Towards verified computer algebra

Maxime Dénès
joint work with Cyril Cohen and Anders Mörtberg

Inria Paris-Rocquencourt, Inria Sophia-Antipolis and University of Gothenburg

January 26, 2014

Context

Computers play an increasingly important role in mathematical proofs.

A few examples:

- Four color theorem [Appel and Haken 1977; Gonthier 2007]
- Robbins conjecture [McCune 1997]
- Kepler conjecture [Hales 2005]
- Existence of the Lorenz attractor [Tucker 2002]
- The odd order theorem [Gonthier et al. 2013]

Context

Computers play an increasingly important role in mathematical proofs.

A few examples:

- Four color theorem [Appel and Haken 1977; Gonthier 2007]
- Robbins conjecture [McCune 1997]
- Kepler conjecture [Hales 2005]
- Existence of the Lorenz attractor [Tucker 2002]
- The odd order theorem [Gonthier et al. 2013]

The motivation for using computers is often (but not always) computational power.

Context

Computers play an increasingly important role in mathematical proofs.

A few examples:

- Four color theorem [Appel and Haken 1977; Gonthier 2007]
- Robbins conjecture [McCune 1997]
- Kepler conjecture [Hales 2005]
- Existence of the Lorenz attractor [Tucker 2002]
- The odd order theorem [Gonthier et al. 2013]

The motivation for using computers is often (but not always) computational power. Sometimes, stubbornness is the killing feature.

Software for mathematics

Currently (at least) two families of software for mathematicians:

- Computer algebra systems
- Theorem provers

Software for mathematics

Currently (at least) two families of software for mathematicians:

- Computer algebra systems
- Theorem provers

Surprisingly different. The first emphasize equational reasoning and efficiency of computations, the second semantics and logical reasoning.

Software for mathematics

Currently (at least) two families of software for mathematicians:

- Computer algebra systems
- Theorem provers

Surprisingly different. The first emphasize equational reasoning and efficiency of computations, the second semantics and logical reasoning.

Analogy: Babylonian and Greek mathematics [Barendregt and Barendsen 2002]

Software for mathematics

Currently (at least) two families of software for mathematicians:

- Computer algebra systems
- Theorem provers

Surprisingly different. The first emphasize equational reasoning and efficiency of computations, the second semantics and logical reasoning.

Analogy: Babylonian and Greek mathematics [Barendregt and Barendsen 2002]

Our goal: teach the Greeks to speak Babylonian

Motivation

Verifiying computer algebra, 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)

Computations in formal proofs

Traditionally, three ways to incorporate computations in formal proofs:

- Believing
- Skeptical
- Autarkic

Computations in formal proofs

Traditionally, three ways to incorporate computations in formal proofs:

- Believing
- Skeptical
- Autarkic

In our context, we consider the last two, with emphasis on the third.

> *Give a man a fish and you feed him for a day. Teach a man to fish and you feed him for a lifetime.*

How to verify efficient programs?

How to verify efficient programs?

Specificity of our context: efficiency matters, and programs can have complex specifications (good proof tools required)

How to verify efficient programs?

Specificity of our context: efficiency matters, and programs can have complex specifications (good proof tools required)

Observation: tension between proof-oriented descriptions and efficient implementations

How to verify efficient programs?

Specificity of our context: efficiency matters, and programs can have complex specifications (good proof tools required)

Observation: tension between proof-oriented descriptions and efficient implementations

Our proposal: a framework for top-down stepwise refinements from specifications to programs, achieving 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)

Outline

1. A refinement framework
2. Case study: Strassen's algorithm
3. Scaling up: verified homology computations

Outline

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's standard lib

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

$\rightarrow$ 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.

Traditional refinements (e.g. B method)

Successive and progressive refinements

$$P_1 \to P_2 \to \ldots \to P_n$$

where P_1 is an *abstract* version of the program
and P_n a *concrete* version of the program.

Key invariant: P_{n+1} must be correct with regard to P_n.

Our refinements

Successive and progressive refinements

$$P_1 \to P_2 \to \ldots \to P_n$$

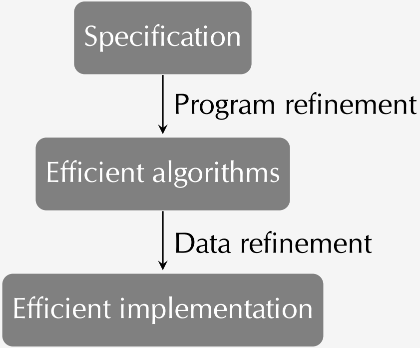
where P_1 is an proof-oriented version of the program
and P_n a computation-oriented version of the program.

Key invariant: P_{n+1} must be correct with regard to P_n.

Program and data refinements

Our 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*.



Context: Libraries, Conventions, Examples

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

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`, `seqmatrix...`

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

Example: matrices

- Proof-oriented matrices over a ring $M[R]$.
- Computation-oriented matrices $M'[R]$

$$A *_{M[R]} B \longrightarrow A *_{\mathtt{Strassen}(M[R])} B \longrightarrow A *_{\mathtt{Strassen}(M'[R])} B$$

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 express that `rat` is isomorphic to a quotient of a subset of pairs of integers.

$\longrightarrow$ refinement relation

Example: matrices

- Proof-oriented matrices over a ring $M[R]$.
- Computation-oriented matrices $M'[R]$

$$A *_{M[R]} B \longrightarrow A *_{\mathtt{Strassen}(M[R])} B \longrightarrow A *_{\mathtt{Strassen}(M'[R])} B$$

Example: matrices

- Proof-oriented matrices over a ring $M[R]$.
- Computation-oriented matrices $M'[R']$

$$A *_{M[R]} B \rightarrow A *_{\mathtt{Strassen}(M[R])} B \rightarrow A *_{\mathtt{Strassen}(M'[R'])} B$$

Example: matrices

- Proof-oriented matrices over a ring $M[R]$.
- Computation-oriented matrices $M'[R']$

$$A *_{M[R]} B \longrightarrow A *_{\mathtt{Strassen}(M[R])} B \longrightarrow A *_{\mathtt{Strassen}(M'[R])} B \longrightarrow A *_{\mathtt{Strassen}(M'[R'])} B$$

$\longrightarrow$ Compositionality

Example with rationals

- Proof-oriented rationals `rat`,
  based on unary integers `int`.
- Computation-oriented rationals `Q Z`,
  based on any implementation on integers `Z`.

$$a +_{\texttt{rat}} b \rightarrow a +_{\texttt{Q int}} b \rightarrow a +_{\texttt{Q Z}} b$$

Generic programming: addition over rationals

Generic datatype

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

Generic programming: addition over rationals

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).
```

Generic programming: addition over rationals

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)`.

Generic programming: addition over rationals

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.

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.

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.

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

Lemma param_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.

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

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

The proof of `RQint_add` is interesting, but the one 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.

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

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

The proof of `RQint_add` is interesting, but the one of `param_addQ` is boring. The lemma `param_addQ` is in fact a "theorem for free!"

Parametricity

Parametricity for closed terms

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

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

Proof of `param_addQ`

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

Automating proof transport

COQ's formalism does not admit parametricity internally, but we use it externally, to define a meta-program transporting proofs of correctness.

Automating proof transport

COQ's formalism does not admit parametricity internally, but we use it externally, to define a meta-program transporting proofs of correctness.

We implement this meta-program by logic programming with type classes.

Automating proof transport

COQ's formalism does not admit parametricity internally, but we use it externally, to define a meta-program transporting proofs of correctness.

We implement this meta-program by logic programming with type classes.

We obtain a refinement framework where:

- The refinement interface is flexible (heterogeneous relations)
- Correctness proofs are done in a proof-oriented context (reusing tools provided by SSREFLECT).
- Transporting these proofs to computation-oriented instance is mostly automated thanks to parametricity.

CoqEAL

In collaboration with C. Cohen and A. Mörtberg, we used refinements to design a library of effective algebra (CoqEAL). It provides verified effective implementations for integers, rational numbers, polynomials, matrices. [Dénès et al. 2012; Cohen et al. 2013]

The library covers:

- Basic matrix algebra, rank computation, PLU decomposition
- Strassen's matrix product
- Fast triangular matrix inversion
- Smith normal form
- Existence proofs for canonical forms: Frobenius, Jordan
- Karatsuba's product of polynomials
- Sasaki-Murao algorithm for determinant over a (commutative) ring

Outline

Strassen's algorithm (Winograd variant)

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

Strassen's algorithm (Winograd variant)

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

$$\begin{array}{lll}
S_1 = A_{2,1} + A_{2,2} & P_1 = A_{1,1} \times B_{1,1} & U_1 = P_1 + P_6 \\
S_2 = S_1 - A_{1,1} & P_2 = A_{1,2} \times B_{2,1} & U_2 = U_1 + P_7 \\
S_3 = A_{1,1} - A_{2,1} & P_3 = S_4 \times B_{2,2} & U_3 = U_1 + P5 \\
S_4 = A_{1,2} - S_2 & P_4 = A_{2,2} \times T_4 & C_{1,1} = P_1 + P_2 \\
T_1 = B_{1,2} - B_{1,1} & P_5 = S_1 \times T_1 & C_{1,2} = U_3 + P_3 \\
T_2 = B_{2,2} - T_1 & P_6 = S_2 \times T_2 & C_{2,1} = U_2 - P_4 \\
T_3 = B_{2,2} - B_{1,2} & P_7 = S_3 \times T_3 & C_{2,2} = U_2 + P_5 \\
T_4 = T_2 - B_{2,1} & &
\end{array}$$

Strassen's algorithm (Winograd variant)

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

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

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

```
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:

$$\begin{aligned} R_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}b \\ R_{2,1} &= A_{2,1}B_{1,1} + aB_{2,1} \\ R_{2,2} &= A_{2,1}B_{1,2} + ab \end{aligned}$$

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 (we used overloading)!

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 (we used overloading)!

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:

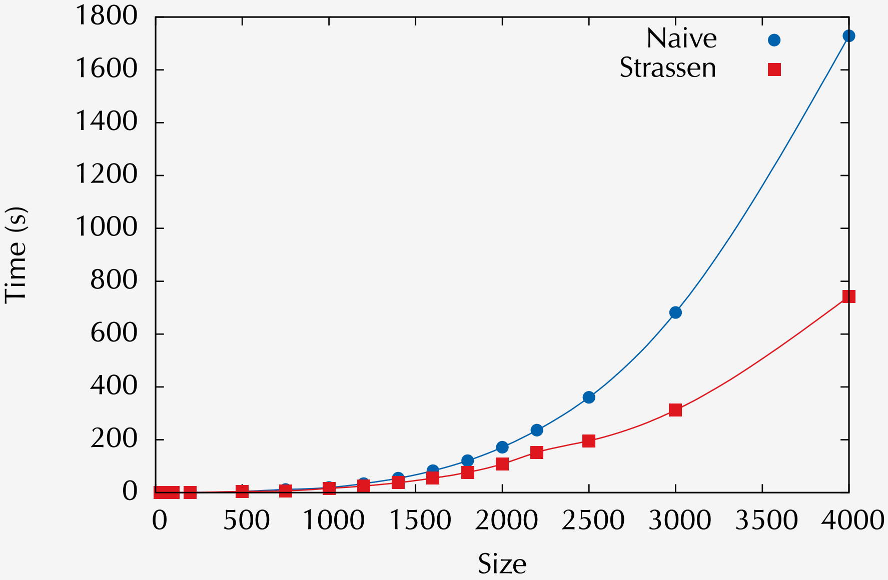
```
Lemma StrassenP (n : positive) (M N : 'M[R]_n) :
  Strassen M N = M *m N.
```

In fact, `Strassen_step` and `Strassen` were generic (we used overloading)!

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

```
Variable (A : ringType) (mxC : nat -> nat -> Type).
Variable (RmxA : forall {m n}, 'M[A]_(m, n) -> mxC m n ->
    Prop).
Instance param_Strassen p :
  param (RmxA ==> RmxA ==> RmxA) (Strassen (matrix A) p)
      (Strassen mxC p).
```

Benchmarks

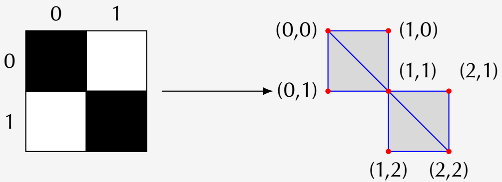


1800
1600
1400
1200
1000
800
600
400
200
0
Time (s)
0
500
1000
1500
2000
2500
3000
3500
4000
Size
Naive
Strassen

Outline

Homology of digital images

Digital image → Simplicial complex → Reduction → Homology



$$H_n = \frac{Z_n}{B_n}$$

$$\beta_n = \dim H_n$$

β_0: number of connected components ; β_1: number of holes

In our context, the computation of β_n is reduced to rank computations, for which we reuse an algorithm we verified. [Heras, Poza, et al. 2011; Heras, Dénès, et al. 2012]

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)

Conclusion and future work

Lessons learned:

- Separation of concerns is critical
- Refinements are a convenient way of abstracting in type theory
- Significant examples were required to make sure the framework scaled up
- Used in the recent formal proof of the irrationality of zeta(3) (Chyzak, Mahboubi et al.)

Future work:

- Other representations (e.g. sparse matrices)
- Better way to get parametricity than typeclasses
- Try on algorithms outside algebra
- Scale up to dependent types

Thank you!