Bidirectional Certified Programming

Daisuke Kinoshita
University of Electro-Communications
kinoshita@ipl.cs.uec.ac.jp

Keisuke Nakano
University of Electro-Communications
ksk@cs.uec.ac.jp

Abstract

Certified programming is one of the desirable approaches to developing dependable software, where expected properties of programs are formally proved by proof assistants such as Coq. One way for certified programming with Coq is to define a function, give proofs for its properties in Coq, and then extract a program in OCaml. Another way for certified programming with Coq is to import the definition from OCaml and give proofs for its properties in Coq. Since translations in both methods are unidirectional, we can modify only either of Coq and OCaml. That makes it hard to develop large certified programs.

To solve this problem, we propose a new framework for certified programming through bidirectional transformation between Coq functions and OCaml programs. In our system, one can develop certified programs by modifying both Coq functions and OCaml programs alternatingly. All updates of the OCaml program are reflected to the Coq function, and vice versa, while reusing as many parts of the original one as possible.

1 Introduction

Certified programming is one of the desirable approaches to developing dependable software, where most expected properties of programs are formally proved by proof assistants such as Coq [Coq16]. The CompCert verified C compiler [Ler09] is a good model of certified programming. Products and systems developed by certified programming have been enabled to meet the formal requirements of the highest common criteria (ISO 15408) as "EAL7: formally verified design and tested". For example, Trusted Logic formalized the full specification of JavaCard using Coq and succeeded to be awarded the EAL7 [BC10].

The Coq system provides a mechanism of *program extraction* to support certified programming. All functions defined in Coq can be translated into a program described in (executable) programming languages, OCaml, Haskell, or Scheme. Thanks to the extraction mechanism, what we should do for certified programming is to specify the definitions of functions and to prove their properties in Coq. The portion of proofs and types is discarded by the extraction because they are not used for executing the program whose properties have already been verified. We may develop a fully verified software through program extraction by specifying all functions in Coq.

However, it is much harder to define functions in Coq than to write programs in generic programming languages like OCaml even though OCaml has a close connection with Coq for historical reasons and Coq itself is written

In: R. Eramo, M. Johnson (eds.): Proceedings of the Sixth International Workshop on Bidirectional Transformations (Bx 2017), Uppsala, Sweden, April 29, 2017, published at http://ceur-ws.org

in OCaml. The major difference is that we must care for totality of functions and termination of recursion in Coq. In addition, all functions have to be defined without either side effects or exception handling. These requirements make certified programing in Coq more difficult.

In actual development with certified programming, we make a compromise by specifying functions in Coq only for a part of the large programs. One may develop a software using both Coq and OCaml, where the whole program is written in OCaml and a part of functions in it are verified in Coq. There are two approaches to this mixed style of programing, *prove-and-extract* and *import-and-prove*.

In the prove-and-extract method, we define important functions in Coq and give proofs for their properties in Coq. Then we translate them into equivalent OCaml programs by the extraction mechanism of Coq so that they can be referred from the other user-written OCaml programs. In this style, we have to carefully specify functions in Coq because the program extraction of Coq may generate a program that behaves equivalently but has an unexpected type. For example, consider the following function of type `nat * nat -> nat` specified in Coq:

```
Definition add_pair (p: nat * nat) := let (x, y) := p in x + y.
```

which takes a pair of natural numbers and returns their sum. The program extraction produces an OCaml program

```
let add_pair = function Pair (x, y) -> add x y
```

that is semantically equivalent to the original one but has type `(nat, nat) prod -> nat` even if the OCaml function of type `nat * nat -> nat` might be expected. Although some cases may be solved by the representation mapping mechanism of Coq (e.g., `Extraction Inductive`), it is not easy in general to predict extracted OCaml programs from Coq scripts.

In the import-and-prove method, we write the whole program in OCaml language and only important functions in it are translated into functions to be accepted in Coq. Then we prove their expected properties in Coq independently from the original program. In this style, we never suffer from the unexpected program extraction of Coq. CFML [Cha11] and CoqOfOCaml [Cla14] are possible approaches to the import-and-prove method, in which one can verify existing OCaml programs by translating them into Coq. However, the proof of the properties cannot be reused if the specification of the OCaml programs is updated. We need to retranslate the program and prove the properties from scratch. In addition, the existing approaches generate different structures of functions in Coq. This also makes it hard to reuse the original proof for updates.

In this paper, we propose a new framework which combines both advantages of the prove-and-extract and import-and-prove methods. Our key idea is a bidirectional transformation between Coq functions and OCaml programs. In our system, one can develop certified programs by modifying both Coq functions and OCaml programs alternatingly. All updates of the OCaml program are reflected to the Coq function, and vice versa, while reusing as many parts of the original one as possible.

Our system is designed so as to meet the following requirements:

- The specification of functions are given in OCaml so that we can obtain the advantage of the import-and-proof style for certified programming. Since we need to combine with the other user-written OCaml programs in the final product, the specification should be provided by OCaml instead of Coq.

- OCaml programs are translated into Coq with preserving as much information as possible. Initial translation just generates an equivalent Coq script from the OCaml program. If the translation has been done before, Coq-specific portions, in particular statements and their proofs, should be left as they are. The user may have to revise the proof of properties of a function in Coq if its definition is updated in OCaml.

- Updates in Coq are reflected to OCaml as far as needed so that we can modify the definition of types and functions in Coq (without changing their behaviors) to make it easy to verify their properties. The translated Coq functions may be too complicated to treat for certification because OCaml has a less powerful type system than Coq (e.g., partial type application). Hence the Coq script should be modifiable and the modification should be reflected to OCaml whenever needed.

- Whatever is common between Coq and OCaml should be synchronized so that we can modify as flexibly as possible in the both side of Coq and OCaml. One may prefer to use the syntax only allowed in one side, e.g., inlined pattern in OCaml and binding variables sharing the same type in Coq. The system should translate them correctly and be preserved as much as possible against updating.

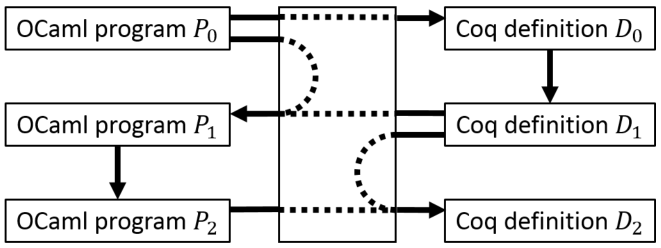

Figure 1: Possible scenario for our system

All of these requirements are achieved by combining with four bidirectional transformations between five data, Coq script text, Coq AST, common data, OCaml AST, and OCaml program text. Two bidirectional transformations between text and AST are implemented with BiYacc [KZH16] which preserving comments and indentations appropriately. The other two between ASTs are implemented with BiGUL [ZZK+16] which is a formally-verified bidirectional language.

2 System overview

This section explains how users develop a program through bidirectional transformation between Coq and OCaml, giving a possible scenario using our system and a concrete example of practical situations.

2.1 Possible scenario

Figure 1 shows a possible scenario using our system as follows:

1. We first write an OCaml program P_0 to be verified. Our system translates it into a function D_0 in Coq.
2. When the function definition in Coq is required to change for making it easier for the users to prove its property, we may modify the definition and give the proof. Our system translates the modified function D_1 into a new OCaml program P_1 which is obtained by updating with a possibly small modification.
3. When the specification of the OCaml program is required to be changed due to updates of the other user-written program, we may modify the translated program P_1 to one that satisfies the requirement. Our system translates the updated program P_2 into a new function definition D_2 in Coq as a possibly small modification for D_1.
4. The proof should be modified in general because the function definition is changed. Most of the original proof may be reused for the modification. Our system does not try to translate the updated script in Coq unless the function definition is changed.

This is a typical scenario for certified programming using our system, where can modify both OCaml programs and their corresponding functions in Coq. The 'possibly small modification' requirement in steps 2 and 3 is fulfilled by a put-based implementation written in BiGUL as mentioned later. Our solution in the current implementation may not be optimal, though. What should be emphasized here is that both Coq script and OCaml program can be a 'source' of bidirectional transformation in terms of bidirectional lenses [FGM+07]. The 'view' is an internal data which contains common information of the both languages.

2.2 Example of the usage

We shall show a concrete example of the usage of our system. Consider the case where the user initially writes an OCaml program:

```
type 'a binary_tree =
| Leaf of 'a
| Node of 'a binary_tree * 'a binary_tree

let rec size (bt : int binary_tree) : int = match bt with
| Leaf i -> i + 1
| Node (bt1, bt2) -> size bt1 + size bt2
```

where the `type` declaration specifies a polymorphic type `binary_tree` with a type variable `'a` and the `let rec` declaration defines a recursive function that computes a size of binary trees by summing up all integer values at leaves after incrementing by one. Our system translates the OCaml program into a script in Coq:

```
Inductive binary_tree ( A : Type ) : Type :=
| Leaf : A -> binary_tree A
| Node : binary_tree A * binary_tree A -> binary_tree A.

Fixpoint size ( bt : binary_tree nat ) : nat := match bt with
| Leaf _ i => i + 1
| Node _ (bt1, bt2) => size bt1 + size bt2
end.
```

where type `int` is translated into `nat` (type of natural numbers) in Coq by default. The user may modify `nat` into `Z` (type of integers) without changing the type `int` in OCaml after reflecting the update. Although the definition of recursive functions in Coq requires the evidence of termination in general, this kind of structural recursion is automatically guaranteed to terminate. Because of the difference of the type systems between Coq and OCaml, the constructors `Leaf` and `Node` have to take an additional argument for instantiating a type variable in the polymorphic type.

The user may certify the property of the `size` function that it returns a positive integer for any inputs. The Coq script is updated as:

```
Require Import Omega.

Inductive binary_tree ( A : Type ) : Type :=
| Leaf : A -> binary_tree A
| Node : binary_tree A * binary_tree A -> binary_tree A.

Fixpoint size ( bt : binary_tree nat ) : nat := match bt with
| Leaf _ i => i + 1
| Node _ (bt1, bt2) => size bt1 + size bt2
end.

Functional Scheme size_ind := Induction for size Sort Prop.

Theorem size_gt_zero : forall bt, 0 < size bt.
Proof.
  apply (size_ind (fun bt s => 0 < s)); intros; omega.
Qed.
```

where the statement `size_gt_zero` represents the property to be certified. The user adds a few commands for proving the statement. We do not give details of the proof here because it is not essential. In our system, this modification does not update the original OCaml program because the definitions of types and functions are left as they are.

Now consider the case where the specification of the `size` function is changed so that it counts the number of leaves as follows:

```
let rec size (bt : int binary_tree) : int = match bt with
| Leaf i -> 1
| Node (bt1, bt2) -> size bt1 + size bt2
```

where we just modify the branch `Leaf` in the pattern matching of `bt` to be `1` instead of `i + 1`. For this modification, our system updates only the definition of the `size` function as:

```
Fixpoint size ( bt : binary_tree nat ) : nat := match bt with
| Leaf _ i => 1
| Node _ (bt1, bt2) => size bt1 + size bt2
end.
```

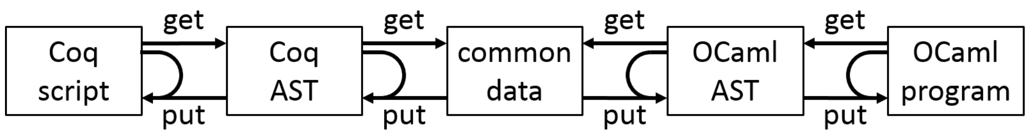

Figure 2: Data flow in our system

and the other descriptions are left as it was. Since our system leaves all proofs of statements as they were, the user might have to modify the proof of the `size_gt_zero` statement because the definition of the involved function `size` is changed. However, in this case, fortunately, any modification is not required due to the genericity of the `omega` tactic that solves Presburger arithmetic formulas. The original proof can be used without changing for this modification.

The user may change the type of natural numbers `nat` into the type of integers `Z` and replace `*` in the definition of inductive data type with `->` to make it curried. This modification does not affect updates of the OCaml program but makes it easy to prove its properties in many cases as explained in Section 3.2. In addition, if the termination of recursive definition is not automatically detected, the user should add the proof by redefining the function.

3 Design and implementation

We design our system so as to meet our requirements through combining four bidirectional transformations as mentioned in Section 1. This section gives a summary of architecture of our system and how we achieve a bidirectional transformation between a Coq script and an OCaml program.

3.1 Architecture of our system

Our system is based on combination of four bidirectional transformations as shown in Fig. 2 where common data contains shared information between Coq and OCaml ASTs. Two outer bidirectional transformations, between Coq script and Coq AST and between OCaml program and OCaml AST, are described in BiYacc [ZZK$^+$16] where the text side is a source and the AST side is a view. BiYacc makes it possible to modify an AST with preserving comments and indentations in the text. Two inner bidirectional transformations, between Coq AST and common data and between OCaml AST and common data, are described in BiGUL [KZH16] where the AST side is a source and the common data side is a view. BiGUL makes it easy to write a translation between Coq and OCaml because of its put-based semantics.

3.2 Filling the gap between Coq and OCaml

Even though Coq and OCaml have similar syntax because of their close relationship, the gap between them is not so small for translating one into another. For example, OCaml allows inlined patterns like `let f (x,y) = ...` in a function definition but Coq does not, and Coq allows curried data constructors like `Cons: A -> List -> List` but OCaml does not. Although such differences could be ignored just as unsupported notations, our system tries to allow them for flexible programming in both Coq and OCaml sides. We give a brief summary for their differences and how to deal with them in our system. For simplicity, we will explain here the bidirectional translation between Coq and OCaml as it were directly defined without intermediate common data. The actual translation of our system is done through an intermediate data structure which stores shared information between Coq and OCaml. This is implemented by combining with two bidirectional transformations in BiGUL.

Type conversion

Since Coq has a more powerful type system than OCaml, the translation from Coq to OCaml either drops some information or merges informative types into a single type. For example, a user-defined data type in Coq is usually declared as a subtype of `Type` or `Set` depending on the usage. This information is dropped in the translation to OCaml but preserved in the update reflection from OCaml. Another example is two types for the numbers in Coq, `nat` (natural numbers) and `Z` (integers). Our system translates these types into the same type `int` in OCaml. The update reflection of this type from OCaml to Coq preserves the original type.

Binding variables of the same type

Coq has abbreviated syntax for multiple parameters sharing the same type, while OCaml requires to declare a type for each parameter separately. One can write `(`x y z`:`τ`)` for `(`x`:`τ`)(`y`:`τ`)(`z`:`τ`)` in Coq. In our system, the translation from the separated style to the abbreviated one checks if the consecutive parameters have the same type. For example, consider the case where `(x y z:nat)` in Coq is translated into `(x:int)(y:int)(z:int)` in OCaml. When the specification in OCaml is changed to `(x:int)(y:int)(z:string)`, our system updates the Coq script as `(x y:nat)(z:string)`.

Inlined parameter patterns

OCaml allows to describe a pattern as a parameter in the function definition like `let f (x, y) = x + y`, which is not supported in Coq. Our system solves the problem only for a pair pattern. Introducing a new variable `x_y`, this OCaml program is translated into

```
Definition f x_y := let (x, y) := x_y in x + y.
```

where two parameters `x` and `y` are extracted from `x_y` by the `let` binding. Patterns other than pairs in OCaml can be supported in a similar way.

Locally-defined recursive functions

Coq and OCaml have different syntax to represent a recursive function in a local context. An expression `fix` f x `:=` e in Coq is evaluated to a recursive function itself, while an expression `let rec` f x `=` e `in` e' in OCaml is evaluated by e where f is bound to the recursive function. For Coq-to-OCaml translation, the expression `fix` f x `:=` e maps into `(let rec` f x `=` e `in` f`)` as default. When the expression occurs at the beginning of the `let`-binding (of the same function name) like `let` f `:= (fix` f x `:=` e`) in` e', the whole expression is translated into `let rec` f x `=` e `in` e'. For OCaml-to-Coq translation, the expression `let rec` f x `=` e `in` e' is translated into `let` f `:= (fix` f x `:=` e`) in` e'.

Curried data constructors

Coq allows a curried data constructor like `Node` in the following definition.

```
Inductive binary_tree1 (A: Type): Type :=
| Leaf : A -> binary_tree1 A
| Node : binary_tree1 A -> binary_tree1 A -> binary_tree1 A.
```

For example, `Node (Leaf 1) (Leaf 2)` is a binary tree of type `binary_tree1 nat`. OCaml requires uncurried style for data constructors like

```
type 'a binary_tree2 =
| Leaf of 'a
| Node of 'a binary_tree2 * 'a binary_tree2
```

where `Node` must take a pair of binary trees to construct a binary tree. The type variables are translated into those in the OCaml style. As we have seen in Section 2.2, our system translates into

```
Inductive binary_tree2 (A: Type): Type :=
| Leaf : A -> binary_tree2 A
| Node : binary_tree2 A * binary_tree2 A -> binary_tree2 A.
```

where the `Node` constructor is not uncurried. This uncurried definition for inductive data types is known to be quite harmful for proving the statement in Coq because its induction principle is futile unlike that of the curried type `binary_tree1`.

To solve this problem, our system allows the users to change the definition in Coq into the curried one like `binary_tree1` without affecting the reflection to the OCaml programs. Accordingly, every occurrence of `Node` e_1 e_2 in Coq is translated into `Node(`e'_1`,`e'_2`)` in OCaml where e'_1 and e'_2 are appropriate translations of e_1 and e_2, respectively.

Curried type constructors

Coq allows also a curried type constructor like `either` in the following definition:

```
Inductive either (A B: Type): Type :=
| Left : A -> either A B.
| Right : B -> either A B.
```

corresponding to

```
type ('a, 'b) either =
| Left of 'a
| Right of 'b
```

in OCaml where polymorphic type variables should be passed as a tuple. Our system assumes that all type constructors of Coq are curried and those of OCaml are uncurried, hence it converts the curried/uncurried style for Coq/OCaml translation.

Function expressions

OCaml has special syntax for function expressions `function` ... that is equivalent to `fun` x `=> match` x `with` ... `end` in Coq. This is a convenient syntax sugar for pattern matching functions in OCaml. The translation of this expression into Coq is not so straightforward. Consider the recursive definition of the `fact` function:

```
let rec fact = function 0 -> 1 | n -> n * fact (n - 1)
```

This should not be simply translated into

```
Fixpoint fact := fun X => match X with 0 => 1 | n => n * fact (n - 1) end.
```

with a fresh variable `X` because the recursive function must have parameters. Our system solves the problem by translating it into

```
Fixpoint fact X := match X with 0 => 1 | n => n * fact (n - 1) end.
```

by detecting that the location is the beginning of the body of the recursive function definition.

3.3 Limitation of syntax

Our system imposes several restrictions to both Coq and OCaml. First, it is for data constructors in Coq. We must use capital names for data constructors and partial application of data constructors are not allowed. Second, user-defined prefix and infix operators are prohibited in OCaml. This is because of the current limitation of the BiYacc parsers. Additionally, we only support the basic feature of Coq and OCaml. Record, type class, and extending notation are not supported.

4 Related work

There are several approaches to certified programming by a 'unidirectional' translation but in both Coq-to-OCaml and OCaml-to-Coq directions. However, we cannot combine these translations for bidirectional certified programming we aim for.

For Coq-to-OCaml, we could use the program extraction mechanism [Let08] provided by Coq. Many existing certified softwares such as CompCert C compiler [Ler09] were developed in this approach. For OCaml-to-Coq, we could use CFML [Cha11] or CoqOfOCaml [Cla14]. In CFML, all function definitions in OCaml are translated into Coq axioms, which are never translated into OCaml by the program extraction. In CoqOfOCaml, the type of functions may be changed through translating OCaml to Coq. For example, the translation of a recursive function adds a 'fuel' as an additional argument to define a corresponding terminating function because Coq does not accept definitions of non-terminating functions. As the result of the program extraction from the translated function, a function of a different type is obtained. Our system does not change the type of functions. The proof of its termination can be added after the translation if Coq fails to detect the evidence of the termination. Adding the proof will not affect the function definition in OCaml when reflecting the modification.

5 Conclusion

We proposed bidirectional certified programming which have advantages of both proof-and-extract and import-and-proof approaches, where a Coq script and an OCaml program are modified alternatingly. Every modification in one side is reflected into another side with preserving the original description as much as possible by combining four bidirectional transformations. This makes it easy to prove properties in Coq and to write programs in OCaml. Our system succeeded to translate more than half (27 out of 48) of function definitions in the `List` module provided in OCaml. Most of the failures are caused by lack of exception handling in Coq.

The current implementation still imposes restriction on the style of programming, e.g., data constructors in Coq should be capitalized, user-defined infix operators are prohibited. Although some of these restrictions come from the limitation of BiYacc, it will be soon resolved in near future.

Additionally, our system itself is not developed by certified programming, that is, the translation is not guaranteed to preserve the semantics. This problem is common with the program extraction of Coq [Let08] in the proof-and-extract approach and existing work [Cha11, Cla14] in the import-and-proof approach. We wish for BiGUL to have a facility for formally-specified semantic preserving translation because our bidirectional translation strongly relies on BiGUL. Or otherwise we should take an approach in a way similar to the success of CakeML [MO14] which achieves certified extraction from HOL.

Acknowledgments

The authors would like to thank anonymous reviewers for their many insightful suggestions and corrections. They are also indebted to Hsiang-Shang Ko, Zhenjiang Hu, and Mirai Ikebuchi for their helpful comments.

References

[BC10] Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions.* Springer Publishing Company, Incorporated, 1st edition, 2010.

[Cha11] Arthur Charguéraud. Characteristic Formulae for the Verification of Imperative Programs. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 418–430, New York, NY, USA, 2011. ACM.

[Cla14] Guillaume Claret. Coq of OCaml. In *Proceedings of the 2014 OCaml Workshop*, OCaml '14, 2014.

[Coq16] Coq Development Team. *The Coq Reference Manual*, version 8.6 edition, 2016. URL: `http://coq.inria.fr/`.

[FGM$^+$07] 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), May 2007.

[KZH16] 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.

[Ler09] Xavier Leroy. Formal Verification of a Realistic Compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[Let08] Pierre Letouzey. Extraction in Coq: An Overview. In *Proceedings of the 4th Conference on Computability in Europe: Logic and Theory of Algorithms*, CiE '08, pages 359–369, Berlin, Heidelberg, 2008. Springer-Verlag.

[MO14] Magnus O. Myreen and Scott Owens. Proof-producing translation of higher-order logic into pure and stateful ML. *J. Funct. Program.*, 24(2-3):284–315, 2014.

[ZZK$^+$16] 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.