Refinements for free! and applications

30/01/2019

Cyril Cohen – Seminari Cuc

1

Motivation

Strassen's algorithm, Winograd variant

[Dénès, Mörtberg, Silès 2012]

Matrix product can be programmed efficiently:

$$\left(\begin{array}{c|c} A_{1,1} & A_{1,2} \\ \hline A_{2,1} & A_{2,2} \end{array}\right) \times \left(\begin{array}{c|c} B_{1,1} & B_{1,2} \\ \hline B_{2,1} & B_{2,2} \end{array}\right) = \left(\begin{array}{c|c} C_{1,1} & C_{1,2} \\ \hline C_{2,1} & C_{2,2} \end{array}\right)$$

Strassen's algorithm, Winograd variant

[Dénès, Mörtberg, Silès 2012]

Matrix product can be programmed efficiently:

$$\left(\begin{array}{c|c} A_{1,1} & A_{1,2} \\ \hline A_{2,1} & A_{2,2} \end{array}\right) \times \left(\begin{array}{c|c} B_{1,1} & B_{1,2} \\ \hline B_{2,1} & B_{2,2} \end{array}\right) = \left(\begin{array}{c|c} C_{1,1} & C_{1,2} \\ \hline C_{2,1} & C_{2,2} \end{array}\right)$$

$$\begin{array}{lll}
S_1 = A_{2,1} + A_{2,2} & P_1 = A_{1,1} \times B_{1,1} & U_1 = P_1 + P_6 \\
S_2 = S_1 - A_{1,1} & P_2 = A_{1,2} \times B_{2,1} & U_2 = U_1 + P_7 \\
S_3 = A_{1,1} - A_{2,1} & P_3 = S_4 \times B_{2,2} & U_3 = U_1 + P_5 \\
S_4 = A_{1,2} - S_2 & P_4 = A_{2,2} \times T_4 & C_{1,1} = P_1 + P_2 \\
T_1 = B_{1,2} - B_{1,1} & P_5 = S_1 \times T_1 & C_{1,2} = U_3 + P_3 \\
T_2 = B_{2,2} - T_1 & P_6 = S_2 \times T_2 & C_{2,1} = U_2 - P_4 \\
T_3 = B_{2,2} - B_{1,2} & P_7 = S_3 \times T_3 & C_{2,2} = U_2 + P_5 \\
T_4 = T_2 - B_{2,1} & &
\end{array}$$

Strassen's algorithm, Winograd variant

[Dénès, Mörtberg, Silès 2012]

Matrix product can be programmed efficiently:

$$\left(\begin{array}{c|c} A_{1,1} & A_{1,2} \\ \hline A_{2,1} & A_{2,2} \end{array}\right) \times \left(\begin{array}{c|c} B_{1,1} & B_{1,2} \\ \hline B_{2,1} & B_{2,2} \end{array}\right) = \left(\begin{array}{c|c} C_{1,1} & C_{1,2} \\ \hline C_{2,1} & C_{2,2} \end{array}\right)$$

$$T(2^{k+1}) = 7T(2^k) + 15 \times 2^{2k}$$

$$T(n) = \mathcal{O}(n^{\log 7})$$

Strassen's algorithm, Winograd variant

[Dénès, Mörtberg, Silès 2012]

Matrix product can be programmed efficiently:

$$\left(\begin{array}{c|c} A_{1,1} & A_{1,2} \\ \hline A_{2,1} & A_{2,2} \end{array}\right) \times \left(\begin{array}{c|c} B_{1,1} & B_{1,2} \\ \hline B_{2,1} & B_{2,2} \end{array}\right) = \left(\begin{array}{c|c} C_{1,1} & C_{1,2} \\ \hline C_{2,1} & C_{2,2} \end{array}\right)$$

$$\begin{array}{lll}
S_1 = A_{2,1} + A_{2,2} & P_1 = A_{1,1} \times B_{1,1} & U_1 = P_1 + P_6 \\
S_2 = S_1 - A_{1,1} & P_2 = A_{1,2} \times B_{2,1} & U_2 = U_1 + P_7 \\
S_3 = A_{1,1} - A_{2,1} & P_3 = S_4 \times B_{2,2} & U_3 = U_1 + P_5 \\
S_4 = A_{1,2} - S_2 & P_4 = A_{2,2} \times T_4 & C_{1,1} = P_1 + P_2 \\
T_1 = B_{1,2} - B_{1,1} & P_5 = S_1 \times T_1 & C_{1,2} = U_3 + P_3 \\
T_2 = B_{2,2} - T_1 & P_6 = S_2 \times T_2 & C_{2,1} = U_2 - P_4 \\
T_3 = B_{2,2} - B_{1,2} & P_7 = S_3 \times T_3 & C_{2,2} = U_2 + P_5 \\
T_4 = T_2 - B_{2,1} & &
\end{array}$$

How would you prove this product is associative?

Program verification

Traditional approaches to program verification:

- Bottom-up verification (e.g. annotations/direct proof)
- Program synthesis from specifications (e.g. Coq's extraction)
- 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

Solution: 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)

Methodology: algorithm refinement

1. Write naive matrix product $c_{i,j} = \sum_{k<n} a_{i,k} b_{k,j}$.
2. Prove it is associative.
3. Prove that the two products are the same.

End of the story?

Methodology: algorithm refinement

1. Write naive matrix product $c_{i,j} = \sum_{k<n} a_{i,k} b_{k,j}$.
2. Prove it is associative.
3. Prove that the two products are the same.

End of the story?

No!

We need to be generic over data representation.

$\Rightarrow$ Abstraction / Parametrization

2

Coq, Mathematical Components, and abstraction

Formal proof and Coq

Tools

- Write definitions, theorems and proofs that can be mechanically checked.
- Numerous tools: Coq, Matita, Agda, Isabelle/HOL, HOL Light, PVS, Nuprl, ACL2, Mizar...

Coq

- **Interactive** Theorem Prover.
- Tactic langage to build proofs.
- Functional programming language.
- Based on type theory.

Abstraction in Coq

In Coq, abstraction using:

- The module system, or
- Records (+ typeclass-like inference)

Abstract data is characterized by

- Types
- Operations signature
- Axioms

$$\forall M : \left\{ \begin{array}{l} (A : \mathrm{Type}), \\ (* : A \to A \to A), \\ (*\mathrm{assoc} : \forall a\ b\ c, a * (b * c) = (a * b) * c \end{array} \right\}, \quad \mathrm{My\ Theory}(M)$$

Example: natural numbers in Coq standard lib

In Coq standard library:
`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 Coq standard library, proofs are factored using abstraction with the module system and can be instantiated to any of these two implementations.

$\longrightarrow$ The axioms of natural numbers are instantiated twice

Problem with traditional abstraction

We often have concrete constructions e.g. $\mathbb{N}$, matrices, polynomials,. . .

Should everything concrete be abstracted?

- Many abstractions with only one implementation.
- Difficult to find the right set of axioms to delimit an interface.
- Lose computational behaviour.

Mathematical Components

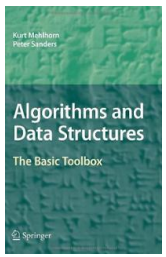
A library for Algebra in Coq [Gonthier et al 2006-2019]

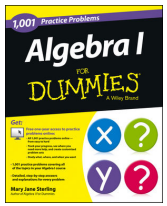
Origins:

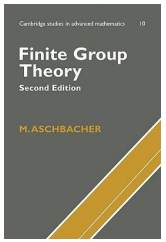
- Created for the Four Colour Theorem [Gonthier 2006] and
- Extended for the Odd Order Theorem [Gonthier et al 2013]

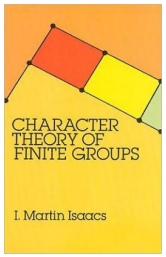
Contents:

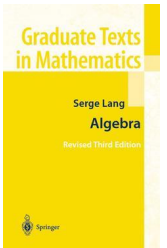
- Containers, Basic datatypes, Elementary arithmetic, ...
- Finite graph theory, depth-first search, ...
- Finite group theory, Representation theory, Character theory, ...
- Algebraic structures, Linear Algebra, Galois Theory, ...
- ...

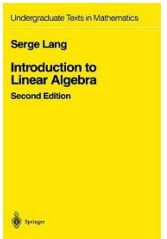


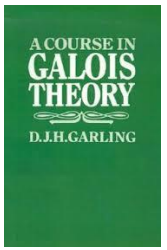












...

Context: Libraries, Conventions, Examples

Proof-oriented types.
E.g.: `nat`, `int`, `rat`, `{poly R}`, `'M[R]_(m,n)`...

Proof-oriented programs.
E.g.: `0`, `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}`, `'M[R]_(m,n)`...

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

Proof-oriented programs.
E.g.: `0`, `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 objects to computation oriented objects to achieve separation of concerns.

3

Refinements

Refinement [Dénès, Mörtberg, Silès 2012]

Sequence of refinement steps

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

where:

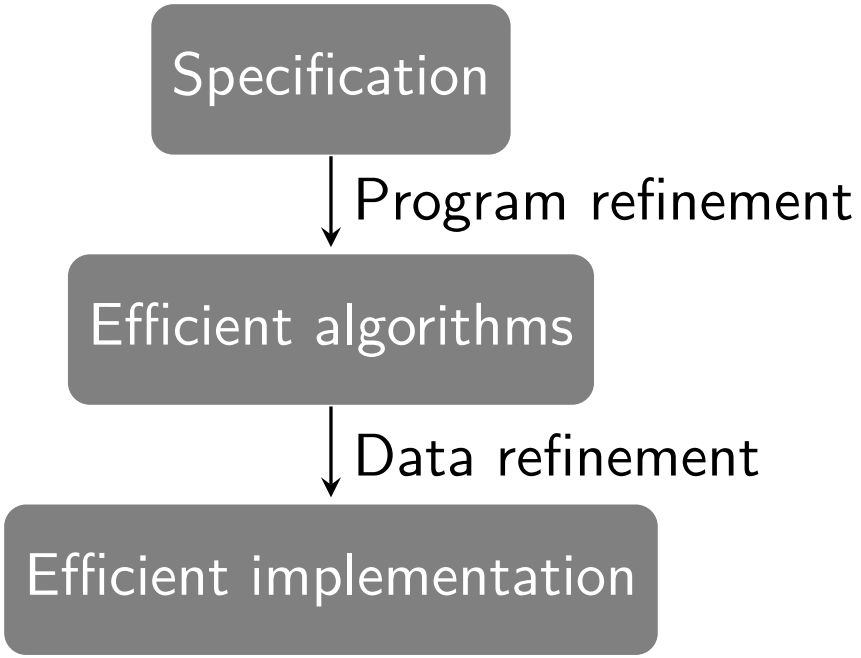
In the literature	In CoqEAL
• P_1 is an abstract version of the program,	• P_1 is an proof-oriented version of the program,
• P_n is a concrete version of the program,	• P_n is a computation-oriented version of the program,

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

Program and data refinements

The methodology consists in refining in two steps

1. Program refinement: improving the algorithms *without changing the data structures*.
2. Data refinement: switching to more efficient data representations, *using the same algorithm*.



Data refinements

Program refinement is complicated by essence

Data refinements should be automated:

- User provides primitive operations,
- explains compositionality,
- write **generic** programs,
- gives **no abstract properties** as in traditional abstraction.

Generic programming example: addition on $\mathbb{Q}$

Generic datatype

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

Generic operations

```
Definition addQ 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 instantiated 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 (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).
```

Correctness of `addQ`

```
Definition addQ 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 (+) (*) : [...] ->
  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).
```

Correctness of `addQ`

```
Definition addQ 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.

Notation "R ==> S" :=
  fun f g => forall x y, R x y -> S (f x) (g y).

Lemma RQ_add (+) (*) : [...] ->
  (RQ ==> RQ ==> RQ) add_rat (addQ (+) (*))
```

Correctness of `addQ`

```
Definition addQ 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.

Notation "R ==> S" :=
  fun f g => forall x y, R x y -> S (f x) (g y).

Lemma RQ_add (+) (*) :
  (RZ ==> RZ ==> RZ) addz (+) ->
  (RZ ==> RZ ==> RZ) mulz (*) ->
  (RQ ==> RQ ==> RQ) add_rat (addQ (+) (*))
```

Correctness of `addQ`

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

Notation "R ==> S" :=
  fun f g => forall x y, R x y -> S (f x) (g y).

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

Lemma refine_addQ (+) (*) :
(RZ ==> RZ ==> RZ) addz (+) -> (RZ ==> RZ ==> RZ) mulz (*) ->
(RZ * RZ ==> RZ * RZ ==> RZ * RZ)
  (addQ addz mulz) (addQ (+) (*))
```

Correctness of `addQ`

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

Notation "R ==> S" :=
  fun f g => forall x y, R x y -> S (f x) (g y).

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

Lemma param_addQ :
  ((RZ ==> RZ ==> RZ) ==> (RZ ==> RZ ==> RZ) ==>
   (RZ * RZ ==> RZ * RZ ==> RZ * RZ)) addQ addQ
```

Correctness of `addQ`

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

Notation "R ==> S" :=
  fun f g => forall x y, R x y -> S (f x) (g y).

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

Lemma param_addQ :
  ((RZ ==> RZ ==> RZ) ==> (RZ ==> RZ ==> RZ) ==>
   (RZ * RZ ==> RZ * RZ ==> RZ * RZ)) addQ addQ
```

- The proof of `RQint_add` is interesting
- The proof of `param_addQ` is boring

Correctness of `addQ`

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

Notation "R ==> S" :=
  fun f g => forall x y, R x y -> S (f x) (g y).

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

Lemma param_addQ :
  ((RZ ==> RZ ==> RZ) ==> (RZ ==> RZ ==> RZ) ==>
   (RZ * RZ ==> RZ * RZ ==> RZ * RZ)) addQ addQ
```

- The proof of `RQint_add` is interesting
- The proof of `param_addQ` is for free!

Parametricity / “[my result] for free!”

[Reynolds 1983, Wadler 1989, Keller-Lasson 2012]

A parametric polymorphic function cannot inspect its arguments!

Parametricity / "[my result] for free!"

[Reynolds 1983, Wadler 1989, Keller-Lasson 2012]

A parametric polymorphic function cannot inspect its arguments!

Parametricity for closed terms

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)

Free proof of `param_addQ`

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

Using refinements [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

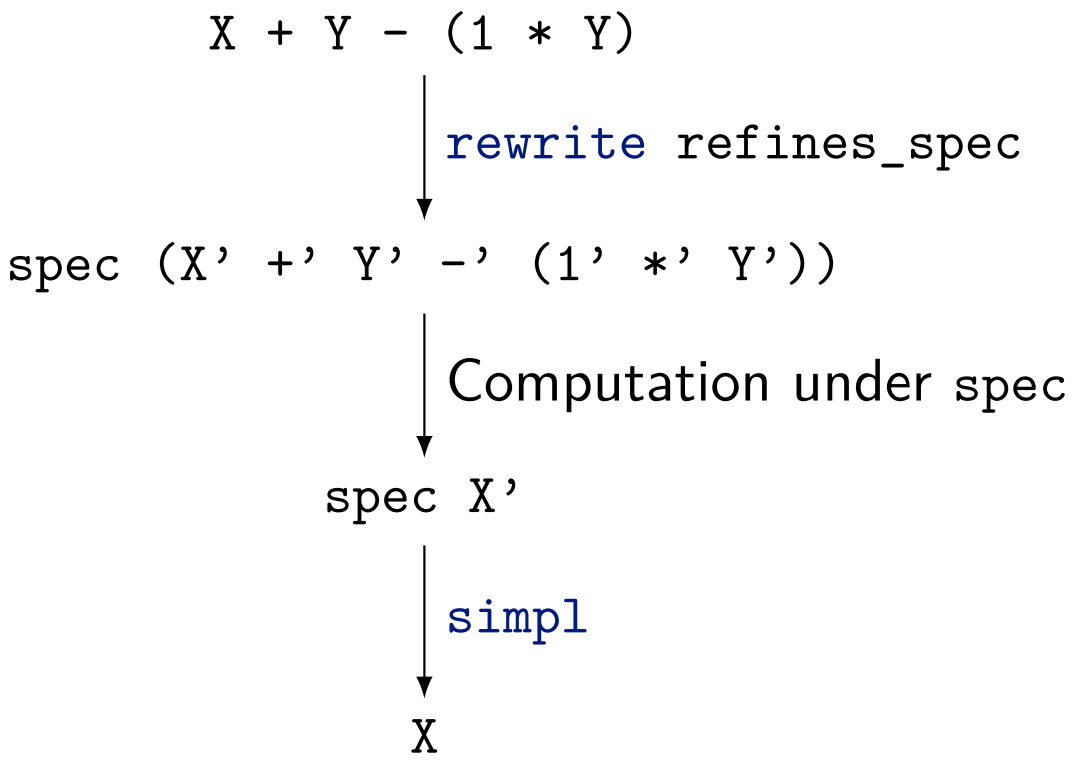
```
?proof : refines ?relation input ?output.
```

Back and forth translation:

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

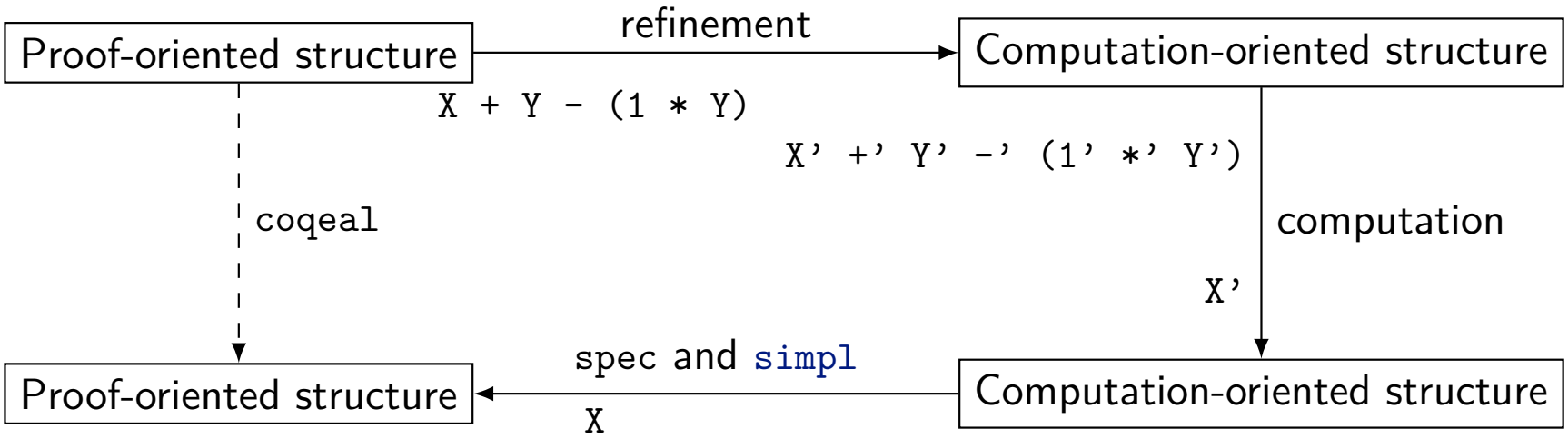
The coqeal tactic

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



The coqeal tactic

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



4

Extracting linear algebra using refinements

Strassen's algorithm, Winograd variant

[Dénès, Mörtberg, Silès 2012]

Matrix product can be programmed efficiently:

$$\left(\begin{array}{c|c} A_{1,1} & A_{1,2} \\ \hline A_{2,1} & A_{2,2} \end{array}\right) \times \left(\begin{array}{c|c} B_{1,1} & B_{1,2} \\ \hline B_{2,1} & B_{2,2} \end{array}\right) = \left(\begin{array}{c|c} C_{1,1} & C_{1,2} \\ \hline C_{2,1} & C_{2,2} \end{array}\right)$$

$$\begin{array}{lll}
S_1 = A_{2,1} + A_{2,2} & P_1 = A_{1,1} \times B_{1,1} & U_1 = P_1 + P_6 \\
S_2 = S_1 - A_{1,1} & P_2 = A_{1,2} \times B_{2,1} & U_2 = U_1 + P_7 \\
S_3 = A_{1,1} - A_{2,1} & P_3 = S_4 \times B_{2,2} & U_3 = U_1 + P_5 \\
S_4 = A_{1,2} - S_2 & P_4 = A_{2,2} \times T_4 & C_{1,1} = P_1 + P_2 \\
T_1 = B_{1,2} - B_{1,1} & P_5 = S_1 \times T_1 & C_{1,2} = U_3 + P_3 \\
T_2 = B_{2,2} - T_1 & P_6 = S_2 \times T_2 & C_{2,1} = U_2 - P_4 \\
T_3 = B_{2,2} - B_{1,2} & P_7 = S_3 \times T_3 & C_{2,2} = U_2 + P_5 \\
T_4 = T_2 - B_{2,1} & &
\end{array}$$

```
Definition Strassen_step p (A B : 'M_(p + p)) f :=
  let A11 := ulsubmx A in let A12 := ursubmx A in
  let A21 := dlsubmx A in let A22 := drsubmx A in
  let B11 := ulsubmx B in let B12 := ursubmx B in
  let B21 := dlsubmx B in let B22 := drsubmx B in
  let X := A11 - A21 in let Y := B22 - B12 in
  let C21 := f X Y in let X := A21 + A22 in
  let Y := B12 - B11 in let C22 := f X Y in
  let X := X - A11 in let Y := B22 - Y in
  let C12 := f X Y in let X := A12 - X in
  let C11 := f X B22 in let X := f A11 B11 in
  let C12 := X + C12 in let C21 := C12 + C21 in
  let C12 := C12 + C22 in let C22 := C21 + C22 in
  let C12 := C12 + C11 in let Y := Y - B21 in
  let C11 := f A22 Y in let C21 := C21 - C11 in
  let C11 := f A12 B21 in let C11 := X + C11 in
  block_mx C11 C12 C21 C22.
```

Correctness of `Strassen_step`

We prove the correctness of `Strassen_step` relatively to the matrix product `*m` defined in SSReflect.

```
Lemma Strassen_stepP p (A B : 'M[R]_(p + p)) f :
  f =2 mulmx -> Strassen_step A B f = A *m B.
```

Correctness of `Strassen_step`

We prove the correctness of `Strassen_step` relatively to the matrix product `*m` defined in SSReflect.

```
Lemma Strassen_stepP p (A B : 'M[R]_(p + p)) f :
  f =2 mulmx -> Strassen_step A B f = A *m B.
```

Then we define a function `Strassen` which, if applied to an even-sized matrix, cuts it in two submatrices `A` and `B` and calls recursively `Strassen_step A B Strassen`.

Correctness of `Strassen_step`

We prove the correctness of `Strassen_step` relatively to the matrix product `*m` defined in SSReflect.

```
Lemma Strassen_stepP p (A B : 'M[R]_(p + p)) f :
  f =2 mulmx -> Strassen_step A B f = A *m B.
```

Then we define a function `Strassen` which, if applied to an even-sized matrix, cuts it in two submatrices `A` and `B` and calls recursively `Strassen_step A B Strassen`.

What about odd-sized matrices?

The case of odd sizes

$$\left[\begin{array}{c|c} A_{1,1} & A_{1,2} \\ \hline A_{2,1} & a \end{array}\right] \times \left[\begin{array}{c|c} B_{1,1} & B_{1,2} \\ \hline B_{2,1} & b \end{array}\right]$$

$$= \left[\begin{array}{c|c} A_{1,1}B_{1,1} + A_{1,2}B_{2,1} & R_{1,2} \\ \hline R_{2,1} & R_{2,2} \end{array}\right]$$

with:

$$R_{1,2} = A_{1,1}B_{1,2} + A_{1,2}b$$
$$R_{2,1} = A_{2,1}B_{1,1} + aB_{2,1}$$
$$R_{2,2} = A_{2,1}B_{1,2} + ab$$

Final function

We obtain a function `Strassen` which we prove correct:

```
Lemma StrassenP (n : positive) (M N : 'M[R]_n) :
  Strassen M N = M *m N.
```

Final function

We obtain a function `Strassen` which we prove correct:

```
Lemma StrassenP (n : positive) (M N : 'M[R]_n) :
  Strassen M N = M *m N.
```

In fact, `Strassen_step` and `Strassen` were generic.

So we get for free an instance on `seqmatrix C`, for any `C` refining a ring.

Final function

We obtain a function `Strassen` which we prove correct:

```
Lemma StrassenP (n : positive) (M N : 'M[R]_n) :
  Strassen M N = M *m N.
```

In fact, `Strassen_step` and `Strassen` were generic.

So we get for free an instance on `seqmatrix C`, for any `C` refining a ring. Correctness is derived from parametricity:

```
Variable (A : ringType) (mxC : nat -> nat -> Type).
Variable (RmxA : ∀ m n, 'M[A]_(m, n) ->mxC m n -> Prop).

Instance param_Strassen p :
  refines (RmxA ==> RmxA ==> RmxA)
    (Strassen (matrix A) p) (Strassen mxC p).
```

5

Internal computations using refinements

My personal initial motivation

```
Definition ctmat1 : 'M[int]__ := \matrix_(i < 3, j < 3)
  (nth [::] [:: [:: 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.
```

With CoqEAL

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

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

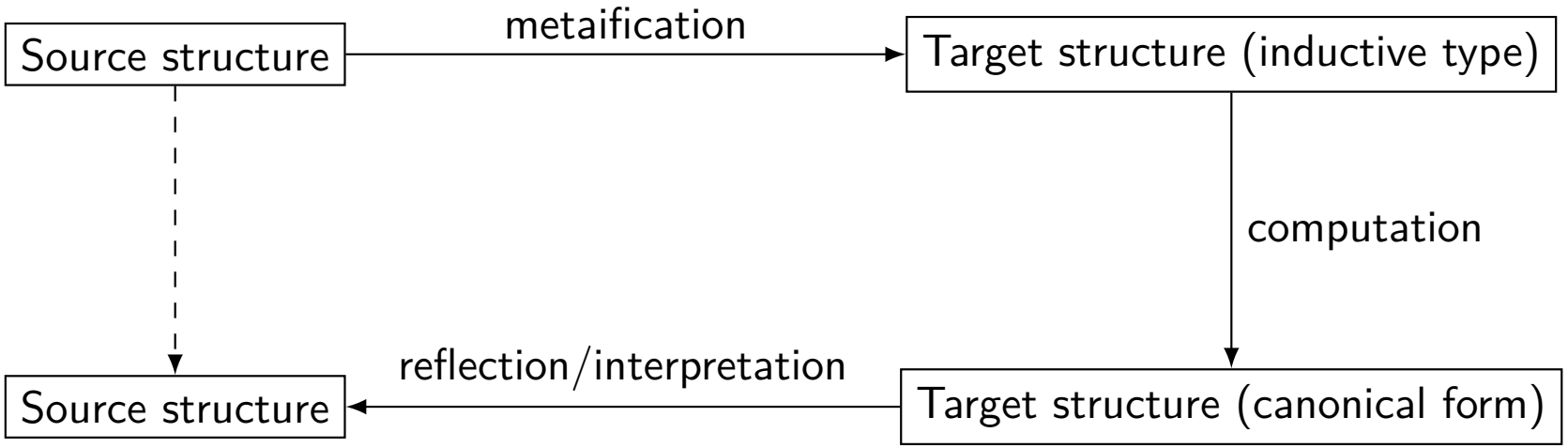
Another example

```
Lemma test_irred : irreducible_poly ('X^5 + 'X^2 + 1 : {poly
     'F_2}).
Proof.
apply/irreducibleP; rewrite /irreducibleb -[size _]/(sizep _
    ).
rewrite -[[forall _, _]]/(_ == _) /= /Pdiv.Ring.rdvdp.
by coqeal.
Qed.
```

6

Large scale automation using refinements

Reflection
[Boutin 97]



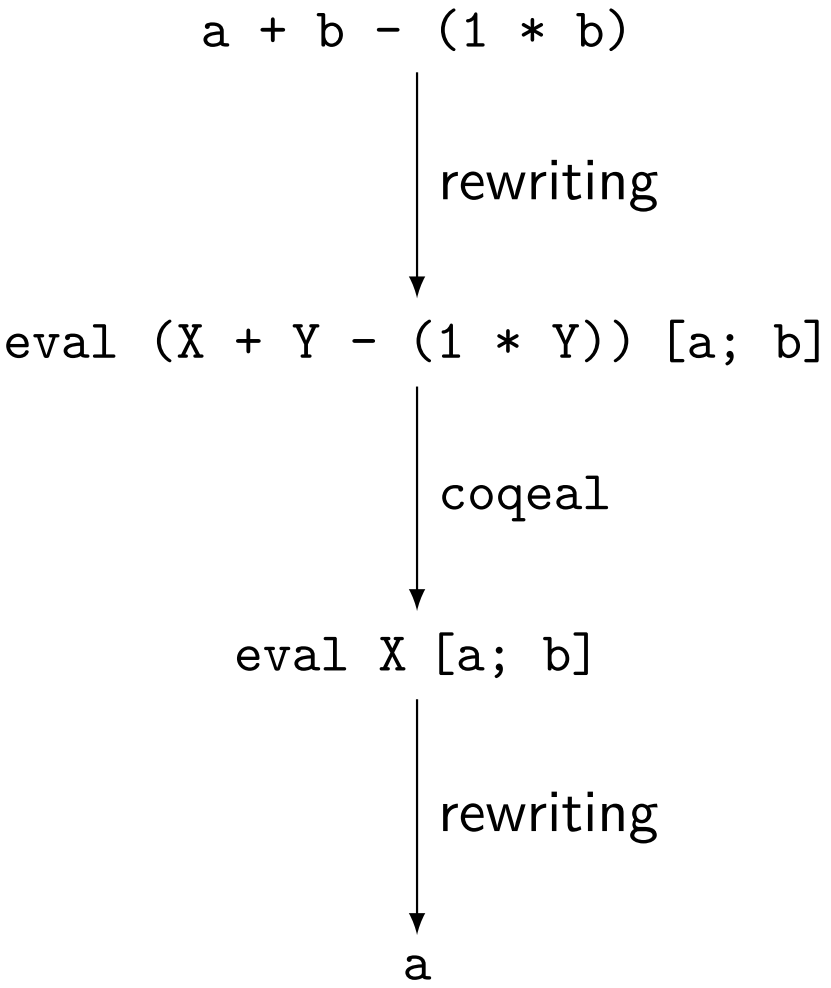
Reflection on rings [Grégoire, Mahboubi 2005]

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)` $\longrightarrow$ `X + Y - (1 * Y)` with variable map `[a; b]`.

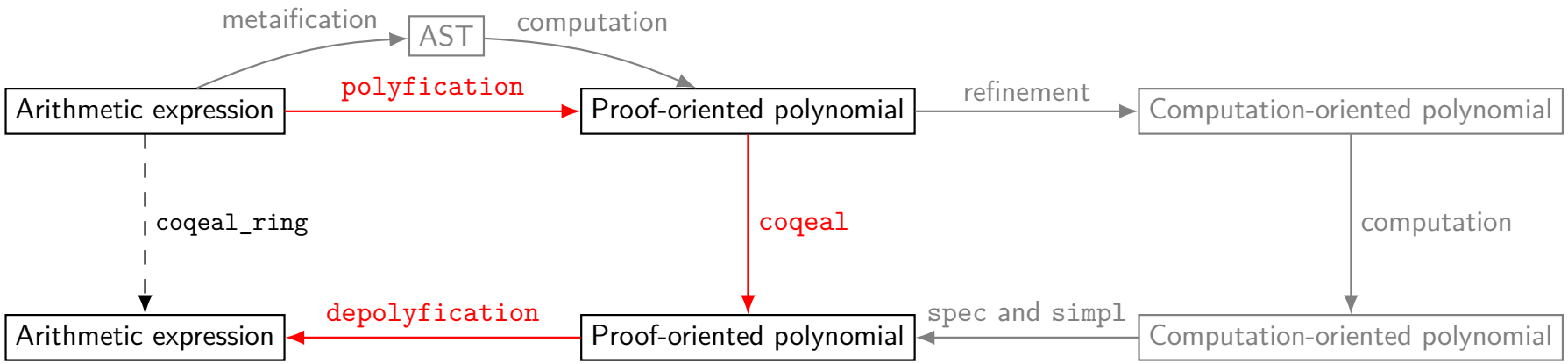
Computation:
The goal of the computation step is to normalise the obtained polynomials.
`X + Y - (1 * Y)` $\longrightarrow$ `X`.

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

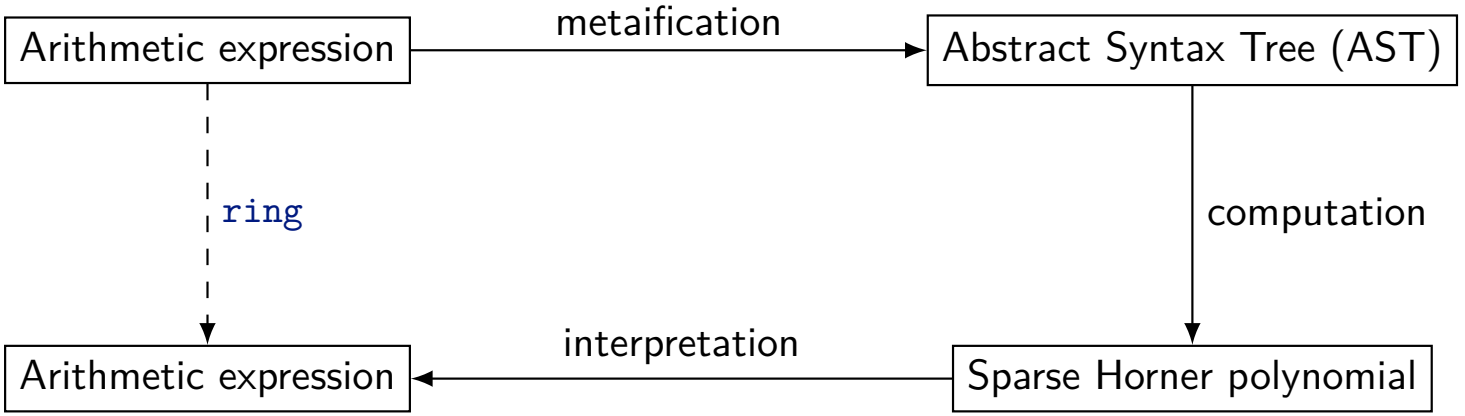
The `coqeal_ring` tactic [Cohen, Rouhling 2017]



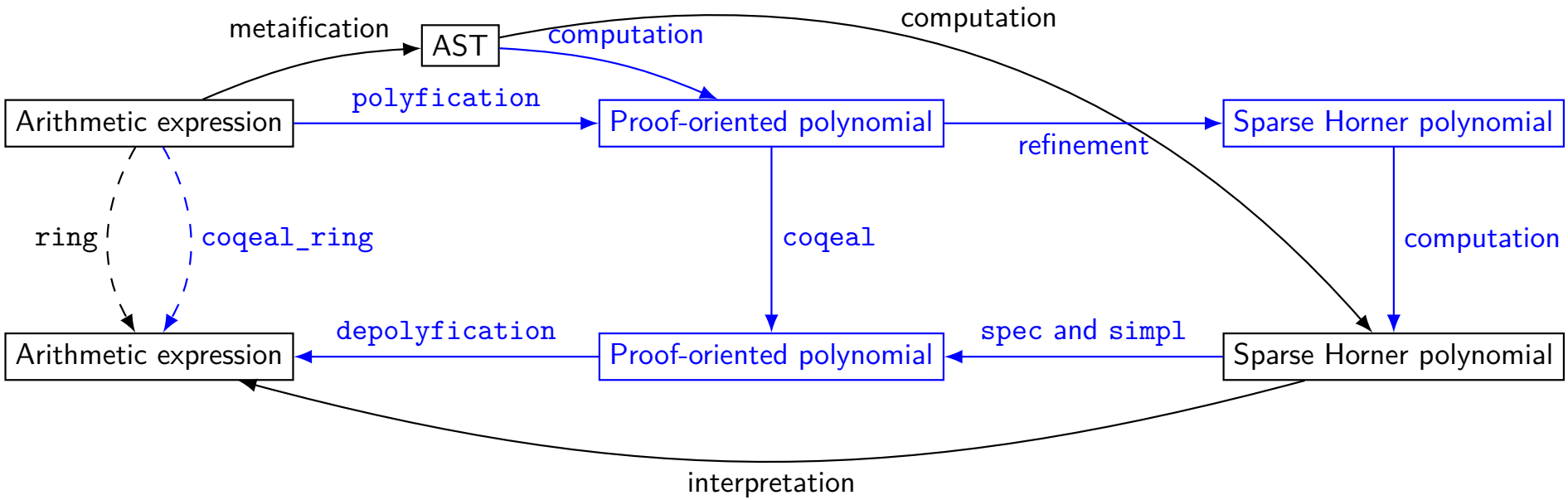
The `coqeal_ring` tactic [Cohen, Rouhling 2017]



The `ring` tactic [Grégoire, Mahboubi 2005]



Comparison


metaification
AST
computation
computation
Arithmetic expression
polyfication
Proof-oriented polynomial
refinement
Sparse Horner polynomial
ring
coqeal_ring
coqeal
computation
depolyfication
spec and simpl
Arithmetic expression
Proof-oriented polynomial
Sparse Horner polynomial
interpretation

Further work

- 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 nested data-structures, improved `depolyfication`.
- Implement new features: morphisms, Gröbner bases, other reduction strategies, user-defined operations...
- Generalise to other decision procedures: `field`? `lra`???

7

Conclusion

Inspirations

- A New Look at Generalized Rewriting in Type Theory
  [Sozeau, JFR'09]
- Univalence: Isomorphism is equality
  [Coquand - Danielsson, '13]
- Parametricity in an Impredicative Sort
  [Keller - Lasson, CSL'12]
- Theorems for free!
  [Wadler, '89]
- Types, abstraction and parametric polymorphism
  [Reynolds, '83]

This work

- A refinement-based approach to large scale reflection for algebra
  [C. - Rouhling, JFLA'17]
- Refinements for free!
  [C. - Dénès - Mörtberg, CPP'13]
- A refinement-based approach to computational algebra in Coq
  [Dénès - Mörtberg - Siles, ITP'12]

Some closely related work

- Equivalences for Free!
  [Tabareau, Tanter, Sozeau, ICFP'18]
- Automatic and Transparent Transfer of Theorems along Isomorphisms in the Coq Proof Assistant.
  [Zimmerman, Herbelin, CICM'15]
- Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant.
  [Delaware, Pit-Claudel, Gross, Chlipala, POPL'15]
- Automatic data refinements in ISABELLE/HOL
  [Lammich, ITP'13]
- ...

Further work

- Write a plugin using MetaCoq (collaboration with S. Boulier)
- or using Coq-elpi (collaboration with E. Tassi),
- Generalize framework to transfer theorems when possible,
- Have an exhaustive refinements available:
  - for basic programming datatypes (maps, sets, trees, graphs, . . .)
  - for algebraic datatypes (polynomials, matrices, semi-algebraic and algebraic sets, . . .)
  - for real number approximations?
- Experiment with extraction to Ocaml and Haskell
- Develop more large scale reflection tactics
  (Current target: Gröbner bases)

Thanks