Advanced program calculation using typeclass in Coq

Kosuke Murata[1] and Emoto Kento[1]

Kyushu Institute of Technology, Japan
murata@pl.ai.kyutech.ac.jp, emoto@ai.kyutech.ac.jp

Abstract. Program calculation, which is a programming technique to derive efficient but elusive programme from accessible but inefficient programme, is an important activity for developing program optimization.

Keywords: Program calculation · Functional programming · Coq · Interactive theorem proving.

1 Introduction

Simple programme tend to be easy to implement, but inefficient and useless. On the contrary, efficient programme are useful but they require difficult programming techniques for implementation. Making "program efficiency" and "easiness of implementation" compatible is important challenge for study of programming.

Program calculation [2, 3] is programming technique to derive highly technical and efficient programme from simple and inefficient programme by using rules for programme. Of course, preserving semantics of programme is essential for correctness of program calculations. In this paper, correctness of program calculation means preserving semantics of programme.

For certificating correctness of program calculation, Tesson, et al. [4] designed a tactic library for program calculation in Coq [], which is an interactive theorem prover. Their tactic library has provided tactic notations for write Coq scripts like chains of equality, which is common in program calculation. They also formalized theory of lists [2], which provides a calculation rules for list-function base on Bird-Meertence Formalism (BMF).

However, not only the list but also several algebraic datatypes (ADTs) such as natural numbers, binary trees and more general trees are essential for functional programming in practice. Fortunately, several studies describe calculation rules for arbitrary ADTs[]. In this paper, we call such rules "ADT-generic rules." ADT-generic rules are stated by using F-initial algebra and F-terminal coalgebra for modeling algebraic datatypes, and *recursion schemes* such as catamorphism, anamorphism and paramorphism for modeling programs which can be recursively defined on arbitrary ADTs.

The present paper aims to formalize ADT-generic rules described in Uustalu's paper [5], which describes basic ADT-generic rules for basic recursion schemes such as catamorphism and anamorphism, and for little advanced recursion schemes such as futumorphism and histmorphism.

In this study, we emphasize on using shallow embedding as much as possible. For proving theorems about programs by using interactive theorem provers, here are two styles to define semantics of domain languages: shallow embedding and deep embedding. The former is using the semantics of a host language as it as the semantics of a domain language, the latter is to construct another semantics of domain language on a host language. Deep embedding is flexible and has high power of expressions, but it tends to need high-cost proof. Unfortunately, depending on the way of constructing a semantics in the host language, it is difficult to get a "runnable" program. On the other hands, shallow embedding is less flexible, but the cost of proof tends to be low. Moreover, it is possible to make proof using a program that can be processed by a host language processing system, that is, it is actually runnable. From the perspective of program calculation in Coq, shallow embedding that uses Gallina, which is a core functional language in Coq, as a domain language makes it possible to safely derivation of runnable Gallina programs.

The organization of this paper is as follows. 後で書く.

2 Preliminaries

In this section, we introduce Coq and its tactic library for program calculation proposed by Tesson, et al. [4]. We also introduce notions in basic category theory for describing ADT-generic theorems.

2.1 Overview of Coq and an tactic library for program calculation

Coq [] is a popular interactive theorem prover. It based on the calculus of inductive constructions, which is a higher-order typed lambda calculus. It is widely used in computer science field, e.g. program verifications. A Coq script consists of definitions for data types, formula of propositions which we want to prove, and their proofs, and more. Coq proof are written using tactics, which are commands for construct proofs. Coq system has a sub-language Ltac for programming original tactics. Ltac provides **Tactic Notations** commands, which enable us to use various novel notations in Coq. For the details, see Coq reference manual [].

Tesson, et al. [4] designed and implemented a tactic library for program calculation in Coq using Ltac. Their tactic library has provided tactic notations which make it possible to write Coq script like equational reasoning, which is common notation in program calculation. In consideration of maintainability, we re-implemented our own tactic library using the same methods as theirs.

The our library mainly for equational reasoning provides the following three tactics:

Tactic 1. `Left =` ⟨*term*⟩ `{` ⟨*tactic*⟩ `}`
Tactic 2. `=` ⟨*term*⟩ `{` ⟨*tactic*⟩ `}`
Tactic 3. `= Right`

Fig. 1 shows the Coq script using our library, for a proof of

$$\forall\ (x\ y\ z : \texttt{nat}),\ x = y \to z = x + x \to x + y = z,$$

and progress of its interactive proof in Coq IDE. For a goal $t = s$, tactic 1 starts the proof by rewriting LHS t to the given *term* by using the specified tactic *tactic*, to produce the next goal *term* $= y$. This can be seen in the first step in Fig. 1, althought its specified *term* is exactly the same as the LHS of the goal and thus the goal is unchanged. Tactic 2 advance the chain in the similar way; see the 2nd through 4th steps in Fig. 1. Tactic 3 finished the chain. It directly proofs the current goal $s = s$ by using **reflexivity**. Note that there term "`d := direction Rightwards : Prop`" appearing in the Fig. 1 is used to memorize the direction of the chain growth.

To make script more shord and more readable, we implement short hands of these tactics. For instance,

- Tactic 2′. `=` ⟨*term*⟩

which is equal to `=` ⟨*term* `{ easy }`, and

- Tactic 2″. `=` ⟨*term*⟩ `{ by` ⟨*term*⟩ `}`

which is equal to `=` ⟨*term*⟩ `{ rewrite` ⟨*term*⟩ `+ rewrite <-` ⟨*term*⟩ `}`.

In this paper, in order to improve readability, we write Coq scripts using roman and san serif fonts such as mathematical expressions, instead of typewriter fonts.

2.2 Basic notions of category theory

We use basic notions of category theory for describing ADT-generic theorems. We work on the category **Set**, which is category of set and functions.

Given two sets A, B, we write $A{*}B$ for product (just like `Coq.Init.Datatypes`, the Coq standard library), and $A + B$ for sum. We write $\texttt{fst} : A * B \to A$ and $\texttt{snd} : A * B \to B$ for the left and right projection function, $\texttt{inl} : A \to A + B$ and $\texttt{inr} : B \to A + B$ for the constructors for sum.

We use also following notations, which are non-standard in Coq but familiar in program calculation, just like [5]. For $f : A \to B$ and $g : A \to C$, we use $\langle f, g\rangle : A \to B * C$ for unique morphism h such that $\texttt{fst} \circ h = f$ and $\texttt{snd} \circ h = g$. For $f : A \to C$ and $g : B \to C$, we use $[f, g] : A + B \to C$ for unique morphism h such that $h \circ \texttt{inl} = f$ and $h \circ \texttt{inr} = g$. Fig. 2. shows the definitions for such morphisms in Coq.

We use traditional way for modeling datatypes, which use F-initial algebras and terminal coalgebras where $F : \mathbf{Set} \to \mathbf{Set}$ is a polynomial functor. For a polynomial functor $F : \mathbf{Set} \to \mathbf{Set}$, we write $(\mu F, \mathsf{in}_F)$ for the F-initial algebra, $(\nu F, \mathsf{out}_F)$ for the F-terminal coalgebra. Note that for any polynomial functor $F : \mathbf{Set} \to \mathbf{Set}$, there exists unique a F-initial algebra and a F-terminal coalgebra up to isomorphism. We also use polynomial functors indexed by constant set

Step	Script buffers	Gola window
0	intros x y z H0 H1. Left = (x + y) { reflexivity }. = (x + x) { rewrite <- H0 }. = z { rewrite H1 }. = Right.	============================ forall x y z : nat, x = y -> z = x + x -> x + y =
1	intros x y z H0 H1. Left = (x + y) { reflexivity }. = (x + x) { rewrite <- H0 }. = z { rewrite H1 }. = Right.	x, y, z : nat H0 : x = y H1 : z = x + x ============================ x + y = z
2	intros x y z H0 H1. Left = (x + y) { reflexivity }. = (x + x) { rewrite <- H0 }. = z { rewrite H1 }. = Right.	x, y, z : nat H0 : x = y H1 : z = x + x d := direction Rightwards : Prop ============================ x + y = z
3	intros x y z H0 H1. Left = (x + y) { reflexivity }. = (x + x) { rewrite <- H0 }. = z { rewrite H1 }. = Right.	x, y, z : nat H0 : x = y H1 : z = x + x d := direction Rightwards : Prop ============================ x + x = z
4	intros x y z H0 H1. Left = (x + y) { reflexivity }. = (x + x) { rewrite <- H0 }. = z { rewrite H1 }. = Right.	x, y, z : nat H0 : x = y H1 : z = x + x d := direction Rightwards : Prop ============================ z = z
5	intros x y z H0 H1. Left = (x + y) { reflexivity }. = (x + x) { rewrite <- H0 }. = z { rewrite H1 }. = Right.	

Fig. 1. Progress of interactive proof on Coq IDE

A; e.g. $F_A : \mathbf{Set} \to \mathbf{Set}$ such that $F_A(X) = \mathbf{1} + A * X$. From the aspect of formilization, indexed functors just like $F_A(X)$ can be assumed that is partial application of bifunctor $F : \mathbf{Set} \to \mathbf{Set} \to \mathbf{Set}$.

Let $F\colon \mathbf{Set} \to \mathbf{Set}$ be a polynomial functor, and $(\mu F, \mathsf{in}_F)$ be an initial F-algebra. Given a morphism $\varphi\colon F\,X \to X$, the morphism $(\!|\varphi|\!)\colon F\,\mu F \to \mu F$ is *catamorphism*[1] if following diagram commutes:

$$\begin{array}{ccc} F\,\mu F & \xrightarrow{\mathsf{in}_F} & \mu F \\ {\scriptstyle F\,(\!|\varphi|\!)}\downarrow & & \downarrow{\scriptstyle (\!|\varphi|\!)} \\ F\,X & \xrightarrow[\varphi]{} & X \end{array}.$$

In other words, $(\!|f|\!)$ is characterized by the universal property:

$$f \circ \mathsf{in}_F = \varphi \circ F\,f \iff f = (\!|\varphi|\!).$$

Anamorphism とかについても後で書く.

[1] The term 'catamorphism' is derived from the greek preposition $\kappa\alpha\tau\alpha$ meaning 'downwards'.

```
Definition fprod {A B C : Type} (f : A → B) (g : A → C) : A → B * C :=
  λ (x : A) ⇒ (f x, g x).
Definition fsum {A B C : Type} (f : A → B) (g : A → C) : A + B → C :=
  λ (x : A + B) ⇒ match x with
                  | inl x ⇒ f x
                  | inr x ⇒ g x
                  end.
Notation "⟨f, g⟩" := (fprod f g).
Notation "[f, g]" := (fsum f g).
```

Fig. 2. The definitions for product and sum of morphisms in Coq

$$\begin{array}{ll} \mathtt{fst} \circ \langle f, g\rangle = f & [f, g] \circ \mathtt{inl} = f \\ \mathtt{snd} \circ \langle f, g\rangle = g & [f, g] \circ \mathtt{inr} = g \\ \langle \mathtt{fst}, \mathtt{snd}\rangle = id & [\mathtt{inl}, \mathtt{inr}] = id \end{array}$$

Fig. 3. Rules for product and coproduct

3 Modeling Datatypes in Coq

In this section, we propose a way to models concepts to model datatypes in Coq. Types and functions in Coq can be regarded as object and morphisms in category **Set**.

3.1 Polynomial Functor

Fig. 4. shows the definitions for polynomial functor in Coq. Functor consists of following two mappings: mappings from object to object, and mappings from morphisms to morphisms. To define polynomial functor inductively, we first define `PolyF`, which is the type for AST (Abstract Syntax Tree) of polynomial functor, and then use it to define following two mappings:

- $[\![F]\!] : \mathsf{Type} \to \mathsf{Type} \to \mathsf{Type}$ (`inst` F) is to map from types to types.
- $F(-)[-]$: $(A_0 \to A_1) \to (X_0 \to X_1) \to [\![F]\!]\ A_0\ X_0 \to [\![F]\!]\ A_1\ X_1$(`fmap`) is mapping from morphisms to morphisms.

These definitions are for bifunctor, but we can get definitions for functor by partial applications, i.e. $[\![F]\!]\ A$ and F[@id A](−) define a functor (indexing by type A).

Polynomial functors must satisfy the functor laws, so we proved following 2 lemmas for functor laws by using easy induction on F.

- **Lemma** `fmap_functor_distr` :
  $\forall\ (F : \mathtt{PolyF})\ \{A\ X_0\ X_1\ X_2 : \mathsf{Type}\}\ (g_0 : X_0 \to X_1)\ (g_1 : X_1 \to X_2)$,
  $F(@\mathrm{id}\ A)[g_1 \circ g_0] = F[g_1] \circ F[g_0]$

```
Inductive PolyF : Type :=
| zer  : PolyF
| one  : PolyF
| arg1 : PolyF
| arg2 : PolyF
| Sum  : PolyF → PolyF → PolyF
| Prod : PolyF → PolyF → PolyF.

Inductive inst (F : PolyF) (A X : Type) : Type :=
 match F with
 | zer       ⇒ Empty_set
 | one       ⇒ Unit
 | arg1      ⇒ A
 | arg2      ⇒ X
 | Sum F G   ⇒ (inst F A X) + (inst G A X)
 | Prod F G ⇒ (inst F A X) * (inst G A X)
 end.

Notation "⟦F⟧" := (inst F).

Fixpoint fmap (F : PolyF) {A0 A1 X0 X1 : Type}
    (f : A0 → A1) (g : X0 → X − 1) : ⟦F⟧ A0 X0 → ⟦F⟧ A1 X1
    := match F with
        | zer       ⇒ id
        | one       ⇒ id
        | arg1      ⇒ f
        | arg2      ⇒ g
        | Sum F G   ⇒ λx ⇒ match x with
                           inl x ⇒ inl (fmap F f g x)
                           inr x ⇒ inr (fmap F f g x)
                           end
        | Prod F G ⇒ λx ⇒ (fmap f g (fst x), fmap G f g x (snd x))
        end.

Notation "F[f]" := (@fmap F _ _ _ _ id f) (at level 10).
```

Fig. 4. The definitions for polynomial functor in Coq

```
Class F_initial_algebra (F : PolyF) (A : Type) (μF : Type)
      (c : ∀ (X : Type), ([[F]] A X → X) → (μF → X)) :=
 {
   cata := c;
   in_         : [[F]] A μF → μF;
   cata_charn : ∀ (X : Type) (f : μF → X) (φ : [[F]] A X → X),
                  f ∘ in_ = φ ∘ F[f] ↔ f = cata X φ
 }.

Notation "(|f|)" := (cata _ _ f) (at level 5).
```

Fig. 5. Definitions for F-initial algebras using type class in Coq

- **Lemma** `fmap_functor_id` :
  $\forall\ (F : \texttt{PolyF})\ \{A\ X : \texttt{Type}\}\ (g_0 : X_0 \to X_1)\ (g_1 : X_1 \to X_2)$,
  $F(@\mathrm{id}\ A)[@\mathrm{id}\ X] = \mathrm{id}$

Next, we define F-initial algebras using type class. Fig. 5. shows the definitions for F-initial algebras using type class in Coq.

Initial algebras and catamorphisms satisfy the following 3 properties, where

```
Variable (F : PolyF) (A : Type) (μF : Type)
         (c : ∀ (X : Type), ([[F]] A X → X) → (μF → X))
         (ia : F_initial_algebra F A μF c).
```

- **Proposition** `cata_cancel` :
  $\forall\ (X\ Y : \texttt{Type})\ (\varphi : [\![F]\!]\ A\ X \to X)$,
  $(\!|\varphi|\!) \circ \texttt{in\_} = \varphi \circ F[(\!|\varphi|\!)]$.
- **Proposition** `cata_refl` : $(\!|\texttt{in\_}|\!) = \mathrm{id}$
- **Proposition** `cata_fusion` :
  $\forall\ (X\ Y : \texttt{Type})\ (\varphi : [\![F]\!]\ A\ X \to X)\ (\psi : [\![F]\!]\ A\ Y \to Y)\ (f : X \to Y)$,
  $f \circ \varphi = \psi \circ F[f] \to f \circ (\!|\varphi|\!) = (\!|\psi|\!)$.

Fig. 6 shows the Coq proof of `cata_fusion`.

3.2 Instanciation of the Class for initial algebra

As a benefit of using type class for definition for initial algebra, it is possible to use the theorems for runnable Coq programs.

First, we give an example of making `nat` an instance of `F_initial_algebra`, where `nat` is standard type of natural numbers in Coq. The catamorphism of nat can define as follows:

```
Definition cata_nat (X : Type) (f : unit + X → X) : nat → X :=
  fix cataf (n : nat) := match n with
                         | O    ⇒ f (inl ())
                         | S n' ⇒ f (inr (cataf n'))
                         end.
```

Proposition `cata_fusion` :
$\forall\ (X\ Y : \mathsf{Type})\ (\varphi : [\![F]\!]\ A\ X \to X)\ (\psi : [\![F]\!]\ A\ Y \to Y)\ (f : X \to Y)$,
$f \circ \varphi = \psi \circ F[f] \to f \circ (\!|\varphi|\!) = (\!|\psi|\!)$.
Proof.
`intros; apply cata_charn`.
Left
$= (f \circ ((\!|\varphi|\!) \circ \texttt{in\_}))$.
$= (f \circ \varphi \circ F[(\!|\varphi|\!)])$ {**by** `cata_cancel`}.
$= (\psi \circ (F[f] \circ F[(\!|\varphi|\!)]))$ {**by** H}.
$= (\psi \circ F[f \circ (\!|\varphi|\!)])$ {**by** `fmap_functor_distr`}.
$=$ **Right**.
Qed.

Fig. 6. A coq proof for catamorphism fusion-law, using tactic libraries

Now, we can define `Nat_IA`, which is an instance of `F_initial_algebra`, as follows.

Instance `Nat_IA` :
`F_initial_algebra (Sum one arg2) unit cata cata_nat` :=
{
`in_` := $[\lambda\ x \Rightarrow$ `O, S`$]$;
}.

There are incomplete proofs of `cata_charn`, but the proof is easy to complete (see Appendix). `Nat_IA` make us possible to evaluate catamorphisms in `nat`, therefore, if Coq system reads following command:

$$\textbf{Eval}\ \text{cbv in}\ (\lambda(y : \texttt{nat}) \Rightarrow (\!|[\lambda\_ \Rightarrow y, S]|\!))\ 100\ 11.$$

it will return the answer, 111.

4 More advanced recursion schemes

5 Related Work

Furthermore, some research[] uses interactive theorem prover such as Coq and Agda.

6 Conclusion and Future Work

References

1. Bird, R.: An introduction to the theory of lists. In: Bory, M. (ed.) Logic of Programming and Calculi of Discrete Design. pp. 5–42. Springer, Heidelberg (1987)

2. Bird, R.: Lectures on constructive functional programming. In: Broy, M. (ed.) Constructive Methods in Computer Science. pp. 151–218. Springer-Verlag (1988), nATO ASI Series F Volume 55. Also available as Technical Monograph PRG-69, from the Programming Research Group, Oxford University
3. Bird, R., de Moor, O.: Algebra of Programming. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1997)
4. Tesson, J., Hashimoto, H., Hu, Z., Loulergue, F., Takeichi, M.: Program Calculation in Coq. In: Proceedings of the 13th International Conference on Algebraic Methodology and Software Technology. pp. 163–179. AMAST'10, Springer-Verlag, Berlin, Heidelberg (2011)
5. Uustalu, T., Vene, V.: Primitive (co)recursion and course-of-value (co)iteration, categorically. Informatica **10**, 5–26 (1999)

A Coq Proof for Instantiation of Nat

```
Instance Nat_ia :
  F_initial_algebra (Sum one arg2) unit nat cata_nat :=
  {
    in_  := [ fun x => O , S ];
  }.
Proof.
  intros X f φ.
  split.
  - intros H. unfold cata_nat.
    extensionality x.
    induction x.
    + specialize (equal_f H (inl tt)) as H0; cbv in H0.
      exact H0.
    + specialize (equal_f H (inr x)) as H1; cbv in H1.
      rewrite <- IHx.
      exact H1.
  - intros H; extensionality x; induction x.
    + rewrite H. induction a. easy.
    + rewrite H; easy.
Qed.
```