Refinement: a reflection on proofs and computations

Cyril Cohen & Damien Rouhling

Based on previous work by
Maxime Dénès, Anders Mörtberg & Vincent Siles

Université Côte d'Azur, Inria, France

March 6, 2017

Context

- Computers are increasingly used for mathematical proofs, especially for their computational power.

  For instance:
  - The four color theorem [Appel, Haken 1977; Gonthier 2008].
  - Kepler conjecture [Hales 2005].
  - The odd order theorem [Gonthier et al. 2013].
- Different tools with different purposes (really rough approximation):
  - Computer algebra software: efficient computations.
  - Automatic theorem provers: efficient logical reasoning.
  - Interactive theorem provers: sound logical reasoning.
- We want to ensure that efficient tools use sound techniques.
- Ease of use matters.

We will focus on sound and efficient computations.

Motivations

Program verification closes the gap between paper proofs and implementations:

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

↓

Program verification

↓

```
Fixpoint karatsuba_rec n p q := match n with
| 0 => p * q
| n'.+1 =>
    let sp := size p in let sq := size q in
    if (sp <= 2) || (sq <= 2) then p * q else
      let m := (minn sp./2 sq./2) in
      let (a,b) := splitp m p in
      let (c,d) := splitp m q in
      let ac := karatsuba_rec n' a c in
      let bd := karatsuba_rec n' b d in
      let apb := a + b in
      let cpd := c + d in
      let apb_cpd := karatsuba_rec n' apb cpd in
      (shiftp (2 * m) ac + (shiftp m (apb_cpd - ac - bd)) + bd)
end.

Definition karatsuba p q :=
  karatsuba_rec (maxn (size p) (size q)) p q.
```

Motivations (cont.)

Computations shorten proof terms and make the users' life easier.

- `1 + (2 + 3) = 6` by reflexivity instead of using the rules:

  ```
  n + 0 = n.
  n + (S m) = S (n + m).
  ```

- `M` is invertible iff `\det M` is not 0.

Separation of concerns

Issues:

- Efficient algorithms are often hard to prove correct.
  For instance: the Sasaki-Murao algorithm [Coquand, Mörtberg, Siles 2012].
- Structures that are adapted to proofs are often inefficient for computations.
  For instance in Coq: `nat` or Mathematical Components polynomials.
- We do not want to develop a theory for each representation of the same object.

Ideal world:

1. Develop one 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.

Outline

1. CoqEAL's refinement framework
2. Automation
3. Applications

Refinement [Dénès, Mörtberg, Siles 2012]

Sequence of refinement steps

$$P_1 \to P_2 \to \cdots \to P_n$$

where:

In the literature

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

In CoqEAL

- 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: improve 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 : {poly A},
  karatsuba p q = p *_{poly A} 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 : {poly A} -> hpoly A -> Type :=
  fun p hp => to_poly hp = p.
```

Example: Horner's polynomials (cont.)

Compositionality:

```
Definition hpoly_R A B (R : A -> B -> Type) :
  hpoly A -> hpoly B -> Type := ...

Rhpoly o (hpoly_R R) : {poly A} -> hpoly B -> Type
```

Example: full refinement path

```
karatsubaE : forall A (p q : {poly A}),
  karatsuba p q = p *_{poly A} q

Rhpoly : forall A, {poly A} -> hpoly A -> Type

hpoly_R : forall A B (R : A -> B -> Type),
  hpoly A -> hpoly B -> Type
```

$$\begin{tikzcd}
\texttt{p} *_{\texttt{\{poly A\}}} \texttt{q} \arrow[r, dashed] \arrow[d, "\text{(karatsubaE)}"', "\text{program refinement}"] & \texttt{karatsuba (hpoly B) hp' hq'} \\
\texttt{karatsuba \{poly A\} p q} \arrow[r, "\text{data refinement (\texttt{\{poly A\}})}", "\text{(\texttt{Rhpoly})}"'] & \texttt{karatsuba (hpoly A) hp hq} \arrow[u, "\text{(\texttt{hpoly\_R})}", "\text{data refinement (\texttt{hpoly A}) by compositionality}"']
\end{tikzcd}$$

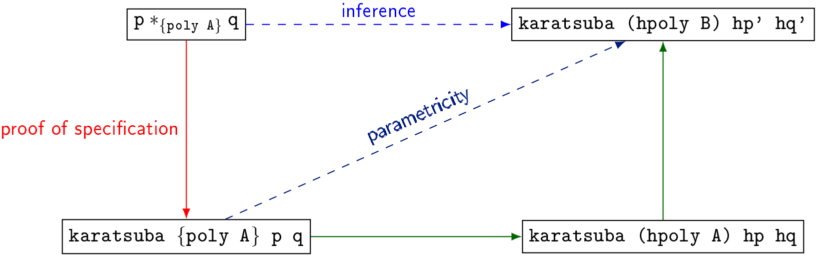
Outline

1 CoqEAL's refinement framework

2 Automation

3 Applications

Degrees of automation



User input.

Requirement: correctness of primitives.

Type classes.

Plugin: ParamCoq [Keller, Lasson 2012].

The parametricity theorem [Reynolds 1983; Wadler 1989]

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

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

Moreover, one can compute $[\![t]\!]$.

Example

```
Inductive hpoly A := ...
```

```
⟦hpoly⟧ : ∀A, ∀B, ∀R : A -> B -> Type, ...
```

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

```
Definition hpoly_R A B (R : A -> B -> Type) :
  hpoly A -> hpoly B -> Type := ...
```

Example

```
Inductive hpoly A := ...

⟦hpoly⟧ : ∀A, ∀B, ∀R : A -> B -> Type, ...
```

$$\begin{tikzcd}[column sep=huge, row sep=huge]
\texttt{p} *_{\texttt{\{poly A\}}} \texttt{q} \arrow[r, dashed] \arrow[d, "\text{program refinement}"'] & \boxed{\texttt{karatsuba (hpoly B) hp' hq'}} \\
\boxed{\texttt{karatsuba \{poly A\} p q}} \arrow[r, "\text{data refinement}", "\texttt{(Rhpoly)}"'] & \boxed{\texttt{karatsuba (hpoly A) hp hq}} \arrow[u, "\texttt{(hpoly\_R)}", "\substack{\text{data refinement} \\ \text{compositionality}}"']
\end{tikzcd}$$

```
Definition hpoly_R A B (R : A -> B -> Type) :
  hpoly A -> hpoly B -> Type := ⟦hpoly⟧ R.
```

Example (cont.)

$$\llbracket \texttt{karatsuba} \rrbracket \;:\; \llbracket \forall \mathrm{P}, \mathrm{P} \to \mathrm{P} \to \mathrm{P} \rrbracket \;\texttt{karatsuba}\;\texttt{karatsuba}$$

i.e.

$$\begin{array}{l}\llbracket \texttt{karatsuba} \rrbracket \;\texttt{:}\; \forall\texttt{P},\; \forall\texttt{C},\; \forall\;\texttt{R}\;\texttt{:}\;\texttt{P -> C -> Type,} \\ \quad \texttt{(R ==> R ==> R) (karatsuba P) (karatsuba C)}\end{array}$$

$$\begin{tikzcd}
\texttt{p} *_{\texttt{\{poly A\}}} \texttt{q} \arrow[rr, dashed] \arrow[d, "\texttt{karatsubaE}"'] & & \texttt{karatsuba (hpoly B) hp' hq'} \\
\texttt{karatsuba \{poly A\} p q} \arrow[rru, dashed, "\llbracket\texttt{karatsuba}\rrbracket\texttt{ (Rhpoly o (}\llbracket\texttt{hpoly}\rrbracket\texttt{ R))}"] \arrow[rr, "\llbracket\texttt{karatsuba}\rrbracket\texttt{ Rhpoly}"] & & \texttt{karatsuba (hpoly A) hp hq} \arrow[u, "\llbracket\texttt{karatsuba}\rrbracket\texttt{ (}\llbracket\texttt{hpoly}\rrbracket\texttt{ R)}"']
\end{tikzcd}$$

Refinement inference [Cohen, Dénès, Mörtberg 2013]

A type class for refinement:

```
Class refines P C (R : P -> C -> Type) (p : P) (c : C) :=
  refines_rel : R p c.
```

Program/term synthesis:

We solve by type class inference

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

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

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

Example

Global goal:

```
refines ?R (X + Y - (1 * Y)) ?P.
```

Current goal(s):

```
refines ?R (X + Y - (1 * Y)) ?P.
```

Example

Global goal:

```
refines ?R (X + Y - (1 * Y)) (?f ?P1).
```

Current goal(s):

```
refines (?S ==> ?R) (fun P => X + P) ?f,
refines ?S (Y - (1 * Y)) ?P1.
```

Example

Global goal:

```
refines ?R (X + Y - (1 * Y)) (?g ?P2 ?P1).
```

Current goal(s):

```
refines (?T ==> ?S ==> ?R) + ?g,
refines ?T X ?P2,
refines ?S (Y - (1 * Y)) ?P1.
```

Example

Global goal:

```
refines R (X + Y - (1 * Y)) (?P2 +' ?P1).
```

Assuming

```
refines (R ==> R ==> R) + +'.
```

Current goal(s):

```
refines R X ?P2,
refines R (Y - (1 * Y)) ?P1.
```

Example

Global goal:

```
refines R (X + Y - (1 * Y)) (X' +' ?P1).
```

Assuming

```
refines (R ==> R ==> R) + +',
refines R X X'.
```

Current goal(s):

```
refines R (Y - (1 * Y)) ?P1.
```

Example

Proven:

```
refines R (X + Y - (1 * Y)) (X' +' Y' -' (1' *' Y')).
```

Assuming

```
refines (R ==> R ==> R) + +',
refines (R ==> R ==> R) - -',
refines (R ==> R ==> R) * *',
refines R X X',
refines R Y Y',
refines R 1 1'.
```

Logic programming for refinement

Rules to decompose expressions, such as

```
Instance refines_apply
P C (R : P -> C -> Type) P’ C’ (R’ : P’ -> C’ -> Type) :
  forall (f : P -> P’) (g : C -> C’),
  refines (R ==> R’) f g ->
    forall (p : P) (c : C), refines R p c ->
      refines R’ (f p) (g c).

Lemma refines_trans P I C (rPI : P -> I -> Type)
(rIC : I -> C -> Type) (rPC : P -> C -> Type)
(p : P) (i : I) (c : C) :
  rPI o rIC <= rPC ->
    refines rPI p i -> refines rIC i c ->
      refines rPC p c.
```

Example

```
refines ?R (matrix_of_fun (fun i j => i + (i * j))) ?m
```

Example

```
refines ?R (matrix_of_fun (fun i j => i + (i * j))) ?m
```

Assume

```
refines Rord i i',
refines Rord j j'.
```

Global goal:

```
refines ?R (i + (i * j)) (?f i' j').
```

Current goal(s):

```
refines ?R (i + (i * j)) (?f i' j').
```

Example

```
refines ?R (matrix_of_fun (fun i j => i + (i * j))) ?m
```

Assume

```
refines Rord i i',
refines Rord j j'.
```

Global goal:

```
refines ?R (i + (i * j)) (?f i' j').
```

Current goal(s):

```
refines (?R' ==> ?R) (fun k => i + k) (?f i'),
refines ?R' (i * j) j'.
```

Example

```
refines ?R (matrix_of_fun (fun i j => i + (i * j))) ?m
```

Assume

```
refines Rord i i’,
refines Rord j j’.
```

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) (i + (i * j)) (?f i’ j’),
```

which splits into

```
  refines ?R (i + (i * j)) ?e,
  refines unify ?e (?f i’ j’).
```

Outline

1. CoqEAL's refinement framework

2. Automation

3. Applications

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.
```

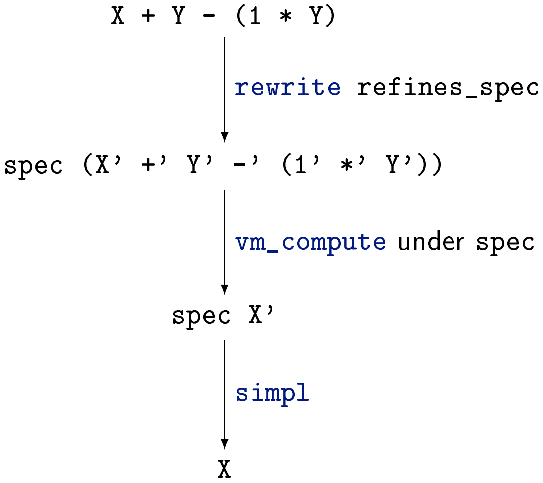
```
Lemma det_ctmat1 : \det ctmat1 = 2.
Proof. by coqeal. Qed.
```

or

```
Definition det_ctmat1 :=
  [coqeal vm_compute of \det ctmat1].
--> det_ctmat1 : \det ctmat1 = 2
```

About [coqeal vm_compute of _]

```
Lemma refines_spec R p c : refines R p c -> p = spec c.
```



X + Y - (1 * Y)

rewrite refines_spec

spec (X' +' Y' -' (1' *' Y'))

vm_compute under spec

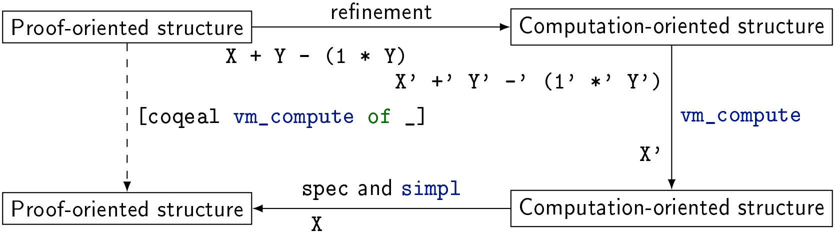
spec X'

simpl

X

About [coqeal vm_compute of _]

```
Lemma refines_spec R p c : refines R p c -> p = spec c.
```



Proof-oriented structure

refinement

Computation-oriented structure

`X + Y - (1 * Y)`

`X' +' Y' -' (1' *' Y')`

`[coqeal vm_compute of _]`

`vm_compute`

`X'`

spec and `simpl`

Proof-oriented structure

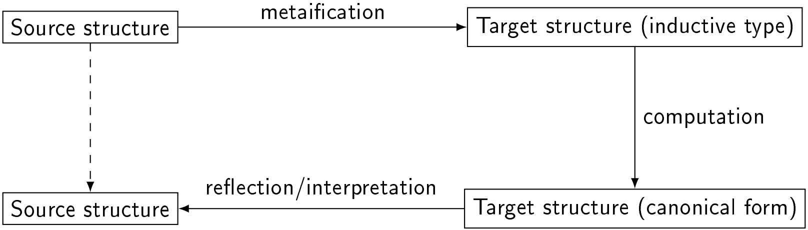
Computation-oriented structure

`X`

Proof by reflection

- Use computation to automate and to shorten proofs.
- Issue: ad-hoc computation-oriented data-structures and problem-specific implementations make it hard to maintain and improve reflection-based tactics.
- Our contribution: a modular reflection methodology that uses generic tools to minimise the code specific to a given tactic.
- Our example case:
  - The `ring` Coq tactic: a reflection-based tactic to reason modulo ring axioms (and a bit more).
  - Generic tools: the Mathematical Components library and CoqEAL refinement framework.
  - Code specific to our prototype: around 200 lines.

Reflection [Boutin 97]



Source structure

metaification

Target structure (inductive type)

computation

Target structure (canonical form)

reflection/interpretation

Source structure

Reflection on rings [Grégoire, Mahboubi 2005]

Metaification:
Symbolic arithmetic expressions in a ring (using `+`, `-`, `*` and `.^`^{n}) can be represented as multivariate polynomials over integers, together with a variable map.
`a + b - (1 * b)` ⟶ `X + Y - (1 * Y)` with variable map `[a; b]`.

Computation:
The goal of the computation step is to normalise the obtained polynomials.
`X + Y - (1 * Y)` ⟶ `X`.

Reflection:
The polynomials in normal form are evaluated on the variable map to get back ring expressions.
`X[a; b]` ⟶ `a`.

The ring of coefficients [Grégoire, Mahboubi 2005]

The ring of integers is a canonical choice since there is a canonical injection from integers to any ring: the ring of integers is an initial object of the category of rings.

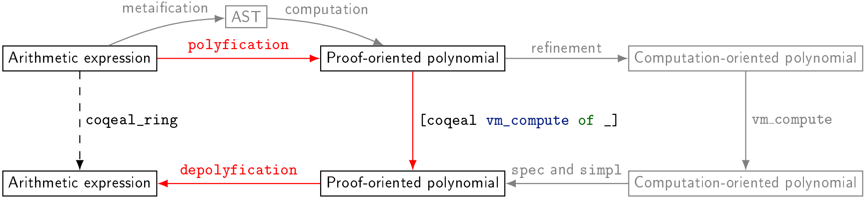
However it may happen that another ring ($\mathbb{Z}_{/n\mathbb{Z}}$, rational numbers...) is a better choice. For instance `a + a = 0` is provable in the ring of booleans, using the ring of booleans itself as the ring of coefficients.

$$\texttt{a + a} \longrightarrow \texttt{(X + X)[a]} \longrightarrow \texttt{((1 + 1)X)[a]} \longrightarrow \texttt{(0X)[a]} \longrightarrow \texttt{0.}$$

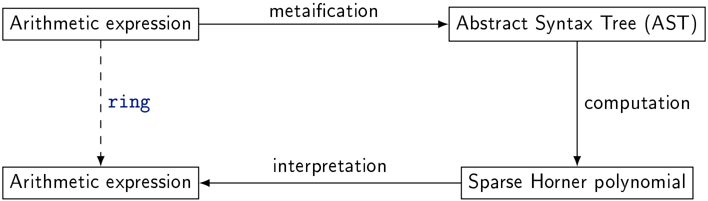
The coqeal_ring tactic [Cohen, Rouhling 2017]

$$\begin{tikzcd}
\texttt{a + b - (1 * b)} \arrow[d, "\text{rewriting}"] \\
\texttt{eval (X + Y - (1 * Y)) [a; b]} \arrow[d, "{\texttt{[coqeal vm\_compute of \_]}}"] \\
\texttt{eval X [a; b]} \arrow[d, "\text{rewriting}"] \\
\texttt{a}
\end{tikzcd}$$

The coqeal_ring tactic [Cohen, Rouhling 2017]



The `ring` tactic [Grégoire, Mahboubi 2005]



Arithmetic expression

metaification

Abstract Syntax Tree (AST)

computation

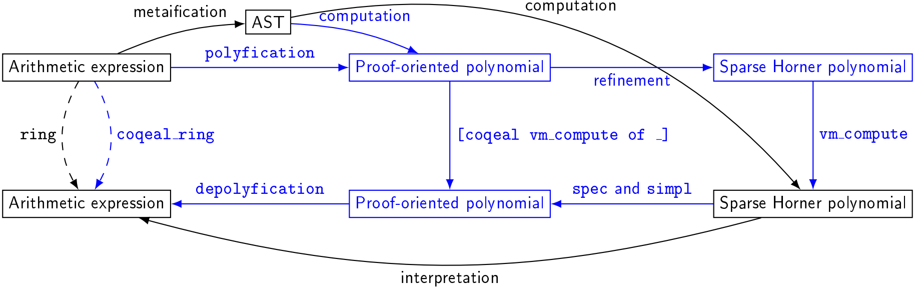
Sparse Horner polynomial

interpretation

Arithmetic expression

`ring`

Comparison



metaification

AST

computation

computation

Arithmetic expression

polyfication

Proof-oriented polynomial

refinement

Sparse Horner polynomial

ring

coqeal_ring

[coqeal vm_compute of _]

vm_compute

Arithmetic expression

depolyfication

Proof-oriented polynomial

spec and simpl

Sparse Horner polynomial

interpretation

Further work

- On `coqeal_ring`:
  - Catch up with `ring`: operations such as the power function, ring of coefficients as parameter, non-commutative rings, semi-rings...
  - Make `coqeal_ring` efficient: refinement of the translation AST → polynomial, improved `depolyfication`.
  - Implement new features: morphisms, Gröbner bases (Théry, using multivariate polynomials by Strub, and a refinement by Martin-Dorel, Roux), user-defined operations...
  - Generalise to other decision procedures: `field`? `lra`???
- On CoqEAL:
  - More refinements, especially outside algebra, e.g. finite sets (Dagand, Gallego Arias).
  - Improve CoqEAL's interface, e.g. a better debugging system.
  - Make refinement faster, in particular on nested structures.

Conclusion

- Efficient computations require proofs, refinement simplifies them.
- Proofs are automated by computations, reflection does that.
- Refinement is not so far from reflection.

Conclusion

- Efficient computations require proofs, refinement simplifies them.
- Proofs are automated by computations, reflection does that.
- Refinement is not so far from reflection.

Thank you!

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:

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

Correctness of addQ (end)

Goal:

$$\begin{array}{l}\texttt{(Rint * Rint ==> Rint * Rint ==> Rint * Rint)}\\ \quad\texttt{(addQ int }+_{\texttt{int}}\ *_{\texttt{int}}\texttt{) (addQ C }+_{\texttt{C}}\ *_{\texttt{C}}\texttt{).}\end{array}$$

By parametricity:

$$\begin{array}{l}[\![\forall \mathtt{Z}.\,(\mathtt{Z} \to \mathtt{Z} \to \mathtt{Z}) \to (\mathtt{Z} \to \mathtt{Z} \to \mathtt{Z}) \to \mathtt{Z} * \mathtt{Z} \to \mathtt{Z} * \mathtt{Z} \to \mathtt{Z} * \mathtt{Z}]\!]\ \texttt{addQ}\\ \qquad\texttt{addQ,}\end{array}$$

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').
```

Soundness of polyfication

```
Lemma polyficationP (R : comRingType) (env : seq R) N p : size env == N ->
  PExpr_to_Expr env p = Nhorner env (PExpr_to_poly N p).
Proof.
elim: p=> [n|n|p IHp q IHq|p IHp q IHq|p IHp|p IHp n] /=.
- by rewrite NhornerE !rmorph_int.
- rewrite NhornerE; elim: N env n=> [|N IHN] [|a env] [|n] //= senv.
    by rewrite map_polyX hornerX [RHS]NhornerRC.
  by rewrite map_polyC hornerC !IHN.
- by move=> senv; rewrite (IHp senv) (IHq senv) !NhornerE !rmorphD.
- by move=> senv; rewrite (IHp senv) (IHq senv) !NhornerE !rmorphM.
- by move=> senv; rewrite (IHp senv) !NhornerE !rmorphN.
- by move=> senv; rewrite (IHp senv) !NhornerE !rmorphX.
Qed.
```

Example of user-defined operation: factoring

$$\begin{tikzcd}
\boxed{e(x)} \arrow[r, "\text{polyfication}"] \arrow[d, dashed] & \boxed{P[x]} \arrow[d, "\text{theorem rewriting}"'] \\
\boxed{(x-a)e'(x)} & \boxed{((X-a)Q)[x]} \arrow[l, "\text{depolyfication}"']
\end{tikzcd}$$

Where $P[a] = 0$.