

Programming and Proving

Practice with FoCaLiZe

François Pessaux
ENSTA ParisTech

EJCP
May 19, 2014

<http://perso.ensta-paristech.fr/~pessaux/ejcp-2014/>

Why Proving ?



In Industry...

- Hardware replaced by software.
- Military, medical, transport, energy, finances, telecoms...
- Effect of software failure ? → †, €/, 😡, ...
- Standards (IEC-61508, EN-50128, CC2, DO-178C...) rule systems developments.
- Formal methods required for **highest** safety/security levels.

In Science...

- As a **computer scientist**: want to be **sure** of my algorithm.
- As another scientist: want to be sure that the software tools **I use** do not alter **my** results.

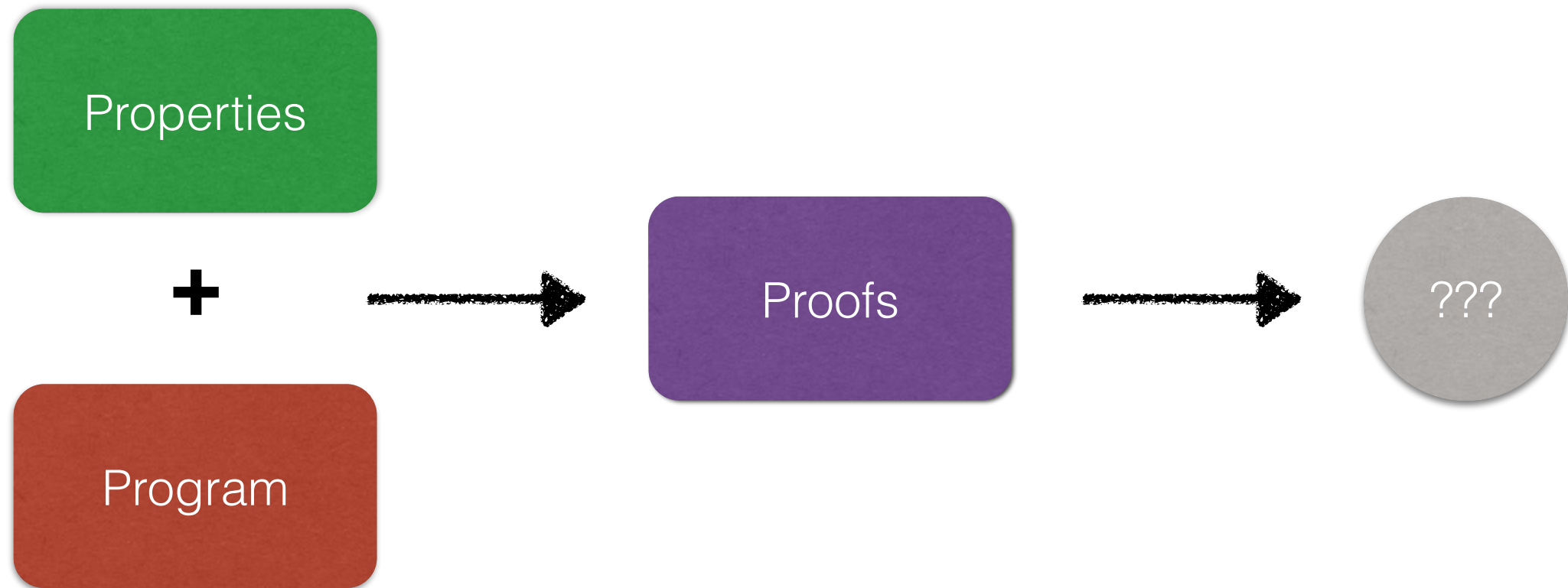
Proofs of What?



Proof \Rightarrow Property

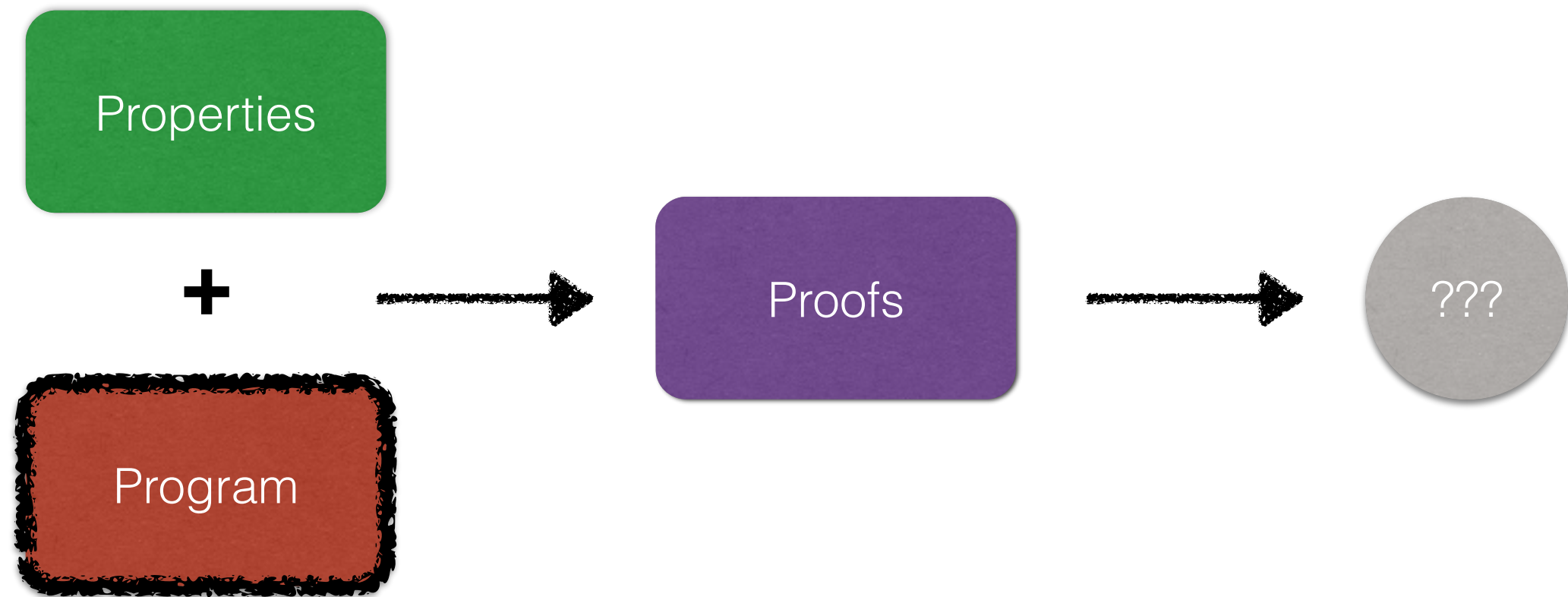
- Never trust « *I proved my program works* ».
- A proof **is not** an absolute essence of a program.
- Need statements (i.e. stated **properties**) representing that « *my program works* »
- Then **prove** that indeed the **program implementation** satisfies these **properties**.
- ... according to
- Properties are the **specification(s)** of the program.

Proofs in their Environment



- 4 shapes, 4 questions...

Question 1: a Program?



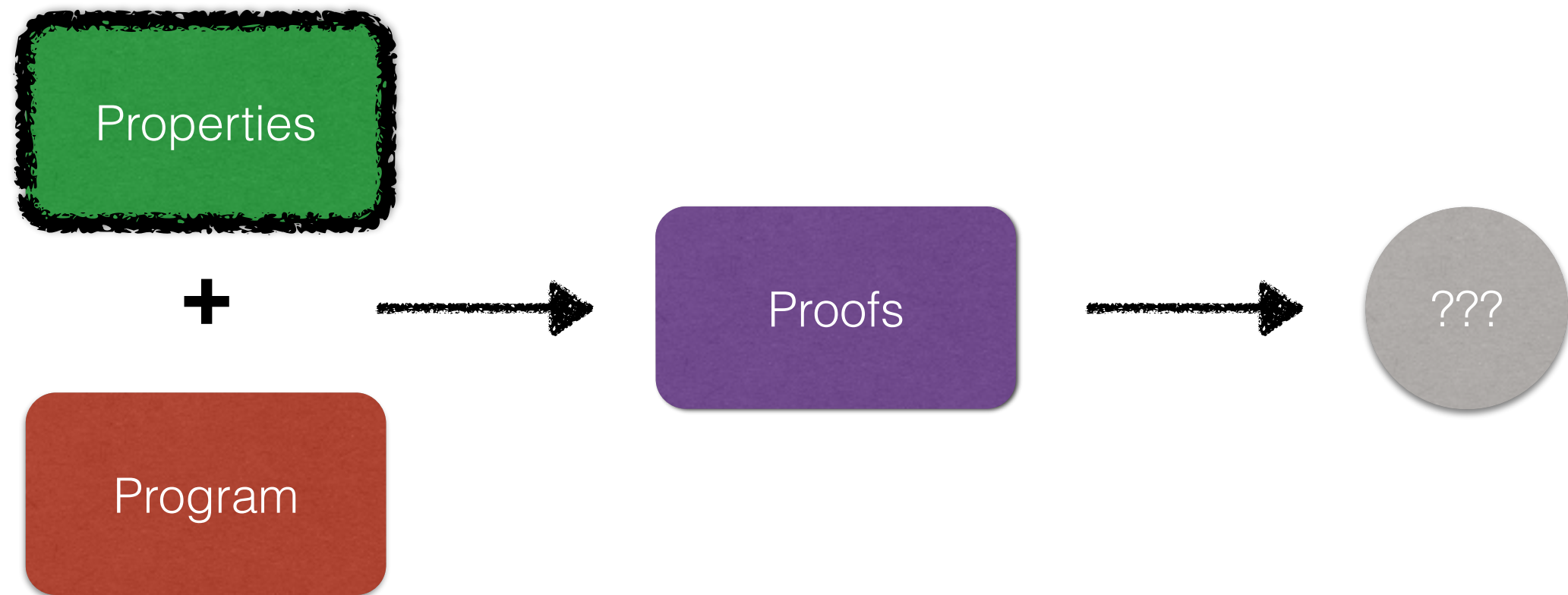
- In which **language**? Pseudo-code, C, Java, DSL...
- In which **semantical** framework? Imperative, functional...

Answer 1: a Programming Language...

- A **pure functional** language *à la* ML: **FoCaLiZe**
 - ➔ Functions, sum types, pattern-matching, ...
- + modularity, inheritance, late-binding, abstraction, parameterization.

```
species AssocMap (Key is Comparable, Value is Comparable, OptValue is OptComparable (Value)) =  
inherit Setoid ;  
representation = pair_list_t (Key, Value) ;  
  
let empty : Self = Nil ;  
  
let add (k, v, m : Self) : Self = Node (k, v, m) ;  
  
let rec find (k, m : Self) : OptValue =  
  match m with  
  | Nil -> OptValue!none  
  | Node (kcur, v, q) ->  
    if Key!eq (kcur, k) then OptValue!some (v) else find (k, q)  
termination proof = structural m ;  
...  
end ;;
```

Question 2: Properties?



- Which **logical** language?
- Depends on the **specifications** we want to be able to express.
- Too rich: impossible to (even partly) **automate** proofs.
- Too poor: impossible to express **specifications** through the **programming language**.

Answer 2: a Logical Language

- **First-order** logic
- + prog. language **constructs** (function, type, pattern-matching...)
- + **equality**.

```
theorem implications : all a b : bool, a -> (b -> a)
```

```
proof = ... ;;
```

```
species Comparable =
```

```
...
```

```
property eq_symmetric: all x y : Self, eq (x, y) -> eq (y, x) ;
```

```
...
```

```
end ;;
```

```
species AssocMap (Key is Comparable, Value is Comparable, OptValue is OptComparable (Value)) =
```

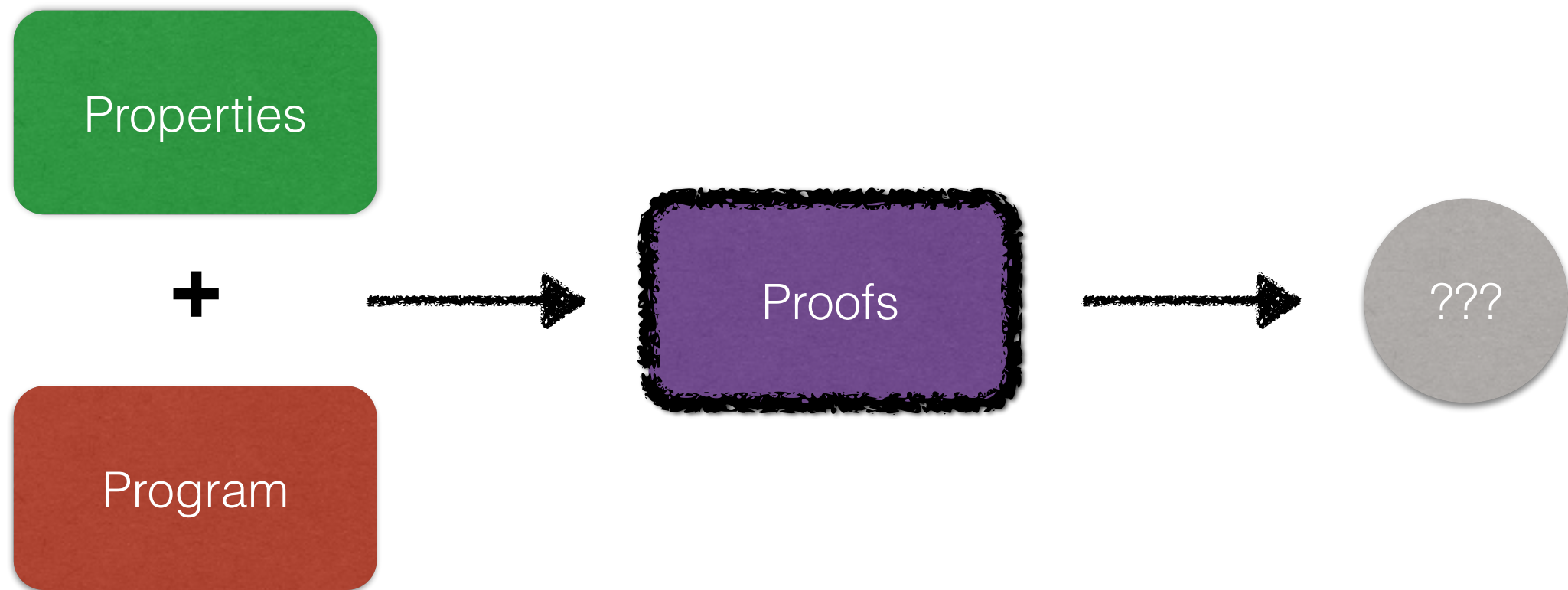
```
...
```

```
property find_added_not_fails: all k : Key, all v : Value, all m1 m2 : Self,  
  m2 = add (k, v, m1) -> ~ OptValue!eq (find (k, m2), OptValue!none) ;
```

```
...
```

```
end ;;
```

Question 3: Proof?



- Which **proof** language? Coq, HOL, PVS...
- Independence for the user? Easing user proof task ?
 - No need for deep knowledge of a logical framework.
 - No need to know how the user proof language is compiled.
- User-readable proofs wanted.

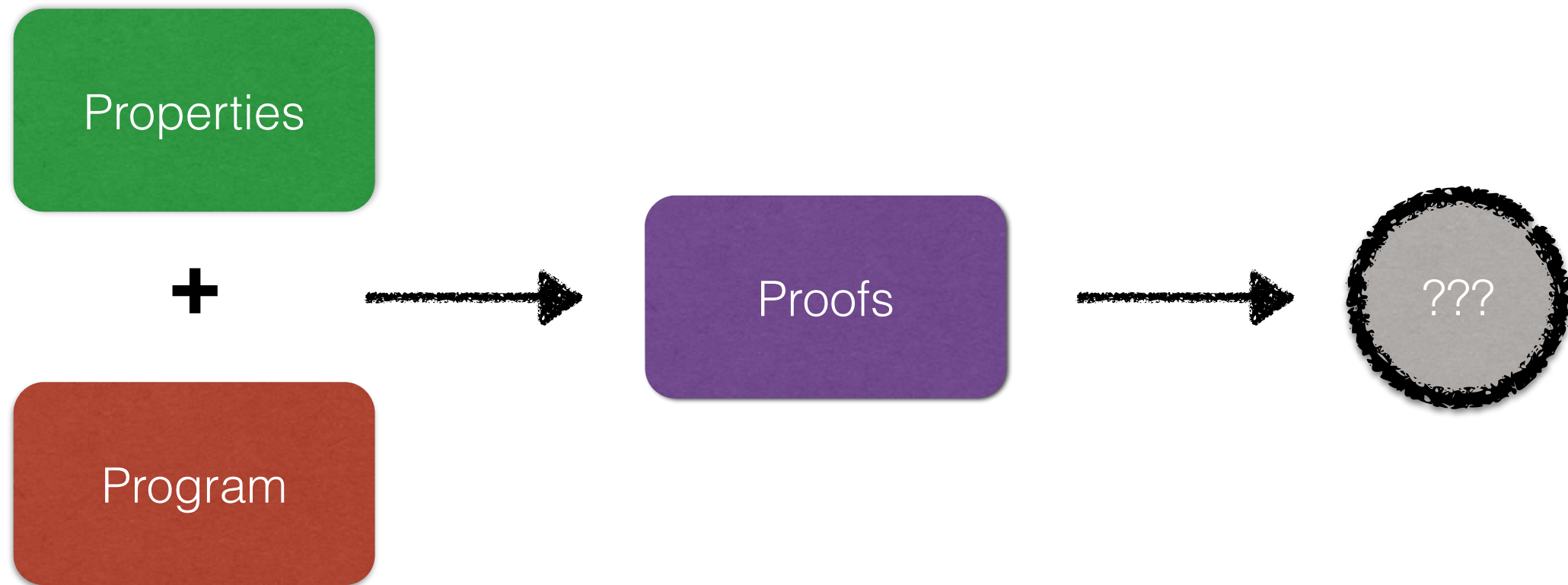
Answer 3: a Proof Language

- Dedicated proof language provided (FCL).
- **Fully part** of the FoCaLiZe language (consistency).
- Allows **hierarchical** proofs *à la* **natural deduction** (readability).
- Relies on the **Zenon** theorem prover (discharges the user).
- Compilation to a target logical language by FoCaLiZe compiler.

proof =

```
<1>1 assume m : Self,  
  assume s k : Key,  
  assume v : Value,  
  prove (find (s, m) = OptValue!some (v)  $\vee$  Key!eq (s, k))  $\leftrightarrow$  find (s, add (k, v, m)) = OptValue!some (v)  
  <2>1 hypothesis H1: find (s, m) = OptValue!some (v)  $\vee$  Key!eq (s, k),  
    prove find (s, add (k, v, m)) = OptValue!some (v)  
    <3>1 prove add (k, v, m) = Node (k, v, m) by definition of add type pair_list_t  
    <3>e qed by step <3>1 definition of find hypothesis H1 type pair_list_t property Key!eq_symmetric  
  <2>2 hypothesis H2: find (s, add (k, v, m)) = OptValue!some (v),  
    prove find (s, m) = OptValue!some (v)  $\vee$  Key!eq (s, k)  
    <3>1 prove add (k, v, m) = Node (k, v, m) by definition of add type pair_list_t  
    <3>e qed by step <3>1 definition of find hypothesis H2 type pair_list_t property Key!eq_symmetric  
  <2>e qed by step <2>1, <2>2  
<1>e conclude ;
```

Question 4: ???



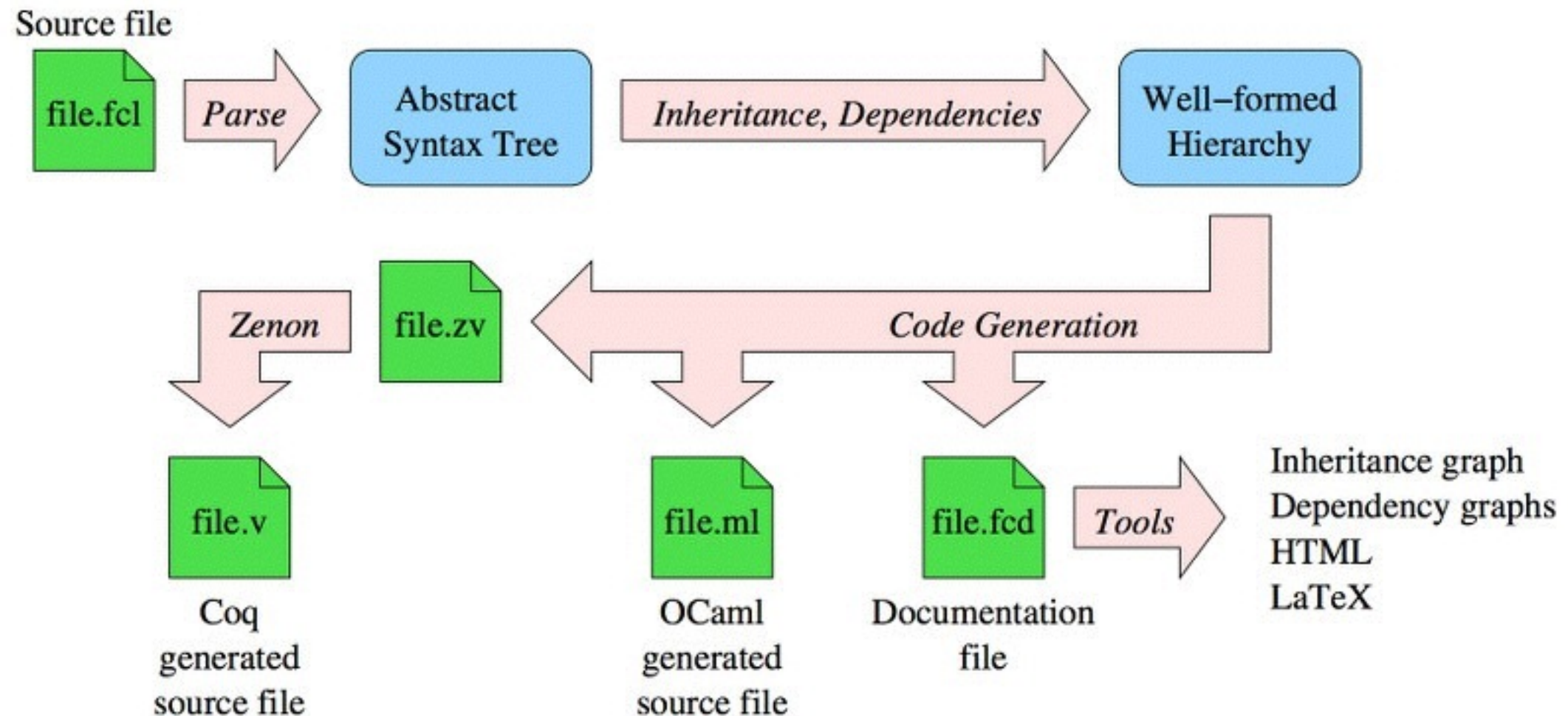
- What to do with the proof?
- Verified by a human ? **Error prone!**
- We want a **formal** proof.

Answer 4: a Checker

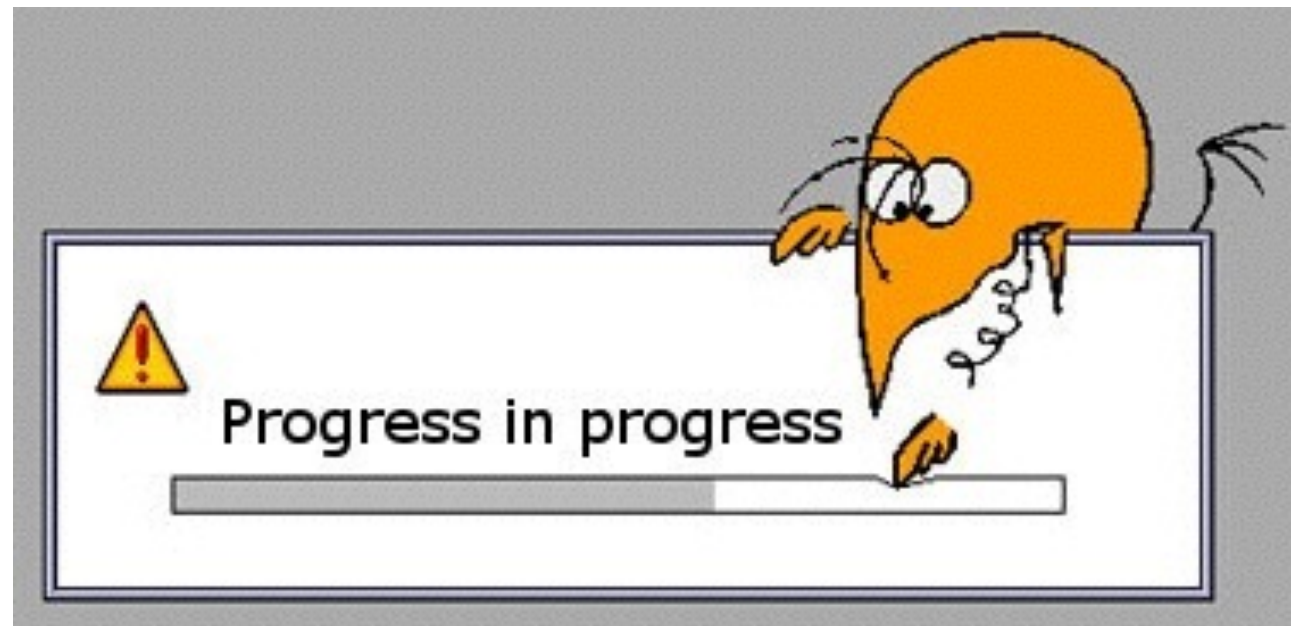
- Proofs must be checked **mechanically** !
- ➔ The proof assistant **Coq** acts as an **assessor**.
- **Zenon** issues **Coq terms** for **proofs**.
- **Whole model** of the program **compiled** into a **Coq term**.
types, functions, properties and proofs.

```
...
(* From species assoc_map3#OptComparable. *)
(* Section for proof of theorem 'eq_reflexive'. *)
Section Proof_of_eq_reflexive.
  Variable _p_C_T : Set.
  Variable _p_C_eq : _p_C_T -> _p_C_T -> basics.bool__t.
  Variable _p_C_eq_reflexive : forall x : _p_C_T, ls_true ((_p_C_eq x x)).
  Let abst_T := (option_t__t _p_C_T).
  Let abst_eq := eq _p_C_T _p_C_eq.
(* File "assoc_map3.fcl", line 42, characters 4-61 *)
Theorem for_zenon_eq_reflexive : (forall x : abst_T, (ls_true (abst_eq x x))).
Proof.
apply NNPP. intro zenon_G.
apply (zenon_notallex_s (fun x : abst_T => (ls_true (abst_eq x x))) zenon_G); [ zenon_intro zenon_H2; idtac ].
elim zenon_H2. zenon_intro zenon_Tx_d. zenon_intro zenon_H4.
...
```

FoCaLiZe Compilation Flow



Plan of the Lecture



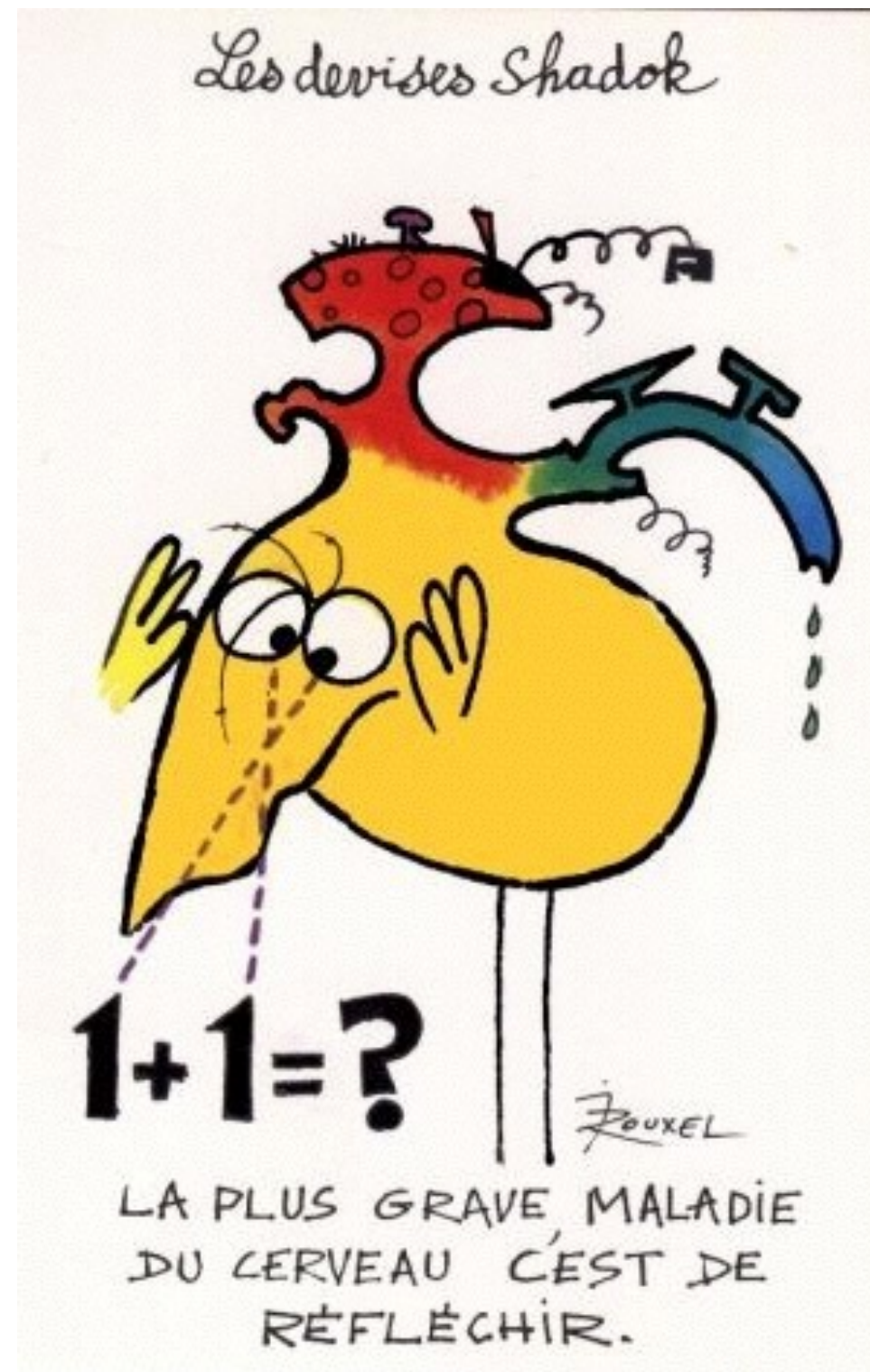
- First-Order Logic in FoCaLiZe.
- Basic Programming in FoCaLiZe.
- Adding Structure in Programs.
- Adding Proofs in Programs.
- Adding Parameterization.
- When to Prove What?
- Advanced Proofs.

First-Order Connectors

Semantics	Connector	FoCaLiZe Syntax
Conjunction	\wedge	\wedge
Disjunction	\vee	\vee
Implication	\Rightarrow	\rightarrow
Equivalence	\Leftrightarrow	\leftrightarrow
Negation	\neg	\sim
Universal quantification	\forall	all
Existential quantification	\exists	ex

+ programming expressions.

Starting with Proofs...



A First Tautologie

$$\forall a, b : \text{boolean}, a \Rightarrow b \Rightarrow a$$

- In a file ***ex_implications.fcl***:

```
open "basics" ;;  
theorem implications : all a b : bool, a -> (b -> a)  
proof = assumed ;;
```

- Let's compile:

```
$ focalizec ex_implications.fcl
```

```
Invoking ocamlc...
```

```
>> ocamlc -I /usr/local/lib/focalize -c ex_implications.ml
```

```
Invoking zvtov...
```

```
>> zvtov -zenon zenon -new ex_implications.zv
```

```
Invoking coqc...
```

```
>> coqc -I /usr/local/lib/focalize -I /usr/local/lib/zenon ex_implications.v
```

```
$
```


What do we got ?

- Two source files:
 - **ex_implication.ml**: « **executable** » code.
 - **ex_implication.v**: **logical** term.
- Both sent to their respective compiler (**ocamlc**, **coqc**).
- Time to **write** a proof!
- **Sequent** of the proof:

Hypotheses

$$\frac{\frac{\frac{\forall a, b : \text{boolean}, a, b \vdash a}{\forall a, b : \text{boolean}, a \vdash b \Rightarrow a}}{\forall a, b : \text{boolean} \vdash a \Rightarrow b \Rightarrow a}}{\vdash \forall a, b : \text{boolean}, a \Rightarrow b \Rightarrow a}$$

Goals

A Proof in FoCaLiZe

Bullet

Goal

Fact

Hypothesis

Compound proof

theorem implications : **all** a b : **bool**, a -> (b -> a)

proof =

<1>1 **assume** a : bool, b : bool,

hypothesis h1 : a,

prove b -> a

<2>1 **hypothesis** h2 : b,

prove a

by hypothesis h1

<2>2 **qed by step** <2>1

<1>2 **conclude**

(* or: **qed conclude**

or: **qed by step** <1>1 *) ;;

Steps

$$\frac{\forall a, b : \text{boolean}, a, b \vdash a}{\forall a, b : \text{boolean}, a \vdash b \Rightarrow a}$$

$$\frac{\forall a, b : \text{boolean}, a \vdash b \Rightarrow a}{\forall a, b : \text{boolean} \vdash a \Rightarrow b \Rightarrow a}$$

$$\frac{\forall a, b : \text{boolean} \vdash a \Rightarrow b \Rightarrow a}{\vdash \forall a, b : \text{boolean}, a \Rightarrow b \Rightarrow a}$$

Mimic the Sequent Proof

theorem implications : **all** a b : **bool**, a -> (b -> a)

proof =

<1>1 **assume** a : bool, b : bool,

hypothesis h1 : a,

prove b -> a

<2>1 **hypothesis** h2 : b,

prove a

by hypothesis h1

<2>2 **qed by step** <2>1

<1>2 **conclude**

(* or: **qed conclude**

or: **qed by step** <1>1 *) ;;

$$\forall a, b : \text{boolean}, a, b \vdash a$$

$$\forall a, b : \text{boolean}, a \vdash b \Rightarrow a$$

$$\forall a, b : \text{boolean} \vdash a \Rightarrow b \Rightarrow a$$

$$\vdash \forall a, b : \text{boolean}, a \Rightarrow b \Rightarrow a$$

The less I work, the better I feel

- **Zenon** knows (among other things) about basic logic rules.
- Let's it work instead of us.

```
open "basics" ;;
```

```
theorem implications : all a b : bool, a -> (b -> a)  
proof = conclude ;;
```

- Fact **conclude**: « *Zenon, handle it by yourself!* »
- **Zenon** is there to **help automating** proofs.

Basic Programming in FoCaLiZe



Core Programming Language

- FoCaLiZe: **pure functional** programming language:
 - Basic types: int, bool, string, ... + polymorphism.
 - Construct types: product, sum (recursive), records.
 - Let-definition, pattern-matching, if-then-else, ...
- Properties will deal with these constructs...
- ... so will proofs.

First Program: Loose Sets

- Implementation of (poor) sets as lists.
- In a file **loose_sets.fcl**

```
open "basics" ;;           ← Make a module of the standard library visible

type set_t ('a) =
  | Empty
  | Elem ('a, set_t ('a)) ;; ← Polymorphic sum type definition

let is_empty (s) = s = Empty ;; ← Non-recursive function definition

let add (x, s) = Elem (x, s) ;;

let rec belongs (x, s) =
  match s with
  | Empty -> false
  | Elem (e, s) -> if e = x then true else belongs (x, s)
  termination proof = structural s ;; ← Recursive function definition
                                     ← Termination proof
```

Add a Second File ... with Properties

- « Specialize », implementing integers sets.
- Add a function « refusing to add 0 ».
- Add some properties ... (and next, proofs).
- In a file **int_loose_sets.fcl**

```
open "basics" ;;
```

```
use "loose_sets" ;;
```



Allows to access module loose_sets

```
let add_except_0 (x : int, s) =  
  if x = 0 then s else loose_sets#add (x, s) ;;
```



Access function add of module loose_set

```
theorem zero_not_added: all x : int, all s : loose_sets#set_t (int),  
  (loose_sets#is_empty (s)  $\wedge$  x = 0) -> loose_sets#is_empty (add_except_0 (x, s))
```

```
proof = ??? ;;
```



States that add_except_0 never add 0 to an empty set

```
theorem zero_not_added_weaker: all s : loose_sets#set_t (int),  
  loose_sets#is_empty (s) -> loose_sets#is_empty (add_except_0 (0, s))
```

```
proof = ??? ;;
```



States nearly the same thing...

... And the Proofs?

```
let is_empty (s) = s = Empty ;;  
let add (x, s) = Elem (x, s) ;;  
  
let add_except_0 (x : int, s) =  
  if x = 0 then s else loose_sets#add (x, s) ;;
```

```
theorem zero_not_added: all x : int, all s : loose_sets#set_t (int),  
  (loose_sets#is_empty (s)  $\wedge$  x = 0)  $\rightarrow$  loose_sets#is_empty (add_except_0 (x, s))  
proof =
```

- The proof comes from the **definition** of add_except!
- Not even needed to know what is_empty does.
- Use the Zenon fact **definition of**.

```
proof = by definition of add_except_0 ;;
```

... And the other Proof?

```
theorem zero_not_added: all x : int, all s : loose_sets#set_t (int),  
  (loose_sets#is_empty (s)  $\wedge$  x = 0) -> loose_sets#is_empty (add_except_0 (x, s))  
proof = by definition of add_except_0 ;;
```

```
theorem zero_not_added_weaker: all s : loose_sets#set_t (int),  
  loose_sets#is_empty (s) -> loose_sets#is_empty (add_except_0 (0, s))  
proof =
```

- The proof comes from the previous **lemma** zero_not_added!
- x was directly instantiated by 0.
- Use the Zenon fact **property**.

```
proof = by property zero_not_added ;;
```

The Whole file

```
open "basics" ;;
use "loose_sets" ;;

let add_except_0 (x : int, s) =
  if x = 0 then s else loose_sets#add (x, s) ;;

theorem zero_not_added: all x : int, all s : loose_sets#set_t (int),
  (loose_sets#is_empty (s)  $\wedge$  x = 0)  $\rightarrow$  loose_sets#is_empty (add_except_0 (x, s))
proof = by definition of add_except_0 ;;

theorem zero_not_added_weaker: all s : loose_sets#set_t (int),
  loose_sets#is_empty (s)  $\rightarrow$  loose_sets#is_empty (add_except_0 (0, s))
proof = by property zero_not_added ;;
```

- We saw 2 Zenon **facts: definition of** and **property**.

- Let's compile.

```
$ focalizec int_loose_sets.fcl
Invoking ocamlc...
>> ocamlc -I /usr/local/lib/focalize -c int_loose_sets.ml
Invoking zvtov...
>> zvtov -zenon zenon -new int_loose_sets.zv
Invoking coqc...
>> coqc -I /usr/local/lib/focalize -I /usr/local/lib/zenon int_loose_sets.v
$
```

A Simple Proof by Cases

- A last theorem: « *if we add an element to a set, then it belongs to the resulting set* ».

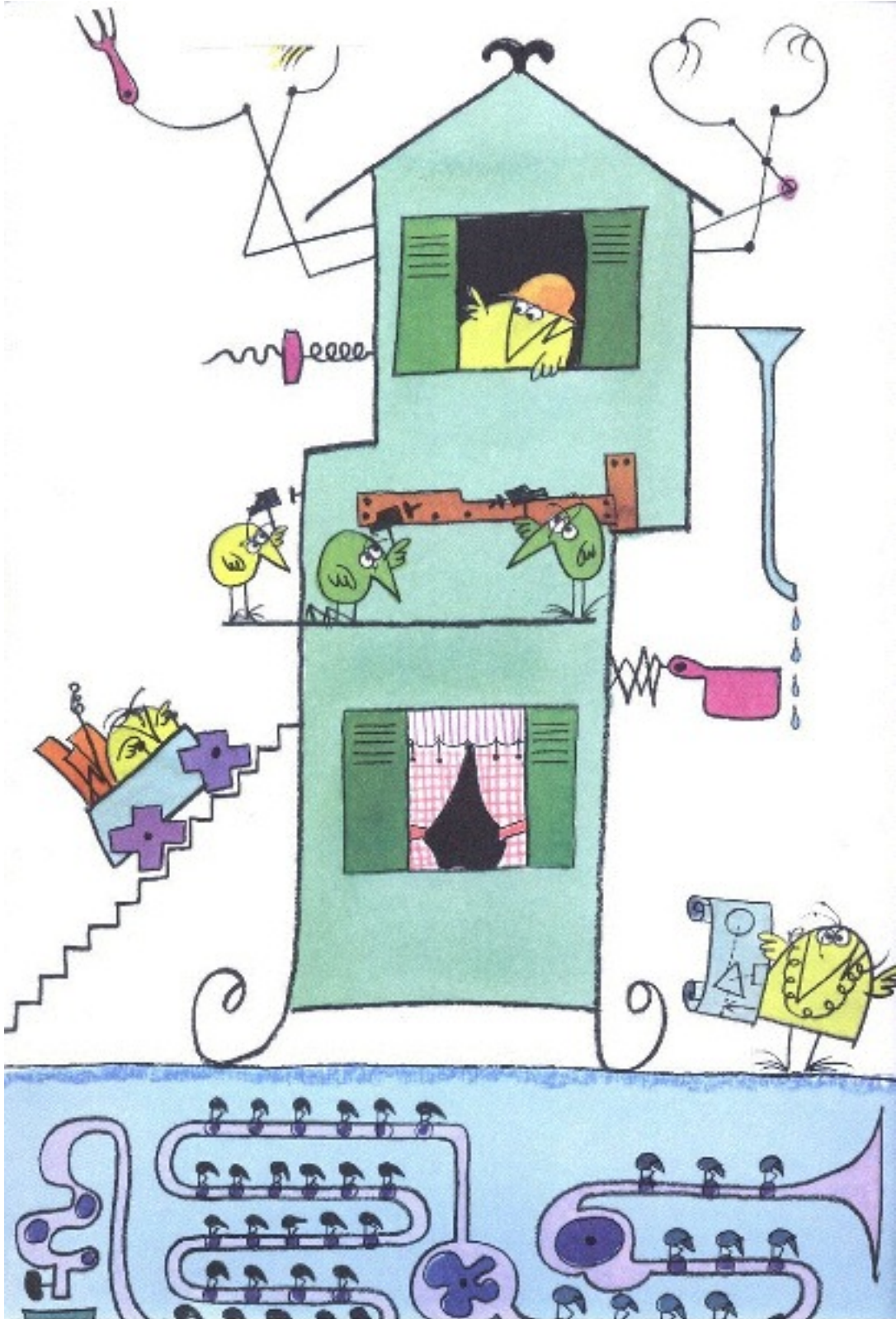
```
open "basics" ;;  
open "loose_sets" ;;
```

```
theorem added_forcibly_belongs: all x : int, all s : set_t (int), belongs (x, add (x, s))  
proof =
```

- Proof to do
 - by using the definition of `belong` and `add`,
 - by **case** on values of type `set_t`.
- Zenon needs to know about the **type**, about its **induction principle**.
- Use the Zenon fact **type**.

```
proof = by definition of add, belongs type set_t ;;
```

Structuring Programs



In Realistic Developments

- Express **specifications**,
- and go **step by step** to **design** and **implementation**,
- while **proving** that such an implementation meets its specification or design **requirements**.
- ➔ Need for an **incremental** approach.
- Reduce coupling: enhances reusability and robustness.
- Common techniques: **modularity** and **encapsulation**.
- ➔ Need for some kind of **Abstract Data-Types**.

Grouping Structure: Species

- Aim: **grouping** « *things* » related to a **same concept**:
 - an underlying **data-type**,
 - its manipulation **functions**,
 - its their **properties/theorems**.
- « *Things* » called **methods**.

```
species Name =  
  meth1 ;  
  meth2 ;  
  ...  
end ;;
```

- Species name: always: **capitalized**.

Methods of Species

- **representation**: the « data representation » of the entities manipulated by the species
 - ➔ type definition.
- **signature**: announces a function to be defined later
 - ➔ name + type.
- **let (rec/logical)**: introduces a **definition**
 - ➔ name + optional type + expression.
- **property**: **logical formula** not yet proved.
 - ➔ name + first-order statement.
- **theorem**: **proved** logical formula.
 - ➔ name + first-order statement + proof.
- **proof of**: proof to attach to an existing **property**.
 - ➔ name + proof.

Example Please...

representation is

not defined

defined

```
species BasicStuff =  
  signature eq : Self -> Self -> bool ;  
  let different (x, y) = ~~ eq (x, y) ;  
  property eq_reflexive: all x : Self, eq (x, x) ;  
end ;;
```

```
species OrderedPairs =  
  representation = int * int ;  
  let make (x, y) : Self = if x >0 x y then (y, x) else (x, y) ;  
  let first (x : Self) =  
    match (x) with  
    | (x1, x2) -> x1 ;  
  let second (x : Self) =  
    match (x) with  
    | (x1, x2) -> x2 ;  
  theorem make_safe : all x : Self, all i1 : int, all i2 : int,  
    x = make (i1, i2) -> second (x) >0 x first (x)  
  proof = ..... ;  
end ;;
```

Not yet proved

Definition using only declared method

« Greater » on ints

« Type » of the representation

More Advanced Features Later...



Case of Study: Association Maps

- Data-structure allowing to bind a « **key** » to a « **value** ».
- Want to **retrieve** the value bound to a certain key.
- 3 operations:
 - ***empty*** : *map*
 - Initial map containing **no** binding.
 - ***add*** (k, v, m) : *key* \rightarrow *value* \rightarrow *map* \rightarrow *map*
 - **Adds** the **binding** ($k \mapsto v$) to the map m .
 - ***find*** (k, m) : *key* \rightarrow *map* \rightarrow « *value or error* »
 - Looks for and returns the **value** bound to the **key** if some exists. Otherwise, signal « *an error* ».

Naive Implementation: Used Types

- Hardwired:
 - Type of keys: **int**.
 - Type of values: **string**.
- Recording structure: something like a **list**.
- « Value or error »: encoded in an **option** type.
- Type definitions always at top-level in FoCaLiZe.

```
(* Structure recording bindings of a map: a hand-made basic list. *)  
type int_str_list_t =  
  | Nil  
  | Node (int, string , int_str_list_t) ;;  
  
(* Return value of the lookup function: nothing or something. *)  
type option_t ('a) =  
  | None  
  | Some ('a) ;;
```

Maps with their Functions

```
species AssocMap =  
  representation = int_str_list_t ;  
  let empty : Self = Nil ;   (* Empty association map: no bindings. *)  
  
  (* Addition to the map m of the value v bound to the key k. *)  
  let add (k: int, v: string, m : Self) : Self = Node (k, v, m) ;  
  
  (* Lookup the the value bound to the key k in the map m. *)  
  let rec find (k: int, m: Self) =  
    match m with  
    | Nil -> None  
    | Node (kcur, v, q) -> if kcur = k then Some (v) else find (k, q)  
  termination proof = structural m ;  
end ;;
```

From type `int_str_list_t`

From type `option_t`

Termination proof required
(here, structural)

Compiling...

- We can compile...

```
$ focalizec assoc_map1.fcl
Invoking ocamlc...
>> ocamlc -I /usr/local/lib/focalize -c assoc_map1.ml
Invoking zvtov...
>> zvtov -zenon zenon -new assoc_map1.zv
Invoking coqc...
>> coqc -I /usr/local/lib/focalize -I /usr/local/lib/zenon assoc_map1.v
$
```

- Right, but not yet executable...
- (And no proofs yet...)

Final Encapsulation: Collection

- Until now: just grouped methods of assoc maps.
- To test we need to turn the species into the expected **Abstract Data-Type**.
- ➔ Need to build a **collection**: kind of « *instance* » of a species.
- Collection: **definitions** get **opaque**.
 - Only **types** of **methods** visible.
 - Only **statements** of **theorems** visible.

```
collection MyMap = implement AssocMap ; end ;;
```

```
(* Printer of value of type option (string). *)
```

```
let print_string_option (v) = match v with
```

```
  | None -> print_string ("Not found\n")
```

```
  | Some (s) -> let _a = print_string ("Found value: ") in let _b = print_string (s) in  
    print_newline (()) ;;
```

```
let m = MyMap!add (5, "five", MyMap!empty) ;;
```

```
print_string_option (MyMap!find (5, m)) ;;
```

```
print_string_option (MyMap!find (3, m)) ;;
```


Compiling / Running

- Invoke **focalizec**:
 - Create the OCaml **object file**: need to **link** to get an executable.
 - Create the Coq source file: **directly checked** by **Coq**.

```
$ focalizec assoc_map1_partial_test.fcl
Invoking ocamlc...
>> ocamlc -I /usr/local/lib/focalize -c assoc_map1_partial_test.ml
Invoking zvtov...
>> zvtov -zenon zenon -new assoc_map1_partial_test.zv
Invoking coqc...
>> coqc -I /usr/local/lib/focalize -I /usr/local/lib/zenon assoc_map1_partial_test.v
print_string_option (MyMap.find 5 m)
  : basics.unit__t
print_string_option (MyMap.find 3 m)
  : basics.unit__t
```

- Link using **ocamlc** and a few **object files** of FoCaLiZe std. lib.

```
$ ocamlc -I /usr/local/lib/focalize ml_builtins.cmo basics.cmo assoc_map1_partial_test.cmo
```

- Run...

```
$ ./a.out
Found value: five
Not found
$
```

Encapsulation Won

- In a **collection**, the **representation** of the **implemented** species is **hidden**.
- « *Manually* » exploit its **internal type's** structure is **no more possible**.

```
...  
let m = MyMap!add (5, "five", Nil) ;;
```

```
$ focalizec assoc_map1_partial_test.fcl  
File "assoc_map1_partial_test.fcl", line 42, characters 8-34:  
Error: Types assoc_map1_partial_test#int_str_list_t and  
assoc_map1_partial_test#MyMap are not compatible.  
$
```

- Prevents from « **savage** » manipulations.
- Ensure that provided (and proved) properties **always hold**.

Time to Prove!

- « *Finding the value bound to a key just inserted in a map never fails* ».
- « *Calling find with a key k on a map built by add-ing to it k with any bound value never returns None.* »

(* Add make find a success. *)

theorem find_added_not_fails: **all** k : int, **all** v : string, **all** m1 m2 : **Self**,
m2 = add (k, v, m1) -> ~ (find (k, m2) = None)

proof = ????

Time to Prove!

- « *Finding the value bound to a key just inserted in a map never fails* ».
- « *Calling find with a key k on a map built by add-ing to it k with any bound value never returns None.* »

(* Add make find a success. *)

theorem find_added_not_fails: **all** k : int, **all** v : string, **all** m1 m2 : **Self**,
m2 = add (k, v, m1) -> ~ (find (k, m2) = None)

proof = by definition of add, find ;

- Consequence of how add and find are **implemented**.
- ➔ Need facts by **definition of fact, add**.

Time to Prove!

- « *Finding the value bound to a key just inserted in a map never fails* ».
- « *Calling find with a key k on a map built by add-ing to it k with any bound value never returns None.* »

(* Add make find a success. *)

theorem find_added_not_fails: **all** k : int, **all** v : string, **all** m1 m2 : **Self**,
m2 = add (k, v, m1) -> ~ (find (k, m2) = None)

proof = by definition of add, find *type int_str_list_t, option_t* ;

- Consequence of how add and find are **implemented**.
→ Need facts *by definition of fact, add*.
- Constructors of types int_str_list_t and option_t must be known.
→ Need facts *type int_str_list_t, option_t*.

Harder Proof

- Need for a **specification** of function `find`.

```
theorem find_spec: all m : Self, all s k : int, all v : string,  
  (find (s, m) = Some (v) V s = k) <-> find (s, add (k, v, m)) = Some (v)  
  ...
```

- Let's hope we are lucky...

```
proof = by definition of add, find type int_str_list_t, option_t ;
```

- Let's compile...

```
$ focalizec -zvtovopt -script assoc_map1.fcl  
Invoking ocamlc...  
>> ocamlc -I /usr/local/lib/focalize -c assoc_map1.ml  
Invoking zvtov...  
>> zvtov -zenon zenon -new -script assoc_map1.zv  
42 *****#####-
```

- Bad luck! Automation failed \Rightarrow **We** have to work!

Splitting the Proof (1)

- **We** have to **split** the proof in **intermediate steps**.
 1. Introduce **hypotheses** in the context.
 2. State the new goal.
 3. Leave it unproved.
 4. Add a **qed** step to conclude.

```
theorem find_spec: all m : Self, all s k : int, all v : string,  
  (find (s, m) = Some (v)  $\forall$  s = k)  $\leftrightarrow$  find (s, add (k, v, m)) = Some (v)  
proof =  
  <1>1 assume m : Self,  
    assume s k : int,  
    assume v : string,  
    prove (find (s, m) = Some (v)  $\forall$  s = k)  $\leftrightarrow$  find (s, add (k, v, m)) = Some (v)  
    assumed  
  <1>e conclude ;
```

- Compile... \Rightarrow Accepted! Go on...

Splitting the Proof (2)

- Prove the **assumed** <1>1.
- **Implication** to prove \Rightarrow Prove **equivalence** in **both ways**.
- Split the proof:
 1. Add a step whose goal is **left to right**, leave it **assumed**.
 2. Same from **right to left**.
 3. Add a **qed** step to conclude.

```
theorem find_spec: all m : Self, all s k : int, all v : string,  
  (find (s, m) = Some (v)  $\forall$  s = k)  $\Leftrightarrow$  find (s, add (k, v, m)) = Some (v)  
proof =  
  <1>1 assume m : Self,  
    assume s k : int,  
    assume v : string,  
    prove (find (s, m) = Some (v)  $\forall$  s = k)  $\Leftrightarrow$  find (s, add (k, v, m)) = Some (v)  
    <2>1 hypothesis H1: find (s, m) = Some (v)  $\forall$  s = k,  
      prove find (s, add (k, v, m)) = Some (v)  
      assumed  
    <2>2 hypothesis H2: find (s, add (k, v, m)) = Some (v),  
      prove find (s, m) = Some (v)  $\forall$  s = k  
      assumed  
    <2>e qed by step <2>1, <2>2  
  <1>e conclude ;
```

Splitting the Proof (3 ... 🤔)

- Prove the previously **assumed** <2>1.
- Hope that Zenon can solve from **find**, **add**, **H1** and **int_str_list_t** ?

```
theorem find_spec: all m : Self, all s k : int, all v : string,  
  (find (s, m) = Some (v) V s = k) <-> find (s, add (k, v, m)) = Some (v)  
proof =  
  <1>1 assume m : Self,  
    assume s k : int,  
    assume v : string,  
    prove (find (s, m) = Some (v) V s = k) <-> find (s, add (k, v, m)) = Some (v)  
    <2>1 hypothesis H1: find (s, m) = Some (v) V s = k,  
      prove find (s, add (k, v, m)) = Some (v)  
      by definition of add, find type int_str_list_t hypothesis H1  
    <2>2 hypothesis H2: find (s, add (k, v, m)) = Some (v),  
      prove find (s, m) = Some (v) V s = k  
      assumed  
    <2>e qed by step <2>1, <2>2  
  <1>e conclude ;
```

- No 🤔
- **Split** again...

Splitting the Proof (4 ... 🙄)

- Prove the previously **assumed** <2>1.
- Why $\text{find } (s, \text{add } (k, v, m)) = \text{Some } (v)$?
 - because $\text{add } (k, v, m)$ has the shape $\text{Node } (k, v, m)$
 - and $\text{find } (s, \text{Node } (k, v, m))$, by H1,
 - if $k = s$, find returns especially a Some
 - otherwise we know that $\text{find } (s, m) = \text{Some } (v)$ and find exactly recurses on m .

```
theorem find_spec: all m : Self, all s k : int, all v : string,  
  (find (s, m) = Some (v) V s = k) <-> find (s, add (k, v, m)) = Some (v)  
proof =  
  <1>1 assume m : Self,  
    assume s k : int,  
    assume v : string,  
    prove (find (s, m) = Some (v) V s = k) <-> find (s, add (k, v, m)) = Some (v)  
  <2>1 hypothesis H1: find (s, m) = Some (v) V s = k,  
    prove find (s, add (k, v, m)) = Some (v)  
    <3>1 prove add (k, v, m) = Node (k, v, m)  
      assumed  
    <3>e qed by step <3>1 definition of find hypothesis H1 type int_str_list_t  
  <2>2 hypothesis H2: find (s, add (k, v, m)) = Some (v),  
    prove find (s, m) = Some (v) V s = k  
    assumed  
  ...
```

Splitting the Proof (5 ... 🙄)

- Prove the previously **assumed** <3>1.
- Simply due to how **add** is written (and the type of lists).

```
theorem find_spec: all m : Self, all s k : int, all v : string,  
  (find (s, m) = Some (v) V s = k) <-> find (s, add (k, v, m)) = Some (v)  
proof =  
  <1>1 assume m : Self,  
    assume s k : int,  
    assume v : string,  
    prove (find (s, m) = Some (v) V s = k) <-> find (s, add (k, v, m)) = Some (v)  
    <2>1 hypothesis H1: find (s, m) = Some (v) V s = k,  
      prove find (s, add (k, v, m)) = Some (v)  
      <3>1 prove add (k, v, m) = Node (k, v, m)  
        by definition of add type int_str_list_t  
      <3>e qed by step <3>1 definition of find hypothesis H1 type int_str_list_t  
    <2>2 hypothesis H2: find (s, add (k, v, m)) = Some (v),  
      prove find (s, m) = Some (v) V s = k  
      assumed  
    <2>e qed by step <2>1, <2>2  
  <1>e conclude ;
```

Splitting the Proof (6 ...)

- Prove the previously **assumed** <2>1.
- **Same proof** than for <2>1 (using hypothesis H2)... Finished!

```
theorem find_spec: all m : Self, all s k : int, all v : string,  
  (find (s, m) = Some (v) V s = k) <-> find (s, add (k, v, m)) = Some (v)  
proof =  
  <1>1 assume m : Self,  
    assume s k : int,  
    assume v : string,  
    prove (find (s, m) = Some (v) V s = k) <-> find (s, add (k, v, m)) = Some (v)  
    <2>1 hypothesis H1: find (s, m) = Some (v) V s = k,  
      prove find (s, add (k, v, m)) = Some (v)  
      <3>1 prove add (k, v, m) = Node (k, v, m)  
        by definition of add type int_str_list_t  
      <3>e qed by step <3>1 definition of find hypothesis H1 type int_str_list_t  
    <2>2 hypothesis H2: find (s, add (k, v, m)) = Some (v),  
      prove find (s, m) = Some (v) V s = k  
      <3>1 prove add (k, v, m) = Node (k, v, m)  
        by definition of add type int_str_list_t  
      <3>e qed by step <3>1 definition of find  
        hypothesis H2 type int_str_list_t  
    <2>e qed by step <2>1, <2>2  
  <1>e conclude ;
```

More Structure and Encapsulation

- If we have time...



- Otherwise, for a next or longer lecture...

Parameterization

- In our previous association maps, we used **basic** types.
- Both for keys and values.
- ➔ **No properties** available on these « structures ».
- ➔ Impossible to assume **holding invariants** on them.
- ➔ Possible to **manually** (incorrectly) **alter** them.
- How to build a species taking benefits from other ones?
- We need **parameterization**.

Adding Parameters

```
species AssocMap (Key «is ???», Value «is ???», OptValue is «??? (Value)») =  
representation = pair_list_t (Key, Value) ;  
let empty : Self = Nil ;  
let add (k, v, m : Self) : Self = Node (k, v, m) ;  
let rec find (k, m : Self) : OptValue = ... ;  
...
```

- We need a « structure » for **keys**, one for **values** and one for **optional values**.
- **Optional values** « are made of » **values**.

Use Methods of Parameters (1)

- Since we get **parameters**, we want to use **their methods** to build **those of the species**.

```
species AssocMap (Key «is ???», Value «is ???», OptValue is «??? (Value)») =  
  representation = pair_list_t (Key, Value) ;  
  let empty : Self = Nil ;  
  let add (k, v, m : Self) : Self = Node (k, v, m) ;  
  
  let rec find (k, m : Self) : OptValue =  
    match m with  
    | Nil -> OptValue!none  
    | Node (kcur, v, q) ->  
      if Key!eq (kcur, k) then OptValue!some (v) else find (k, q)  
  termination proof = structural m ;  
  ...
```

← No more **None** but
« the none » of OptValue

- Use the **method** of a **parameter** P by qualifying it by **P!**

Use Methods of Parameters (2)

- Since we get **parameters**, we want to use **their properties** to prove **theorems of the species**.

```
species AssocMap (Key «is ???», Value «is ???», OptValue is «??? (Value)») =  
  representation = pair_list_t (Key, Value) ;  
  let empty : Self = Nil ;  
  let add (k, v, m : Self) : Self = Node (k, v, m) ;  
  
  theorem find_added_not_fails: all k : Key, all v : Value, all m1 m2 : Self,  
    m2 = add (k, v, m1) -> ~ OptValue!eq (find (k, m2), OptValue!none)  
  proof =  
    ... prove Key!eq (k, k) by property Key!eq_reflexive ... ;  
    ...
```

- A property is a **method**.

➔ From a **parameter** P ... qualifying it by **P!**

A property (may be 😊) stating
that Key!eq is reflexive

« is ??? » is what ?

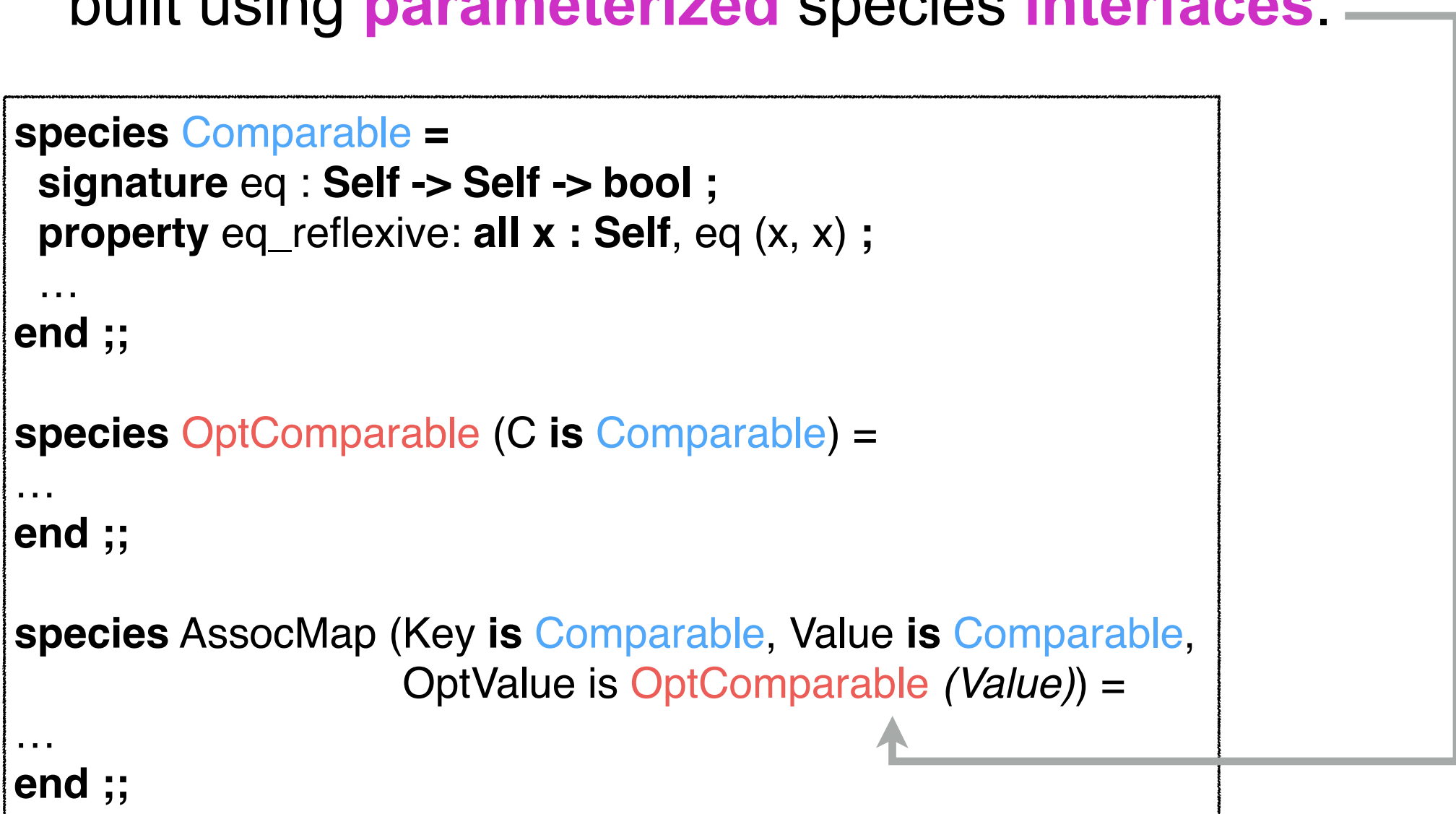
species AssocMap (**Key** «is ???», **Value** «is ???», **OptValue** is «??? (**Value**)») =

- But, finally, « is ??? » **is what** ?
- Some good remarks:
 1. We want to use **methods**... But they must **exist**!
 2. We want to use **functions**... But they have to be **implemented**!
 3. We want an **effective** underlying **type definition**.
 4. We want to rely on **properties**... But, they must be **proved**!
 5. We want to **preserve invariants**, not « manually » accessing parameters internals.
- Some (may be) good answers:
 1. We need a notion of **interface**: the « promised » methods of a species.
 2. **Collections** enforce functions to be **defined**.
 3. **Collections** enforce the **representation** to be **defined**.
 4. **Collections** enforce properties to be **proved**.
 5. **Collections** enforce **abstraction**.

Collection Parameters

- Hence, we need **collection parameters**...
- having **at least** the methods of a specified **interface**.
- A **parameterized** species can have **collection** parameters built using **parameterized** species **interfaces**.

```
species Comparable =  
  signature eq : Self -> Self -> bool ;  
  property eq_reflexive: all x : Self, eq (x, x) ;  
  ...  
end ;;  
  
species OptComparable (C is Comparable) =  
  ...  
end ;;  
  
species AssocMap (Key is Comparable, Value is Comparable,  
                  OptValue is OptComparable (Value)) =  
  ...  
end ;;
```



Inheritance

- A key or a value has to be « something that can be compared ».
 - An optional value is **built from** (**parameterized** by) a **Comparable**.
 - ... but **is** also « something that can be compared ».
- ➔ It also **inherits** from **Comparable**.

```
species Comparable =  
  signature eq : Self -> Self -> bool ;  
  property eq_reflexive: all x : Self, eq (x, x) ;  
  ...  
end ;;  
  
species OptComparable (C is Comparable) =  
  inherit Comparable ;  
  representation = option_t (C) ;  
  ...  
end ;;  
  
species AssocMap (Key is Comparable, Value is Comparable,  
                  OptValue is OptComparable (Value)) =  
  ...  
end ;;
```

Entity Parameters

- FoCaLiZe also proposes **entity parameters**.
- Parameters being **values** of the underlying **representation** of their **collection**.

```
species Comparable =  
  signature eq : Self -> Self -> bool ;  
  property eq_reflexive: all x : Self, eq (x, x) ;  
  ...  
end ;;  
  
species Truc (Value is Comparable, v in Value) =  
  ...  
end ;;
```

- Out of the scope of this lecture (not difficult however).

So what ?

- Parameterization does not change so much the shape of the proofs.
- **Internals** of parameters **no more visible**:
 - ➔ representation **abstracted** (\Rightarrow restrictions on theorems statements).
 - ➔ Body of functions **not visible** (no more facts by **definitions of**).
- ➔ Need to use **properties** on parameters instead.
- Sometimes reveals subtils required properties...
 - usually « well-known »,
 - not even thought about,
- **Equality** properties for instance !!! (c.f. lecture notes).

When to Prove What ?

- In the lecture notes, but for a next or longer presentation...



Advances Proofs



Proof by Cases

- When a property has to be proved on **each possible case**.
- In other words, on **each possible value**.
- Examples:
 - $f(x) = \text{if } x < 0 \text{ then } \dots \text{ else if } x > 0 \text{ then } \dots \text{ else } \dots$
→ 3 cases: $x < 0$, $x > 0$, $x = 0$.
 - $g(y) = \text{match } y \text{ with } | \text{opened} \rightarrow \dots | \text{closed} \rightarrow \dots$
 - Assuming the type of y **only** has the values `opened` and `closed`.
→ 2 cases: $y = \text{opened}$, $y = \text{closed}$.

Simple ... Case(s?)

```
open "basics" ;;  
type flag_t = | On | Off ;;  
  
let constant (x) =  
  match x with  
    | On -> 1  
    | Off -> 1 ;;  
  
theorem constant_is_one: all x : flag_t, constant (x) = 1  
proof = by definition of constant type flag_t ;;
```

- Very simple: Zenon automatically handles!
- Need for the **type** of x.
- Need for the **definition** of constant.

No More Simple for Zenon...

```
open "basics" ;;
```

```
type flag_t = | On | Off ;;
```

```
type answer_t = | Yes | No | Maybe (flag_t) ;;
```

```
(* A pretty complex way to write the identity function... *)
```

```
let f (x) = match x with
```

```
  | Yes -> Yes
```

```
  | No -> No
```

```
  | Maybe (y) -> if y = On then Maybe (On) else Maybe (Off) ;;
```

```
(* Prove that f is indeed the identity. *)
```

```
theorem is_id: all x : answer_t, f (x) = x
```

```
proof = by type answer_t, flag_t definition of f ;;
```

Compile



```
$ focalizec answer_bad.fcl
Invoking ocamlc...
>> ocamlc -I /usr/local/lib/focalize -c answer_bad.ml
Invoking zvtov...
>> zvtov -zenon zenon -new answer_bad.zv
File "answer_bad.fcl", line 17, characters 8-48:
Zenon error: exhausted search space without finding a proof
### proof failed
$
```

Shape of Goals by Cases

- Proof by cases \Rightarrow **Zenon** must apply an **induction principle**.
- The goal **must** have the shape:
 $\text{all } x : t, P(x)$
with the « $\text{all } x : t$ » **explicitly** stated.
- Next, decline the goal for **each case** of value of type **t**.
- **Conclude** the proof by the fact **by type t** and all **intermediate cases** steps.

theorem is_id: **all** $x : \text{answer_t}, f(x) = x$

proof =

<1>1 **prove** $f(\text{Yes}) = \text{Yes}$ **assumed**

<1>2 **prove** $f(\text{No}) = \text{No}$ **assumed**

<1>3 **prove all** $y : \text{flag_t}, f(\text{Maybe}(y)) = \text{Maybe}(y)$ **assumed**

<1>e **qed by step** <1>1, <1>2, <1>3 **type** $\text{answer_t}, \text{flag_t}$ **definition of f**

Prove Cases as Usual

- Proof of each case goal is no more special.

```
theorem is_id: all x : answer_t, f (x) = x
```

```
proof =
```

```
<1>1 prove f (Yes) = Yes by definition of f type answer_t
```

```
<1>2 prove f (No) = No by definition of f type answer_t
```

```
<1>3 prove all y : flag_t, f (Maybe (y)) = Maybe (y)  
    by definition of f type flag_t, answer_t
```

```
<1>e qed by step <1>1, <1>2, <1>3 type answer_t, flag_t definition of f
```


Proof by Induction: Reminder

- Prove a property on **all** the elements of a **set** S.
- Requires a **well-founder strict order** $<$ on S.
- Generalization of recurrence on integers:
 $P(0) \Rightarrow (\forall n \in \mathbb{N}, P(n) \Rightarrow P(n+1)) \Rightarrow \forall n \in \mathbb{N}, P(n).$

- For us: **set** = **type**.

```
type t =  
| C1  
| C2 (t)  
| C3  
| C4 (t * t) ;;
```

- Base cases: **non-recursive** constructors C1, C3.
- Induction cases: **recursive** constructors C2, C4.

$$\begin{aligned} P(C1) \Rightarrow (\forall v : t, P(v) \Rightarrow P(C2(v))) \Rightarrow P(C3) \Rightarrow \\ (\forall v1, v2 : t, P(v1) \Rightarrow P(v2) \Rightarrow P(C4(v1, v2))) \Rightarrow \\ \forall v : t, P(v) \end{aligned}$$

An Example

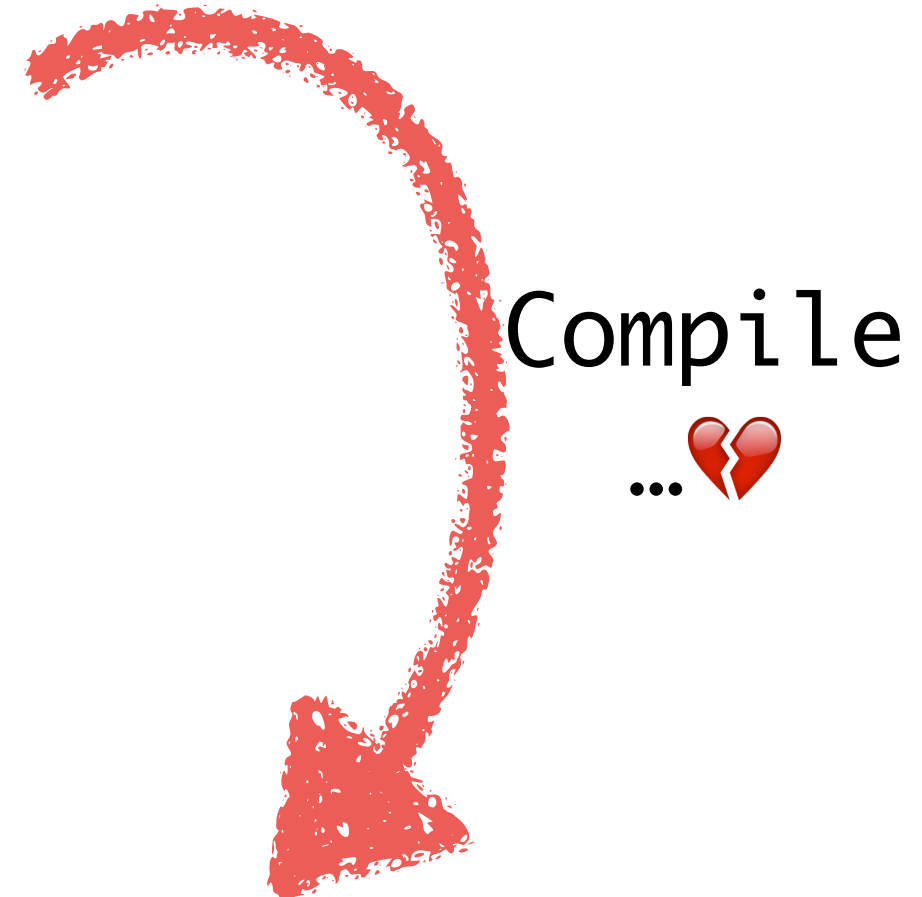
```
open "basics" ;;

type bintree_t =
  | Leaf
  | Node (bintree_t, bool, bintree_t) ;;

let rec f (t) =
  match t with
  | Leaf -> false
  | Node (l, b, r) -> b && f (l) && f (r)
termination proof = structural t ;;

theorem always_false: all t : bintree_t, ~ f (t)
proof = by definition of f type bintree_t ;;
```

- Prove that f always return false.
- Automated way fails...



```
$ focalizec -zvtovopt -script stupid_tree_ko.fcl
Invoking ocamlc...
>> ocamlc -I /usr/local/lib/focalize -c stupid_tree_ko.ml
Invoking zvtov...
>> zvtov -zenon zenon -new -script stupid_tree_ko.zv
File "stupid_tree_ko.fcl", line 16, characters 8-41:
Zenon error: could not find a proof within the memory size limit
### proof failed
$
```

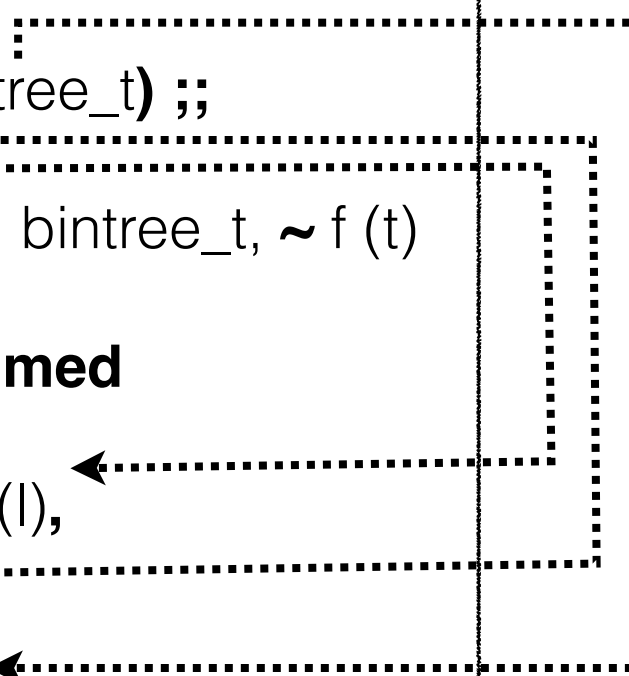
« Manual » Proof by Induction

- Induction: **more general** than proof by cases.
- Requires Zenon to apply an **induction principle**.
- ➔ Same constraint on the global goal: **all $x : t, P(x)$** .
- For **each constructor** C_i of the type **t** one must prove that $P(C_i)$ holds in a **sub-step**.
- If a constructor C_i is **recursive**, then one must introduce each quantified variable and its related **induction hypothesis** in the **same order** than their related parameter appear in the **definition of the type**.
- The **last** sub-step must be a **qed** step using the above steps and the type **t** (and other things if needed).

Proof Scheme

- Theorem's goal has a good shape: **all $t : \text{bintree_t}$, $\sim f(t)$** .
- First step, <1>1: prove the property for the only **base case**.
- Second step, <1>2: prove the property for the only **inductive case**.
- Last step, <1>e: **conclude** using the **type** and **2 above steps**.

```
type bintree_t =  
  | Leaf  
  | Node (bintree_t, bool, bintree_t) ;;  
...  
theorem always_false: all t : bintree_t, ~ f (t)  
proof =  
  <1>1 prove ~ f (Leaf) assumed  
  <1>2 assume l: bintree_t,  
    hypothesis HRecL: ~ f (l),  
    assume b: bool,  
    assume r: bintree_t,  
    hypothesis HRecR: ~ f (r),  
    prove ~ f (Node (l, b, r))  
    assumed  
  <1>e qed by step <1>1, <1>2 type bintree_t
```



End of the Proof

- <1>1: Consequence of f's **body** (and knowledge of `bintree_t`).
 - <1>2:
 - By **induction hypotheses** f returns `false` when called on both `l` and `r`.
 - f's **definition** shows a « logical and » between these returned values and the current node's value.
- Result is `false`. CQFD.

```
theorem always_false: all t : bintree_t, ~ f (t)
proof =
  <1>1 prove ~ f (Leaf)
    by definition of f type bintree_t
  <1>2 assume l: bintree_t,
    hypothesis HRecL: ~ f (l),
    assume b: bool,
    assume r: bintree_t,
    hypothesis HRecR: ~ f (r),
    prove ~ f (Node (l, b, r))
    by hypothesis HRecL, HRecR type bintree_t definition of f
  <1>e qed by step <1>1, <1>2 type bintree_t
```

Conclusion

- We examined:
 - What does mean « *proving programs* ».
 - How to write formal proofs of first-order properties in FoCaLiZe.
 - How to basically program in FoCaLiZe.
 - How to state properties on programs and prove them
 - How to write more advanced proofs in a « *manual* » way.

Not Seen in this Lecture

- **Inheritance**, **parametrization** features of FoCaLiZe (only in lecture notes).
- **Termination** proofs of **recursive** functions.
- **Higher order** properties and their proofs.
- **Imperative** programming.

The End



Material for this lecture at

<http://perso.ensta-paristech.fr/~pessaux/ejcp-2014>

Additional resources for FoCaLiZe at

<http://focalize.inria.fr>

Bug (if any)-tracker at

<http://focalize.inria.fr/bugzilla>