Proofs-as-programs for programmers

Pierre Corbineau and Jean-François MONIN

Disordered summary

- Target audience
- Challenges
- Conventional order
- Contents
- Unusual design decisions

Target audience: software students

Practical motivation

Understand better software pieces

- principled reasoning on programs
- theoretical culture on semantics and notions of computation
- (certified) compilation

But

- Don't expect anything about maths
  don't like, hate, are afraid of, not confident
- Don't teach math, but some skills in mathematical thinking
- Science (including programming) = mixture of rigour and creativity

Target audience: software students

Practical motivation

Understand better software pieces

- principled reasoning on programs
- theoretical culture on semantics and notions of computation
- (certified) compilation

But

- Don't expect anything about maths
  don't like, hate, are afraid of, not confident
- Don't teach math, but some skills in mathematical thinking
- Science (including programming) = mixture of rigour and creativity

Conventional order

Coq'Art, SF, author's first courses...

- Propositional logic; first order quantifiers
- Basic data: `bool`, `nat`; simple (recursive) programs
- Induction on `nat`
- Lists (monomorphic, polymorphic); tree-like data-structures
- "Rich" data-types (e.g., `sig` types)
- Inductive relations
- Applications to
  - maths
  - Computer Science: program correctness
  - Computer Science: semantics
  - ...
- Exotic topics, e.g., Curry-Howard correspondence

Conventional order

Coq'Art, SF, author's first courses...

- Propositional logic; first order quantifiers
- Basic data: `bool`, `nat`; simple (recursive) programs
- Induction on `nat`
- Lists (monomorphic, polymorphic); tree-like data-structures
- "Rich" data-types (e.g., `sig` types)
- Inductive relations
- Applications to
  - maths
  - Computer Science: program correctness
  - Computer Science: semantics
  - ...
- Exotic topics, e.g., Curry-Howard correspondence

Conventional order

Coq'Art, SF, author's first courses...

- Propositional logic; first order quantifiers
- Basic data: `bool`, `nat`; simple (recursive) programs
- Induction on `nat`
- Lists (monomorphic, polymorphic); tree-like data-structures
- "Rich" data-types (e.g., `sig` types)
- Inductive relations
- Applications to
  - maths
  - Computer Science: program correctness
  - Computer Science: semantics
  - ...
- Exotic topics, e.g., Curry-Howard correspondence

Conventional order

Coq'Art, SF, author's first courses...

- Propositional logic; first order quantifiers
- Basic data: `bool`, `nat`; simple (recursive) programs
- Induction on `nat`
- Lists (monomorphic, polymorphic); tree-like data-structures
- "Rich" data-types (e.g., `sig` types)
- Inductive relations
- Applications to
  - maths
  - Computer Science: program correctness
  - Computer Science: semantics
  - ...
- Exotic topics, e.g., Curry-Howard correspondence

Conventional order

Coq'Art, SF, author's first courses...

- Propositional logic; first order quantifiers
- Basic data: `bool`, `nat`; simple (recursive) programs
- Induction on `nat`
- Lists (monomorphic, polymorphic); tree-like data-structures
- "Rich" data-types (e.g., `sig` types)
- Inductive relations
- Applications to
  - maths
  - Computer Science: program correctness
  - Computer Science: semantics
  - ...
- Exotic topics, e.g., Curry-Howard correspondence

Conventional order

Coq'Art, SF, author's first courses...

- Propositional logic; first order quantifiers
- Basic data: `bool`, `nat`; simple (recursive) programs
- Induction on `nat`
- Lists (monomorphic, polymorphic); tree-like data-structures
- "Rich" data-types (e.g., `sig` types)
- Inductive relations
- Applications to
  - maths
  - Computer Science: program correctness
  - Computer Science: semantics
  - ...
- Exotic topics, e.g., Curry-Howard correspondence

Conventional order, assessment

Good points

- simple $\to$ complex
- theory $\to$ applications

Drawbacks

- slow: interesting things come late
- mysterious interactions

Possible repair: make things closer to usual math

- different interface
- hide more
- use more automation

Conventional order, assessment

Good points

- simple $\rightarrow$ complex
- theory $\rightarrow$ applications

Drawbacks

- slow: interesting things come late
- mysterious interactions

Possible repair: make things closer to usual math

- different interface
- hide more
- use more automation

Conventional order, assessment

Good points

- simple $\to$ complex
- theory $\to$ applications

Drawbacks

- slow: interesting things come late
- mysterious interactions

Possible repair: make things closer to usual math

- different interface
- hide more
- use more automation

Contract: don't cheat

We don't want students to cheat

We should not cheat as well (as much as possible)

The more explicit and transparent, the better

Use automation only when what happens is perfectly understood

Contract: don't cheat

We don't want students to cheat

We should not cheat as well (as much as possible)

The more explicit and transparent, the better

Use automation only when what happens is perfectly understood

Issues

Math-oriented people

- Bizarre notations (e.g., function application)
- Interesting theories come late
- Why bother with Curry-Howard?

Computer-oriented people

- Requires too much patience
- Amazing and funny things come late

Moreover, Coq's feedback sometimes hard to understand

Two languages: formulae and scripts

Formulae

More or less familiar

Tactics

- Collection of receipes
  you face this typical situation, use that tactic.
- Alternative: a little bit of proof-theory?
  CIC is complex and subtle, cannot be a starting point.

Two languages: formulae and scripts

Formulae

More or less familiar

Tactics

- Collection of receipes
  you face this typical situation, use that tactic.
- Alternative: a little bit of proof-theory?
  CIC is complex and subtle, cannot be a starting point.

Alternative: go fast

But carefully!

Expose interesting / challenging examples as soon as possible

RELY ON PROGRAMMERS INTUITIONS

- trees, easy to visualize
- (structural) recursive programs on them

Alternative: go fast

But carefully!

Expose interesting / challenging examples as soon as possible

RELY ON PROGRAMMERS INTUITIONS

- trees, easy to visualize
- (structural) recursive programs on them

Trees everywhere

- Easy and common data structure
- Abstract Syntax Trees
- Proof trees
  structured thinking
- Typing rules
- (Structured, big step, small step) Operational Semantics
- All kinds of inductive definitions

This is consistent with Type Theory

Focus on functional programming,
not on proof-theory

Thank you Curry-Howard

- Implication and universal quantification seen as types for functional programs
- Structural induction seen in the same way as structural recursion
- Natural explanation of implication (>> truth tables)

First steps

Basic types

- `bool` is not that good: interferences with logic
  $\rightarrow$ take another enumerated type (e.g., traffic colors)
- binary trees and ASTs for simple arithmetic expressions

Functions

- curryfied syntax
- pattern-matching

Simple theorems and basic tactics

- equalities: `reflexivity` and `rewrite`
- `intro`duction of hypotheses or variables, computation steps (`cbn`)
- reasoning by case: `destruct`
- induction on trees

First steps

Basic types

- bool is not that good: interferences with logic
  → take another enumerated type (e.g., traffic colors)
- binary trees and ASTs for simple arithmetic expressions

Functions

- curryfied syntax
- pattern-matching

Simple theorems and basic tactics

- equalities: reflexivity and rewrite
- introduction of hypotheses or variables, computation steps (cbn)
- reasoning by case: destruct
- induction on trees

First steps

Basic types

- `bool` is not that good: interferences with logic
  → take another enumerated type (e.g., traffic colors)
- binary trees and ASTs for simple arithmetic expressions

Functions

- curryfied syntax
- pattern-matching

Simple theorems and basic tactics

- equalities: `reflexivity` and `rewrite`
- `intro`duction of hypotheses or variables, computation steps (`cbn`)
- reasoning by case: `destruct`
- induction on trees

Smooth introduction to the proofs-as-program paradigm

Interactive development of functions

Proofs of implicational / universal theorems are functions

Reasoning by case = programming by case

Deep patterns can be useful (with `refine`)

Smooth introduction to the proofs-as-program paradigm

Interactive development of functions

Proofs of implicational / universal theorems are functions

Reasoning by case = programming by case
Deep patterns can be useful (with `refine`)

Basic example: traffic colors

```
Inductive tlcolor : Set :=
  | Green  : tlcolor
  | Orange : tlcolor
  | Red    : tlcolor.

Definition next_col : tlcolor -> tlcolor :=
  fun c =>
    match c with
    | Green  => Orange
    | Orange => Red
    | Red    => Green
    end.

Lemma nextnextnext_id :
  forall c:tlcolor, next_col (next_col (next_col c)) = c.
```

Basic example: traffic colors

```
Inductive tlcolor : Set :=
  | Green  : tlcolor
  | Orange : tlcolor
  | Red    : tlcolor.

Definition next_col : tlcolor -> tlcolor :=
  fun c =>
    match c with
    | Green  => Orange
    | Orange => Red
    | Red    => Green
    end.


Lemma nextnextnext_id :
  forall c:tlcolor, next_col (next_col (next_col c)) = c.
```

Challenging example 1, on binary trees

```
Fixpoint revt t : bintree :=
  match t with
  | L c   => L c
  | N l r => N (revt r) (revt l)
  end.

Theorem revt_revt : forall t, revt (revt t) = t.
```

- (for teachers) it is cheap!
  much simpler as the corresponding result on lists
- (for students)
  try the same exercise in your favorite imperative/OO progr. lang.

Challenging example 1, on binary trees

```
Fixpoint revt t : bintree :=
  match t with
  | L c   => L c
  | N l r => N (revt r) (revt l)
  end.

Theorem revt_revt : forall t, revt (revt t) = t.
```

- (for teachers) it is cheap!
  much simpler as the corresponding result on lists
- (for students)
  try the same exercise in your favorite imperative/OO progr. lang.

Challenging example 1, on binary trees

```
Fixpoint revt t : bintree :=
  match t with
  | L c   => L c
  | N l r => N (revt r) (revt l)
  end.

Theorem revt_revt : forall t, revt (revt t) = t.
```

- (for teachers) it is cheap!
  much simpler as the corresponding result on lists
- (for students)
  try the same exercise in your favorite imperative/OO progr. lang.

Funny example: functions returning a type, dependent types

```
Definition waow : tlcolor -> Set :=
  fun c =>
    match c return Set with
    | Green => nat
    | Orange => tlcolor
    | Red => tlcolor -> nat
    end.

Definition waow_waow : forall c : tlcolor, waow c :=
  fun c =>
    match c return waow c with
    | Green => 2
    | Orange => Green
    | Red => fun c' => match c' with Orange => 6 | _ => 1 end
    end.
```

Predicate = dependent type, Curry-Howard at work

```
Lemma nextnextnext_id :
  forall c:tlcolor, next_col (next_col (next_col c)) = c.
Proof.
  refine (fun c => _).
  refine (match c with   (* Reasoning by case ([destruct c]) *)
          | Green => _
          | Orange => _
          | Red => _
          end).
  all:exact refl_equal.
Qed.
```

```
Definition id_nextnextnext_pgm :
  forall c, c = next_col (next_col (next_col c)) :=
  fun c =>  match c with
             | Green  => eq_refl
             | Orange => eq_refl
             | Red    => eq_refl
             end.
```

Predicate = dependent type, Curry-Howard at work

```
Lemma nextnextnext_id :
  forall c:tlcolor, next_col (next_col (next_col c)) = c.
Proof.
  refine (fun c => _).
  refine (match c with   (* Reasoning by case ([destruct c]) *)
          | Green => _
          | Orange => _
          | Red => _
          end).
  all:exact refl_equal.
Qed.

Definition id_nextnextnext_pgm :
  forall c, c = next_col (next_col (next_col c)) :=
  fun c =>  match c with
             | Green  => eq_refl
             | Orange => eq_refl
             | Red    => eq_refl
             end.
```

Challenging example 2: code optimization

```
Inductive aexp : Set :=
| Cst : nat -> aexp
| Apl : aexp -> aexp -> aexp
| Amu : aexp -> aexp -> aexp.


Fixpoint eval (a : aexp) : nat :=
```

Challenging example 2 (cont'd): syntactic simplification on ASTs

```
           Apl
simpl0    / \        =  a2            (* 0 + a2 = a2 *)
         Cst  a2
          |
          0

           Amu          Cst
simpl0    / \        =   |            (* 0 * a2 = 0  *)
         Cst  a2         0
          |
          0

simpl0    a           =  a        in all other cases
```

Challenging example 2 (cont'd): correcness of the optimization

Use ad-hoc case analysis

```
Lemma eval_simpl0: forall a, eval (simpl0 a) = eval a.
Proof.
  intro a.
  refine ( match a with
           | Apl (Cst 0) a2 => _
           | Amu (Cst 0) a2 => _
           | a'             => eq_refl (eval a')
           end).
```

```
Fixpoint simpl_rec (a : aexp) :=

Lemma eval_simpl_rec: forall a, eval (simpl_rec a) = eval a.
```

Challenging example 2 (cont'd): correcness of the optimization

Use ad-hoc case analysis

```
Lemma eval_simpl0: forall a, eval (simpl0 a) = eval a.
Proof.
  intro a.
  refine ( match a with
           | Apl (Cst 0) a2 => _
           | Amu (Cst 0) a2 => _
           | a'             => eq_refl (eval a')
           end).
```

```
Fixpoint simpl_rec (a : aexp) :=

Lemma eval_simpl_rec: forall a, eval (simpl_rec a) = eval a.
```

“Absurd” hypotheses → contagious equalities

Explaining `discriminate` without black magic (nor large eliminations to some extent).

```
Lemma absurd: 5 = 4 -> 15 = 12.
```

```
Lemma true_false_eq : true = false -> forall n1 n2 : nat, n1 = n2.
```

```
Lemma eq_eqnatb : forall n1 n2, eqnatb n1 n2 = true -> n1 = n2.
```

Then we can get `discriminate` using large eliminations, seen earlier in the `waow` function.

Next step: (small) inversion

“Absurd” hypotheses → contagious equalities

Explaining discriminate without black magic (nor large eliminations to some extent).

```
Lemma absurd: 5 = 4 -> 15 = 12.

Lemma true_false_eq : true = false -> forall n1 n2 : nat, n1 = n2.

Lemma eq_eqnatb : forall n1 n2, eqnatb n1 n2 = true -> n1 = n2.
```

Then we can get discriminate using large eliminations, seen earlier in the waow function.

Next step: (small) inversion

"Absurd" hypotheses → contagious equalities

Explaining `discriminate` without black magic (nor large eliminations to some extent).

```
Lemma absurd: 5 = 4 -> 15 = 12.

Lemma true_false_eq : true = false -> forall n1 n2 : nat, n1 = n2.

Lemma eq_eqnatb : forall n1 n2, eqnatb n1 n2 = true -> n1 = n2.
```

Then we can get `discriminate` using large eliminations, seen earlier in the `waow` function.

Next step: (small) inversion

“Absurd” hypotheses → contagious equalities

Explaining discriminate without black magic (nor large eliminations to some extent).

```
Lemma absurd: 5 = 4 -> 15 = 12.

Lemma true_false_eq : true = false -> forall n1 n2 : nat, n1 = n2.

Lemma eq_eqnatb : forall n1 n2, eqnatb n1 n2 = true -> n1 = n2.
```

Then we can get discriminate using large eliminations, seen earlier in the waow function.

Next step: (small) inversion

“Absurd” hypotheses → contagious equalities

Explaining `discriminate` without black magic (nor large eliminations to some extent).

```
Lemma absurd: 5 = 4 -> 15 = 12.

Lemma true_false_eq : true = false -> forall n1 n2 : nat, n1 = n2.

Lemma eq_eqnatb : forall n1 n2, eqnatb n1 n2 = true -> n1 = n2.
```

Then we can get `discriminate` using large eliminations, seen earlier in the `waow` function.

Next step: (small) inversion

Logic

No pressure to talk about

- constructive / classical logic
- negation
  - needed lemmas happen to be stated in a positive way
  - deal with "absurd" hypotheses in a positive way as well
- conjunction, disjunction and existential quantifier
  (not really needed)
- truth
  (not really needed)

Because

- equalities, at the beginning
- custom inductive types, later

are enough

Logic

No pressure to talk about

- constructive / classical logic
- negation
    - needed lemmas happen to be stated in a positive way
    - deal with "absurd" hypotheses in a positive way as well
- conjunction, disjunction and existential quantifier
  (not really needed)
- truth
  (not really needed)

Because

- equalities, at the beginning
- custom inductive types, later

are enough

Logic

No pressure to talk about

- constructive / classical logic
- negation
  - needed lemmas happen to be stated in a positive way
  - deal with “absurd” hypotheses in a positive way as well
- conjunction, disjunction and existential quantifier
  (not really needed)
- truth
  (not really needed)

Because

- equalities, at the beginning
- custom inductive types, later

are enough

Logic

No pressure to talk about

- constructive / classical logic
- negation
  - needed lemmas happen to be stated in a positive way
  - deal with "absurd" hypotheses in a positive way as well
- conjunction, disjunction and existential quantifier
  (not really needed)
- truth
  (not really needed)

Because

- equalities, at the beginning
- custom inductive types, later

are enough

Logic

No pressure to talk about

- constructive / classical logic
- negation
  - needed lemmas happen to be stated in a positive way
  - deal with “absurd” hypotheses in a positive way as well
- conjunction, disjunction and existential quantifier
  (not really needed)
- truth
  (not really needed)

Because

- equalities, at the beginning
- custom inductive types, later

are enough

Conclusions

7 years at M1 level (engineering school, $\sim$ 50 students/year)

Feedback was never bad and has improved to "reasonably good" in the recent years.

Show simple amazing things

We tried to introduce complex notions on simpl(e | istic) examples.

Rely on (functional) programmers intuition

Yet another reason to teach functional programming to scientists.

Don't be afraid of inductive types

There is room for them next to numbers and other mathematical notions.

Conclusions

7 years at M1 level (engineering school, $\sim$ 50 students/year)

Feedback was never bad and has improved to "reasonably good" in the recent years.

Show simple amazing things

We tried to introduce complex notions on simpl(e | istic) examples.

Rely on (functional) programmers intuition

Yet another reason to teach functional programming to scientists.

Don't be afraid of inductive types

There is room for them next to numbers and other mathematical notions.

Conclusions

7 years at M1 level (engineering school, $\sim$ 50 students/year)

Feedback was never bad and has improved to "reasonably good" in the recent years.

Show simple amazing things

We tried to introduce complex notions on simpl(e | istic) examples.

Rely on (functional) programmers intuition

Yet another reason to teach functional programming to scientists.

Don't be afraid of inductive types

There is room for them next to numbers and other mathematical notions.

Conclusions

7 years at M1 level (engineering school, $\sim$ 50 students/year)

Feedback was never bad and has improved to "reasonably good" in the recent years.

Show simple amazing things

We tried to introduce complex notions on simpl(e | istic) examples.

Rely on (functional) programmers intuition

Yet another reason to teach functional programming to scientists.

Don't be afraid of inductive types

There is room for them next to numbers and other mathematical notions.