# Another Tutorial for FoCaLize: Playing with Proofs

François Pessaux
ENSTA ParisTech
francois.pessaux@ensta-paristech.fr

March 2014

## 1 Forewords

### 1.1 Content

This document is a tutorial for FoCaLize, describing how to develop proofs of properties using Zenon. Differently from other tutorials, we won't focalize on mathematical developments, preferring to show the language in action on programs closer to what "usual programers" develop in the "everyday life".

To get in touch with basic Zenon capabilities, we will first address very simple first order logic properties with their proofs. This will allow introducing the notion of hierarchical proofs. Then, we will program a simple 3 traffic signals controller to apply these skills on properties directly related to the program we will write. The aim is to show what are the kind of properties one may want to state and how their proofs get related to the types and functions definition of a program.

### 1.2 Notations and Recommandations

In the rest of this tutorial, pieces of FoCaLize code will be presented in frames, as in this example:

```
use "basics" ;;
species Controller =
```

When introduced in the running text, FoCaLize keywords will appear in a special font like **`property`**. Terms representing specific concepts of FoCaLize are introduced using an emphasized font, for example *collection*. Finally, commands and file names are in bold font, for example **focalizec fo_logic.fcl**.

## 2 Dealing with first order logic theorems

The first part of this tutorial intends teaching how to use Zenon on simple boolean properties. The aim is to show how Zenon can help making whole part of basic inference steps one usually makes explicit in tools like Coq. It must clearly understood that Zenon is not a proof checker but a theorem prover. Obviously it will not automatically demonstrate itself any property. However, combined with the FoCaLize proof language, it will automate tedious combinations of “sub-lemmas” one usually “think intuitively feasible”.

In the following examples, we won’t create species. Instead, to get rid of complexity induced by FoCaLize structures, we will state and prove theorems at “top-level”.

## 2.1 A so simple property: fully automated proof

Let’s first address the following property: $\forall a, b : \texttt{boolean}, a \Rightarrow b \Rightarrow a$. We write this in FoCaLize as follows, in a source file **focalizec ex_implications.fcl**:

Listing 1: ex_implications.fcl

```
open "basics" ;;

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

Here, we stated a property and directly asked Zenon to find a proof without any direction. Zenon then uses its internal knowledge of first order logic to solve the goal. At this stage it is possible to compile the program using the command **focalizec ex_implications.fcl** and get few messages with not error:

```
Invoking ocamlc...
>> ocamlc -I /usr/local/lib/focalize -c ex_implications.ml
Invoking zvtov...
>> zvtov -zenon /usr/local/bin/zenon -new ex_implications.zv
Invoking coqc...
>> coqc -I /usr/local/lib/focalize -I /usr/local/lib/zenon ex_implications.v
```

Compilation, code generation and Coq verification were successful. Let’s investigate the tree of this proof, where we wrote hypotheses (context) in green and goals in blue. The intuition is that we lift the two implications as hypotheses and from the hypothesis a we trivially prove our goal.

$$\dfrac{\dfrac{a, b \vdash a}{a \vdash b \Rightarrow a}\ (\Rightarrow\text{-INTRO})}{\vdash a \Rightarrow b \Rightarrow a}\ (\Rightarrow\text{-INTRO})$$

In FoCaLize, this *proof* is achieved as a simple hierarchical sequence of intermediate *steps*. A proof step starts with a *proof bullet*, which gives its nesting level. The top-level of a proof is 0. In a *compound proof*, the steps are at level one plus the level of the proof itself.

Listing 2: ex_implications.fcl (2)

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

```
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 *) ;;
```

Here, the *steps* <1>1 and <1>2 are at level 1 and form the compound proof of the top-level theorem. Step <1>1 also has a compound proof (whose goal is b **->**a), made of steps <2>1 and <2>2. These latter are at level 2 (one more than the level of their enclosing step).

After the proof bullet comes the *statement* of the step, introduced by the keyword **prove**. This is the proposition that is asserted and proved by *this step*. At the end of this step's proof, it becomes available as a *fact* for the next steps of this proof and deeper levels sub-goals. In our example, step <2>1 is available in the proof of <2>2, and <1>1 is available in the proof of <1>2. Note that <2>1 is **not** available in the proof of <1>2 since <1>2 is located at a strictly lower nesting level than <2>1.

After the statement is the *proof of the step*. This is where either you ask Zenon to do the proof from *facts* (hints) you give it, or you decide to split the proof in "sub-steps" on which Zenon will finally by called and that will serve (still with Zenon) to finally prove the current goal by combining these "sub-steps" lemmas.

For instance, the proof of the whole theorem (which is itself a statement) is not directly asked to Zenon as it was the case in the first example (with the simple fact **proof =conclude**). It has been decided to split it in one sub-goal <1>1: prove b **->**a. The same structure is applied to this goal which is split in the sub-goals <2>1 and <2>2.

In the proof of sub-goal <2>1, appears a *fact*: **by hypothesis** h1. Here Zenon is asked to find a proof of the current goal, using hints it is provided with. You are responsible in giving Zenon facts it will need to finally find a proof. It will combine them in accordance with logical rules, but if it is missing material for the proof, it will never succeed. Here, Zenon is told that it should be able to prove the goal only using the hypothesis h1 we introduced. It will obviously succeed since the goal is exactly the hypothesis h1.

From the proof of a, i.e. the step <2>1, we can conclude the enclosing goal (**prove** b **->**a). This is done by the **qed** step whose aim is to close the enclosing proof by mean of the provided facts (here the intermediate lemma stated by the step <2>1).

Finally, coming back to the remaining part of the proof, i.e. the previous nesting level, we want to solve its goal (which is the whole theorem). In the same manner, from the step `<1>1` that proved that `b ->a` under the hypothesis `a`, we can conclude. We then invoke the statement **`conclude`** which is equivalent to tell Zenon to use as facts, all the available proof steps in the scope. Hence this is equivalent here to write **`qed by step`** `<1>1`. Note that with **`conclude`** the **`qed`** is optional since this statement implicitly marks the end of the proof related to the current goal.

Having a look backward to compare our Coq and FoCaLize proofs, we can clearly see the same processing order. We introduced the implications as hypotheses, with `intro` in Coq and **`assume`** in FoCaLize. Then we used the hypothesis `h1`, with `exact h1` in Coq and **`by hypothesis`** `h1` in FoCaLize. The implicit nested structure of the proof is made explicit in FoCaLize.

## 2.2 Still simple but easier with Zenon

Let's continue with simple first-order logic properties and let's try to demonstrate the following statement: $\forall a, b : \texttt{boolean}, (a \wedge b) \Rightarrow (a \vee b)$. We can easily build the tree of this proof as follows:

$$\cfrac{\cfrac{\cfrac{a \wedge b \vdash a \wedge b}{a \wedge b \vdash b}\ (\wedge\text{-INTRO})}{a \wedge b \vdash a \vee b}\ (\vee\text{-ELIM})}{\vdash a \wedge b \Rightarrow a \vee b}\ (\Rightarrow\text{-INTRO})$$

Note that we chose to prove `b` but we could have chose to prove `a` instead. We now address this property in FoCaLize again, making explicit all the steps of the proof we did.

Listing 3: and_or.fcl

```
open "basics" ;;

theorem and_or : all a b : bool, (a /\ b) -> (a \/ b)
  proof =
  (* Sketch: assume a /\ b, then prove b as trivial consequence of a /\ b.
       *)
  <1>1 assume a : bool, b : bool,
       hypothesis h1: a /\ b,
       prove a \/ b
       <2>1 prove b
            by hypothesis h1
       <2>2 qed
            by step  <2>1
  <1>2 conclude (* or, qed conclude, or even qed by step <1>1 *) ;;
```

We can see that step `<2>1` is directly proven by hypothesis `h1`, without making explicit the `∧-ELIM` rule of the proof tree. Here we rely on Zenon to find the proof of this step. Again, this property is simple enough to get it proved in a fully automated way by Zenon.

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

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

## 3 Playing with programs

Let's now introduce more material than simple first-order logic formulae. In this section we will first introduce functions, then inductive types in stated properties. Finally we will see that such previously stated properties can be used as lemmas to prove further theorems.

### 3.1 Introducing functions

One may want to prove that logical or ($\vee$) is commutative, i.e. that $(a \vee b) \Rightarrow (b \vee a)$. But, on atomic properties, this would again be trivial for Zenon. Instead, of making again explicit trivial proof steps, we will now extend this formula with the (trivial again) identity function. Hence we want to prove that if `id` is defined as $\lambda x.x$, then $\forall a, b, c, d : int, (id(a) = b \vee id(c) = d) \Rightarrow (c = id(d) \vee a = id(b))$. We write this in FoCaLize like:

Listing 4: or_id_com.fcl

```
1  open "basics" ;;
2
3  let id (x) = x ;;      (* Definition of the identity function. *)
4
5  theorem or_id_commutative:
6    all a b c d : int, (id (a) = b \/ id (c) = d) -> (c = id (d) \/ a = id (
         b))
7  proof = conclude ;;
```

As shown in the previous code snippet, we optimistically asked Zenon to automatically handle the proof. So, let's invoke the compilation command: **focalizec or_id_com.fcl** and we get:

```
Invoking ocamlc...
>> ocamlc -I /usr/local/lib/focalize -c or_id_com.ml
Invoking zvtov...
>> zvtov -zenon zenon -new or_id_com.zv
File "or_id_com.fcl", line 7, characters 8-16:
Zenon error: exhausted search space without finding a proof
### proof failed
```

We should have not been so optimistic: line 7, where we invoked **`conclude`**, Zenon did not find any proof despite it natively knows equality and basic logic. We will investigate this point incrementally, and once understood, we will see that we could have fixed the proof more quickly (and more lazily). The aim here is to make again hierarchical proof steps explicit to train splitting proofs in intermediate cases (which will quickly become mandatory with realistic proofs).

What is the sketch of the proof ? We basically want to assume that $id(a) = b \vee id(c) = d$ then prove $c = id(d) \vee a = id(b)$. We will now build the structure of the proof incrementally, adding steps from our intuition, but leaving temporarily them unproved. By this mean, we do not yet focus on each sub-goal, rather on the global scheme of the proof. In some sense, we add intermediate lemmas and want to ensure that (obviously provided they will be proven) Zenon can find a proof of the global theorem combining these lemmas. So, let's simply add a step assuming $id(a) = b \vee id(c) = d$ and "fake-prove" that $c = id(d) \vee a = id(b)$. Then, we ask Zenon to conclude the whole theorem by this step.

Listing 5: or_id_com.fcl (2)

```
open "basics" ;;

let id (x : int) = x ;;

theorem or_id_commutative:
  all a b c d : int, (id (a) = b \/ id (c) = d) -> (c = id (d) \/ a = id (
      b))
proof =
  (* Sketch: assume (a = b \/ c = d), then prove (c = id (d) \/ a = id (b)
      . *)
  <1>1 assume a : int, b : int, c : int, d : int,
       hypothesis h1: id (a) = b \/ id (c) = d,
       prove c = id (d) \/ a = id (b)
       assumed
  <1>2 qed by step <1>1 ;;
```

We remark the apparition of the keyword **`assumed`** whose aim is to loosely make a "fake" proof. In a sense, this allows stating the related goal as being an axiom. Obviously, this is cheating since the proof gets admitted, hence do not reflect anymore a really holding property. Decent FoCaLize developments should not have such "proofs" remaining. However, this can be the only solution when dealing with properties that can't be proved because relying of third-party code, not available in FoCaLize, or when properties deal with higher-order (Zenon doesn't handle this aspect). In the present case, we only use it as a temporary placeholder to help us refining our proof from the general idea to the fine-grain sequence of steps.

At this point the source file can be compiled invoking **focalizec or_id_com.fcl** which hopefully gives not error. It is pretty satisfactory that from the only step of the proof, having lifted the left part of the implication as hypothesis, the whole theorem can be proved !

It remains now to really prove that $c = id(d) \vee a = id(b)$ under our hypothesis. This is achieved by proving it in both cases where we have the left and the right parts of our disjunctive hypothesis. We can then add these new steps, still assuming their proofs, just to ensure that our intuition of the scheme is consistent.

Listing 6: or_id_com.fcl (3)

```
open "basics" ;;

let id (x : int) = x ;;

theorem or_id_commutative:
  all a b c d : int, (id (a) = b \/ id (c) = d) -> (c = id (d) \/ a = id (
      b))
proof =
  (* Sketch: assume (a = b \/ c = d), then prove (c = id (d) \/ a = id (b)
      . *)
  <1>1 assume a : int, b : int, c : int, d : int,
       hypothesis h1: id (a) = b \/ id (c) = d,
       prove c = id (d) \/ a = id (b)
       <2>1 hypothesis h2: id (c) = d,
            prove c = id (d)
            assumed
       <2>2 hypothesis h3: id (a) = b,
            prove a = id (b)
            assumed
       <2>3 qed by step <2>1, <2>2 hypothesis h1
  <1>2 qed by step <1>1 ;;
```

We introduced steps `<2>1` and `<2>2` and said that they should be sufficient for Zenon to prove the enclosing goal. To conclude step `<2>3` , we must make explicit that the 2 steps `<2>1` and `<2>2` are performed under assumptions being the two parts of the disjunction we had in hypothesis `h1`, otherwise these 2 cases are not relevant (in other words, why did we state and prove them). Hence, step `<2>3` is only missing this information: **`by ... hypothesis`** `h1` ! We again compile the program and get pretty happy to see that, provided these two steps, Zenon really succeeds.

So, we now need to continue our incremental process and really prove that on one side `c =id (d)` and on the other `a =id (b)`. Since Zenon looks smart, why not asking him to **`conclude`** ? Let's try...

Listing 7: or_id_com.fcl (4)

```
open "basics" ;;

let id (x : int) = x ;;

theorem or_id_commutative:
  all a b c d : int, (id (a) = b \/ id (c) = d) -> (c = id (d) \/ a = id (
      b))
proof =
  (* Sketch: assume (a = b \/ c = d), then prove (c = id (d) \/ a = id (b)
      . *)
  <1>1 assume a : int, b : int, c : int, d : int,
       hypothesis h1: id (a) = b \/ id (c) = d,
       prove c = id (d) \/ a = id (b)
       <2>1 hypothesis h2: id (c) = d,
            prove c = id (d)
            conclude
       <2>2 hypothesis h3: id (a) = b,
            prove a = id (b)
            conclude
       <2>3 qed by step <2>1, <2>2 hypothesis h1
  <1>2 qed by step <1>1 ;;
```

It is now time to compile the program, again with the command **focalizec or_id_com.fcl** and we get:

```
Invoking ocamlc...
```

```
>> ocamlc -I /usr/local/lib/focalize -c or_id_com3.ml
Invoking zvtov...
>> zvtov -zenon zenon -new or_id_com3.zv
File "or_id_com3.fcl", line 14, characters 12-20:
Zenon error: exhausted search space without finding a proof
### proof failed
```

clearly stating that Zenon didn't find any proof. Let's just inspect the proof tree we tried to build:

$$\dfrac{\dfrac{(\vee\text{-INTROL})\;\dfrac{\begin{array}{c}\text{How to know id (a) is equal to a ?}\\ \text{How to know b is equal to id (b) ?}\end{array}}{id(a)=b\vdash a=id(b)}}{id(a)=b\vdash c=id(d)\vee a=id(b)}\quad\dfrac{\dfrac{\begin{array}{c}\text{How to know id (c) is equal to c ?}\\ \text{How to know d is equal to id (d) ?}\end{array}}{id(c)=d\vdash c=id(d)}\;(\vee\text{-INTROR})}{id(c)=d\vdash c=id(d)\vee a=id(b)}}{\dfrac{id(a)=b\vee id(c)=d\vdash c=id(d)\vee a=id(b)}{\vdash (id(a)=b\vee id(c)=d)\Rightarrow(c=id(d)\vee a=id(b))}\;(\Rightarrow\text{-INTRO})}\;(\vee\text{-ELIM})$$

The blocking point is that the proof strongly rely on the fact that `id` being the identity, $id(a)=a, id(b)=b, id(c)=c$ and $id(d)=d$, but Zenon is not aware of this. What Zenon needs is to know about the *definition* of the function `id`.

Here comes a new fact (in addition to the already seen facts **conclude**, **hypothesis** and **step**): the **definition of** stating that Zenon must consider a whole function (i.e. including its body – its *definition*) to try finding a proof. Hence, our proof of each intermediate steps `<2>1` and `<2>2` will be done **by definition of** `id`. Moreover, as shown in our above proof tree, both goals (`<2>1` and `<2>2`) rely on their related hypothesis (`h2` and `h3`).

Listing 8: or_id_com.fcl (5)

```
open "basics" ;;

let id (x : int) = x ;;

theorem or_id_commutative:
  all a b c d : int, (id (a) = b \/ id (c) = d) -> (c = id (d) \/ a = id (
      b))
proof =
  (* Sketch: assume (a = b \/ c = d), then prove (c = id (d) \/ a = id (b)
      . *)
  <1>1 assume a : int, b : int, c : int, d : int,
       hypothesis h1: id (a) = b \/ id (c) = d,
       prove c = id (d) \/ a = id (b)
       <2>1 hypothesis h2: id (c) = d,
            prove c = id (d)
            by hypothesis h2 definition of id
       <2>2 hypothesis h3: id (a) = b,
            prove a = id (b)
            by hypothesis h3 definition of id
      <2>3 qed by step <2>1, <2>2 hypothesis h1
  <1>2 qed by step <1>1 ;;
```

We now compile our whole and definitive program and get proofs finally done and accepted by Coq:

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

Now we suffered enough, splitting the proof of this theorem in several parts and learned the **`by definition of`** fact, let's just discover that all the intermediate steps we did, dealing with *basic logic combinations* . . . could again be automatically done by Zenon and that, only telling it that it should use the definition of `id` would have been sufficient !

Listing 9: or_id_com_shortest.fcl

```
open "basics" ;;

let id (x : int) = x ;;

theorem or_id_commutative:
  all a b c d : int, (id (a) = b \/ id (c) = d) -> (c = id (d) \/ a = id (
      b))
proof = by definition of id ;;
```

We can invoke the compiler on this shortened version of our program (assuming the source file is **or_id_com_shortest.fcl**): **focalizec or_id_com_shortest.fcl** and get the same successful happy end:

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

## 3.2 Introducing pairs

FoCaLize natively provides the type of tuples. Zenon knows only about *pairs* (i.e. 2-components tuples). However, until enhancements of FoCaLize and/or Zenon, it is possible to encode general tuples as nested pairs. For instance, instead of manipulating the type `(int * bool * string)`, one will manipulate `(int * (bool * string))` even if it is a bit cumbersome.

We will now study some proofs dealing with pairs, see what Zenon is able to handle and how we can explicitly write such proofs. We first start by the initial type definition, aliasing pairs of `int`s to a type `int_pair_t`. Such a definition is written:

```
type int_pair_t = alias (int * int) ;;
```

and simply declares a new type constructor compatible with `(int * int)`.

Zenon natively knows about `fst :('a * 'b)->'a` and `snd :('a * 'b)->'b` functions, extracting the first and second component of a pair. For instance, it will be able to prove that extracting components of one pair with 2 equal components will lead to 2 equal values:

Listing 10: same_comps.fcl

```
open "basics" ;;

type int_pair_t = alias (int * int) ;;

theorem same_components :
  all v : int_pair_t, all x : int, v = (x, x) -> fst (v) = snd (v)
  proof = conclude ;;
```

## 3.3 Playing with pairs

We will now prove another simple property to continue using the hierarchical way to write proofs, hence train to make explicit steps for later, when such splits will be mandatory. We want to prove the property:

$$\forall v_1, v_2 : \texttt{int}, \forall v : \texttt{int\_pair\_t},\ v = (v1, v2) \Rightarrow \sim (v1 = v2) \Rightarrow\ (\texttt{fst}(v) = \texttt{snd}(v))$$

This obviously can be proven by Zenon as shows the following formulation in FoCaLize:

Listing 11: diff_comps.fcl

```
open "basics" ;;

type int_pair_t = alias (int * int) ;;

theorem different_components :
  all v1 v2 : int, all v : int_pair_t,
  v = (v1, v2) -> ~ (v1 = v2) -> ~ (fst (v) = snd (v))
  proof = conclude ;;
```

However, we want to prove it ourselves (nearly, Zenon will finally still provide the glue between our steps)! We first need to expose the sketch of the proof: first assume the 2 implications, then prove $\sim (\texttt{fst}(v) = \texttt{snd}(v))$. To do so, we will demonstrate that in fact $\texttt{fst}(v) = v_1$, that $\texttt{snd}(v) = v_2$, and conclude by the hypothesis that $v_1 \neq v_2$.

Listing 12: diff_comps.fcl (2)

```
open "basics" ;;

type int_pair_t = alias (int * int) ;;

theorem different_components_manual :
 6   all v1 v2 : int, all v : int_pair_t,
 7   v = (v1, v2) -> ~ (v1 = v2) -> ~ (fst (v) = snd (v))
 8   proof =
 9   <1>1 assume v1 : int, v2 : int, v : int_pair_t,
10        hypothesis h1: v = (v1, v2),
11        hypothesis h2: ~ (v1 = v2),
12        prove ~ (fst (v) = snd (v))
13        <2>1 prove fst (v) = v1
14             by hypothesis h1
15        <2>2 prove snd (v) = v2
16             by hypothesis h1
17        <2>3 qed
18             by step <2>1, <2>2 hypothesis h2
19   <1>2 conclude ;;
```

In lines 10 and 11, we lift the implications premises as hypotheses, then the remaining goal is **prove** `~(fst (v)=snd (v))`. Then we prove in step `<2>1` that `fst (v)=v1` which is obtained from the fact that `v` is a pair (hypothesis `h1` ) and Zenon's knowledge about `fst`. We prove that `snd (v)=v2` by the same means. And finally from these 2 intermediate steps and the hypothesis that $v_1 \neq v_2$ (`h2`) we achieve demonstration of the goal `<1>1`.

## 3.4 Introducing inductive types

Realistic programs usually do not only involve integers and pairs: inductive type definitions are a powerful mean to model data-structures. FoCaLize doesn't escape this rule and Zenon makes possible to reason on such type definitions to automate proofs. An inductive type definition introduces several *value constructors* for a type. For instance:

```
type signal_t = | Red | Orange | Green ;;
```

declares the **new** type `signal_t` as containing the **only** 3 values `Red`, `Orange` and `Green`. These values are all different from each other.

Moreover, an inductive type definition can introduce parametrised constructors, possibly by values of the type itself: we have a recursive type definition:

```
type peano_t = | Z | S (peano_t) ;;
```

declares the **new** type `peano_t` as containing the **only** 2 values `Z` and `S`, this latter embedding a value of type `peano_t` itself. We recognize here the usual definition of Peano's integers.

As a summary, inductive definitions natively introduce 2 important concepts used all over proofs:

- *The injectivity of value constructors*: a value of such a type is one of its constructors and nothing else, constructors being all different from each other.
- *The induction principle*: assuming a property holding on constant value constructors, if this property holds for any parametrised value constructor, then it holds for any values of this type. This is indeed a generalization of the well-known recurrence principle on natural numbers.

We will first show that Zenon greatly helps by knowing injectivity of constructors. The aim will be to demonstrate that any value of type `signal_t` is equal to either `Red`, or `Orange` or `Green`. Hence we state the theorem:

Listing 13: signal.fcl

```
open "basics" ;;

type signal_t = | Red | Orange | Green ;;

theorem signal_t_exclu :
  all a : signal_t, (a = Red) \/ (a = Orange) \/ (a = Green)
  proof = conclude ;;
```

and invoke the compiler to get:

```
Invoking ocamlc...
>> ocamlc -I /usr/local/lib/focalize -c signal.ml
Invoking zvtov...
>> zvtov -zenon zenon -new signal.zv
File "signal.fcl", line 7, characters 10-18:
Zenon error: exhausted search space without finding a proof
### proof failed
```

Zenon didn't find any proof despite we promised it knew how to reason on inductive types! In fact, it was given no fact, no clue, so how could it guess that this property was induced by the underlying type definition? It it important to keep in mind that Zenon only implicitly uses basic logic combinations: it will never use all the material available in a program! So, we just need to tell him that this proof can be deduced from the type definition of `signal_t`.

Here comes a new fact (in addition to the already seen facts **`conclude`**, **`hypothesis`**, **`step`** and **`definition of`**): the **`type`** fact stating that the definition of the following type must be used. We modify our program just inserting this new fact and get:

Listing 14: signal.fcl (2)

```
open "basics" ;;

type signal_t = | Red | Orange | Green ;;

theorem signal_t_exclu :
  all a : signal_t, (a = Red) \/ (a = Orange) \/ (a = Green)
  proof = <1>1 qed by type signal_t ;;
```

which is perfectly proven now. This could seem not so wonderful on such an obvious property, but this means that using Zenon, such intrinsic property of inductive type definitions is natively understood, as long as Zenon is told to use it by the fact **`by ... type ...`**. There is no need to explicitly invoke and manipulate the induction principle.

We can also show that mutual exclusion of value constructors are native for Zenon: let's prove that if a value of type `signal_t` is equal to `Red`, then it is different of `Green`. This can appear more than obvious, such a property, often used while reasoning by cases, requires some intermediate steps (mostly applying the induction principle and discriminations on the constructors). Let's state and prove this property in FoCaLize:

Listing 15: signal2.fcl

```
open "basics" ;;

type signal_t = | Red | Orange | Green ;;

theorem signal_t_exclu2 :
  all a : signal_t, a = Red -> ~ (a = Green)
  proof = <1>1 qed by type signal_t ;;
```

## 3.5 When automatic induction fails

In the previous examples, we proved very simple facts and Zenon directly found proofs in one shot **`by type ...`**, i.e. implicitly using induction on the related type. However it is not always the case. It may be needed to explicitly write some proofs, proving the base cases then each inductive cases. In such a configuration, the **`by type ...`** won't apply alone.

It must be clear that using **`by type ...`** alone (again, implying simple induction of the type) only applies in case where the goal has a shape **`all`** `x :t, P (x)`. This especially means that a "one shot proof" must not start by eliminating the **`all`** `x :t` as we usually did.

The fact **`by type ...`** is not reduced to induction: it also states that a proof needs to know about the constructors of a type. Hence, that's not because a "one shot proof" failed that the fact **`by type ...`** will not be needed.

### 3.5.1 Simple example

We first start with a simple theorem stating that a function `zero` defined recursively always return 0. We try to directly ask Zenon to apply the induction principle to solve the goal:

Listing 16: zero.fcl (1)

```
open "basics" ;;

type peano_t = | Z | S (peano_t) ;;

let rec zero (x) =
 match x with
 | Z -> 0
 | S (y) -> zero (y)
termination proof = structural x ;;

theorem always_0: all x : peano_t, zero (x) = 0
proof = by definition of zero type peano_t ;;
```

and see that the proof failed:

```
Invoking ocamlc...
>> ocamlc -I /usr/local/lib/focalize -c zero.ml
Invoking zvtov...
>> zvtov -zenon zenon -new zero.zv
File "zero.fcl", line 12, characters 8-42:
Zenon error: could not find a proof within the memory size limit
### proof failed
```

We now need to split the proof in 3 steps: one for the base case, one for the inductive case, and the final **qed** combining the two former. As usually, we leave the difficult part (the induction case) initially **assumed** to ensure that our idea of the proof passes with Zenon. We only really prove the base case since it is very simple and only depends on the definition of the function `zero` and the type `peano_t`.

Listing 17: zero.fcl (2)

```
open "basics" ;;

type peano_t = | Z | S (peano_t) ;;

let rec zero (x) =
 match x with
 | Z -> 0
 | S (y) -> zero (y)
termination proof = structural x ;;

theorem always_0: all x : peano_t, zero (x) = 0
proof =
  <1>1 prove zero (Z) = 0
       by definition of zero type peano_t
  <1>2 prove all y : peano_t, zero (y) = 0 -> zero (S (y)) = 0
       assumed
  <1>3 qed by step <1>1, <1>2 type peano_t
;;
```

The proof is now found. **The most important point to understand is that Zenon could apply the induction principle in step <1>3 because it has two steps of the form:**

- **P (base case)**
- **all $y$ : t, P ($y$) $->$ P (inductive case using $y$)**

Our property being `zero (...)=0`, the step `<1>1` is the base case: `zero (Z)=0` and the step `<1>2` is the induction case: `all y :peano_t, zero (y)=0 ->zero (S (y))=0`.

We can now end the proof by really proving the step `<1>2`. We first introduce `y` and the induction hypothesis `inH` in the context, then we must prove `zero (S (y))=0`. This last goal is simply a consequence of the induction hypothesis, the definition of the function `zero` and the definition of the type `peano_t`. Note that in this step, the type `peano_t` is not used for induction: Zenon only needs it to know the constructor `S`.

Listing 18: zero.fcl (3)

```
open "basics" ;;

type peano_t = | Z | S (peano_t) ;;

let rec zero (x) =
 match x with
 | Z -> 0
 | S (y) -> zero (y)
termination proof = structural x ;;

theorem always_0: all x : peano_t, zero (x) = 0
proof =
  <1>1 prove zero (Z) = 0
       by definition of zero type peano_t
  <1>2 prove all y : peano_t, zero (y) = 0 -> zero (S (y)) = 0
       <2>1 assume y : peano_t,
            hypothesis indH: zero (y) = 0,
            prove zero (S (y)) = 0
            by hypothesis indH definition of zero type peano_t
       <2>2 qed by step <2>1
  <1>3 qed by step <1>1, <1>2 type peano_t
;;
```

The proof is now complete and the compilation is a success:

```
Invoking ocamlc...
>> ocamlc -I /usr/local/lib/focalize -c zero.ml
Invoking zvtov...
>> zvtov -zenon zenon -new  -script zero.zv
Invoking coqc...
>> coqc  -I /usr/local/lib/focalize  -I /usr/local/lib/zenon zero.v
```

### 3.5.2 More complex example

In this next example, we propose to prove that given a structure of binary tree, mirroring it twice is the identity (result tree is the same than initial tree. We first define the tree structure, the mirror function and state our property asking Zenon to prove it itself.

Listing 19: tree_mirror1.fcl

```
open "basics" ;;

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

let rec mirror (t) =
  match t with
  | Leaf -> Leaf
  | Node (l, i, r) -> Node (mirror (r), i, mirror (l))
termination proof = structural t ;;

theorem double_mir_is_id : all t : bintree_t, mirror (mirror (t)) = t
proof = by type bintree_t definition of mirror ;;
```

As planned and unfortunately, after a while, Zenon does not find any proof with so much material :

```
Invoking ocamlc...
>> ocamlc -I /usr/local/lib/focalize -c tree_mirror1.ml
Invoking zvtov...
>> zvtov -zenon zenon -new tree_mirror1.zv
File "tree_mirror1.fcl", line 14, characters 8-46:
Zenon error: could not find a proof within the memory size limit
### proof failed
```

As in the previous example, we need to split the proof in 3 steps: one for the base case, one for the inductive case, and the final **qed** combining the two former. We postpone the difficult part (the induction case) for later and mark it **assumed**. We however prove the base case since it is very simple and only depends on the definition of the function `mirror` and the type `bintree_t`. Note that for the induction case, we however state the induction hypotheses (`ir1` and `ir2`) before stating the goal of this case.
**TODO 03/2014**: Add info about order of the premises when several recursive calls and variables to bind. Add info between equivalence between implications and universal quantifications versus **hypothesis** and **assume**.

Listing 20: tree_mirror2.fcl

```
open "basics" ;;

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

let rec mirror (t) =
  match t with
  | Leaf -> Leaf
  | Node (l, i, r) -> Node (mirror (r), i, mirror (l))
termination proof = structural t
;;

theorem double_mir_is_id : all t : bintree_t, mirror (mirror (t)) = t
proof =
  <1>1 (* Base case. *)
       prove mirror (mirror (Leaf)) = Leaf
       by definition of mirror type bintree_t
  <1>2 assume ll : bintree_t,
       (* Induction hypothesis. *)
       hypothesis ir2: mirror (mirror (ll)) = ll,
       assume i : int,
       (* Induction hypothesis. *)
       assume rr : bintree_t,
       hypothesis ir1: mirror (mirror (rr)) = rr,
       (* Recursive case. *)
       prove mirror (mirror (Node (ll, i, rr))) = Node (ll, i, rr)
       assumed
  <1>3 qed by step <1>1, <1>2 type bintree_t
;;
```

Invoking focalizec on this program succeeds and it is now time to complete our last proof. This proof, even if intuitive, can't be solved directly by Zenon. Hence we have to detail it. Basically the proof sketch is to "unfold" twice the function `mirror` (i.e. to look at the result of `mirror (Node (...)))` then to then use

induction hypotheses (`ir1` and `ir2`) to finally get the effective equality of our two terms.
**TODO 03/2014**: Not really, just a bit more of work as shown in the code. Update discussion with the fixed proof.

Listing 21: tree_mirror.fcl

```
open "basics" ;;

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

let rec mirror (t) =
  match t with
  | Leaf -> Leaf
  | Node (l, i, r) -> Node (mirror (r), i, mirror (l))
termination proof = structural t
;;

theorem double_mir_is_id : all t : bintree_t, mirror (mirror (t)) = t
proof =
  <1>1 (* Base case. *)
       prove mirror (mirror (Leaf)) = Leaf
       by definition of mirror type bintree_t
  <1>2 assume ll : bintree_t,
        (* Induction hypothesis. *)
       hypothesis ir2: mirror (mirror (ll)) = ll,
       assume i : int,
        (* Induction hypothesis. *)
       assume rr : bintree_t,
       hypothesis ir1: mirror (mirror (rr)) = rr,
        (* Recursive case. *)
       prove mirror (mirror (Node (ll, i, rr))) = Node (ll, i, rr)
       <2>1 prove mirror (Node (ll, i, rr)) = Node (mirror (rr), i, mirror
            (ll))
            by definition of mirror type bintree_t
       <2>2 prove mirror (Node (mirror (rr), i, mirror (ll))) =
                  Node (mirror (mirror(ll)), i, mirror (mirror (rr)))
            by type bintree_t definition of mirror
       <2>3 prove Node (mirror (mirror(ll)), i, mirror (mirror (rr))) =
                  Node (ll, i, rr)
            by hypothesis ir1, ir2
       <2>4 qed by step <2>1, <2>2, <2>3
  <1>3 qed by step <1>1, <1>2 type bintree_t
;;
```

## 3.6 Introducing lemmas

Until now we stated properties and demonstrated them writing "all-in-one" proofs, i.e. using intermediate (hence nested) steps, hypotheses, types and functions definitions. However, depending on the complexity of the property to prove, it may be easier to define intermediate lemmas, or even involve previously demonstrated theorems. This answers a need for modularity (intermediate lemmas can be used for other proofs) and readability (intermediate lemmas can make proofs more numerous but smaller) when writing proofs.

Still addressing proofs on programs, we now want to prove that the absolute value of a difference is always …positive. The only thing is, we won't write the program computing such a value using a predefined `abs` function bringing its property stating it always returns a positive value. Instead, we write this function using a test and a subtraction:

Listing 22: lemmas.fcl

```
open "basics" ;;

let abs_diff (x, y) = if x > y then x - y else y - x ;;
```

We now state the property we want to demonstrate, and as always we initially state it as **`assumed`**:

Listing 23: lemmas.fcl (2)

```
open "basics" ;;

let abs_diff (x, y) = if x > y then x - y else y - x ;;

theorem always_pos :
  all x y : int, abs_diff (x, y) >= 0
proof =
  <1>1 assume x : int, y : int,
       prove abs_diff (x, y) >= 0
       assumed
  <1>2 conclude ;;
```

We must now elaborate the sketch of the proof to introduce intermediate steps. From the definition of our function `abs_diff`, it is clear that we must reason by cases, one if $x > y$ and one if $\sim (x > y)$, i.e. $x \leq y$. Hence, we will introduce 2 steps for these cases, and an ending one using the former to conclude the goal.

Listing 24: lemmas.fcl (3)

```
open "basics" ;;

let abs_diff (x, y) = if x > y then x - y else y - x ;;

theorem always_pos :
  all x y : int, abs_diff (x, y) >= 0
proof =
  <1>1 assume x : int, y : int,
       prove abs_diff (x, y) >= 0
       <2>1 hypothesis h1: x > y,
            prove abs_diff (x, y) >= 0
            assumed
       <2>2 hypothesis h2: x <= y,
            prove abs_diff (x, y) >= 0
            assumed
       <2>3 qed by step <2>1, <2>2
  <1>2 conclude ;;
```

In both intermediate steps `<2>1` and `<2>2` the goal is the same than the global one: we did not yet split it, changed it by any refinement. However, we introduced 2 different (and complementary) hypotheses. Having in mind that having covered cases $x > y$ and $x \leq y$ we covered all the cases of integers, we run the compiler and get:

```
Invoking ocamlc...
>> ocamlc -I /usr/local/lib/focalize -c lemmas.ml
Invoking zvtov...
>> zvtov -zenon zenon -new lemmas.zv
File "lemmas2.fcl", line 16, characters 16-34:
Zenon error: exhausted search space without finding a proof
```

Oops, it goes wrong, Zenon didn't find any proof! So what? So why ? As naively said in the above paragraph, "*Having in mind that having covered cases $x > y$ and $x \leq y$ we covered all the cases of integers*", we assume that it is obvious that 2 integers are either greater or lower-or-equal together. But, this fact is **not** obvious: Zenon does not known arithmetic! So we need to give it such a property as a fact to hope it will finally find a proof.

We are currently trying to make a proof, and now we need to prove another property. So, first we don't want to spread our effort in several directions. We need to have this other property: why not state it, not prove it yet, and check that our current proof pass with this new property ? We just need a lemma to make our proof, so we will introduce some. We then write the theorem `two_ints_are_gt_or_le` stating that $\forall x, y : int,\ x \leq y \vee x > y$ and give it as a new fact to Zenon.

Here comes a new fact (in addition to the already seen facts **`conclude`**, **`hypothesis`**, **`step`**, **`definition of`** and **`type`**: the **`property`** fact, stating that Zenon should use the given property (i.e. logical statement) to find a proof.

Listing 25: lemmas.fcl (4)

```
open "basics" ;;

let abs_diff (x, y) = if x > y then x - y else y - x ;;

theorem two_ints_are_gt_or_le: all x y : int, (x > y) \/ (x <= y)
  proof = assumed ;;

theorem always_pos :
  all x y : int, abs_diff (x, y) >= 0
proof =
  <1>1 assume x : int, y : int,
       prove abs_diff (x, y) >= 0
       <2>1 hypothesis h1: x > y,
            prove abs_diff (x, y) >= 0
            assumed
       <2>2 hypothesis h2: x <= y,
            prove abs_diff (x, y) >= 0
            assumed
       <2>3 qed by step <2>1, <2>2 property two_ints_are_gt_or_le
  <1>2 conclude ;;
```

We now compile again our development and see that with this new fact, Zenon finally succeeded.

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

Obviously, we get one more theorem to demonstrate. However, we know that provided this theorem holds, the proof of our main program property also holds. We can now go on further on it, leaving the new lemma for later.

But. . . by a wonderful coincidence, FoCaLize comes with a "standard library"! And looking among available theorems (in **basics.fcl**) we find a theorem:

```
theorem int_gt_or_le : all x y : int, (x > y) \/ (x <= y)
```

exactly fitting what we need! So, proof of our lemma will be trivial since it will simply be done **by property** `basics.int_gt_or_le`. But, we can make even simpler: in our proof, let's just use this theorem from the library instead of aliasing it by our lemma! Hence, in our proof, we change `<2>3` **qed** by adding **by** `...` `int_gt_or_le` instead of **by** `...` `two_ints_are_gt_or_le` and remove this latter from our source code.

Now, let's going on with our proof. We need to really prove the 2 steps `<2>1` and `<2>2`. For `<2>1`, we can prove that `abs_diff`$(x, y) = x - y$ and that $x - y \geq 0$. This way, we will really have proved that `abs_diff`$(x, y) \geq 0$: our sub-proof will then be conclude "by these 2 steps" as show below in step `<3>2`

Similarly, for `<2>2` we will prove something like that `abs_diff`$(x, y) = y - x$ and $y - x \geq 0$. We let this second case aside for the moment (i.e. **assumed**), only dealing with the first one.

Listing 26: lemmas.fcl (5)

```
open "basics" ;;

let abs_diff (x, y) =
  if x > y then x - y
  else y - x ;;

theorem always_pos :
  all x y : int, abs_diff (x, y) >= 0
proof =
  <1>1 assume x : int, y : int,
       prove abs_diff (x, y) >= 0
       <2>1 hypothesis h1: x > y,
            prove abs_diff (x, y) >= 0
            <3>1 prove abs_diff (x, y) = x - y
                 assumed
            <3>2 prove x - y >= 0
                 assumed
            <3>3 qed by step <3>1, <3>2
       <2>2 hypothesis h2: x <= y,
            prove abs_diff (x, y) >= 0
            assumed
       <2>3 qed by step <2>1, <2>2 property int_gt_or_le
  <1>2 conclude ;;
```

Compiling our program, we will see that the proof continues passing: our idea of sub-proofs was correct. So, we now want to really prove the new intermediate steps `<3>1` and `<3>2`. The sketch of the proof is to prove that $\texttt{abs\_diff}(x, y) = x - y$ and that $x - y \geq 0$ knowing we are in the context of hypothesis `h1` stating that $x > y$.

Lets start by step `<3>1`. We want to prove that $\texttt{abs\_diff}(x, y) = x - y$. This is a direct consequence of the definition of the function `abs_diff` since we are in the case of hypothesis `h1`. Hence, this proof is simply done **`by definition of`** `abs_diff` **`hypothesis`** `h1`.

We now address step `<3>2`. What do we have as material? We know by hypothesis `h1` that $x > y$. From this point, it looks obvious to us that in effect, $x - y \geq 0$. However, like above for the "trivial" lemma on arithmetic, it won't probably be so for Zenon. We can again introduce a new lemma, or . . . have a look to see if there would not already be a suitable theorem in the FoCaLize standard library! And hopefully, we find in **basics.fcl** the theorem:

```
theorem int_diff_ge_is_pos : all x y : int, x >= y -> x - y >= 0
```

It is nearly won, but not yet. In effect, having a deeper look at our hypothesis `h1`, we see that it states that $x > y$ although the theorem `int_diff_ge_is_pos` requires as hypothesis that $x \geq y$. However, our intuition immediately makes us thinking that if $x > y$ then it is inevitable that $x \geq y$. Again, a new lemma to introduce or a look to have in the standard library. . .

Fortunately, we again discover a theorem fitting our expectations in **basics.fcl**:

```
theorem int_gt_implies_ge : all x y : int, x > y -> x >= y
```

Note that in "real life", it will happen that the library do not already contains the theorem you need: in this case, you will really state it as a new theorem (lemma) and finally will need to prove it!

Now we found the 2 former theorems, our goal should be solved by Zenon **`by property`** `...` of them and the hypothesis `h1`.

Listing 27: lemmas.fcl (6)

```
open "basics" ;;

let abs_diff (x, y) =
  if x > y then x - y
  else y - x ;;

theorem always_pos :
  all x y : int, abs_diff (x, y) >= 0
proof =
  <1>1 assume x : int, y : int,
       prove abs_diff (x, y) >= 0
       <2>1 hypothesis h1: x > y,
            prove abs_diff (x, y) >= 0
            <3>1 prove abs_diff (x, y) = x - y
                 by definition of abs_diff hypothesis h1
            <3>2 prove x - y >= 0
                 by hypothesis h1
```

```
                property int_diff_ge_is_pos, int_gt_implies_ge
            <3>3 qed by step <3>1, <3>2
        <2>2 hypothesis h2: x <= y,
            prove abs_diff (x, y) >= 0
            assumed
        <2>3 qed by step <2>1, <2>2 property int_gt_or_le
    <1>2 conclude ;;
```

As planned, the proof is accepted and we go on, trying to prove the remaining step <2>2. We will proceed in the same way, proving that assuming hypothesis h2 we have $\texttt{abs\_diff}(x, y) = y - x$ and $y - x \geq 0$.

However, we can note that the theorem int_diff_ge_is_pos we used above states $(a \geq b) \Rightarrow (a - b \geq 0)$. But, our hypothesis h2 states that $x \leq y$ and we need to prove that $y - x \geq 0$. But in fact, in our hypothesis, if we swap $x$ and $y$ and replace $\leq$ by $\geq$ we get into the right hypothesis of the theorem. Again, we will need a theorem stating that $x \leq y \Rightarrow y \geq x$ which already exists as int_le_ge_swap. We then have one more step than in the previous case, to demonstrate that $y \geq x$ **by property** int_le_ge_swap **hypothesis** h2.

We finally show the new form of the proof, skipping the (*a priori* obvious) proof that abs_diff (x, y) = y - x. Although it seems it is only a consequence of the definition of abs_diff, we will see later that it requires something more.

Listing 28: lemmas.fcl (7)

```
open "basics" ;;

let abs_diff (x, y) = if x > y then x - y else y - x ;;

theorem always_pos :
  all x y : int, abs_diff (x, y) >= 0
proof =
  <1>1 assume x : int, y : int,
       prove abs_diff (x, y) >= 0
       <2>1 hypothesis h1: x > y,
            prove abs_diff (x, y) >= 0
            <3>1 prove abs_diff (x, y) = x - y
                 by definition of abs_diff hypothesis h1
            <3>2 prove x - y >= 0
                 by hypothesis h1
                    property int_diff_ge_is_pos, int_gt_implies_ge
            <3>3 qed by step <3>1, <3>2
       <2>2 hypothesis h2: x <= y,
            prove abs_diff (x, y) >= 0
            <3>1 prove abs_diff (x, y) = y - x
                 assumed
            <3>2 prove y >= x
                 by property int_le_ge_swap hypothesis h2
            <3>3 prove y - x >= 0
                 by step <3>2 hypothesis h2 property int_diff_ge_is_pos
            <3>4 qed by step <3>1, <3>2, <3>3
       <2>3 qed by step <2>1, <2>2 property int_gt_or_le
  <1>2 conclude ;;
```

Finally, it only remains to inspect step `<3>1`. As previously mentioned, it looks trivial that it only depends on the fact we are in hypothesis `h2`, i.e. $x \leq y$ and the definition of `abs_diff` falling in the “else-case”. Let simply make the proof with these 2 facts:

```
...
       <2>2 hypothesis h2: x <= y,
                 prove abs_diff (x, y) >= 0
            <3>1 prove abs_diff (x, y) = y - x
                      by definition of abs_diff hypothesis h2
...
```

We compile and get:

```
Invoking ocamlc...
>> ocamlc -I /usr/local/lib/focalize -c lemmas.ml
Invoking zvtov...
>> zvtov -zenon zenon -new lemmas.zv
File "lemmas.fcl", line 21, characters 17-56:
Zenon error: exhausted search space without finding a proof
### proof failed
```

Having closer look at our hypothesis `h2:x <=y` and the way `abs_diff` has its conditional written **`if`** `x > y`, we see that the **`if`** tests $x > y$, hence in the “else-case” we have $\sim (x > y)$ and not $x \leq y$ as stated in the hypothesis! In effect, in a “else-branch” the holding property is “**not**-the-tested-condition”. And again, for Zenon, it is not obvious that $\sim (x > y)$ is the same thing than $x \leq y$. Again, we need to guide Zenon with such a theorem which hopefully exists in FoCaLize standard library:

```
theorem int_le_not_gt : all x y : int, (x <= y) -> ~ (x > y)
```

At this point, adding the fact `int_le_not_gt` to the proof of our step `<3>1` will finally conclude the whole proof:

Listing 29: lemmas.fcl (8)

```
open "basics" ;;

let abs_diff (x, y) = if x > y then x - y else y - x ;;

theorem always_pos :
  all x y : int, abs_diff (x, y) >= 0
proof =
  <1>1 assume x : int, y : int,
       prove abs_diff (x, y) >= 0
       <2>1 hypothesis h1: x > y,
            prove abs_diff (x, y) >= 0
            <3>1 prove abs_diff (x, y) = x - y
                 by definition of abs_diff hypothesis h1
            <3>2 prove x - y >= 0
                 by hypothesis h1
                    property int_diff_ge_is_pos, int_gt_implies_ge
            <3>3 qed by step <3>1, <3>2
       <2>2 hypothesis h2: x <= y,
            prove abs_diff (x, y) >= 0
            <3>1 prove abs_diff (x, y) = y - x
```

```
                by definition of abs_diff hypothesis h2
                   property int_le_not_gt  (* To rewrite <= into ~ > *)
           <3>2 prove y >= x
                by property int_le_ge_swap hypothesis h2
           <3>3 prove y - x >= 0
                by step <3>2 hypothesis h2 property int_diff_ge_is_pos
           <3>4 qed by step <3>1, <3>2, <3>3
      <2>3 qed by step <2>1, <2>2 property int_gt_or_le
 <1>2 conclude ;;
```

## 4 A first simple program

After having seen how to write hierarchical proofs in FoCaLize using Zenon in the context of pretty ad-hoc properties, we will finally apply previous technics on proving properties related to a (still very simple) program.

We deliberately make no use of FoCaLize advanced modeling features like inheritance, parametrisation, incremental conception and refinement mechanisms. We only consider a raw software model, obviously not supporting evolution, but that's not the aim. More information on these points can be found in [1].

### 4.1 The goal

We want to model a simplified traffic signals controller. The system will be made of 3 signals with 3 states: green, orange and red. The controller will alternatively make each signal becoming green along a predefined sequence, making so that other signals are red. As any usual signals, they turn orange before turning red. We can then simply model the controller as a finite state automaton representing cycling sequences (where R stands for red, G for green and O for orange, representing the state of each managed traffic signal):

$$GRR \rightarrow ORR \rightarrow RGR \rightarrow ROR \rightarrow RRG \rightarrow RRO$$

### 4.2 Modeling data structures

Without surprise, to represent the color of a signal, we define a sum type with 3 values:

```
open "basics" ;;

(** Type of signals colors. *)
type color_t = | C_green | C_orange | C_red ;;
```

Obviously, the automaton having 6 states, we need to define a sum type with as many values. For readability, we name each case "S_" followed by the corresponding signals color initials. For instance `S_orr` stands for *"State where signal 1 is orange, signal 2 is red and signal 3 is red"*.

```
(** Type of states the automaton can be. Simply named with letters
     corresponding to the colors of signal 1, 2 and 3. *)
type state_t = | S_grr | S_orr | S_rgr | S_ror | S_rrg| S_rro ;;
```

Finally, the state of the controller will consists in the current state of the automaton and the state of each signal. We then embed the controller inside a species whose **`representation`** reflects this data structure.

```
(** Species embedding the automaton controlling the signals colors changes
    . *)
species Controller =
  (* Need to encode tuples as nested pairs because of limitations of  Coq
     and Zenon. *)
  representation = (state_t * (color_t * (color_t * color_t))) ;
end ;;
```

One may note that instead of defining the **`representation`** as a 4-components tuple, we nested pairs up to have 4 components. The reason is that currently FoCaLize compiler and Zenon don't yet transparently generalize pairs, hence making very difficult proofs to be compiled to Coq. However, this do not reduce the expressivity of the language: it only makes things a bit more cumbersome.

## 4.3 The main algorithm

The controller is now modeled as a transition function taking the current state of the controller as input and returning the next state. Roughly speaking, it will discriminate on the state of the automaton (first component of the **`representation`** which is the state of the controller), then determine the new state as well as the new states of the signals. Hence, the transition function `run_step` will have type **`Self ->Self`**.

Because we modelled the state of the controller as a tuple-like data structure, we first define projection functions to access individual components of the controller state (i.e. the automaton state and each signal state – color).

Listing 30: controller.fcl

```
open "basics" ;;

(** Type of signals colors. *)
type color_t = | C_green | C_orange | C_red ;;


(** Type of states the automaton can be. Simply named with letters
    corresponding to the colors of signal 1, 2 and 3. *)
type state_t = | S_grr | S_orr | S_rgr | S_ror | S_rrg| S_rro ;;


(** Species embedding the automaton controlling the signals colors
    changes. *)
species Controller =
  (* Need to encode tuples as nested pairs because of limitations of Coq
     and Zenon. *)
  representation = (state_t * (color_t * (color_t * color_t))) ;

  let init : Self = (S_grr, (C_green, (C_red, C_red))) ;

  (** Extractors of "tuples" components. *)
  let get_s (x : Self) = match x with | (a, _) -> a ;
  let get_s1 (x : Self) =
    match x with | (_, a) -> match a with | (b, _) -> b ;
  let get_s2 (x : Self) =
    match x with | (_, a) ->
      match a with | (_, b) ->
        match b with | (c, _) -> c ;
  let get_s3 (x :Self) =
    match x with | (_, a) ->
      match a with | (_, b) ->
        match b with | (_, c) -> c ;

  (** Main controller function: automaton's 1 step run. *)
  let run_step (state : Self) : Self =
    match get_s (state) with
    | S_grr -> (S_orr, (C_orange, (C_red, C_red)))
    | S_orr -> (S_rgr, (C_red, (C_green, C_red)))
    | S_rgr -> (S_ror, (C_red, (C_orange, C_red)))
    | S_ror -> (S_rrg, (C_red, (C_red, C_green)))
    | S_rrg -> (S_rro, (C_red, (C_red, C_orange)))
    | S_rro -> (S_grr, (C_green, (C_red, C_red))) ;
end ;;
```

We only defined the behavioral, computational aspects of our controller: no properties yet. However we can compile this program and get a usable piece of software.

## 4.4 Introducing the main property

It is now time to *"prove our program"*. Behind this unclear but widely used expression is hidden the task of characterizing the safety properties of a system, then prove they hold. In our very simple case, one interesting property is that we never have 2 green signals at the same time. Since we have 3 signals we will state this property as the negation of 3 disjunctions, each stating 2 of the 3 signals are green:

$$\begin{array}{ll}\sim ( & (signal_1\ is\ green \wedge signal_2\ is\ green) \vee \\ & (signal_1\ is\ green \wedge signal_3\ is\ green) \vee \\ & (signal_2\ is\ green \wedge signal_3\ is\ green) \qquad )\end{array}$$

Such a property leads to the following FoCaLize theorem, still left unproven for the moment:

```
  (** The complete theorem stating that no signals are green at the same
      time. *)
  theorem never_2_green :
    all s r : Self,
    r = run_step (s) ->
    ~ ((get_s1 (r) = C_green /\ get_s2 (r) = C_green) \/
       (get_s1 (r) = C_green /\ get_s3 (r) = C_green) \/
       (get_s2 (r) = C_green /\ get_s3 (r) = C_green))
  proof = assumed ;
```

## 4.5 Making the proof

It is now time to prove our theorem. One sketch of the proof is to prove that we never have $signal_1$ and $signal_2$ green at the same time, neither $signal_1$ and

$signal_3$ nor $signal_2$ and $signal_3$. From these 3 properties, Zenon should be able to find the remaining “glue” and prove the whole theorem!

We then just try to see if our intuition is right. We define the 3 intermediate lemmas `never_s1_s2_green`, `never_s1_s3_green` and `never_s2_s3_green` , let them unproven for the moment, and ask Zenon to prove our main theorem `never_2_green` **`by property ...`** our 3 lemmas:

Listing 31: controller.fcl (2)

```
open "basics" ;;

(** Type of signals colors. *)
type color_t = | C_green | C_orange | C_red ;;


(** Type of states the automaton can be. Simply named with letters
    corresponding to the colors of signal 1, 2 and 3. *)
type state_t = | S_grr | S_orr | S_rgr | S_ror | S_rrg| S_rro ;;


(** Species embedding the automaton controlling the signals colors
    changes. *)
species Controller =
  (* Need to encode tuples as nested pairs because of limitations of Coq
     and Zenon. *)
  representation = (state_t * (color_t * (color_t * color_t))) ;

  let init : Self = (S_grr, (C_green, (C_red, C_red))) ;

  (** Extractors of "tuples" components. *)
  let get_s (x : Self) = match x with | (a, _) -> a ;
  let get_s1 (x : Self) =
    match x with | (_, a) -> match a with | (b, _) -> b ;
  let get_s2 (x : Self) =
    match x with | (_, a) ->
      match a with | (_, b) ->
        match b with | (c, _) -> c ;
  let get_s3 (x :Self) =
    match x with | (_, a) ->
      match a with | (_, b) ->
        match b with | (_, c) -> c ;

  (** Main controller function: automaton's 1 step run. *)
  let run_step (state : Self) : Self =
    match get_s (state) with
    | S_grr -> (S_orr, (C_orange, (C_red, C_red)))
    | S_orr -> (S_rgr, (C_red, (C_green, C_red)))
    | S_rgr -> (S_ror, (C_red, (C_orange, C_red)))
    | S_ror -> (S_rrg, (C_red, (C_red, C_green)))
    | S_rrg -> (S_rro, (C_red, (C_red, C_orange)))
    | S_rro -> (S_grr, (C_green, (C_red, C_red))) ;


  (** Lemma stating that s1 and s2 are never green together. It's 1/3 of
      the final property stating that no signals are green at the same
      time. *)
  theorem never_s1_s2_green :
    all s r : Self,
    r = run_step (s) ->
    ~ (get_s1 (r) = C_green /\ get_s2 (r) = C_green)
  proof = assumed ;

  (* Same proof kind than for never_s1_s2_green. *)
  theorem never_s1_s3_green :
    all s r : Self,
    r = run_step (s) ->
    ~ (get_s1 (r) = C_green /\ get_s3 (r) = C_green)
  proof = assumed ;

  (* Same proof kind than for never_s1_s2_green. *)
  theorem never_s2_s3_green :
    all s r : Self,
    r = run_step (s) ->
    ~ (get_s2 (r) = C_green /\ get_s3 (r) = C_green)
  proof = assumed ;

  (** The complete theorem stating that no signals are green at the same
      time. *)
  theorem never_2_green :
    all s r : Self,
    r = run_step (s) ->
    ~ ((get_s1 (r) = C_green /\ get_s2 (r) = C_green) \/
       (get_s1 (r) = C_green /\ get_s3 (r) = C_green) \/
       (get_s2 (r) = C_green /\ get_s3 (r) = C_green))
  proof =
    by property never_s1_s2_green, never_s1_s3_green, never_s2_s3_green ;

end ;;
```

We invoke the compilation by the regular command: **focalizec controller.fcl**:

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

and see that our proof passed, assuming our 3 pending lemmas. It will then be time to actually prove these lemmas. One imagine easily that their proofs will be similar, since the only change between statements is the involved signals.

### 4.5.1 Proving the first lemma

We will now address proving the first lemma, namely `never_s1_s2_green` , using the incremental approach we previously introduced: setting-up the proof sketch, the main intermediate steps with their goal to prove left assumed, then refining these steps until nothing remains assumed. Obviously, our lemma won't be fully automatically proved by one Zenon step since its statement is too complex. Hence, we forget a proof of the shape:

```
theorem never_s1_s2_green :
  all s r : Self,
  r = run_step (s) ->
  ~ (get_s1 (r) = C_green /\ get_s2 (r) = C_green)
proof = by definition of ... type ... step ... hypothesis ... ;
```

and prepare us to write a hierarchical one, whose first step is the simple introduction of hypotheses of our theorem, leaving its goal to prove (i.e. currently left **assumed** ):

```
theorem never_s1_s2_green :
  all s r : Self,
  r = run_step (s) ->
  ~ (get_s1 (r) = C_green /\ get_s2 (r) = C_green)
proof =
<1>1 assume s : Self, r : Self,
     hypothesis h1 : r = run_step (s),
     prove ~ (get_s1 (r) = C_green /\ get_s2 (r) = C_green)
     assumed
<1>2 conclude ;
```

The sketch of the proof is a study by cases on the values of the automaton state, showing that in each state, the resulting state of the signals 1 and 2 is never green for both (i.e. there is never 2 `C_green` values in the 2nd and 3rd components of the controller state).

How can we prove this ? Simply by exhibiting, in each case, that the result contains at least one color not equal to `C_green`. Obviously, for each case we chose to target the signal whose value is really not green! Hence, we refine our proof and state each case of the proof, as many as there are states in the automaton, hence as many as there are cases in the transition function `run_step`. In the first case, no signal is green since $signal_1$ is orange, and $signal_2$ is red: we chose to prove that $signal_1$ is not green. Conversely, in the second case, $signal_2$ is green: we do not have the choice and must prove that $signal_1$ is not.

```
(** Lemma stating that s1 and s2 are never green together. It's 1/3 of the
    final property stating that no signals are green at the same time. *)
theorem never_s1_s2_green :
  all s r : Self,
  r = run_step (s) ->
  ~ (get_s1 (r) = C_green /\ get_s2 (r) = C_green)
proof =
<1>1 assume s : Self, r : Self,
     hypothesis h1 : r = run_step (s),
     prove ~ (get_s1 (r) = C_green /\ get_s2 (r) = C_green)

     (* Proof by cases on values of the "automaton state" of s.
        For each case, we will prove that one of the 2 signal at least is
        not green. *)
     <2>1 hypothesis h2: get_s (s) = S_grr,
          prove ~ (get_s1 (r) = C_green)
          assumed

     (* Same proof kind for all the cases of automaton state. *)
     <2>2 hypothesis h3: get_s (s) = S_orr,
          prove ~ (get_s1 (r) = C_green)
          assumed

     (* Same proof kind for all the cases of automaton state. *)
     <2>3 hypothesis h4: get_s (s) = S_rgr,
          prove ~ (get_s1 (r) = C_green)
          assumed

     <2>4 hypothesis h5: get_s (s) = S_ror,
          prove ~ (get_s1 (r) = C_green)
          assumed

     <2>5 hypothesis h6: get_s (s) = S_rrg,
          prove ~ (get_s1 (r) = C_green)
          assumed

     <2>6 hypothesis h7: get_s (s) = S_rro,
          prove ~ (get_s2 (r) = C_green)
          assumed

     <2>7 qed by
          step <2>1, <2>2, <2>3, <2>4, <2>5, <2>6
          definition of run_step
          hypothesis h1
          type state_t
<1>2 conclude ;
```

The conclusion of our proof is step <2>7 and obviously relies on the 6 preceding steps, but also on the definition of the function `run_steps`, the type `state_t` and the hypothesis `h1:r =run_step (s)`.

In effect, the intermediate steps can only be combined by Zenon, hoping to find a complete proof, if it knows that they represent all the possible cases of the function `run_steps`, knows that the type `state_t` only contains the values on which `run_steps` discriminates and finally knows that the `r` used in all steps goals is the result of calling `run_steps` (so is the resulting controller state), i.e. the hypothesis `h1`.

As usual, we can compile the source and will see that Zenon finds the proof and the whole theorem gets accepted by Coq. Removing one of the facts provided in step <2>7 really causes the whole proof to fail.

It remains now to refine our proof by removing all the **`assumed`** we set to "prove" intermediate steps <2>1 to <2>6. In each case, to prove that a signal is not green, we simply prove it has an effective other color value. From this exhibited value (obviously not being `C_green`) and the definition of the type `color_t`, Zenon can establish that – this type being an inductive definition – all its constructors are different 2 by 2. In other words, the fact that `C_red` is not equal to `C_green` requires Zenon to know the underlying type definition. For this reason, each proof requires the exhibition of the computed color (**`step`**<3>1) and the type `color_t`.

The way the proof exhibiting the effective color value (the one different from green) works is still left assumed, hence following our refinement tactic.

```
(** Lemma stating that s1 and s2 are never green together. It's 1/3 of the
    final property stating that no signals are green at the same time. *)
theorem never_s1_s2_green :
  all s r : Self,
  r = run_step (s) ->
  ~ (get_s1 (r) = C_green /\ get_s2 (r) = C_green)
proof =
<1>1 assume s : Self, r : Self,
     hypothesis h1 : r = run_step (s),
     prove ~ (get_s1 (r) = C_green /\ get_s2 (r) = C_green)
```

```
      (* Proof by cases on values of the "automaton state" of s.
         For each case, we will prove that one of the 2 signal at least is
         not green. *)
      <2>1 hypothesis h2: get_s (s) = S_grr,
           prove ~ (get_s1 (r) = C_green)
           (* To prove the signal s1 is not green, we prove it is orange.
                *)
           <3>1 prove get_s1 (r) = C_orange
                assumed
           <3>2 qed by step <3>1 type color_t

      (* Same proof kind for all the cases of automaton state. *)
      <2>2 hypothesis h3: get_s (s) = S_orr,
           prove ~ (get_s1 (r) = C_green)
           (* To prove the signal s1 is not green, we prove it is red. *)
           <3>1 prove get_s1 (r) = C_red
                assumed
           <3>2 qed by step <3>1 type color_t

      (* Same proof kind for all the cases of automaton state. *)
      <2>3 hypothesis h4: get_s (s) = S_rgr,
           prove ~ (get_s1 (r) = C_green)
           <3>1 prove get_s1 (r) = C_red
                assumed
           <3>2 qed by step <3>1 type color_t

      <2>4 hypothesis h5: get_s (s) = S_ror,
           prove ~ (get_s1 (r) = C_green)
           <3>1 prove get_s1 (r) = C_red
                assumed
           <3>2 qed by step <3>1 type color_t

      <2>5 hypothesis h6: get_s (s) = S_rrg,
           prove ~ (get_s1 (r) = C_green)
           <3>1 prove get_s1 (r) = C_red
                assumed
           <3>2 qed by step <3>1 type color_t

      <2>6 hypothesis h7: get_s (s) = S_rro,
           prove ~ (get_s2 (r) = C_green)
           <3>1 prove get_s2 (r) = C_red
                assumed
           <3>2 qed by step <3>1 type color_t

      <2>7 qed by
           step <2>1, <2>2, <2>3, <2>4, <2>5, <2>6
           definition of run_step
           hypothesis h1
           type state_t
<1>2 conclude ;
```

Finally, once the compilation shown that this new refinement passes Zenon searches and Coq assessment, it is time to complete the last holes of the proof, the last **assumed** remaining. Each such case aims at proving that the value we chose and exhibited as being different from `C_green` is really the one computed by the related call to `get_s1 (r)`. In other words, we need to demonstrate that in the case `<2>1<3>1`, we really have `get_s1 (r)=C_orange` holding (and similarly for the other cases).

One may be easily convinced that this is intrinsically due to the way the function `run_step` is written! But not only: this is also due to the way `get_s1` is written since it appears in the goal to prove. Moreover, each property holds in the context of the hypothesis representing the examined case of the pattern-matching **match** `get_s (state)` **with** of `run_step` : the hypothesis `h2` in the first case (`h3` in the second, `h4` in the third, and so on). Finally, our hypothesis deals with a value of type `state_t` and our goal with a value of type `color_t`. Hence Zenon will for sure need to know about them!

Giving Zenon all these facts, we hope it will find a proof for each case, which will really be the case. Hence our complete proof of the initial lemma is:

```
(** Lemma stating that s1 and s2 are never green together. It's 1/3 of the
    final property stating that no signals are green at the same time. *)
theorem never_s1_s2_green :
  all s r : Self,
  r = run_step (s) ->
  ~ (get_s1 (r) = C_green /\ get_s2 (r) = C_green)
proof =
<1>1 assume s : Self, r : Self,
     hypothesis h1 : r = run_step (s),
     prove ~ (get_s1 (r) = C_green /\ get_s2 (r) = C_green)

     (* Proof by cases on values of the "automaton state" of s.
        For each case, we will prove that one of the 2 signal at least is
        not green. *)
     <2>1 hypothesis h2: get_s (s) = S_grr,
          prove ~ (get_s1 (r) = C_green)
          (* To prove the signal s1 is not green we prove it is orange. *)
          <3>1 prove get_s1 (r) = C_orange
               by hypothesis h1, h2
                  definition of get_s1, run_step
                  type state_t, color_t
          <3>2 qed by step <3>1 type color_t

     (* Same proof kind for all the cases of automaton state. *)
     <2>2 hypothesis h3: get_s (s) = S_orr,
          prove ~ (get_s1 (r) = C_green)
          (* To prove the signal s1 is not green, we prove it is red. *)
          <3>1 prove get_s1 (r) = C_red
               by hypothesis h1, h3
                  definition of get_s1, run_step
                  type state_t, color_t
          <3>2 qed by step <3>1 type color_t

     (* Same proof kind for all the cases of automaton state. *)
     <2>3 hypothesis h4: get_s (s) = S_rgr,
          prove ~ (get_s1 (r) = C_green)
          <3>1 prove get_s1 (r) = C_red
               by hypothesis h1, h4
                  definition of get_s1, run_step
                  type state_t, color_t
          <3>2 qed by step <3>1 type color_t

     <2>4 hypothesis h5: get_s (s) = S_ror,
          prove ~ (get_s1 (r) = C_green)
          <3>1 prove get_s1 (r) = C_red
               by hypothesis h1, h5
                  definition of get_s1, run_step
                  type state_t, color_t
          <3>2 qed by step <3>1 type color_t

     <2>5 hypothesis h6: get_s (s) = S_rrg,
          prove ~ (get_s1 (r) = C_green)
          <3>1 prove get_s1 (r) = C_red
               by hypothesis h1, h6
                  definition of get_s1, run_step
                  type state_t, color_t
          <3>2 qed by step <3>1 type color_t

     <2>6 hypothesis h7: get_s (s) = S_rro,
          prove ~ (get_s2 (r) = C_green)
          <3>1 prove get_s2 (r) = C_red
               by hypothesis h1, h7
                  definition of get_s2, run_step
                  type state_t, color_t
          <3>2 qed by step <3>1 type color_t

     <2>7 qed by
          step <2>1, <2>2, <2>3, <2>4, <2>5, <2>6
          definition of run_step
          hypothesis h1
          type state_t
<1>2 conclude ;
```

### 4.5.2 Proving other lemmas : THE END

We initially decided to split our main safety property `never_2_green` into 3 lemmas. We proved above the first of them. All of them having an identical structure, their proofs will obviously be strongly similar. Hence, we do not detail again their proofs but provide the complete source file implementing our controller.

Listing 32: controller.fcl (3)

```
open "basics" ;;

(** Type of signals colors. *)
type color_t = | C_green | C_orange | C_red ;;


(** Type of states the automaton can be. Simply named with letters
    corresponding to the colors of signal 1, 2 and 3. *)
type state_t = | S_grr | S_orr | S_rgr | S_ror | S_rrg| S_rro ;;


(** Species embedding the automaton controlling the signals colors
    changes. *)
species Controller =
  (* Need to encode tuples as nested pairs because of limitations of Coq
     and Zenon. *)
  representation = (state_t * (color_t * (color_t * color_t))) ;
  let init : Self = (S_grr, (C_green, (C_red, C_red))) ;

  (** Extractors of "tuples" components. *)
  let get_s (x : Self) = match x with | (a, _) -> a ;
  let get_s1 (x : Self) =
    match x with | (_, a) -> match a with | (b, _) -> b ;
  let get_s2 (x : Self) =
    match x with | (_, a) ->
      match a with | (_, b) ->
        match b with | (c, _) -> c ;
  let get_s3 (x :Self) =
    match x with | (_, a) ->
      match a with | (_, b) ->
        match b with | (_, c) -> c ;


  (** Main controller function: automaton's 1 step run. *)
  let run_step (state : Self) : Self =
    match get_s (state) with
    | S_grr -> (S_orr, (C_orange, (C_red, C_red)))
    | S_orr -> (S_rgr, (C_red, (C_green, C_red)))
    | S_rgr -> (S_ror, (C_red, (C_orange, C_red)))
    | S_ror -> (S_rrg, (C_red, (C_red, C_green)))
    | S_rrg -> (S_rro, (C_red, (C_red, C_orange)))
    | S_rro -> (S_grr, (C_green, (C_red, C_red))) ;


  (** Lemma stating that s1 and s2 are never green together. It's 1/3 of
      the final property stating that no signals are green at the same
      time. *)
  theorem never_s1_s2_green :
    all s r : Self,
    r = run_step (s) ->
    ~ (get_s1 (r) = C_green /\ get_s2 (r) = C_green)
  proof =
  <1>1 assume s : Self, r : Self,
       hypothesis h1 : r = run_step (s),
       prove ~ (get_s1 (r) = C_green /\ get_s2 (r) = C_green)

       (* Proof by cases on values of the "automaton state" of s.
          For each case, we will prove that one of the 2 signal at least
          is not green. *)
       <2>1 hypothesis h2: get_s (s) = S_grr,
            prove ~ (get_s1 (r) = C_green)
            (* To prove the signal s1 is not green we prove it is orange.
                *)
            <3>1 prove get_s1 (r) = C_orange
                 by hypothesis h1, h2
                    definition of get_s1, run_step
                    type state_t, color_t
            <3>2 qed by step <3>1 type color_t

       (* Same proof kind for all the cases of automaton state. *)
       <2>2 hypothesis h3: get_s (s) = S_orr,
            prove ~ (get_s1 (r) = C_green)
            (* To prove the signal s1 is not green, we prove it is red. *)
            <3>1 prove get_s1 (r) = C_red
                 by hypothesis h1, h3
                    definition of get_s1, run_step
                    type state_t, color_t
            <3>2 qed by step <3>1 type color_t

       (* Same proof kind for all the cases of automaton state. *)
       <2>3 hypothesis h4: get_s (s) = S_rgr,
            prove ~ (get_s1 (r) = C_green)
            <3>1 prove get_s1 (r) = C_red
                 by hypothesis h1, h4
                    definition of get_s1, run_step
                    type state_t, color_t
            <3>2 qed by step <3>1 type color_t
       <2>4 hypothesis h5: get_s (s) = S_ror,
            prove ~ (get_s1 (r) = C_green)
            <3>1 prove get_s1 (r) = C_red
                 by hypothesis h1, h5
                    definition of get_s1, run_step
                    type state_t, color_t
            <3>2 qed by step <3>1 type color_t

       <2>5 hypothesis h6: get_s (s) = S_rrg,
            prove ~ (get_s1 (r) = C_green)
            <3>1 prove get_s1 (r) = C_red
                 by hypothesis h1, h6
                    definition of get_s1, run_step
                    type state_t, color_t
            <3>2 qed by step <3>1 type color_t

       <2>6 hypothesis h7: get_s (s) = S_rro,
            prove ~ (get_s2 (r) = C_green)
            <3>1 prove get_s2 (r) = C_red
                 by hypothesis h1, h7
                    definition of get_s2, run_step
                    type state_t, color_t
            <3>2 qed by step <3>1 type color_t

       <2>7 qed by
            step <2>1, <2>2, <2>3, <2>4, <2>5, <2>6
            definition of run_step
            hypothesis h1
            type state_t
  <1>2 conclude ;


  (* Same proof kind than for never_s1_s2_green. *)
  theorem never_s1_s3_green :
    all s r : Self,
    r = run_step (s) ->
    ~ (get_s1 (r) = C_green /\ get_s3 (r) = C_green)
  proof =
  <1>1 assume s : Self, r : Self,
       hypothesis h1 : r = run_step (s),
       prove ~ (get_s1 (r) = C_green /\ get_s3 (r) = C_green)

       (* Proof by cases on values of the "automaton state" of s. *)

       <2>1 hypothesis h2: get_s (s) = S_grr,
            prove ~ (get_s1 (r) = C_green)
            <3>1 prove get_s1 (r) = C_orange
                 by hypothesis h1, h2
                    definition of get_s, get_s1, run_step
                    type state_t, color_t
            <3>2 qed by step <3>1 type color_t

       <2>2 hypothesis h3: get_s (s) = S_orr,
            prove ~ (get_s1 (r) = C_green)
            <3>1 prove get_s1 (r) = C_red
                 by hypothesis h1, h3
                    definition of get_s1, run_step
                    type state_t, color_t
            <3>2 qed by step <3>1 type color_t

       <2>3 hypothesis h4: get_s (s) = S_rgr,
            prove ~ (get_s1 (r) = C_green)
            <3>1 prove get_s1 (r) = C_red
                 by hypothesis h1, h4
                    definition of get_s1, run_step
                    type state_t, color_t
            <3>2 qed by step <3>1 type color_t

       <2>4 hypothesis h5: get_s (s) = S_ror,
            prove ~ (get_s1 (r) = C_green)
            <3>1 prove get_s1 (r) = C_red
                 by hypothesis h1, h5
                    definition of get_s1, run_step
                    type state_t, color_t
            <3>2 qed by step <3>1 type color_t

       <2>5 hypothesis h6: get_s (s) = S_rrg,
            prove ~ (get_s1 (r) = C_green)
            <3>1 prove get_s1 (r) = C_red
                 by hypothesis h1, h6
                    definition of get_s1, run_step
                    type state_t, color_t
            <3>2 qed by step <3>1 type color_t

       <2>6 hypothesis h7: get_s (s) = S_rro,
            prove ~ (get_s3 (r) = C_green)
            <3>1 prove get_s3 (r) = C_red
                 by hypothesis h1, h7
                    definition of get_s3, run_step
                    type state_t, color_t
            <3>2 qed by step <3>1 type color_t

       <2>7 qed by
            step <2>1, <2>2, <2>3, <2>4, <2>5, <2>6
            definition of run_step
            hypothesis h1
            type state_t
  <1>2 conclude ;


  (* Same proof kind than for never_s1_s2_green. *)
  theorem never_s2_s3_green :
    all s r : Self,
    r = run_step (s) ->
    ~ (get_s2 (r) = C_green /\ get_s3 (r) = C_green)
  proof =
  <1>1 assume s : Self, r : Self,
       hypothesis h1 : r = run_step (s),
       prove ~ (get_s2 (r) = C_green /\ get_s3 (r) = C_green)

       (* Proof by cases on values of the "automaton state" of s. *)

       <2>1 hypothesis h2: get_s (s) = S_grr,
            prove ~ (get_s2 (r) = C_green)
            <3>1 prove get_s2 (r) = C_red
                 by hypothesis h1, h2
                    definition of get_s2, run_step
                    type state_t, color_t
            <3>2 qed by step <3>1 type color_t

       <2>2 hypothesis h3: get_s (s) = S_orr,
            prove ~ (get_s3 (r) = C_green)
            <3>1 prove get_s3 (r) = C_red
```

```
                  by hypothesis h1, h3
                     definition of get_s3, run_step
                     type state_t, color_t
             <3>2 qed by step <3>1 type color_t

        <2>3 hypothesis h4: get_s (s) = S_rgr,
             prove ~ (get_s2 (r) = C_green)
             <3>1 prove get_s2 (r) = C_orange
                  by hypothesis h1, h4
                     definition of get_s2, run_step
                     type state_t, color_t
             <3>2 qed by step <3>1 type color_t

        <2>4 hypothesis h5: get_s (s) = S_ror,
             prove ~ (get_s2 (r) = C_green)
             <3>1 prove get_s2 (r) = C_red
                  by hypothesis h1, h5
                     definition of get_s2, run_step
                     type state_t, color_t
             <3>2 qed by step <3>1 type color_t

        <2>5 hypothesis h6: get_s (s) = S_rrg,
             prove ~ (get_s2 (r) = C_green)
             <3>1 prove get_s2 (r) = C_red
                  by hypothesis h1, h6
                     definition of get_s2, run_step
                     type state_t, color_t
             <3>2 qed by step <3>1 type color_t

        <2>6 hypothesis h7: get_s (s) = S_rro,
             prove ~ (get_s2 (r) = C_green)
             <3>1 prove get_s2 (r) = C_red
                  by hypothesis h1, h7
                     definition of get_s2, run_step
                     type state_t, color_t
             <3>2 qed by step <3>1 type color_t

        <2>7 qed by
             step <2>1, <2>2, <2>3, <2>4, <2>5, <2>6
             definition of run_step
             hypothesis h1
             type state_t
   <1>2 conclude ;


  (** The complete theorem stating that no signals are green at the same
      time. *)
  theorem never_2_green :
    all s r : Self,
    r = run_step (s) ->
    ~ ((get_s1 (r) = C_green /\ get_s2 (r) = C_green) \/
       (get_s1 (r) = C_green /\ get_s3 (r) = C_green) \/
       (get_s2 (r) = C_green /\ get_s3 (r) = C_green))
  proof =
    by property never_s1_s2_green, never_s1_s3_green, never_s2_s3_green ;

end ;;
```

## 5 Conclusion

This tutorial illustrated the way proofs can be carried out in FoCaLize, using Zenon to make them easier. We addressed here development much more “algorithm-oriented” than other documents more oriented toward “mathematical-modeling”.

We didn’t used powerful modeling constructs of FoCaLize to only concentrate on hierarchical split of proofs, intermediate lemmas stating and kinds of facts available to guide Zenon in its proofs searches and in which case to use them.

## References

[1] The FoCaLize Team. *A Short Tutorial for FoCaLize: Implementing Sets*. LIP6-ENSIIE-ENSTA, 2009–2012.