FoCaLiZe FAQ

For version 0.9.1

March 2017

- **Q:** I get the error message:
`Error: The reference basics.int__t was not found in the current environment`
when Coq compiles.

- **A:** You probably forgot to open the module `basics` in you FoCaLiZe program. Add the directive `open "basics";;` at the top of your source file.

- **Q:** I get the error message:
`Error: Types Self and ... are not compatible.`
when focalizec compiles.

- **A:** You probably have created a def-dependency on the representation in the statement of a property or a theorem, like in:

```
species Bug =
  representation = int ;
  property wrong : all x : Self, x = x + 0 ;
end ;;
```

This statement reveals that the representation is indeed `int` since to have `x + 0` well-typed, `Self` must exactly be `int`. This makes the interface of the species impossible to be typed as a collection since the representation will be abstracted. You may need to add extra methods hiding the dependency on representation.

- **Q:** How to I make a function taking a tuple in argument ?

- **A:** `let f ( (x, y))=... ;;`

- **Q:** How to I make a function taking `unit` in argument ?

- **A:** `let f (_x :unit)=... ;;`

- **Q:** What is the difference between a constructor having several arguments and a constructor having one argument being a tuple ?

- **A:** A constructor with one argument being a tuple is defined using the “tupling” type constructor `*`:

```
type with_1_tuple_arg =
  | A (int * bool * string) ;;    (* Note the stars. *)
```

A valid usage of this constructor is:

```
let ok = A ( (1, false, "") ) ;;
```

where it is important to note the double parentheses. This constructor has 1 argument that is a tuple. The syntax for constructors with arguments already requires parentheses, that's the reason for these double parentheses.

Trying to use this constructor as:

```
let ko = A (1, false, "") ;;
```

would lead to an error telling that types `int * bool * string` and `int` are not compatible. In effect in this case, `A` is considered to be applied to several arguments, the first one being `1` that is of type `int`. And `int` is really incompatible with a tuple type.

A similar constructor with several separate arguments is defined using the "comma" construct:

```
type with_several_args =
  | B (int, bool, string) ;;   (* Note the comas. *)
```

A valid usage of this constructor is:

```
let ok = B (1, false, "") ;;
```

where it is important to note the unique pair of parentheses. This constructor has 3 arguments.

Trying to use this constructor as:

```
let ko = B ( (1, false, "") ) ;;
```

would lead to an error telling that types `int` and `int * bool * string` are not compatible. In effect, we try to pass to `B` one unique argument that is a tuple. And a tuple is incompatible with the first expected argument of `B`, that is `int`.

- **Q:** I get a syntax error on a sum type definition or a pattern-matching.
- **A:** Beware that conversely to OCaml, the first bar is not optional in FoCaLiZe.

Wrong

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

match x with
    Z -> ...
  | S (y) ...
```

Correct

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

match x with
  | Z -> ...
  | S (y) ...
```

- **Q:** I get the error message:

```
Zenon error: uncaught exception File "coqterm.ml", line
325, characters 6-12: Assertion failed
```

- **A:** This is a current issue in Zenon. Compile your program adding the option `-zvtovopt -script` to focalizec. This asks Zenon to output proofs as a Coq script instead of a Coq term. This should be fixed in the future.

• **Q:** I get the error message:
`Error: Types coq_builtins#prop and basics#bool are not compatible.`

• **A:** You confused the "not" operators `~` and `~~` or probably used a **`logical let`** definition in a **`let`** definition like in the following example.

```
species Bug =
  logical let not0 (x) = ~~ (x = 0) ;
  let g (y) = if not0 (y) then ... else ... ;
end ;;
```

`logical let` are definitions whose result type is `prop`, i.e. the type of logical **statement**. They are intended to be used in theorems or properties and are discarded in OCaml code since this latter doesn't deal with logical/proof aspects. The type of logical **expressions** is `bool` and is automatically transformed into `prop` in the context of logical statements of theorems or properties. However, in the context of computational definitions, `prop` is always rejected. In the above example, you may have defined the *computational* function `not0` by:

```
species Bug =
  let not0 (x) = ~ (x = 0) ;
  let g (y) = if not0 (y) then ... else ... ;
end ;;
```

where `~` is the "not" on booleans, whereas `~~` is the "not" on logical propositions.

• **Q:** I get the error message: `Zenon error: cannot infer a value for a variable of type XYZ.`

• **A:** Make sure that you are not using a theorem/property with variables not used in your goal. This sometimes arise with theorem/property "too general".

For instance, suppose a theorem `p_foo` being a conjunction of 3 cases with 3 variables used to prove a goal only using 2 of these cases (hence and of these variables) like below.

```
open "basics";;

species Dummy =
  signature foo : int -> int ;
  property p_foo: all x y z : int, foo (x) = foo (y) /\ foo (y) = foo (z) ;

  theorem junk: all a b : int, foo (a) = foo (b)
  proof = by property p_foo ;
end ;;
```

Zenon complains:

```
File "gloups.fcl", line 8, characters 10-27:
Zenon error: cannot infer a value for a variable of type basics.int__t
```

Indeed, the variable `z` of the property `p_foo` is not used but must be instantiated by a value by Zenon. Zenon does not yet have any way to find values of a type.

You may rewrite without loss of generally the property `p_foo`. Instead of quantifying all the variables at once then stating the conjunction of cases, simply put the needed quantifications inside each case of the conjunction:

```
  property p_foo:
    (all x y : int, foo (x) = foo (y)) /\
    (all y z : int, foo (y) = foo (z)) ;
```