Refinements for free![1]

Cyril Cohen

joint work with Maxime Dénès and Anders Mörtberg

University of Gothenburg and Inria Sophia-Antipolis

May 8, 2014

[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/direct proof)
- 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

Traditional refinements (e.g. B method)

Successive and progressive refinements

$$P_1 \to P_2 \to \dots P_n$$

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.

Goal: 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)

Example (Strassen's algorithm, Winograd variant)

Matrix product can be refined to an efficient version:

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

Example (Strassen's algorithm, Winograd variant)

Matrix product can be refined to an efficient version:

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

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

Example (Strassen's algorithm, Winograd variant)

Matrix product can be refined to an efficient version:

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

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

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

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

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

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.

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`, ...

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.

Our refinements

Successive and progressive refinements

$$P_1 \to P_2 \to \dots P_n$$

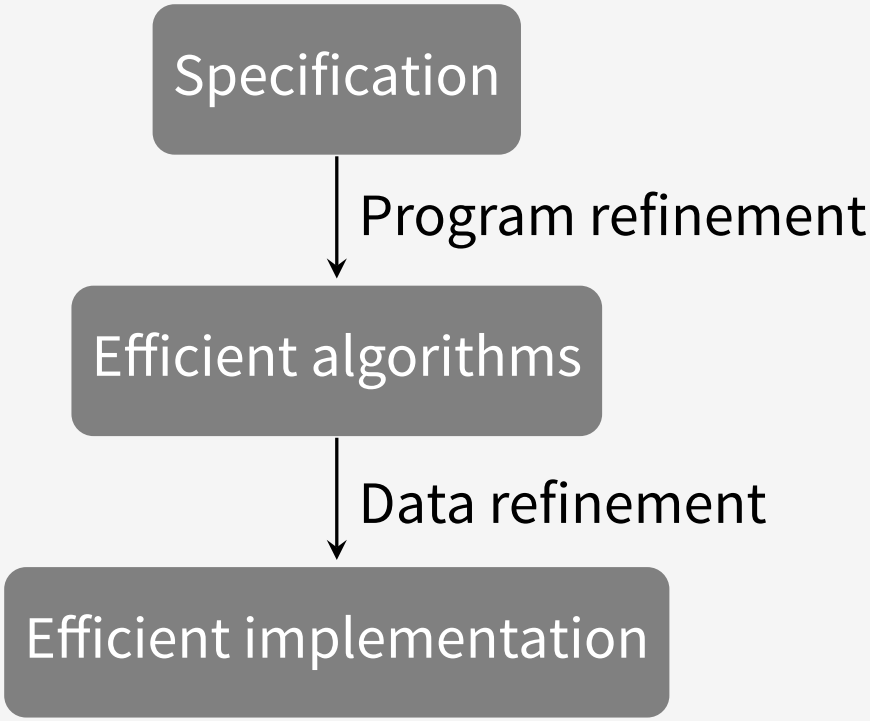
where P_1 is an proof-oriented version of the program
and P_n a computation-oriented version of the program.

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*.



Example

- Proof-oriented matrices over a ring $\mathrm{M}[\mathrm{R}]$.
- Computation-oriented matrices $\mathrm{M}'[\mathrm{R}]$

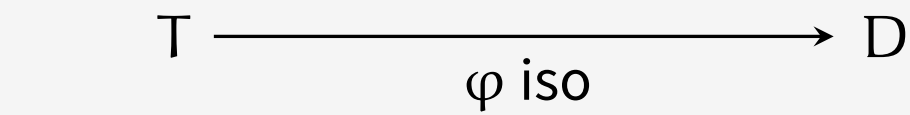
$$\mathrm{A} *_{\mathrm{M}[\mathrm{R}]} \mathrm{B} \to \mathrm{A} *_{\mathrm{Strassen}(\mathrm{M}[\mathrm{R}])} \mathrm{B} \to \mathrm{A} *_{\mathrm{Strassen}(\mathrm{M}'[\mathrm{R}])} \mathrm{B}$$

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)



$$\mathbf{f} \;\; \frac{}{\mathbf{f} = \mathbf{f}'} \;\; \mathbf{f}' \;\; \frac{}{\varphi \circ \mathbf{f}' = \mathbf{e} \circ \varphi} \;\; \mathbf{e}$$

Prog refinement Data refinement

Theory on $\mathbf{f}$

ITP 2012 (Dénès, Mörtberg, Siles)

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

$$\mathbf{f} \;\; \frac{}{\mathbf{f} = \mathbf{f}'} \;\; \mathbf{f}' \;\; \frac{}{\varphi \circ \mathbf{f}' = \mathbf{e} \circ \varphi} \;\; \mathbf{e}$$

Prog refinement | Data refinement

Theory on $\mathbf{f}$

Issues:

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

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.

$\longrightarrow$ refinement relation

Example

- Proof-oriented matrices over a ring $\mathrm{M}[\mathrm{R}]$.
- Computation-oriented matrices $\mathrm{M}'[\mathrm{R}]$

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

Example

- Proof-oriented matrices over a ring $\mathrm{M}[\mathrm{R}]$.
- Computation-oriented matrices $\mathrm{M}'[\mathrm{R}']$

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

Example

- Proof-oriented matrices over a ring $\mathrm{M}[\mathrm{R}]$.
- Computation-oriented matrices $\mathrm{M}'[\mathrm{R}']$

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

$\to$ 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`.

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

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
- 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.

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
- The proof 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 $[.]$, 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!, (C. - Dénès - Mörtberg, CPP'13)
- A refinement-based approach to computational algebra in Coq (Dénès - Mörtberg - Siles, ITP'12)

- Automatic data refinements in ISABELLE/HOL (Lammich, ITP'13)

- 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)

Applications and future work

- Applied to algorithms we had previously verified: Karatsuba's polynomial multiplication, Strassen's matrix product,
- Still porting others from the 2012 framework: Sasaki-Murao algorithm, Smith normal form.

Possible future work:

- have a better way to get parametricity than typeclasses (e.g. a plugin),
- refine my implementation of algebraic numbers in Coq,
- apply to homology group computation (refining C. - Mörtberg ITP'14)
- try on algorithms outside algebra,
- scale up to dependent types,
- extract efficient implementations.

Thanks!