，OCaml プログラムへ反映することができる．さらに，全ての変換プロセスを双方向変換としているため，以前のプログラムに対し，差分が小さなプログラムを得ることができ，既存の証明が再利用でき，証明の手間が軽減されることが期待される．

評価では，OCaml の `List` モジュールを Coq プログラムへ変換することで，Coq で扱える構文に変換される OCaml プログラムに対応していることを示した．また変換速度に関しては，ターゲットプログラムに対し，ソースプログラムの情報を埋め込むという処理のオーバーヘッドによって，比較対象のシステムよりも遅いことがわかった．

今後の課題は 4 つあげられる．まず，初めて変換した際などに一行で出力されるプログラムの整形である．これは，1 行で出力されてしまう場合に，整形を行うプログラムを挟むことで解決できると考えられる．次に，オーバーヘッドの削減による変換速度の向上である．これは，BiYacc で定義した構文を改良することで，変換におけるオーバーヘッドを減らすことができると考えられる．さらに，OCaml と Coq における，現在対応していない構文への対応である．これは，BiYacc へ定義を追加することで実現できるが，どのような構文があればより有用であるかを調査する必要がある．最後に，変換においてセマンティックが保存されることを保証することである．これは，BiGUL 自体を拡張するか，証明支援系 HOL からセマンティックを保存した変換を実現した CakeML [13] の手法を参考にすることで実現できる．

謝辞

本研究を行うに当たり，終始変わらぬ御指導を賜りました中野圭介准教授，岩崎英哉教授に深く感謝いたします．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申し上げます．

参考文献

[1] The Coq Development Team. *The Coq Reference Manual*, version 8.5pl3 edition, 2016. URL: `http://coq.inria.fr`.

[2] Xavier Leroy. Formal Verification of a Realistic Compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[3] Yoshihiro Imai. Applying Coq to Practical Software Development. TPP '10, 2010. http://www.itpl.co.jp/tech/formal/index.html.

[4] June Andronick, Boutheina Chetali, and Olivier Ly. *Using Coq to Verify Java CardTM Applet Isolation Properties*, pages 335–351. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[5] Guillaume Claret. Coq of ocaml. OCaml Workshop 2014, 2014.

[6] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-update Problem. *ACM Trans. Program. Lang. Syst.*, 29(3):Article 17, May 2007.

[7] Sebastian Fischer, ZhenJiang Hu, and Hugo Pacheco. The essence of bidirectional programming. *Science China Information Sciences*, 58(5):1–21, 2015.

[8] Hsiang-Shang Ko, Tao Zan, and Zhenjiang Hu. Bigul: A Formally Verified Core Language for Putback-based Bidirectional Programming. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '16, pages 61–72, New York, NY, USA, 2016. ACM.

[9] Ulf Norell. Dependently Typed Programming in Agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming*, AFP'08, pages 230–266, Berlin, Heidelberg, 2009. Springer-Verlag.

[10] Tim Sheard and Simon Peyton Jones. Template Meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, pages 1–16, New York, NY, USA, 2002. ACM.

[11] Zirun Zhu, Yongzhe Zhang, Hsiang-Shang Ko, Pedro Martins, João Saraiva, and Zhenjiang Hu. Parsing and Reflective Printing, Bidirectionally. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2016, pages 2–14, New York, NY, USA, 2016. ACM.

[12] Arthur Charguéraud. Program Verification Through Characteristic Formulae. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Program-*

ming, ICFP '10, pages 321–332, New York, NY, USA, 2010. ACM.

[13] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. Cakeml: A verified implementation of ml. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 179–191, New York, NY, USA, 2014. ACM.