Lecture at École Jeunes Chercheurs en Programmation (EJCP) 2014

Programming and Proving: Practice with FoCaLiZe

François Pessaux

May 13, 2015

Contents

This lecture has been prepared using FoCaLiZe version 0.8.0 "++", i.e. the snapshot of the GIT repository of March 2014. The material related to this lecture will be available online at:

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

Additionally, resources related to FoCaLiZe can be consulted from its WEB site at `http://focalize.inria.fr`. Issues and bugs (related to FoCaLiZe or Zenon) can be reported via the dedicated bug-tracker at `http://focalize.inria.fr/bugzilla/`.

This document was written also taking inspiration in several documents and presentations the FoCaLiZe team and other folks did write. I want to thank them even if I can't cite all of them.

About the author:

François Pessaux
ENSTA ParisTech - U2IS
828 boulevard des Maréchaux, 91120 Palaiseau
France

Email: `francois.pessaux@ensta-paristech•fr`
WEB Page: `http://perso.ensta-paristech.fr/~pessaux/`

1 Introduction

Any decent developer should wonder when he wrote a piece of code whether this program "works". However, even if this fact seems trivial, one may be surprised by the amount of people not even addressing this question. When it is stated, in most cases, the programmer gets convinced (possibly convinces his boss or customer) that the code "works" on the basis of nebulous intuitions based on his (also nebulous) understanding of what "works" means (we will come back later on this point). And debug is still an important part of the cost of software development.

1.1 Why Proving?

Nowadays, software engineering is faced to systems of increasing complexity, with sharpened safety and/or security requirements. The problems differ, according to the domain: military, medical, transport, energy, finances, telecoms, etc. But the objective is the same: any software component embedded in those systems has to "properly work". This goes through the respect of standards[10, 9, 4], which ask for the demonstration that the software is designed, developed, tested according to their prescriptions. Among the recommended practices, formal methods start to be advocated. Formal methods encompass number of techniques: abstract interpretation, model-checking, other static analyses, etc and formal proofs. These latter will be the topic of the present lecture.

For safety or security high levels, the demonstration is mandatory and must be done with formal methods. Moreover, this demonstration is examined by third parties experts, along what is called the assessment process. Aside this "basic industrial" point of view (which remains crucial since it aims at avoiding nuclear plants from exploding, planes from landing under the ground or banks from dividing their capital by zero), the exigence of a true demonstration is shared by numerous scientists, who want to demonstrate that indeed their program computes exactly what they wish, that a function terminates, that an inference system is sound and complete, etc.

Even if the concerns may differ in size and complexity, the question is the same: ensuring, with the highest possible certainty, that a piece of software behaves "correctly".

1.2 Proofs, Formal Proofs, Proofs of What?

One can sometimes hear "I proved my program". What does it mean ? Basically nothing. First, one can only prove that a program satisfies a given *property*: a proof is not an absolute essence of a program. This point is crucial since it means that to prove that a program "works", one needs to state its expected *properties*, and to ultimately prove that the program satisifies them.

Now, let's consider a program computing the absolute value of the difference between its arguments: $\texttt{abs\_diff}(x, y) = \texttt{ if } x > y \texttt{ then } x - y \texttt{ else } y - x$. Now, I will "prove" it…I prove the property $\forall x, x = x$. Right, since $=$ is an equivalence relation, it is symmetric, transitive, and (what we need) reflexive: QED. What did I prove? I proved a property for sure, but it has nothing related to our `abs_diff` function: it does not demonstrate that our function really computes what we expect!

From these two points, we understand that "proving" that a program "works" implies proving *properties* that are related to the *expected behavior* of the program. Such a set of properties is the *specification* of the software. In other words, one has to demonstrate that the software *implementation* fulfills the software *specification*. The specification describes in detail the requirements which must be satisfied by the developed system. Hence is a document issued (normally) *prior* to the development itself.

Returning to our example, we could have proved the more interesting fact: the result of `abs_diff` is always positive. But how to state "Is always positive"? One needs a *language* to express such a property and we also need to "process" this language to prove the property. Ensuring that proofs are mathematically rigorous needs a *logical language* and a *logic* to work with properties. For instance, one can express the previous property by $\forall x, y \in N, \texttt{abs\_diff}(x, y) \geq 0$.

The question of extracting relevant properties from the specification and express them in a given logic is not trivial and is not be the topic of this lecture. It is not rare that specifications are provided in natural language, hence with ambiguities, not even complete, even sometimes containing inconsistencies. Moreover some parts of the specifications cannot be expressed either because the chosen logical language does not fit to them or because they are *non-functional* properties (e.g. "the program must run in less than 10ms and in at most 10 Kb of stack" or "it must induce a power consumption lower than 10 mW").

Now that it is clear that proof of programs requires a specification, and a logical framework, there are still several ways to carry out proofs. The first one is by "hand" in an informal style. Considering our previous example of absolute value of a difference, we can demonstrate that is it always positive as follows.

> *Let's have the function* $\texttt{abs\_diff}(x, y) = \texttt{ if } x > y \texttt{ then } x - y \texttt{ else } y - x$.
> *We want to prove that* $\forall x, y \in N, \texttt{abs\_diff}(x, y) \geq 0$.
> *There are two cases. If $x > y$ then one computes $x - y$ and "obviously", since x is greater than y the result is positive. Otherwise, x is not greater than y, hence it is lower or equal, hence y is greater or equal to x, and "obviously" $y - x$ is positive (or null). Hence in both cases the result is positive or null. QED.*

In this style, the reader must get convinced that the proof is correct by its own reading. He must himself infer than all the cases have been taken into account. In particular, it is implicit that the `if then else` induces two cases. Moreover, the "*obviously*" used twice is a well-known property of natural numbers but is totally informal: in fact it is due to the definition of the subtraction on natural numbers and can (must?) be demonstrated. In more complex proofs, hypotheses and intermediate steps can quickly increase, making the understanding and verification pretty more risky. In such a situation, how is it possible to be confident in the correctness of the proof, hence in the reliability of the related software? On another hand, if the program is modified, the proof has to be modified accordingly and has to be read again to be verified again (with the same risks).

An answer to these problems is to write proofs that can mechanically be checked by a computer: a *formal proof*. Unfortunately (or not?) the proof still has to be done by the human. Unfortunately this requires more work, more explicit details since a machine will have to be able to check the proof. But ... at least verification can be done automatically and without risk of error. Fortunately, recent progresses in automated theorem provers allow to automatically discharge an important amount of proof obligation steps. Nonetheless, in the today state of the art, "fully proving" industrial-size programs is still out of reach for several reasons:

- huge number of lines of code,
- non-functional properties impossible to express in logic,
- complex programming constructs difficult to logically model (mutability, pointers, objects, etc.)
- ...

However, it is already possible to address more reduced parts of the software, especially the critical ones, possibly mixing usage of other formal methods, in order to increase the confidence one can have in the software.

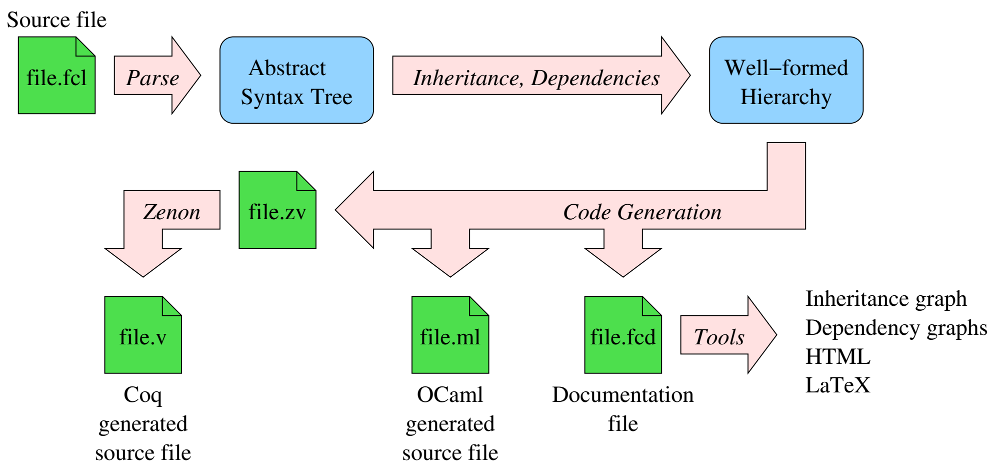
1.3 Scope of the Lecture

In this lecture we will introduce FoCaLiZe, a language allowing to write specifications, programs and proofs that the latter fulfill the former. The user may only be interested in working on specifications, checking for absence of inconsistencies or for completeness with regard to a set of specification requirements (i.e. properties). In this case, he will only work at the specification level, without providing an effective implementation.

FoCaLiZe provides a logical language to express requirements by first-order formulae depending on identifiers present in the context, a strongly typed pure functional language, powerful data-types, polymorphism and pattern-matching, object-oriented and modular features and language of proofs done by Zenon[3, 1], which may use unfolding of functions defined by pattern-matching.

A FoCaLiZe development is a source file containing properties, definitions and proofs. It must be compiled by `focalizec` which generates two outputs. The executable view of the development is an OCaml[7] source file. The logical view is a Coq[5] source file containing the complete model of the program: definitions (i.e. functions) and properties accompanied by their proofs. Proofs are obtained by invocations of the Zenon automated theorem prover which returns Coq terms that are embedded in the global generated Coq model. This model is then submitted to Coq whose role is simply to check for its consistency. In other words, Coq is an assessor of the whole development.

FoCaLiZe aims, as far as possible, at providing facilities to develop using a "programming-like" language, providing high-level constructs to allow structuring, incremental development and modularity. Moreover, its role is to hide the underlying logical target language from the user, allowing him to write proofs without mastering the specificities of a particular proof assistant. This also allows to change the target underlying languages without having the user to change his program and proofs. Finally, FoCaLiZe provides automatic documentation generation through various outputs (inheritance graph, dependency graph, interfaces, user documentation in HTML, LaTeX, extracted from the program and the comments it contains). The general compilation flow can be represented as follows:



Conversely to some other formal tools and environment like [5], [6], FoCaLiZe cannot be used in an interactive mode. The user writes is program, compiles it. Proofs he wrote are sent to Zenon which may succeed in finding what was delegated to it. In this case, compilation is a success and the resulting OCaml code is compiled, leading to an executable that can be ran. Otherwise, Zenon says a proof was not found and the user has to modify his source code to fix it before compiling it again. Proofs goals are not directed by the tool: they are explicitly stated . . . and proved by the user.

1.4 Plan of the Lecture

The remaining of this document is organized as follows. Chapter 2 introduces the basis of proofs in FoCaLiZe on first-order properties, showing both fully automated and step-by-step methods. Chapter 3 presents the usage of FoCaLiZe as a standard functional programming language allowing to write proofs about functions, without dealing with the structuring, parameterization and abstraction features of the language. The following section, 4, shows the basic FoCaLiZe concepts to structure a development, presenting a partially satisfactory architecture and abstraction, but introducing only few concepts and no proofs. Proofs are re-introduced in chapter 5 to take benefit from the structuring features. In section 6 we come back on the non satisfactory model introduced in 4 to show in two steps how to take benefit from parametrization to ensure modularity and reusability, going on with proofs and seeing the impact on their shapes. Having seen how to write specifications, programs and proofs incrementally, we shortly address in chapter 7 hints of good practices in order to minimize the amount of proof to write, giving clues to guess what to prove and at which level of the development. Finally, section 8 explains the way to "manually" decompose hard proofs, to make them fitting case analysis and, more generally, induction.

1.5 Notations

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

```
use "basics" ;;
species Sample =
```

When introduced in the running text, FoCaLiZe keywords will by typeset 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 Starting by the Beginning: First-Order Logic

Before entering in the "proof of programs", we just introduce the basics of formal proofs on the simple first order logic. This will be the occasion to remind ground notions and introduce FoCaLiZe's logical and basic proof language. We will first examine how to "manually" write proofs using Zenon, then see that this later can indeed work for us, making whole parts of basic inference steps itself.

It must be clearly understood that Zenon is not a proof checker but a **theorem prover**. Clearly, 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".

2.1 A First Tautology

The FoCaLiZe logical language provides usual typed first order formulae extended by usual functional languages *à la ML* expressions. As we will see later, these formulae can make reference to identifiers bound in the context, i.e. functions and properties names. The table below shows the usual logical connectors and their syntax in FoCaLiZe.

Semantics	Connector	FoCaLiZe syntax
Conjunction	$\wedge$	/\
Disjunction	$\vee$	\/
Implication	$\Rightarrow$	->
Equivalence	$\Leftrightarrow$	<->
Negation	$\neg$	~
Universal quantification	$\forall$	all
Existential quantification	$\exists$	ex

We first start without using complex structures of the language and state the theorem: $\forall a, b$: `boolean`, $a \Rightarrow b \Rightarrow a$. In a file **ex_implications.fcl** we write the following source code:

Listing 1: ex_implications.fcl

```
open "basics" ;;

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

This program defines a *theorem* with its statement and its proof, currently *assumed*, i.e. **not done**. In other words, this is an axiom. As we will see later, it is a very common and handy way to mark steps of a proof assumed, to check if these steps make the global proof feasible, then to incrementally remove the **`assumed`** and replace them by effective proofs. Since we are in a typed world, note the type information labeling the (universally) quantified variables a and b. The directive **`open`** allows to import the definitions of the file **basics.fcl** of the standard library, avoiding to make them explicitly qualified (for instance, using `bool` instead of `basics#bool`). This file can be compiled invoking the **focalizec** compiler with the file in argument:

focalizec ex_implications.fcl
resulting in few messages with no errors:

```
$ 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
$
```

Two source files have been generated: **ex_implications.ml** and **ex_implications.v**. The first one is empty since theorems are not leading to executable (OCaml) code while the second contains a few lines of Coq code . . . ending by a "suspicious" `apply coq_builtins.magic_prove.` (which is normal since we admitted the proof). As stated in the output messages, both generated source have been compiled by their respective compiler (`ocamlc` and `coqc`).

It is now time to write a real proof. Following the usual techniques of proofs in natural deduction, such a proof is done by inserting in the context the two hypotheses introduced by the implications, then applying exactly the hypothesis a. In other words, this proof is represented by the following sequent where hypotheses (context) are in green and goals in blue:

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

Hence, we can mimic such a proof in the FoCaLiZe Proof Language modifying our program as follows:

Listing 2: ex_implications.fcl (2)

```
1  open "basics" ;;
2
3  theorem implications :  all a b : bool, a -> (b -> a)
4  proof =
5    <1>1 assume a : bool, b : bool,
6         hypothesis h1 : a,
7         prove b -> a
8         <2>1 hypothesis h2 : b,
9              prove a
10             by hypothesis h1
11        <2>2 qed
12             by step <2>1
13   <1>2 conclude
14        (* or: qed conclude
15           or: qed by step <1>1 *) ;;
```

A **compound proof** is made of steps identified by a **bullet** (e.g. `<1>1`). A step introduces hypotheses then a **goal** following the keyword **`prove`**. It is the proposition that is asserted and proved by **this** step.

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.

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).

At the end of a step's proof, this step becomes available as a *fact* for the next steps of this proof and deeper levels' subgoals. 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`.

Let's dissect this proof. In line 5, we introduce the two variables a and b as hypotheses in the context. Since we will make reference to them, this can be seen as the way to make them bound. Line 6 exactly performs the lower $\Rightarrow$-intro of our above sequent, lifting a in hypothesis in the context under the name `h1`. Hence, our goal is now to prove `b ->a` as stated in line 7.

Now, we do not inspect the proof deeper and remain at the same nesting level. Under the hypothesis a, the step `<1>1` proves $b \rightarrow a$. Hence it is the exact global goal of our theorem. Hence, to end the proof, we just need an ending step (a **`qed`** step), `<1>2`, telling "use step `<1>1`". This is what is written in line 13 with different shortcuts. We 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.

We now return to the proof of the step `<1>1` we left aside. At this point, we had to prove $b \rightarrow a$ under the hypothesis `h1:a`. We apply the same process, mimicking the upper $\Rightarrow$-intro of our above sequent lifting b in hypothesis under the name `h2` in line 8, leaving to prove the goal a stated in line 9. This goal is exactly our hypothesis `h1`. Hence we use the fact **`by hypothesis`** `h1`. Zenon is asked to find a proof of the current goal, using hints it is provided with.

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`). Our proof is now complete. We can compile our source file again with no error.

Note that **the user** is 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.

2.2 Yet Another Tautology

In the previous example, we did all the job of the proof by hand, up to splitting implications until the goal to prove was exactly one of the hypotheses. We will now discover a little of Zenon's magic by proving the statement $\forall a, b :$ `boolean`, $(a \wedge b) \Rightarrow (a \vee b)$. The tree of this proof is very simple

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

where the implication is pushed as hypothesis by the $\Rightarrow$-intro, then we split the disjunction by the $\vee$-intro to chose to prove b and finally use the stronger hypothesis by a $\wedge$-elim. We mimic this proof tree in a source file **and_or.fcl**:

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 *) ;;
```

As previously, the step `<1>1` introduces the bound variables and the implication in the context, leaving to prove $a \vee b$ under the hypothesis `h1` stating we have $a \wedge b$. What is interesting is that the proof of the step `2<1>` is done using the hypothesis `h1` which is not *exactly* the desired goal: we did not make explicit the use of $\wedge$-elim: we rely on Zenon to find itself the proof. Would it be the begin of its magic?

2.3 *Je ne veux pas travailler…* (Pink Martini)

That was quite instructive to elaborate a proof manually from alpha to omega . . . but it was a bit boring! We have a theorem prover to help proofs, couldn't it help us more? Obviously yes (and it is satisfactory since for tautologies, automated decision procedures are well-known)! Such formulae can be proved automatically by Zenon who takes care of the "administrative" steps of basic logic. In our example, replacing each compound proof by a simple **`conclude`** step is sufficient:

Listing 4: tautos.fcl

```
open "basics" ;;

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

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

3 Basic Programming in FoCaLiZe

Since the aim is to write programs and make proofs on these pieces of code, it is time to have a look at the programming languages provided by FoCaLiZe. As previously introduced, the core language is a functional language *à la* ML with:

- basic types, like `int`, `bool`, `string`, ...
- type constructors: products, sums and records, possibly recursive,
- **`let`**-definitions, pattern-matching, **`if-then-else`**.

Definitions contained in a compilation unit (a "**.fcl**" file) can be used from other compilation units, either transparently by using the **`open`** directive seen in Section 2.1 or by explicitly prefix their name by their hosting compilation unit followed by a **#** (dash character).

In a file **loose_sets.fcl**' let's define a quick and dirty structure to represent sets, in other words, something based on lists (which are native in FoCaLiZe with the OCaml notations `::` and `[]`).

Listing 5: loose_sets.fcl

```
open "basics" ;;

type set_t ('a) =
  | Empty
  | Elem ('a, set_t ('a))
;;

let is_empty (s) = s = Empty ;;

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
;;
```

In this program, we first define a polymorphic recursive sum type with two value constructors: `Empty` with no argument and `Elem` parametrized by two arguments, one being one element of the set, the other being the remaining elements of the set (i.e. itself a set). Obviously this representation is naive since elements should not appear several times to have an efficient representation.

Then we define three functions:

- `is_empty`: it takes a set and returns the boolean value telling if this set is empty,
- `add`: it takes an element, a set and returns the set extended with this element (no verification of multiple times the same element),
- `belongs`: it take an element, a set and returns a boolean value telling if the element appears in the set. This function is defined by pattern-matching on values of the type `set_t`.

We can already notice that the third function is recursive. To ensure logical consistency, its termination has to be demonstrated. Termination proofs won't be addressed in this lecture, but we can simply see that structural recursion is simply stated by the **`structural`** keyword followed by the name of the strictly decreasing argument of the function.

In another source file, **int_loose_sets.fcl**, we can define some functions using definitions of our previous source file and prove a few theorems on sets of integers. In the following source, we intentionally not **`open`**-ed the compilation unit **loose_sets** to explicitly name qualification using the **#**-notation. However, to allow access to this unit we still at least need to add a **`use`** directive.

Listing 6: int_loose_sets.fcl

```
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) /\ 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 = by property zero_not_added ;;
```

The proof of the theorem `zero_not_added` is simple, not even made of several steps and shows a new Zenon fact: **`definition of`**. It indicates Zenon to try find a proof using the body of a definition, unfolding it. In effect, to prove that if $x = 0$ then the set returned by `add_except_0` is unchanged, we need to look at the body of this function. Note that we do not even need to consider the function `loose_sets#is_empty` even if it appears in the statement of the theorem. In effect, the proof doesn't matter what this function does, the only important point is that if $x = 0$ `add_except_0` returns the same set than it was given.

Once this theorem is proved, it may serve as lemma for proving other ones. This is what is done in the proof of the second theorem, `zero_not_added_weaker`. This latter is in fact a simple instance of the former, having directly instantiated x by 0. We see a new Zenon fact: **`property`** which indicates to use the **type** of a definition, which in case of theorems, by the Curry-Howard isomorphism, is a **logical statement**. Hence, Zenon will use the property (previously proved in this case) to prove the current one.

We can compile our program and see that Zenon completes the proofs:

```
$ 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
$
```

In order to end this first contact with "programming in FoCaLiZe like in ML and making basic proofs", we write in a file **basic_induct.fcl** a theorem telling that if we add an element to a set, then it belongs to the resulting set.

Listing 7: basic_induct.fcl

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

theorem added_forcibly_belongs: all x : int, all s : set_t (int),
  belongs (x, add (x, s))
proof = by definition of add, belongs type set_t ;;
```

This proof has to be done using the definition of `belongs`, by case on the **values of the type** `set_t`. Proof by case is a weak particular case of proof by induction and is triggered in Zenon by the new (and last) fact: **`by type`** . Such a fact tells to Zenon to invoke the induction principle of the related type to solve goals.

Later, in Section 8.2, we will come back deeper on proofs by induction to exactly understand what must be the shape of the goals allowing Zenon to succeed with induction and how to proceed when automatic proving fails.

4 Structuring the Development by Encapsulation: *Species* and *Collection*

In the previous sections, we developed programs "at toplevel", i.e. with all the definitions laying in a flat way, directly coping with implementation.

In a more realistic development, the usage is rather to express specifications and to go step by step (in an incremental approach) to design and implementation while proving that such an implementation meets its specification or design requirements.

Moreover, to reduce coupling between software components, modularity and encapsulation is widely used, grouping data-structures and operations manipulating them inside structures forming some kind of Abstract Data-Type. The structuring constructs of FoCaLiZe will allow this encapsulation, preventing users of a software component from accessing directly the internals of a data-structure. Users will hence be forced to manipulate the data-structure via the provided functions, preventing from breaking possible invariants this data-structure requires.

4.1 Components of a species

FoCaLiZe also follows this programming approach and provides a basic brick the **species**. It can be viewed as a record grouping "things" related to the same concept. Like in most modular design systems (i.e. objected oriented, algebraic abstract types), the idea is to group a data structure with the operations that operate on it. Since in FoCaLiZe we don't only address data-type and operations, these "things" also comprise the declaration (or specification) of the operations, their stated properties (which represent the requirements for the operations), and the proofs of these properties. Species are the nodes of the FoCaLiZe hierarchy. Without spoiling the coming chapters, species will be combined by inheritance to incrementally build more complex ones.

A species is a sequence of **methods** or **fields** , each one ended by a semi character (";"). Hence, a basic species looks like:

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

Species names are always capitalized. As any toplevel definition, a species ends with a double semi-character (";;"). A species can contain several kinds of methods:

- The **`representation`**, giving the data representation of the entities manipulated by the species. It is a type defined by a type expression. The **`representation`** definition may be deferred, which means that the structure of the embedded data-type does not need to be known at this point but each species has one unique **`representation`**.
- Declarations which are made of the keyword **`signature`** followed by a name and a type. They announce a method to be defined **later**. The type of the method is given but the implementation is still "omitted". Once a method is declared, this method can be used in the text following the declaration, in particular in the definition of other methods. In effect, the type provided by the signature allows the compiler to check that the method is consistently used in all the contexts with a type compatible with the declared type. Furthermore, the inheritance, late-binding and collection mechanisms introduced in Section 6.2.1, ensure that the definition of the method is known when the method is effectively invoked. Hence, signature serve for **specification** purposes since they do not bring any implementation information.
- Definitions are made of the keyword **`let`**, followed by a name, an optional type, and an expression. They serve to introduce constants or functions, i.e. computational operations. Mutually recursive definitions are introduced by **`let rec`** . Hence, definitions serve for **implementation** purposes.

A variant of definition is the **`logical let`** allowing to bind an identifier to a logical expression. For example, "being even" is a proposition, not always true, hence is not a theorem. However we can use it to build more complex logical statements which may be part of theorems statements. Hence a **`logical let`** makes possible to create parametrized logical expressions, which may hold or not (conversely to a property / theorem which has to be proved holding).

- Property statements are composed of the keyword **`property`** followed by a name and a first-order formula. Hence properties serve to express requirements, i.e. for **specification** purposes. Formulae can use methods names known within the *species*' context, especially to state requirements on these methods. Proof of properties are in fact delayed. Despite this delay, like for signatures, properties can be used to prove theorems.
- Theorems (**`theorem`**) made of a name, a statement and a proof are in fact properties packed with the formal proof that their statement holds in the context of the species. As said above, the proof of a theorem may use properties available in the context of the species, despite these former are not yet proved.
- Proofs are introduced by the keyword **`proof of`** followed by the name of a previously introduced **`property`**. They are intended to be merged with their related property to form a theorem.

4.2 Our Case of Study: Association Maps

We want to write a program providing **association maps**. As usually, such a data-structure allows to bind a **key** to a **value** and to retrieve the value bound to a given key afterwards in the map. To restrict the scope of this lecture, we won't address removing bindings, sorting and so on. Hence, an association map is a structure providing at least three operations:

- `empty :map` the initial map containing no binding,
- `add (k, v, m):key ->value ->map ->map` the function adding the binding $(k \mapsto v)$ to the map m (hiding the possibly existing previous binding of k),
- `find (k, m):key ->map ->''value or error''` the function looking for and returning the value bound to the key k in the map m if it finds it, otherwise signaling an error. Since in FoCaLiZe exceptions do not exist, we will encode the result of `find` in a type "option":

```
type option_t ('a) =
  | None           (* No available value. *)
  | Some ('a)      (* Value available, argument of the constructor. *)
```

4.3 A First Attempt: Building our First Species (Poor Structuring)

Instead of laying all the functions to manage maps et toplevel, we will group them inside a species. However, species cannot contain type definitions (except the **`representation`**) and we need to define the "option" type and the type allowing to record bindings in a map. Hence, we need to put these 2 type definitions outside the species.

For sake of simplicity, the recording structure of a map will be a list whose type is explicitly given by ourselves (we could use the built-in type of lists but this would hide some subtleties in coming proofs since the theorem prover Zenon internally knows this type). Because we are lazy, we decide that keys will be integers and values will be strings. From this we can start writing our source code:

Listing 8: (Beginning of) assoc_map1.fcl

```
open "basics" ;;

(* Structure recording bindings of a map: a hand-made basic list. *)
type int_str_list_t =
 | Nil
 | Node (int, string , int_str_list_t)    (* Keys: ints, values: strings. *)
;;

(* Return value of the lookup function: nothing or something. *)
type option_t ('a) =
 | None
 | Some ('a)
;;
```

It is now time to implement the species providing the association maps between integers and strings. Since we decided that its underlying data-structure is a list of type `int_str_list_t`, we can define its **`representation`**. Next we define the `empty` map as the empty list, the `add` function as the "cons" of the pair `(k, v)` in head of the list representing the current map and finally the lookup function as a recursive search along the list, returning `None` if no binding if found for the key `k` in the map `m`.

Listing 9: (Continuation of) assoc_map1.fcl

```
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 ;;
```

It is worthy to note that we explicitly annotated the type of `m`'s with **`Self`** to ensure that the inferred types will contain **`Self`** instead of the type expression of the representation. In effect, one submitted to the encapsulation process to make an Abstract Data-Type, the structure of the representation will be hidden, hence the methods `empty`, `add` and `find` must only reveal that they work on the abstracted type and not on its real structure.

We can compile this draft of program invoking the **focalizec** command, which should raise no errors:

```
$ 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
$
```

4.4 Final Encapsulation: Turning the Species into a Collection

At this point, we did no proof, we only encapsulated all the methods dealing with our association maps into a grouping structure, the species. Before going further in proving, we can try our implementation and turn our species into the expected Abstract Data-Type: let's create a *collection*.

A collection can be seen as an "instance" of a species where definitions get opaque: only their types remain visible. For theorems (not yet written for our species), only their statement will be visible. A collection is the only **executable proved** code. This has two major consequences:

- Since code must be executable, all the declared methods (**`signature`**, not yet used in our example) must be defined, even the representation.
- Since code must be proved, all the stated properties (**`property`**, also not yet used in our example) must be given a proof, i.e. turned into a theorem (**`theorem`**).

A species complying with these two conditions is said **complete** . In our sample code everything is defined, hence we can create a collection and add some toplevel expressions to test our software component. We also add a function printing values of type `option_t (string)`, we build a new map containing the binding `(5, ''five'')`, and print the results of looking up for the values bound to the keys 5 and 3. The complete source code is available in `code/assoc_map1_partial_test.fcl`.

Listing 10: (Continuation of) assoc_map1.fcl

```
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)) ;;
```

We then can compile our program using the **focalizec** command, without error.

```
$ 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
```

We can see the Coq verified the generated code and displays the types of trailing toplevel definitions. By default, the generated OCaml source code is compiled as an object file (to possibly link with some other software components, either coming from FoCaLiZe or manually written in OCaml). To get the running executable, we need to link the OCaml object file with a few object files of FoCaLiZe's standard library (the ones corresponding to the **`open`**-ed or **`use`**-ed libraries in the FoCaLiZe program).

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

And finally, we can run the whole stuff:

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

with the expected result. To clearly see that our software component (the collection) is now encapsulated, i.e. is a kind of Abstract Data-Type, hence can be only manipulated via the functions it provides, we can try to modify our program by trying to build a map "by hand", using its internal representation outside the species. We just change our creation of the map using the `Nil` constructor instead of the `empty` method (whose definition is however exactly `Nil`).

Listing 11: Broken program

```
...

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.
$
```

The FoCaLiZe compiler hence complains, telling that the type `int_str_list_t` whom `Nil` belongs to is not compatible with the (now abstracted) representation of our collection `MyMap`: encapsulation prevented from seeing the internals of our data-type.

5 Adding Proofs in Species

Until now, our association maps have no proved nor even stated properties. It is now time to add some.

5.1 Simple Proofs

We start by two properties that will be proved by simple facts to illustrate that the shape of proofs introduced in 3 is not impacted by the structuring added using species.

Let's first demonstrate that finding the value bound to a key just inserted in a map never fails. In other words, calling `find` with a key `k` on a map built by `add`-ing to it `k` with any bound value never returns `None`. We add this method after the definition of `find`.

```
 (* 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 = ????
```

We now need to give Zenon facts to find a proof. Clearly, this is a consequence of how `add` and `find` are implemented: the former adds in head of the list and the latter destruct this head. Hence the proof needs **`definition of`** `add, find`. However, Zenon needs to know the constructors of the type `int_str_list_t` to destruct it and those of `option_t` since they also appear in `find`'s body. Hence the proof also needs a **`type`** `int_str_list_t, option_t`. The complete theorem is then:

```
 (* 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 ;
```

Let's add another theorem, stating that looking up for two identical keys returns the same result.

```
   (* Same key -> same value. *)
  theorem find_same_key_same_value: all k1 k2: int, all m : Self,
    k1 = k2 -> find (k1, m) = find (k2, m)
  proof = conclude ;
```

Hopefully Zenon natively knows the equality **=**, its reflexive, transitive, symmetry properties... and especially the fact that any function is a morphism for this relation. Hence, Zenon is able to determine itself that calling a same function with the "same" arguments will lead to a same result. Hence the proof is reduced to **`conclude`**, telling Zenon to work itself with its internal laws and without any other required information.

Compiling the obtained FoCaLiZe code is achieved as usual and invokes Zenon which works a bit, finds proofs, and compiles the generated OCaml and Coq files without errors.

Conversely to our initial program, we now proved on it that some properties are holding with respect to our implementation of computational methods.

5.2 Proofs Getting Harder

We now try a new property, reflecting the **specification** of a decent lookup function: a bound value is found for a key if and only if either this key is the first one of the map or it is found in the remaining bindings of the map. We now state the theorem and try a simple proof. This proof is for sure a consequence of the definitions of `add`, `find` and of the types `int_str_list_t`, `option_t`.

```
  theorem find_spec: all m : Self, all s k : int, all v : string,
    (find (s, m) = Some (v) \/ s = k) <-> find (s, add (k, v, m)) = Some (v)
  proof = by definition of add, find type int_str_list_t, option_t ;
```

We now compile the program.

Attention: we still have an issue in Zenon when it generates some Coq proofs as a λ-term: the printer is missing a case:

```
Zenon error: uncaught exception File "coqterm.ml", line 335, characters 6-12: Assertion failed
File "assoc_map1.fcl", line 33, characters 10-66:
```

This has to be fixed in a coming release. The workaround is to as it to generate the proof as a Coq **script** using the extra option **-zvtovopt -script** when invoking the FoCaLiZe compiler (in the remaining of this lecture, we will use this option pretty often):

```
$ 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 *****######-
```

. . . And bad luck, Zenon goes on searching for a while, a long while. . . This simply means that the proof is too complex and must be manually split into intermediate steps!

We will now proceed incrementally: we split the current goal into several subgoals and mark them **assumed** (i.e. we don't prove them), then we add a **qed** step using the intermediate steps (and possibly other facts) to see if, with such a decomposition, Zenon now succeeds. If so, then we have to now prove each assumed intermediate step using the same method. If Zenon doesn't find any proof, this means that our split is not fine enough or we forgot some facts. In this case, it is not useful to try proving intermediate assumed steps since they will not be enough: let's not do some job for nothing.

The first step is to introduce the hypotheses of the theorem in the context and to state the remaining goal to prove. We then end by a **qed** step (**conclude**) that will trivially succeed since we only introduced hypotheses and left the body of the formula assumed. However, this is a pretty mechanical way of proceeding, so let's apply.

```
  theorem find_spec: all m : Self, all s k : int, all v : string,
    (find (s, m) = Some (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) \/ s = k) <->
               find (s, add (k, v, m)) = Some (v)
         assumed
    <1>e conclude ;
```

We can compile the program and hopefully get a success: Zenon finds a proof and both OCaml and Coq generated code are accepted.

```
$ 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
Invoking coqc...
>> coqc  -I /usr/local/lib/focalize  -I /usr/local/lib/zenon assoc_map1.v
$
```

Let's now find intermediate steps to prove the goal of step `<1>1`. We have to prove an equivalence, this means that we have two intermediate goals, each one being one side (left/right) of the equivalence under the hypothesis of the other side. Hence, we write these two intermediate steps, mark them assumed and ensure, by a **qed** step using these two intermediate steps that Zenon succeeds:

```
  theorem find_spec: all m : Self, all s k : int, all v : string,
    (find (s, m) = Some (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) \/ s = k) <->
               find (s, add (k, v, m)) = Some (v)
         <2>1 hypothesis H1: find (s, m) = Some (v) \/ 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) \/ s = k
              assumed
         <2>e qed by step <2>1, <2>2
    <1>e conclude ;
```

And . . .

```
$ 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
Invoking coqc...
>> coqc  -I /usr/local/lib/focalize  -I /usr/local/lib/zenon assoc_map1.v
$
```

. . . it works! We refined our proof in two steps that are **simpler** and that suffice to prove the above goal. It now remains to address the effective proof of these assumed steps. Let's start by the first one and try to prove the goal of `<2>1`. We can hope we will be lucky by telling Zenon that it should find a proof using the definitions of `find`, `add`, the hypothesis `H1` and the type `int_str_list_t`:

```
  theorem find_spec: all m : Self, all s k : int, all v : string,
    (find (s, m) = Some (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) \/ s = k) <->
               find (s, add (k, v, m)) = Some (v)
         <2>1 hypothesis H1: find (s, m) = Some (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) \/ s = k
              assumed
         <2>e qed by step <2>1, <2>2
    <1>e conclude ;
```

Unfortunately, compiling the program shows that Zenon gets unable to find a proof, searching for a (too) long time : we have to split again! How to prove that `find (s, add (k, v, m))=Some (v)` knowing by `H1` that `find (s, m)=Some (v)\/s =k`? May be we can first prove that add (k, v, m)=Node (k, v, m) and from this, using the definition of find, the hypothesis H1 and the type int_str_list_t (since the proof must make a case analysis on it in the body of find), the proof should be feasible. Hence, we introduce two new steps, the first one, <3>1 still left assumed, and the conclusion one, <3>e, trying to conclude the goal.

```
theorem find_spec: all m : Self, all s k : int, all v : string,
  (find (s, m) = Some (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) \/ s = k) <->
             find (s, add (k, v, m)) = Some (v)
       <2>1 hypothesis H1: find (s, m) = Some (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) \/ s = k
            assumed
       <2>e qed by step <2>1, <2>2
  <1>e conclude ;
```

We can compile the program and see that indeed Zenon found the proof of <3>e that concludes the surrounding goal of <2>1! Again, we have split a proof "too complex for Zenon" into simpler steps allowing it to find a proof. It now remains to provide an effective proof for the step <3>1, and we decently guess that it can be done using the definition of add and the type int_str_list_t. In effect, it suffices to unfold the definition of add and to know that Node is a constructor of the type int_str_list_t:

```
theorem find_spec: all m : Self, all s k : int, all v : string,
  (find (s, m) = Some (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) \/ s = k) <->
             find (s, add (k, v, m)) = Some (v)
       <2>1 hypothesis H1: find (s, m) = Some (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) \/ s = k
            assumed
       <2>e qed by step <2>1, <2>2
  <1>e conclude ;
```

We can again compile our program and see it is a success. There only remains to apply the same methodology to prove the left aside step <2>2, trying some hints (**by definition**, **type**, **step**, etc.) and introducing intermediate steps if Zenon fails finding a proof. The path is exactly similar to what we did until now and the proof of <2>2 will be in fact the same that for <2>1 in our case:

```
theorem find_spec: all m : Self, all s k : int, all v : string,
  (find (s, m) = Some (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) \/ s = k) <->
             find (s, add (k, v, m)) = Some (v)
       <2>1 hypothesis H1: find (s, m) = Some (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) \/ 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 ;
```

6 More Abstraction: Parameterization

In the previous example, we chose to implement association maps between integers and strings. However such a structure is naturally polymorphic: it behaves the same way whatever the key and value types are. The only thing one needs is to be able to test the equality between keys: a kind of "equal" function on the data-type representing the keys. Moreover, using basic types does not provide any properties on them (except some low-level toplevel theorems of the FoCaLiZe standard library we won't investigate here). Only species and collections are the way to get Abstract Data-Type with (proved) properties. Hence, to get more abstract, less specialized association maps, we want to parameterize them by **collection parameters** representing the data-types of keys and values. We hence will achieve a kind of polymorphism of species, a way to make a species depending on other ones, relying on their functions and properties.

6.1 A Second Attempt: Adding (not Enough) Collection Parameters

As introduced, an association map depends on the data-type of its keys and of its values. It has hence two parameters. We said we just need to be able to **test the equality** of keys to implement the function `find`. However, probably we will need to also compare values for other functions or to state some theorems about maps. So, we can decide that both keys and values will be "at least" of the same data-type, i.e. a **collection** whose signature is a "subclass" of this same data-type.

6.1.1 Abstracting the Maps Keys/Values

Let's define the corresponding species and call it `Comparable`. It has to contain an `eq` **signature** taking two parameters of type **`Self`** and returning a boolean. We do not need for it to be **defined** yet. Next, what are the basic characteristics of an equality function? It has to be an equivalence relation, hence must be reflexive, symmetric an transitive. Hence these must be declared as **properties** of this species.

Listing 12: (Beginning of) assoc_map2.fcl

```
open "basics" ;;

species Comparable =
  signature eq : Self -> Self -> bool ;
  property eq_reflexive: all x : Self, eq (x, x) ;
  property eq_symmetric: all x y : Self, eq (x, y) -> eq (y, x) ;
  property eq_transitive:
    all x y z : Self, eq (x, y) -> eq (y, z) -> eq (x, z) ;
end ;;
```

Note that this species is not yet **complete**, hence we can't make yet a collection from as we did in 4.4. But this is not our concern now. We will simply use it as an **interface** for the coming **collection** parameters, that will ultimately instantiated by **effective** parameters issued from collections (i.e. with everything **defined**) implementing the interface of Comparable.

6.1.2 Adapting the Species

We can start our species AssocMap, still keeping the type option_t and making our previous type of lists now polymorphic (definition of pair_list_t instead of the previous int_str_list_t).

Listing 13: (Continuation of) assoc_map2.fcl

```
 (* Structure recording bindings of a map: a hand-made basic list. *)
type pair_list_t ('a, 'b) =
  | Nil
  | Node ('a, 'b , pair_list_t ('a, 'b))
;;

 (* Return value of the lookup function: nothing or something. *)
type option_t ('a) =
  | None
  | Some ('a)
;;

species AssocMap (Key is Comparable, Value is Comparable) =
  representation = int_str_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) =
    match m with
    | Nil -> None
    | Node (kcur, v, q) -> if Key!eq (kcur, k) then Some (v) else find (k, q)
  termination proof = structural m ;
end ;;
```

The important change in the method find is that we replaced the = operator by the eq method provided by the collection parameter Key.

6.1.3 Re-Introducing a First Proof

This program perfectly compiles but does not include the proofs we wrote in its previous version. We will now re-introduce them and check if differences occur. We start by the theorem find_added_not_fails and leave both the statement and the proof unchanged up to the name of the type of our lists:

Listing 14: (Continuation of) assoc_map2.fcl

```
  (* Add make find a success. *)
  theorem find_added_not_fails: all k : Key, all v : Value, all m1 m2 : Self,
    m2 = add (k, v, m1) -> ~ (find (k, m2) = None)
  proof = by definition of add, find type pair_list_t, option_t ;
```

We now compile the program:

```
$ focalizec assoc_map2.fcl
Invoking ocamlc...
>> ocamlc -I /usr/local/lib/focalize -c assoc_map2.ml
Invoking zvtov...
>> zvtov -zenon zenon -new  assoc_map2.zv
>> zvtov -zenon zenon -new  -script assoc_map2.zv
42 ##************-
```

...and $#!@, the proof does not pass anymore: Zenon does not stop searching! The question is what did we change? The answer is "the definition of find" in which we use now Key!equal instead of **=**. Indeed, it is not so surprising that the proof fails since in it, we didn't state anything about the method `Key!eq` appearing in the body of `find` and implying that **a key is equal to another**. And furthermore, this method is only declared so how could Zenon guess? The fact that the `k` appearing in `add (k, v, m1)` and in `find (k, m2)` is the same is clear in the statement of the theorem by syntactic equality, however, Zenon cannot guess that it also means that `Key!eq (k, k)`! To solve this, we need to use the properties we stated for the `Key!eq`, i.e. that it is an equivalence relation, hence symmetric, transitive and reflexive.

For this proof, the reflexivity will be sufficient. How did we determine this? The answer is "by guessing what Zenon can encounter as subgoals during its search". This is a bit informal and surely tricky, but basically, in such a case, because we clearly understand that we need to tell Zenon that `eq` is an equality-like function, we first add the three properties as facts for the proof. If the proof succeeds, we try removing some of them to check if Zenon really needs them. Hence, we complete our old proof by just adding the fact **`property`** `Key!eq_reflexive`:

Listing 15: (Continuation of) assoc_map2.fcl

```
  (* Add make find a success. *)
  theorem find_added_not_fails: all k : Key, all v : Value, all m1 m2 : Self,
    m2 = add (k, v, m1) -> ~ (find (k, m2) = None)
  proof = by definition of add, find type pair_list_t, option_t
          property Key!eq_reflexive ;
```

```
$ focalizec assoc_map2.fcl
Invoking ocamlc...
>> ocamlc -I /usr/local/lib/focalize -c assoc_map2.ml
Invoking zvtov...
>> zvtov -zenon zenon -new  assoc_map2.zv
>> zvtov -zenon zenon -new  -script assoc_map2.zv
>> zvtov -zenon zenon -new  -script assoc_map2.zv
Invoking coqc...
>> coqc  -I /usr/local/lib/focalize  -I /usr/local/lib/zenon assoc_map2.v
$
```

and it finally works!

6.1.4 The Proof that Kills

Since we adapted our species very easily, with minor changes due to the gain of abstraction provided by parameterization, we want now to go on and continue the adaptation of our initial version of association maps implementation. We will re-introduce the second theorem, `find_same_key_same_value`. Obviously, the first thing is to hope we are lucky and that it can be directly copy-pasted modulo the use of the method `Key!eq` in place of **=**, and (by intuition) the use of its three properties (of equivalence relation) instead of a simple **`conclude`**...

Listing 16: (Continuation of) assoc_map2.fcl

```
  (* Same key -> same value. *)
  theorem find_same_key_same_value: all k1 k2: Key, all m : Self,
    Key!eq (k1, k2) -> find (k1, m) = find (k2, m)
  proof = by property Key!eq_reflexive, Key!eq_transitive, Key!eq_symmetric ;
```

Let's compile:

```
$ focalizec -zvtovopt -script assoc_map2.fcl
Invoking ocamlc...
>> ocamlc -I /usr/local/lib/focalize -c assoc_map2.ml
Invoking zvtov...
>> zvtov -zenon zenon -new  -script assoc_map2.zv
File "assoc_map2.fcl", line 48, characters 10-75:
```

```
Zenon error: exhausted search space without finding a proof
### proof failed
$
```

. . . and bad luck again, Zenon fails, telling it tried **all the possible** combinations using the given facts without finding any proof. We are missing something, but what? To understand we need to try splitting the proof as usual. Hence we make two steps, `<1>1` introducing the hypotheses in the context and leaving assumed the remaining of the goal, and `<1>e` simply concluding by the previous step.

Listing 17: (Continuation of) assoc_map2.fcl

```
  theorem find_spec: all m : Self, all s k : int, all v : string,
    (find (s, m) = Some (v) \/ s = k) <-> find (s, add (k, v, m)) = Some (v)
  proof =
    <1>1 assume k1 k2: Key,
         assume m : Self,
         hypothesis H: Key!eq (k1, k2),
         prove find (k1, m) = find (k2, m)
         assumed
    <1>e conclude ;
```

This obviously works since we only lifted the hypotheses leaving the core of our property assumed. We now try to add intermediate steps. We know by the hypothesis `H` that *"k1 equals k2"*, so if we prove that `k1 =k2` this should be fine because "obviously applying a same function twice with equal arguments leads to the same result" (c.f. remark in 5.1)! So, let's write down this proof:

Listing 18: (Continuation of) assoc_map2.fcl

```
  theorem find_spec: all m : Self, all s k : int, all v : string,
    (find (s, m) = Some (v) \/ s = k) <-> find (s, add (k, v, m)) = Some (v)
  proof =
    <1>1 assume k1 k2: Key,
         assume m : Self,
         hypothesis H: Key!eq (k1, k2),
         prove find (k1, m) = find (k2, m)
         <2>1 prove k1 = k2
              assumed
         <2>e qed by step <2>1
    <1>e conclude ;
```

We can now compile our program and see that it is accepted. It only remains now to prove our last goal, the one of `<2>1` left assumed. The argument is that since by hypothesis `H` with have `Key!eq (k1, k2)` and `Key!eq` is our "equality test function", an equivalence relation, i.e. symmetric, transitive and reflexive, may be it would be sufficient . . .

Listing 19: (End of) assoc_map2.fcl

```
  theorem find_same_key_same_value: all k1 k2: Key, all m : Self,
    Key!eq (k1, k2) -> find (k1, m) = find (k2, m)
  proof =
    <1>1 assume k1 k2: Key,
         assume m : Self,
         hypothesis H: Key!eq (k1, k2),
         prove find (k1, m) = find (k2, m)
         <2>1 prove k1 = k2
              by property Key!eq_reflexive, Key!eq_transitive, Key!eq_symmetric
                 hypothesis H
         <2>e qed by step <2>1
    <1>e conclude ;
```

Trying to compile the program. . .

```
$ focalizec -zvtovopt -script assoc_map2.fcl
Invoking ocamlc...
>> ocamlc -I /usr/local/lib/focalize -c assoc_map2.ml
```

```
Invoking zvtov...
>> zvtov -zenon zenon -new  -script assoc_map2.zv
File "assoc_map2.fcl", line 54, character 14, line 55, character 29:
Zenon error: exhausted search space without finding a proof
### proof failed
$
```

. . . the proof search again fails after having tried everything. In other words, we can't prove that `Key!eq (k1, k2)` means that `k1 =k2`. And this is pretty normal! The operator **=** stands for the equality relation, known and built-in in Zenon.

In addition to the three properties we stated about `Key!eq (k1, k2)`, the equality natively has the property that $\forall f, x = y \Rightarrow f(x) = f(x)$ (in term of category, all functions are morphisms), which is exactly the required property for `<1>1` (and in fact the theorem in its whole). And our method `Key!eq (k1, k2)` does not have this property, and if we decided to add it, we would never be able to prove it, hence to create collections implementing the species `Comparable`! In effect, this is not a first-order formula and anyway, it is an axiom of the equality!

The only solution is to explicitly prove that each method (if needed) is compatible with "our equality". And indeed, the present theorem especially represents such a property for the method `find`, except it still mixes the syntactic equality **=** (on the right side of the statement) and an "abstract" equality `Key!eq` (on the left side). Such a mix may cause serious difficulties in proofs and we will see in the next section how to avoid such caveats. So, let's give up with this incorrect implementation.

6.2 A Third Attempt: Inheritance and Correct Modeling

In the previous attempts of association maps implementation, we first did no abstraction with no parameters (hence we had a screwed implementation of maps between integers and strings), second we parametrized by the same collection for both keys and values, leaving the optional aspect of lookup result screwed with the `option_t` type. We saw that both implementations were not satisfactory, either for reusability or provability of properties.

6.2.1 "Species-fying" Optional Values

Basically, the process of abstraction on keys and values we did allowed to introduce a kind of polymorphism on "proved data-types". Why not applying this up to the result of lookup? The `option_t` we used does not provide any properties. Moreover, possible properties of values it embeds are not usable by Zenon since it will not destruct automatically values of type `option_t` to apply properties.

The result of lookup is a partial type and can be also encapsulated in a species with two "values" `some` and `none`, an equality function `eq` with its own properties of equivalence relation. This species will then be parametrized by a collection being the data-type it wraps in `some`. Such a collection parameter must be "compared" in our implementation of maps, hence has to be of interface `Comparable` we already defined.

But, we also need to be able to compare "optional values" in our implementation of maps! Hence, a species `OptComparable` also has to be a `Comparable`! Inheritance is exactly the required mechanism: the species `OptComparable` we will define will be parametrized by a `Comparable` and will **inherit** from `Comparable`. We can now start our program, still keeping the toplevel type definitions `pair_list_t` and `option_t` since they will serve as effective **representations** of our species.

Listing 20: (Beginning of) assoc_map3.fcl

```
(* **** No changes here. **** *)
open "basics" ;;

type pair_list_t ('a, 'b) =
 | Nil
 | Node ('a, 'b , pair_list_t ('a, 'b))
;;

type option_t ('a) =
 | None
 | Some ('a)
;;


species Comparable =
  signature eq : Self -> Self -> bool ;
  property eq_reflexive: all x : Self, eq (x, x) ;
  property eq_symmetric: all x y : Self, eq (x, y) -> eq (y, x) ;
  property eq_transitive:
    all x y z : Self, eq (x, y) -> eq (y, z) -> eq (x, z) ;
end ;;

(* **** Changes start here. **** *)
species OptComparable (C is Comparable) =
  inherit Comparable ;
  representation = option_t (C) ;
  let eq (ox : Self, oy : Self) =
    match ox with
    | None ->
      begin
      match oy with
      | None -> true
      | Some (_) -> false
      end
    | Some (x) ->
      begin
      match oy with
      | None -> false
      | Some (y) -> C!eq (x, y)
      end ;
(* ... To be continued ... *)
end ;;
```

This species uses the type `option_t` as effective implementation of its representation and defines an `eq` method allowing to test the equality between two values. This method naturally makes a case analysis on the constructors of the values and, in case of equality, it relies on the `eq` method of the parameter to test equality between the two embedded values. Note that despite FoCaLiZe allows nested patterns in pattern-matching, we do not use them since Zenon does not handle them yet. Hence, instead we explicitly nest two **`match`** constructs which makes the code a bit bigger but does not change the expressivity of the language.

Since `OptComparable` "is a" `Comparable`, it inherits the three (unproved) properties `eq_reflexive` , `eq_symmetric` and `eq_transitive`. Assuming these properties from `Comparable`, we here can prove them for `OptComparable` since they are trivial consequences of those of `Comparable`, applied on the definition of `eq`, knowing the type `option_t`. Hence, we can continue our species by adding **`proof of`**-s the three **properties** after the method `eq`, hence turning them into **theorems**.

Listing 21: (Continuation of) assoc_map3.fcl

```
  (* ... *)
  proof of eq_reflexive =
    by definition of eq property C!eq_reflexive type option_t ;

  proof of eq_symmetric =
    by definition of eq property C!eq_symmetric type option_t ;

  proof of eq_transitive =
    by definition of eq property C!eq_transitive type option_t ;
```

Finally, we end the species adding the two foresaid methods none and some, explicitly annotating them with the type **Self** to prevent the typechecker from inferring its effective implementation. In addition, we state an extra property we will use later and that was indeed implicit when we were directly using sum type constructors (by the notion of sum type itself) instead of "abstract" values: they are two-by-two different.

Listing 22: (Continuation of) assoc_map3.fcl

```
  (* ... *)
  let none : Self = None ;
  let some (v : C) : Self = Some (v) ;

  property none_different_some: all v : C, ~ eq (some (v), none) ;
end ;;      (* End of the species already written in the first listing. *)
```

6.2.2 Adapting the Species AssocMap

This species must now also be parametrized by "optional values", i.e. must have a collection parameter having the interface OptComparable. Such a collection is obtained by applying an effective parameter to the species which is just the collection parameter representing values. Hence our species now has three collection parameters, the third one built using the second one. The representation, and the methods empty and add do not change.

Listing 23: (Continuation of) assoc_map3.fcl

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

  let empty : Self = Nil ;

  let add (k, v, m : Self) : Self = Node (k, v, m) ;

  (* ... To be continued ... *)
end ;;
```

We must now adapt the method find to make it using the abstract optional values parameterizing our species. This simply involves using the methods of the collection parameter OptValue instead of the constructors of the type option_t and the syntactic equality **=**.

Listing 24: (Continuation of) assoc_map3.fcl

```
  (* ... Inside the species AssocMap ... *)
  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 ;

  (* ... To be continued ... *)
```

6.2.3 Re-Introducing Proofs

Like we did for our previous attempt, we "only" have to adapt proofs to the new structure of the species. We will only concentrate on the proof of the first theorem, find_added_not_fails. The second one requires an explicit proof by induction. This advanced topic will be addressed in section 8.2 with a simpler example first.

The theorem find_added_not_fails proved in 6.1.3 was using the syntactic equality and the constructor None. We now must replace them by OptValue!eq and OptValue!none.

Listing 25: (Continuation of) assoc_map3.fcl

```
(* ... Inside the species AssocMap ... *)
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 = ???

(* ... To be continued ... *)
```

Before, its proof was rather simple, with a unique step listing the facts to use, i.e: definitions of `find` and `add`, types `pair_list_t` and `option_t`, property `Key!eq_reflexive`. We can (naively?) hope that by only adding similar facts for `OptValue` and removing the type `option_t` it will work. . .

Listing 26: (Continuation of) assoc_map3.fcl

```
(* ... Inside the species AssocMap ... *)
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 = by definition of add, find type pair_list_t
        property Key!eq_reflexive, OptValue!eq_reflexive,
                 OptValue!eq_symmetric, OptValue!eq_transitive ;

(* ... To be continued ... *)
```

Trying to compile the program. . .

```
$ focalizec assoc_map3fcl
Invoking ocamlc...
>> ocamlc -I /usr/local/lib/focalize -c assoc_map3.ml
Invoking zvtov...
>> zvtov -zenon zenon -new  assoc_map3.zv
77 ##*****|
```

. . . Not so simple, we have again to work. Hence, we follow the same way than usual, introducing intermediate assumed steps and gradually proving them. Before starting the proof, let's guess its sketch. We want to prove the negation of an equality. Clearly, this proof does not involve "iterations" of the method `find`. Only the fact that `add` added a binding is sufficient. Hence, no induction is required. So, to prove this negation of equality, a solution is to prove . . . the equality with something different. And to prove an equality, nothing better than exhibiting the witness, i.e. the unique result both parts evaluate to. In other words, to prove that both sides of the equality evaluate to `OptValue!some (v)`.

We first start by introducing the hypotheses in the context, leaving the body of the theorem to prove:

Listing 27: (Continuation of) assoc_map3.fcl

```
(* Add make find a success. *)
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 =
  <1>1 assume k : Key,
       assume v : Value,
       assume m1 m2 : Self,
       hypothesis H1: m2 = add (k, v, m1),
       prove ~ OptValue!eq (find (k, m2), OptValue!none)
       assumed
  <1>e conclude ;
```

This obviously pass to Zenon since the core of the proof is left assumed. It is now time to prove that `eq` does not return `OptValue!none` by proving that it returns `OptValue!some (v)`. We introduce this step, `<2>1` and a **qed** step to conclude using this former:

Listing 28: (Continuation of) assoc_map3.fcl

```
(* Add make find a success. *)
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 =
  <1>1 assume k : Key,
       assume v : Value,
       assume m1 m2 : Self,
       hypothesis H1: m2 = add (k, v, m1),
       prove ~ OptValue!eq (find (k, m2), OptValue!none)
       <2>1 prove OptValue!eq (find (k, m2), OptValue!some (v))
            assumed
       <2>e qed by step <2>1
  <1>e conclude ;
```

Before going further, let's check that this sketch is sufficient to make Zenon finding the proof:

```
$ focalizec assoc_map3fcl
Invoking ocamlc...
>> ocamlc -I /usr/local/lib/focalize -c assoc_map3.ml
Invoking zvtov...
>> zvtov -zenon zenon -new  assoc_map3.zv
File "assoc_map3.fcl", line 85, characters 18-30:
Zenon error: exhausted search space without finding a proof
### proof failed
$
```

The proof failed, Zenon having explored all the possibilities given by the provided facts. Immediately we remember that equality sometimes requires the use of properties of reflexivity, symmetry and transitivity. Trust me or not, even adding them for the parameter `OptValue` to the **qed** step does not solve the problem and triggers the same error message! This means that there is something, fact(s) missing in this proof.

Indeed, we "proved" (still assumed) that `find (...)` "equals" `OptValue!none`. This `OptValue !none` is not a literal, is not a constructor. It is only a method. How can Zenon know that `OptValue !none` is **not** the same thing than `OptValue!some`? It cannot! This is the missing fact! You may now guess why we added a property `none_different_some` in the species `OptValue` in 6.2.1. Hence, let's add a step `<2>2` and complete the **qed** step `<2>e` with `<2>2` and (found by intuition) the equivalence relation properties `eq_symmetric`, `eq_transitive` of `OptValue`.

Listing 29: (Continuation of) assoc_map3.fcl

```
  (* Add make find a success. *)
  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 =
    <1>1 assume k : Key,
         assume v : Value,
         assume m1 m2 : Self,
         hypothesis H1: m2 = add (k, v, m1),
         prove ~ OptValue!eq (find (k, m2), OptValue!none)
         <2>1 prove OptValue!eq (find (k, m2), OptValue!some (v))
              assumed
         <2>2 prove ~ OptValue!eq (OptValue!none, OptValue!some (v))
              assumed
         <2>e qed by step <2>1, <2>2
              property OptValue!eq_symmetric, OptValue!eq_transitive
    <1>e conclude ;
```

This proof now passes to Zenon. You may note a slight issue when the obtained code is compiled by Coq:

```
...
>> coqc  -I /usr/local/lib/focalize  -I /usr/local/lib/zenon assoc_map3.v
File "./assoc_map3.v", line 80, characters 59-60:
Error: Cannot infer this placeholder.
$
```

We know, we still have a problem in the term generated by Zenon where a type annotation is currently missing in a few cases. The generated source file compiles fine if slightly modified by hand, replacing an implicit argument by the “right” type. In fact, at some point in the generated Coq code, we put a “hole” telling Coq “guess what to put there”, but Coq’s inference does not succeed in finding what type to put. However, this does not impact the global consistency of the model and generated code and will have to be fixed in a next release.

It is time to replace the assumed parts by effective proofs. We will start by the second (and simplest) step: `<2>2`. Literally, we have to prove that `OptValue!none` is not “equal” to `OptValue!some (v)`. The property `OptValue!none_different_some` states that `OptValue!some (v)` is not “equal” to `OptValue!none`, i.e. the right thing but in the opposite order. Only using this property would not be sufficient since Zenon must be able to reverse both parts of the equality. This is however exactly the property of symmetry! Hence the proof simply uses these two facts:

Listing 30: (Continuation of) assoc_map3.fcl

```
(* Add make find a success. *)
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 =
  <1>1 assume k : Key,
       assume v : Value,
       assume m1 m2 : Self,
       hypothesis H1: m2 = add (k, v, m1),
       prove ~ OptValue!eq (find (k, m2), OptValue!none)
       <2>1 prove OptValue!eq (find (k, m2), OptValue!some (v))
            assumed
       <2>2 prove ~ OptValue!eq (OptValue!none, OptValue!some (v))
            by property OptValue!none_different_some,
                        (* Eh yes, in none_different_some the statement is in
                           the inverse order of our goal. *)
                        OptValue!eq_symmetric
       <2>e qed by step <2>1, <2>2
            property OptValue!eq_symmetric, OptValue!eq_transitive
  <1>e conclude ;
```

We now address the step `<2>1`, that is, `find` returns a `some`. Again, trust me or not, but trying to prove it by simply enumerating the definitions of `add`, `find`, of the two used types and other properties of `OptValue!eq` fails. We need to refine the proof in intermediate steps.

Why does the result is `some`? Simply because by hypothesis `H1` we added a binding for `h` and `find` is called with this `h`. So we have `Key!eq (k, k)` which is the condition for having `find` returning our `some`. So, we got the proof! First prove that `Key!eq (k, k)`, then prove that `OptValue!eq ( find (k, add (k, v, m1)), OptValue!some (v))` (implied by the previous step among other things) and finally conclude by the hypothesis `H1` and the previous step:

Listing 31: (Continuation of) assoc_map3.fcl

```
(* Add make find a success. *)
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 =
  <1>1 assume k : Key,
       assume v : Value,
       assume m1 m2 : Self,
       hypothesis H1: m2 = add (k, v, m1),
       prove ~ OptValue!eq (find (k, m2), OptValue!none)
       <2>1 prove OptValue!eq (find (k, m2), OptValue!some (v))
            <3>0 prove Key!eq (k, k)
                 assumed
            <3>1 prove
                   OptValue!eq (find (k, add (k, v, m1)), OptValue!some (v))
                 assumed
            <3>e qed by step <3>1 hypothesis H1
```

```
        <2>2 prove ~ OptValue!eq (OptValue!none, OptValue!some (v))
             by property OptValue!none_different_some,
                         (* Eh yes, in none_different_some the statement is in
                            the inverse order of our goal. *)
                         OptValue!eq_symmetric
        <2>e qed by step <2>1, <2>2
             property OptValue!eq_symmetric, OptValue!eq_transitive
   <1>e conclude ;
```

With such a sketch Zenon succeeds in finding a proof. It only remains to prove the steps `<3>0` and `<3>1`. The first one is the obvious consequence of the property `Key!eq_reflexive`. The second one seems to be the consequence of the definitions of `add`, `find`, `pair_list_t` and the fact proved in `<3>0`, i.e. `Key!eq (k, k)`. In fact, only using these facts Zenon fails with all the possibilities explored: we also need the property `OptValue!eq_reflexive`:

Listing 32: (Continuation of) assoc_map3.fcl

```
  (* Add make find a success. *)
  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 =
    <1>1 assume k : Key,
         assume v : Value,
         assume m1 m2 : Self,
         hypothesis H1: m2 = add (k, v, m1),
         prove ~ OptValue!eq (find (k, m2), OptValue!none)
         <2>1 prove OptValue!eq (find (k, m2), OptValue!some (v))
              <3>0 prove Key!eq (k, k) by property Key!eq_reflexive
              <3>1 prove
                     OptValue!eq (find (k, add (k, v, m1)), OptValue!some (v))
                   by definition of find, add type pair_list_t
                      property OptValue!eq_reflexive step <3>0
              <3>e qed by step <3>1 hypothesis H1
         <2>2 prove ~ OptValue!eq (OptValue!none, OptValue!some (v))
              by property OptValue!none_different_some,
                          (* Eh yes, in none_different_some the statement is in
                             the inverse order of our goal. *)
                          OptValue!eq_symmetric
         <2>e qed by step <2>1, <2>2
              property OptValue!eq_symmetric, OptValue!eq_transitive
    <1>e conclude ;
```

And with this…

```
$ focalizec -zvtovopt -script assoc_map3.fcl
Invoking ocamlc...
>> ocamlc -I /usr/local/lib/focalize -c assoc_map3.ml
Invoking zvtov...
>> zvtov -zenon zenon -new  -script assoc_map3.zv
87 ####/
```

…it does not stop searching…Time to refine the proof! Why does `find (k, add (k, v, m1))` returns `some`? Because `add` adds a node `Node` to the map and because `find` returns `some` on such a node assuming that the `k` is the same in both calls (which is especially what we proved in step `<3>0`!). Let's try to add such an intermediate step `<4>1` and its related **qed** step `<4>e`:

Listing 33: (Continuation of) assoc_map3.fcl

```
  (* Add make find a success. *)
  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 =
    <1>1 assume k : Key,
         assume v : Value,
         assume m1 m2 : Self,
         hypothesis H1: m2 = add (k, v, m1),
         prove ~ OptValue!eq (find (k, m2), OptValue!none)
         <2>1 prove OptValue!eq (find (k, m2), OptValue!some (v))
              <3>0 prove Key!eq (k, k) by property Key!eq_reflexive
              <3>1 prove
                     OptValue!eq (find (k, add (k, v, m1)), OptValue!some (v))
                     <4>1 prove
                       OptValue!eq (find (k, Node (k, v, m1)),
                                    OptValue!some (v))
                       by definition of find type pair_list_t
                          property OptValue!eq_reflexive step <3>0
                     <4>e qed by step <4>1 type pair_list_t definition of add
              <3>e qed by step <3>1 hypothesis H1
         <2>2 prove ~ OptValue!eq (OptValue!none, OptValue!some (v))
              by property OptValue!none_different_some,
                          (* Eh yes, in none_different_some the statement is in
                             the inverse order of our goal. *)
                          OptValue!eq_symmetric
         <2>e qed by step <2>1, <2>2
                  property OptValue!eq_symmetric, OptValue!eq_transitive
    <1>e conclude ;
```

Here we are, the proof passes to Zenon and is finally complete. No more steps remain assumed. We see in the last proof that just introducing one intermediate step (and obviously its **qed** step) may unlock the situation.

7 When to Prove What?

Until now, we wrote proofs without wondering if they were done at the right level in the inheritance hierarchy, and we even stated properties without wondering if their statements were abstract enough to represent a decent **specification** of a system or, conversely, were to close from a particular implementation, preventing reusing proved results for different implementations.

In effect, writing properties whose proofs depends on the **definitions** of some methods make them very fragile. If the method is redefined by inheritance, then the proof is **invalidated** and must be **redone** (the compiler tells the user about and ensures that the proof is discarded).

May be some more abstract properties could have been stated instead, not depending on a particular implementation and however still representing the expected behavior of the methods they depict. In other words, **specification** properties would have been preferred.

7.1 When to prove?

We will now examine some considerations helping to prove properties “just in time” to reduce dependencies and the amount of work to do again when reusing components by inheritance. In [8], from which this section is heavily written (nearly copy-pasted?) two important questions are identified to answer this question:

- The first one is to find a set of properties so that the redefinition of the specified function invalidates as few proofs as possible.
- The second one is to decide at which level of the inheritance hierarchy a proof has to be done.

Very often the typical development process of a component or a library of components starts by the elaboration of the hierarchy of species with the computational methods attached to each of them, and the specifications of these functions. The proofs are usually done after the code has been tested. Even if this does not respect the “best practices” recommended in standards or even any software engineering course, this allows to check CPU-time or memory needed but also avoids trying to prove erroneous implementations. However, proofs that only depend on specifications can be done before any test and may already identify inconsistencies in the future system.

7.1.1 Expressing Specifications

Before doing any proof, the first thing to do is to express the properties that have to be verified by the methods of a species. Writing such properties may by helped following the suggested guidelines:

- Even if we do not yet formally write the proof, we should have in mind what are its dependencies, i.e. stuff required to do the proof: in other words, which `by definition of` and which `by property` we will need. We refer to these two different needs as **def-dependencies** (for "def-inition") and **decl-depedencies** (for "decl-aration").
- There should be as few theorems having def-dependencies as possible.
- Theorems with def-dependencies have to be simple, in order to minimize the work to be redone in case of redefinition.
- A given theorem should have at most one def-dependency. Otherwise, it might indicate that a function has been under-specified and that we have to rely on its definition instead of its specification.

7.2 When Should we do the Proofs?

Indeed, it is quite difficult to have a unique answer to determine the "right moment" when to do a proof. If we want to prove a property p in a given hierarchy of species, we must take into account two things:

- The framework of each species of the hierarchy. In particular, we have to know which **functions** are **defined** and which **properties** are **available** in a given species.
- The global inheritance graph. Here, the number of child(ren) of each species can be very important. Indeed, if this number is big, it may be a good choice to do proofs as soon as possible, while if this number is rather small, it may be better to delay the proofs.

By inspecting the local context of each species, we can find the first species S which is refined enough in order to prove the theorem (i.e. which has the material required to make the proof).

However, it does not mean that we *prove* p in this species S: if p def-depends upon a method x and x is redefined in every child of S, then the proof given in S will never be used in any implementation. In this case, it may be better to delay the proof of p, even if it could be done already in S. This avoids writing an unused proof.

On the other hand, if the definition of x found in S persists until a collection is implemented, then p might be proved in S.

7.2.1 How the Compiler may Help?

When the hierarchy of structures is large, it becomes a pain to manually track inheritance steps in order to find out where a particular proof has to be done. To face this problem, some tools are provided by the **focalizec** compiler in order to trace dependencies. They take advantage of the analyzes performed by the compiler to show some information that may help the user in making his choice.

First, a warning is issued each time a proof of a property p is erased due to the redefinition of a function x. In addition, the name of the species S_1 where p is proved as well as the name of the species S_2 where x is redefined are reported. If this warning is issued too many times for the same property p and function x, it may indicate that the proof of p came too early in the inheritance graph and that S_2 might be a better choice.

Second, **focalizec** can tell the user that in a species S, all the functions involved in the statement of a property p are defined. Actually, some experiences showed that most of the time, if a proof def-depends upon x, then x appears in the statement of the theorem. This is especially the case when all the functions come with a suitable set of specifying properties (or theorems for the prototype-base approach). Thus this warning may be a sign that the species S is a good place to prove p.

Aside these architecture considerations, instead of addressing afterwards, once the code is written, where to prove what, another solution is to structure a development according to a strict development cycle, compliant with standards: a V-cycle. This approach was proposed in [2] but goes beyond the scope of this lecture.

8 Advanced Proofs

This section will present some typical proof schemes and how they can be done using Zenon. We address cases where the simple enumeration of facts is not sufficient and where the structure of the proof has to be made more explicit with **particular shapes** to make Zenon able to "complete the administrative steps" of the proof.

8.1 Proofs by Cases

When dealing with properties related to a non-recursive sum type, it is pretty usual to have to prove the property on each value (case) of the sum type. This is well-known as a "proof by cases". On simple properties, Zenon behaves transparently as long as it is provided (in addition to possible other facts) a **`by type`**. For instance, the following simple example is automatically handled by Zenon.

Listing 34: case_simple.fcl

```
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 ;;
```

However, in more complex cases, it may be needed to manually make the steps of case analysis explicit. In the following example, we simply define an identity function `f` on a sum type nesting another one. For sake of simplicity, we use toplevel definitions instead of a hierarchy of species, but this absolutely changes nothing to the coming proofs shapes. After the definition of this function, we write a theorem stating that this function is indeed the simple identity. Since we are optimistic, we make the proof leaving Zenon handling it in an automatic way.

Listing 35: answer_bad.fcl

```
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 ;;
```

We then compile this program...

```
$ 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
$
```

and we understand that we have to work a bit more. A proof by cases (and more generally as we will see in 8.2) of a property P on a sum type `t` requires Zenon to know the type (i.e. to be given a fact **`by type ...`** but also **to have a goal of the shape**: **`all`** `x :t, P(x)` with the **`all`** `x :t` **explicitly** present in the goal! Hence, we can start writing our proof again. The global goal (i.e. the theorem statement) starts by a **`all`** `:answer_t` and we especially want to reason by cases on this type. So, it's a perfect shape, and we do not introduce `x` as hypothesis otherwise we would break the "good" shape!

We directly introduce the three subgoals relate to the three constructors and leave their proofs assumed. Then we conclude using these three steps, the definition of `f` and the two types `answer_t` and `flag_t`. The first one is especially used to apply the reasoning by cases, the second one is needed since its constructors appear in the goal.

Listing 36: (Beginning of) answer.fcl

```
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 =
  <1>1 prove f (Yes) = Yes assumed
  <1>2 prove f (No) = No assumed
  <1>3 prove all y : flag_t, f (Maybe (y)) = Maybe (y) assumed
  <1>e qed by step <1>1, <1>2, <1>3 type answer_t, flag_t definition of f
;;
```

We then compile and see that the proof is found by Zenon. It now remains to prove the three assumed intermediate steps. The two first ones are simple consequences of the definition of `f` and type `answer_t`. In effect, it suffices to see that `f` patter-matches on its argument and returns the same value than the pattern-matching case.

Listing 37: (Modifications of) answer.fcl

```
(* Prove that f is indeed the identity. *)
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) assumed
  <1>e qed by step <1>1, <1>2, <1>3 type answer_t, flag_t definition of f
;;
```

The third and last one is more complex and requires... a proof by cases on values of the type `flag_t` ! Luckily, the goal we stated for `<1>3` has the good shape for this kind of proof. Hence we can hope Zenon will succeed using the definition of `f` and the two types whose constructors appear in the goal.

Listing 38: (Final modifications of) answer.fcl

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

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

. . . and it finally works. Note that if we didn't state the goal `<1>3` with a **`all`** `y :flag_t`, the case analysis would not have been successfully applied by Zenon. For instance, we could have instead stated `<1>3` like:

Listing 39: (Wrong modifications of) answer.fcl

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

which has intuitively the same meaning. But in term of automated proof search, provers have heuristics and decision procedures requiring particular inputs. And such a shape is not the good one for Zenon. . .

8.2 "Manual" Induction

In the same spirit than for proofs by case, proofs by induction on very simple properties are transparently handled by Zenon as shortly mentioned in 3.

Indeed, case analysis is only a special case of induction! Hence, we guess that the shape of goals required for Zenon to apply the induction principle is the same: **`all`** `x :t, P(x)`. However, when Zenon does not succeeds automatically in finding such a proof with direct facts, one must split the proof into intermediate steps having particular shapes. The following rules are the key for "manual" proofs by induction:

1. The form of the goal to prove a property P on a type `induct_t` by induction must be **`all`** `x :induct_t, P (x)`.
2. For each constructor C_i of the type `induct_t` one must prove that $P(C_i)$ holds in a sub-step.
3. If a constructor C_i is recursive, then on 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.
4. The last sub-step must be a **`qed`** step using the above steps and the type `induct_t` (and other things if needed).

8.2.1 Simple Example

To illustrate these rules, we start with a ad-hoc but simple program that Zenon does however not handle automatically. We define a dummy binary tree containing booleans and a (also dummy) function `f` always returning **`false`** when it reaches a leaf. Then at each node, `f` makes a “logical and” of the node’s value and both results of recursive calls. This function obviously always returns. . . **`false`**! And that’s what we want to prove in the theorem `always_false`.

Listing 40: stupid_tree_ko.fcl

```
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 ;;
```

The shape of the goal is correctly a **`all`** `t :bintree_t, P (t)` with `bintree_t` correctly being the inductive type on which we want to reason by induction on the argument `t`. And. . .

```
$ 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
$
```

. . . despite the good conditions, the proof fails. Hence, we need to introduce the intermediate steps. Since the type `bintree_t` has two constructors, we will have two intermediate steps in which one must prove our property, plus one ending step applying the induction principle to conclude.

The first step, `<1>1` has to prove the property for the only base case of the type, i.e. the constructor `Leaf`.

The second step, `<1>2` has to prove the property for the only inductive case of the type, i.e. the constructor `Node`, obviously using induction hypotheses.

And the last step, `<1>e`, must conclude using the type `bintree_t` and the two above steps by applying Zenon’s knowledge of the induction principle induced by the type `bintree_t`.

For the moment we leave intermediate step assumed except the concluding one since we especially ant to see that the split we did is suitable for induction by the concluding step. Hence the proof becomes:

Listing 41: (Progress of) stupid_tree_ok.fcl

```
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
;;
```

In the above code, it is important to note that **assume**-d variables l, b and r have been introduceb in **the same order** than their related parameter in the constructor Node in the definition of the type bintree_t, and the induction hypotheses related to each recursive argument are introduced immediately after their related variables. In other words, the constructor Node being Node (bintree_t, bool, bintree_t), we need to introduce a l :bintree_t, then its induction hypothesis HRecL , then a b :bool, then a r :bintree_t and finally its induction hypothesis HRecR.

Hence there is an interleaved sequence of **assume** and **hypothesis** following the constructor's definition. Each induction hypothesis states that the property holds on (the smaller) its related variable: HRecL states it for the constructor argument l, and HRecL for r.

It is now time to really prove the assumed parts. The first one is clearly a consequence of the definition of f and the type bintree_t (just look to see that f returns **false** if its argument is Leaf). The second one is the inductive case. Indeed, since by induction hypotheses f returns **false** when called on both l and r, making a "logical and" between these returned values and the current node's value will obviously give **false** (independently from the node's boolean value). Hence, the proof is done using the induction hypotheses, f and bintree_t.

Listing 42: (Completed) stupid_tree_ok.fcl

```
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
;;
```

Reading the above proof sketch, you may notice that, with just the fact that one of the recursive call inductively returns **false**, it is not even needed to use both induction hypotheses: just one is sufficient to trigger **false** with a "logical and". It's right, and indeed, it is possible to remove either HRecL or HRecR from the facts **by hypothesis** given to Zenon in step <1>2! The proof obviously pass (and is a bit shorter).

8.2.2 Back to Association Maps

The modifications we incrementally did in the previous sections (5.1) on our model of association maps left, in the third version, some proofs not re-written, especially the theorem find_same_key_same_value which has to be proved by induction. With the parameterization and the use of custom equality functions instead of the =, the theorem must now be written as follows (with an attempt of simple proof giving about all what we know as facts):

Listing 43: (Continuation of) assoc_map3.fcl

```
  (* Same key -> same value. *)
  theorem find_same_key_same_value: all k1 k2: Key, all m : Self,
    Key!eq (k1, k2) -> OptValue!eq (find (k1, m), find (k2, m))
  proof = by definition of find type pair_list_t property OptValue!eq_reflexive,
         OptValue!eq_transitive, OptValue!eq_symmetric, Key!eq_reflexive,
         Key!eq_transitive, Key!eq_symmetric ;
```

We can compile this new version of our file `assoc_map3.fcl` ...

```
$ focalizec -zvtovopt -script assoc_map3.fcl
Invoking ocamlc...
>> ocamlc -I /usr/local/lib/focalize -c assoc_map3.ml
Invoking zvtov...
>> zvtov -zenon zenon -new  -script assoc_map3.zv
106 ####*****/
```

... and quickly see that the automated search of Zenon does not succeed. Hence, following the same principles and rules stated in the previous section, we will manually split the proof, introducing intermediate steps to prove this theorem ... by induction.

First, since we want to have a goal of the form **all** `x :...,  P(x)` of the simplest shape, we remark that `k1` and `k2` are not involved in the induction: they do not change, the induction will not be applied on them. Hence, as a first step, we remove them from the property to prove (by induction) by introducing them in the context. This is simply to make our work easier. Hence, we also add a **qed** step concluding trivially (indeed, we did not change lot of things in our proof problem):

Listing 44: (Continuation of) assoc_map3.fcl

```
 (* Same key -> same value. *)
theorem find_same_key_same_value: all k1 k2: Key, all m : Self,
  Key!eq (k1, k2) -> OptValue!eq (find (k1, m), find (k2, m))
proof =
  <1>1 assume k1 k2: Key,
       prove
         all m : pair_list_t (Key, Value)
           Key!eq (k1, k2) -> OptValue!eq (find (k1, m), find (k2, m))
       assumed
  <1>e conclude ;
```

We now must address the assumed proof of the step `<1>1` to really do some job. We will prove that `Key!eq (k1, k2)->OptValue!eq (find (k1, m), find (k2, m))` for all association map `m` by induction on this latter. In other words, we have to prove it for `m` being the `Nil` map and for `m` being a map of the form `Node (..., ..., ...)`. Hence, we introduce two steps plus an ending one.

Listing 45: (Continuation of) assoc_map3.fcl

```
 (* Same key -> same value. *)
theorem find_same_key_same_value: all k1 k2: Key, all m : Self,
  Key!eq (k1, k2) -> OptValue!eq (find (k1, m), find (k2, m))
proof =
  <1>1 assume k1 k2: Key,
       prove
         all m : pair_list_t (Key, Value)
           Key!eq (k1, k2) -> OptValue!eq (find (k1, m), find (k2, m))
       <2>1 prove
              Key!eq (k1, k2) -> OptValue!eq (find (k1, Nil), find (k2, Nil))
              assumed
       <2>2 assume kcur : Key,
            assume v : Value,
            assume q : Self ,
            hypothesis HRec:
              Key!eq (k1, k2) -> OptValue!eq (find (k1, q), find (k2, q)),
            prove
              Key!eq (k1, k2) ->
                OptValue!eq
                  (find (k1, Node (kcur, v, q)), find (k2, Node (kcur, v, q)))
            assumed
       <2>e qed by step <2>1, <2>2 type pair_list_t
  <1>e conclude ;
```

In the above proof, the step `<2>1` is the only base case of the inductive type `pair_list_t`. Its statement is clearly the one of `<1>1` having replaced the **all** `m :pair_list_t (Key, Value)` (which is important for having Zenon applying the induction principle!) by the value `Nil` of the type on which we make the induction.

The step `<2>2` is the induction step, introducing each variable corresponding to the constructor `Node`'s parameters, in the same order than in the definition of this type, and with for each (indeed only one here) parameter introducing recursion, its related induction hypothesis (**immediately** after the parameter).

The ending step, `<2>e` simply triggers Zenon to invoke the induction principle using the type and the two previous steps. With these two steps, still assumed, Zenon finds a proof since we indeed proved our property for base and recursive cases of our type `pair_list_t`.

Continuing the proof is usual work like we previously did, providing the two assumed steps with effective proofs (which do not need anymore induction here). The remaining work for this proof is a bit long and we won't develop it here: it does not present any new concept. The step `<2>1` is the simplest since it is straight handled by Zenon using facts **`property`** `OptValue!eq_reflexive`, **`definition of`** `find` and **`type`** `pair_list_t`. The step `<2>2` is really longer. It first need to introduce the hypothesis (of the goal) that `Key!eq (k1, k2)`, leaving to prove the remaining of the statement.

Listing 46: (Continuation of) assoc_map3.fcl

```
 (* Same key -> same value. *)
theorem find_same_key_same_value: all k1 k2: Key, all m : Self,
  Key!eq (k1, k2) -> OptValue!eq (find (k1, m), find (k2, m))
proof =
  <1>1 assume k1 k2: Key,
       prove
         all m : pair_list_t (Key, Value) (* And NOT Self ! *) ,
           Key!eq (k1, k2) -> OptValue!eq (find (k1, m), find (k2, m))
       <2>1 prove
              Key!eq (k1, k2) -> OptValue!eq (find (k1, Nil), find (k2, Nil))
              by property OptValue!eq_reflexive definition of find
              type pair_list_t
       <2>2 assume kcur : Key,
            assume v : Value,
            assume q : Self ,
            hypothesis HRec:
              Key!eq (k1, k2) -> OptValue!eq (find (k1, q), find (k2, q)),
            prove
              Key!eq (k1, k2) ->
                OptValue!eq
                  (find (k1, Node (kcur, v, q)), find (k2, Node (kcur, v, q)))
            <3>1 hypothesis H1: Key!eq (k1, k2),
                 prove
                   OptValue!eq
                     (find (k1, Node (kcur, v, q)),
                      find (k2, Node (kcur, v, q)))
                 assumed
            <3>e conclude
       <2>e qed by step <2>1, <2>2 type pair_list_t
  <1>e conclude ;
```

Then the "remaining of the statement", i.e. the goal of step `<3>1` has to be provided with an effective proof. The sketch of this proof is to consider the two possible cases: either `kcur` "equals" `k1`, or `kcur` "does not equal" `k1`. This hence introduces three new deeper steps, `<4>1` and `<4>2`, each introducing their related hypothesis, and the concluding **`qed`** step using only the two former.

Listing 47: (Continuation of) assoc_map3.fcl

```
(* ........ *)
            <3>1 hypothesis H1: Key!eq (k1, k2),
                 prove
                   OptValue!eq
                     (find (k1, Node (kcur, v, q)),
                  find (k2, Node (kcur, v, q)))
              <4>1 hypothesis H2: Key!eq (kcur, k1),
                   prove OptValue!eq (find (k1, Node (kcur, v, q)),
                                      find (k2, Node (kcur, v, q)))
                   assumed
              <4>2 hypothesis H3: ~ Key!eq (kcur, k1),
                   prove OptValue!eq (find (k1, Node (kcur, v, q)),
                                      find (k2, Node (kcur, v, q)))
                   assumed
              <4>e qed by step <4>1, <4>2
(* ........ *)
```

Next, step `<4>1` needs to be proved. The idea to prove the "equality" of `find (k1, Node (kcur, v, q))` and `find (k2, Node (kcur, v, q))` is to prove that they lead to **a same value** `OptValue!some (v)`, i.e **exhibiting the witness** of "equality". So, we first prove that the "first" call, `find (k1, Node (kcur, v, q))`, returns `OptValue!some (v)`. Then we prove that indeed `kcur` "equals" `k2`. From this step clearly we can prove that `find (k2, Node (kcur , v, q))` also returns `OptValue!some (v)`. So, with the two steps showing that both calls return `OptValue!some (v)` we can conclude by `<4>e` ...but we need some of the administrative properties of our "equals" (reflexivity and transitivity). Hence, four deeper steps appear to continue the proof.

Listing 48: (Continuation of) assoc_map3.fcl

```
(* ........ *)
              <4>1 hypothesis H2: Key!eq (kcur, k1),
                   prove OptValue!eq (find (k1, Node (kcur, v, q)),
                                      find (k2, Node (kcur, v, q)))
                   <5>1 prove OptValue!eq (find (k1, Node (kcur, v, q)),
                                           OptValue!some (v))
                        assumed
                   <5>2 prove Key!eq (kcur, k2)
                        assumed
                   <5>3 prove OptValue!eq (find (k2, Node (kcur, v, q)),
                                           OptValue!some (v))
                        assumed
                   <5>e qed by step <5>1, <5>3
                        assumed
(* ........ *)
```

The process continues to prove `<5>1` to `<5>e` (and hopefully, no more nested steps are needed). Then, there will be remaining the pending step `<4>2` to prove, with a similar structure. We leave the complete proof as exercise (or lecture in the accompanying source code) since no new skills would be added.

9 Further Concerns

In the present lecture, we addressed the way to write specifications, programs and proofs. We started from first-order logic, introduced basic programming constructs close to the ones usually found in functional languages, then we dealt with the FoCaLiZe structuring, modularity and abstraction features while going on writing proofs. Finally we explored advanced proofs shapes.

We left aside proofs of termination of recursive functions. We only quite silently used the proof **`termination proof =structural on ...`** for each of our recursive functions. Indeed, all of those we wrote were clearly structural, which is very easily handled in FoCaLiZe (and also in Coq). We did not even addressed what has to be proved to demonstrate that a recursive function terminates.

Recent developments, still experimental, in the FoCaLiZe compiler are close to provide the user ways to prove terminations that are not structural (by an order or a measure for instance). This goes beyond the scope of this lecture but it must be known that such proofs kinds also exist in logic.

All along this lecture, we addressed first-order logic properties. No higher-order was introduced. The reason is that Zenon does not support higher-order reasoning. Hence, it would not be possible to write proofs using the FoCaLiZe Proof Language, i.e. it would not be possible to write them "easily". The only solution remaining in such cases is to write proofs directly in Coq, which is much more complicated for at least three reasons. First, the user must be aware of the transforms done by the FoCaLiZe compiler. Second, he will not be helped by Zenon to manage the "administrative glue" of the proofs (e.g. prove P_2 having $P_1 \wedge P_2 \wedge P_3$ which is trivial but need a little of steps). Third, the user has to (well) know Coq, and in general the logical target language toward which the compiler generates code.

References

[1] `http://zenon-prover.org/`.

[2] P. Ayrault, T. Hardin, and F. o. Pessaux. Development of a generic voter under focal. In *TAP'09*, volume 5608 of *LNCS*, pages 10–26. Springer-Verlag, 2009.

[3] R. Bonichon, D. Delahaye, and D. Doligez. Zenon : an extensible automated theorem prover producing checkable proofs. In N. Dershowitz and A. Voronkov, editors, *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, pages 151–165. Springer, Oct. 2007.

[4] Common Criteria. *Common Criteria for Information Technology Security Evaluation, Norme ISO 15408 – Version 3.0 Rev 2*, 2005.

[5] The Coq development team. *The Coq proof assistant reference manual.* LogiCal Project, 2012. Version 8.4.

[6] The Isabelle development team. *The Isabelle/Isar reference manual*, 2013.

[7] The OCaml development team. The objective caml reference manual. `http://caml.inria.fr/`.

[8] V. Prevosto and M. Jaume. Making proofs in a hierarchy of mathematical structures. In *Proceedings of the 11th Calculemus Symposium*, Sept. 2003.

[9] Standard Cenelec EN 50128. *Railway Applications - Communications, Signaling and Processing Systems - Software for Railway Control and Protection Systems*, 1999.

[10] Standard IEC-61508, International Electrotechnical Commission. *Functional safety of electrical/-electronic/programmable electronic safety-related systems*, 1998.

Index