Automatic refinements in Coq

Damien Rouhling
Under the supervision of Cyril Cohen

ENS Lyon & INRIA Sophia Antipolis

June 20, 2016

Context and motivations

- Interactive theorem proving: Coq.
- Simplifying proofs: reflection.
- Small scale reflection: Mathematical Components and SSReflect.
- Sources of inefficiency: locks, unadapted structures.
- Ideal world:
  1. develop a theory using well-adapted structures *independently* of what people want to compute with them,
  2. *reuse* this theory to get proofs on more complex structures.
- Method: CoqEAL's refinement framework.

Goals and results

- Redevelop CoqEAL with a better automation of refinement.
- Make CoqEAL user-friendly.
- Extend CoqEAL with new refinements.
- Develop applications of CoqEAL.

Goals and results

- Redevelop CoqEAL with a better automation of refinement.
  Almost everything redevelopped: (positive) natural numbers, integers, polynomials, Karatsuba's algorithm, matrices, Bareiss' algorithm, rational numbers, $\mathbb{F}_2$.
- Make CoqEAL user-friendly.
  `CoqEAL` tactic, refinement at definition time.
- Extend CoqEAL with new refinements.
  $\mathbb{Z}/n\mathbb{Z}$, sequences, new operations on matrices.
- Develop applications of CoqEAL.
  Large scale reflection.

Refinements

Sequence of refinement steps

$$P_1 \rightarrow P_2 \rightarrow \cdots \rightarrow P_n$$

where:

- P_1 is an abstract version of the program,
- P_n is a concrete version of the program,

- P_1 is an proof-oriented version of the program,
- P_n is a computation-oriented version of the program,

- Each P_i is correct w.r.t. P_{i-1}.

Two kinds of refinement

We distinguish two kinds of refinement:

- program refinement: improving the algorithms without changing the data structures,
- data refinement: use the same algorithms on more efficient data representations and primitives.

An important property for data refinement: compositionality.

Example: Karatsuba's algorithm

Program refinement:

Karatsuba's algorithm is an algorithm for fast polynomial multiplication ($O\left(n^{\log_2 3}\right)$) inspired from the following equation:

$$(aX^n + b)(cX^n + d) = acX^{2n} + ((a + b)(c + d) - ac - bd)X^n + bd.$$

Specification

```
Lemma karatsubaE : forall p q, karatsuba p q = p * q.
```

Example: Horner's polynomials

Data refinement:

```
Inductive hpoly A :=
  | Pc : A -> hpoly A
  | PX : A -> pos -> hpoly A -> hpoly A.
```

$$aX^n + b \to \begin{cases} \texttt{PX}\ \ b\ n\ (\texttt{Pc}\ \ a) \text{ if } n > 0, \\ \texttt{Pc}\ \ (a + b) \text{ otherwise.} \end{cases}$$

Refinement relation

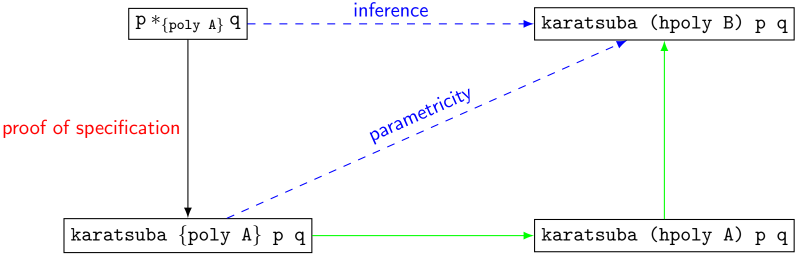
```
Definition Rhpoly A B (R : A -> B -> Type) :
  {poly A} -> hpoly B -> Type :=
    fun p hp => to_poly hp = p.
```

Example: full refinement path

$$\begin{tikzcd}
\boxed{\texttt{p} *_{\texttt{\{poly A\}}} \texttt{q}} \arrow[r, dashed] \arrow[d, "\text{program refinement}"'] & \boxed{\texttt{karatsuba (hpoly B) p q}} \\
\boxed{\texttt{karatsuba \{poly A\} p q}} \arrow[r, "\text{data refinement}"] \arrow[ur, dashed, "\text{parametricity}"] & \boxed{\texttt{karatsuba (hpoly A) p q}} \arrow[u, "\text{compositionality}"']
\end{tikzcd}$$

Parametricity: PARAMCOQ plugin by C. Keller and M. Lasson.

Automation



$\texttt{p} *_{\texttt{\{poly A\}}} \texttt{q}$

inference

`karatsuba (hpoly B) p q`

proof of specification

parametricity

`karatsuba {poly A} p q`

`karatsuba (hpoly A) p q`

User input.

Requirement: proofs of primitives.

Automatic.

Refinement inference

A type class for refinements:

```
Class refines A B (R : A -> B -> Type) (m : A) (n : B) :=
  refines_rel : R m n.
```

Program/term synthesis:
We solve

```
?proof : refines ?relation input ?output,
```

e.g. with `input := 2 *: 'X`, we get

```
?relation := Rhpoly R,
?output := PX 0 1 (Pc 2),
?proof := P :
  refines (Rhpoly R) (2 *: 'X + 1) (PX 0 1 (Pc 2)).
```

Example

Assume

R : A -> B -> Type,
refines R x y,
refines R z t,
refines (R ==> R ==> R) $+_A$ $+_B$,
refines (R ==> R ==> R) $*_A$ $*_B$.

Global goal:

refines ?R (x $+_A$ (x $*_A$ z)) ?y.

Current goal(s):

refines ?R (x $+_A$ (x $*_A$ z)) ?y.

Example

Assume

```
R : A -> B -> Type,
refines R x y,
refines R z t,
refines (R ==> R ==> R) +_A +_B,
refines (R ==> R ==> R) *_A *_B.
```

Global goal:

```
refines ?R (x +_A (x *_A z)) (?f ?y').
```

Current goal(s):

```
refines (?R' ==> ?R) (fun a => x +_A a) ?f,
refines ?R' (x *_A z) ?y'.
```

Example

Assume

```
R : A -> B -> Type,
refines R x y,
refines R z t,
refines (R ==> R ==> R) +_A +_B,
refines (R ==> R ==> R) *_A *_B.
```

Global goal:

```
refines ?R (x +_A (x *_A z)) (?f' ?y'' ?y').
```

Current goal(s):

```
refines (?R'' ==> ?R' == ?R) +_A ?f',
refines ?R'' x ?y'',
refines ?R' (x *_A z) ?y'.
```

Example

Assume

```
R : A -> B -> Type,
refines R x y,
refines R z t,
refines (R ==> R ==> R) +A +B,
refines (R ==> R ==> R) *A *B.
```

Global goal:

```
refines R (x +A (x *A z)) (?y'' +B ?y').
```

Current goal(s):

```
refines R x ?y'',
refines R (x *A z) ?y'.
```

Example

Assume

```
R : A -> B -> Type,
refines R x y,
refines R z t,
refines (R ==> R ==> R) +_A +_B,
refines (R ==> R ==> R) *_A *_B.
```

Global goal:

```
refines R (x +_A (x *_A z)) (y +_B ?y').
```

Current goal(s):

```
refines R (x *_A z) ?y'.
```

Example

Assume

```
R : A -> B -> Type,
refines R x y,
refines R z t,
refines (R ==> R ==> R) +_A +_B,
refines (R ==> R ==> R) *_A *_B.
```

Global goal:

```
refines R (x +_A (x *_A z)) (y +_B (?f ?y)).
```

Current goal(s):

```
refines (?R ==> R) (fun a => x *_A a) ?f,
refines ?R z ?y.
```

Example

Assume

```
R : A -> B -> Type,
refines R x y,
refines R z t,
refines (R ==> R ==> R) +_A +_B,
refines (R ==> R ==> R) *_A *_B.
```

Global goal:

```
refines R (x +_A (x *_A z)) (y +_B (?f' ?y' ?y)).
```

Current goal(s):

```
refines (?R' ==> ?R ==> R) *_A ?f',
refines ?R' x ?y',
refines ?R z ?y.
```

Example

Assume

R : A -> B -> Type,
refines R x y,
refines R z t,
refines (R ==> R ==> R) $+_A$ $+_B$,
refines (R ==> R ==> R) $*_A$ $*_B$.

Global goal:

refines R (x $+_A$ (x $*_A$ z)) (y $+_B$ (?y' $*_B$?y)).

Current goal(s):

refines ?R' x ?y',
refines ?R z ?y.

Example

Assume

```
R : A -> B -> Type,
refines R x y,
refines R z t,
refines (R ==> R ==> R) +A +B,
refines (R ==> R ==> R) *A *B.
```

Proven:

```
refines R (x +A (x *A z)) (y +B (y *B t)).
```

Example (cont.)

Assume

```
R : A -> B -> Type,
refines R x y,
refines R z t,
refines (R ==> R ==> R) +_A +_B,
refines (R ==> R ==> R) *_A *_B.
```

Global goal:

```
refines ?R (x +_A (x *_A z)) (?f y t).
```

Current goal(s):

```
refines ?R (x +_A (x *_A z)) (?f y t).
```

Example (cont.)

Assume

```
R : A -> B -> Type,
refines R x y,
refines R z t,
refines (R ==> R ==> R) +_A +_B,
refines (R ==> R ==> R) *_A *_B.
```

Global goal:

```
refines ?R (x +_A (x *_A z)) (?f y t).
```

Current goal(s):

```
refines (?R' ==> ?R) (fun a => x +_A a) (?f y),
refines ?R' (x *_A z) t.
```

Example (cont.)

Assume

```
R : A -> B -> Type,
refines R x y,
refines R z t,
refines (R ==> R ==> R) +_A +_B,
refines (R ==> R ==> R) *_A *_B.
```

Solution:

```
Class unify A (x y : A) := unify_rel : x = y.
Instance unifyxx A (x : A) : unify x x := erefl.
```

With the goal:

```
refines (?R o unify) (x +_A (x *_A z)) (?f y t),
```

which splits into

```
refines ?R (x +_A (x *_A z)) ?e,
refines unify ?e (?f y t).
```

Proofs by computation

```
Definition ctmat1 : 'M[int]_(3, 3) :=
  \matrix_(i, j) ([:: [::  1 ; 1 ; 1 ]
                    ; [:: -1 ; 1 ; 1 ]
                    ; [::  0 ; 0 ; 1 ] ]`_i)`_j.
```

```
Lemma det_ctmat1 : \det ctmat1 = 2.
Proof.
  by do ?[rewrite (expand_det_row _ ord0) //=;
  rewrite ?(big_ord_recl,big_ord0) //= ?mxE //=;
  rewrite /cofactor /= ?(addn0, add0n, expr0, exprS);
  rewrite ?(mul1r,mulr1,mulN1r,mul0r,mul1r,addr0) /=;
  do ?rewrite [row' _ _]mx11_scalar det_scalar1 !mxE
     /=].
Qed.
```

Proofs by computation

```
Definition ctmat1 : 'M[int]_(3, 3) :=
  \matrix_(i, j) ([:: [::  1 ; 1 ; 1 ]
                    ; [:: -1 ; 1 ; 1 ]
                    ; [::  0 ; 0 ; 1 ] ]`_i)`_j.
```

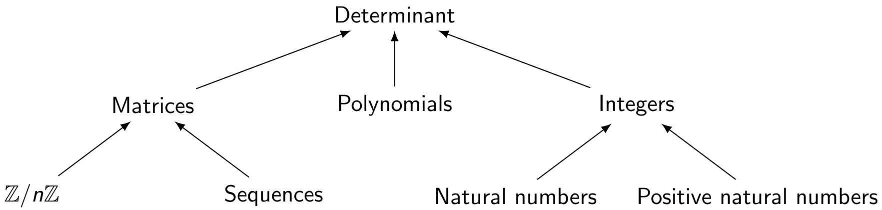
```
Lemma det_ctmat1 : \det ctmat1 = 2.
Proof. by CoqEAL. Qed.
```

Proofs by computation

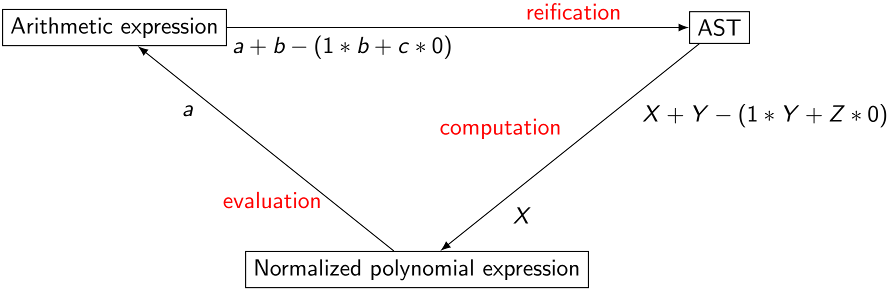
```
Definition ctmat1 : 'M[int]_(3, 3) :=
  \matrix_(i, j) ([:: [::  1 ; 1 ; 1 ]
                    ; [:: -1 ; 1 ; 1 ]
                    ; [::  0 ; 0 ; 1 ] ]`_i)`_j.

Definition det_ctmat1 := refine_value (\det ctmat1).
```

Refinements for determinant computation

To compute the determinant of `ctmat1`:



The tactic `ring`



Arithmetic expression

reification

AST

$a + b - (1 * b + c * 0)$

$X + Y - (1 * Y + Z * 0)$

computation

a

evaluation

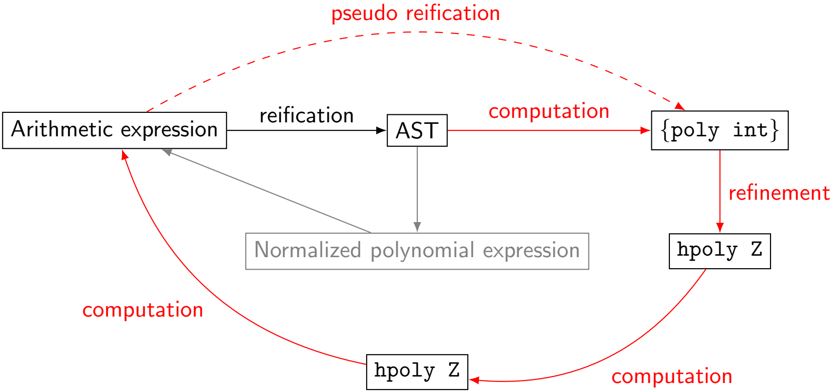
X

Normalized polynomial expression

- *Specific* refinements.
- Proof of correctness, roughly:

```
Lemma ring_correct env t p :
  poly_of_ast t = p -> interp env t = eval env p.
```

Refactoring the tactic `ring`



Further work

- Fix issues with partial refinements and partial application of parametricity.
- Better interface for the user, e.g. a better debugging system.
- More refinements: finite sets (P.-E. Dagand and E. Gallego Arias), Strassen's algorithm, Smith normal form computation...
- Finish the refactoring of `ring`.

Conclusion

Results:

- Redevelopment of CoqEAL with a more systematic use of parametricity.
- A better interface to trigger refinement.
- New refinements: $\mathbb{Z}/n\mathbb{Z}$, more operations on matrices.
- More use cases: determinants.
- Prototype of a more modular tactic `ring` with the possibility of using CoqEAL's refinement framework.

Conclusion

Results:

- Redevelopment of CoqEAL with a more systematic use of parametricity.
- A better interface to trigger refinement.
- New refinements: $\mathbb{Z}/n\mathbb{Z}$, more operations on matrices.
- More use cases: determinants.
- Prototype of a more modular tactic `ring` with the possibility of using CoqEAL's refinement framework.

Thank you!

The parametricity theorem

Relational interpretation for types:

$$\begin{array}{rcl} [\![A \to B]\!] & := & \{(f,g) \mid \forall\, (x,y) \in [\![A]\!] \,.\, (f\ x, g\ y) \in [\![B]\!]\}\,, \\ [\![\forall X.A]\!] & := & \{(f,g) \mid \forall R.\, (f,g) \in [\![A]\!]\,\{R/\,[\![X]\!]\}\}\,. \end{array}$$

Parametricity theorem (J. Reynolds 1983, P. Wadler 1989)

For all closed type A and all closed term t of type A, $[\![t]\!]$ is a term of type $[\![A]\!]\ \ t\ t$.

Generic programming

From

```
Record rat : Set := Rat {
  valq : int * int ;
  _ : (0 < valq.2) && coprime `|valq.1| `|valq.2|
}
```

to

```
Definition Q Z := Z * Z.
```

Generic operation

```
Definition addQ Z +_Z *_Z : Q Z -> Q Z -> Q Z :=
  fun x y => (x.1 *_Z y.2 +_Z y.1 *_Z x.2, x.2 *_Z y.2).
```

Correctness of `addQ`

- Proof-oriented correctness: instantiate `Z` with `int`,
- relation `Rrat: rat -> Q int -> Type`,
- prove the following theorem:

```
Lemma Rrat_addQ :
  (Rrat ==> Rrat ==> Rrat) +rat (addQ int +int *int).
```

Correctness of addQ (cont.)

Generalization using compositionality: from the refinement relation `Rint : int -> C -> Type`,

```
Definition RratC : rat -> C * C -> Type :=
  Rrat o (Rint * Rint).
```

Goal:

```
Lemma RratC_add :
  (RratC ==> RratC ==> RratC) +rat (addQ C +C *C).
```

Correctness of addQ (cont.)

Generalization using compositionality: from the refinement relation `Rint : int -> C -> Type`,

```
Definition RratC : rat -> C * C -> Type :=
  Rrat o (Rint * Rint).
```

Goal:

```
Lemma RratC_add :
  (RratC ==> RratC ==> RratC) +rat (addQ C +C *C).
```

This splits into

```
(Rrat ==> Rrat ==> Rrat) +rat (addQ int +int *int),
```

already proven and

```
(Rint * Rint ==> Rint * Rint ==> Rint * Rint)
  (addQ int +int *int) (addQ C +C *C).
```

Correctness of `addQ` (end)

Goal:

$$\texttt{(Rint * Rint ==> Rint * Rint ==> Rint * Rint)}$$
$$\texttt{(addQ int }+_{\texttt{int}}\ *_{\texttt{int}}\texttt{) (addQ C }+_{\texttt{C}}\ *_{\texttt{C}}\texttt{).}$$

Correctness of addQ (end)

Goal:

$$\texttt{(Rint * Rint ==> Rint * Rint ==> Rint * Rint)}$$
$$\texttt{(addQ int }+_{\texttt{int}}\ *_{\texttt{int}}\texttt{) (addQ C }+_{\texttt{C}}\ *_{\texttt{C}}\texttt{).}$$

By parametricity:

$$[\![\forall \texttt{Z}.\,(\texttt{Z} \to \texttt{Z} \to \texttt{Z}) \to (\texttt{Z} \to \texttt{Z} \to \texttt{Z}) \to \texttt{Z} * \texttt{Z} \to \texttt{Z} * \texttt{Z} \to \texttt{Z} * \texttt{Z}]\!]\ \texttt{addQ}\ \texttt{addQ},$$

i.e.

```
∀Z : Type. ∀Z’ : Type. ∀R : Z -> Z’ -> Type.
∀addZ : Z -> Z -> Z. ∀addZ’ : Z’ -> Z’ -> Z’.
(R ==> R ==> R) addZ addZ’ ->
  ∀mulZ : Z -> Z -> Z. ∀mulZ’ : Z’ -> Z’ -> Z’.
  (R ==> R ==> R) mulZ mulZ’ ->
    (R * R ==> R * R ==> R * R)
    (addQ Z addZ mulZ) (addQ Z’ addZ’ mulZ’).
```

Type classes

Definition

A type class is defined by a finite set of constraints (axioms) over one or several type variables. It is the class of the (tuples of) types that satisfy the constraints.

Example: the class of the types that admit an equality operator, and an instance of that class.

```
Class eq_of A := eq_op : A -> A -> bool.
Instance eq_N : eq_of N := N.eqb.
```

Logic programming for refinements

Rules to decompose expressions, such as

```
Instance refines_apply
A B (R : A -> B -> Type) A’ B’ (R’ : A’ -> B’ -> Type) :
  forall (f : A -> A’) (g : B -> B’),
  refines (R ==> R’) f g ->
    forall (a : A) (b : B), refines R a b ->
      refines R’ (f a) (g b).

Lemma refines_trans A B C (rAB : A -> B -> Type)
(rBC : B -> C -> Type) (rAC : A -> C -> Type)
(a : A) (b : B) (c : C) :
  composable rAB rBC rAC ->
    refines rAB a b -> refines rBC b c ->
      refines rAC a c.
```