定理証明支援系言語 Coq からの Java プログラム抽出

渡辺　優樹　（指導教員　西村　俊二）

平成30年1月26日

Java Program Extraction from Theorem Proving Support System Language Coq

Yuki Watanabe　(ACADEMIC ADVISOR　Shunji Nishimura)

概要:プログラムにバグがないことを保証できる言語として，定理証明支援系言語がある．定理証明支援系言語の１つであるCoqでは，アルゴリズムを記述し正当性を証明した後，OCamlやHaskell，Schemeなどの関数型言語のプログラムを抽出することができる．しかし，抽出先の言語が3つのみであり実用的でない．さらに多くの言語への抽出が可能になればCoqの使用者は増加し，バグのないプログラムも増加すると考えられる．．これまで，RubyやScala言語の抽出について研究が行われている．本研究では，CoqからJavaのプログラム抽出の簡単な流れと，Coqの型からJavaの型への変換を考え，Coqで記述したアルゴリズムからどのようなJavaプログラムが抽出されるか具体例を示した.

キーワード: Coq，Java，証明駆動開発，関数型言語

1．はじめに

プログラムは私達の生活と密接な関係があり，プログラムのバグ 1 つが命に関わることもある．これまで．バグの発生を抑えるために，様々な開発方法が編み出されてきた.

現在，バグの発生を抑える開発方法として，テストを用いた開発が一般的である．記述するプログラムが所望の性質を満たすことをテストで確認することでプログラムの動作を保証する．しかし，抜け漏れやレアケースを見逃す事があり，全てを完全にテストすることは難しい.

テストの代替として，プログラムの性質の証明をする開発方法が証明駆動開発である[1]．証明駆動開発は定理証明支援系言語を使う開発方法であり，プログラムの満たすべき性質を記述，証明し，他の言語のプログラムに抽出する[2]．よって，証明駆動開発はテストなしでプログラムにバグがないことを保証できる.

定理証明支援系言語であれば，証明駆動開発を行えるが，証明の記述が正しいことの判断を機械的に行える(以降,証明の機械的確認とする)かつ，プログラムの変換前後でプログラムの振舞いが観測等価であると保証できることが望ましい．Coq[3]は，両方の性質を満たしており，証明したプログラムを他言語に抽出する機能を備えている．また，変換先プログラム言語を追加できる構造となっている.

しかしながら Coq は扱える技術者が少なく，証明に必要な工数が比較的大きいため証明駆動開発は普及していない.また，抽出先の言語として公式でサポートされているのは OCaml，Haskell，Scheme の 3 つのみであり，少ない．抽出先の言語の選択肢が増加すれば，Coq の使用者は増加し，証明駆動開発も普及すると考えられる．これまで，Ruby や Scala 言語[4]の抽出について研究が行われている[5]．また，Java プラットホーム上での Coq 検証済みコードの実行についての研究も行われている[6]．しかし，Java プログラムの抽出については報告されていない.

関数型言語の抽出方法では Java プログラムの抽出は難しく，Coq から MiniML という中間言語に変換した後，抽出先の言語のプログラムとして書き出す，しかしながら，MiniML に変換する段階で型情報が消失する．Java は，型を明示する必要があるため適用できない．また，Coq から Java のプログラムに変換するときに型変換を行う必要がある．Java は関数型言語ではないので，Coq とは型が異なる．プログラムの変換前後でプログラムの振る舞いが観測等価であるために，正確な型変換を行う必要がある.

本研究では，Coq から Java プログラムの抽出が関数型言語の抽出と同じ方法を適用できないこと，Coq から Java の型へ正確に変換しなければならないことの 2 つに着目し，Coq から Java プログラムの抽出方法と Coq と Java の型の対応について考察した.

本論文の構成は以下の通りである．第 2 節で Coq について詳しく述べる．第 3 節では Coq から他言語のプログラム抽出方法について説明する．第 4 節では Coq から Java の型変換について述べ，Coq で記述したプログラムからどのような Java プログラムが抽出されるかについて例を挙げて説明する．最後の節でまとめと今後の課題について述べる.

2. Coq の概要

2.1 定理証明支援系言語

定理証明支援とは，ステップバイステップでコンピュータに命令を送り，数学の定理などの証明を補助するシステムである．主な言語として Coq，Agda[7]の 2 つが挙げられる．第 2 節では Coq について述べた後，Coq での関数や型の定義方法や証明，プログラムの抽出の仕方について簡単に説明する．

2.1.1 Coq について

Coq は定理証明支援系言語の 1 つで，フランス国立情報学自動制御研究所によって開発されている．型や関数を自ら定義することにより，関数型プログラミング言語として用いることができる[8]．また，カリー＝ハワード同型対応[9]（数学的証明と関数型プログラムの間の直接的な対応関係のことで，「プログラム＝証明」，「型＝命題」などとして知られている）により，証明を構成することができる．正当性を証明するとプログラムを抽出することが可能で，関数型言語である OCaml（ML），Haskell，Scheme の 3 つの言語の抽出が公式でサポートされている．

2.1.2 型の定義

以下は Coq での型の定義の例である．

```
Inductive day : Type :=
    | Monday : day
    | Tuesday : day
    | Wednesday : day
    | Thursday : day
    | Friday : day
    | Saturday : day
    | Sunday : day.
```

Inductive コマンドで Coq に対して，新しいデータ型のセット（集合）である‘型’を定義できる．今回は day を，Type 型をもつ型として定義している．2 行目以降の’|’以降の Monday，Tuesday，Wednesday…が型 day の構成子で，それぞれ day 型である．

2.1.3 関数の定義

以下は Coq での関数の定義の例である．2.1.2 で定義した型 day を使用する．

```
Definition next_day (d:day) : day :=
    match d with
      | Monday => Tuesday
      | Tuesday => Wednesday
      | Wednesday => Thursday
      | Thursday => Friday
      | Friday => Saturday
      | Saturday => Sunday
      | Sunday => Monday
    end.
```

関数の定義は Definition コマンドや Fixpoint コマンドでできる．Fixpoint コマンドは再帰関数を定義するときに使用するが，今回は再帰的でないため Definition コマンドを使用する．1 行目の next_day が関数名である．（d : day）は引数 d と，型が day であることを示す．“:day”は結果の型が day であることを示しており，(d : day) : day 全体が関数 next_day の型である．”:=”以降が関数の定義で，今回は match 構文により d によるパターンマッチを行っている．next_day 関数では d が Monday であった場合は Tuesday を，Tuesday であった場合は Wednesday を，Wednesday であった場合は Thursday を返す．

2.1.4 記法の定義

以下は Coq での記法の定義の例である．

```
Notation “x + y” :=
            (plus x y) (at level 50, left associativity)
            : nat_scope.
```

Coq では，Notation コマンドや Infix コマンドで，新たに記法を定義することができる．上は，“plus x y”を”x+y”で表現可能にするものである．

2.1.5 証明

Coq による証明は，tactic と呼ばれる言語を用いて対話的に証明していくのが一般的である．以下にその対話証明モードでの証明の例を示す．

```
Definition prop1 : forall (A : Prop),A->A.
intros.
apply H.
Qed.
```

上記は証明の全文である．2.1.3 の関数の定義の場合と同じく Definition コマンドを使用している．関数の定義の場合は型の後の“:=”以降に関数の定義を記述したが，すぐ“. ”で締めることで対話証明モード（proof-editing mode）に入ることができる．今回のプログラムは命題名を prop1 とし，「任意の命題 A に対して，A ならば A である」を証明すべき定理としている．prop1 の 1 行目までを Coq に読み込ませると以下のように出力される．

```
1 subgoal
```

```
_____(1/1)
forall A : Prop, A -> A
```

3 行目は prop1 の型が表示されているが，これはサブゴールと呼ばれ, 証明においての現在の結論を示す. 1 行目の 1 subgoal はサブゴールが 1 つあることを示している. 2 行目の 1/1 は 1 つあるサブゴールのうち 1 つ目を表示しているという意味で，サブゴールが 2 つあると 1/2 と表示される．2 行目までを Coq に読み込ませると以下のように出力される.

```
1 subgoal
A : Prop
H : A
_____(1/1)
A
```

4 行目の“__”の上側は現在の前提(宣言および定義)を，下側は現在のサブゴールを示す．prop1 の 2 行目の intros tactic により関数型であったサブゴールの引数の型がすべて前提に移動していることが分かる. prop1 の 3 行目を読み込ませると Proof completed.と表示され，証明終了となる．4 行目は証明終了の目印として記述する.

今回の証明では Definition コマンドを使用して対話証明モードに入ったが，他にも証明に入れるコマンドが存在する.

```
Theorem　：定理
Lemma　：補題(他の命題を証明するための小定理)
Remark　：注意
Fact　　：事実
Corollary：系（ある定理から直ちに導かれる定理）
Proposition　：命題
```

2.1.6 プログラムの抽出

Coq では Extraction コマンドを使用して，正当性を証明したプログラムを他の言語のプログラムに抽出することができ，Coq で性質を証明したプログラムを他の言語で組まれたプログラムに組み込むことが可能になる. 現在, 公式で対応している言語は Ocaml, Haskell, Scheme である．例を以下に示す.

```
Definition plus (n : nat)(m : nat) : nat :=
                n + m.
```

関数 plus は 2 つの nat 型の引数を受けとり，2 つの値を加算して返す関数である．関数 plus を Haskell で出力する場合，次のようにコマンドを入力する.

```
Extraction Language Haskell.
Extraction "plus.hs" plus.
```

以下の内容で plus.hs ファイルが出力される.

```
module Plus where

import qualified Prelude

data Nat =
   O
 | S Nat

add :: Nat -> Nat -> Nat
add n m =
  case n of {
   O -> m;
   S p -> S (add p m)}

plus :: Nat -> Nat -> Nat
plus n m =
   add n m
```

3. Coq からのプログラム抽出方法

Coq から他の言語のプログラムを抽出する具体的な方法について述べ，Java プログラムに変換する場合に用いることができる方法について説明する.

3.1 中間言語を介する方法

中間言語（MiniML）は，通常の関数型言語のサブセットとなっており，MiniML の項や型は自然に関数型言語のプログラムとして書き出すことができるが，Java などの非関数型言語のプログラムに書き出すのは難しい.

3.1.1 従来の方法

Coq は，記述したプログラムから直接他の言語のプログラムに抽出しない．まず，Coq のプログラムから MiniML という中間言語を抽出する．次に，MiniML からそれぞれ対象の言語に変換する. 関数型言語として，ML に類似したものであれば同じ方法を使用することができる．抽出の正当性については Letouzey[10]が証明した．中間言語 MiniML に抽出された段階で型情報が消失するため，変換先の言語は型推論を持っていることが必須であり，型推論を備えていない言語への変換は難しい.

3.1.2 型情報を復元する方法

3.1.1 において MiniML を抽出した後，MiniML’

（MiniML に型情報を追加した言語）に変換し，他の言語のプログラムを抽出する方法である．MiniML から MiniML'への変換はアルゴリズム M''[5]（アルゴリズム M[11]をベースに改良した型推論アルゴリズム）を使用する．型推論を備えていない言語の抽出も可能であるが，抽出先の言語は MiniML の特性上，ラムダ式（無名関数）の機能を備えている必要がある．Java の抽出は可能であるが C 言語の抽出は難しい．Scala 言語の抽出が製作されている．

3.2 中間言語を介さない方法

3.2.1 直接抽出する方法

Coq プログラムを，中間言語を介さずに，直接他言語のプログラムを抽出する方法は，MiniML を使用しないため，抽出先の言語はラムダ式（無名関数）の機能を備える必要はない．従って，抽出後のプログラムは，より多くの人が理解しやすいようにラムダ式を使わない方が望ましい．

3.3 Coq から Java プログラムの抽出方法

これまで Coq から他言語へ変換する方法について述べたが，Java へ抽出する方法は 2 通りあると考えられる．

3.3.1 中間言語を介する方法

3.1.2 で述べた方法と同様.

メリット

・Scala 言語の抽出が作成されている.

問題点

・ラムダ式で記述する必要があり，ラムダ式の記述に慣れていない人にとっては，抽出後のプログラムが読みづらい.

・Coq と Java における型の違い.

3.3.2 直接抽出する方法

3.2.1 で述べた方法と同様.

メリット

・抽出後のプログラムはラムダ式で記述する必要はないので，ラムダ式に慣れていない人でも使いやすい.

問題点

・実装が難しい.

・Coq と Java における型の違い.

4. Coq と Java のプログラムの対応

第 3 節で，Coq から Java プログラム抽出の方法について，2 通り述べたが，共通の問題点として，Coq と Java における型や関数の定義の違いが挙げられる．プログラムの変換前後でプログラムの振る舞いが観測等価であるために，正確な型変換を行う必要がある．第 4 節では Coq の型から Java の型への変換について述べ，Coq プログラムが最終的に抽出される Java プログラム について説明する.

4.1 Coq から Java の型・関数の変換

Coq から Java の型・関数の変換の説明をする.

4.1.1 自然数

（Coq）

```
x : nat
```

（Java）

```
int x
```

Java の int は自然数のみを表す型ではないが, Java には基本のデータ型を新たに定義する方法はないので代用として int 型を扱う．int 型に変換することにより負の値を持てるようになる点には注意を払う必要がある.

4.1.2 真偽値

（Coq）

```
x : bool
```

（Java）

```
boolean x
```

真偽値に関しては Coq と Java で違いはなく, それぞれ扱う値は true と false の 2 つである.

4.1.3 四則演算

（Coq）

```
+, −, *, /
```

（Java）

```
+, −, *, /
```

四則演算は Coq と Java で同じように使える．ただし，Coq で除算するときは，“Reals” をインポートする必要がある.

4.1.4 リスト

（Coq）

```
x : list nat
```

（Java）

```
ArrayList<Integer> x
```

Coq の list と同様の構造をしているのは，Java の ArrayList であると考えられる．しかし，Java ではリストを扱うときに，x::l などといった書き方はできない．Coq のように直感的に，短く記述することができないため，リストを扱う際，現状では変換後のプログラムの可読性が低くなると考えられる．従って，リストの変換に関しては今後検証する必要がある.

4.1.5 関数呼び出し

（Coq）

```
next_day Monday
```

（Java）

```
next_day(Monday)
```

関数呼び出しは Coq と Java において差異はないが，Java では渡す値は()で囲んでおく必要がある．next_day 関数は，2.1.3 のものである．

4.1.6 if 文

（Coq）

```
if leb a x then …
else …
```

（Java）

```
if(leb(a x)){
        …
}else{
        …
}
```

leb 関数は，2 つの値を比べ，第 1 引数が第 2 引数よりも小さければ true，それ以外は false を返す関数である．…の部分には処理が入る．Coq では条件分岐のために if 文のみを使用したりすることはない．条件に応じて処理を変えたい場合は，match 式を使用することが多い．

4.1.7 match 式

（Coq）

```
match l with
    |c0 => l1
    |…
    |cn => ln
end.
```

（Java）

```
if(l == c0){
      l1
}else if(…){
    …
}else {
      ln
}
```

（Java（外部ライブラリ使用，ラムダ式可））

```
match(l,
    caseBoolean_(c0, …),
    …
    caseBoolean_(cn, …)
}
```

Coq の match 式は，条件分岐式の 1 つである．Java では switch 式が同様の書き方をするが，switch 式の条件式には定数のみ記述可能であるため，switch 式では代用できない．従って，ラムダ式の記述をしない場合，if 文で条件分岐を行うのが同じ処理として当てはまると考えられる．

ラムダ式で記述する場合は外部ライブラリ pattern-matching4j の match メソッドを用いる方法が考えられる．match メソッドは，Scala の match 式を Java で実装したもので，Scala の match 式は，Coq から Scala へ変換するときに，Coq の match 式と対応している．従って，match メソッドは，Coq の match 式と同様の処理が可能であると判断した．

・match メソッド（pattern-matching4j ライブラリ）

match メソッドは第 1 引数に対象の値，第 2 引数以降は各ケースの表現（条件関数+処理関数）を可変長引数で渡す．ラムダ式で記述することができる．caseBoolean_メソッドは第 1 引数にマッチング条件として真偽値を返す関数を指定，第 2 引数にマッチしたときに実行される関数を指定し，値を返す．今回は真偽値で条件分岐を判断するパターンであるが，定数パターンとして caseValue_メソッドがある．

4.2 Coq から Java プログラムの抽出

Coq から Java プログラムを抽出する上で，型や関数以外でも必要のない部分や置き換える部分が存在する．また，実用性のないため抽出する必要がないと考えられるプログラムもある．

4.2.1 抽出しないもの

・命題・証明

命題や証明は，計算結果に直接影響するものではなく，仮に Java のプログラムに変換したとしても実用性はない．従って変換はしない．同様の理由により，型 Prop の Java の型への対応も必要ない．

・コマンド

コマンドは，プログラムが定義なのか，定理の証明なのかといった判別には利用できるが，計算結果に直接影響しない．従って，変換は必要ないが，Definition コマンド以外の対話証明モードに入るコマンド（2.1.5 参照）を使用していた場合，証明であると判断して，変換は行わないなどの処理の実装には利用できると考えられる．

4.2.2 抽出の流れ

4.1 で説明した型変換以外ついて説明する．これらは主に，3.2.1 の方法に関連する．

・関数名

Coq の関数名は，Java のメソッド名としてそのまま活用できる．

・関数の型

（Coq）

コマンド 関数名 (引数:型): 返り値の型 ;=…

（Java）

返り値の型　メソッド名（型　引数）{…}

Coq と Java では関数（メソッド）の記述の仕方が少し異なるので，上記のように変換する．

4.3 Coq と Java プログラムの変換例

Coq のプログラムが最終的に抽出される Java プログラムについて 3 つの例をあげて説明する．3.2.1 の直接抽出する方法で，Java プログラムを抽出した場合をメインに説明する．変換後のプログラムは，メソッド部分のみ掲載する．

4.3.1 例 1

（Coq）

```
Definition plus (n : nat) (m : nat) : nat :=    n + m.
```

（Java）

```
Integer plus ( int n,int m){
  int x=0;
  x = n + m;
  return x;
}
```

関数 plus は引数を 2 つ受け取り，和を返す関数である．Coq のプログラムを見ると，コマンドが Definition であり，証明や命題でない可能性が高いので，プログラムの抽出を続行する．関数名 plus は，Java のメソッド名に使う．(n : nat) (m : nat)の部分は引数の部分であるが，Java では nat は int に変換するので，その 2 つの引数をまとめて(int n, int m)と記述する．引数を表す部分の後は返り値の型が記述してあり，今回は nat 型であることが分かる．従って，Java では返り値は int 型とする．同時に，返り値を扱う変数を生成する．上記の Java プログラムでは“int x=0;”に相当する．ここまでは関数の型の変換である．4.2.2 のように記述する．

“:=”という記号は Java でいうところの{}に相当し，関数の処理の記述がある．上記の Coq の関数では n+m とのみ記述される．従って，n+m 以外の処理は行わないことが分かるので，返り値に n+m を代入して値を返すという処理を Java で記述する．上記の Java プログラムでは“x = n + m; return x;”に相当する．

4.3.2 例 2

（Coq）

```
Fixpoint mult (n m : nat) : nat :=
    match n with
      | O => O
      | S n’ => plus m (mult n’ m)
    end.
```

（Java）

```
Integer mult(int n, int m){
  int x=0;
  if(n == 0){
      x = 0;
  }else {
      x = plus(m, (mult(n−1, m)));
  }
  return x;
}
```

（Java（外部ライブラリ使用，ラムダ式可））

```
Integer mult(int n, int m){
  return match(n,
      caseValue_( 0, o −> 0),
      caseDefault_( o −> plus(m, (mult(n−1, m)))
  );
}
```

2 つの引数を受け取り，積を再帰的に計算し，結果を返す関数である．Coq のプログラムの関数 S は，受け取った引数に 1 を足した数を返す関数で，今回は n’（任意の数）に 1 を足した数を返す．以上を踏まえて説明する．今回は，外部ライブラリを使用した場合のパターンについての説明も行う．

Coq のプログラムを見ると，コマンドが Fixpoint であり，プログラムは証明や命題ではないので抽出を続行する．関数名 mult は Java のメソッド名として使用する．(n m : nat)は引数を表している．Java では nat 型を int 型にするので，(int n, int m)とする．関数 mult の返り値は nat 型，即ち Java では int 型にする．例 1 と同様に変数を生成する．

関数の処理の部分に着目する．今回の関数では，match 式が使用されている．match 式は if 文によって代用できるので，if 文で同じ意味になるように Java で記述する．S n’には 0 以外のすべての自然数が当てはまる．従って，今回の条件分岐は引数 n が 0 であるか，それ以外であるかに分かれている．S n’を Java で強引に記述すると n == (n−1)+1 といった数式になるが，Coq で S n’という条件式が来るのは，match 式では最後の分岐であることが多い．よって Java では else と記述する．後はそれぞれの分岐先の処理の結果を先ほど用意した変数 x に代入し，最後に計算結果を return 文で返せばプログラムの抽出は終了である．今回は処理に再帰が使われているため，関数の呼び出しとして 4.1.5 に示す処理を行う．次に，外部ライブラリを使用する場合について説明する．関数の型の変換については説明を省く．今回の場合は引数の値を保存する変数はあらかじめ生成する必要はない．

Coq の match 式は外部ライブラリを使用すると 4.1.7 のように記述できる．今回は条件分岐のメソッドとして caseBoolean_メソッドではなく，caseValue_メソッドを利用する．まず，条件分岐の 1 つ目として，第 1 引数のマッチング条件は 0 とのみ記述する．これは match メソッドの第 1 引数として渡された値が 0 であるか判断する．第 2 引数では行う処理を記述する．今回は 0 を返す．条件分岐の 2 つ目のメソッドは caseDefault_メソッドを使用する．caseDefault_メソッドはここまでの条件に当てはまらなかったものが必ず通るメソッドである．処理は x −> plus(m, (mult(n−1, m)))と記述する．“−>”の右側は呼ばれる関数となっており，外部ライブラ

リを使用しない場合と同様に記述する．

4.3.3 例 3

（Coq）

```
Definition sample (n m:bool) : bool :=
  match n,m with
    | true , false => true
    | false , true => true
    | _ , _ => false
  end.
```

（Java）

```
Boolean sample(Boolean n, Boolean m){
  Boolean x=true;
  if(n==true && m==false){
        x=true;
  }
  else if(n==false && m==true){
        x=true;
  }
  else {
        x=false;
  }
  return x;
}
```

（Java（外部ライブラリ使用，ラムダ式可））

```
Boolean sample(Boolean n, Boolean m){
   return match(n, m,
        caseBoolean_(true, false, (i1, i2) -> true),
        caseBoolean_(false, true, (i1, i2) -> true),
        caseDefault((i1, i2) -> false)
   );
}
```

2 つの引数を受け取り，両方の真偽値が同じであれば false，それ以外は true を返す関数である．今回も外部ライブラリを使用しない場合と使用する場合の 2 つに分けて解説する．

Coq のプログラムを見ると，コマンドは Definition である．従って，関数 sample の Java の抽出を続行する．関数名は Java のメソッド名として利用する．(n m:bool) は引数を表しており，今回は，引数は 2 つで型は bool であるため，Java では Boolean 型に変換する．関数の返り値は bool 型であることが分かるため，従って，Java でも先ほどと同様に返り値は Boolean 型とし，4.2.2 に倣って記述する．返り値を管理する変数を生成する．

次に，関数の処理の部分に着目する．今回も match 関数が使われているが，マッチ対象は 1 つではなく，2 つである．2 つになっても主な流れは 4.3.2 の時と同じである．Coq ではコンマで条件が区切られているが，Java では&&で区切る．最初の分岐は true , false なので，Java に直すと n==true && m==false となる．2 つ目の分岐も同様．3 つ目の分岐に関しては，Coq では“_,_”と記述してある．条件分岐における“_”は，ワイルドカードパターンと呼ばれるもので，任意の値でマッチし，値が使用されないことを意味している．今回は最後の分岐であり，全てのパターンがここを通るので else を使用する．各条件分岐の後に，各々の処理を記述する．今回はそれぞれ真偽値を返すだけなので，先ほど生成した変数に値を代入し，最後に return 文で返し，関数の処理は終了となる．

次に外部ライブラリを使用する場合について説明する．

今回は条件分岐に使用する変数が Java でいう Boolean 型であることが分かっているので，条件分岐のメソッドとして caseBoolean_メソッドを使用する．第 1 引数にマッチング条件として真偽値を返す関数を指定する．今回は真偽値を Coq のプログラムの通りに記述する．マッチ対象が 2 つなので，2 つ記述する．その後の処理は，Coq のプログラムと同様に記述する．最後のワイルドカードパターンについては caseDefault_メソッドを使用する．

5. まとめと今後の課題

Coq のプログラム抽出先の言語は，Ocaml，Haskell，Scheme の 3 つのみであり，Java の抽出は実装されていない．本論文では，Java プログラムの抽出の流れと，Coq から Java の型変換の提案，Coq のプログラムが最終的に抽出される Java プログラムについて説明した．プログラムの抽出は，関数型言語であれば Coq から OCaml への変換のプログラムについて少し改良すれば実装可能であるが，Java では難しい．Java と Scala はともに JavaVM で動作するので，Scala の抽出方法を改良する方法での実装（3.1.2），もしくは直接抽出する方法での実装となる．Coq から Java への型変換においては，Coq で多用する型について変換の説明をした．しかし，Coq から Java の型変換は正確に出来ているとは言えず，今後検証する必要がある．そのため，抽出の具体的なプログラムについては提案できていない．

抽出の正当性の証明も行う必要がある．Coq のプログラムから Java のプログラムへ正しく変換されていることが証明されていなければ，実際のソフトウェア開発において使用されることはない．

課題はあるが全て解決し，Java プログラムの抽出が実現すれば Coq という言語がさらに多くの人に利用され，証明駆動開発が普及していくと考えられる．

謝辞

本研究に際して，様々なご指導を頂きました西村俊二講師に深謝いたします．また，この研究の機会をくださった情報工学科の先生方，そして多くの知識やご指摘を下さいました同研究室の同期の皆様に厚く御礼申し上げます．

参考文献

[1] 山本 晃治, 宗像 一樹 : 証明駆動開発の現実的な開発プロジェクトへの適用に向けて（ソフトウエアエンジニアリングシンポジウム 2016）

[2] Pierre Letouzey:Extraction in Coq : an Overview(Laboratoire PPS, Université Paris Diderot - Paris 7Case 7014, F-75205 Paris Cedex 13, France letouzey@pps.jussieu.fr)

[3] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant,Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet,César Muñoz, Chetan Murthy, et al.: The Coq Proof Assistant Reference Manual

[4] Martin Odersky : The Scala Language Specification

[5] 逸見 港, 田辺 良則, 今井 宜洋, 萩谷 昌己 : 検証済みのプログラムによる Coq から Scala へのプログラム抽出（日本ソフトウェア科学会第 31 回大会（2014 年度）講演論文集）

[6] 湯浅 能史, 田辺 良則 : Java プラットホーム上での Coq 検証済みプログラムの実行について（日本ソフトウエア科学会第 32 回大会（2015 年度）講演論文集）

[7] 池上大介：Agda プラグイン機構（産業技術総合研究所システム検証研究センター）

[8] 橋本　英樹（指導教員：武市　正人）：定理証明支援系 Coq を用いたプログラム運算（東京大学大学院情報理工学系研究科数理情報学専攻（2010 年））

[9] 石井忠夫：構成的型理論に基づいた定理証明プログラムの試作（新潟国際情報大学　情報文化学部　紀要）

[10] Letouzey, P.: Certified functional programming - Program extraction within Coq proof assistant, PhD Thesis, University Paris-sud, 2004

[11] Oukseh Lee, K. Y.: Proofs about a folklore let-polymorphic type inference algorithm, ACM Transactions on Programming Languages and Systems, Vol. 20, No. 4(1998), pp. 707–723