A Short Tutorial for FoCaLize: Implementing Sets

The FoCaLize Team

July 2009

Contents

1 Forewords

1.1 Content

This document is a short tutorial for FoCaLize, version 0.2.0 released in July 2009[1]. It describes a full but simple development using the main features of the FoCaLize language, from the specification to the implementation, including proofs of correction.

The reader is expected to have some basic knowledge of the OCaml language, of object-oriented programming as well as an understanding of the standard logical operators ($\forall$, $\Rightarrow$, etc.) and notions about proofs.

This tutorial is not intended to be complete with regard to the FoCaLize features, nor does it recommend a specific design philosophy. In particular, It does not make use of the FoCaLize library (providing numerous mathematical structures) but on the contrary builds from scratch such a structure, for the sake of illustration.

[1] FoCaLize can be downloaded from *http://focalize.inria.fr*.

1.2 Notations and Recommandations

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

```
use "basics" ;;
species Superset =
```

This type of code has to be inserted in Unix ASCII files with a *fcl* extension. FoCaLize keywords, types or identifiers can also be inserted directly in the text, for example `Superset`.

The FoCaLize syntax is case sensitive, uppercase and lowercase identifier being associated to different sorts of entities. Delimiters (blank spaces, tabulations, line feeds) are often required to separate syntactical constructs. Finally, the use in FoCaLize sources of OCaml or Coq keywords (such as `Set`) or of standard library file names (such as `list.v`) may result in name clashes. For these reasons, the reader is advised to respect as much as possible the form and the names used in this tutorial.

The FoCaLize syntax includes several forms of comments. Single line comments start with a -- and go to the end of the line. Block comments are delimited by `(*` and `*)`, and can be nested, that is, it is possible to comment a block of code which includes comments. However, for the sake of readability, we will not use comments in our examples; the reader is free to add comments were he feels appropriate. It is advised to avoid some forms of comments, such as `(**`, `(*--`, or `"` (double quote) in block comments; please refer to the FoCaLize reference manual for detailed explanations.

Commands, file names as well as outputs or error messages are in bold font, for example **focalizec superset.fcl**. Terms representing specific FoCaLize concepts are introduced using an emphasised font, for example *collection*. Finally, mathematical statements are presented using the standard notations, for example $\forall\,(x:T),\ x=x$ (for any x belonging to the set/type T, x is equal to x).

2 A Quick Overview of FoCaLize

FoCaLize is an integrated development environment (IDE) with formal features. Beyond the compiler, that from a FoCaLize source file produces source code for OCaml and proof scripts to be checked by Coq, FoCaLize also provides an automated prover (Zenon), a test tool (FoCTest), a documentation tool (FoCDoc), and other utilities. We mainly focus in this tutorial on the use of the compiler (**focalizec**) and of the Zenon automated prover (**zvtov**). Note that FoCaLize is the successor of the FoC and FoCaL tools, with a fully redeveloped compiler.

FoCaLize being a formal method, a typical development includes not only data structures and programs, but also logical properties. This allows for the description of specifications and of implementations, while providing a mathematical guarantee that implementations are indeed compliant with their specifications. A *specification* consists of function signatures, but also of associated properties that have to be satisfied, as in the following example describing a commutative binary operation f over a set $\mathbb{S}$ with a left neutral element:

$$f:\mathbb{S}\to\mathbb{S}\to\mathbb{S} \qquad \forall\, x\ y,\ f(x,y)=f(y,x) \qquad \forall\, x,\ f(Z,x)=x$$

The left part is frequently encountered in classical, non-formal languages, under the names signature, interface, prototype or specification. The two properties on the right part, on the other hand, are less usual; in a formal method, they do not represent comments or assertions checked at runtime, but a true requirement constraining *implementations* – that is the programs with concrete datatypes and algorithms. The correctness of the implementations has to be statically ensured at compilation time by proving that they satisfy the properties, an activity relying on *formal proofs* (in the mathematical sense), whose production is only partially automated but which are mechanically checked.

The cost of develop ping formal specifications and proofs is compensated by the important gain w.r.t. the confidence in the development. Various studies furthermore indicate that the total development and ownership cost is likely to be reduced, due to disambiguation of the specifications, earlier detection of problems, lighter test obligations and reduced maintenance activities. Note also that it is possible to further assess the specification itself by deriving consequences, to check that the specification is both correct and sufficiently detailed; taking our previous example, it is possible to show that a valid implementation of f is such that $f(x, Z) = x$ – just combining required properties, without having to know what is this implementation.

The FoCaLize language therefore combines a pure functional programming language inspired by OCaml and a logical language. It is also object-flavoured, developments being organised through a hierarchy of *species*, build by *inheritance* or *parameterisation*. In essence, a *species* is a kind of record combining function *signatures* (that is their type), function *definitions* (that is their code), *logical properties* and *proofs*.

When a species S_2 *inherits* from a species S_1, that means that S_2 includes at least all the features of S_1, and can be enriched with its own functions or properties – intuitively speaking, any correct implementation of S_2 is therefore also a correct implementation of S_1.

A species can also use the implementation of other species to build its own definitions, properties and proofs, through a *parameterisation mechanism*. The parameters are required to be effective and consistent, that is to be such that every function declaration has an associated definition, and every property a proof that it holds. Furthermore, the validity of the properties of these effective parameters is likely to depend upon the implementation choices done during their development, and it is therefore very important to preserve them. This is addressed in FoCaLize by the notion of *collection*: the implementation of a complete species (that is a species whose all functions are defined and all properties are proved) whose concrete data representation is hidden – a form of abstract data type. A collection only exposes the declaration of its functions and the statement of its properties, through a structure called the *interface* of the collection; collections can therefore be used safely as effective parameters.

Practically, if a species S has a formal parameter C of interface I, then S can use all the functions and properties named in I in its body to build the definition of its functions and to prove its properties, but cannot modify these functions or even examine their definition. The compiler ensures that any collection D given as an effective parameter for C indeed offers all the functions and properties stated in I. Note that species can also be parameterised by entities, that is values of a collection. This is a powerful feature, e.g. to describe structures such as $\mathbb{Z}/n\mathbb{Z}$ where n is such a parameter, but we will not use it in this tutorial.

3 The Development Case: Sets and Subsets

We deal in this tutorial with the classical example of subsets. We want to describe the implementation of subsets of values, ensuring that it is bug-free, at least w.r.t. a specification that we will make as complete as possible.

It is important, in such a development, to learn to abstract the fundamental concepts. Indeed, one can easily describe such mathematical structures having a precise idea about their implementation in mind, and consequently describing this implementation rather that the structure itself. For example, in the case of subsets, a classical choice is to use sorted and injective[2] lists; whereas this is a valid implementation, there are other choices, and it would be a mistake to capture too early some of the specificities of this encoding, such as for example the existence of a comparison between values or of an order inside the concrete representation of a subset.

As we will see, we adopt here a progressive approach, justified by various engineering considerations regarding FoCaLize developments, but also by the will to stick to the formal vision presented just before. An interesting consequence of this vision is that a genuine freedom is given to the developer, that can choose between various approaches for example to favour genericity and reusability, or to optimize algorithms.

The rest of this paper describes the various stages of the development, distinguishing between specification activities (the "what to do") and refinements, that is the activities ensuring progress toward an implementation (the "how to do"). We describe supersets, from which values are obtained, and subsets of these supersets, before implementing these specifications. Our main objective is to implement finite subsets as lists of values, but we will consider possible alternatives as well.

4 Specifying Supersets

When attempting to specify what is a subset of a superset – for example subsets of $\mathbb{N}$ – we need first to specify this superset. The simplest approach would be to decide to represent subsets of a given superset, for example subsets of values from *int*. Yet of course that would force us to fully redevelop such an implementation for any relevant type, a rather inappropriate strategy.

We therefore favour the generic approach, through a form of polymorphic specification. In FoCaLize, this is possible by using parametrisation. Species can indeed be parametrised by an *interface*, which means that implementation of the species will have to instantiate this parameter by a collection having the required interface. In practice, this means that before defining a species for subset, parametrized by an interface describing what is a superset, we need first to define the superset interface. This is done by creating first a species for superset, whose only role is to be passed as a parameter.

4.1 The Superset Species

What do we expect from a superset? Not much, as we can consider subsets of nearly anything. Furthermore we do not need to enforce supersets to have a rich

[2] That is, lists in which a given value appears at most once.

structure with powerful operations. For now, only one very generic operator appears useful, an equivalence relation – representing a form of equality, for example to be able to test whether or not a subset contains a value. Note that we do not require this equivalence to be a strict equality, for example if we store strings that encodes natural values, we may decide that "1", "01" and "+001" are equivalent – and checking if a given subset contains for example "+1" will return true even if the stored value is in fact "01".

We therefore edit a FoCaLize source file, named **superset.fcl**, and type in the following code:

```
use "basics" ;;

species Superset =

 signature equal : Self -> Self -> basics#bool ;

end ;;
```

The first line of our code indicates that we intend to use declarations and definitions from the file **basics.fcl** (or more precisely its compiled version, **basics.fo**, which has to be in the library path of the compiler). This standard file contains the definition of the type `bool` which is needed here, as well as other utilities, and is generally required for any FoCaLize development.

Tip 1 *[Compilation Units] Access to declarations and definitions from other files is possible with the* `use` *or* `open` *directives; the latter allows for shorter notations (e.g.* `#bool` *or* `bool` *instead of* `basics#bool` *which is referred to "file qualified notation". We will see later a similar mechanism to invoke species methods that we call "species qualified notation". In absence of ambiguity between these two qualification mechanisms, we will generally use the shorter term of "qualified notation"). Note that the* `use` *and* `open` *directives are not transitive, that is for example if B opens A and C opens B, C does not have access to declaration and definitions of A.*

We then define a species `Superset` to represent the very abstract entity that we call a superset. This species represents in fact any collection (any implementation) whose interface contains a function `equal` with the required type; as we do not provide the definition (the code) of this function, but just its type, we use the keyword `signature`. The type of the function is $Self \rightarrow Self \rightarrow bool$, which means that it takes two parameters of the current species[3] and returns a boolean value.

Tip 2 *[Syntax] Species names have to start with an uppercase character.*

Tip 3 *[Syntax] Separators (that is blank spaces, tabulations, line feeds) are often required between syntactical constructs.*

At this stage, we are not using any formal feature; indeed, we are just specifying here a form of object with a given method. To have a meaningful specification, we further require the equal function to represent an equivalence relation (that is to be reflexive, symmetric and transitive). In addition, we also change the name of the method `equal`: to be easier to read, we benefit from the FoCaLize ability to manage symbols, and use the much more explicit symbol `=` instead. We therefore modify `Superset` as follows:

[3] More precisely, `equal` expects two values of the concrete type used as the support for the collection implementing the species.

```
species Superset =

 signature ( = ) : Self -> Self -> basics#bool ;
 property eq_refl : all x : Self, x = x ;
 property eq_symm : all x y : Self, x = y -> y = x ;
 property eq_tran : all x y z : Self, x = y -> y = z -> x = z ;

end ;;
```

The symbol =, in FoCaLize, is associated to infix notations (but can still be used as a prefix operator, when put between parentheses, as in its signature). This allows for a very user-friendly presentation of the three properties that we expect about an equivalence relation, namely reflexivity, symmetry and transitivity. We use the keyword *property*, hence we do not prove anything at this stage. We just require any collection implementing this species to provide such a proof at some point of the development – for example as soon as = is defined.

Tip 4 *[Using Symbols] Methods can be denoted by names or symbols; the choice of an appropriate symbol may greatly improve the readability of a code.*

Superset now specifies any collection offering a function named = taking two parameters in the collection and returning a boolean, that is a relation; but it also requires this relation to be reflexive, transitive and symmetric. This is a pure specification: there are no definitions, no code. We explain what we expect, but the developer is free to propose any compliant implementation. Note by the way that our specification does not even enforce *Superset* to contain a value: it can be implemented by an empty type.

At this stage, it is possible to compile our development with the command **focalizec superset.fcl** – of course, there is not much to expect from this compilation, except for checking the syntax. It invokes the FoCaLize compiler, as well as OCaml compiler, the Zenon prover – but there are no proof obligations at this stage, the species only containing properties – and the Coq proof checker, producing the following files:

File name	Produced by	Used by	Description
superset.fcl	Text editor	**focalizec**	FoCaLize source file
superset.fo	**focalizec**	**focalizec**	FoCaLize object file
superset.zv	**focalizec**	**zvtov**	Proof obligations
superset.ml	**focalizec**	**ocamlc**	Program source file
superset.pfc	**zvtov**	**zvtov**	Zenon proof cache
superset.v	**zvtov**	**coqc**	Proofs
superset.vo	**coqc**	**coqc**	Coq object file
superset.cmi	**ocamlc**	**ocamlc**	OCaml interface file
superset.cmo	**ocamlc**	**ocamlc**	OCaml object code

4.2 A First Proof

Whereas *Superset* is a very short and simple specification, it can be enriched with additional results without further requirements. For example, combining reflexivity and transitivity, it is possible to prove the following property about = of the *Superset* species as follows:

```
theorem eq_symmtran : all x y z : Self, x = y -> x = z -> y = z
 proof = by property eq_symm, eq_tran ;
```

We use the keyword *theorem* instead of *property* to indicate that the proof follows – it is provided after the keyword *proof*. Hence, a *property* accompanied by its *proof* is equivalent to a *theorem*. In fact, we do not detail a real proof, but rather tips that allow ZENON to automatically derive the proof[4]; we claim that *eq_symmtran* is a direct consequence of *eq_symm* and *eq_tran*. At this stage, we can invoke the compiler to check that indeed ZENON succeeds.

Tip 5 *[Derived properties] Once primitive properties have been introduced, other results can be proved, for example user-relevant corollaries, to check the validity and the completeness of the specification.*

Note that *eq_symm* and *eq_tran* are in fact used as hypothesis to prove *eq_symmtran*, in other words we have proven $eq\_symm \Rightarrow eq\_tran \Rightarrow eq\_symmtran$. That means that the validity of the latter depends upon the consistence of the formers, and that there is therefore no real assurance until proofs are provided for them.

In the rest of this tutorial, we will not further mention standard compilation steps; the reader is invited to compile when he considers it is relevant.

4.3 A First Definition

To further illustrate the *late binding* feature of FOCALIZE, that is in essence the ability to describe and use entities which are not yet defined, we again enrich the species *Superset*, this time with the *definition* of a function:

```
let ( <> ) (x, y) = basics#( ~~ )(x = y) ;

theorem diff_irrefl : all x : Self, ~(x <> x)
 proof = by definition of ( <> ) property eq_refl ;

theorem diff_symm : all x y : Self, x <> y -> y <> x
 proof = by definition of ( <> ) property eq_symm ;
```

In this case, we provide the code for the function <>, introduced by the keyword *let*. Without surprise, we indicate that <> is the negation of =, but one should note that = itself is not yet defined. However, this is sufficient to prove properties of <>, such as *diff_irrefl* and *diff_symm*. The proof of these properties is again automatically derived by ZENON, the tips in this case also ensuring that the definition of <> is visible from the proof context.

A definition of <> is provided in the species *Superset*, but it is possible to override it later (in inheriting species or collection), for example to optimize the implementation, once more information about the structure are provided.

It is fundamental to remember that FOCALIZE enforces encapsulation; this means that any definition in a species, such as the one for <> here, is visible only in this species (or the inheriting ones). As definitions are never visible outside a species, we recommend for such early definition to associate dedicated properties capturing the essence of the definition in a logical form, visible from other species, such as *diff_eq* that we add to our species as follows:

```
theorem diff_eq : all x y : Self, x <> y <-> ~(x = y)
 proof = by definition of ( <> ) ;
```

[4]The success of ZENON for complex proofs depends upon the computer architecture and the parameters used for its invocation.

diff_eq is here described as a theorem whose proof depends upon the definition of <>. Any later redefinition of <> will delete this proof, and transform back *diff_eq* into a proof obligation. The interest of a property like *diff_eq* is that it can be used instead of the definition of <> to prove *diff_irrefl* and *diff_symm*: if <> is indeed redefined later, only the proof of *diff_eq* will have to be redone.

Tip 6 *[Early Definitions] Definitions can be introduced at any stage of a* FoCaLize *development; it is recommended to capture the meaning of early definitions in properties (proved using the definition), and to use these properties in proofs, instead of the definitions themselves, to limit the consequences of redefinitions.*

4.4 Clearing of the Superset Species

We have enriched our *Superset* species just to illustrate some of the features of FoCaLize. As these artificial examples are not used later in this tutorial, we suggest to delete them. The file **superset.fcl** should be modified as follows:

```
use "basics" ;;

species Superset =

 signature ( = ) : Self -> Self -> basics#bool ;
 property eq_refl : all x : Self, x = x ;
 property eq_symm : all x y : Self, x = y -> y = x ;
 property eq_tran : all x y z : Self, x = y -> y = z -> x = z ;

end ;;
```

5 Specifying Subsets

It is now time to specify subsets of values from a superset in a new species. It is possible to append the new code in the previous file, but we prefer to create a new file named **subset.fcl**. As previously stated, the directives *use* or *open* can be used to access previous definitions. Remember that with *use* explicit and detailed names are required, for example *superset#Superset!eq_refl*, whereas *Superset!eq_refl* is sufficient if we use *open* instead.

```
use "basics" ;;
open "superset" ;;
```

5.1 Generic Subsets

The species *Subset* is parametrised by a collection *Val* which implements the interface defined by the species *Superset*. Regarding the method, we provide a membership operation deciding whether or not a value belongs to a subset, denoted <<; practically, membership defines a subset, as any subset is characterised by the values it contains[5]:

```
species Subset(Val is Superset) =

 signature ( << ) : Val -> Self -> basics#bool ;
```

[5]Note that we describe in fact subsets for which membership is decidable.

```
end;;
```

Of course using only this method we cannot do much: neither do we have a way to exhibit a subset, nor to express any property about membership. So additional methods have to be declared, in association with properties that sufficiently describe what we expect from these methods. However, we also have to be cautious not to add too powerful methods, that would lead to over-specify the species and reduce the acceptable implementations (or logical models).

It appears reasonable to add a method to build an empty subset. Indeed, it always exists such a subset, even if for example `Val` is itself empty. Another seemingly harmless feature (in the sense that it is likely to be implementable whatever the concrete representation we choose) is the ability to derive subsets from other subsets by adding or removing a value. Following these principles, our species is enriched as follows:

```
signature empty : Self ;
property mem_empty : all v : Val, ~(v << empty) ;

signature ( + ) : Self -> Val -> Self ;
property mem_insert : all v1 v2 : Val, all s : Self,
                      v1 << s + v2 <->
                      (Val!( = )(v1, v2) \/ v1 << s) ;

signature ( - ) : Self -> Val -> Self ;
property mem_remove : all v1 v2 : Val, all s : Self,
                      v1 << s - v2 <->
                      (~(Val!( = )(v1, v2)) /\ v1 << s) ;
```

In this specification, each method producing a subset is associated to a property detailing its behaviour with regard to membership. As membership indeed characterises a specific subset, such properties are as precise as a definition. This is typical of FoCaLize specifications, where the concrete datatype stays hidden from outside the species: properties are generally relations between methods, describing the structure of a species.

We have adopted a very dense presentation of these properties: `mem_insert` claims that a value is a member of `s + v` if and only if this is `v` or if it was already a member of `s`. It could have been split in three different properties, but we here trust Zenon to exploit automatically all variants of this properties, and we favor this form to reduce the number of tips to be provided for proofs.

Note that the notation `=`, in the context of the species `Subset`, would refer either to a method defined in the current species or to a method defined at *top level* (that is outside a species or a collection), possibly in one of the included files. As we want to use the equality of the parameter `Val`, we need the explicit notation `Val!( = )`, which is a prefix notation.

Are additional methods desirable at this stage of the specification? We choose here not to go further, yet this is a matter of taste and style. For example, it may be acceptable to specify union and intersection, as it does not seem to restrict the possible concrete representations for subsets. On the other hand, from a practical point of view, that means that any implementation of the species `Subset`, in addition to providing the definition for `<<`, `+`, `-` as well as the proofs of the associated properties, would also have to provide a definition and the proofs for union and intersection, even if these last operations are not useful

for a given application. So it may be more appropriate to add such operations in a later species and let developers to choose to use the *Subset* specification or one of its extended versions.

5.2 Extensional Subsets

Whereas our main objective is to implement finite subsets as lists, we choose here to introduce very gradually various levels of specifications. Note that nothing enforces at this stage the species *Subset* to only describe finite subsets, and we do not want to further constrain it by requiring other methods and properties in *Subset*. We prefer to adopt a smoother approach, adding another level of specification by creating a new species to represent finite subsets, or more precisely extensional subsets.

Indeed, it is not trivial to specify finite subsets. One of the most classical approaches is to define a cardinal operator in $Self \rightarrow \mathbb{N}$, returning the number of elements in the subset. The simple existence of this operator, total over $Self$, combined with properties indicating its semantics, is indeed sufficient. Yet that would require using natural values, that is to have a collection parameter with a species representing $\mathbb{N}$. Appropriate species exist in the FoCaLize standard library, but we prefer to avoid in this tutorial such dependencies for now.

We therefore adopt here a slightly different approach, by specifying extensional subsets, that is subset whose content can be analysed systematically.

5.2.1 Inclusion

Extensional subsets being subsets, we use the inheritance mechanism: we create a new species *ExtSubset* parametrised by a collection *Val*, which inherits of *Subset(Val)*, that is of all its methods and properties. Of course, we intend to enrich this specification with new methods and properties.

The new requirement in *ExtSubset* is to have a method checking whether or not a subset is included in another, denoted *<:* in our code. Taking benefit from the specification of the (yet undefined) inclusion method, it is already possible to prove that inclusion is reflexive and transitive:

```
species ExtSubset(Val is Superset) =

 inherit Subset(Val) ;

 signature ( <: ) : Self -> Self -> basics#bool ;
 property mem_incl : all s1 s2 : Self,
                     s1 <: s2 <-> all v : Val, v << s1 -> v << s2 ;
 theorem incl_refl : all s : Self, s <: s
  proof = by property mem_incl ;
 theorem incl_tran : all s1 s2 s3 : Self,
                     s1 <: s2 -> s2 <: s3 -> s1 <: s3
  proof = by property mem_incl ;

end;;
```

Any collection implementing *ExtSubset* will therefore have to provide code for the method *<:* as well as a proof that this code indeed represents inclusion; provided this proof reflexivity and transitivity will be ensured as well without further work.

Tip 7 (Inheritance) *The* `inherit` *clause has to be the first one in a species; it indicates that the current species includes all the signatures, definitions, properties and proofs on the inherited species.*

5.2.2 Extensional Equality

An evident enrichment is to define the equality of two finite subsets as the reciprocal inclusion. This should however ring a bell: if the species *ExtSubset* offers an equality, then it can probably also inherits from the species *Superset*. That's our design choice here, using the multiple inheritance feature provided by FoCaLize. We therefore modify and extends the species *ExtSubset* as follows:

```
species ExtSubset(Val is Superset) =

 inherit Superset, Subset(Val) ;

 signature ( <: ) : Self -> Self -> basics#bool ;
 property mem_incl : all s1 s2 in Self,
                     s1 <: s2 <-> all v in Val, v << s1 -> v << s2 ;
 theorem incl_refl : all s in Self, s <: s
  proof = by property mem_incl ;
 theorem incl_tran : all s1 s2 s3 in Self,
                     s1 <: s2 -> s2 <: s3 -> s1 <: s3
  proof = by property mem_incl ;

 let ( = ) (s1, s2) = if (s1 <: s2) then (s2 <: s1) else false;

end;;
```

ExtSubset is now a species combining the interface of *Superset* and *Subset*: equality (of subsets), membership, insertion, removal, as well as the associated properties[6]. Equality is also defined using inclusion[7]. Using these properties and the definition of `=`, we can discharge the proof obligations inherited from *Superset*:

```
 proof of eq_refl = by definition of ( = ) property incl_refl ;
 proof of eq_symm = by definition of ( = ) ;
 proof of eq_tran = by definition of ( = ) property incl_tran ;
```

Defining `=` using `<:` is apparently relevant, and allows us to discharge the associated proof obligations. This is not constraining, as we know that thanks to late binding it is possible to later change the definition of `=`, for example for optimisation. But of course such a redefinition would erase the proofs depending upon the current definition, here *eq_refl*, *eq_symm* and *eq_tran*.

Now, we can already consider that such a redefinition is more than likely to happen. For example, if we implement subsets as lists, checking equality will require checking twice inclusion between lists, while comparing directly the two lists is more efficient. So it is indeed pleasant to be able to discharge so soon the proof obligations, but it is probably useless – at least in the current form of the species. Therefore, exactly as we have done when dealing with the definition of `<>` of the species *Superset* in Sub. 4.3, instead of defining `=` using `<:`, we capture the meaning of the definition in a property, which is used to discharge the proof obligations:

[6] As *ExtSubset* inherits from *Superset*, it can parametrise *Subset*, that is it is possible to consider subsets of a superset, but also subsets of subsets of a superset, and so on.

[7] `=` is defined with an *if*, which is primitive in FoCaLize; it is also possible to use the boolean and, denoted `&&` and provided in **basics.fcl**.

```
property eq_incl : all s1 s2 : Self,
                   s1 = s2 <-> s1 <: s2 /\ s2 <: s1 ;
proof of eq_refl = by property eq_incl, incl_refl ;
proof of eq_symm = by property eq_incl ;
proof of eq_tran = by property eq_incl, incl_tran ;
theorem mem_eq : all s1 s2 : Self, s1 = s2 <->
                 (all v : Val, v << s1 <-> v << s2)
 proof = by property eq_incl, mem_incl ;
```

Note that here we can delete the definition of `=`. It is trivial to show that `s1<:s2 && s2<:s1` indeed satisfies `eq_incl`, if we decide to use such a definition; on the other hand, if we provide a more efficient algorithm for `=`, we have just to check its validity by proving `eq_incl`, never having to prove again `eq_refl`, `eq_symm` and `eq_tran`.

5.2.3 A Few Trivial Properties

Having proposed a definition of extensional subsets by enforcing a signature containing inclusion with some standard properties, we now complete our species with additional facts that we consider relevant and useful for later uses. We add the following properties and proofs in the species `ExtSubset`:

```
theorem incl_empty : all s : Self, empty <: s
 proof = by property mem_incl, mem_empty ;
theorem incl_insert : all s : Self, all v : Val, s <: s + v
 proof = by property mem_insert, mem_incl ;
theorem incl_remove : all s : Self, all v : Val, s - v <: s
 proof = by property mem_remove, mem_incl ;
theorem incl_insert_mem : all s : Self, all v : Val,
                          v << s -> s + v <: s
 proof = by property mem_insert, mem_incl ;
```

Compiling this new version, Zenon is able to derive a proof for `incl_empty`, `incl_insert` and `incl_remove`, but not for `incl_insert_mem`. Before blaming Zenon, let's produce by hand a derivation tree of the property `incl_insert_mem`, giving a hierarchical vision of the deductive proof. In such a tree the goal is at the bottom, and using deduction rules we progress toward trivial subgoals at the top of the tree, possibly branching (for example, to prove $A \wedge B$ one may provide a proof of A and a proof of B). We got something like:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{?}{s:Self, v\ w:Val, v\in s, v=w \vdash w\in s} \quad \dfrac{\text{Trivial, by assumption}}{s:Self, v\ w:Val, v\in s, w\in s \vdash w\in s}}{s:Self, v\ w:Val, v\in s, w=v \vee w\in s \vdash w\in s}}{s:Self, v\ w:Val, v\in s, w\in s+v \vdash w\in s}}{s:Self, v:Val, v\in s \vdash \forall\, w:Val,\ w\in s+v \Rightarrow w\in s}}{\vdash \forall\, s:Self,\ v:Val,\ v\in s \Rightarrow s+v\subseteq s}$$

This identifies the difficulty preventing the derivation of the proof by Zenon, as something is missing in the left branch, the subgoal denoted by the interrogation mark. Indeed, this subgoal has no reason to be valid: `=` is an equivalence relation, but nothing enforces it to be a congruence w.r.t. membership, that is we have never required $v\in s, v=w \vdash w\in s$. To understand why this may not be the case even in perfectly legitimate cases, remember the comment in Sub. 4.1, about the encoding of natural values as strings.

Of course, using the symbol = it is clear that our intention was to capture a form of equality; so we can emphasise this interpretation by adding the congruence property at the beginning of the species *ExtSubset*:

```
property mem_congr : all v1 v2 in Val, Val!( = )(v1, v2) ->
                     (all s in Self, (v1 << s) <-> (v2 << s)) ;
```

Given this property as an additional tip, ZENON indeed succeeds:

```
theorem incl_insert_mem : all s : Self, all v : Val,
                          v << s -> s + v <: s
 proof = by property mem_congr, mem_insert, mem_incl ;
```

5.2.4 Proving Non Trivial Properties

We can also try to prove the following property:

```
theorem incl_remove_mem : all s : Self, all v : Val,
                          ~(v << s) -> s <: s - v
 proof = by property mem_congr, mem_remove, mem_incl ;
```

ZENON does not succeeds[8], but in this case it is apparently not because of missing information as a proof by hand appears possible:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{Assuming } w=v \text{ is absurd}}{s:Self, v\ w:Val, v\notin s, w\in s \vdash w\neq v} \quad \dfrac{\text{Trivial, by assumption}}{s:Self, v\ w:Val, v\notin s, w\in s \vdash w\in s}}{s:Self, v\ w:Val, v\notin s, w\in s \vdash w\neq v \wedge w\in s}}{s:Self, v\ w:Val, v\notin s, w\in s \vdash w\in s-v}}{s:Self, v:Val, v\notin s \vdash \forall\, w:Val,\ w\in s \Rightarrow w\in s-v}}{\vdash \forall\, s:Self,\ v:Val,\ v\notin s \Rightarrow s\subseteq s-v}$$

It is likely to be a problem of complexity, and human guidance is required.

Two main approaches are possible in FOCALIZE to prove non-trivial properties using ZENON. The first one consists into introducing progressively lemmas, easier to prove, then to use these lemmas to derive the complex results. Unfortunately, this makes the signature of the species more complex with numerous uninteresting results; in other word, the documentation of the species can quickly becomes clumsy. The second approach requires user-guided proof using the FOCALIZE Proof Language (FPL). We adopt the latter, detailing the important steps of the proof and expecting ZENON to complete it, as follows:

```
theorem incl_remove_mem : all s : Self, all v : Val,
                          ~(v << s) -> s <: s - v
 proof = <1>1 assume s : Self, v : Val, hypothesis Hv : ~(v << s),
              prove s <: s - v
          <2>1 assume w : Val, hypothesis Hw : w << s,
               prove w << s - v
           <3>1 prove ~(Val!( = )(w, v)) /\ w << s
            <4>1 prove ~(Val!( = )(w, v))
                 by property mem_congr hypothesis Hv, Hw
            <4>2 prove w << s
                 by hypothesis Hw
            <4>f conclude
           <3>f qed by property mem_remove step <3>1
          <2>f qed by property mem_incl step <2>1
         <1>f conclude ;
```

[8] Remember that actual results depend upon the computer and the parameters of ZENON.

This script details the proof structure to be followed by ZENON. The label *<x>y* indicates the level *x* and the step *y* in this level; levels have to be managed consistently, while steps are just names. In essence, to prove a result labelled *<x>y*, one can introduce a few lemmas labelled *<x+1>y1*, ..., *<x+1>yn* and conclude with a command *<x+1>f qed by step <x+1>y1, ..., <x+1>yn* (and additional hypotheses and properties if required). Note that the step *f* here is the user notation for the final step for a given level, but that the compiler does not care about the value specifying the step as long as it is a unique identifier for the current level. Similarly, the label *<1>f* does not mark the proof of the statement *<1>1* but use this statement to prove the theorem *incl_remove_mem*. Take care to use different step identifiers for a given level, as repeating an identifier will not cause an error but will mask the associated results and prevent ZENON to conclude.

A goal step is associated with assumptions (keyword *assume* or *hypothesis*), a goal (keyword *prove*), and possibly the tips to solve the goal (keyword *by* completed with *definition of*, *property*, *hypothesis*, *step* and *type*). If the goal *<x>y* is not provided with tips, then the next steps (at level *x+1*) provide a strategy to solve it, the last step of the level *x+1* being expected to be a *qed* with tips (or a *conclude*, which is equivalent to a *qed by step <x+1>**).

Writing such a script is better done incrementally and interactively. The first attempt is generally a fully automated proof by ZENON, levels being introduced gradually to give additional tips and guidances for tricky parts. Let's develop such a proof for a new theorem in our species *ExtSubset* that describes a way to split a subset, first attempting an automated proof:

```
theorem remove_insert : all s : Self, all v : Val,
                        v << s -> s = (s - v) + v
 proof = by property eq_incl, mem_incl, mem_insert, mem_remove,
                     mem_congr, Val!diff_eq ;
```

ZENON is however unlikely to derive the proof, so we have to give more details:

```
theorem remove_insert : all s in Self, all v in Val,
                        v << s -> s = (s - v) + v
 proof = <1>1 assume s : Self, v : Val, hypothesis Hv : v << s,
              prove s = (s - v) + v
          <2>1 assume w : Val, hypothesis Hw : w << s,
               prove w << (s - v) + v
               by property mem_insert, mem_remove
          <2>2 assume w : Val, hypothesis Hw : w << (s - v) + v,
               prove w << s
               by property mem_insert, mem_remove, mem_congr,
                  Val!diff_eq
          <2>f qed by property eq_incl, mem_incl step <2>1, <2>2
         <1>f qed by step <1>1 ;
```

The level *<1>* is associated to the theorem we want to prove, and the level *<2>* indicates that to prove the equality, we have to prove the mutual inclusion. Invoking the compiler, we got a message indicating that no proof has been found for *<2>1* and *<2>2*; on the other hand the absence of error messages for *<1>f* also indicates that provided a proof for *<2>1* and *<2>2* ZENON will indeed conclude. So we provide more details, but only for the failing steps, first *<2>1*:

```
          <2>1 assume w : Val, hypothesis Hw : w << s,
               prove w << (s - v) + v
           <3>1 prove ~(Val!( = )(w, v)) -> w << s - v
                by property mem_remove hypothesis Hw
```

```
            <3>f qed by property mem_insert step <3>1
```

The important indication given in step *<3>1* is to consider the case $w \neq v$; ZENON is then able to conclude (by also considering the case $w = v$, for which the property *mem_insert* apply) and prove the subgoal *<2>1*.

Given the same amount of details for *<2>2*, the proof is finally derived:

```
           <2>2 assume w : Val, hypothesis Hw : w << (s - v) + v,
                prove w << s
            <3>1 prove Val!( = )(w, v) -> w << s
                 by property mem_congr hypothesis Hv
            <3>2 prove w << s - v -> w << s
                 by property mem_remove
            <3>f qed by property mem_insert hypothesis Hw
                        step <3>1, <3>2
```

ZENON provides various indications when failing or succeeding, for example:

- When succeeding, the message **unused hypothesis** (alternatively **unused variable**) indicates that some of the tips or variables are useless; note however that useless tips are not always be detected as there is no guarantee that ZENON will find the most "efficient" proof[9].

- When failing, the message **exhausted search space without finding a proof** indicates that it is impossible to prove the goal, at least with just the provided tips. The goal is not derivable from the provided assumptions, or may be false.

- When failing, the message **could not find a proof within the time limit** (alternatively **within the memory size limit** or **within the inference steps limit**) indicates that ZENON has reached the fixed limits passed as parameters; there may be a proof or not. It is then possible either to modify the parameters used when invoking ZENON, or to reduce the number of tips, or to use the FPL to split the proof in subproofs.

To avoid error messages, an alternative solution is to incrementally develop the proof using the keyword *assumed*. For example, to prove a goal, you may consider 3 subgoals and start your proof as follows:

```
proof = <1>1 assume ...
             prove subgoal1
             assumed (* TODO *)
        <1>2 assume ...
             prove subgoal2
             assumed (* TODO *)
        <1>3 assume ...
             prove subgoal3
             assumed (* TODO *)
        <1>f conclude ;
```

The compiler can be invoked to check that indeed the goal can be proved using the three subgoals. It is then possible to focus for example on the proof of step *<1>2*, etc.

[9] In exceptional cases, it may also happen that ZENON indicates that a tip is useless, yet is not able to find a proof without it.

Tip 8 (Proofs) *Complex proofs are better developed incrementally, from the general levels to the more detailed ones, and invoking regularly* ZENON *to check the validity of the currently developed steps.*

5.3 Finite Subsets

We have used inheritance and parametrisation as composition operators between specifications, describing a species *Superset*, a species *Subset* and a species *ExtSubset*.

Our aim is to define additional species inheriting from *ExtSubset*, to provide more and more details, for example indicating that the representation (the concrete datatype of the elements of the species) is based on lists and providing the associated algorithms for the methods. In such a case the inheritance acts as a refinement operator – that is progress toward an implementation, with concrete datatypes and algorithms.

The most straightforward approach would be to embed the definition of the type and operations for lists in the species inheriting from *ExtSubset*. Yet we choose here to first specify and implement lists in an independent hierarchy of species, and then to use this hierarchy to refine *ExtSubset*. Beyond the illustration provided in this tutorial, this approach is also fully justifiable, ensuring reusability of lists in other contexts.

6 Specifying Lists

We edit a new file, named **mylist.fcl**[10], to describe lists, with two directives:

```
use "basics" ;;
open "superset" ;;
```

6.1 Co-Lists

We start with a species *CoList* that represents a very abstract form of lists, possibly infinite. As for *Subset*, *CoList* needs to be parametrised by a *Superset*, representing the values that are put in a list. We also specify various methods, as follows:

```
species CoList(Val is Superset) =

 signature nil : Self ;
 signature cons : Val -> Self -> Self ;
 signature isnil : Self -> basics#bool ;
 signature head : Self -> Val ;
 signature tail : Self -> Self ;

 property isnil_nil : all l : Self,
                      isnil(l) <-> basics#( = )(l, nil) ;
 property isnil_cons : all v : Val, all l : Self,
                       ~ isnil(cons(v, l)) ;
 property head_cons : all v : Val, all l : Self,
                      basics#( = )(head(cons(v, l)), v) ;
 property tail_cons : all v : Val, all l : Self,
                      basics#( = )(tail(cons(v, l)), l) ;
```

[10] The file name **list.fcl** cannot be used because of name clashes with COQ library.

```
 property list_dec : all l : Self,
                     ~ isnil(l) ->
                     basics#( = )(l, cons(head(l), tail(l))) ;

end ;;
```

The *nil* and *cons* methods are used to build *CoList*s, while the *isnil*, *head* and *tail* methods are used to analyse and destruct them. In the properties, *basics#(=)* represents the structural equality, that is the standard equality in both Coq and OCaml. The combination of the properties *isnil_nil* and *list_dec* is very strong, indicating that the concrete implementation of any *CoList* is either *nil* or *cons*, a form of surjectivity of these methods w.r.t. the species.

Note that whereas subsets are things with a decidable membership, *CoList*s are things defining a succession of values: membership is not specified[11] but we have a function returning the head element of a *CoList* . Another characteristic of *CoList*s is that values may appear several time – that is it is possible for example to have $head(l) = head(tail(l))$. This illustrates the difference of point of view: (finite) subsets and (finite) lists can appear pretty similar once implemented, but the intentions are different (with the only difference being the number of occurrences of a same element).

The properties listed in *CoList* are sufficient to prove other expected results, such as for example the injectivity of *cons*:

```
 theorem cons_left : all l1 l2 : Self, all v1 v2 : Val,
                     basics#( = )(cons(v1, l1), cons(v2, l2)) ->
                     basics#( = )(v1, v2)
  proof = <1>1 assume t1 t2 : Self, h1 h2 : Val,
               hypothesis Heq : basics#( = )(cons(h1, t1),
                                             cons(h2, t2)),
               prove basics#( = )(h1, h2)
           <2>1 prove basics#( = )(head(cons(h1, t1)),
                                   head(cons(h2, t2)))
                by hypothesis Heq
           <2>2 prove basics#( = )(h1, head(cons(h2, t2)))
                by step <2>1 property head_cons
           <2>f qed by step <2>2 property head_cons
          <1>f qed by step <1>1 ;

 theorem cons_right : all l1 l2 : Self, all v1 v2 : Val,
                      basics#( = )(cons(v1, l1), cons(v2, l2)) ->
                      basics#( = )(l1, l2)
  proof = <1>1 assume l1 l2 : Self, v1 v2 : Val,
               hypothesis Heq : basics#( = )(cons(v1, l1),
                                             cons(v2, l2)),
               prove basics#( = )(l1, l2)
           <2>1 prove basics#( = )(tail(cons(v1, l1)),
                                   tail(cons(v2, l2)))
                by hypothesis Heq
           <2>2 prove basics#( = )(l1, tail(cons(v2, l2)))
                by step <2>1 property tail_cons
           <2>f qed by step <2>2 property tail_cons
          <1>f qed by step <1>1 ;
```

As a final point, one can remark that the *head* function is specified only for non-empty list. For an empty list, it has to return a value from *Val*[12], but

[11] It may not be possible to browse all elements in a *CoList* to find a specific value.

[12] Unless an exception is raised.

which one? Remember that *Val*, which represents a collection implementing the interface of *Superset*, can be empty; in such a case, the list species still contains the empty list (and only the empty list), and implementing the *head* function can be "difficult".

6.2 Finite Lists

An abstract entity such as *CoList* is not very interesting. We aim at representing standard lists, as they are defined in OCaml and Coq for example, which are in particular finite (well-founded), even if the species of all lists is infinite.

Finiteness is a tricky concept to capture. Similarly to what is mentioned in Sub. 5.2, a possible approach is for example to specify a function in $Self \to \mathbb{N}$ computing the length of the list provided as a parameter. The existence of such a function, that has to be total, indeed ensures finiteness of any value of the species. But how do we specify that natural values are themselves finite? We are just pushing back the problem to another species.

Another possible approach is to provide a concrete representation which is itself finite and encodes only finite values. This would correspond, in our case, into using the inductive definition of lists of Coq or OCaml, knowing that such a definition only describes finite constructions.

But it is also possible indeed to specify finiteness, as illustrated here by introducing a new species *FiniteList*:

```
species FiniteList(Val is Superset) =

 inherit CoList(Val) ;

 property finite :
  all f : (Self -> basics#bool),
  f(nil) ->
  (all l : Self, f(l) -> all v : Val, f(cons(v, l))) ->
  all l : Self, f(l) ;

end ;;
```

The property *induction* requires the standard induction principle for lists to be valid: to prove that a boolean function f is true for any list, it is sufficient to prove that it is true for the empty list and that if it is true for a list t then it is also true for the list $cons(h, t)$.

It indeed provides a strategy to prove e.g. $f(cons(a, cons(b, cons(c, nil))))$: from $f(nil)$ and $f(t) \Rightarrow f(cons(h, t))$, one can prove $f(cons(c, nil))$, using again $f(t) \Rightarrow f(cons(h, t))$ he derives $f(cons(b, cons(c, nil)))$, etc.

The property *induction* is higher-order: it quantifies over all possible decidable predicates that can be applied to *Self* – that is in fact over all the methods of the species acting over *Self*, as encapsulation prevents manipulations from outside the species. Note that higher-order properties can be expressed in FoCaLize, but are not manageable by Zenon; in such a case, a proof in Coq is required, as illustrated later.

Why does the property *induction* prove that lists are finite? Let's first define the finiteness predicate *finite* by two axioms:

$$finite(nil) \qquad \forall\, t,\ finite(t) \Rightarrow \forall\, h,\ finite(cons(h, t))$$

The first claims that the empty list is finite, and the second that if a list is finite, then it is still finite after having added an element to it. It is then possible to prove the theorem $\forall\, l$, *finite*(l) by applying `induction`.

Having ensured that `FiniteList` only describes finite lists, we may also add the specification of two methods, membership `mem` and deletion `del`, whose implementation is likely to have to browse all the elements of a list:

```
signature mem : Val -> Self -> basics#bool ;
property mem_nil : all v : Val, ~ mem(v, nil) ;
property mem_cons : all v h : Val, all t : Self,
                    mem(v, cons(h, t)) <->
                    Val!( = )(v, h) \/ mem(v, t) ;

signature del : Self -> Val -> Self ;
property mem_del : all v w : Val, all l : Self,
                   mem(v, del(l, w)) <->
                   ~ Val!( = )(v, w) /\ mem(v, l) ;
```

Note that the property `mem_del` states that the method `del` has to remove all occurrences of values which are equal to its second parameter.

7 Refining Lists

Intuitively, we have specified in the previous section what we need: `FiniteList` describes structures to collect finitely many values, with methods to build and analyse such structures.

In this section, we continue the development by inheritance, but having in mind a *refinement* approach: we do not specify anymore the structure itself, but instead we provide an implementation. Still favouring a very progressive approach in this tutorial, we *refine* the species `FiniteList` by a species `EnumList`, with a few algorithms, before providing a complete species `InductiveList` in which the *representation*, that is the concrete datatype used to encode the specified structures, is defined.

7.1 Enumerable Lists

We create first a new species `EnumList` describing intuitively lists whose elements can be enumerated. Editing the file **mylist.fcl**, we start `EnumList` as follows:

```
species EnumList(Val is Superset) =

 inherit FiniteList(Val) ;

end ;;
```

7.1.1 Recursive Membership

The first definition that we provide in `EnumList` is the membership. Knowing that any list in *Self* is finite, we browse the structure using the dedicated methods to look for a specific value:

```
 let rec mem(v : Val, l : Self) =
  if isnil(l)
  then false
```

```
  else (if Val!( = )(v, head(l)) then true else mem(v, tail(l))) ;
 (* TODO : Termination Proof *)
```

The definition for the method *mem* is recursive, as computing *mem(v,l)* requires in some cases to compute *mem(v, tail(l))*. In a formal development, one has to prove that recursive functions always terminate to avoid logical inconsistency[13]. In its current version, FoCaLize offers two ways to deal with such a function:

- The keyword *recstruct* indicates that the function is recursive, and that any recursive call is done on a parameter which is structurally "smaller" than the initial one. It is illustrated later in this tutorial.
- The keyword *rec* indicates that the function is recursive. In this case a termination proof is normally added after the definition, for example:

  ```
  let rec f(...)=... termination proof = by order ....
  ```

With regard to the second case, the full support in FoCaLize of recursive functions, with proofs of termination based on measures, orders and so on is still in development and is not further detailed here. For this reason, the termination proof is for now admitted and is not required in the code. The FoCaLize compiler however warns that the species is potentially unsafe and will generate a pseudo-termination proof instead, in order to have the whole program however accepted by Coq. In some sense, we afford lying to Coq, being optimistic about the part of the consistency of our program involved by the termination of this function.

We adopt here the second version, noting that if the property *finite* is valid, the function *mem* terminates. Indeed, a possible interpretation of *finite* is that it enforces any list to be build-able (accessible) starting from the list *nil* and using only the function *cons*; reciprocally, because of the properties stated in *CoList* we know that *head* and *tail* act as decomposition operators and allow for a systematic enumeration of the values of the list.

Having defined *mem* (and admitted termination), we can prove that this function has the expected properties as follows:

```
proof of mem_nil = by definition of mem property isnil_nil ;

proof of mem_cons =
 <1>1 assume v h : Val, t : Self, hypothesis H : Val!( = )(v, h),
      prove mem(v, cons(h, t))
  <2>1 prove Val!( = )(v, head(cons(h, t)))
       by property head_cons hypothesis H
  <2>f qed by definition of mem property isnil_cons step <2>1
 <1>2 assume v h : Val, t : Self, hypothesis H : mem(v, t),
      prove mem(v, cons(h, t))
  <2>1 prove mem(v, tail(cons(h, t)))
       by property tail_cons hypothesis H
  <2>f qed by step <2>1 definition of mem property isnil_cons
 <1>3 assume v h : Val, t : Self, hypothesis H : mem(v, cons(h, t)),
      prove Val!( = )(v, h) \/ mem(v, t)
  <2>1 hypothesis H2 : ~ Val!( = )(v, h),
       prove mem(v, tail(cons(h, t)))
       by hypothesis H, H2 definition of mem
```

[13] Consider for example the function *let rec f(n in Nat) = f(n)*: should it be a valid definition, it could have any return type – and in particular it could be used as a trick to build a value from an empty type.

```
          property isnil_cons, head_cons
  <2>f qed by step <2>1 property tail_cons
 <1>f conclude ;
```

7.1.2 Recursive Deletion

We can add a second definition in our species, for the deletion method:

```
let rec del(l : Self, v : Val) =
 if isnil(l)
 then nil
 else if Val!( = )(v, head(l))
      then del(tail(l), v)
      else cons(head(l) , del(tail(l), v)) ;
(* TODO : Termination Proof *)
```

Again, termination is admitted in this case.

This second method is very illustrative. Indeed, provided this recursive definition, it is possible to prove the property *mem_del*, yet we need to use the induction principle. That is, as indicated by the property *induction*[14] to prove $\forall\, l,\ P(l)$ (any list satisfies a property P), it is sufficient to prove $P(nil)$ (the property is true for the empty list) and that for any list t, assuming $P(t)$ (the property is true for t), then for any h, $P(cons(h,t))$ is provable as well (the property is still true when adding an element to t).

We can foresee a rather complex proof, as *mem_del* is a composite statement and because we have to use *induction*. To avoid a very long proof we suggest here a decomposition of *mem_del* into three subgoals, and further to use the *logical let* feature to improve the readability of the proof. Let's start with a first subgoal:

```
logical let mdm_(v : Val, w : Val, l : Self) =
 mem(v, del(l, w)) -> mem(v, l) ;

theorem mem_del_mem : all v w : Val, all l : Self, mdm_(v, w, l)
```

We first define a statement *mdm_*, using the keyword *logical let* to introduce a parametrised statement. Note that we do not claim that such a statement is valid: we just introduce a name for a complex **logical expression**. The theorem *mem_del_mem*, on the contrary, claims that *mdm_* is always valid.

The first level of the proof of *mem_del_mem* is as follows:

```
 proof = <1>1 assume v w : Val,
              prove mdm_(v, w, nil)
              assumed (* TODO *)
         <1>2 assume v w : Val, t : Self,
              hypothesis Hind : mdm_(v, w, t), assume h : Val,
              prove mdm_(v, w, cons(h, t))
              assumed (* TODO *)
         <1>3 assume v w : Val,
              prove mdm_(v, w, nil)->
                    (all t : Self, mdm_(v, w, t) ->
                     all h : Val, mdm_(v, w, cons(h, t))) ->
                    all l : Self, mdm_(v, w, l)
              assumed (* Standard induction principle *)
         <1>f conclude ;
```

[14] In fact, we need a slightly more generic form as we need to quantify over all predicates.

Thanks to the use of the *mdm_* name, we can see clearly that the step *<1>1* is of the form $P(nil)$, and the step *<1>2* of the the form $P(t) \Rightarrow P(cons(h,t))$. They have to be proved but for the current stage we just assume these results.

The label *<1>3* states very clearly the induction principle, instantiated for the property we are trying to prove, that is *mdm_*. We also assume this result, but in this case because the current version of ZENON is not able to manage higher-order declarations such as *induction* – a COQ proof is still possible, but here we decide to just accept this result as a fact to prevent running in too complex notions out of the scope of this tutorial . The step *<1>f* concludes.

We can check that this proof structure is valid by invoking the compiler, before pursuing the development. For the step *<1>1*, it is straightforward:

```
proof = <1>1 assume v w : Val,
             prove mdm_(v, w, nil)
             by definition of mdm_, del property isnil_nil
```

Note that we need to make visible the definition of the statement *mdm_*.

For the step *<1>2*, it is a little longer. The idea is that the definition of the *del* function is visible, and can be analysed by case reasoning – for example, an *if* construct is dealt with by assuming that the condition is true in a first case, and false in a second case. The resulting proof is as follows:

```
          <1>2 assume v w : Val, t : Self,
               hypothesis Hind : mdm_(v, w, t), assume h : Val,
               prove mdm_(v, w, cons(h, t))
           <2>1 hypothesis H1 : mem(v, del(cons(h, t), w)),
                prove mem(v, cons(h, t))
            <3>1 hypothesis H2 : Val!( = )(v, h),
                 prove mem(v, cons(h, t))
                 by property mem_cons hypothesis H2
            <3>2 hypothesis H2 : ~ Val!( = )(v, h),
                               H3 : Val!( = )(w, h),
                 prove mem(v, cons(h, t))
             <4>1 prove ~ isnil(cons(h, t))
                  by property isnil_cons
             <4>2 prove Val!( = )(w, head(cons(h, t)))
                  by property head_cons hypothesis H3
             <4>3 prove mem(v, del(tail(cons(h, t)), w))
                  by definition of del step <4>1, <4>2 hypothesis H1
             <4>4 prove mem(v, del(t, w))
                  by step <4>3 property tail_cons
             <4>f qed by step <4>4 hypothesis Hind definition of mdm_
                      property mem_cons
            <3>3 hypothesis H2 : ~ Val!( = )(v, h),
                               H3 : ~ Val!( = )(w, h),
                 prove mem(v, cons(h, t))
             <4>1 prove ~ isnil(cons(h, t))
                  by property isnil_cons
             <4>2 prove ~ Val!( = )(w, head(cons(h, t)))
                  by property head_cons hypothesis H3
             <4>3 prove mem(v, cons(head(cons(h, t)),
                                    del(tail(cons(h, t)), w)))
                  by definition of del step <4>1, <4>2 hypothesis H1
             <4>4 prove mem(v, cons(h, del(t, w)))
                  by step <4>3 property head_cons, tail_cons
             <4>5 prove mem(v, del(t, w))
                  by step <4>4 property mem_cons hypothesis H2
             <4>f qed by step <4>5 hypothesis Hind definition of mdm_
                      property mem_cons
```

```
            <3>f conclude
           <2>f qed by step <2>1 definition of mdm_
```

This proof is simple but rather long, essentially because we have to explicit a lot of transformations such as $cons(head(cons(h,t)), tail(cons(h,t))) = cons(h,t)$. Remember indeed that `cons`, `head` and `tail` are not yet defined, but are just described by properties. Once these functions implemented, those simplifications can be the result of computations, with shorter proofs; yet having only an axiomatised form of these functions for now, we have to provide some guidance. In this case, a more straightforward approach (jumping directly to an implementation by inductive lists) is likely to be simpler.

A similar approach is chosen for the second subgoal:

```
logical let mde_(v : Val, w : Val, l : Self) =
 mem(v, del(l, w)) -> ~ Val!( = )(v, w) ;

theorem mem_del_eq : all v w : Val, all l : Self, mde_(v, w, l)
proof = <1>1 assume v w : Val,
             prove mde_(v, w, nil)
             by definition of mde_, del property isnil_nil, mem_nil
        <1>2 assume v w : Val, t : Self,
             hypothesis Hind : mde_(v, w, t), assume h : Val,
             prove mde_(v, w, cons(h, t))
         <2>1 hypothesis H1 : mem(v, del(cons(h, t), w)),
              prove ~ Val!( = )(v, w)
          <3>1 hypothesis H2 : Val!( = )(w, h),
               prove ~ Val!( = )(v, w)
           <4>1 prove mem(v, del(t, w))
            <5>1 prove ~ isnil(cons(h, t))
                 by property isnil_cons
            <5>2 prove Val!( = )(w, head(cons(h, t)))
                 by property head_cons hypothesis H2
            <5>3 prove mem(v, del(tail(cons(h, t)), w))
                 by hypothesis H1 definition of del step <5>1, <5>2
            <5>f qed by step <5>3 property tail_cons
           <4>f qed by step <4>1 hypothesis Hind definition of mde_
          <3>2 hypothesis H2 : ~ Val!( = )(w, h),
               prove ~ Val!( = )(v, w)
           <4>1 hypothesis H3 : Val!( = )(v, h),
                prove ~ Val!( = )(v, w)
                by hypothesis H2, H3
                   property Val!eq_symm, Val!eq_tran
           <4>2 hypothesis H3 : ~ Val!( = )(v, h),
                prove ~ Val!( = )(v, w)
            <5>1 prove ~ isnil(cons(h, t))
                 by property isnil_cons
            <5>2 prove ~ Val!( = )(w, head(cons(h, t)))
                 by property head_cons hypothesis H2
            <5>3 prove mem(v, cons(head(cons(h, t)),
                                   del(tail(cons(h, t)), w)))
                 by hypothesis H1 definition of del step <5>1, <5>2
            <5>4 prove mem(v, cons(h, del(t, w)))
                 by step <5>3 property head_cons, tail_cons
            <5>5 prove mem(v, del(t, w))
                 by step <5>4 property mem_cons hypothesis H3
            <5>f qed by step <5>5 hypothesis Hind definition of mde_
           <4>f conclude
          <3>f conclude
         <2>f qed by definition of mde_ step <2>1
        <1>3 assume v w : Val,
             prove mde_(v, w, nil) ->
```

```
                    (all t : Self, mde_(v, w, t) ->
                     all h : Val, mde_(v, w, cons(h, t))) ->
                    all l : Self, mde_(v, w, l)
              assumed (* Standard induction principle *)
        <1>f conclude ;
```

The third subgoal is left as an exercise to the reader; note that we have slightly adapted the statement of *mem_del_inv* to ease the proof by induction:

```
theorem mem_del_inv : all v w : Val, ~ Val!( = )(v, w) ->
                      all l : Self, mem(v, l) -> mem(v, del(l, w))
 proof = assumed (* TODO *) ;
```

Finally we prove the global result

```
proof of mem_del = by property mem_del_mem, mem_del_eq, mem_del_inv
                      definition of mdm_, mde_ ;
```

7.2 Inductive Lists

It is now time to define the species *InductiveList* which refines, or more precisely implements, the species *EnumList* and therefore the species *FiniteList* and *CoList*.

7.2.1 Non Empty Supersets

First, taking into account the comment at the end of Sub. 6.1, we append to the file **superset.fcl** the definition of a new species *SupersetWitness* as follows:

```
species SupersetWitness  =

 inherit Superset ;

 signature witness : Self ;

end ;;
```

The *witness* constant ensures that a *SupersetWitness* is never empty. More specifically, it is used in our case as the default value returned e.g. when the *head* function is applied to the empty list. Do not forget to recompile **superset.fcl** once edited.

7.2.2 A Type for Lists

Back to the file **mylist.fcl**, we now define the representation of the species as being a list ; yet rather than using the predefined *list* type provided in **basics.fcl**, we introduce our own definition for the sake of illustration. Type definitions are allowed in FoCaLize, but only at top level, that is outside a species:

```
type mylist('a) = | FNil | FCons ('a, mylist('a)) ;;
```

Such an inductive definition , called a “sum type”, claims that the type only contains values build from the constructors (surjectivity), and that two values of this type are equal if and only if they are structurally equal (injectivity); well-foundation (finiteness of the constructs) is also ensured.

Tip 9 *[Syntax] Type definitions are only authorised at top-level.*

This is also a polymorphic type: `'a` represents a type variable parametrisation our definition, `mylist(int)` or `mylist(char)` being examples of types obtained by instantiating this variable. Note that the FoCaLize syntax requires sum type constructor identifiers to start by an uppercase character.

Tip 10 *[Syntax] Sum type constructor identifiers have to start with an uppercase character.*

7.2.3 Inductive Lists

Provided this type, it is possible to introduce associated top level definitions and theorems (but not declarations and properties, in the absence of inheritance mechanism). In this tutorial, however, we stick to the species vision and include all the definitions and theorems in a new species. Starting with trivial definitions, the `IndList` species, to be inserted in the file **mylist.fcl**, is as follows:

```
species IndList(Val is SupersetWitness) =

 inherit FiniteList(Val) ;

 representation = mylist(Val) ;

 let nil = #FNil ;
 let cons(h : Val, t : Self) = #FCons(h, t) ;
 let isnil(l : Self) = match l with | #FNil -> true | _ -> false ;
 let head(l : Self) =
  match l with | #FCons(h, _) -> h | _ -> Val!witness ;
 let tail(l : Self) = match l with | #FCons(_, t) -> t | _ -> #FNil ;

end ;;
```

The keyword `representation` is used to associate a concrete datatype to the species; once the representation is defined it cannot be changed at inheritance. Note that we parametrise `mylist` with `Val` (that is its representation).

For the functions `isnil`, `head` and `tail` we use pattern matching (as in OCaml and Coq). In FoCaLize pattern matching has to be complete and without redundancy; in other words, all cases should be addressed, and patterns included in previous patterns are not allowed.

7.2.4 Higher-Order Proofs

We have now a definition for all the declared methods, but we are still lacking the proofs of the properties. Let's first address the higher-order `induction` property; as mentioned, Zenon is not able to tackle it. The most simple approach is to admit this property as an axiom:

```
 proof of finite =  assumed (* Requires a Coq proof *) ;
```

The keyword `assumed` is the logical backdoor in FoCaLize: the proof is not provided, but the property is accepted as true. It is recommended to associate, in such a case, a comment justifying why no proof is given. That is, it is possible to introduce axioms in a FoCaLize development, but those axioms are traced and documented.

The alternative approach is to provide the Coq proof. It is beyond the scope of this tutorial to describe how to build such a proof; ideally the use of Coq scripts should be limited to the standard library of FoCaLize, and only expert users should try to use Coq. We just provide in the rest of this paragraph a sketch of the process, that can be skipped by most readers.

The first step is to identify the relevant properties and definitions to derive the proof – exactly as when Zenon is used. This is required to make these definitions and properties visible from the proof context. It is then possible to build a template for the Coq proof, as follows:

```
proof of finite =  coq proof definition of nil, cons
                   {* *} ;
```

The keywords *coq proof* indicates that the proof is attached and has to be checked directly by Coq, without assistance of Zenon. It is followed by the tips, in this case we need to make sure that the definition of the methods *nil* and *cons* are visible, as we will in fact reason on the inductive definition of *mylist* and on the constructors *Nil* and *Cons*. These tips will not be used by Coq but serves FoCaLize to be aware that the following proof (that it cannot analyse) will depend on the listed functions, properties and theorems. During dependencies computation, this allows to invalidate the proof in children species who will have redefined some of these listed dependencies (or inductively, dependencies on these dependencies). The Coq script is then expected to follow, between *{** and **}*; here it is empty, just marking a hole that we will fill later using Coq.

It is then possible to compile the file with the command **focalizec mylist.fcl**; the compilation of course fails when Coq checks the proof file **mylist.v**. Using Coq (and providing the path to the FoCaLize and Zenon libraries, that are prompted at compilation time) the proof can then be completed by hand, and copied into the FoCaLize source file:

```
proof of finite =  coq proof definition of nil, cons
                   {* Proof.
                       intros.
                       unfold abst_cons, abst_nil, cons, nil in *.
                       induction l.
                        trivial.
                       apply H0; apply IHl.
                      Qed. *} ;
```

A new compilation of **mylist.fcl** now succeeds.

7.2.5 Proofs on Inductive Types

Other properties related to inductive definitions can be done in Coq, or can be assumed. We choose for this tutorial the second option:

```
proof of isnil_nil = assumed (* TODO *) ;

proof of isnil_cons = assumed (* TODO *) ;

proof of head_cons = assumed (* TODO *) ;

proof of tail_cons = assumed (* TODO *) ;

proof of list_dec = assumed (* TODO *) ;
```

Note however that ZENON is currently evolving to provide some support for such results, by allowing for a new form of tip, *type*, when the very definition of a type provides the required information. The expected use is as follows:

```
proof of isnil_nil = by type mylist definition of nil, isnil ;
```

It can be experimentally tested, but only at top level for now.

7.2.6 Structural Recursion

In the species *IndList*, we redefine the methods *mem* and *del* as follows:

```
let mem(v : Val, l : Self) =
 let recstruct mem_(l : Self) =
  match l with
  | #FNil -> false
  | #FCons(h, t) -> if Val!( = )(v, h) then true else mem_(t)
 in mem_(l) ;

proof of mem_nil = assumed (* TODO *) ;

proof of mem_cons = assumed (* TODO *) ;

let recstruct del(l : Self, v : Val) =
 match l with
 | #FNil -> #FNil
 | #FCons(h, t) -> if Val!( = )(v, h)
                   then del(t, v)
                   else #FCons(h, del(t, v)) ;

proof of mem_del =  assumed (* TODO *) ;
```

In this case, we are using the keyword *recstruct* to indicate a structurally recursive function, that is recursive calls on subterms of the parameter – termination then being trivially true. For such functions, however, the decreasing parameter has to be the first one; that's why we are using a *let in* construct, as the declaration of *mem* puts the list as the second parameter.

Such redefinitions, of course, cause the deletion of the associated proofs for *mem_nil*, *mem_cons*, *mem_del*. Yet the use of pattern matching on an inductive type leads to simpler proofs of these properties. For this tutorial, we just admit these properties.

Note that the keyword *recstruct* may become deprecated in future versions of FOCALIZE, as it is an *ad hoc* adaptation to deal with a simple form of recursion.

8 A Complete Implementation

We are now at the final stage of the development, in which we will define collections. A *collection* is a frozen implementation obtained by abstracting the concrete representation of a *complete species*, that is a species which has a concrete datatype representation, a definition for every signature and a proof for every property.

Provided what has been developed up to now, there are still two approaches to implement subsets as lists. In both case, we need a collection implementing a complete species, but this species can either:

- inherits from both *ExtSubset* and *IndList*.

- inherits from *ExtSubset* and be parametrised by *IndList*.

These two approaches have similarities – for example, in both cases the membership method << of subsets is implemented as the membership method *mem* of lists – but they represent very different intentions.

In the first case, using multiple inheritance, we claim that any subset is also a list. So the methods of both interfaces can be used to manipulate these values; a set can be created with two elements, then it can be considered as a list in which we exchange the position of these elements.

In the second case, using parametrisation, we use lists to represent subsets, but emphasising that lists and subsets are different in nature: manipulating a subset can only be done using the methods of the subset interface – and it is possible to ensure that some "dangerous" methods defined for lists are not used when dealing with subsets.

We discuss in the rest of this section of the strategy to build an executable program using the implementation of subsets with the second approach, parametrisation by lists. As a collection is never parametrised, and always implement one and only one complete species whose parameters are instantiated by collections, we have to follow a strict discipline: to create a collection from a given species, we have to create a collection for its parameters. That is creating a subset collection requires creating a list collection, which itself requires creating a superset collection.

We edit a new file, named **main.fcl**, and include the required information with the *use* and *open* directives:

```
use "basics" ;;
open "superset" ;;
open "subset" ;;
open "mylist" ;;
```

The rest of the section deals with a superset collection, a list collection and a subset collection, that are implementing complete species.

8.1 Integers

8.1.1 Adding Inputs and Outputs to Supersets

We can implement collections with the currently defined interfaces, unfortunately this would not be very demonstrative in the absence of HMI. As a first mandatory step toward a program we therefore create a new species with methods to import or export values:

```
species PrintParseSuperset =

 inherit SupersetWitness ;

 signature print : Self -> basics#unit ;

 signature parse : basics#string -> Self ;

end ;;
```

Of course, it is possible to add, at any point of the development, a *print* and a *parse* methods to any species – as illustrated later in this section for subsets. But the definition of a *PrintParseSpecies* interface is required. Indeed, to implement

for example the *print* method for a subset, one has to be able to rely on the *print* method of its superset parameter, that is to ensure that such method exists in the interface.

8.1.2 A Complete Integer Species

Provided the specification *PrintParseSuperset*, we develop a complete species representing machine integers:

```
species Int =

 inherit PrintParseSuperset ;

 representation = basics#int ;

 let ( = )(x : Self, y : Self) = basics#( = )(x, y) ;
 proof of eq_refl = by definition of ( = ) ;
 proof of eq_symm = by definition of ( = ) ;
 proof of eq_tran = by definition of ( = ) ;

 let witness = 0 ;

 let print(s : Self) = basics#print_int(s) ;

 let parse(s : basics#string) = basics#int_of_string(s) ;

end ;;
```

In this species, the representation is defined, all the declared methods are defined[15] and all the properties are proved.

8.1.3 An Integer Collection

The species *Int* being complete, it can be transformed into a collection:

```
collection Int_Coll = implement Int; end ;;
```

Int now denotes a form of abstract data type, whose representation is hidden. It is not possible to inherit from *Int*, or to modify otherwise its structure; of course, it is still possible to use *Int* as a parameter.

Tip 11 (Implementation) *Collections are the final entities of* FoCaLize *developments and represent executable implementations. They cannot be parametrised or inherited from, but they can be used as parameters for other species.*

8.2 Lists if Integers

As we want to create a collection from the parametrised species *ExtSubset*, we need first to create a collection for the parameter itself, that is a collection representing the species *IndList*:

```
collection IntList_Coll = implement IndList(Int_Coll) ; end ;;
```

[15] For = it is possible to use the *int* equality, denoted *=0x* in FoCaLize, but Zenon does not know much about it and cannot prove *eq_refl*, *eq_symm* and *eq_tran*.

Remember that a collection cannot be parametrised, and that it has to implement a species whose all formal parameters are instantiated by a collection, as it is the case here.

Tip 12 (Implementation) *A collection can only implements a complete species whose parameters are instantiated by collections.*

8.3 Subsets

8.3.1 A Complete Subset Species

Having chosen to implement subsets as a species parametrised by lists, it is straightforward to complete *ExtSubset* in a new species *ListSubset*. We just provide some proofs, assuming most of the properties, and we add a *print* method for subsets:

```
species ListSubset(Val is PrintParseSuperset, Sup is IndList(Val)) =

 inherit ExtSubset(Val) ;

 representation = Sup ;

 let ( << )(v : Val, s : Self) = Sup!mem(v, s) ;

 let empty = Sup!nil ;
 proof of mem_empty = by definition of ( << ), empty
                         property Sup!mem_nil ;

 let ( + )(s : Self, v : Val) =
  if v << s then s else Sup!cons(v, s) ;

 proof of mem_insert =
  <1>1 assume v1  v2 : Val, s : Self, hypothesis H1 : v1 << s + v2,
       prove Val!( = )(v1, v2) \/ v1 << s
       by definition of ( << ), ( + )
          property Sup!mem_cons hypothesis H1
  <1>2 assume v1 v2 : Val, s : Self,
       hypothesis H1 : Val!( = )(v1, v2),
       prove v1 << s + v2
       assumed (* TODO *)
  <1>3 assume v1 v2 : Val, s : Self, hypothesis H1 : v1 << s,
       prove v1 << s + v2
       assumed (* TODO *)
  <1>f conclude ;

 let ( - )(s : Self, v : Val) = Sup!del(s, v) ;

 proof of mem_remove = assumed (* TODO *) ;

 proof of mem_congr = assumed (* TODO *) ;

 let rec ( <: )(s1 : Self, s2 : Self) =
  if Sup!isnil(s1)
  then true
  else if Sup!head(s1) << s2
       then Sup!tail(s1) <: s2
       else false ;
 (* TODO : Termination Proof *)

 proof of mem_incl = assumed (* TODO *) ;
```

```
 let rec ( = )(s1 : Self, s2 : Self) =
  if Sup!isnil(s1)
  then Sup!isnil(s2)
  else let h1 = Sup!head(s1) and h2 = Sup!head(s2) in
       if basics#( && )(h1 << s2, h2 << s1)
       then ((s1 - h1) - h2) = ((s2 - h1) - h2)
       else false ;
 (* TODO : Termination Proof *)

 proof of eq_incl = assumed (* TODO *) ;

 let print(s : Self) =
  let x = basics#print_string("{") in
  let rec print_(s : Self)=
   if Sup!isnil(s)
   then basics#print_string("}")
   else let y = Val!print(Sup!head(s)) in print_(s - Sup!head(s))
  in print_(s) ;

end ;;
```

8.3.2 A Subset Collection

We can finally implement *ListSubset* in a collection *IntSubset_Coll* as follows:

```
collection IntSubset_Coll =

 implement ListSubset(Int_Coll, IntList_Coll) ;

end ;;
```

8.4 Using Subsets

8.4.1 Top Level Use

We have now an implementation of finite subsets of integers, whose concrete representation is hidden. To use this implementation, we can for example use the OCaml files produced by FoCaLize as proved libraries for OCaml programs. But it is also possible to use directly the various methods of any collection from the FoCaLize top level, as follows:

```
basics#print_string("\n") ;;
basics#print_string("Subsets of Integers :\n") ;;
basics#print_string("---------------------\n") ;;

let subset1 = IntSubset_Coll!empty ;;
basics#print_string("Creating empty set : ") ;;
IntSubset_Coll!print(subset1) ;;
basics#print_string("\n") ;;

let subset2 = IntSubset_Coll!( + )(subset1, Int_Coll!parse("1")) ;;
basics#print_string("Inserting 1 : ") ;;
IntSubset_Coll!print(subset2) ;;
basics#print_string("\n") ;;

let subset3 = IntSubset_Coll!( + )(subset2, Int_Coll!parse("2")) ;;
basics#print_string("Inserting 2 : ") ;;
IntSubset_Coll!print(subset3) ;;
```

```
basics#print_string("\n") ;;

let subset4 = IntSubset_Coll!( + )(subset3, Int_Coll!parse("3")) ;;
basics#print_string("Inserting 3 : ") ;;
IntSubset_Coll!print(subset4) ;;
basics#print_string("\n") ;;

let subset5 = IntSubset_Coll!( + )(subset4, Int_Coll!parse("2")) ;;
basics#print_string("Inserting 2 : ") ;;
IntSubset_Coll!print(subset5) ;;
basics#print_string("\n") ;;

let subset6 = IntSubset_Coll!( - )(subset5, Int_Coll!parse("2")) ;;
basics#print_string("Removing 2 : ") ;;
IntSubset_Coll!print(subset6) ;;
basics#print_string("\n") ;;

let subset7 = IntSubset_Coll!( - )(subset6, Int_Coll!parse("3")) ;;
basics#print_string("Removing 3 : ") ;;
IntSubset_Coll!print(subset7) ;;
basics#print_string("\n") ;;
```

8.4.2 Producing an Executable

Once all FoCaLize files have been compiled by **focalizec**, there is still a final stage of compilation required to produce an executable, an OCaml compilation, with the following command[16]:

ocamlc -I /focalize/focalizec/src/stdlib/
-o main
ml_builtins.cmo basics.cmo
superset.cmo subset.cmo mylist.cmo main.cmo

This produces an executable file named **main**, which can be executed:

Creating empty set: {}
Inserting 1: {1}
Inserting 2: {21}
Inserting 3: {321}
Inserting 2: {321}
Removing 2: {31}
Removing 3: {1}

9 Some Remarks

We discuss here more theoretical aspects, considering extensions and alternatives to our development and their consequences.

9.1 Over-specifications

We have considered, at the end of Sub. 5.1, the opportunity to declare additional methods and properties for the species *Subset*. Our concern was not to enrich the specification of *Subset* to enforces inhabitants of the species to be finite, but to provide more features – recognising that it may causes unnecessary burden for the developer if these features are not used.

[16] Modulo the path for the **stdlib**, and the use of the correct version of the OCaml compiler.

For example, one could ask for the union, the intersection or the complement operations. If the two formers seems to be straightforward, the situation of the latter if more complex. Indeed the complement, for any subset S, returns the subset of elements of the superset not belonging to S. This is a perfectly valid requirement, and as we will see later in this section, we can even implement directly the species *Subset* enriched with this operation. Yet the complement operation would also be part of all the other inheriting species, including *ExtSubset*, *ListSubset*. This would clearly cause some difficulties, preventing the parameter *Val* to be infinite, as in such a case for S a finite subset, the complement of S would not be finite. In other words, provided an infinite superset, complement would not be an internal operation on finite subsets.

It is not always easy to identify well in advance this type of traps, and a reengineering of the inheritance tree can be required when facing similar problems during development. Yet to make such difficulties less likely to happen as well as to ease possible modifications of the species structure, a good recommendation is to multiply the inheritance steps and branches, introducing very gradually new methods and properties. Remember that early branching is not a problem in a system such as FoCaLize, as it supports multiple inheritance: having for example species *ExtSubset* and *SubsetComplement* (inheriting from *Subset* and enriched with the complement operation) does not prevent a later definition of a species *CoSubset* inheriting from both if we are able to define an appropriate concrete representation. Although deep and multiple inheritance can seem difficult to understand, we must not forget that the compiler is able to provide traces of methods provenance which greatly help understanding in case of a complex development.

9.2 Closure Reasoning's

It is worth mentioning that a species defines an interface which can be completed later, therefore closure reasoning does not apply for such an interface.

Taking the example of the species *Subset*, we only provide a few methods to build subsets: the empty subset, the insertion and the removal of an element. Using only these methods it is not possible to build an infinite subset. But that does not mean that the species *Subset* only describes finite subsets – nothing prevent the developer to introduce through inheritance more powerful methods such as a complement operation. The fact that the species *Subset* only requires methods building finite subsets actually ensures that it is indeed valid as an ancestor for the species representing finite subsets.

For the same reason, species invariants have to be handled with care. It is possible to write a fully defined species, with methods such that all returned values of the species are of a specific form (for example sorted lists). It is however important to note that inheritance may, either by creation of new functions or by redefinition of existing ones, break such invariants. The only mechanism provided in FoCaLize for preventing such modification is to freeze a complete species by transforming it into a collection.

9.3 Observability Considerations

We consider here the creation of a new species inheriting from *ExtSubset*, providing a choice operator. This is a standard notion in set theory: the choice

operator returns a value belonging to a subset, provided this subset is not empty. One of the interest of such an operator in our development is to define a method for enumerating all elements of a subset, as indicated by the theorem *enumerate*:

```
species ExtSubsetChc(Val is Superset) =

 inherit ExtSubset(Val) ;

 signature chc : Self -> Val ;
 property mem_chc : all s : Self, ~(s = empty) -> chc(s) << s ;
 theorem enumerate : all s : Self,
                     s = empty \/ s = (s - chc(s)) + chc(s)
  proof = by property mem_choice, remove_insert ;

end;;
```

The choice operator has interesting properties. Indeed, it is able to extract a value from any non empty subset, but we have no indication about which value will be returned provided the subset passed as a parameter contains at least two of them. In fact, whereas the choice operator has to be implemented as a FoCaLize function at some point in the development, it may not be a function in the sense of set theory. Indeed, provided S_1 and S_2 two sets that are extensionally equal, *i.e.* containing the same elements, nothing enforces to have $chc(S_1) = chc(S_2)$. Of course, we can prevent those strange behaviours by adding a property requiring extensional equality to be a congruence w.r.t. *chc*:

```
 property chc_congr : all s1 s2 : Self,
                      s1 = s2 -> Val!( = )(chc(s1), chc(s2)) ;
```

But this deserves additional consideration, as it may be inadequate. Indeed, any implementation of our subset species has to be parametrised by a collection having an interface compatible with *Superset*. That is, values in our subsets can be of any sort, as soon as we can define an equivalence relation (representing a form of equality). The point is that even if other operations are available for the provided collection, such as for example a comparison, they will not be visible (usable) from the species *ExtSubsetChc*; as a consequence, it may not be possible to satisfy the property *chc_congr*.

Consider an implementation of *ExtSubsetChc* using lists: thanks to `=`, it is possible to build injective lists (no repetition of elements), but without a comparison operator there is no way to sort such lists. So for example the set $\{1, 2\}$ has two possible implementations, $cons(1, cons(2, nil))$ and $cons(2, cons(1, nil))$. Whereas we can distinguish them using the structural equality *basics#(=)*, there is no way to characterise a good one and a bad one. If we now try to implement *chc*, we have to exhibit a function that provided a non-empty list, returns an element of this list. Remember that beyond an equality, we have no operation on values, that is for example we cannot code *chc* such that it always returns the smallest element of the list. The only criteria which is available is the position of the value in the list, and *chc* can for example return the head value. This definition complies with *mem_chc* but not with *chc_congr*.

It is possible to provide an implementation of *ExtSubsetChoice* with the property *chc_congr*, but we need to slightly make our inheritance tree more complex. For example, we can define a species *OrderedSuperset* inheriting from *Superset*, with a total order between values, and a species *NormalisedSubset* as follows:

```
species NormalisedSubset(Val is OrderedSuperset) =

 inherits ExtSubsetChc(Val) ;

 property eq_congr : all s1 s2 : Self,
                     s1 = s2 -> basics#( = )(s1, s2) ;

end ;;
```

With this approach, any collection implementing *OrderedSubset* of course implements *ExtSubsetChc* as well, but is parametrised by a collection providing a total order; using this order, a normal form for subsets as well as a *chc* operator can be defined, ensuring that the property *chc_congr* is indeed satisfied, for example returning the smallest element of the subset.

9.4 Functional Representations

For the sake of illustration, we consider in this subsection a totally different implementation of subsets based on a functional representation. Our intent is not to advice to use such type of code – which is generally considered as inefficient – but rather to emphasise the freedom offered to a developer facing an abstract specification, provided this specification is appropriate.

As mentioned in Sub. 5.1, a subset is characterised by the values it contains, in other words a subset is characterised by its membership function. It is therefore possible to represent a subset by this very function, FoCaLize being a functional (higher order) language. One of the interest of this representation is that we can represent infinite subsets: the function deciding if a natural value is even or not defines the subset of even values. The complement operation is also naturally supported, the difficulties identified in Sub. 9.1 being irrelevant here. The description is straightforward:

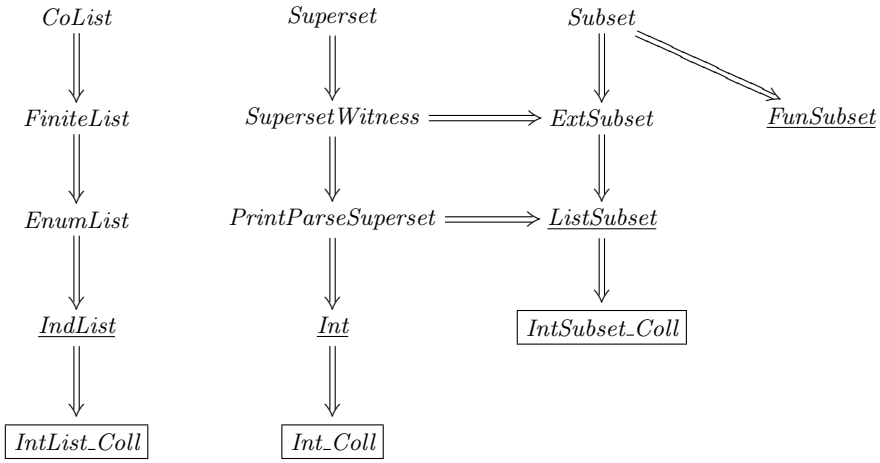
```
species SubsetFun(Val is Superset) =

 inherit Subset(Val) ;

 representation = Val -> basics#bool ;

 let ( << )(v : Val, s : Self) = s(v) ;

 let empty = let empty_in(v : Val) = false in empty_in ;
 proof of mem_empty = assumed (* TODO *) ;

 let ( + )(s : Self, v : Val) =
  if v << s
  then s
  else let inner(w : Val) = if Val!( = )(w, v) then true else w << s
       in inner ;
 proof of mem_insert = assumed (* TODO *) ;

 let ( - )(s : Self, v : Val) =
  if v << s
  then let inner(w : Val) = if Val!( = )(w, v) then false else w << s
       in inner
  else s;
 proof of mem_remove = assumed (* TODO *) ;

 let comp(s : Self) =
  let inner(v : Val) = basics#( ~~ )(v << s) in inner ;
```

```
end ;;
```

Being higher order this encoding is not supported by Zenon, so the properties have to be assumed or proved in Coq; yet this is just a technical concern. More fundamentally, *SubsetFun* is not an extensional representation – it is clearly not compatible with the interface of *ExtSubset*. Inclusion or equality between subsets cannot be implemented in the general case. It is also interesting to note that the *choice* operator, described in Sub. 9.3, is not implementable as well. Finally, a property such as *mem_congr*, defined in Par. 5.2.3, is true but not provable because once we have represented a subset by a function, we cannot later analyse this function to check its structure.

A Inheritance Graph

Inheritance in our development is as follows:



If B inherits of A then an arrow is drawn from A to B. Complete species are underlined, and collections are boxed. Note that parametrisation is not represented here to avoid over-complexification.

We have slightly amended the inheritance relation, compared to what is described in this tutorial. For example, the species `ExtSubset` inherits not from the species `Superset` as in Par. 5.2.2, but from the species `SupersetWitness` introduced in Par. 7.2.1. Indeed, we have noted that whatever his parameters is, the species `Superset` is never empty, something characteristic of the species `SupersetWitness`. This requires of course to add the definition `let witness = empty`. Similarly, the species `ListSubset` now inherits from the species `PrintParseSuperset`; it indeed offers a `print` method, and the `parse` method can be implemented for example by the stub `let parse(s : basics#string) = empty`.

Once fine tuned, our development is more consistent and allows for easy extensions. This is illustrated in the final code given at annex B. It includes a new collection implementing subsets of subsets of integers at a minimal cost (2 collections, 6 lines of code); executing this program, we get:

Subsets of Integers :	**Subsets of Subsets of Integers :**
Creating empty set : {}	**Creating empty set : {}**
Inserting 1 : {1}	**Inserting {} : {{}}**
Inserting 2 : {21}	**Inserting {1} : {{1}{}}**
Inserting 3 : {321}	**Inserting {21} : {{21}{1}{}}**
Inserting 2 : {321}	**Removing {1} : {{21}{}}**
Removing 2 : {31}	**Creating {12} : {12}**
Removing 3 : {1}	**Inserting {12} : {{21}{}}**

B Full Sources

B.1 superset.fcl

```
use "basics" ;;

species Superset =
 signature ( = ) : Self -> Self -> basics#bool ;
 property eq_refl : all x : Self, x = x ;
 property eq_symm : all x y : Self, x = y -> y = x ;
 property eq_tran : all x y z : Self, x = y -> y = z -> x = z ;
end ;;

species SupersetWitness  =
 inherit Superset ;
 signature witness : Self ;
end ;;

species PrintParseSuperset =
 inherit SupersetWitness ;
 signature print : Self -> basics#unit ;
 signature parse : basics#string -> Self ;
end ;;
```

B.2 subset.fcl

```
use "basics" ;;
open "superset" ;;

species Subset(Val is Superset) =
 signature ( << ) : Val -> Self -> basics#bool ;
 signature empty : Self ;
 property mem_empty : all v : Val, ~(v << empty) ;
 signature ( + ) : Self -> Val -> Self ;
 property mem_insert : all v1 v2 : Val, all s : Self,
                       v1 << s + v2 <-> (Val!( = )(v1, v2) \/ v1 << s) ;
 signature ( - ) : Self -> Val -> Self ;
 property mem_remove : all v1 v2 : Val, all s : Self,
                       v1 << s - v2 <-> (~(Val!( = )(v1, v2)) /\ v1 << s) ;
end ;;

species ExtSubset(Val is Superset) =
 inherit SupersetWitness, Subset(Val);
 property mem_congr : all v1 v2 : Val, Val!( = )(v1, v2) ->
                      (all s : Self, (v1 << s) <-> (v2 << s)) ;
 signature ( <: ) : Self -> Self -> basics#bool ;
 property mem_incl : all s1 s2 : Self,
                     s1 <: s2 <-> all v : Val, v << s1 -> v << s2 ;
 theorem incl_refl : all s : Self, s <: s
  proof = by property mem_incl ;
 theorem incl_tran : all s1 s2 s3 : Self, s1 <: s2 -> s2 <: s3 -> s1 <: s3
  proof = by property mem_incl ;
 property eq_incl : all s1 s2 : Self, s1 = s2 <-> s1 <: s2 /\ s2 <: s1 ;
 proof of eq_refl = by property eq_incl, incl_refl ;
 proof of eq_symm = by property eq_incl ;
 proof of eq_tran = by property eq_incl, incl_tran ;
 theorem mem_eq : all s1 s2 : Self, s1 = s2 <->
                  (all v : Val, v << s1 <-> v << s2)
  proof = by property eq_incl, mem_incl ;
 theorem incl_empty : all s : Self, empty <: s
  proof = by property mem_incl, mem_empty ;
 theorem incl_insert : all s : Self, all v : Val, s <: s + v
  proof = by property mem_insert, mem_incl ;
 theorem incl_remove : all s : Self, all v : Val, s - v <: s
  proof = by property mem_remove, mem_incl ;
 theorem incl_insert_mem : all s : Self, all v : Val,
                           v << s -> s + v <: s
  proof = by property mem_congr, mem_insert, mem_incl ;
 theorem incl_remove_mem : all s : Self, all v : Val,
```

```
                          ~(v << s) -> s <: s - v
  proof = <1>1 assume s : Self, v : Val, hypothesis Hv : ~(v << s),
               prove s <: s - v
           <2>1 assume w : Val, hypothesis Hw : w << s,
                prove w << s - v
            <3>1 prove ~(Val!( = )(w, v)) /\ w << s
             <4>1 prove ~(Val!( = )(w, v))
                  by property mem_congr hypothesis Hv, Hw
             <4>2 prove w << s
                  by hypothesis Hw
             <4>f conclude
            <3>f qed by property mem_remove step <3>1
           <2>f qed by property mem_incl step <2>1
          <1>f conclude ;
 theorem remove_insert : all s : Self, all v : Val,
                         v << s -> s = (s - v) + v
  proof = <1>1 assume s : Self, v : Val, hypothesis Hv : v << s,
               prove s = (s - v) + v
           <2>1 assume w : Val, hypothesis Hw : w << s,
                prove w << (s - v) + v
            <3>1 prove ~(Val!( = )(w, v)) -> w << s - v
                 by property mem_remove hypothesis Hw
            <3>f qed by property mem_insert step <3>1
           <2>2 assume w : Val, hypothesis Hw : w << (s - v) + v,
                prove w << s
            <3>1 prove Val!( = )(w, v) -> w << s
                 by property mem_congr hypothesis Hv
            <3>2 prove w << s - v -> w << s
                 by property mem_remove
            <3>f qed by property mem_insert hypothesis Hw step <3>1, <3>2
           <2>f qed by property eq_incl, mem_incl step <2>1, <2>2
          <1>f qed by step <1>1 ;
 let witness = empty;
end ;;

species SubsetFun(Val is Superset) =
 inherit Subset(Val) ;
 representation = Val -> basics#bool ;
 let ( << )(v : Val, s : Self) = s(v) ;
 let empty = let empty_in(v : Val) = false in empty_in ;
 proof of mem_empty = assumed (* TODO *) ;
 let ( + )(s : Self, v : Val) =
  if v << s
  then s
  else let inner(w : Val) = if Val!( = )(w, v) then true else w << s
       in inner ;
 proof of mem_insert = assumed (* TODO *) ;
 let ( - )(s : Self, v : Val) =
  if v << s
  then let inner(w : Val) = if Val!( = )(w, v) then false else w << s
       in inner
  else s;
 proof of mem_remove = assumed (* TODO *) ;
 let comp(s : Self) = let inner(v : Val) = basics#( ~~ )(v << s) in inner ;
end ;;
```

B.3 mylist.fcl

```
use "basics" ;;
open "superset" ;;

species CoList(Val is Superset) =
 signature nil : Self ;
 signature cons : Val -> Self -> Self ;
 signature isnil : Self -> basics#bool ;
 signature head : Self -> Val ;
 signature tail : Self -> Self ;
 property isnil_nil : all l : Self, isnil(l) <-> basics#( = )(l, nil) ;
 property isnil_cons : all v : Val, all l : Self, ~ isnil(cons(v, l)) ;
 property head_cons : all v : Val, all l : Self,
                      basics#( = )(head(cons(v, l)), v) ;
```

```
 property tail_cons : all v : Val, all l : Self,
                      basics#( = )(tail(cons(v, l)), l) ;
 property list_dec : all l : Self,
                     ~ isnil(l) -> basics#( = )(l, cons(head(l), tail(l))) ;
 theorem cons_left : all l1 l2 : Self, all v1 v2 : Val,
                     basics#( = )(cons(v1, l1), cons(v2, l2)) ->
                     basics#( = )(v1, v2)
  proof = <1>1 assume t1 t2 : Self, h1 h2 : Val,
              hypothesis Heq : basics#( = )(cons(h1, t1), cons(h2, t2)),
              prove basics#( = )(h1, h2)
          <2>1 prove basics#( = )(head(cons(h1, t1)), head(cons(h2, t2)))
               by hypothesis Heq
          <2>2 prove basics#( = )(h1, head(cons(h2, t2)))
               by step <2>1 property head_cons
          <2>f qed by step <2>2 property head_cons
         <1>f qed by step <1>1 ;
 theorem cons_right : all l1 l2 : Self, all v1 v2 : Val,
                      basics#( = )(cons(v1, l1), cons(v2, l2)) ->
                      basics#( = )(l1, l2)
  proof = <1>1 assume l1 l2 : Self, v1 v2 : Val,
              hypothesis Heq : basics#( = )(cons(v1, l1), cons(v2, l2)),
              prove basics#( = )(l1, l2)
          <2>1 prove basics#( = )(tail(cons(v1, l1)), tail(cons(v2, l2)))
               by hypothesis Heq
          <2>2 prove basics#( = )(l1, tail(cons(v2, l2)))
               by step <2>1 property tail_cons
          <2>f qed by step <2>2 property tail_cons
         <1>f qed by step <1>1 ;
end ;;

species FiniteList(Val is Superset) =
 inherit CoList(Val) ;
 property finite :
  all f : (Self -> basics#bool),
  f(nil) ->
  (all l : Self, f(l) -> all v : Val, f(cons(v, l))) ->
  all l : Self, f(l) ;
 signature mem : Val -> Self -> basics#bool ;
 property mem_nil : all v : Val, ~ mem(v, nil) ;
 property mem_cons : all v h : Val, all t : Self,
                     mem(v, cons(h, t)) <-> Val!( = )(v, h) \/ mem(v, t) ;
 signature del : Self -> Val -> Self ;
 property mem_del : all v w : Val, all l : Self,
                    mem(v, del(l, w)) <-> ~ Val!( = )(v, w) /\ mem(v, l) ;
end ;;

species EnumList(Val is Superset) =
 inherit FiniteList(Val) ;
 let rec mem(v : Val, l : Self) =
  if isnil(l)
  then false
  else (if Val!( = )(v, head(l)) then true else mem(v, tail(l))) ;
 (* TODO : Termination Proof *)
 proof of mem_nil = by definition of mem property isnil_nil ;
 proof of mem_cons =
  <1>1 assume v h : Val, t : Self, hypothesis H : Val!( = )(v, h),
       prove mem(v, cons(h, t))
   <2>1 prove Val!( = )(v, head(cons(h, t)))
        by property head_cons hypothesis H
   <2>f qed by definition of mem property isnil_cons step <2>1
  <1>2 assume v h : Val, t : Self, hypothesis H : mem(v, t),
       prove mem(v, cons(h, t))
   <2>1 prove mem(v, tail(cons(h, t)))
        by property tail_cons hypothesis H
   <2>f qed by step <2>1 definition of mem property isnil_cons
  <1>3 assume v h : Val, t : Self, hypothesis H : mem(v, cons(h, t)),
       prove Val!( = )(v, h) \/ mem(v, t)
   <2>1 hypothesis H2 : ~ Val!( = )(v, h),
        prove mem(v, tail(cons(h, t)))
        by hypothesis H, H2 definition of mem property isnil_cons, head_cons
   <2>f qed by step <2>1 property tail_cons
  <1>f conclude ;
 let rec del(l : Self, v : Val) =
  if isnil(l)
```

```
 then nil
 else if Val!( = )(v, head(l))
      then del(tail(l), v)
      else cons(head(l) , del(tail(l), v)) ;
(* TODO : Termination Proof *)
logical let mdm_(v : Val, w : Val, l : Self) =
 mem(v, del(l, w)) -> mem(v, l) ;
theorem mem_del_mem : all v w : Val, all l : Self, mdm_(v, w, l)
 proof = <1>1 assume v w : Val,
             prove mdm_(v, w, nil)
             by definition of mdm_, del property isnil_nil
         <1>2 assume v w : Val, t : Self, hypothesis Hind : mdm_(v, w, t),
             assume h : Val,
             prove mdm_(v, w, cons(h, t))
          <2>1 hypothesis H1 : mem(v, del(cons(h, t), w)),
               prove mem(v, cons(h, t))
           <3>1 hypothesis H2 : Val!( = )(v, h),
                prove mem(v, cons(h, t))
                by property mem_cons hypothesis H2
           <3>2 hypothesis H2 : ~ Val!( = )(v, h), H3 : Val!( = )(w, h),
                prove mem(v, cons(h, t))
            <4>1 prove ~ isnil(cons(h, t))
                 by property isnil_cons
            <4>2 prove Val!( = )(w, head(cons(h, t)))
                 by property head_cons hypothesis H3
            <4>3 prove mem(v, del(tail(cons(h, t)), w))
                 by definition of del step <4>1, <4>2 hypothesis H1
            <4>4 prove mem(v, del(t, w))
                 by step <4>3 property tail_cons
            <4>f qed by step <4>4 hypothesis Hind definition of mdm_
                        property mem_cons
           <3>3 hypothesis H2 : ~ Val!( = )(v, h), H3 : ~ Val!( = )(w, h),
                prove mem(v, cons(h, t))
            <4>1 prove ~ isnil(cons(h, t))
                 by property isnil_cons
            <4>2 prove ~ Val!( = )(w, head(cons(h, t)))
                 by property head_cons hypothesis H3
            <4>3 prove mem(v, cons(head(cons(h, t)),
                                   del(tail(cons(h, t)), w)))
                 by definition of del step <4>1, <4>2 hypothesis H1
            <4>4 prove mem(v, cons(h, del(t, w)))
                 by step <4>3 property head_cons, tail_cons
            <4>5 prove mem(v, del(t, w))
                 by step <4>4 property mem_cons hypothesis H2
            <4>f qed by step <4>5 hypothesis Hind definition of mdm_
                        property mem_cons
           <3>f conclude
          <2>f qed by step <2>1 definition of mdm_
         <1>3 assume v w : Val,
              prove mdm_(v, w, nil)->
                    (all t : Self, mdm_(v, w, t) ->
                     all h : Val, mdm_(v, w, cons(h, t))) ->
                    all l : Self, mdm_(v, w, l)
              assumed (* Standard induction principle *)
         <1>f conclude ;
logical let mde_(v : Val, w : Val, l : Self) =
 mem(v, del(l, w)) -> ~ Val!( = )(v, w) ;
theorem mem_del_eq : all v w : Val, all l : Self, mde_(v, w, l)
proof = <1>1 assume v w : Val,
             prove mde_(v, w, nil)
             by definition of mde_, del property isnil_nil, mem_nil
         <1>2 assume v w : Val, t : Self, hypothesis Hind : mde_(v, w, t),
             assume h : Val,
             prove mde_(v, w, cons(h, t))
          <2>1 hypothesis H1 : mem(v, del(cons(h, t), w)),
               prove ~ Val!( = )(v, w)
           <3>1 hypothesis H2 : Val!( = )(w, h),
                prove ~ Val!( = )(v, w)
            <4>1 prove mem(v, del(t, w))
             <5>1 prove ~ isnil(cons(h, t))
                  by property isnil_cons
             <5>2 prove Val!( = )(w, head(cons(h, t)))
                  by property head_cons hypothesis H2
             <5>3 prove mem(v, del(tail(cons(h, t)), w))
```

```
                by hypothesis H1 definition of del step <5>1, <5>2
            <5>f qed by step <5>3 property tail_cons
          <4>f qed by step <4>1 hypothesis Hind definition of mde_
        <3>2 hypothesis H2 : ~ Val!( = )(w, h),
             prove ~ Val!( = )(v, w)
          <4>1 hypothesis H3 : Val!( = )(v, h),
               prove ~ Val!( = )(v, w)
               by hypothesis H2, H3 property Val!eq_symm, Val!eq_tran
          <4>2 hypothesis H3 : ~ Val!( = )(v, h),
               prove ~ Val!( = )(v, w)
           <5>1 prove ~ isnil(cons(h, t))
                by property isnil_cons
           <5>2 prove ~ Val!( = )(w, head(cons(h, t)))
                by property head_cons hypothesis H2
           <5>3 prove mem(v, cons(head(cons(h, t)),
                                  del(tail(cons(h, t)), w)))
                by hypothesis H1 definition of del step <5>1, <5>2
           <5>4 prove mem(v, cons(h, del(t, w)))
                by step <5>3 property head_cons, tail_cons
           <5>5 prove mem(v, del(t, w))
                by step <5>4 property mem_cons hypothesis H3
           <5>f qed by step <5>5 hypothesis Hind definition of mde_
          <4>f conclude
        <3>f conclude
       <2>f qed by definition of mde_ step <2>1
      <1>3 assume v w : Val,
           prove mde_(v, w, nil) ->
                 (all t : Self, mde_(v, w, t) ->
                  all h : Val, mde_(v, w, cons(h, t))) ->
                 all l : Self, mde_(v, w, l)
           assumed (* Standard induction principle *)
      <1>f conclude ;
 theorem mem_del_inv : all v w : Val, ~ Val!( = )(v, w) ->
                       all l : Self, mem(v, l) -> mem(v, del(l, w))
  proof = assumed (* TODO *) ;
 proof of mem_del = by property mem_del_mem, mem_del_eq, mem_del_inv
                       definition of mdm_, mde_ ;
end ;;

type mylist('a) = | FNil | FCons ('a, mylist('a)) ;;

species IndList(Val is SupersetWitness) =
 inherit FiniteList(Val) ;
 representation = mylist(Val) ;
 let nil = #FNil ;
 let cons(h : Val, t : Self) = #FCons(h, t) ;
 let isnil(l : Self) = match l with | #FNil -> true | _ -> false ;
 let head(l : Self) = match l with | #FCons(h, _) -> h | _ -> Val!witness ;
 let tail(l : Self) = match l with | #FCons(_, t) -> t | _ -> #FNil ;
 proof of finite =  coq proof definition of nil, cons
                    {* Proof.
                        intros.
                        unfold abst_cons, abst_nil, cons, nil in *.
                        induction l.
                         trivial.
                        apply H0; apply IHl.
                       Qed. *} ;
 proof of isnil_nil = assumed (* TODO *) ;
 proof of isnil_cons = assumed (* TODO *) ;
 proof of head_cons = assumed (* TODO *) ;
 proof of tail_cons = assumed (* TODO *) ;
 proof of list_dec = assumed (* TODO *) ;
 let mem(v : Val, l : Self) =
  let recstruct mem_(l : Self) =
   match l with
   | #FNil -> false
   | #FCons(h, t) -> if Val!( = )(v, h) then true else mem_(t)
  in mem_(l) ;
 proof of mem_nil = assumed (* TODO *) ;
 proof of mem_cons = assumed (* TODO *) ;
 let recstruct del(l : Self, v : Val) =
  match l with
  | #FNil -> #FNil
  | #FCons(h, t) -> if Val!( = )(v, h)
```

```
                    then del(t, v)
                    else #FCons(h, del(t, v)) ;
 proof of mem_del =  assumed (* TODO *) ;
end ;;
```

B.4 main.fcl

```
use "basics" ;;
open "superset" ;;
open "subset" ;;
open "mylist" ;;

species Int =
 inherit PrintParseSuperset ;
 representation = basics#int ;
 let ( = )(x : Self, y : Self) = basics#( = )(x, y) ;
 proof of eq_refl = by definition of ( = ) ;
 proof of eq_symm = by definition of ( = ) ;
 proof of eq_tran = by definition of ( = ) ;
 let witness = 0 ;
 let print(s : Self) = basics#print_int(s) ;
 let parse(s : basics#string) = basics#int_of_string(s) ;
end ;;

collection Int_Coll = implement Int; end ;;

collection IntList_Coll =  implement IndList(Int_Coll) ; end ;;

species ListSubset(Val is PrintParseSuperset, Sup is IndList(Val)) =
 inherit ExtSubset(Val), PrintParseSuperset ;
 representation = Sup ;
 let ( << )(v : Val, s : Self) = Sup!mem(v, s) ;
 let empty = Sup!nil ;
 proof of mem_empty = by definition of ( << ), empty property Sup!mem_nil ;
 let ( + )(s : Self, v : Val) = if v << s then s else Sup!cons(v, s) ;
 proof of mem_insert =
  <1>1 assume v1  v2 : Val, s : Self, hypothesis H1 : v1 << s + v2,
       prove Val!( = )(v1, v2) \/ v1 << s
       by definition of ( << ), ( + )
          property Sup!mem_cons hypothesis H1
  <1>2 assume v1 v2 : Val, s : Self, hypothesis H1 : Val!( = )(v1, v2),
       prove v1 << s + v2
       assumed (* TODO *)
  <1>3 assume v1 v2 : Val, s : Self, hypothesis H1 : v1 << s,
       prove v1 << s + v2
       assumed (* TODO *)
  <1>f conclude ;
 let ( - )(s : Self, v : Val) = Sup!del(s, v) ;
 proof of mem_remove = assumed (* TODO *) ;
 proof of mem_congr = assumed (* TODO *) ;
 let rec ( <: )(s1 : Self, s2 : Self) =
  if Sup!isnil(s1)
  then true
  else if Sup!head(s1) << s2
       then Sup!tail(s1) <: s2
       else false ;
 (* TODO : Termination Proof *)
 proof of mem_incl = assumed (* TODO *) ;
 let rec ( = )(s1 : Self, s2 : Self) =
  if Sup!isnil(s1)
  then Sup!isnil(s2)
  else let h1 = Sup!head(s1) and h2 = Sup!head(s2) in
       if basics#( @@ )(h1 << s2, h2 << s1)
       then ((s1 - h1) - h2) = ((s2 - h1) - h2)
       else false ;
 (* TODO : Termination Proof *)
 proof of eq_incl = assumed (* TODO *) ;
 let print(s : Self) =
  let x = basics#print_string("{") in
  let rec print_(s : Self)=
   if Sup!isnil(s)
```

```
   then basics#print_string("}")
   else let y = Val!print(Sup!head(s)) in print_(s - Sup!head(s))
  in print_(s) ;
 let parse(s : basics#string) = empty;
 let witness = empty ;
end ;;

collection IntSubset_Coll =
 implement ListSubset(Int_Coll, IntList_Coll) ;
end ;;

collection IntSubsetList_Coll = implement IndList(IntSubset_Coll) ; end ;;

collection IntSubset2_Coll =
 implement ListSubset(IntSubset_Coll, IntSubsetList_Coll) ;
end ;;

basics#print_string("\n") ;;
basics#print_string("Subsets of Integers :\n") ;;
basics#print_string("---------------------\n") ;;

let subset1 = IntSubset_Coll!empty ;;
basics#print_string("Creating empty set : ") ;;
IntSubset_Coll!print(subset1) ;;
basics#print_string("\n") ;;

let subset2 = IntSubset_Coll!( + )(subset1, Int_Coll!parse("1")) ;;
basics#print_string("Inserting 1 : ") ;;
IntSubset_Coll!print(subset2) ;;
basics#print_string("\n") ;;

let subset3 = IntSubset_Coll!( + )(subset2, Int_Coll!parse("2")) ;;
basics#print_string("Inserting 2 : ") ;;
IntSubset_Coll!print(subset3) ;;
basics#print_string("\n") ;;

let subset4 = IntSubset_Coll!( + )(subset3, Int_Coll!parse("3")) ;;
basics#print_string("Inserting 3 : ") ;;
IntSubset_Coll!print(subset4) ;;
basics#print_string("\n") ;;

let subset5 = IntSubset_Coll!( + )(subset4, Int_Coll!parse("2")) ;;
basics#print_string("Inserting 2 : ") ;;
IntSubset_Coll!print(subset5) ;;
basics#print_string("\n") ;;

let subset6 = IntSubset_Coll!( - )(subset5, Int_Coll!parse("2")) ;;
basics#print_string("Removing 2 : ") ;;
IntSubset_Coll!print(subset6) ;;
basics#print_string("\n") ;;

let subset7 = IntSubset_Coll!( - )(subset6, Int_Coll!parse("3")) ;;
basics#print_string("Removing 3 : ") ;;
IntSubset_Coll!print(subset7) ;;
basics#print_string("\n") ;;

basics#print_string("\n") ;;
basics#print_string("Subsets of Subsets of Integers :\n") ;;
basics#print_string("--------------------------------\n") ;;

let power1 = IntSubset2_Coll!empty ;;
basics#print_string("Creating empty set : ") ;;
IntSubset2_Coll!print(power1) ;;
basics#print_string("\n") ;;

let power2 = IntSubset2_Coll!( + )(power1, subset1) ;;
basics#print_string("Inserting {} : ") ;;
IntSubset2_Coll!print(power2) ;;
basics#print_string("\n") ;;

let power3 = IntSubset2_Coll!( + )(power2, subset2) ;;
basics#print_string("Inserting {1} : ") ;;
IntSubset2_Coll!print(power3) ;;
basics#print_string("\n") ;;
```

```
let power4 = IntSubset2_Coll!( + )(power3, subset3) ;;
basics#print_string("Inserting {21} : ") ;;
IntSubset2_Coll!print(power4) ;;
basics#print_string("\n") ;;

let power5 = IntSubset2_Coll!( - )(power4, subset2) ;;
basics#print_string("Removing {1} : ") ;;
IntSubset2_Coll!print(power5) ;;
basics#print_string("\n") ;;

let subset8 =
 IntSubset_Coll!( + )
  (IntSubset_Coll!( + )(IntSubset_Coll!empty, Int_Coll!parse("2")),
   Int_Coll!parse("1"));;
basics#print_string("Creating {12} : ") ;;
IntSubset_Coll!print(subset8) ;;
basics#print_string("\n") ;;

let power6 = IntSubset2_Coll!( + )(power5, subset8) ;;
basics#print_string("Inserting {12} : ") ;;
IntSubset2_Coll!print(power6) ;;
basics#print_string("\n") ;;

basics#print_string("\n") ;;
```