Computable data refinements by quotients and parametricity[1]

Maxime Dénès, Cyril Cohen and Anders Mörtberg

Inria Sophia-Antipolis and University of Gothenburg

April 24, 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

Motivation

Verifying computer algebra algorithms

What for?

Motivation

Verifying computer algebra algorithms

What for?

- Computer algebra algorithms can help automate proofs

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

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:

Context

Traditional approaches to program verification:

- Bottom-up verification (e.g. annotations)

Context

Traditional approaches to program verification:

- Bottom-up verification (e.g. annotations)
- Program synthesis from specifications (e.g. Coq's extractor)

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

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!

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
We suggest a methodology based on refinements to achieve separation of concerns

Separation of concerns

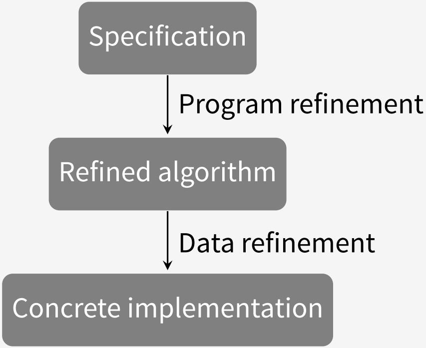
We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day, so to speak. [...] But nothing is gained – on the contrary! – by tackling these various aspects simultaneously. It is what I sometimes 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 algorithmics
- Data refinement: switching to more efficient data representation



Isomorphic structures

First example: natural numbers

Isomorphic structures

First example: natural numbers

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`

Isomorphic structures

First example: natural numbers

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

Isomorphic structures

First example: natural numbers

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

Isomorphic structures

First example: natural numbers

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) instanciated to these two implementations.

Isomorphic structures

First example: natural numbers

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) instanciated to these two implementations.

Pb: this goes against the "small scale reflection" approach (following SSReflect)

Partial operators

Second example: polynomials in SSReflect

```
Variable R : ringType.
Record polynomial :=
  Polynomial {polyseq :> seq R; _ : last 1 polyseq != 0}.
```

Partial operators

Second example: polynomials in SSReflect

```
Variable R : ringType.
Record polynomial :=
  Polynomial {polyseq :> seq R; _ : last 1 polyseq != 0}.
```

For computations, we drop the proof component and see polynomials as lists (sequences).

Partial operators

Second example: polynomials in SSREFLECT

```
Variable R : ringType.
Record polynomial :=
  Polynomial {polyseq :> seq R; _ : last 1 polyseq != 0}.
```

For computations, we drop the proof component and see polynomials as lists (sequences).

Our proof-oriented type `polynomial` is isomorphic to a subset of `(seq R)`.

Partial operators

Second example: polynomials in SSReflect

```
Variable R : ringType.
Record polynomial :=
  Polynomial {polyseq :> seq R; _ : last 1 polyseq != 0}.
```

For computations, we drop the proof component and see polynomials as lists (sequences).

Our proof-oriented type `polynomial` is isomorphic to a subset of `(seq R)`.

Operators over `(seq R)` are partially specified as refinements of their counterparts from `(polynomial R)`.

Quotient

Third example: rational numbers

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

Quotient

Third example: rational numbers

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

The proof-oriented rat enforces that fractions are reduced

Quotient

Third example: rational numbers

```
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

Quotient

Third example: rational numbers

```
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

Quotient

Third example: rational numbers

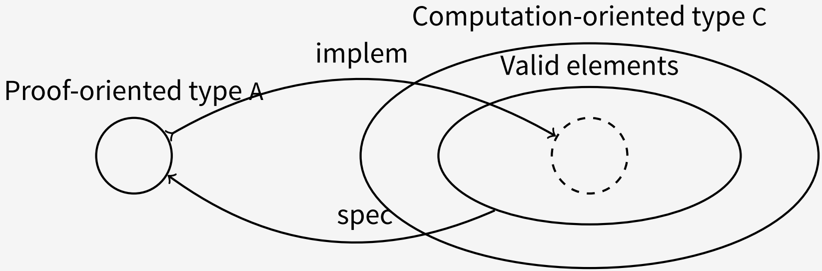
```
Record rat : Set := Rat {
  valq : (int * int) ;
  _ : (0 < valq.2) && coprime `|valq.1| `|valq.2|
}.
```

The proof-oriented rat enforces that fractions are reduced

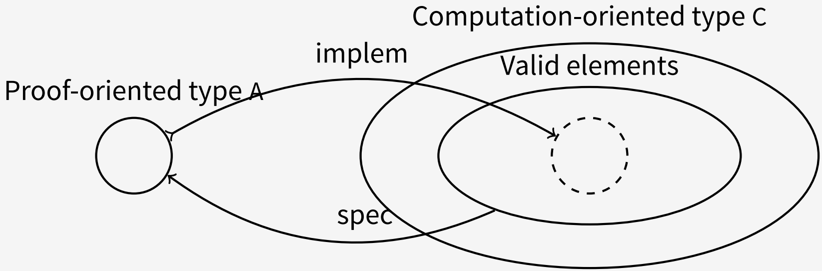
- Allows to use Leibniz equality in proofs
- This invariant is costly to maintain during computations

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

Interface for refinements



Interface for refinements




```
Class refinement A C := Refinement {
  implem : A -> C;
  spec : C -> option A;
  implemK : forall x : A, spec (implem x) = Some x
}.
```

Expressing a refinement

```
Class refines {A C} `{refinement A C} (a : A) (b : C) :=
  spec_refines_def : Some a = spec b.
```

Expressing a refinement

```
Class refines {A C} `{refinement A C} (a : A) (b : C) :=
  spec_refines_def : Some a = spec b.
```

Addition over N refines the one over `nat`:

```
Lemma refines_add (m n : nat) (u v : N) : refines m u ->
  refines n v -> refines (addn m n) (N.add u v)
```

Adding new rules to the game

- Generic programming: only one description of the algorithm, then specialized for proofs or computations

Adding new rules to the game

- Generic programming: only one description of the algorithm, then specialized for proofs or computations
- Compositionality: refining `(polynomial R)` to `(seq R)`, what is `R`?

Adding new rules to the game

- Generic programming: only one description of the algorithm, then specialized for proofs or computations
- Compositionality: refining `(polynomial R)` to `(seq R)`, what is `R`?
- Automating correctness proofs when changing representations

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`, operators `(+ : add Z)` and `(* : mul Z)` are instanciated to proof-oriented definitions.
When computing, these operators are instanciated to more efficient ones.

Compositionality

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).

Lemma refines_addQ Z ‘{refinement int Z, add Z, mul Z} :
  [...] ->
  forall (x y : rat) (u v : Q Z), refines m u ->
  refines n v -> refines (addq m n) (addQ u v).
```

This will be provable as soon as the addition and multiplication over `Z` refines the ones over `int`.
Hence, refinements are composable: for any `Z` refining `int`, `(Q Z)` refines `rat` (with associated operators).

Automation

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).

Lemma refines_addQ Z ‘{refinement int Z, add Z, mul Z} :
  [...] ->
  forall (x y : rat) (u v : Q Z), refines m u ->
  refines n v -> refines (addq m n) (addQ u v).
```

Automation

Correctness of addQ

```
Class param A B (R : A -> B -> Prop) (m : A) (n : B) :=
  param_rel : R m n.

Lemma refines_addQ Z `{refinement int Z, add Z, mul Z} :
  [...] ->
  forall (x y : rat) (u v : Q Z), refines m u ->
  refines n v -> refines (addq m n) (addQ u v).
```

Automation

Correctness of addQ

```
Class param A B (R : A -> B -> Prop) (m : A) (n : B) :=
  param_rel : R m n.

Lemma refines_addQ Z `{refinement int Z, add Z, mul Z} :
  param (refines ==> refines ==> refines) addz (+) ->
  param (refines ==> refines ==> refines) mulz (*) ->
  param (refines ==> refines ==> refines) addq addQ
```

Automation

Correctness of addQ

```
Class param A B (R : A -> B -> Prop) (m : A) (n : B) :=
  param_rel : R m n.

Lemma refines_addQ_int :
  param (refines ==> refines ==> refines)
    addq (@addQ int addz mulz)

Lemma refines_addQ Z `{refinement int Z, add Z, mul Z} :
  param (refines ==> refines ==> refines) addz (+) ->
  param (refines ==> refines ==> refines) mulz (*) ->
  param (refines ==> refines ==> refines) addq addQ
```

Automation

Correctness of addQ

```
Class param A B (R : A -> B -> Prop) (m : A) (n : B) :=
  param_rel : R m n.

Lemma refines_addQ_int :
  param (refines ==> refines ==> refines)
    addq (@addQ int addz mulz)

Lemma refines_addQ Z ‘{refinement int Z, add Z, mul Z} :
  param (refines ==> refines ==> refines) addz (+) ->
  param (refines ==> refines ==> refines) mulz (*) ->
  param (refines ==> refines ==> refines) addq addQ
```

- We prove `refines_addQ_int` manually

Automation

Correctness of addQ

```
Class param A B (R : A -> B -> Prop) (m : A) (n : B) :=
  param_rel : R m n.

Lemma refines_addQ_int :
  param (refines ==> refines ==> refines)
    addq (@addQ int addz mulz)

Lemma refines_addQ Z `{refinement int Z, add Z, mul Z} :
  param (refines ==> refines ==> refines) addz (+) ->
  param (refines ==> refines ==> refines) mulz (*) ->
  param (refines ==> refines ==> refines) addq addQ
```

- We prove `refines_addQ_int` manually
- Then we deduce `refines_addQ` by meta-programmming

Parametricity of addQ

```
  Z : Type
  _ : refinement int Z
  addZ : add Z
  mulZ : mul Z
  _ : param (refines ==> refines ==> refines) addz (+)
  _ : param (refines ==> refines ==> refines) mulz (*)
=====================================================
param (refines ==> refines ==> refines)
  addq (@addQ Z (+) (*))
```

Parametricity of addQ

```
   Z : Type
   _ : refinement int Z
   addZ : add Z
   mulZ : mul Z
   _ : param (refines ==> refines ==> refines) addz (+)
   _ : param (refines ==> refines ==> refines) mulz (*)
============================================================
param (refines ==> refines ==> refines)
  addq (@addQ int addz mulz)


============================================================
param (refines ==> refines ==> refines)
  (@addQ int addz mulz) (@addQ Z (+) (*))
```

Parametricity of addQ

```
   Z : Type
   _ : refinement int Z
   addZ : add Z
   mulZ : mul Z
   _ : param (refines ==> refines ==> refines) addz (+)
   _ : param (refines ==> refines ==> refines) mulz (*)
================================================
param (refines ==> refines ==> refines)
  (@addQ int addz mulz) (@addQ Z (+) (*))
```

Parametricity of addQ

```
   Z : Type
   _ : refinement int Z
   addZ : add Z
   _ : param (refines ==> refines ==> refines) addz (+)
=============================================================
param ((refines ==> refines ==> refines) ==>
      refines ==> refines ==> refines)
  (@addQ int addz) (@addQ Z (+))
```

Parametricity of addQ

```
  Z : Type
  _ : refinement int Z
=============================================================
param ((refines ==> refines ==> refines) ==>
      (refines ==> refines ==> refines) ==>
       refines ==> refines ==> refines)
  (@addQ int) (@addQ Z)
```

Conclusion and ongoing work

The approach we described:

- Reconciles convenient proofs with efficient computations
- Provides a mechanism to smoothly switch from one world to the other
- Avoids duplication of code

We are currently:

- Applying it to a variety of data structures (polynomials, matrices) supporting algorithms we had previously verified: Karatsuba's polynomial multiplication, Strassen's matrix product, Sasaki-Murao algorithm
- Polishing technical details to improve performance of proof search and efficiency of the generic code

Thanks!