Computations, algebra and computer algebra in Coq

Assia Mahboubi

INRIA Microsoft Research Joint Centre (France)
INRIA Saclay – Île-de-France
École Polytechnique, Palaiseau

January 30th 2012

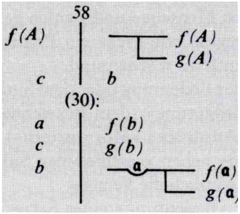
Proof assistants

A proof assistant is a software helping its user to check her own mathematical proof:

- Because its correctness plays a critical role in a critical application;
- Because it is too large and pedestrian for a human reader;
- Because it is too intricate and heterogeneous for a single reviewer.

Proof assistants

Using a proof assistant means you trust a computer to check your proofs.



- Use formal logic as assembly code to describe statements and proofs.
- Use a proof assistant to develop and to check this code.

A flavor of the assembly code

A fixed, finite set of symbols are used to construct mathematical statements.

$\wedge$ $\exists$

$\vee$

$\forall$

$\neg$

$\Longrightarrow$

$\Longleftrightarrow$

A flavor of the assembly code

Each symbol is associated with some grammar rules:

Grammar rule (intro) of the conjunction connector $\wedge$

These rules are presented like arithmetic operations.

A flavor of the assembly code

Each symbol is associated with some grammar rules:

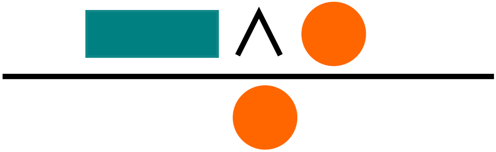
Grammar rule (left elim) of the conjunction connector : $\wedge$

These rules are presented like arithmetic operations.

A flavor of the assembly code

Each symbol is associated with some grammar rules:

Grammar rule (right elim) of the conjunction connector: $\bigwedge$

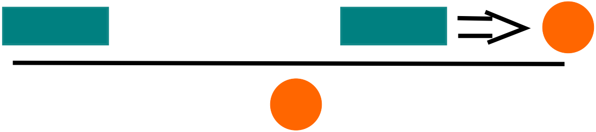


These rules are presented like arithmetic operations.

A flavor of the assembly code

Each symbol is associated with some grammar rules:

Grammar rule (elim) of the implication connector: 



These rules are presented like arithmetic operations.

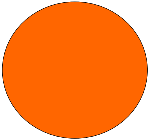
A flavor of the assembly code

What is a formal proof:

- Choose a set of axioms (things one takes for granted without proof).
- Form the desired conclusion.
- Solve the puzzle leading from the axiom to the conclusion, using only the previous grammar rules.

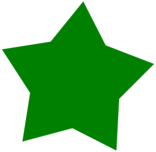
Example of formal proof

Let us fix some notations:



: mortal

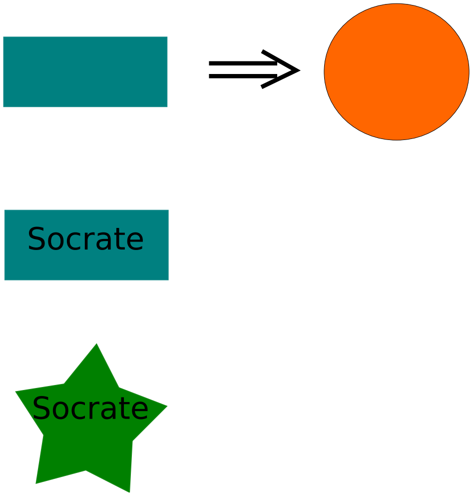
: man



: bearded

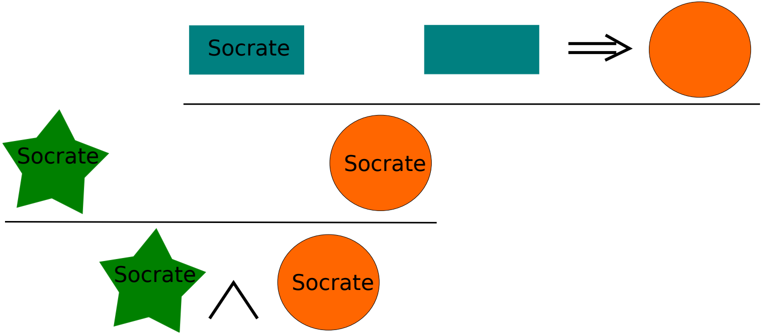
Example of formal proof

We choose some axioms:



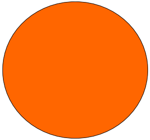
Example of formal proof

A proof that Socrate is both bearded and mortal



Example of formal proof

Let us fix some extra notations:

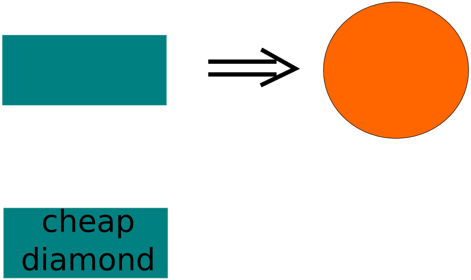


: expensive

: rare

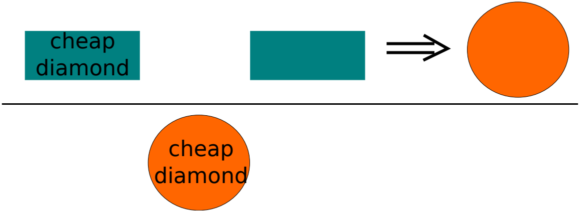
Example of formal proof

And different axioms:

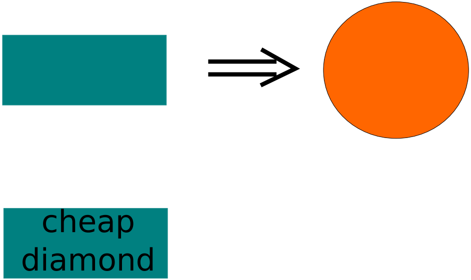


Example of formal proof

A proof that a cheap diamond is expensive:



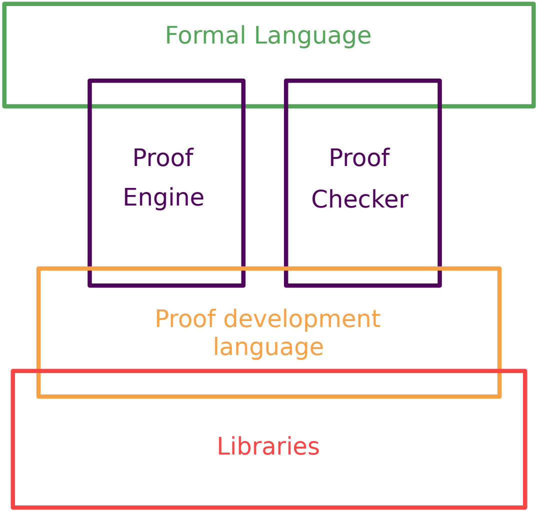
Example of formal proof



Example of formal proofs

- Formal proofs are trees.
- Nodes are labeled with logical rules.
- The proof assistant checker checks the tree is well-formed.
- But the proof assistant won't check your axioms are reasonable.

Architecture of a proof assistant



The Coq proof assistant

- Coq's type theory is a (kind of) typed functional programming language.
- A statement is a type.
- A proof is a term (a program).
- The system (without user axioms) provides a constructive framework.
- Computation has a special status in the inference rules of the system.

Modeling natural numbers in Coq

A standard presentation in the literature is the axiomatic one, which can be mimicked in the proof assistant:

Axioms (Relating Addition to O and S)

$\texttt{Anat} : \texttt{Type}$

$\texttt{AO} : \texttt{Anat}$

$\texttt{AS} : \texttt{Anat} \to \texttt{Anat}$

$+ : \texttt{Anat} \to \texttt{Anat} \to \texttt{Anat}$

$\texttt{addO} : \forall b,\ \texttt{O} + b = b$

$\texttt{addS} : \forall a\ b,\ (\texttt{AS}\ a) + b = a + (\texttt{AS}\ b)$

Deductive Reasoning for Peano's Arithmetic

Example (Deductive Proof of "$4 + (2 + 3) = 9$")

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\dfrac{\dfrac{}{9 = 9}\ \texttt{refl\_equal}}{0 + 9 = 9}\ \texttt{add0} \\ \vdots \qquad \texttt{addS} \times 4 \\ 4 + 5 = 9\end{array}}{4 + (0 + 5) = 9}\ \texttt{add0}}{4 + (1 + 4) = 9}\ \texttt{addS}}{4 + (2 + 3) = 9}\ \texttt{addS}$$

9 steps

The bigger the natural numbers in the proof, the more theorems have to be instantiated to prove the statement.

Deductive Reasoning for Peano's Arithmetic

This growth has a non-negligible cost.

- Time complexity: matching and applying theorems

  (any prover)

- Space complexity: storing proof terms

  (Coq-like provers)

Definitional vs axiomatic

Formal systems tending to prefer definitional extensions for consistency, they most often won't contain the above axioms.

The Coq system allows the definition of inductive types:

```
Inductive nat := O : nat | S : nat -> nat.
```

We can program an interpretation `[_] : nat -> Anat` as a recursive function, which transforms `O` into `AO` and `S` into `AS`.

Computing a bit inside proofs

We can moreover now program addition as a recursive function:

```
Fixpoint plus x y : nat :=
  match x with
  | O => y
  | S x' => plus x' (S y)
  end.
```

which is correct with respect to the previous specifications:

Lemma (Soundness)

`plus_xlate` : $\forall a\ b : nat,\ [a] + [b] = [\texttt{plus}\ a\ b]$.

Computing a little inside Proofs

Example (Proof of “$4 + (2 + 3) = 9$”)

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[9] = [9]}\ \texttt{refl\_equal}}{[\texttt{plus}\ 4\ (\texttt{plus}\ 2\ 3)] = [9]}\ {\color{red}???}}{[4] + [\texttt{plus}\ 2\ 3] = [9]}\ \texttt{plus\_xlate}}{[4] + ([2] + [3]) = [9]}\ \texttt{plus\_xlate}}{4 + ([2] + [3]) = [9]}\ \texttt{cst\_xlate}}{4 + (2 + [3]) = [9]}\ \texttt{cst\_xlate}}{4 + (2 + 3) = [9]}\ \texttt{cst\_xlate}}{4 + (2 + 3) = 9}\ \texttt{cst\_xlate}
$$

Computing a little inside Proofs

Example (Proof of “$4 + (2 + 3) = 9$”)

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[9] = [9]}\;\texttt{refl\_equal}}{[\texttt{plus}\ 4\ (\texttt{plus}\ 2\ 3)] = [9]}\;{\color{red}???}}{[4] + [\texttt{plus}\ 2\ 3] = [9]}\;\texttt{plus\_xlate}}{[4] + ([2] + [3]) = [9]}\;\texttt{plus\_xlate}}{4 + ([2] + [3]) = [9]}\;\texttt{cst\_xlate}}{4 + (2 + [3]) = [9]}\;\texttt{cst\_xlate}}{4 + (2 + 3) = [9]}\;\texttt{cst\_xlate}}{4 + (2 + 3) = 9}\;\texttt{cst\_xlate}
$$

One could consider λ-calculus as a rewriting system
and iteratively reduce “plus 4 (plus 2 3) = 9” to 9.
But this is no less costly than previous axiomatic axioms.

Type Theory and Conversion

Theorem (Curry-Howard Isomorphism)

Formula A^* is valid if and only if type A is inhabited.

Example : $(\Gamma \vdash_{\text{typing}} f : P \to Q)$ is equivalent to $(\Gamma^* \vdash_{\text{proving}} P^* \Rightarrow Q^*)$.

Property (Type Theory)

Convertible types have the same inhabitants.

$$\frac{p : A}{p : B}\ A \equiv B$$

β-conversion: $(\lambda x.t)u \equiv t[x \leftarrow u]$ $\quad (+\iota\zeta\delta\text{-rules})$

Type Theory and Conversion

Example (Proof of "$4 + (2 + 3) = 9$")

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p2 : [9] = [9]}\ \texttt{refl\_equal}}{p2 : [\texttt{plus}\ 4\ (\texttt{plus}\ 2\ 3)] = [9]}\ \text{conversion}}{[4] + [\texttt{plus}\ 2\ 3] = [9]}\ \texttt{plus\_xlate}}{p1 : [4] + ([2] + [3]) = [9]}\ \texttt{plus\_xlate}}{p1 : 4 + (2 + 3) = 9}\ \texttt{conversion}
$$

5 steps

Amount of theorem instantiations no longer depends on the size of the constants, only on the number of arithmetic operators.

Note: conversion is implicit when typechecking:
term "`refl_equal [9]`" has also type "$[\texttt{plus}\ 4\ (\texttt{plus}\ 2\ 3)] = [9]$".

Libraries of formalized mathematics

Like in (more) traditional programming languages or computer algebra systems, etc., the user stands on available, previously developed, libraries.

Here libraries should:

- Define mathematical objects and structures and their specifications;
- Develop the theory of these objects (possibly including programs);
- Organize this content so that it is generic, modular and reusable.

Explicit representations of mathematical objects

Like programming, formalizing mathematics imposes to choose explicit representations for mathematical objects.

Choosing the appropriate data structure(s) is of primary importance.

Formalization issues: comprehension style

- In set theory: comprehension rule forges:

  the set $\{x \mid P\ x\}$
- In type theory: Sigma types (dependent pairs) forge:

  the types $\{x \mid P\ x\}$
- Is there more to say?

Formalization issues: comprehension style

- In set theory: comprehension rule forges:

  the set $\{x \mid P\ x\}$
- In type theory: Sigma types (dependent pairs) forge:

  the types $\{x \mid P\ x\}$
- Is there more to say?

Yes, about the status of equality.

Formalization issues: comprehension style

The sigma type of duplicate free lists on type T is:

$$\{l \ : \ list T \mid \text{duplicate\_free } l\}$$

- An inhabitant t_l of this type is a pair (l, p_l)
- Comparing two inhabitants t_1 and t_2 means comparing them component-wise:
  $$t_1 = t_2 \quad \text{iff} \quad (l_1 = l_2) \wedge p_{l_1} = p_{l_2}$$
- The proof component should be irrelevant here.

But in general Coq is not a proof irrelevant system...

Formalization issues: comprehension style

Some (classical) predicates have a taste of proof-irrelevance:

$$\forall\ (x\ y\ :\ bool)\ (p_1\ p_2\ :\ x = y),\ p_1 = p_2$$

- Suppose that duplicate_free : list T $\rightarrow$ bool

Formalization issues: comprehension style

Some (classical) predicates have a taste of proof-irrelevance:

$$\forall\ (x\ y\ :\ bool)\ (p_1\ p_2\ :\ x = y),\ p_1 = p_2$$

- Suppose that duplicate_free : list T $\rightarrow$ bool
- Now the sigma type is: $\{l\ :\ list T \mid \text{duplicate\_free } l = \text{true}\}$

Formalization issues: comprehension style

Some (classical) predicates have a taste of proof-irrelevance:

$$\forall\ (x\ y\ :\ bool)\ (p_1\ p_2\ :\ x = y),\ p_1 = p_2$$

- Suppose that duplicate_free : list T $\rightarrow$ bool
- Now the sigma type is: $\{l\ :\ list T \mid \text{duplicate\_free } l = \text{true}\}$
- Compare (l_1, p_1) with (l_2, p_2) when $l_1 = l_2$.
  - $p_1\ :\ \text{duplicate\_free } l_1\ =\ true$
  - $p_2\ :\ \text{duplicate\_free } l_2\ =\ true$.

Formalization issues: comprehension style

Some (classical) predicates have a taste of proof-irrelevance:

$$\forall\ (x\ y\ :\ bool)\ (p_1\ p_2\ :\ x = y),\ p_1 = p_2$$

- Suppose that duplicate_free : list T $\rightarrow$ bool
- Now the sigma type is: $\{l\ :\ list T \mid$ duplicate_free $l =$ true$\}$
- Compare (l_1, p_1) with (l_2, p_2) when $l_1 = l_2$.
  - p_1 : duplicate_free $l_1 = true$
  - p_2 : duplicate_free $l_2 = true$.

Using the theorem, we prove that $p_1 = p_2$.

Formalization issues: comprehension style

Some (classical) predicates have a taste of proof-irrelevance:

$$\forall\ (x\ y\ :\ bool)\ (p_1\ p_2\ :\ x = y),\ p_1 = p_2$$

- Suppose that duplicate_free : list T $\rightarrow$ bool
- Now the sigma type is: $\{l\ :\ list T \mid$ duplicate_free $l =$ true$\}$
- Compare (l_1, p_1) with (l_2, p_2) when $l_1 = l_2$.
  - p_1 : duplicate_free l_1 $=$ *true*
  - p_2 : duplicate_free l_2 $=$ *true*.

Using the theorem, we prove that $p_1 = p_2$.

Comparing inhabitants of boolean sigma types is comparing values.

Other issues with equality

In Coq, there is no way in general to conclude that:

$$f = g$$

from the fact that:

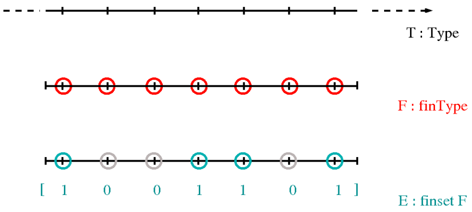
$$\forall x, f(x) = g(x)$$

In particular, the naive representation of sets as characteristic functions might become uncomfortable.

Finite sets as finite characteristic functions

A finite type F is an enumeration of its inhabitants.

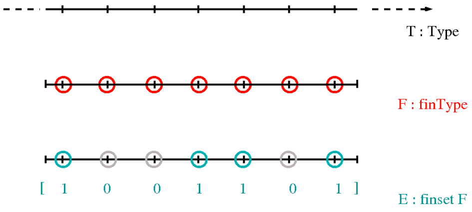
A finite set `(s : set F)` is represented as a mask on a finite type `F`:



Finite sets as finite characteristic functions

A finite type F is an enumeration of its inhabitants.

A finite set `(s : set F)` is represented as a mask on a finite type `F`:

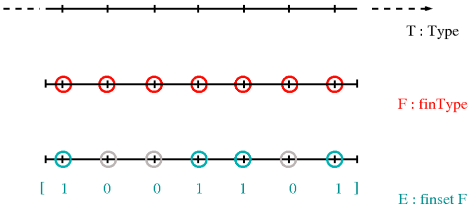


It is a boolean list of fixed length `♯ F`.

Finite sets as finite characteristic functions

A finite type F is an enumeration of its inhabitants.

A finite set `(s : set F)` is represented as a mask on a finite type `F`:



A mask coerces to a characteristic function `(s : F -> bool)`, such that

$$s_1 = s_2 \Leftrightarrow (\forall x, s_1\ x = s_2\ x)$$

Mathematical datas, mathematical structures

Types are used to classify:

- datas

```
Inductive nat := O : nat | S : nat -> nat.
Check 5.
>> 5 : nat
Inductive list (A : Type) :=
nil : list A | cons : list A -> list A.
Check (cons 3 nil).
>> (cons 3 nil) : nat
```

Mathematical datas, mathematical structures

Types are used to classify:

- datas

```
Inductive nat := O : nat | S : nat -> nat.
Check 5.
>> 5 : nat
Inductive list (A : Type) :=
nil : list A | cons : list A -> list A.
Check (cons 3 nil).
>> (cons 3 nil) : nat
```

- mathematical specifications and structures

```
Definition set0 F : {set F} := ...
Definition zint_Ring : ringType := ...
```

Mathematical structures

The previous Σ-types generalize to record types that can be used to represent interfaces of mathematical structures:

```
Structure orderedType := mkOrderedType {
  car : Type;
  ord : car -> car -> Prop;
  anti_ord : antisym ord;
  trans_ord : transitive ord}.
```

Mathematical structures

Organization of the development:

- Definition of the structure

```
Structure zmodType := ZmodType {...}
```

Mathematical structures

Organization of the development:

- Definition of the structure

```
Structure zmodType := ZmodType {...}
```

- Definition of the associated notations

```
Notation "0" := (zero _).
Notation "x + y" := (add x y).
Notation "x - y" := (x + - y).
```

Mathematical structures

Organization of the development:

- Definition of the structure

```
Structure zmodType := ZmodType {...}
```

- Definition of the associated notations

```
Notation "0" := (zero _).
Notation "x + y" := (add x y).
Notation "x - y" := (x + - y).
```

- Definition of the theory shared by any instance of the structure

```
Lemma subr0 x : x - 0 = x. Proof. ... Qed.
```

Mathematical structures

Organization of the development:

- Definition of the structure

```
Structure zmodType := ZmodType {...}
```

- Definition of the associated notations

```
Notation "0" := (zero _).
Notation "x + y" := (add x y).
Notation "x - y" := (x + - y).
```

- Definition of the theory shared by any instance of the structure

```
Lemma subr0 x : x - 0 = x. Proof. ... Qed.
```

- Instanciation of the structure

```
Definition Zp_zmodType := ZmodType 'I_p Zp_modMixin.
```

Mathematical structures

Organization of the development:

- Definition of the structure

```
Structure zmodType := ZmodType {...}
```

- Definition of the associated notations

```
Notation "0" := (zero _).
Notation "x + y" := (add x y).
Notation "x - y" := (x + - y).
```

- Definition of the theory shared by any instance of the structure

```
Lemma subr0 x : x - 0 = x. Proof. ... Qed.
```

- Instanciation of the structure

```
Definition Zp_zmodType := ZmodType 'I_p Zp_modMixin.
```

- For each instance, more specific results, which use the generic theory.

Mathematical structures

The hierarchy of algebraic structures can be designed to achieve the desired:

- Inheritance
- Sharing
- Inference

This is a place where we benefit from working in a type theory:

- Types are used to carry a rich mathematical content
- But the user does not need to provide the whole information since the system implements a type inference mechanism similar to modern functional programming languages.

Implicit content of mathematical notations

It is folklore that a number of mathematical notations carry some implicit content:

- Sometimes implicitly containing the preservation of the structure:

$$G \times H \quad G * H \quad G \cap H \quad G \rtimes H \quad G/H$$

- Sometimes only for the expression to make sense:

$$det(M) := \sum_{s \in S_n} (-1)^{\epsilon_s} \prod_i M_{i,s(i)}$$

Finding a way to infer this implicit content automatically is mandatory in order for a formalization to scale...

Implicit content of mathematical notations

And Coq's type system and implementation does the job:

```
Variable R : ringType.
Definition determinant n (A : 'M_n) : R :=
  \sum_(s : 'S_n) (-1) ^+ s * \prod_i A i (s i).
```

Modeling and specification of algorithms

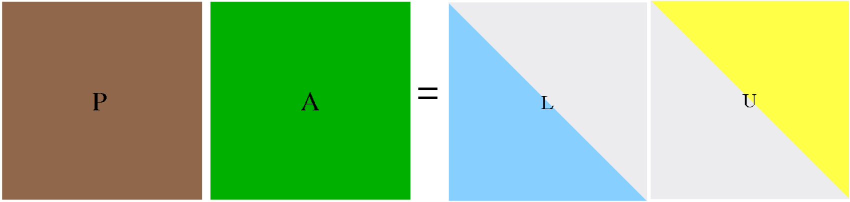
The theory developed for the instances of abstract structures can include the modeling of algorithms.

- Coq can be considered as a pseudo-language for the description of the algorithm.
- The data structures chosen should be the most convenient representations for the proofs.

Example: summation of the first natural numbers.

LUP matrix decomposition

Any square matrix A can be decomposed as:

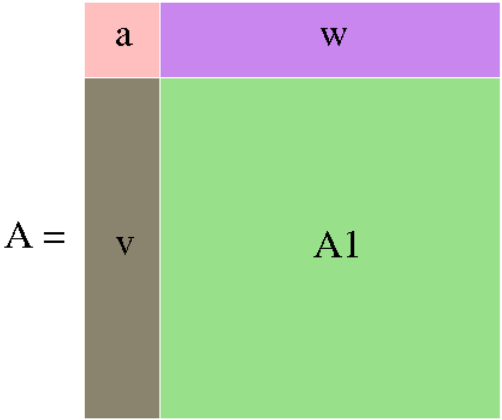


with:

- P a permutation matrix ($=$ possible row swaps)
- L a lower triangular matrix
- U an upper triangular matrix

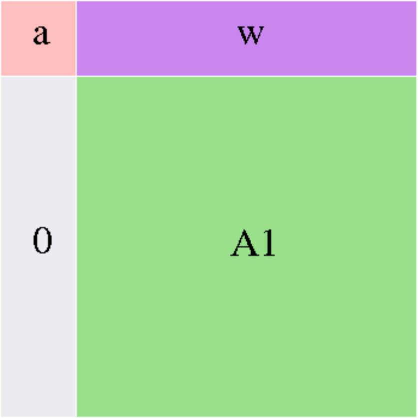
LUP matrix decomposition

By recursion on the size n of A :



LUP matrix decomposition

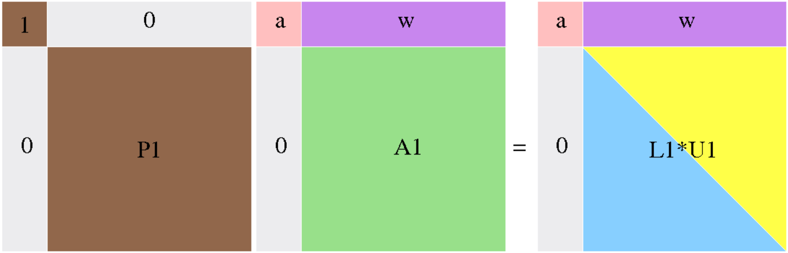
By recursion on the size n of A :



Easy case: when v is zero

LUP matrix decomposition

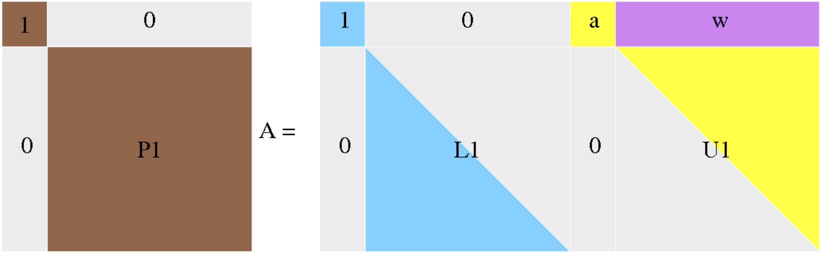
By recursion on the size n of A :



then by recursion hypothesis, $P_1 A_1 = L_1 U_1$

LUP matrix decomposition

By recursion on the size n of A :



we obtain an LUP decomposition.

LUP matrix decomposition

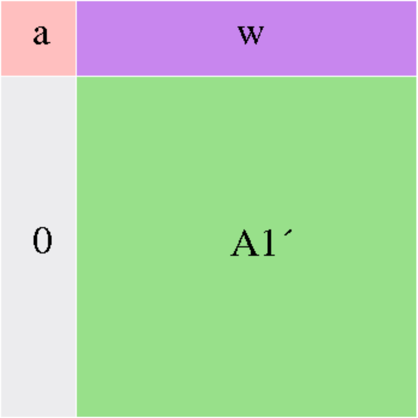
Now in the general case:

$$QA = \begin{bmatrix} a & w \\ v & A1 \end{bmatrix}$$

We use a permutation matrix Q to perform a swap and get $a \neq 0$.

LUP matrix decomposition

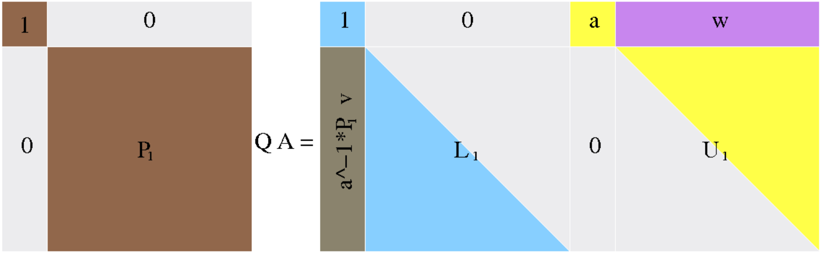
Now in the general case:



- We use this a to annihilate the rest of the first column of QA :
  $A'_1 = A_1 - \text{Schur} \quad \text{with Schur} = a^{-}1 * v * w$
- We apply the recursion hypothesis to A'_1 : $P_1 A'_1 = L_1 U_1$

LUP matrix decomposition

Now in the general case:



Modeling and specification of algorithms

The logic underlying the Coq system is constructive: excluded middle is not an admissible rule, hence classical reasoning is now allowed on an arbitrary statement.

- The `bool` data type is distinct from the `Prop` sort.

Modeling and specification of algorithms

The logic underlying the Coq system is constructive: excluded middle is not an admissible rule, hence classical reasoning is now allowed on an arbitrary statement.

- The `bool` data type is distinct from the `Prop` sort.
- A significant part of the mathematics formalized inside Coq has the flavor presented in this example.

Modeling and specification of algorithms

The logic underlying the Coq system is constructive: excluded middle is not an admissible rule, hence classical reasoning is now allowed on an arbitrary statement.

- The `bool` data type is distinct from the `Prop` sort.
- A significant part of the mathematics formalized inside Coq has the flavor presented in this example.
- A global excluded-middle axiom is not that convenient in practice.

Modeling and specification of algorithms

The logic underlying the Coq system is constructive: excluded middle is not an admissible rule, hence classical reasoning is now allowed on an arbitrary statement.

- The `bool` data type is distinct from the `Prop` sort.
- A significant part of the mathematics formalized inside Coq has the flavor presented in this example.
- A global excluded-middle axiom is not that convenient in practice.
- Programming and proving the correctness of a decision procedure for the first-order theory of a mathematical structure (eg. algebraically closed, real closed fields) legitimates classical reasoning on this fragment inside proofs.

Execution of the algorithms

The previous code cannot be executed as such: the data-structures that are appropriate for proofs are not the efficient ones for computation.

How to link this ideal description with a concrete, executable implementation?

Two possibilities:

- Use a direct translation to a functional programming language
- Work further to obtain an efficient execution inside Coq

Extraction

P. Letouzey (Paris 7)

A mechanism of automated translation, called extraction is available:

- Targets are presently OCaml and Haskell.
- Proofs and specifications are erased.
- One can specify target data-structures.
- The correctness of the extraction mechanism should be trusted.
- The correctness of the language compiler should also be trusted.

Example: the CompCert C(light) compiler (X. Leroy et al.) consists in Ocaml code extracted from a Coq formalization of correctness.

Execution of the algorithms inside Coq

Several levels of optimization can lead to executable programs inside Coq:

- New data-structures, proved correct wrt to the ideal ones;
- Optimized versions of the algorithms;
- Optimized reduction inside Coq (so-called virtual-machine);
- Semi-imperative features: machine integers, arrays.

Note that the two last options increase the size of the trusted code.

Proof automation by computation

Computer algebra systems allow to use a computer to perform computations that are too large, intricate to be tractable by hand by the mathematician.

A formal proof can involve a number of relatively small computational steps, that would become too tedious if not automatized.

Example: the `ring` tactic.

Proof automation by computation

Computer algebra systems allow to use a computer to perform computations that are too large, intricate to be tractable by hand by the mathematician.

A formal proof can involve a number of relatively small computational steps, that would become too tedious if not automatized.

Example: the `ring` tactic.

Application: Primality proving with elliptic curves, G. Hanrot and L. Théry

Certification of external oracles

So far we have seen examples where:

- One programs an algorithm in the Coq language;
- One proves a correctness theorem ensuring a property for any value computed by the program;
- By construction, the program and the specification are objects of the Coq formalism.

One can also sometimes use a lighter approach using computations performed outside Coq, by an untrusted code.

Certification of external oracles

Suppose that:

- You dispose of a binary implementing of a powerful factorization algorithm;

Certification of external oracles

Suppose that:

- You dispose of a binary implementing of a powerful factorization algorithm;
- You want to disprove inside Coq the primality of some natural number n;

Certification of external oracles

Suppose that:

- You dispose of a binary implementing of a powerful factorization algorithm;
- You want to disprove inside Coq the primality of some natural number n;
- You can call the external factorization tool, which computes $p_1, \ldots, p_k$ a factorization of n;

Certification of external oracles

Suppose that:

- You dispose of a binary implementing of a powerful factorization algorithm;
- You want to disprove inside Coq the primality of some natural number n;
- You can call the external factorization tool, which computes $p_1, \ldots, p_k$ a factorization of n;
- You somehow communicate this candidate factorization to Coq;

  (this is pure plumbing)

Certification of external oracles

Suppose that:

- You dispose of a binary implementing of a powerful factorization algorithm;
- You want to disprove inside Coq the primality of some natural number n;
- You can call the external factorization tool, which computes $p_1, \ldots, p_k$ a factorization of n;
- You somehow communicate this candidate factorization to Coq;
  (this is pure plumbing)
- You now only need to check inside Coq, that $n = p_1 \times \cdots \times p_k$;
- And to contradict the definition of the primality of n.

Certification of external oracles

This approach is specially relevant when the property of interest can be characterized by a certificate, which is easier to check than to find.

Examples:

- The `psatz` tactic (F. Besson) which proves positivity of polynomial via sums of squares decomposition (calling `csdp`);
- The `Gb` tactic (L. Pottier) which proves that a polynomial equation is consequence of others via Gröbner basis computations. (calling `F4`)

Combined approaches

In the previous examples, the certificates were relatively small and the correctness theorem deriving a proof from their verification, easy to prove.

This approach can be extended in both directions:

- When certificates are larger, they are called traces: they can be use as a path to reconstruct a Coq proof. Example:
  - Automated generation of proof of properties on numerical programs dealing with floating-point or fixed-point arithmetic Gappa (G. Melquiond)
- When one disposes of sophisticated formal libraries, one can use more intricate correctness theorems. Examples:
  - Primality certificates like Pocklington (B. Grégoire, L. Théry, B. Werner)

Conclusion

The Coq system is a type theory based proof assistant:

- Which allows to take benefit from type inference to infer mathematical implicit content;
- Which allows a special place for computation in the formalism, and optimization in its implementation.

Recent evolutions of the system and of the developed libraries:

- Offer various approaches for the formalization of computer algebra algorithms, with various levels of trusted code;
- Allow to stand on a significant body of formalized mathematical theories (see the Mathematical Components project).