Program Calculation in Coq

J. Tesson[1] H. Hashimoto[3] Z. Hu[2,3] F. Loulergue[1]
M. Takeichi[3]

1 : Université d'Orléans, LIFO
2 : National Institute of Informatics, Tokyo
3 : The University of Tokyo, IPL

AMAST, June 2010

Outline

1. Introduction
   - Program calculation
   - Tiny introduction to Coq
2. Program Calculation in Coq
   - Equivalence Proof
   - Constructing functions using existantial quantification
3. Bird Mirteen Formalism Implementation
4. Conclusion and Perspectives

1 Introduction
Program calculation
Tiny introduction to Coq

2 Program Calculation in Coq
Equivalence Proof
Constructing functions using existantial quantification

3 Bird Mirteen Formalism Implementation

4 Conclusion and Perspectives

Program calculation

$maximum = hd \circ sort.$

if law :
$\forall$ f,
f (if a then b else c)
= if a then f b else f c

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

Program calculation

$$maximum = hd \circ sort.$$

if law :
$\forall$ f,
f (if a then b else c)
= if a then f b else f c

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

Program calculation

$$maximum = hd \circ sort.$$

if law :
$\forall$ f,
f (if a then b else c)
= if a then f b else f c

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

Program calculation

$$maximum = hd \circ sort.$$

if law :
$\forall$ f,
f (if a then b else c)
= if a then f b else f c

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

Program calculation

$$maximum = hd \circ sort.$$

if law :
$\forall$ f,
f (if a then b else c)
= if a then f b else f c

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

Pros & Cons

Pros

Correction of

- optimizations
- specifications implementations

Cons

- If done manually
  - errors still possible
  - time consuming
- Mechanized tools
  - hard to implement
  - a lot of details to deal with

Tiny introduction to Coq

Two languages

- Galina for writing terms
- Ltac for writing proofs

```
Fixpoint insert (a :nat) (l : list nat) {struct l} : (list nat) :=
    match l with
      | nil ⇒ nil
      | n::t ⇒ if a ?> n then (a :: l) else(n:: (insert a t))
    end.
Lemma maximum_over_list :
    ∀ a x d,
      maximum d (a::x) =
      if a ?> (maximum a x) then a else maximum a x.
```

Tiny introduction to Coq

Two languages

- Galina for writing terms
- Ltac for writing proofs

```
Lemma maximum_over_list :
    ∀ a x d,
      maximum d (a::x) =  ...
  Proof.
    intros a l d. unfold maximum. unfold sort.
    simpl. fold sort. rewrite if_law. simpl.
    fold (maximum a l). simpl_if. reflexivity.
    reflexivity.
  Qed.
```

Tiny introduction to Coq

Two languages

- Galina for writing terms
- Ltac for writing proofs

```
Ltac simpl_if :=
    destruct le_gt_dec as [H|H];
    try(contradict H; simpl; auto with arith;fail).
```

Tiny introduction to Coq - Ugly example

```
Lemma maximum_over_list :
    ∀ a x d,
      maximum d (a ::x) =
       if a ?> (maximum a x) then a else maximum a x.
  Proof.
    intros a l d.
    unfold maximum. unfold sort.
    simpl. fold sort. rewrite if_law. simpl.
    fold (maximum a l).
    simpl_if.
     reflexivity.
     reflexivity.
  Qed.
```

1 Introduction
Program calculation
Tiny introduction to Coq

2 Program Calculation in Coq
Equivalence Proof
Constructing functions using existantial quantification

3 Bird Mirteen Formalism Implementation

4 Conclusion and Perspectives

Example

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

Starting Derivation

```
Lemma maximum_over_list :
    ∀ a x d,
      maximum d (a::x) =
       if a ?> (maximum a x) then a else maximum a x.
Begin.
```

- Moving premises from goal to environment
- Delaying existential instantiation

Goal selection

```
Lemma maximum_over_list :
    ∀ a x d,
      maximum d (a::x) =
       if a ?> (maximum a x) then a else maximum a x.
Begin.
   LHS
```

LHS select the left hand side of an equation

RHS select the right hand side of an equation

BOTH_SIDES the whole goal

Transformation step

```
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)) ).
```

corresponds to

```
maximum d (a :: x)
=     { def. of maximum }
hd d (sort (a :: x))
```

Transformation step

```
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)) ).
```

corresponds to

maximum d (*a* :: *x*)
= { def. of maximum }
hd d (*sort* (*a* :: *x*))

$$= \{\ \texttt{ta}\ \}\ (\text{term}).$$

- `ta` can be any Ltac tactics
- term is a typecheckable term

This tactic is successful iff term is proved equivalent to selected part by tactic `ta`

End of the proof

```
Lemma maximum_over_list :
    ∀ a x d,
      maximum d (a::x) = if a ?> (maximum a x) then a else maximum a x.
  Proof.
    Begin.
    LHS
          ⋮
    ={ by def maximum}
       (if a ?> (maximum a x) then a else maximum a x).
    [].
  Qed.
```

Closing proof by [] as L.H.S. and R.H.S. are convertible.

Constructing functions - Existential use

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

Define the term (f, p) where

- f is `fun a d ⇒ a` and
- p is the a proof that ∀ a d, maximum d [a] = f a

Existential use - Begin

```
Definition maximum_singleton :
    {f | ∀ a d, maximum d [a] = f a}.
  Begin.
```

- Begin delay instantiation of f so that we can transform the goal

Existential use - Specification transformation

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

We transform the specification as previously

Existential use - Instantiation

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

We end the transformation using `[]`.
This provoke instantiation of f by the current form (here a)

1 Introduction
Program calculation
Tiny introduction to Coq

2 Program Calculation in Coq
Equivalence Proof
Constructing functions using existantial quantification

3 Bird Mirteen Formalism Implementation

4 Conclusion and Perspectives

Bird Mirteen Formalism implementation

We implemented about

- 30 definitions
- 9 notations
- 36 lemmas

from the "Introduction to the Theory of Lists" (*R. Bird*)
It was done in Coq with

- 65 defintions
- 12 notations
- 220 lemmas
- 3 800 lines of code

Bird Mirteen Formalism usage

```
Lemma map_fusion_comp_assoc :
        ∀ (A : Type) (f : A→A) (g : A→A),
  ((mapf) : o : (map f) : o : (map g) : o : (map g))
  ==
  (map(f : o : f) : o : map(g : o : g)).
Proof.
  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.
```

1 Introduction
Program calculation
Tiny introduction to Coq

2 Program Calculation in Coq
Equivalence Proof
Constructing functions using existantial quantification

3 Bird Mirteen Formalism Implementation

4 Conclusion and Perspectives

Conclusion

We have an interactive program calculation tools with

- correctness of transformation proved by the trustworthy proof assistant Coq
- Readable code
- Take advantage of the large base of already proved theorems

Code available at `https://traclifo.univ-orleans.fr/SDPP/`

Perspectives

- Interactively construct fixpoint
- Extend the library with more rules for program calculation
- Define and handle refinement relations

All this may require that we develop a Coq plugin.

any Questions ?

Begin

```
Ltac Begin :=
  intros;
  match goal with
    | [⊢ ?R ?lhs ?rhs] ⇒ is_equivalence R; idtac
    | [⊢ ∃ A ,_] ⇒
      eexists;Begin
    | [⊢ sig _] ⇒
      eexists;Begin
    | _ ⇒ idtac "no existential, no equivalence"
  end.
```

Goal Selection

```
Tactic Notation (at level 1) "RHS" tactic(t) :=
  (if_RHS_LHS_BOTH_NOTH
    (idtac)
    (idtac;
      match goal with
        | [⊢ (?R ?lhs ?rhs)] ⇒ subst rhs
      end
    )
    (idtac)
    (idtac)
  );
  setRHS;
  match goal with
    | [⊢ ?R ?lhs ?rhs] ⇒ is_equivalence R;
      let LHS := fresh "LHS" in set (LHS :=lhs);
      assert(R LHS rhs);[|assumption]
  end
  .
```

Goal Selection

```
Tactic Notation "setRHS" :=
  first [
   (match goal with
    | [ H : memo RHS ⊢ _]⇒ idtac "already RHS"
    | [ H : memo LHS ⊢ _]⇒ clear H ; memorize RHS
    | [ H : memo BOTH_SIDES ⊢ _]⇒ clear H ; memorize RHS
    | [ H : memo (ADHOC ?s) ⊢ _]⇒ memorize RHS
   end)
  | (memorize RHS)]
.
```

Transformation step

```
Tactic Notation (at level 2) "=" "{" tactic(t1) "}" constr(e2) :=
  if_RHS_LHS_BOTH
  (idtac;
    match goal with
      | [⊢ ?R ?lhs ?rhs] ⇒
        is_equivalence R;
        let h := fresh "rewriting" in
          assert(h : R rhs e2) by (t1;reflexivity);rewrite h
    end)
  (... )
  (idtac;
    match goal with
     [⊢ ?R ?lhs ?rhs] ⇒
     is_equivalence R;
     match e2 with
       | ?R ?lhs_e2 ?rhs_e2 ⇒
         is_equivalence R;
         let h := fresh "rewriting" in
           assert(h : R lhs lhs_e2) by (t1;reflexivity);
           rewrite h;
             let h2 := fresh "rewriting" in
               assert(h : R rhs rhs_e2) by (t1;reflexivity);
               rewrite h2
     end
   end
  )
```

Memorization mechanism

```
Inductive state : Prop :=
  RHS : state
| LHS : state
| BOTH_SIDES : state
| ADHOC : ∀ s :string, state.

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

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

Partial functions

- Using dependent types limiting input arguments $\{l \mid l <> nil\}$
- Taking a proof that excluded case are not reachable as argument (p : $l <> nil$)
- Using option A as return type (None | Some a)
- Using a default value for special cases