Program Calculation in Coq

Julien Tesson[1], Hideki Hashimoto[2], Zhenjiang Hu[3], Frédéric Loulergue[1], and Masato Takeichi[2]

[1] Université d'Orléans, LIFO, France
{julien.tesson,frederic.loulergue}@univ-orleans.fr
[2] The University of Tokyo, Japan
{hhashimoto,takeichi}@ipl.t.u-tokyo.ac.jp
[3] National Institute of Informatics, Tokyo, Japan
hu@nii.ac.jp

Abstract. Program calculation, being a programming technique that derives programs from specification by means of formula manipulation, is a challenging activity. It requires human insights and creativity, and needs systems to help human to focus on clever parts of the derivation by automating tedious ones and verifying correctness of transformations. Different from many existing systems, we show in this paper that Coq, a popular theorem prover, provides a cheap way to implement a powerful system to support program calculation, which has not been recognized so far. We design and implement a set of tactics for the Coq proof assistant to help the user to derive programs by program calculation and to write proofs in calculational form. The use of these tactics is demonstrated through program calculations in Coq based on the theory of lists.

1 Introduction

Programming is the art of designing efficient programs that meet their specifications. There are two approaches. The first approach consists of constructing a program and then proving that the program meets its specification. However, the verification of a (big) program is rather difficult and often neglected by many programmers in practice. The second approach is to construct a program and its correctness proof hand in hand, therefore making a posteriori program verification unnecessary.

Program calculation [1–3], following the second approach, is a style of programming technique that derives programs from specifications by means of formula manipulation: calculations that lead to the program are carried out in small steps so that each individual step is easily verified. More concretely, in program calculation, specification could be a program that straightforwardly solves the problem, and it is rewritten into a more and more efficient one without changing the meaning by application of calculation rules (theorems). If the program before transformation is correct, then the one after transformation is guaranteed to be correct because the meaning of the program is preserved by the transformation.

Bird-Meertens Formalism (BMF) [1,4], proposed in late 1980s, is a very useful program calculus for representing (functional) programs, manipulating programs

through equational reasoning, and constructing calculation rules (theorems). Not only many general theories such as the theory of list [4] and the theory of trees [5] have been proposed, but also a lot of useful specific theories have been developed for dynamic programming [6], parallelization [7], etc.

Program calculation with BMF, however, is not mechanical: it is a challenging activity that requires creativity. As a simple example, consider that we want to develop a program that computes the maximum value from a list of numbers and suppose that we have had an insert-sorting program to sort a list. Then a straightforward solution to the problem is to sort a list in descending order and then get the first element:

$$maximum = hd \circ sort.$$

However, it is not efficient because it takes at least the time of *sort*. Indeed, we can calculate a linear program from this solution by induction on the input list.

If the input is a singleton list $[a]$, we have

$$\begin{array}{ll} & maximum\ [a] \\ = & \quad \{ \text{ def. of maximum } \} \\ & (hd \circ sort)\ [a] \\ = & \quad \{ \text{ def. of function composition } \} \\ & hd\ (sort\ [a]) \\ = & \quad \{ \text{ def. of sort } \} \\ & hd\ [a] \\ = & \quad \{ \text{ def. of hd } \} \\ & a \end{array}$$

Otherwise the input is a longer list of the form $a :: x$ whose head element is a and tail part is x, and we have

$$\begin{array}{ll} & maximum\ (a :: x) \\ = & \quad \{ \text{ def. of maximum } \} \\ & hd\ (sort\ (a :: x)) \\ = & \quad \{ \text{ def. of sort } \} \\ & hd\ (\text{if } a > hd(sort\ x) \text{ then } a :: sort\ x \\ & \qquad \text{else } hd(sort\ x) :: insert\ a\ (tail(sort\ x))) \\ = & \quad \{ \text{ by if law } \} \\ & \text{if } a > hd(sort\ x) \text{ then } hd(a : sort\ x) \\ & \text{else } hd(hd(sort\ x) :: insert\ a\ (tail(sort\ x))) \\ = & \quad \{ \text{ def. of hd } \} \\ & \text{if } a > hd(sort\ x) \text{ then } a \text{ else } hd(sort\ x) \\ = & \quad \{ \text{ def. of maximum } \} \\ & \text{if } a > maximum\ x \text{ then } a \text{ else } maximum\ x \\ = & \quad \{ \text{ define } x \uparrow y = \text{if } x > y \text{ then } x \text{ else } y \ \} \\ & a \uparrow (maximum\ x) \end{array}$$

Consequently we derive the following linear program:

$$\begin{array}{ll} maximum\ [a] & = a \\ maximum\ (a :: x) & = a \uparrow (maximum\ x) \end{array}$$

In this derivation, we transform the program by equational reasoning via unfolding definition of functions and applying some existing calculation laws (rules). Sometimes, we even need to develop new calculations to capture important transformation steps. This calls for an environment, and much effort has been devoted to development of systems to support correct and productive program calculation. Examples are KIDS [8], MAG [9], Yicho [10], and so on. In general, this kind of environments should (1) support interactive development of programs by equational reasoning so that users can focus on his/her creative steps, (2) guarantee correctness of the derived program by automatically verifying each calculation step, (3) support development of new calculation rules so that mechanical derivation steps can be easily packed, and (4) make development process easy to maintain (i.e., development process should be well documented.). In fact, developing such a system from scratch is hard and time-consuming, and there are few systems that are really widely used.

The purpose of this paper is to show that Coq [11], a popular theorem prover, provides a *cheap* way to implement a *powerful* system for program calculation, which has not been recognized so far. Coq is an interactive proof assistant for the development of mathematical theories and formally certified software. It is based on a theory called the calculus of inductive constructions, a variant of type theory. Appendix A provides a very short introduction to Coq. Although little attention has been paid on using Coq for program calculation, Coq itself is indeed a very powerful tool for program development. First, we can use dependent types to describe specifications in different levels. For instance, we can write the specification for *sort* by

$$sort : \forall x : list\ nat, \exists y : list\ nat, (sorted(y) \wedge permutation(x, y))$$

saying that for any x, a list of natural numbers, there exists a sorted list y that is a permutation of x, and we, who want to use *sort*, would write the following specification for *maximum*:

$$maximum : \exists \oplus : nat \to nat \to nat, hd \circ sort = foldr1\ (\oplus)$$

saying that the straightforward solution $hd \circ sort$ can be transformed into a *foldr1* program. Note that *foldr1* is a similar higher order function to *foldr*, but it is applied to nonempty lists. Second, one can use Coq to describe rules for equational reasoning with dependent types again. Here are two simple calculation rules, associativity of the append operation, and distributivity of the *map* functions.

Lemma $appAssoc : \forall (A : Type)\ (l\ m\ n : list\ A),$
$(l +\!\!+ m) +\!\!+ n = l +\!\!+ (m +\!\!+ n)$

Lemma $mapDist : \forall (A\ B\ C : Type)\ (f : B \to C)\ (g : A \to B),$
$map\ f \circ map\ g = map\ (f \circ g)$

Third, one can use Coq to prove theorems and extract programs from the proofs. For example, one can prove the specification for *maximum* in Coq. The proof

script, however, is usually difficult to read compared with the calculation previously introduced. This is one of the main problem of using Coq for program calculation.

In this paper, we shall report our first attempt of designing and implementing a Coq tactic library (of only about 200 lines of tactic codes), with which one can perform correct program calculations in Coq, utilize all the theories in Coq for his calculation, and develop new calculation rules, laws and theorems. Section 2 shows an example of the calculation for maximum in Coq. A more interesting use of these tactics are demonstrated by an implementation of the Bird's calculational theory of lists in Coq. All the codes of the library and applications (as well as the *maximum* example with a classic proof script) are available at the following web page: `https://traclifo.univ-orleans.fr/SDPP`.

The organization of the paper is as follows. First we discuss the design and implementation of a set of tactics for supporting program calculation and writing proofs in calculational form in Section 2. Then, we demonstrate an interesting application of calculation in Section 3. Finally, we discuss the related work in Section 4 and conclude the paper in Section 5.

2 Coq Tactics for Program Calculation

This section starts with an overview of tactics we provide and with a demonstration of how they are used in calculation; it is followed by details about the implementation of the tactics in Coq.

2.1 Overview of available tactics

We provide a set of tactics to perform program calculation in Coq. We can use it in two ways: either we want to transform a program but we don't know what the final result will be; or we want to prove a program equivalent to another program.

Let's take the example of maximum [4] presented in introduction. We will first illustrate the case in which we don't know the final result with the singleton list case; then we will illustrate a calculation which just proves the equality between two known programs with the case in which the list has the form a::x.

In the case of a singleton list, we want a function f such that ∀a, maximum d [a] = f a; this is expressed by the type {f | ∀a d, maximum d [a] = f a} of maximum_singleton in what is following.

```
Definition maximum_singleton : {f | ∀a d, maximum d [a] = f a}.
Begin.
  LHS
= { by def maximum}
  (hd d (sort [a]) ).
```

[4] maximum d l *is defined by* hd d (sort l) *where* hd d l *is the function returning the head of the list* l, *or* d *if* l *is an empty list.*

```
= { by def sort; simpl_if }
  (hd d [a]).
= { by def hd }
  a.
[].
Defined.
```

The Begin. tactic starts the session by doing some technical preparation of the Coq system which will be detailed later (sect. 2.3).

Thus then the user specifies by the LHS tactic that he wants to transform the Left Hand Side of the equation maximum d [a] = f a. If he had wanted to transform the right hand side of the equation he would have used the RHS tactic, or BOTH_SIDE tactic to transform the whole equation.

By using

```
= { by def maximum}
  (hd d (sort [a]) ).
```

the user specifies that the left hand side should be replaced by hd d (sort [a]), and that this transformation is correct by the definition of maximum.

For the next transformation, the equality between the term hd d (sort [a]) and the term hd d [a] cannot be proved by the only definition of sort: it also needs some properties over the relation "greater than" used in the definition of sort. The user-defined tactic simpl_if which, in a sense, helps to determinate the value of sort [a], is necessary. Actually, we can place here any necessary tactic sequence to prove the equality.

Once we achieve a satisfying form for our program, we can use []. to end the transformation.

In case the list has the form a::x, we want to prove that maximum d (a::x) is equal to **if** a ?> (maximum a x) **then** a **else** maximum a x[5].

```
Lemma maximum_over_list : ∀a x d,
maximum d (a::x) = if a?> (maximum a x) then a else maximum a x.
  Begin.
    LHS
  = { by def maximum}
    (hd d (sort (a::x)) ).
  = { unfold_sort }
    ( hd d ( let x':=(sort x) in if a ?> (hd a x') then a :: x'
      else (hd a x')):: (insert a (tail x')) ) ).
  ={ rewrite (if_law _ (hd d)) }
   (let x':= (sort x) in
     if a ?> hd a x' then hd d (a :: x')
     else hd d (hd a x' :: insert a (tail x'))) .
  ={ by def hd; simpl_if }
    (let x' := sort l in
```

[5] In Coq, > is defined as a relation, thus we use ?> which is a boolean function

```
      if a ?> hd a x' then a else hd a x') .
  ={ by def maximum}
    (if a?> (maximum a x) then a else maximum a x).
[].
Qed.
```

As previously we use Begin. to start the session. Then, left hand side of the equation is transformed using the definitions of programs and a program transformation law, if_law. This law states that for any function f, f applied to an **if** C **then** g1 **else** g2 statement is equal to the statement **if** C **then** f g1 **else** f g2.

[] ends the proof if the two terms of the equation are convertible.

To get the full code for the linear version of maximum, we have to manually pose a new lemma stating that linear_maximum[6] is equal to maximum. This can be proved easily using previous lemmas.

2.2 More advanced use

For the previous example we use the Coq equality but we can also use the system with a user-defined equality. For doing so we use the Setoid module from the standard library which allows to declare an equivalence relation. Let us take the *extensional equivalence* relation between function defined by

```
Definition ext_eq A B (f : A →B) (g : A →B) := ∀a, f a = g a.
```

Theorems ext_eq_refl, ext_eq_sym and ext_eq_trans which state that ext_eq is respectively reflexive, symmetric and transitive can be proved easily. With Setoid we can declare extensional equality as a user-defined equality by the following code:

```
Add Parametric Relation (A B:Type) : (A→B) (@ext_eq A B )
  reflexivity proved by (@ext_eq_refl A B)
 symmetry proved by (@ext_eq_sym A B)
 transitivity proved by (@ext_eq_trans A B)
 as ExtEq.
```

Afterward, we will denote @ext_eq A B f1 f2 by f1 == f2, the arguments A and B being implicit.

Once we have declared our relation, it can automatically be used by tactics like reflexivity, symmetry, transitivity or rewrite which are normally used with Coq equality.

However, if we know that f1==f2, then for any F, and we want to rewrite F f1 into F f2 with rewrite tactic, we need to declare F as a morphism for ==. For example the function composition

```
Definition comp (A B C : Type) (f : B→C) (g : A →B) := fun (x:A) ⇒f (g x),
```

is a morphism for == for its two arguments f and g. This means that

[6] linear_maximum d l :=
match l **with** | nil ⇒d | a::l' ⇒**if** a?> (linear_maximum a l') **then** a **else** linear_maximum a l' **end**.

∀f1 f2 g, f1 == f2 →comp f1 g == comp f2 g

and that

∀f1 f2 g, f1 == f2 →comp g f1 == comp g f2.

This is implemented by:

```
Add Parametric Morphism A B C : (@comp A B C ) with
  signature (@extensional_equality B C) ==> eq ==>(@extensional_equality A C )
     as compMorphism.
Proof.
   ...
Qed.

Add Parametric Morphism A B C :( @comp A B C ) with
  signature (eq) ==> (@extensional_equality A B ) ==> (@extensional_equality A C )
     as compMorphism2.
Proof.
  ...
Qed.
```

And here is an example of use:

```
Lemma assoc_rewriting : ∀(A : Type) (f : A→A) (g : A→A),
((map f):o:(map f):o:(map g):o:(map g)) == (map(f:o:f):o:map (g:o:g)).
Begin.
   LHS
={ rewrite comp_assoc}
   ( (map f :o: map f ) :o: map g :o: map g ) .
={rewrite (map_map_fusion f f) }
   ( map (f:o: f) :o: map g :o: map g).
={rewrite (map_map_fusion g g) }
   ( map (f:o: f) :o: map (g :o: g) ).
[].
Qed.
```

The :o: is a notation for the function composition, comp_assoc states that function composition is associative and map_map_fusion states that

map f :o: map g == map (f :o: g).

We can see here that once the relation and the morphism are declared we can use the tactics as if it was the Leibniz equality.

2.3 Behind the scene

The Coq system allows the definition of syntax extensions for tactics, using **Tactic Notation**. These extensions associate an interleaving of new syntactic elements and formal parameters (tactics or terms) to a potentially complex sequence of tactics.

For example, ={ta} t1. is defined by

Tactic Notation (at level 2) "=" "{" tactic(t) "}" constr(e) := ... .

where tactic(t) specify that the formal parameter t is a tactic and constr(e) specify that the formal parameter e has to be interpreted as a term.

Our notation ={ta} t1., asserts that the term t1 is equivalent to the goal (or part of the goal) and proves it using the tactic ta (followed by reflexivity). Then, the goal is rewritten according to the newly proved equivalence.

The Begin. tactic introduces all premises and then inspects the goal. If the goal is existentially quantified, we use the eexists tactic which allows to delay the instantiation of existentially quantified variables. This delaying permits to transform the goal until we know what value should have this variable.

The LHS and RHS tactics first verify that the form of the goal is R a b and that R is a registered equivalence relation. Then they memorize a state used by our notations to know on which part of the goal they must work.

As the term at the right or left hand side of the equivalence can be fully contained in the other side of the equation, we have to protect it so that it remain untouched when rewriting is done by the transformation tactics on the opposite side. To protect it, we use the tactic set which replaces a given term (here the side we want to protect) by a variable and adds a binding of this variable with the given term in the context.

When registering the relation ext_eq as ExtEq, ExtEq is proved to be an instance of the type class Equivalence ex_eq. Type classes are a mechanism for having functions overloading in functional languages and are widely used in the module Setoid. They are defined in details in [12]; here, we only need to know that if there is a declared instance of the class Equivalence with parameter R, Equivalence R can be proved only by using the tactic typeclasses eauto. We use this to check that the goal has the right form so that we can be sure that our transformations produce an equivalent program.

Memorisation mechanism. As seen above, our tactics need to memorize informations, but coq does not provide a mechanism to memorize informations between tactic applications. So we introduce a dependent type which carries the informations we want to save. This type memo, with a unique constructor mem, is defined by

Inductive memo (s: state) : Prop := mem : memo s .,

state being an inductive type defining the informations we want to memorize.

The memorization of a state s can now be done by posing mem _ : memo s. We define a shortcut

Tactic Notation "memorize" constr(s) := pose (mem _ : memo s).

which abstracts the memorization mechanism. To access to the memorized informations, we use pattern matching over hypothesis.

The main limitation of this mechanism is that we have no way to memorize informations from one (sub-)goal to another. Indeed our mechanism memorize

informations by adding a variable to the context but the life-time of interactively added variables is limited to current (sub-)goal.

Until now this limitation has not been problematic for our system, but if we want to overcome this later, we would have to develop our system as a coq plugin.

3 Application: BMF in Coq

In this section, we demonstrate the power and usefulness of our Coq tactics library for program calculation through a complete encoding of the lecture note on theory of lists (i.e., Bird-Meertens Formalisms for program calculation, or BMF for short) [4], so that all the definitions, theorems, and calculations in the lecture note can be checked by Coq and guaranteed to be correct. Our encoding[7], about 4000 lines of Coq codes (using our library), contains about 70 definitions (functions and properties) and about 200 lemmas and theorems.

In our encoding, we need to pay much attention when doing calculation in Coq. First, we have to translate partial functions into total ones because Coq can only deal with total functions. Second, we should explore different views of lists, being capable of treating lists as snoc lists or join lists, while implementing them upon the standard cons lists. Third, we should be able to code restrictions on functions that are to be manipulated.

In the following, we give some simple examples to show the flavor of our encoding, before explaining how to deal with the above issues.

3.1 Examples

To give a flavor of how we encode BMF in Coq, let us see some examples. Using Coq, we do not need to introduce a new abstract syntax to define functions. Rather we introduce new notations or define functions directly in Coq syntaxe. For example, the following defines the filter function (and its notation) for keeping the list elements that satisfy a condition and the concat function for flattening a list of lists.

```
Fixpoint filter (A : Type)
      (p : A→bool) (l:list A) : list A :=
  match l with
    | nil ⇒nil
    | x :: l ⇒
      if p x then x :: (filter l) else filter l
  end.
```

```
Fixpoint concat (A : Type)
           (xs : list (list A)) : list A :=
  match xs with
    | nil ⇒nil
    | x :: xs' ⇒app x (concat xs')
  end.
```

```
Notation "p␣<|" := (filter p)(at level 20) : BMF_scope.
```

[7] The code is available at our project page.

And the following shows how to prove the filter promotion rule [4] using our Coq tactics.

```
Theorem filter_promotion :
  p <| :o: @concat A = @concat A :o: p <| *.
Proof.
    LHS
={ rewrite (filter_mapreduce) }
   (++ / :o: f * :o: @concat A).
={ rewrite map_promotion }
   (++ /:o: @concat (list A) :o: f* *).
={ rewrite comp_assoc }
   ((++ /:o: @concat (list A)) :o: f* *).
={ rewrite reduce_promotion }
   (++ / :o: (++ /)* :o: f* *).
={ rewrite concat_reduce }
   (@concat A :o: (++ /)* :o: f* *).
={ rewrite map_distr_comp }
   (@concat A :o: (++ / :o: f *) *).
={ rewrite filter_mapreduce }
   (@concat A :o: (p <|) *).
[].
Qed.
```

The derivation starts with the left hand side of the equation to the proved, and repeatedly rewrites it with rules until it is equivalent to the right hand side of the equation. For instance, the first derivation step rewrite the filter `p<|` to a composition of a map `f*` and a reduce (specific fold) `++ /` using the rule filter_mapreduce. This derivation relies on lemmas using the functional extensionality axiom, therefore we can use Liebniz equality.

In the following, we discuss how we deal with partial functions, different views of lists, and properties imposed on functions.

3.2 Implementation of partial functions

In Coq, all functions must be total and should terminate in order to keep the consistency of the proofs. However, there are many cases in which we want to use partial functions in programming. For example, the function that takes a list and returns the first element of the list is a partial function, because the function can not return any value if the list was empty. To implement this kind of functions as total functions, we may add a "default value", so that if the input is out of the domain, then, the function will return the default value. For example, the "head" function is implemented as follows:

```
Variable A : Type.
Fixpoint head (d : A) (x : list A) : A :=
  match x with
```

```
    | nil ⇒d
    | a :: x' ⇒a
  end.
End head.
```

Actually, this definition is the same as the hd one in the standard Coq library. Compared to the "paper" or functional programming language versions, we have to deal with this extra parameter.

There are several other ways to model a partially defined function in Coq. Another approach is to define a function that returns an optional value, ie a value of the type:

```
Inductive option (A:Set) : Set :=
 | Some: A →option A
 | None: option A.
```

A third possibility is to take as extra argument a proof that the list is non empty. Compared to the first solution, this parameter would be removed during the code extraction leading to a more usual function in a functional programming language. However in some cases having proofs as arguments is difficult to handle without the axiom of proof irrevelance. A fourth solution is to create a new type of non-empty lists, an element of this type being a list together with a proof that this list is not nil. Here again in some cases the development may become difficult without the axiom of proof irrevelance. In the example we chose the first solution.

3.3 Exploiting different views of a data type

There are different views of lists in BMF. We usually define a list as an empty list or a constructor adding an element to a list (cons a l). This captures the view that a list is constructed by adding elements successively to the front of a list (we will call this view "cons list"). However, there are other views of lists. For example, the "snoc list" is a view that a list is constructed by adding an element one by one to the end of an empty list, and the "join list" is another view that a list is constructed by concatenation of two shorter lists.

Adopting different views would make it easy and natural to define functions or to prove theorems on lists, so we exploit these views based on the cons list by implementing functions (or properties) on snoc lists or join lists based on those on cons lists as follows.

```
snoc_ind :
  ∀(A : Type) (P : list A →Prop),
  P nil →
  (∀ (x : A) (l : list A),
   P l →P (l ++ [x])) →
  ∀l : list A, P l
```

```
join_induction :
  ∀(A : Type) (P : list A →Prop),
  P nil →
  (∀ a : A, P ([a])) →
  (∀ x y : list A,
   P x ∧P y →P (x ++ y)) →
  ∀x : list A, P x
```

So, we can make use of induction principles on the snoc list and the join list.

To be concrete, let us explain the theorem that reflects the definition on the snoc list. Consider the following foldl function defined on the cons list.

```
Section foldl.
Variables (A B : Type).
Fixpoint foldl (op : B→A→B) (e : B) (x : list A) : B :=
  match x with
    | nil ⇒e
    | a :: x' ⇒foldl op (op e a) x'
    end.
End foldl.
```

In fact, this foldl can be defined more naturally as a homomorphic map from the snoc lists as follows.

```
Section foldl_snoc.
Variables A B : Type.
Fixpoint foldl_snoc (op : B →A →B) (e : B) (x : slist A) : B :=
  match x with
    | snil ⇒e
    | snoc x' a ⇒op (foldl_snoc op e x') a
  end.
```

Actually, many proofs of theorems in BMF follow from this fact. However, it is difficult to define such foldl_snoc function on cons lists as man can not match a cons list with x ++ [a] (pattern matching being dependent of inductive type construction). To resolve this problem, we introduce the following theorem instead:

```
foldl_rev_char : ∀(A B : Type) (op : B →A →B) (e : B) (f : list A →B),
  f nil = e →
  (∀ (a : A) (x : list A), f (x ++ [a]) = op (f x) a) →
  f = foldl op e
```

This theorem means that it is sufficient to prove "f nil = e and f(x ++ [a]) = op (f x) a for all x and a" in order to prove "f = foldl op e".

3.4 Imposing constraints on a higher-order function

In many cases, we have to define some computation patterns (higher-order function) parametrized with some functions (operators) that satisfy some property. For example, the reduce operator behaves like foldl op e, except that it requires that op and e form a monoid in the sense that op is associative and e is the identity unit of op.

To impose such monoidic constraints on the operators of reduce, we define it as follows.

```
Section reduce_mono.
Variables (A : Type)
          (op : A →A →A) (e : A).
Definition reduce_mono (m : monoid op e) :
  list A →A :=
    foldl op e.
End reduce_mono.
```

This reduce_mono exploits the fact that a proof is a term in Coq. Now if the associative operator doesn't have the identity unit in general, we can define the following reduce1 instead, by changing the constraint of reduce_mono.

```
Section reduce1.
Variables (A : Type) (op : A →A →A).
Definition reduce1 (a : assoc op) (d : A)
                   (x : list A) : A :=
  match x with
    | nil ⇒d
    | a' :: x ⇒foldl op a' x
  end.
End reduce1.
```

This reduce1 takes a term of type assoc op, where assoc op is a dependent type that denotes op is associative. Because it is natural to view reduce1 as a partial function defined on a non-empty list, we can implement it with a "default value" as mentioned in Section 3.2. In other words, if a list is nil, reduce1 returns the default value, otherwise reduce1 is defined in terms of foldl.

4 Related Work

There are many systems [8–10, 13] that have been developed for supporting program calculation. Unlike the existing systems, our system is implemented upon the popular and powerful theorem prover Coq with two important features. First, our implementation is light-weighted with only 200 lines of tactic code in Coq. Secondly, our system allows the rich set of theories in Coq to be used in program calculation for free.

Our work is much related to AoPA [14], a library to support encoding of relational derivations in the dependently typed programming language Agda. This follows the line of research for bridging the gap between dependent types and practical programming [15–17]. With AoPA, a program is coupled with an algebraic derivation whose correctness is guaranteed by the type system. However, Agda does not have a strong auto-proving mechanism yet, which would force users to write complicated terms to encode rewriting steps in calculation. Therefore, we turn to Coq. We have exploited Ltac, a language for programming new tactics for easy construction of proofs in Coq, and have seen many advantages of building calculational systems using Coq. First, the Coq tactics

can be used effectively for automatic proving and automatic rewriting, so that tedious calculation can be hidden with tactics. Secondly, new tactics can coexist with the existing tactics, and a lot of useful theories of Coq are ready to use for our calculation. Third, the system on Coq can be used in a trial-and-error style thanks to Coq's interaction mechanism.

A mechanism to describe equational reasoning is available in C-Zar (Radboud University Nijmegen) [18]. This language is a declarative proof language for the Coq proof assistant that can be used instead of the Ltac language to build proofs. It offers a notation to reason on equation but it is limited to Leibniz equality and doesn't allow to transform terms existentially bounded.

5 Conclusion and Future Work

In this paper, we propose a Coq library to support interactive program calculation in Coq. As far as we are aware, this is the first calculation system built upon Coq. Our experience of using the library in encoding the Bird's theory of lists in Coq shows the usefulness and power of the library.

Our future work includes adding theorem about program calculation and tactics to automate proof of correctness of program transformation. We will consider polytypic calculation [19]. Polytypic programming and proofs begin to have some support within Coq [20, 21].

We will also extend the library to program refinement: Indeed, our system currently imposes restrictions on the relation between transformation by forcing it to be an equivalence. In the future, we could add the possibility to explicitely use relation that are not equivalence but refinement relation.

References

1. Bird, R.: Constructive functional programming. In: STOP Summer School on Constructive Algorithmics, Abeland. (September 1989)
2. Kaldewaij, A.: Programming: the derivation of algorithms. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1990)
3. Bird, R., de Moor, O.: Algebra of Programming. Prentice Hall (1996)
4. Bird, R.: An introduction to the theory of lists. In Broy, M., ed.: Logic of Programming and Calculi of Discrete Design, Springer-Verlag (1987) 5–42
5. Gibbons, J.: Algebras for Tree Algorithms. D. phil thesis (1991) Also available as Technical Monograph PRG-94.
6. de Moor, O.: Categories, relations and dynamic programming. Ph.D thesis, Programming research group, Oxford Univ. (1992) Technical Monograph PRG-98.
7. Hu, Z., Takeichi, M., Chin, W.N.: Parallelization in calculational forms. In: POPL'98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press (1998) 316–328
8. Smith, D.R.: KIDS — a knowledge-based software development system. In Lowry, M.R., Mccartney, R.D., eds.: Automating Software Design, Menlo Park, CA, AAAI Press / The MIT Press (1991) 483–514

9. de Moor, O., Sittampalam, G.: Generic program transformation. In: Third International Summer School on Advanced Functional Programming. Lecture Notes in Computer Science, Springer-Verlag (1998)
10. Yokoyama, T., Hu, Z., Takeichi, M.: Yicho: A system for programming program calculations. Technical Report METR 2002–07, Department of Mathematical Engineering, University of Tokyo (June 2002)
11. Bertot, Y., Casteran, P.: Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions. Springer Verlag (2004)
12. Sozeau, M., Oury, N.: First-Class Type Classes. In Mohamed, O.A., Muñoz, C., Tahar, S., eds.: Theorem Proving in Higher Order Logics (TPHOLs). Volume LNCS 5170., Springer (2008) 278–293
13. Visser, E.: A survey of strategies in rule-based program transformation systems. J. Symb. Comput. **40**(1) (2005) 831–873
14. Mu, S.c., Ko, H.s., Jansson, P.: Algebra of programming in agda: Dependent types for relational program derivation. J. Funct. Program. **19**(5) (2009) 545–579
15. Augustsson, L.: Cayenne - a language with dependent types. In: In International Conference on Functional Programming, ACM Press (1998) 239–250
16. McBride, C.: Epigram: Practical programming with dependent types. In Vene, V., Uustalu, T., eds.: Advanced Functional Programming. Volume 3622 of Lecture Notes in Computer Science., Springer (2004) 130–170
17. Norell, U.: Dependently typed programming in agda. In Kennedy, A., Ahmed, A., eds.: TLDI, ACM (2009) 1–2
18. Corbineau, P.: A Declarative Language for the Coq Proof Assistant. In Miculan, M., Scagnetto, I., Honsell, F., eds.: Types for Proofs and Programs, International Conference, TYPES 2007, Cividale des Friuli, Italy, May 2-5, 2007, Revised Selected Papers. LNCS 4941, Springer (2008) 69–84
19. Meertens, L.: Calculate Polytypically! In: PLILP '96: Proceedings of the 8th International Symposium on Programming Languages: Implementations, Logics, and Programs. LNCS 1140, Springer (1996) 1–16
20. Verbruggen, W., de Vries, E., Hughes, A.: Polytypic programming in Coq. In: WGP '08: Proceedings of the ACM SIGPLAN workshop on Generic programming, ACM (2008) 49–60
21. Verbruggen, W., de Vries, E., Hughes, A.: Polytypic properties and proofs in Coq. In: WGP '09: Proceedings of the 2009 ACM SIGPLAN workshop on Generic programming, ACM (2009) 1–12
22. The Coq Development Team: The Coq Proof Assistant. http://coq.inria.fr
23. Bertot, Y.: Coq in a hurry (2006) http://hal.inria.fr/inria-00001173.

A A Very Short Introduction to Coq

The Coq proof assistant [22] is based on the calculus of inductive constructions. This calculus is a higher-order typed λ-calculus. Theorems are types and their proofs are terms of the calculus. The Coq systems helps the user to build the proof terms and offers a language of tactics to do so.

We illustrate quickly all these notions on a short example :

```
Set Implicit Arguments.

Inductive list (A:Set) : Set :=
```

```
| nil : list A
| cons : A →list A →list A.

Implicit Arguments nil [A].
```

The **Set Implicit Arguments** command indicates that we let the Coq system infer as many arguments as possible of the terms we define. If in some context the system cannot infer the arguments, the user has to specify them. In the remainder of this short introduction, the Coq system always infers them.

```
Fixpoint foldr1 (A:Set)(default:A)(f:A→A→A)(l:list A) {struct l} : A :=
  match l with
    | nil ⇒default
    | cons a nil ⇒a
    | cons h t ⇒f h (foldr1 h f t)
  end.


Theorem foldr1_derivation :
  ∀(A:Set)(f:A→list A→A),
    (∀ default, f default nil = default) →
    (∀ default a, f default (cons a nil) = a) →
    (exists g, ∀default a l, f default (cons a l) = g a (f a l)) →
    exists g, ∀default l, f default l = foldr1 default g l.
Proof.
  intros A f H H0 H1.
  destruct H1 as [g Hg].
  exists g.
  intros d l. generalize dependent d.
  induction l.
    (* Case nil *) assumption.
    (* Case cons *)
    intro d. rewrite Hg.
    destruct l.
      (* Case nil *) rewrite <− (Hg a). apply H0.
      (* Case cons *) rewrite IHl. reflexivity.
Qed.
```

In this example, we first define a new inductive type, the type of lists. This type is a polymorphic type since it takes as an argument a type A, the type of the elements of the list. A has type **Set** which means it belongs to the computational realm of the Coq language similar to ML data structures. The definition of this new type introduces two new functions, the constructors of this type: nil and cons. Then we define a recursive function: foldr1. In this definition we specify the decreasing argument (here the fourth argument) as all functions must be terminating in Coq and we give its type (after ":") as well as a term (after ":=") of this type.

We then define a theorem named foldr1_derivation stating that:

```
∀(A:Set)(f:A→list A→A),
    (∀ default, f default nil = default) →
```

```
(∀ default a, f default (cons a nil) = a) →
(exists g, ∀default a l, f default (cons a l) = g a (f a l)) →
exists g, ∀default l, f default l = foldr1 default g l.
```

If we check (using the Check command of Coq) the type of this expression, we would obtain Prop meaning that this expression belongs to the logical realm. To define foldr1_derivation we also should provide a term of this type, that is a proof of this theorem. We could write directly such a term, but it is usually complicated and Coq provides a language of tactics to help the user to build a proof term. We refer to [23] for a quick yet longer introduction to Coq, including the presentation of basic tactics.

Mixing logical and computational parts is possible in Coq. For example a function of type A→B with a precondition P and a postcondition Q corresponds to a constructive proof of type: ∀x:A, (P x) →exists y:B →(Q x y). This mix could be expressed in Coq by using the inductive type sig:

```
Inductive sig (A:Set) (Q:A→Prop) : Set :=
          | exist: ∀(x:A), (Q x) →(sig A Q).
```

It could also be written, using syntactic sugar, as {x:A|(Q x)}.

This feature is used in the definition of the function head. The specification of this function is:∀(A:Set) (l:list A), l<>nil →{ a:A | exists t, l = cons a t } and we build it using tactics:

```
Definition head (A:Set)(l:list A) :
  l<>nil →{ a:A | exists t, l = cons a t }.
Proof.
  intros.
  destruct l as [ _ | h t].
  (* case nil *) elim H. reflexivity.
  (* case cons *) exists h. exists t. reflexivity.
Defined.
```

The command **Extraction** head would extract the computational part of the definition of head. We could obtain a certified implementation of the head function (here in Objective Caml):

```
(** val head : 'a1 list →'a1 **)
let head = function
  | Nil →assert false (* absurd case *)
  | Cons (a, l0) →a
```