Refinements for free![1]

Cyril Cohen, Maxime Dénès and Anders Mörtberg

University of Gothenburg and Inria Sophia-Antipolis

December 12, 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

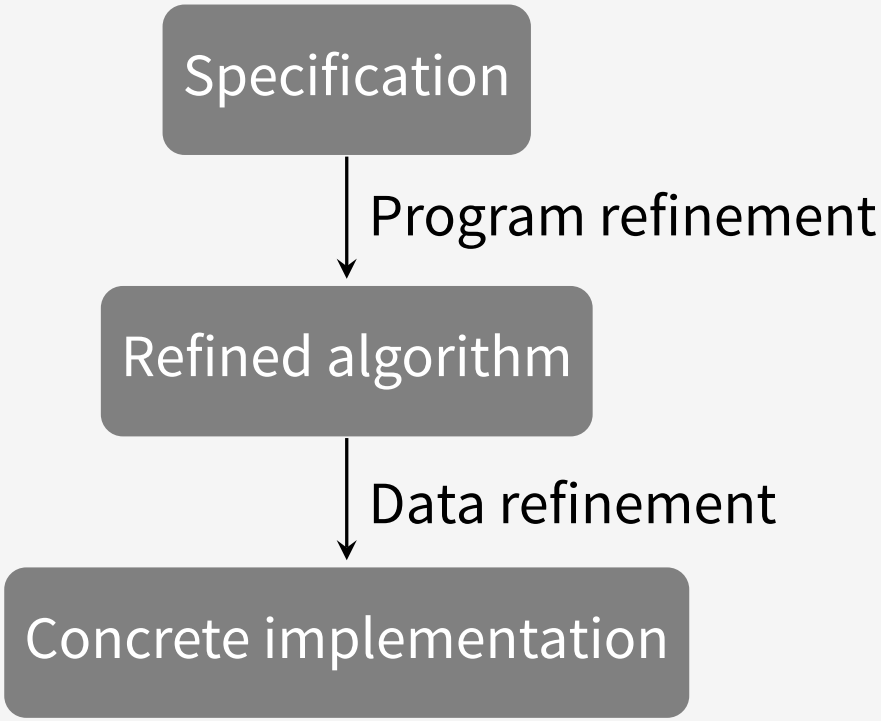
We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day, so to speak. […] But nothing is gained – on the contrary! – by tackling these various aspects simultaneously. It is what I sometimes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



Traditional abstraction

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

1. Abstract the datatype D over $\mathbf{B}$ $\Longrightarrow$ $\mathrm{D}\mathbf{X}$
2. Abstract operators t over $\mathbf{B}$ and $\mathbf{b}$ $\Longrightarrow$ $\mathrm{t}\mathbf{X}\mathbf{x}$
3. Abstract theorems over $\mathbf{B}$ and $\mathbf{b}$ and over theorems about $\mathbf{b}$

- Program refinement by having two extensionally equal $\mathbf{t_0}$ and $\mathbf{t_1}$.
- Data instantiation, but how to change a datatype for another?

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 instanciated to these two implementations.

Using 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).

Using 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 $\mathbb{DB}$ to $\mathbb{DC}$)
- goes against the "small scale reflection" approach (following SSReflect)

Context: Libraries, Conventions, Examples

Proof-oriented types: A, $\mathrm{T}(\mathbf{A})$
E.g.: `nat`, `int`, `rat`, `{poly R}`, `(matrix R)`...

Proof-oriented programs:
$\mathfrak{a}$, $\mathfrak{t}(\mathfrak{a})$
E.g.: `O`, `S`, `addn`, `addz`, ..., `0%R`, `1%R`, `(_+_)%R`...

Rich theory, geared towards interactive proving

Computation-oriented types:
B, $\mathrm{D}(\mathbf{B})$
E.g.: `N`, `Z`, `Q`, `sparse_poly`, ...

Computation-oriented programs: $\mathfrak{b}$, $\mathfrak{d}(\mathfrak{b})$
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: A, $\mathrm{T}(\mathbf{A})$
E.g.: `nat`, `int`, `rat`, `{poly R}`, `(matrix R)`...

Proof-oriented programs:
a, $\mathrm{t}(\mathbf{a})$
E.g.: `O`, `S`, `addn`, `addz`, ..., `0%R`, `1%R`, `(_+_)%R`...

Rich theory, geared towards interactive proving

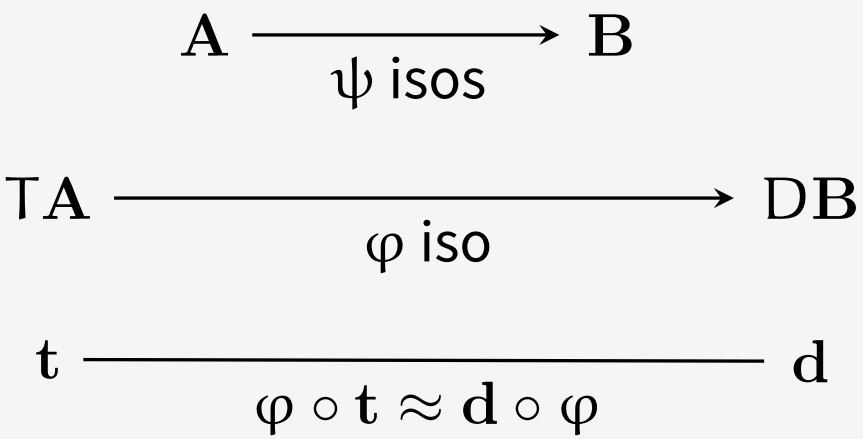
Computation-oriented types:
B, $\mathrm{D}(\mathbf{B})$
E.g.: `N`, `Z`, `Q`, `sparse_poly`, ...

Computation-oriented programs: b, $\mathrm{d}(\mathbf{b})$
E.g.: `xH`, `xI`, `xO`, `addN`, `addQ`, ..., `0%C`, `1%C`, `(_+_)%C`...

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

We suggest a methodology based on refinement from $\mathrm{T}(\mathbf{A})$ to $\mathrm{D}(\mathbf{B})$ to achieve separation of concerns.

Direct style (bad)



Theory on t

No separation of concerns!

ITP 2012 (Dénès, Mörtberg, Siles)

Assuming we have a theory about $\mathbf{t}$ on a type $\mathrm{T}\mathbf{A}$:

1. write efficient algorithms $\mathbf{t}'$ for $\mathrm{T}\mathbf{A}$,
2. prove that $\mathrm{T}\mathbf{A}$ and $\mathrm{D}\mathbf{B}$ are isomorphic,
3. duplicate the algorithms d for $\mathrm{D}\mathbf{B}$,
4. prove extensional equality of algorithms.

ITP 2012 (Dénès, Mörtberg, Siles)

$$\mathbf{A} \xrightarrow[\psi \text{ isos}]{} \mathbf{B}$$

$$\mathsf{T}\mathbf{A} \xrightarrow[\varphi \text{ iso}]{} \mathrm{D}\mathbf{B}$$

$$\mathbf{t} \xrightarrow[\mathbf{t} \approx \mathbf{t}']{} \mathbf{t}' \xrightarrow[\varphi \circ \mathbf{t}' \approx \mathbf{d} \circ \varphi]{} \mathbf{d}$$

Prog refinement | Data refinement

Theory on $\mathbf{t}$

ITP 2012 (Dénès, Mörtberg, Siles)

$$\mathbf{A} \xrightarrow[\psi \text{ isos}]{} \mathbf{B}$$

$$\mathsf{T}\mathbf{A} \xrightarrow[\varphi \text{ iso}]{} \mathsf{D}\mathbf{B}$$

$$\mathbf{t} \xrightarrow[\mathbf{t} \approx \mathbf{t}']{} \mathbf{t}' \xrightarrow[\varphi \circ \mathbf{t}' \approx \mathbf{d} \circ \varphi]{} \mathbf{d}$$

Prog refinement | Data refinement

Theory on $\mathbf{t}$

Issues:

- $\mathbf{t}'$ and $\mathbf{d}$ are duplicates,
- data refinements contain no mathematics,
  but long and time consuming to write down by hand,
- what if $\mathcal{A}$ and $\mathcal{B}$ 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.

The new strategy

Assuming we have a theory on a type A:

1. write efficient algorithms in a generic form $\mathbf{dXx}$,
2. prove $\mathbf{dAa}$ correct with regard to $\mathbf{tAa}$
3. get correctness for $\mathbf{dXx}$ by parametricity.

Overview

$$A \;\overset{R_A}{\text{———}}\; X$$

$$TA \;\overset{R_T \mathbf{R_A}}{\text{———}}\; DA \;\overset{[D]\mathbf{R_A}}{\text{———}}\; D\mathbf{X}$$

$$GTA \;\overset{\mathbf{tAa}\ \ [G]R_T\mathbf{R_A}\ \ \mathbf{dAa}}{\text{———}}\; GDA \;\overset{[G][D]\mathbf{R_A}}{\text{———}}\; GD\mathbf{X}$$

Generic programming: addition over rationals

Generic datatype ($\mathrm{D}\mathbf{X}$)

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

Generic operations ($\mathbf{dXx}$ of type $\mathrm{GD}\mathbf{X}$)

```
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 ($\mathbf{x}$) `(+ : add Z)` and `(* : mul Z)` are instanciated by proof-oriented definitions ($\mathbf{a}$).
When computing, these operators are instanciated to more efficient ones.

Proof-oriented correctness

- The type `int` is the proof-oriented version of integers ($\mathbb{A}$).
- The type `rat` is the proof-oriented version of rationals ($\mathbb{T}$).

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

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

Overview

$$A \;\overset{R_A}{\text{———}}\; X$$

$$FA \;\overset{a\ [F]\mathbf{R_A}\ x}{\text{———}}\; FX$$

$$TA \;\overset{R_T\mathbf{R_A}}{\text{———}}\; DA \;\overset{[D]\mathbf{R_A}}{\text{———}}\; DX$$

$$GTA \;\overset{\mathbf{tAa}\ \ [G]R_T\mathbf{R_A}\ \ \mathbf{dAa}}{\text{———}}\; GDA \;\overset{\mathbf{dAa}\ \ [G][D]\mathbf{R_A}\ \ \mathbf{dXx}}{\text{———}}\; GDX$$

Parametricity

Parametricity for closed terms in

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

(Reynolds, Wadler, Keller and Lasson)

Automatic deduction of

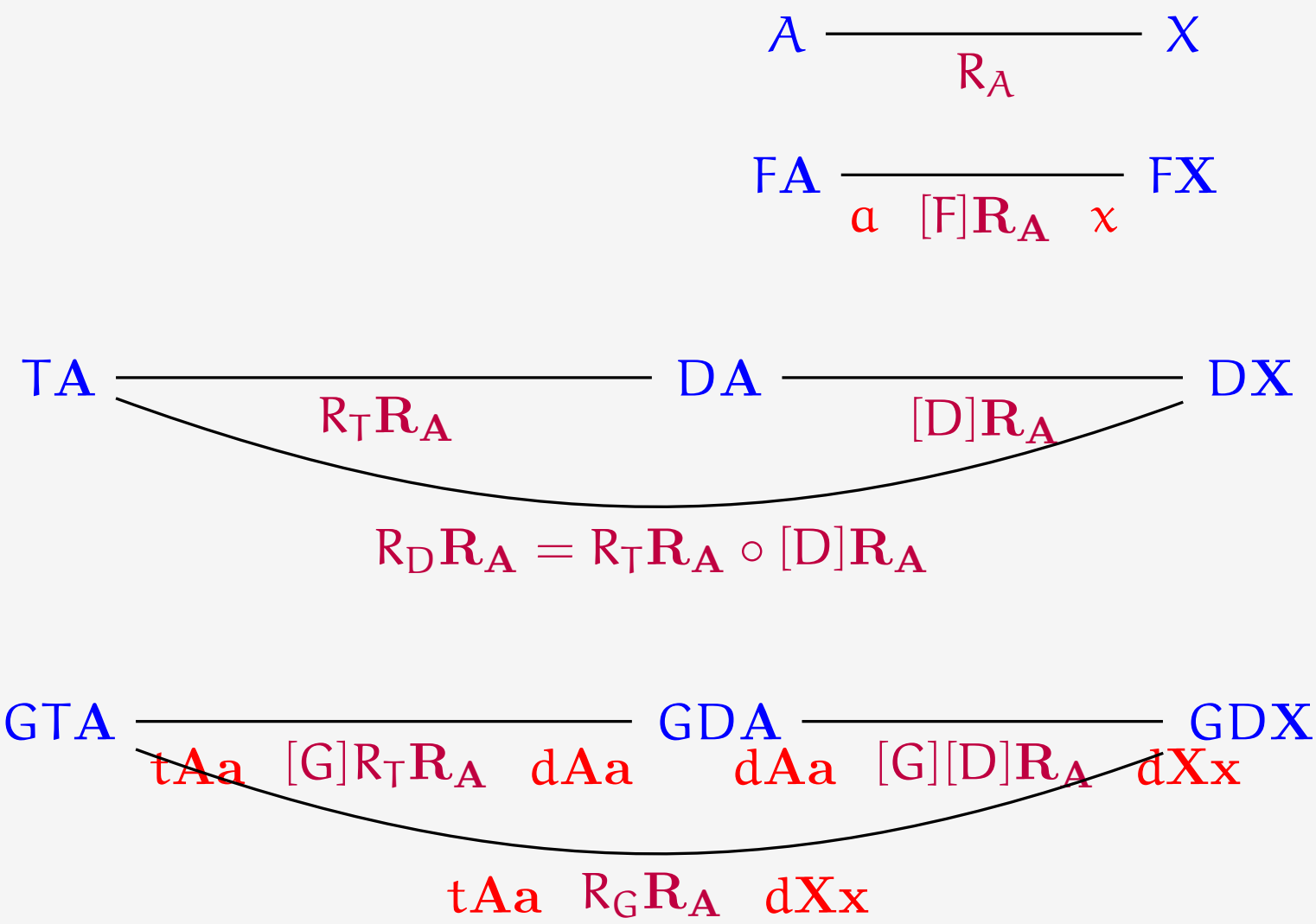
$$\mathbf{dAa} \quad [\mathrm{G}][\mathrm{D}]\mathbf{R_A} \quad \mathbf{dXx}$$

from

$$\mathbf{a} \quad [\mathrm{F}]\mathbf{R_A} \quad \mathbf{x}$$

Currently using a logic program for Coq typeclass mechanism
It is possible because $[\mathrm{d}] : [\mathrm{forall}\mathbf{YX}, \mathrm{GYX}]\mathrm{dd}$

Overview


A
X
R_A
FA
FX
a
$[F]\mathbf{R_A}$
x
TA
DA
DX
$R_T\mathbf{R_A}$
$[D]\mathbf{R_A}$
$R_D\mathbf{R_A} = R_T\mathbf{R_A} \circ [D]\mathbf{R_A}$
GTA
GDA
GDX
tAa
$[G]R_T\mathbf{R_A}$
dAa
dAa
$[G][D]\mathbf{R_A}$
dXx
tAa
$R_G\mathbf{R_A}$
dXx

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).

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

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).

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

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

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).

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

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

Automation

Correctness of addQ

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

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

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

Automation

Correctness of addQ

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

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

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

- `RQint_add` is not for free,

Automation

Correctness of addQ

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

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

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

- `RQint_add` is not for free,
- but `param_addQ` should be!

Using the framework

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

$$\mathrm{spec} : \mathrm{D}\mathbf{X} \to \mathsf{T}\mathbf{A}$$

which is a refinement of the identity function.

The corresponding relation has type:

$$\mathrm{Rspec} : \forall \mathbf{X}\mathbf{x}, \mathbf{R_A}\mathbf{a}\mathbf{x} \to \forall (\mathfrak{u} : \mathsf{T}\mathbf{A})(\mathfrak{v} : \mathrm{D}\mathbf{X}), \mathrm{R_D}\mathfrak{u}\mathfrak{v} \to \mathrm{id}\mathfrak{u} = \mathrm{spec}\mathfrak{v}$$

$$\mathrm{C}[\mathfrak{u}] \to \mathrm{C}[\mathrm{spec}\mathfrak{v}]$$

using `rewrite Rspec`.

Related work

- (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!