Improving the Proof Experience in Coq

Martin Bodin Federico Olmedo
University of Chile

ICSEC Kick-off Workshop
Santiago, Chile — March 2018

What is this talk about?

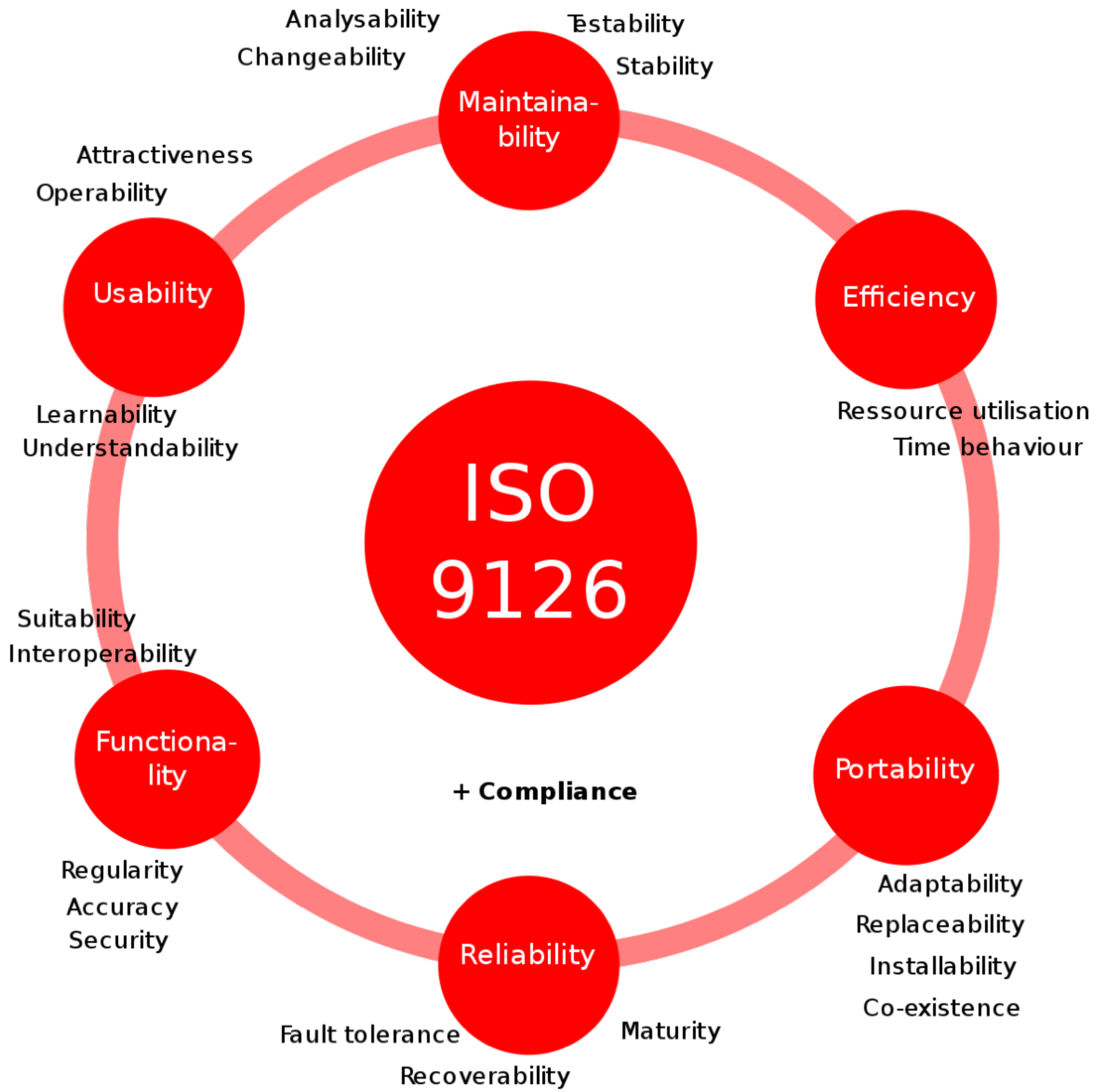
Certified
Cryptography

Proof about
R / JavaScript
programs

Coq User Experience & Wishlist

1

Software quality attributes



Proof developers **tend to** *neglect* **elementary engineering qualities**

Proof developers **tend to** ***neglect*** **elementary engineering qualities —mainly** ***robustness*****.**

Practices precluding the robustness of Coq developments

Practices precluding the robustness of Coq developments

Proof scripts that are sensitive to the naming of automatically generated terms

Practices precluding the robustness of Coq developments

Proof scripts that are sensitive to the naming of automatically generated terms

```
Inductive exp : Set :=
| Const : nat -> exp
| Plus : exp -> exp -> exp.
```

Practices precluding the robustness of Coq developments

Proof scripts that are sensitive to the naming of automatically generated terms

```
Inductive exp : Set :=
| Const : nat -> exp
| Plus : exp -> exp -> exp.

Fixpoint eval (e : exp) : nat :=
  match e with
    | Const n => n
    | Plus e1 e2 => eval e1 + eval e2
  end.
```

Practices precluding the robustness of Coq developments

Proof scripts that are sensitive to the naming of automatically generated terms

```
Inductive exp : Set :=
| Const : nat -> exp
| Plus : exp -> exp -> exp.

Fixpoint eval (e : exp) : nat :=
  match e with
    | Const n => n
    | Plus e1 e2 => eval e1 + eval e2
  end.

Fixpoint times (k : nat) (e : exp) : exp :=
  match e with
    | Const n => Const (k * n)
    | Plus e1 e2 => Plus (times k e1) (times k e2)
  end.
```

Practices precluding the robustness of Coq developments

Proof scripts that are sensitive to the naming of automatically generated terms

```
Theorem eval_times : forall k e,
  eval (times k e) = k * eval e.
```

Practices precluding the robustness of Coq developments

Proof scripts that are sensitive to the naming of automatically generated terms

```
Theorem eval_times : forall k e,
  eval (times k e) = k * eval e.
Proof.
```

Practices precluding the robustness of Coq developments

Proof scripts that are sensitive to the naming of automatically generated terms

```
Theorem eval_times : forall k e,
  eval (times k e) = k * eval e.
Proof.
  induction e.
```

```
k, n : nat
________________________________________
eval (times k (Const n)) = k * eval (Const n)
```

Practices precluding the robustness of Coq developments

Proof scripts that are sensitive to the naming of automatically generated terms

```
Theorem eval_times : forall k e,
  eval (times k e) = k * eval e.
Proof.
  induction e.
    trivial.
```

```
k, n : nat
______________________________________
eval (times k (Const n)) = k * eval (Const n)
```

Practices precluding the robustness of Coq developments

Proof scripts that are sensitive to the naming of automatically generated terms

```
Theorem eval_times : forall k e,
  eval (times k e) = k * eval e.
Proof.
  induction e.
    trivial.
```

```
k : nat
e1, e2 : exp
IHe1 : eval (times k e1) = k * eval e1
IHe2 : eval (times k e2) = k * eval e2
______________________________________

eval (times k (Plus e1 e2)) = k * eval (Plus e1 e2)
```

Practices precluding the robustness of Coq developments

Proof scripts that are sensitive to the naming of automatically generated terms

```
Theorem eval_times : forall k e,
  eval (times k e) = k * eval e.
Proof.
  induction e.
    trivial.

    simpl.
```

```
k : nat
e1, e2 : exp
IHe1 : eval (times k e1) = k * eval e1
IHe2 : eval (times k e2) = k * eval e2
________________________________________

eval (times k e1) + eval (times k e2) =
k * (eval e1 + eval e2)
```

Practices precluding the robustness of Coq developments

Proof scripts that are sensitive to the naming of automatically generated terms

```
Theorem eval_times : forall k e,
  eval (times k e) = k * eval e.
Proof.
  induction e.
    trivial.

    simpl.
    rewrite IHe1.
    rewrite IHe2.
```

```
k : nat
e1, e2 : exp
IHe1 : eval (times k e1) = k * eval e1
IHe2 : eval (times k e2) = k * eval e2
____________________________________________

k * eval e1 + k * eval e2 =
k * (eval e1 + eval e2)
```

Practices precluding the robustness of Coq developments

Proof scripts that are sensitive to the naming of automatically generated terms

```
Theorem eval_times : forall k e,
  eval (times k e) = k * eval e.
Proof.
  induction e.
    trivial.

    simpl.
    rewrite IHe1.
    rewrite IHe2.
    rewrite mul_add_distr_l.
     trivial.
```

```
k : nat
e1, e2 : exp
IHe1 : eval (times k e1) = k * eval e1
IHe2 : eval (times k e2) = k * eval e2
______________________________________

k * eval e1 + k * eval e2 =
k * (eval e1 + eval e2)
```

Practices precluding the robustness of Coq developments

Proof scripts that are sensitive to the naming of automatically generated terms

```
Theorem eval_times : forall k e,
  eval (times k e) = k * eval e.
Proof.
  induction e.
    trivial.

    simpl.
    rewrite IHe1.
    rewrite IHe2.
    rewrite mul_add_distr_l.
     trivial.
Qed.
```

```
k : nat
e1, e2 : exp
IHe1 : eval (times k e1) = k * eval e1
IHe2 : eval (times k e2) = k * eval e2
______________________________________

k * eval e1 + k * eval e2 =
k * (eval e1 + eval e2)
```

✓

Practices precluding the robustness of Coq developments

Proof scripts that are sensitive to the naming of automatically generated terms

Replace
e with x

```
Theorem eval_times : forall k e,
  eval (times k e) = k * eval e.
Proof.
  induction e.
    trivial.

    simpl.
    rewrite IHe1.
    rewrite IHe2.
    rewrite mul_add_distr_l.
     trivial.
Qed.
```

```
k : nat
e1, e2 : exp
IHe1 : eval (times k e1) = k * eval e1
IHe2 : eval (times k e2) = k * eval e2
________________________________________

k * eval e1 + k * eval e2 =
k * (eval e1 + eval e2)
```

Practices precluding the robustness of Coq developments

Proof scripts that are sensitive to the naming of automatically generated terms

```
Theorem eval_times : forall k x,
  eval (times k x) = k * eval x.
```

Practices precluding the robustness of Coq developments

Proof scripts that are sensitive to the naming of automatically generated terms

```
Theorem eval_times : forall k x,
  eval (times k x) = k * eval x.
Proof.
```

Practices precluding the robustness of Coq developments

Proof scripts that are sensitive to the naming of automatically generated terms

```
Theorem eval_times : forall k x,
  eval (times k x) = k * eval x.
Proof.
  induction x.
```

```
k, n : nat
______________________________________
eval (times k (Const n)) = k * eval (Const n)
```

Practices precluding the robustness of Coq developments

Proof scripts that are sensitive to the naming of automatically generated terms

```
Theorem eval_times : forall k x,
  eval (times k x) = k * eval x.
Proof.
  induction x.
    trivial.
```

```
k, n : nat
______________________________________
eval (times k (Const n)) = k * eval (Const n)
```

Practices precluding the robustness of Coq developments

Proof scripts that are sensitive to the naming of automatically generated terms

```
Theorem eval_times : forall k x,
  eval (times k x) = k * eval x.
Proof.
  induction x.
    trivial.
```

```
k : nat
x1, x2 : exp
IHx1 : eval (times k x1) = k * eval x1
IHx2 : eval (times k x2) = k * eval x2
______________________________________________
eval (times k (Plus x1 x2)) = k * eval (Plus x1 x2)
```

Practices precluding the robustness of Coq developments

Proof scripts that are sensitive to the naming of automatically generated terms

```
Theorem eval_times : forall k x,
  eval (times k x) = k * eval x.
Proof.
  induction x.
    trivial.

    simpl.
```

```
k : nat
x1, x2 : exp
IHx1 : eval (times k x1) = k * eval x1
IHx2 : eval (times k x2) = k * eval x2
________________________________________
k * eval x1 + k * eval x2 =
k * (eval x1 + eval x2)
```

Practices precluding the robustness of Coq developments

Proof scripts that are sensitive to the naming of automatically generated terms

```
Theorem eval_times : forall k x,
  eval (times k x) = k * eval x.
Proof.
  induction x.
    trivial.

    simpl.
    rewrite IHe1.
```

```
k : nat
x1, x2 : exp
IHx1 : eval (times k x1) = k * eval x1
IHx2 : eval (times k x2) = k * eval x2
______________________________________

k * eval x1 + k * eval x2 =
k * (eval x1 + eval x2)
```

The reference **IHe1** was not found in the current environment!!!

Practices precluding the robustness of Coq developments

Practices precluding the robustness of Coq developments

Proof scripts that are sensitive to the order of constructors of inductive types.

Practices precluding the robustness of Coq developments

Proof scripts that are sensitive to the order of constructors of inductive types.

```
Inductive exp : Set :=
| Plus : exp -> exp -> exp
| Const : nat -> exp.
```

Flipped the
order of constr.

Practices precluding the robustness of Coq developments

Proof scripts that are sensitive to the order of constructors of inductive types.

```
Inductive exp : Set :=
| Plus : exp -> exp -> exp
| Const : nat -> exp.
```

Flipped the
order of constr.

```
Theorem eval_times : forall k e,
  eval (times k e) = k * eval e.
Proof.
  induction e.
    trivial.

    simpl.
    rewrite IHe1.
    rewrite IHe2.
    rewrite mul_add_distr_l.
     trivial.
Qed.
```

Practices precluding the robustness of Coq developments

Proof scripts that are sensitive to the order of constructors of inductive types.

```
Inductive exp : Set :=
| Plus : exp -> exp -> exp
| Const : nat -> exp.
```

Flipped the order of constr.

```
Theorem eval_times : forall k e,
  eval (times k e) = k * eval e.
Proof.
  induction e.
    trivial.

    simpl.
    rewrite IHe1.
    rewrite IHe2.
    rewrite mul_add_distr_l.
     trivial.
Qed.
```

Attempt to save an incomplete proof

Practices precluding the robustness of Coq developments

Practices precluding the robustness of Coq developments

Proof scripts that are sensitive to the order of lemmas’ hypotheses

Proof developers **tend to** *neglect* **elementary engineering qualities—mainly** *robustness***.**

Proof developers **tend to *neglect* elementary engineering qualities—mainly *robustness*.**

POSSIBLE SOLUTION:

Proof developers tend to *neglect* elementary engineering qualities—mainly *robustness*.

Possible Solution:

- "Proof analysis" identifying possible robustness issues
- Provide a linter implementing the analysis

2

Terrific formalisation in Coq

Terrific formalisation
in Coq

Why not extend
the result?

Terrific formalisation
in Coq

Why not extend
the result?

Ok! Let’s see
what it takes.

Terrific formalisation in Coq

Why not extend the result?

Ok! Let's see what it takes.

- How shall I do it?
  - What is the best way to implement it?

Terrific formalisation in Coq

Why not extend the result?

Ok! Let's see what it takes.

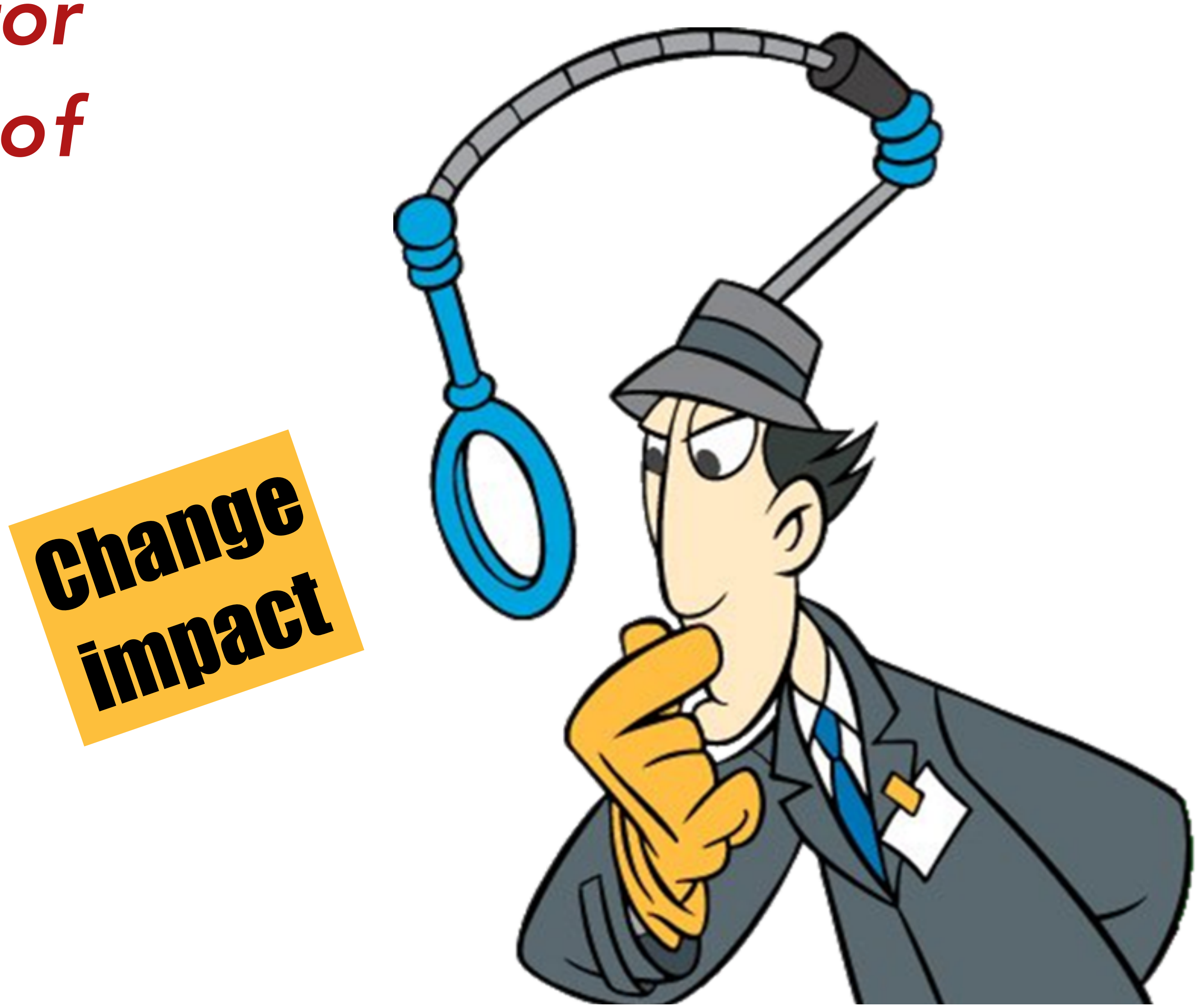
- How shall I do it?

  What is the best way to implement it?

- How much effort would it take?

  Is it really feasible?

Coq developments tend to evolve over time. However, there is *no mechanism for assessing the impact of* introducing *changes*.



How does a change to a part of the development impact on the rest of the development?

How does a change to a part of the development impact on the rest of the development?

- What else should be changed?

HOW DOES A CHANGE TO A PART OF THE DEVELOPMENT IMPACT ON THE REST OF THE DEVELOPMENT?

- What else should be changed?
- What do these changes consist in: extension, removal, adaptation?

How does a change to a part of the development impact on the rest of the development?

- What else should be changed?
- What do these changes consist in: extension, removal, adaptation?
- Where should these changes take exactly place?

Desired tool support

Binary trees with elements in leaves

```
Inductive tree (A : Set) : Set :=
| Leaf : A -> tree A
| Node : tree A -> tree A -> tree A.


Fixpoint size_tree (A : Set)  (t : tree A) : nat :=
  match t with
    | Leaf _ => 1
    | Node t1 t2  => 1 + (size_tree t1) + (size_tree t2)
  end.
```

Desired tool support

Binary trees with elements in leaves and internal nodes

```
Inductive tree (A : Set) : Set :=
| Leaf : A -> tree A
| Node : A -> tree A -> tree A -> tree A.


Fixpoint size_tree (A : Set)  (t : tree A) : nat :=
  match t with
    | Leaf _ => 1
    | Node t1 t2  => 1 + (size_tree t1) + (size_tree t2)
  end.
```

Desired tool support

Binary trees with elements in leaves and internal nodes

```
Inductive tree (A : Set) : Set :=
| Leaf : A -> tree A
| Node : A -> tree A -> tree A -> tree A.


Fixpoint size_tree (A : Set)  (t : tree A) : nat :=
  match t with
    | Leaf _ => 1
    | Node t1 t2  => 1 + (size_tree t1) + (size_tree t2)
  end.
```

Desired tool support

Binary trees with elements in leaves and internal nodes

```
Inductive tree (A : Set) : Set :=
| Leaf : A -> tree A
| Node : A -> tree A -> tree A -> tree A.
```

```
Fixpoint size_tree (A : Set)  (t : tree A) : nat :=
  match t with
    | Leaf _ => 1
    | Node t1 t2  => 1 + (size_tree t1) + (size_tree t2)
  end.
```

Requires attention
- *Constructor has changed*
- *Adapt return expression?*

Desired tool support

Binary trees with elements in leaves and internal nodes

```
Inductive tree (A : Set) : Set :=
| Leaf : A -> tree A
| Node : A -> tree A -> tree A -> tree A.
```

Requires attention
- *Constructor has changed*
- *Adapt return expression?*

```
Fixpoint size_tree (A : Set)  (t : tree A) : nat :=
  match t with
    | Leaf _ => 1
    | Node t1 t2  => 1 + (size_tree t1) + (size_tree t2)
  end.
```

```
Lemma size_map_mirror_tree : forall (A B : Set) (f : A -> B) (t : tree A),
 size_tree (map_tree f t) = size_tree (mirror_tree  t).
Proof.
  intros.
  rewrite size_map_tree, size_mirror_tree.
  trivial.
Qed.
```

Desired tool support

Binary trees with elements in leaves and internal nodes

```
Inductive tree (A : Set) : Set :=
| Leaf : A -> tree A
| Node : A -> tree A -> tree A -> tree A.
```

Requires attention
- *Constructor has changed*
- *Adapt return expression?*

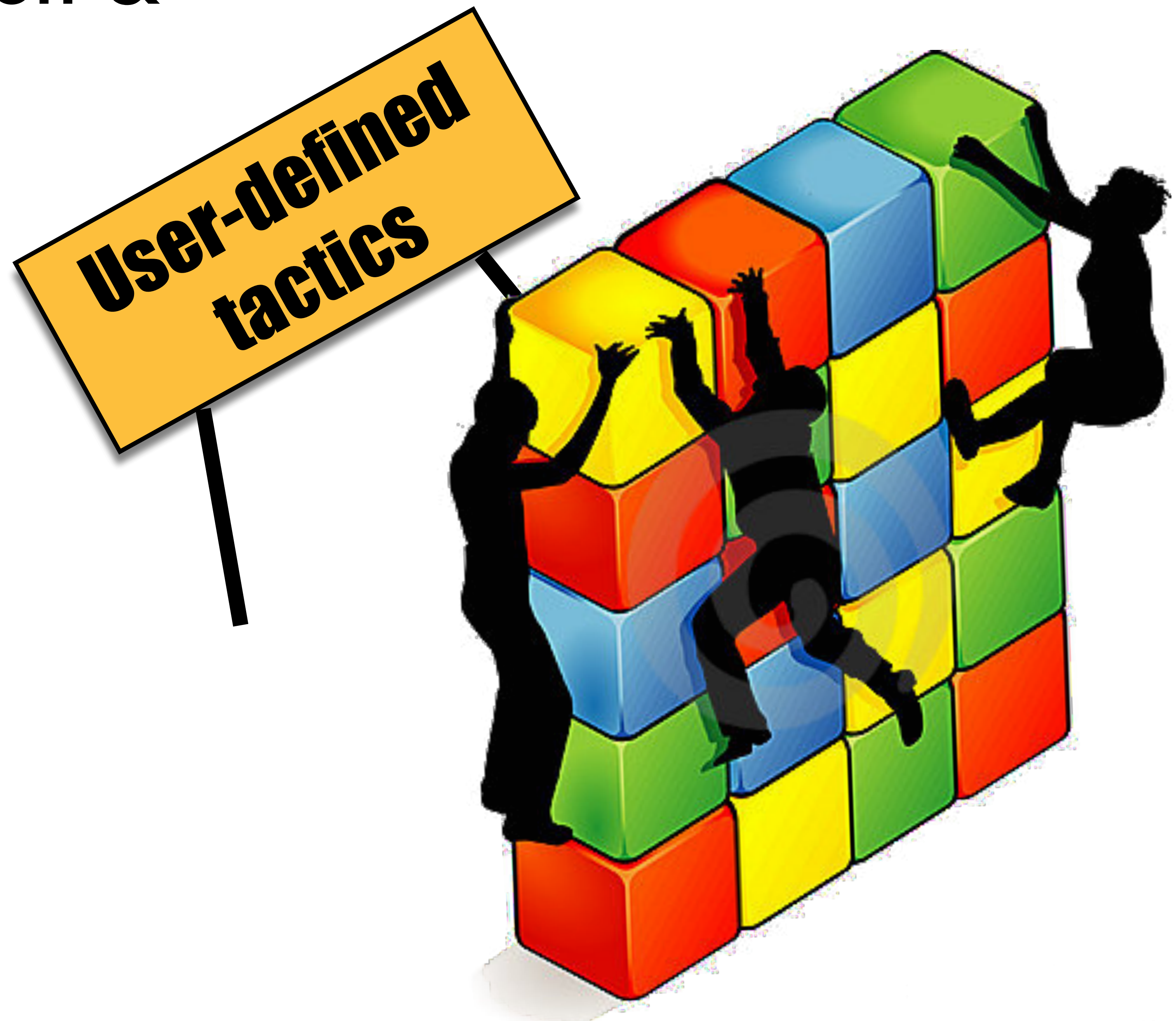
```
Fixpoint size_tree (A : Set)  (t : tree A) : nat :=
  match t with
    | Leaf _ => 1
    | Node t1 t2  => 1 + (size_tree t1) + (size_tree t2)
  end.
```

```
Lemma size_map_mirror_tree : forall (A B : Set) (f : A -> B) (t : tree A),
 size_tree (map_tree f t) = size_tree (mirror_tree  t).
Proof.
  intros.
  rewrite size_map_tree, size_mirror_tree.
  trivial.
Qed.
```

Requires no modification

3

User-defined tactics are awesome (for automation & robustness), but their use is hindered by several *limitations.*



Tactics support no query mechanism

```
$ grep -r Ltac * | wc -l
471
```

→ **There probably are redundant definitions.**

Tactics support no query mechanism

```
$ grep -r Ltac * | wc -l
471
```

→ **There probably are redundant definitions.**

```
Print TLC.LibTactics.
```

→ **All Gallina definitions, no Ltac definitions.**

A tactic “specification” language similar to SearchAbout?

Debugging

A debugger exists, but it is very basic.

9.4.2 Interactive debugger

The L_{tac} interpreter comes with a step-by-step debugger. The debugger can be activated using the command

```
Set Ltac Debug.
```

simple newline:	go to the next step
h:	get help
x:	exit current evaluation
s:	continue current evaluation without stopping
r *n*:	advance *n* steps further
r *string*:	advance up to the next call to "`idtac` *string*"

When debugging, we typically look for a failing branch. The tracing tool of Coq exactly ignores these.

9.4.1 Info trace

It is possible to print the trace of the path eventually taken by an L_{tac} script. That is, the list of executed tactics, discarding all the branches which have failed. To that end the `Info` command can be used with the following syntax.

Two kinds of tactics

Tactics building terms

```
Ltac ltac_inter l1 l2 :=
  match l2 with
  | nil =>
    constr:(@nil
      ltac:(match type of l1 with
                 list ?T => T end))
  | ?a :: ?l =>
    let is_in := ltac_mem a l1 in
    let r := ltac_inter l1 l in
    match is_in with
    | true => constr:(a :: l)
    | false => r
    end
  end.
```

Tactics with side effects

`rewrite`, `idtac`, everything using “;”, etc.

They can not be mixed

`idtac; constr:(1)` will always fail.

Type for tactics?

t ::= <effect> | <constr> | t -> t | 'a

Type for tactics?

t ::= <effect> | <constr> | t -> t | 'a

This would have detected my last week's bug:

```
Ltac get_something e k :=
  let aux k' :=
    let H := fresh "H" in
     assert (H : something e); [ prove_something | k' H ]
   in
   match goal with
   | L : lemma_for_something |- _ =>
     aux (fun H =>
       apply (change_something L) to H;
       k H)
  end.
```

Type for tactics?

t ::= <effect> | <constr> | t -> t | 'a

This would have detected my last week's bug:

```
Ltac get_something e k :=
  let aux k' :=
    let H := fresh "H" in
     assert (H : something e); [ prove_something | k' H (fun r => k r; try clear H) ]
   in
   match goal with
   | L : lemma_for_something |- _ =>
     aux (fun H =>
       apply (change_something L) to H;
       k H)
  end.
```

Type for tactics?

t ::= <effect> | <constr> | t -> t | 'a

This would have detected my last week's bug:

```
Ltac get_something e k :=
  let aux k' :=
    let H := fresh "H" in
     assert (H : something e); [ prove_something | k' H (fun r => k r; try clear H) ]
   in
   match goal with
   | L : lemma_for_something |- _ =>
     aux (fun H =>
       apply (change_something L) to H;
       k H)
  end.
```

→ **Error: No matching clauses for match.**

Miscellaneous

- Fresh and its hints.

  “fresh "IH" e” fails when “e” is not an identifier.

- Lists of hypotheses.

  crush’s done, TLC’s boxer, SSReflect stack, etc.

- Getting constructors and projections as a list.

  ```
  let x := constr:(ltac:(constructor) : T) in ltac:(induction x; exact I) : True
  ```

- A timing and memory model for Ltac?

  My Coq development last month: Fatal error: out of memory.

Conclusion

We can develop in Ltac, but we are lacking some tools

- Any proof analysis tool would be greatly welcomed;
- Any way of looking through already defined tactics;
- Ltac definitely needs more types.

Conclusion

We can develop in Ltac, but we are lacking some tools

- Any proof analysis tool would be greatly welcomed;
- Any way of looking through already defined tactics;
- Ltac definitely needs more types.

Thanks!