CoqEAL - The Coq Effective Algebra Library[1]

Cyril Cohen, Maxime Dénès and Anders Mörtberg

University of Gothenburg and Inria Sophia-Antipolis

February 3, 2013

[1] This work has been funded by the FORMATH project, nr. 243847, of the FET program within the 7th Framework program of the European Commission.

Motivation: verifying computer algebra algorithms

What for?

- Computer algebra algorithms can help automate proofs
- Formal proofs bridge the gap between paper correctness proofs and real-life implementations
- Proof assistants can provide independent verification of results obtained by computer algebra programs (e.g. $\zeta(3)$ is irrational, computation of homology groups)

Context

Traditional approaches to program verification:

- Bottom-up verification (e.g. annotations)
- Program synthesis from specifications (e.g. Coq's extractor)
- Top-down step-wise refinements from specification to programs

Specificity of computer algebra programs:

- Computer algebra algorithms can have complex specifications
- Efficiency matters!

Problem: these aspects are often in tension

Separation of concerns

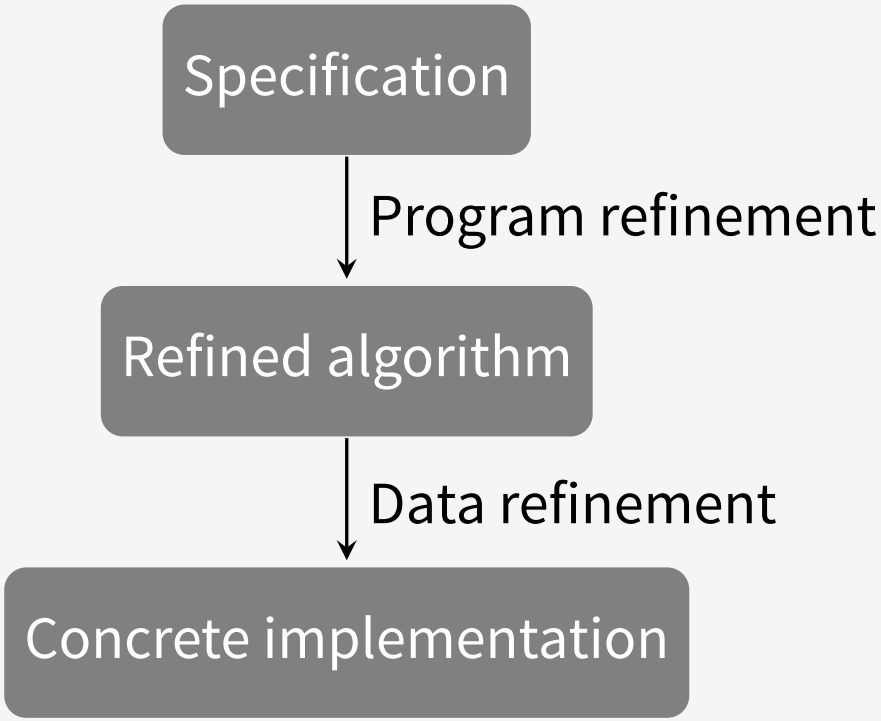
We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day, so to speak. […] But nothing is gained – on the contrary! – by tackling these various aspects simultaneously. It is what I sometimes have called "the separation of concerns"

Dijkstra, Edsger W.
"On the role of scientific thought" (1982)

Program and data refinements

We distinguish two kinds of refinements:

- Program refinement: improving the algorithms
- Data refinement: switching to more efficient data representation



Example: natural numbers in Coq standard lib

In the standard library of Coq: `nat` (unary) and `N` (binary) along with two isomorphisms `N.of_nat : nat -> N` and `N.to_nat : N -> nat`

Here already two aspects in tension:

- `nat` has a convenient induction scheme for proofs
- `N` gives an exponentially more compact representation of numbers

In the standard library, proofs are factored using an abstract axiomatization (module signature) and can be instantiated to these two implementations.

Traditional abstraction

Given a datatype T depending on types $\mathbf{B}$
and operators $\mathbf{f}$ using T and using operators $\mathbf{b}$ on $\mathbf{B}$.

1. Abstract the datatype T over $\mathbf{B}$ $\Longrightarrow$ D s.t. $\mathrm{D}\,\mathbf{B} = \mathsf{T}$
2. Abstract operators f over $\mathbf{B}$ and $\mathbf{b}$ $\Longrightarrow$ g s.t. $\mathrm{g}\,\mathbf{B}\,\mathbf{b} = \mathrm{f}$
3. Abstract theorems over $\mathbf{B}$ and $\mathbf{b}$ and over theorems about $\mathbf{b}$

- Program refinement by having two extensionally equal g_0 and g_1.
- Data instantiation, but how to change a datatype for another?

Achieving traditional abstraction

Ways to achieve it:

- modules (e.g. Coq stdlib), but they are very rigid,
- typeclasses (currently slow),
- canonical structures (currently not adapted to too many classes).

Achieving traditional abstraction

Ways to achieve it:

- modules (e.g. Coq stdlib), but they are very rigid,
- typeclasses (currently slow),
- canonical structures (currently not adapted to too many classes).

Issues and questions:

- how to change data representation? (from DB to DC)
- goes against the "small scale reflection" approach (following SSReflect)

Context: Libraries, Conventions, Examples

Proof-oriented types.
E.g.: `nat`, `int`, `rat`, `{poly R}`, `(matrix R)`...

Proof-oriented programs.
E.g.: `O`, `S`, `addn`, `addz`, ..., `0%R`, `1%R`, `(_+_)%R`...

Rich theory, geared towards interactive proving

Computation-oriented types.
E.g.: `N`, `Z`, `Q`, `sparse_poly`, ...

Computation-oriented programs.
E.g.: `xH`, `xI`, `xO`, `addN`, `addQ`, ..., `0%C`, `1%C`, `(_+_)%C`...

Reduced theory, more efficient data-structures and more efficient algorithms

Context: Libraries, Conventions, Examples

Proof-oriented types.
E.g.: `nat`, `int`, `rat`, `{poly R}`, `(matrix R)`...

Computation-oriented types.
E.g.: `N`, `Z`, `Q`, `sparse_poly`, ...

Proof-oriented programs.
E.g.: `O`, `S`, `addn`, `addz`, ..., `0%R`, `1%R`, `(_+_)%R`...

Computation-oriented programs.
E.g.: `xH`, `xI`, `xO`, `addN`, `addQ`, ..., `0%C`, `1%C`, `(_+_)%C`...

Rich theory, geared towards interactive proving

Reduced theory, more efficient data-structures and more efficient algorithms

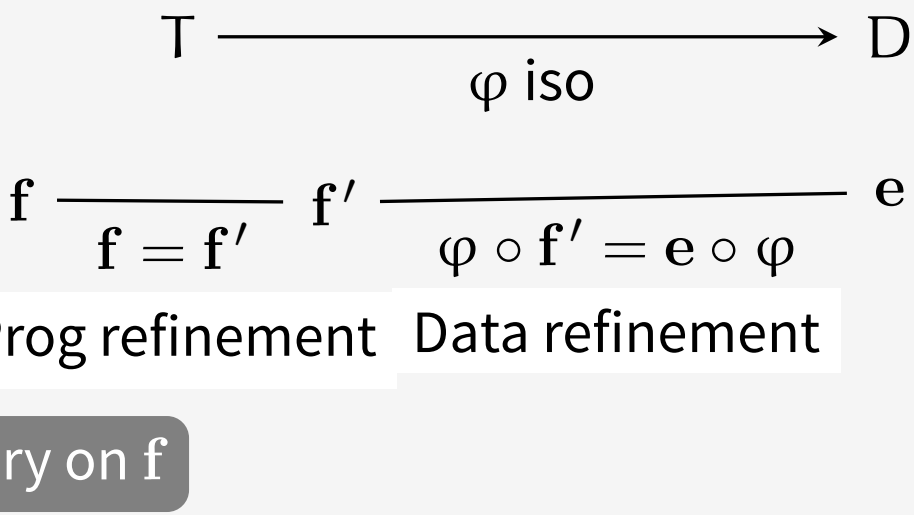
We suggest a methodology based on refinement from proof oriented types to computation oriented types to achieve separation of concerns.

ITP 2012 (Dénès, Mörtberg, Siles)

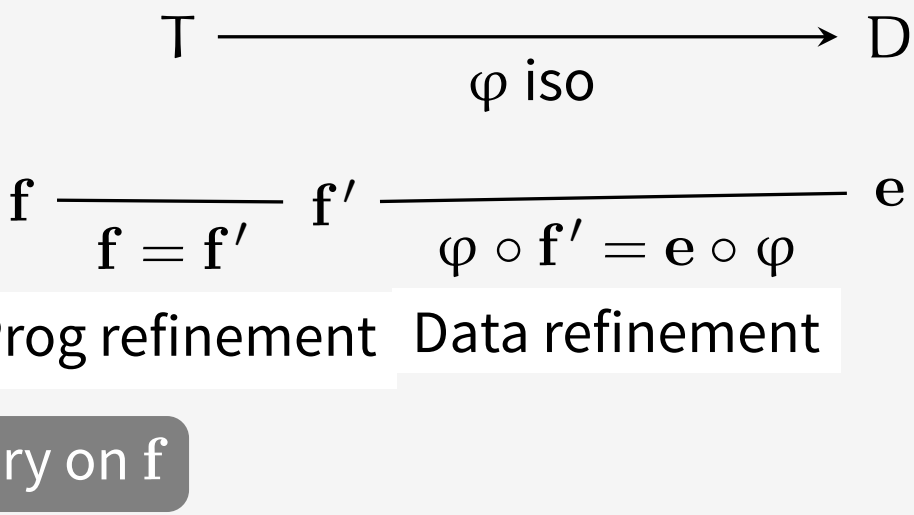
Assuming we have a theory about $\mathbf{f}$ on a type T:

1. write efficient algorithms $\mathbf{f}'$ for T,
2. build a type D for efficient computation,
3. prove that T and D are isomorphic,
4. duplicate the algorithms $\mathbf{f}'$ into $\mathbf{e}$ for D,
5. prove extensional equality of algorithms.

ITP 2012 (Dénès, Mörtberg, Siles)



ITP 2012 (Dénès, Mörtberg, Siles)



Issues:

- f' and e are duplicates,
- data refinements contain no mathematics,
  but long and time consuming to write down by hand,
- what if T and D are not isomorphic?

Non isomorphic types

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

The proof-oriented rat enforces that fractions are reduced

- Allows to use Leibniz equality in proofs
- This invariant is costly to maintain during computations

We would like to relax the constraint and express that `rat` is isomorphic to a quotient of a subset of pairs of integers.

Generic programming: addition over rationals

Generic datatype

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

Generic operations

```
Definition addQ Z `{add Z} `{mul Z} : add (Q Z) :=
  fun x y => (x.1 * y.2 + y.1 * x.2, x.2 * y.2).
```

To prove correctness of `addQ`, abstracted operators `(+ : add Z)` and `(* : mul Z)` are instantiated by proof-oriented definitions `(addz : add int)` and `(mulz : mul int)`.
When computing, these operators are instanciated to more efficient ones.

Proof-oriented correctness

- The type `int` is the proof-oriented version of integers.
- The type `rat` is the proof-oriented version of rationals.

Correctness of `addQ int`

```
Definition addQ Z `{add Z} `{mul Z} : add (Q Z) :=
  fun x y => (x.1 * y.2 + y.1 * x.2, x.2 * y.2).

Definition RQint : rat -> Q int -> Prop :=
  fun r q => Qint_to_rat q = r.

Lemma RQint_add :
  forall (x : rat) (u : Q int), RQint x u ->
  forall (y : rat) (v : Q int), RQint y v ->
  RQint (add_rat x y) (addQ u v).
```

Complete correctness

Correctness of addQ

```
Definition addQ Z `{add Z} `{mul Z} : add (Q Z) :=
  fun x y => (x.1 * y.2 + y.1 * x.2, x.2 * y.2).

Variables (Z : Type) (RZ : int -> Z -> Prop).
Definition RQint : rat -> Q int -> Prop := ...
Definition RQ := (RQint \o (RZ * RZ))%rel.

Lemma RQ_add `{add Z, mul Z} : [...] ->
  forall (x : rat) (u : Q Z), RQ x u ->
  forall (y : rat) (v : Q Z), RQ y v ->
  RQ (add_rat x y) (addQ u v).
```

Complete correctness

Correctness of addQ

```
Definition addQ Z `{add Z} `{mul Z} : add (Q Z) :=
  fun x y => (x.1 * y.2 + y.1 * x.2, x.2 * y.2).

Variables (Z : Type) (RZ : int -> Z -> Prop).
Definition RQint : rat -> Q int -> Prop := ...
Definition RQ := (RQint \o (RZ * RZ))%rel.

Lemma RQ_add `{add Z, mul Z} : [...] ->
  (RQ ==> RQ ==> RQ) add_rat (addQ (+) (*))
```

Complete correctness

Correctness of addQ

```
Definition addQ Z `{add Z} `{mul Z} : add (Q Z) :=
  fun x y => (x.1 * y.2 + y.1 * x.2, x.2 * y.2).

Variables (Z : Type) (RZ : int -> Z -> Prop).
Definition RQint : rat -> Q int -> Prop := ...
Definition RQ := (RQint \o (RZ * RZ))%rel.

Lemma RQ_add `{add Z, mul Z} :
  (RZ ==> RZ ==> RZ) addz (+) ->
  (RZ ==> RZ ==> RZ) mulz (*) ->
  (RQ ==> RQ ==> RQ) add_rat (addQ (+) (*))
```

Complete correctness

Correctness of addQ

```
Variables (Z : Type) (RZ : int -> Z -> Prop).
Definition RQint : rat -> Q int -> Prop := ...
Definition RQ := (RQint \o (RZ * RZ))%rel.

Lemma RQint_add :
  (RQint ==> RQint ==> RQint) add_rat (addQ addz mulz)

Lemma param_addQ `{add Z, mul Z} :
  (RZ ==> RZ ==> RZ) addz (+) ->
  (RZ ==> RZ ==> RZ) mulz (*) ->
  (RZ * RZ ==> RZ * RZ ==> RZ * RZ)
    (addQ addz mulz) (addQ (+) (*))
```

Complete correctness

Correctness of addQ

```
Variables (Z : Type) (RZ : int -> Z -> Prop).
Definition RQint : rat -> Q int -> Prop := ...
Definition RQ := (RQint \o (RZ * RZ))%rel.

Lemma RQint_add :
  (RQint ==> RQint ==> RQint) add_rat (addQ addz mulz)

Lemma param_addQ `{add Z, mul Z} :
  (RZ ==> RZ ==> RZ) addz (+) ->
  (RZ ==> RZ ==> RZ) mulz (*) ->
  (RZ * RZ ==> RZ * RZ ==> RZ * RZ)
    (addQ addz mulz) (addQ (+) (*))
```

- RQint_add is not for free,

Complete correctness

Correctness of addQ

```
Variables (Z : Type) (RZ : int -> Z -> Prop).
Definition RQint : rat -> Q int -> Prop := ...
Definition RQ := (RQint \o (RZ * RZ))%rel.

Lemma RQint_add :
  (RQint ==> RQint ==> RQint) add_rat (addQ addz mulz)

Lemma param_addQ `{add Z, mul Z} :
  (RZ ==> RZ ==> RZ) addz (+) ->
  (RZ ==> RZ ==> RZ) mulz (*) ->
  (RZ * RZ ==> RZ * RZ ==> RZ * RZ)
    (addQ addz mulz) (addQ (+) (*))
```

- RQint_add is not for free,
- but param_addQ should be a theorem for free! (by parametricity)

Parametricity

Parametricity for closed terms in

There is a translation operator : $[.]$, such that for a closed type T and a closed term $x : T$, we get $[x] : [T]xx$.

(Reynolds, Wadler in system F, Keller and Lasson for Coq)

Proof of `param_addQ`

$$[\forall Z, (Z \to Z \to Z) \to (Z \to Z \to Z) \to (Z^2 \to Z^2 \to Z^2)]\ \mathsf{addQ}\ \mathsf{addQ}$$

The new strategy

Assume

- we have type T we want to refine (e.g. `rat`),
- we have a theory about $\mathbf{f}$ (e.g. addition, embedding),
- we have refinements $\mathbf{B}$ (e.g. Z) of $\mathbf{A}$ (e.g. `int`) and refinements $\mathbf{b}$ (e.g. addition on Z) of $\mathbf{a}$ (e.g. addition on `int`).

We

1. build a type D for efficient computation,
2. write efficient algorithms g in a generic form,
3. prove f correct with regard to $\mathrm{g}\ \mathbf{A}\ \mathbf{a}$
4. get correctness for $\mathrm{g}\ \mathbf{B}\ \mathbf{b}$ by parametricity.

Using the framework

For many types (`nat`, `int`, `rat`, `matrix`, ...), there is a specification function, e.g.

```
specQ : Q Z -> rat
```

Such that

```
forall x y, RQ x y -> x = specQ y
```

(i.e. `specQ` is a refinement of the identity function)

Related work

- (Refinements for free!, (Cohen Dénès Mörtberg, CPP'13))
- (A refinement-based approach to computational algebra in Coq (Dénès Mörtberg Siles, ITP'12))
- A New Look at Generalized Rewriting in Type Theory (Sozeau, JFR'09)
- Automatic data refinements in ISABELLE/HOL (Lammich, ITP'13)
- Univalence: Isomorphism is equality (Coquand Danielsson, '13)
- Parametricity in an Impredicative Sort (Keller Lasson, CSL'12)

Applications and future work

- We applied it to algorithms we had previously verified: Karatsuba's polynomial multiplication, Strassen's matrix product,
- we are still porting others from the old framework: Sasaki-Murao algorithm, Smith normal form.

Future work:

- have a better way to get parametricity than typeclasses,
- try on algorithms outside algebra,
- scale up to dependent types,
- try to extract the efficient implementation.

Thanks!