FoCaLiZe

Reference Manual

0.9.2

June 2018

Authors

Thérèse Hardin, François Pessaux, Pierre Weis, Damien Doligez

About FoCaLiZe

FoCaLiZe *is the result of a collective work of several researchers, listed in the following, who designed, defined, compiled, studied, extended, used and debugged the preceding versions. They were helped by many students who had a summer internship under their supervision. They would like to thank all these students and more generally all the persons who brought some contribution to* FoCaLiZe.

FoCaLiZe **contributors**

Philippe Ayrault (SPI-LIP6), William Bartlett (CPR-CEDRIC), Julien Blond (SPI-LIP6), Sylvain Boulmé (SPI-LIP6), Matthieu Carlier (CPR-CEDRIC), David Delahaye (CPR-CEDRIC), Damien Doligez (GALLIUM-INRIA), Catherine Dubois (CPR-CEDRIC), Jean-Frédéric Etienne (CPR-CEDRIC), Stéphane Fechter (SPI-LIP6), Lionel Habib (SPI-LIP6), Thérèse Hardin (SPI-LIP6), Eric Jaeger (SPI-LIP6), Mathieu Jaume (SPI-LIP6), Charles Morisset (SPI-LIP6), Ivan Noyer (SPI-LIP6), François Pessaux (SPI-LIP6)(UIIS-ENSTA ParisTech), Virgile Prevosto (SPI-LIP6), Renaud Rioboo (CPR-CEDRC), Lien Tran (SPI-LIP6), Véronique Viguié Donzeau-Gouge (CPR-CNAM), Pierre Weis (ESTIME-INRIA)

and their institutions

SPI (Semantics, Proofs and Implementations) is a team of LIP6, (Laboratoire d'Informatique de Paris 6) of UPMC (Pierre and Marie Curie University)[1].

CPR (Conception et Programmation Raisonnées) is a team of CEDRIC (Centre d'Etudes et de Recherches du CNAM) of CNAM (Conservatoire National des Arts et Métiers)[2] *and ENSIIE (Ecole Nationale d'Informatique pour l'Industrie et l'Entreprise)*[3].

[1] UPMC-LIP6, 104 avenue du Président Kennedy, Paris 75016, France, `Firstname.Lastname@lip6.fr`
[2] CNAM-CEDRIC, 292 rue Saint Martin, 75003, Paris, France, `Firstname.Lastname@cnam.fr`
[3] ENSIIE-CEDRIC, 1 Square de la Résistance, 91025 Evry Cedex, France, `Lastname@ensiie.fr`

ESTIME and GALLIUM are teams of INRIA Rocquencourt[4].

U2S (Unité d'Informatique et d'Ingénierie des Systèmes) is a team of ENSTA ParisTech[5].

Thanks

The Foc project was first partially supported by LIP6 (Projet Foc, LIP6 1997) then by the Ministry of Research (Action Modulogic). The Focal research team was then partially supported by the French SSURF ANR project ANR-06-SETI-016 (Safety and Security UndeR Focal). The project also benefited of strong collaborations with the EDEMOI ANR project and with the BERTIN and SAFERIVER companies.

The FoCaLiZe language and compiler development effort started around 2005. The architecture conception and code rewritting started from scratch in 2006 to finally make the first focalizec compiler and FoCaLiZe system distribution in 2009, January.

This manual documents the completely revised system with the new syntax and its semantics extensions.

[4]INRIA, Bat 8. Domaine de Voluceau, Rocquencourt, BP 105, F-78153 Le Chesnay, France, `Firstname.Lastname@inria.fr`

[5]ENSTA, 828 boulevard des Maréchaux, F-91120 Palaiseau, France, `Firstname.Lastname@ensta-paristech.fr`

Contents

Introduction

Motivations

The Foc project was launched in 1998 by T. Hardin and R. Rioboo [16] [6] with the objective of helping all stages of development of critical software within safety and security domains. The methods used in these domains are evolving, ad-hoc and empirical approaches being replaced by more formal methods. For example, for high levels of safety, formal models of the requirement/specification phase are more and more considered as they allow mechanized proofs, test or static analysis of the required properties. In the same way, high level assurance in system security asks for the use of true formal methods along the process of software development and is often required for the specification level. Thus the project was to elaborate an Integrated Development Environment (IDE) able to provide high-level and justified confidence to users, but remaining easy to use by well-trained engineers.

To ease developing high integrity systems with numerous software components, an IDE should provide tools to formally express specifications, to describe design and coding and to ensure that specification requirements are met by the corresponding code. But this is not enough. First, standards of critical systems ask for pertinent documentation which has to be maintained along all the revisions during the system life cycle. Second, the evaluation conformance process of software is by nature a skeptical analysis. Thus, any proof of code correctness must be easily redone at request and traceability must be eased. Third, design and coding are difficult tasks. Research in software engineering has demonstrated the help provided by some object-oriented features as inheritance, late binding and early research works on programming languages have pointed out the importance of abstraction mechanisms such as modularity to help invariant preservation. There are a lot of other points which should also be considered when designing an IDE for safe and/or secure systems to ensure conformance with high Evaluation Assurance or Safety Integrity Levels (EAL-5 to 7 or SIL 3 and 4) and to ease the evaluation process according to various standards (e.g. IEC61508, CC, ...): handling of non-functional contents of specification, handling of dysfunctional behaviors and vulnerabilities from the true beginning of development as well as fault avoidance and fault detection by validation testing, vulnerability and safety analysis.

Initial application testbed

When the Foc project was launched by T. Hardin and R. Rioboo, only the specific domain of Computer Algebra was initially considered. Algorithms used in this domain can be rather intricated and difficult to test and this is not rare that computer algebra systems issue a bad result, due to semantical flaws, compiler anomalies, etc. Thus the idea was to design a language allowing to specify the mathematics underlying these algorithms and to go step by step to different kinds of implementations according to the specifities of the problem under consideration[7]. The first step was to design the semantics of such a language, trying to fit to several requirements: easing the expression of mathematical statements, clear distinction between the mathematical structure (semi-ring, polynomial, ..) and its different implementations, easing the development (modularity, inheritance, parametrisation, abstraction, ..), runtime efficiency and confidence in the whole development (mechanised proofs, ..). After an initial phase of conceptual design, the Foc semantics was submitted to a double test. On one hand, this semantics was specified in Coq and in a categorical model of type theories by S. Boulmé (see his thesis [3]), a point which enlightened the borders of this approach,

[6] They were members of the SPI (Semantics, Proofs, Implementations) team of the LIP6 (Lab. Informatique de Paris 6) at Université Pierre et Marie Curie (UMPC), Paris

[7] For example Computer Algebra Libraries use different representations of polynomials according to the treatment to be done

regarding the logical background. On the other hand, as a preliminary step before designing the syntax, a strudy of the typical development style was conducted. R. Rioboo [5, 16] used the OCaml language to try different solutions, recorded in [16].

Initial Focal design

Then the time came to design the syntax of the language and the compiler. To overcome inconsistencies risks, an original dependency analysis was incorporated into the compiler (V. Prevosto thesis [23, 26, 25]) and the correction of the compiler (mostly written by V. Prevosto) against Focal's semantics is proved (by hand) [24], a point which brings a satisfactory confidence in the language's correctness. Then R. Rioboo[4] began the development of a huge Computer Algebra library, offering full specification and implementation of usual algebraic structures up to multivariate polynomial rings with complex algorithms, to extensively test the language and the efficiency of the produced code, aswell as to provide a standard library of mathematical backgrounds. D. Doligez [2] started the development of Zenon, an automatic prover based on tableaux method, which takes a Focal statement and tries to build a proof of it and, when succeeds, issues a Coq term. More recently, M. Carlier and C. Dubois[21] began the development of a test tool for Focal.

Focal has already been used to develop huge examples such as the standard library and the computer algebra library. The library dedicated to the algebra of access control models, developed by M. Jaume and C. Morisset [18, 19, 22], is another huge example, which borrows implementations of orderings, lattices and boolean algebras from the computer algebra library. Focal was also very successfully used to formalize airport security regulations, a work by D. Delahaye, J.-F. Etienne, C. Dubois, V. Donzeau-Gouge [11, 12, 13]. This last work led to the development of a translator [10] from Focal to UML for documentation purposes.

The FoCaLiZe system

The FoCaLiZe development started in 2006, as a continuation of the Foc and Focal efforts. The new system was rewritten from scratch. A new language and syntax was designed and carefully implemented, with in mind ease of use, expressivity, and programmer friendlyness. The addition of powerful data structure definitions – together with the corresponding pattern matching facilities – leads to new expressive power.

The Zenon automatic theorem prover was also integrated in the compiler and natively interfaced within the FoCaLiZe language. New developments for a better support of recursive functions is on the way (in particular for termination proofs).

The FoCaLiZe system in short

The FoCaLiZe system provides means for the developers to formally express their specifications and to go step by step (in an incremental approach) to design and implementation while proving that such an implementation meets its specification or design requirements. The FoCaLiZe language offers high level mechanisms such as multiple inheritance, late binding, redefinition, parametrization, etc. Confidence in proofs submitted by developers or automatically done relies on formal proof verification. FoCaLiZe also provides some automation of documentation production and management.

A formal specification can be built by declaring names of functions and values and introducing properties. Then, design and implementation can incrementally be done by adding definitions of functions and proving that the implementation meets the specification or design requirements. Thus, developing in FoCaLiZe is a kind of refinement process from formal model to design and code, completely done within

FoCaLiZe. Taking the global development in consideration within the same environment brings some conciseness, helps documentation and reviewing.

A FoCaLiZe development is organised as a hierarchy that may have several roots. The upper levels of the hierarchy are built along the specification stage while the lower ones correspond to implementation and each node of the hierarchy corresponds to a progress toward a complete implementation.

We would like to mention several works about safety and/or security concerns within FoCaLiZe and specially the definition of a safety life cycle by P. Ayrault, T. Hardin and F. Pessaux [1] and the study of some traps within formal methods by E. Jaeger and T. Hardin[17].

FoCaLiZe can be seen as an IDE still in development, which gives a positive solution to the three requirements identified above:

1. pertinent documentation is maintained within the system being written, and its extraction is an automatic part of the compilation process,

2. proofs are produced using an automated proved which can be guided using a high level proof language, so that proofs are easier to write and their verification is automatic and reliable,

3. the framework provides powerful abstraction mechanisms to facilitate design and development; however, these mechanisms are carefully ruled: the compiler performs numerous validity checks to ensure that no further development can inadvertantly break the invariants or invalidate the proofs; indeed, the compiler ensures that if a theorem was based on assumptions that are now violated by the new development, then the theorem is out of reach of the programmer and the properties have to be proven again.

Chapter 1

Overview

Before entering the precise description of FoCaLiZe, we give an informal presentation of its main features, to help further reading of the reference manual. Every construction or feature that we sketch here is entirely and precisely described in the following chapters.

1.1 The Basic Brick

The primitive entity of a FoCaLiZe development is the *species*. It can be viewed as a record grouping "things" related to the same concept. Like in most modular design systems (i.e. objected oriented, algebraic abstract types), the idea is to group a data structure with the operations that operate on it. Since in FoCaLiZe we don't only address data type and operations, these "things" also comprise the declaration (or specification) of the operations, their stated properties (which represent the requirements for the operations), and the proofs of these properties.

We now informely describe each of these "things", called the *methods* of the species.

- The `representation` gives the data representation of the entities manipulated by the *species*. It is a type defined by a type expression. The *representation* definition may be deferred, which means that the structure of the embedded data-type does not need to be known at this point. In this case, it is simply a type variable. However, to finally obtain an implementation, the *representation* has to be defined at some point, either by setting `representation = type_exp` where `type_exp` is a type expression or by inheritance (see below). Type expressions in FoCaLiZe are roughly speaking the ML type expressions (variables, basic types, inductive types, record types).
  Each *species* has a unique *representation*. This is not a restriction compared to other languages where programs/objects/modules can own several private variables representing the internal state, since the variables define some part of the data structure of the entities manipulated by the program/object/-module. The equivalent FoCaLiZe *representation* is simply a tuple grouping in one place all these variables that were disseminated in the entire program/object/module.
- Declarations are composed of the keyword `signature` followed by a name and a type. They announce a *method* to be defined later (the type of the method is given but the implementation is still omitted). Once a method is declared, this method can be used in the text following the declaration, in particular in the definition of other methods: indeed, the type provided by the *signature* allows the FoCaLiZe compiler to check that the method is consistently used in all contexts with a type compat-

ible with the declared type. Furthermore, the late-binding and the collection mechanisms introduced below, ensure that the definition of the method is known when the method is effectively invoked.

- Definitions are composed of the keyword `let`, followed by a name, an optional type, and an expression. They serve to introduce constants or functions, i.e. computational operations. The core language used to implement them is roughly ML-like expressions (let-binding, pattern matching, conditional, higher order functions, ...) with the addition of a construction to call a *method* from a given *species*. Mutually recursive definitions are introduced by `let rec`.
- Property statements are composed of the keyword `property` followed by the name of the property and its definition, a first-order formula. A *property* may serve to express requirements (i.e. a fact that the system must hold to conform to the Statement of Work) and in this case we can view the property as a specification purpose *method*, like a *signature* was for `let`-*method*s. A property induces a *proof obligation* to be discharged at some point in the development. A *property* may also be used to express some "quality" information of the system (soundness, correctness, ...) also submitted to a proof obligation. The formulae are written with the usual logical connectors, universal and existential quantifications over a FoCaLiZe type, and cite the name of any *method* known within the *species*' context. For instance, a *property* telling that for any vehicle, if the speed is non-null, then the doors cannot be opened could look like:

  ```
  all m in Self, !speed(m) <> Speed!zero -> ~doors_open(m)
  ```

  In the same way as *signatures*, a yet to be proved *property* can be used as an hypothesis in the proof of other properties or theorems. Once more, the FoCaLiZe late binding and collection mechanisms ensure that the proof of a *property* will be ultimately done.
- Theorems (`theorem`) made of a name, a statement and a proof are *properties* packed with the formal proof that their statement holds in the context of the *species*. The proof accompanying the statement will be processed by FoCaLiZe, a first-order theorem prover (either Zenon or Zenon Modulo) and ultimately checked with a proof checker (either the Coq proof assistant or the Dedukti logical framework).

Regarding properties and theorems, note that like in any formal development, the difficulty may be more to express a true, interesting and meaningful statement, than to prove it. For instance, claiming that a piece of software is "formally proved" because it respects a safety requirement is meaningless if the statement of this requirement is trivially true (see [17] for examples).

Let's illustrate these notions on an example that we incrementally extend. We want to model some simple algebraic structures. Let's start with the description of a "setoid" representing the data structure of "things" belonging to a set, which can be submitted to an equality test and exhibited (i.e. one can get a witness of existence of one of these "things").

```
species Setoid =
  signature ( = ) : Self -> Self -> bool ;
  signature element : Self ;

  property refl : all x in Self, x = x ;
  property symm : all x y in Self, x = y -> y = x ;
  property trans: all x y z in Self, x=y and y=z -> x=z ;
  let different (x, y) = basics#not_b (x = y) ;
  theorem different_irrefl : all x in Self, ~different(x, x)
   proof = by definition of different
              property refl ;
end ;;
```

In this *species*, the *representation* is not explicitly given (the keyword `representation` is not used), since we don't need to set it to the express functions and properties that a "setoid" requires. However, we can refer to the representation via `Self` (in this case a type variable). In the same way, we just specify a *signature* for the equality (operator =). We introduce the three properties that the equality must have (exactly the properties of an equivalence relation).

We complete the example by the definition of the function `different` which uses the (not yet defined) = method, and the predefined boolean negation `basics#not_b`.

Not only we can define `different` out of a not yet defined method =, we can also prove a property of `different` based on the not yet proved properties of =! Indeed we prove that `different` is irreflexive, under the hypothesis that = is an equivalence relation (i.e. that any implementation of = used by `different` will satisfy these properties).

Note: `basics#not_b` stands for the function `not_b` defined in the FoCaLiZe `basics` development (which is in the source file `basics.fcl` of the standard library).

In FoCaLiZe, the *late-binding* feature makes it possible to use *methods* as soon as they have been declared and way before they get a real *definition*. Similarly, FoCaLiZe allows arbitrary *method* redefinition: the effective *definition* of the method inside a species is guarrantied to be the last version of the successive definitions of the method.

1.2 Type of Species, Interfaces and Collections

The *type* of a *species* is obtained by removing all the definitions and proofs. Thus, it is somoe kind of record type, made of all the method types of the species. If the `representation` is still a type variable say α, then the *species* type is prefixed with an existential binder $\exists\alpha$. This binder is eliminated as soon as the `representation` is known. Technically, the existancial type variable is instantiated when the representation type is defined; furthermore, the compiler checks that all existancial type variables have been eliminated before the generation of runnable code.

The *interface* of a species is obtained by abstracting the *representation* type in the *species type*; this abstraction is permanent.

Warning *No special construction is given to denote the interface of a species in the concrete syntax, it is simply denoted by the name of the species.* Do not confuse a species and its interface.

The *species type* remain totally implicit in the concrete syntax, being just used as a step to build the *species interface*. It is used during inheritance resolution.

Interfaces can be ordered by inclusion, a point providing a very simple notion of subtyping. This point will be further commented.

A species is said to be *complete* when the representation and all the declarations have received a definition and all the properties have received a proof.

When *complete*, a species can be submitted to an abstraction process of its representation to create a *collection*. Put it the other way round: a collection abstracts a complete species. Thus, the *interface* of a collection is the *interface* of the abstracted complete species. Thus, a collection is a kind of abstract data type, only usable through the methods of its interface, with the addional guarantee that all the declarations have been defined and all the statements have been proved.

1.3 Combining Bricks by Inheritance

A FoCaLiZe development is organised as a hierarchy which may have several roots. Usually the upper levels of the hierarchy are built during the specification stage while the lower ones correspond to implementations. Each node of the hierarchy, i.e. each *species*, is a progress towards a complete implementation. On the previous example, putting aside `different`, we typically presented a kind of *species* for "specification" since it expresses only *signatures* of functions to be later implemented and properties to be later proved.

We can now create a new *species* by **inheritance** of an already defined one. We can make this new species more "complex" by adding new operations and properties, or we can make it more concrete by providing definitions to *signatures* and proofs to *properties*, without adding new features.

Hence, the FoCaLiZe inheritance notion serves two kinds of evolutions in the development process. The first kind of evolution is *additional complexity*: the inheritance makes more complex *species* out of simpler ones; the new species gets more operations than its parents (keeping the ancestors operations or possibly redefining some of them, if required). The second kind of inheritance is *refinement*: the new species has less and less still unknown parts; it tends to the status of a "runnable" implementation, providing explicit definitions to the *methods* that were previously only declared.

Continuing our example, we want to extend our model to represent "things" with a multiplication and a neutral element for this operation.

```
species Monoid =
  inherit Setoid;
  signature ( * ) : Self -> Self -> Self ;
  signature one : Self ;
  let element = one * one ;
end ;;
```

Monoid are "things" that are *Setoids* but also have an operation $\star$ and a specific value called *one*; besides the new *methods* we also gave a definition to `element`, saying it is the application of the method $\star$ to `one` twice, both of them being only *declared*. Here, we used the inheritance in both the presented ways: making a more complex entity by adding *methods* and getting closer to the implementation by explicitly defining `element`.

FoCaLiZe provides multiple inheritance. For sake of simplicity, the above example uses simple inheritance. When inheriting the same *method* from more than one parent, the order of parents apparition in the `inherit` clause serves to determine the chosen *method* (only the latest definition of any method appearing several times in the list of inherited species is retained).

The *type* of a *species* built using inheritance is defined like for other *species*, the *method* types being those of the *methods* appearing in the *species* after inheritance resolution.

A strong constraint in inheritance is that the type of inherited, and/or redefined *methods* cannot change. This is required to ensure consistency of the FoCaLiZe model, hence of the developed software. More precisely, if the representation is given by a type expression containing some type variables, then it can be more defined by instanciation of these variables. In the same way, two signatures have compatible types if they have a common unifier; thus, roughly speaking, if they are compatible as ML-like types. For example, if the representation was not yet defined (thus being still a type variable), it can be defined by `int`. And if a species S inherits from S_1 and $S2$ a method called m, there is no type clash if $S_1!m$ and $S_2!m$ can be unified, then the method $S!m$ is addigned the most general unifier of these two types.

In a nutshell, if a species *B* inherits from a species *A*, the intuition is that any instance of *B* is also an instance of *A*.

1.4 Combining Bricks by Parameterisation

As indicated, inheritance is used to enrich or to implement *species*. However, we sometimes need to use a *species*, not to take over its *methods*, but rather to use it as an "ingredient" to build a new structure. For instance, a product of setoids is a new structure, using the previous *species* as the "ingredient". Indeed, the structure of a product is not similar to any of its component, but is build using the structures of its components. A product can be seen as *parameterised* by its two components. Following this idea, FoCaLiZe allows two flavors of parameterisation.

1.4.1 Parameterisation by Collection

We first introduce the *collection parameters*. They are *collections* that the hosting species may use through their *methods* to define its own ones.

A *collection parameter* is given a name C and a (species) interface I. The name C serves to call the *methods* declared in I. Intuitively, C will at some stage be implemented by a collection CE whose interface contains the methods of the interface I. Moreover, the collection and late-binding mechanisms ensure that all methods appearing in I are indeed implemented (defined for functions, proved for properties) in CE. Thus, no runtime error, due to linkage of libraries, can occur and any *property* or *theorem* stated in I can be safely used as an hypothesis.

Calling a *species*'s *method* is done via the "bang" notation: `!meth` or `Self!meth` for a *method* of the current *species* (and in this case, even simpler: `meth`, since the FoCaLiZe compiler will resolve scoping issues). To call *collection parameters*'s *method*, the same notation is used: `A!element` stands for the *method* `element` of the *collection parameter* `A`.

To go on with our example, a product of setoids has two components, hence a *species* for products of setoids has two *collection parameters*. It is itself a setoid (that is, a "thing" with an equality), a fact which is simply recorded via the inheritance mechanism: `inherit Setoid` gives to `Setoid_product` all the methods of `Setoid`.

```
species Setoid_product (A is Setoid, B is Setoid) =
  inherit Setoid;

  signature fst : Self -> A ;
  signature snd : Self -> B ;
  signature pair : A -> B -> Self ;

  let element = Self!pair(A!element, B!element) ;

  let ( = ) (x, y) = basics#and_b (A!( = )(fst(x), fst(y)),
                                   B!( = )(snd(x), snd(y))) ;
  proof of refl = by definition of ( = )
                   property A!refl, B!refl ;

end ;;
```

We first declare methods `fst`, `snd` and `pair` to represent the two projections and the construction of pairs. Next, we introduce a definition for `element` by building a pair, using the function `pair` applied to the method `element` of respectively `A` and `B`. We also add a definition for = of `Setoid_product`, relying on the methods = of `A` and `B` (which are not yet defined), and we prove that = of `Setoid_product` is indeed reflexive, upon the hypothesis made on `A!( = )` and `B!( = )`. The part of FoCaLiZe used to write proofs will be shortly presented later, in section 1.6.

Such a species can be refined with `representation = A * B`, indicating that the representation of the product is the Cartesian Product of the representation of the two parameters. In `A * B`, `*` is the FoCaLiZe type constructor of pairs, `A` denotes indeed the representation of the first *collection parameter*, and `B` the one of of the second *collection parameter*.

This way, the *species* `Setoid_product` builds its *methods* relying on those of its *collection parameters*. Note the two different uses of `Setoid` in our *species* `Setoid_product`, which both inherits of and is parameterised by it.

Why *collection parameters* and not simply *species parameters*? There are two reasons. First, effective parameters must provide definitions/proofs for all the methods of the required interface: this is the contract. Thus, effective parameters must be *complete* species. Then, we do not want the parameterisation to introduce dependencies on the parameters' *representation* definitions. For example, it is impossible to express " if `A!representation` is `int` and `B!representation` is `bool` then `A * B` is a list of boolean values". This would dramatically restrict the possibilities to instantiate parameters since assumptions on the *representation*, possibly used in the parameterised *species* to write its own *methods*, could prevent *collections* having the right set of *methods* but a different representation to be used as effective parameters. Such a behaviour would make parameterisation too weak to be usable. We choose to always hide the *representation* of a *collection parameter* to the parameterised hosting *species*. Hence the introduction of the notion of *collection*, obtained by abstracting the representation from a complete species.

1.4.2 Parameterisation by Entity (Value)

Let us imagine we want to make a *species* to implement arithmetic on natural numbers modulo a certain value. In the expression $5 \; \texttt{modulo} \; 2 \; is \; 1$, both 5 and 2 are natural numbers. To be sure that the *species* will consistently work with the same modulo, this last one must be embedded in the *species*. However, the *species* itself doesn't rely on a particular value of the modulo. Hence this value is clearly a **parameter** of the species, but a parameter for which need its **value**, not only by its *representation* and the methods acting on it. Such a parameter is named an *entity parameter*. Being a value, an entity parameter belongs to some collection, and this collection must also be declared as a *collection parameter* of the species. An entity parameter denotes a *value* having the type of the *representation* of its associated *collection parameter*.

As an exemple, let us define a collection `Modulo_n`. We first define a *species* to represent natural numbers:

```
species NatModel =
  signature one : Self ;
  signature inc : Self -> Self ;
  signature modulo : Self -> Self -> Self ;
end ;;
```

Note that `NatModel` can be later implemented in various ways, using Peano's integers, machine integers, arbitrary-precision arithmetic (as well as things that are not really integers, our specification being too simplistic)...

The *species* "working modulo n..." now embeds the value of `n` as an element of a collection for `NatModel`:

```
species Modulo_n (Naturals is NatModel, n in Naturals) =
  let job1 (x : Naturals) =
    ... Naturals!modulo (x, n) ... ;
  let job2 (x : Naturals, ...) =
    ... ... Naturals!modulo (x, n) ... ... ;
```

```
end ;;
```

Using the *entity parameter* `n`, we ensure that the *species* `Modulo_n` works for *any* value of the modulo, but will always use the *same* value `n` of the modulo everywhere inside the *species*.

1.5 The Final Brick

As briefly introduced in 1.2, a *species* needs to be complete to lead to executable code for its functions and checkable proofs for its theorems. When a *species* is complete, it can be turned into a *collection*. Hence, a *collection* represents the final stage of the inheritance tree of a *species* and leads to an effective data representation with executable functions processing it.

For instance, providing that the previous *species* `NatModel` has been refined into a fully-defined species `MachineNativeInt` through inheritances steps, with a *method* `from_string` allowing to create the natural representation of a string, we could get a related collection by:

```
collection MachineNativeIntegers =
  implement MachineNativeInt;
end ;;
```

Next, to get a *collection* implementing arithmetic modulo 8, we can define a *collection* for the *species* `Modulo_n`:

```
collection Modulo_8 =
  implement Modulo_n
    (MachineNativeIntegers, MachineNativeIntegers!from_string ("8");
end;;
```

As exemplify here, a species is applied to effective parameters by giving their values with the same syntax as for parameter passing.

As said before, to ensure modularity and abstraction, the *representation* of a *collection* is hidden (as well as its definitions). It means that any software component using a *collection* will only be able to manipulate its values through the operations (*methods*) that the collection provides via its interface. As a corollary, no other software component can possibly break the invariants required by the internals of a *collection*.

1.6 Properties, Theorems and Proofs

FoCaLiZe not only provides a way to write programs, it also intends to encompass both the executable model (i.e. program) and the properties that this model must satisfy. For this reason, some "special" fields of the species only deal with logic instead of specifying purely behavioural aspects of the program: those logical aspects are *theorems*, *properties* and *proofs*.

Stating a *property* declares that a *proof* that the property **holds** will be given at some stage of the development. The *theorems* are properties for which the *proof* is given with the statement. All the proofs must be done by the developer; the compiler ultimately send them to the proof checker for verification: all the demonstrations made in FoCaLiZe are automatically machine checked for consistency (by Coq or Dedukti).

FoCaLiZe provides several ways to write proofs. The normal and encouraged way is to use the FoCaLiZe's proof language to write the proofs. The FoCaLiZe's proof language (or *FPL* for short), is a hierarchical proof language especially designed to give an easy way to vary the grain of the proof, from rough sketch to fully detailed proof. As in the usual mathematical activity, the idea is to provide hints and direction for a proof and let the reader complete the details. Well, don't panic: you will not have to complete the proofs of all the FoCaLiZe development you may ever read or write! The proofs in FoCaLiZe do not require a human being to read them and complete their numerous omitted details! The FoCaLiZe system delegates this burden to a companion program: the Zenon proof finder.

From the hints given in the FoCaLiZe development, Zenon attempts to generate a complete proof and exhibit a Coq or Dedukti proof term suitable for verification. So far so good! But what happens if Zenon fails to find the (complete) proof ? Well, you can consider this failure as a hint that the given proof was too sketchy: you have to develop it a bit, for instance by stating and proving some intermediate lemmas or by detailing the proof path! Once again, this is rather natural and not so far from normal mathematical activity.

Zenon [2, 14] is developed by D. Doligez. It is a first order theorem prover based on the tableau method incorporating implementation novelties such as sharing.

The FoCaLiZe programmer gives basic hints to Zenon such as: "prove by definition of this *method*" (i.e. look inside the body of the method) or "prove by this *property*" (i.e. use the logical statement of a *theorem* or *property* already proven). This hint mechanism is embedded into the entire FPL description of the proof into steps stating assumptions (that must obviously be demonstrated afterwards) in order to prove some lemmas or parts of the property at hand.

We now give such a demonstration.

```
theorem order_inf_is_infimum : all x y i : Self,
  !order_inf(i, x) -> !order_inf(i, y) ->
    !order_inf(i, !inf(x, y))
  proof =
    <1>1 assume
           x : Self, y : Self, i : Self,
           hypothesis H1 : !order_inf (i, x),
           hypothesis H2 : !order_inf (i, y),
         prove !order_inf (i, !inf (x, y))
      <2>1 prove !equal (i, !inf (!inf (i, x), y))
        by hypothesis H1, H2
           property inf_left_substitution_rule,
             equal_symmetric, equal_transitive
           definition of order_inf
      <2>f qed
        by step <2>1
           property inf_is_associative, equal_transitive
           definition of order_inf
    <1>f conclude
  ;
```

The important point is that Zenon works for the developer: **it searches and completes the proof** for the developer that no more have to elaborate the proof completely formally "from scratch".

Like all other automatic theorem prover, Zenon may fail to find a demonstration. In this case, FoCaLiZe allows the developer to write verbatim Coq or Dedukti proofs. Comfort of automation is lost in favor of an increase in expressive power: the entire Dedukti and Coq vernacular languages are now available to the developer to write the proof.

Finally, the `assumed` keyword is the ultimate proof backdoor: the proof is not given and the property is admitted. Obviously, a safe development should not make a liberal usage of this feature, since `assumed` bypasses the formal verification of the software's model. However, such a functionality remains needed for various reasons. For example, a development may be linked with external code; the properties of the FoCaLiZe code now depends on properties of the external code; to continue the development as safely as possible, it is necessary to carefully state the properties that are assumed for the external code and go on providing properties and proofs about the program: those proofs will give valuable confidence, even if they only hold if the set of assumptions for the external code is valid.

1.7 Around the Language

In the previous sections, we presented FoCaLiZe through its programming model and shortly its syntax. We especially investigated the various entities making a FoCaLiZe program. We now address what becomes a FoCaLiZe program once compiled. We recall that FoCaLiZe supports the redefinition of functions, which permits for example to specialise the code to a specific representation of data (for example, there exists a generic implementation of integer addition modulo `n` but this implementation can be redefined in arithmetics modulo `2`, if boolean values are used to represent the two values). In summary, FoCaLiZe is also a very convenient tool to maintain software.

1.7.1 Consistency of the Software

All along the FoCaLiZe development cycle, the compiler keeps trace of dependencies between *species*, *methods*, and *proofs* ... to ensure that all the modifications are consistently propagated to any place that needs to be changed.

FoCaLiZe deals with two types of dependencies:

- The **decl**-dependency: a *method* A decl-depends on a *method* B, if the **declaration** of B is required to state A.
- The **def**-dependency: a *method* (and more especially, a *theorem*) A def-depends on a *method* B, if the **definition** of B is required to state A (and more especially, to prove the property stated by the *theorem* A).

The redefinition of a function may invalidate the proofs that use the properties of the body of the function. All the proofs which truly depend of the definition are then invalidated by the compiler and must be done again in the updated context where the function gets the new definition. Thus, choose the proper level in the hierarchy to do a proff is a major practical difficulty. In [27], Prevosto and Jaume propose a *coding style* to minimise the number of proofs to be redone in case of redefinition, by using a certain kind of modularisation for the proofs.

1.7.2 Code Generation

FoCaLiZe currently compiles programs toward three languages, OCaml to get an executable standalone, and Coq and Dedukti [7, **?**] to have a formal model of the program, with theorems and proofs entirely machine checked.

In the OCaml code generation, all the logical aspects are discarded since they do not lead to executable code.

In contrast, in Coq and Dedukti, all the *methods* are compiled, i.e. "computational" *methods* and logical *methods* with their proofs. This allows the proof checker to check the entire consistence of the system developed in FoCaLiZe.

1.7.3 Tests

FoCaLiZe incorporates a tool named *FocalTest* [21] for Integration/Validation testing. It allows to confront automatically a property of the specification with an implementation. It automatically generates test cases, executes them and produces a test report as an XML document. The property under test is used to generate the test case but also as an oracle. When a test case fails, it means that a counterexample of the property has been found: the implementation does not match the property. This surely points out an inconsistency, due to a problem in the code or in the specification.

The tool *FocalTest* automatically produces the test environment and the drivers to conduct the tests. It benefits from the inheritance mechanism to isolate the testing harness from the components written by the programmer.

The testable properties are required to be broken down into a precondition and a conclusion, both executable. The current version of *FocalTest* proposes a pure random test cases generation: a test case is generated, if it satisfies the pre-condition then the verdict of the test case is obtained by executing the post-condition. For some form of preconditions the test generation process can be computationally challenging. To overcome this drawback, a constraint based generation is under development: it would allow to directly produce test cases that satisfy the precondition.

1.7.4 Documentation

The FoCaLiZeDoc tool [20] is a documentation generator. The documentation is automatically extracted from the FoCaLiZe source, hence the documentation of a component is always in par with its implementation.

FoCaLiZeDoc uses its own XML format that contains information coming not only from programmer added structured comments (that are parsed and kept in the program's abstract syntax tree) and from the FoCaLiZe concrete syntax but also from the results of the type inference and dependency analysis. From this XML representation and thanks to some XSLT stylesheets, FoCaLiZeDoc generated a bunch of HTML or LaTeX files. The generated documentation cannot be considered as a complete safety case; however, it can helpfully contribute to the elaboration of the safety case. In the same way, it is possible to produce UML models [10] as means to provide a graphical documentation for the FoCaLiZe specifications. The use of graphical notations appears quite useful when interacting with end-users, as these tend to be more intuitive and are easier to grasp than their formal (or textual) counterparts. This transformation is based on a formal schema and captures every aspect of the FoCaLiZe language, so that it has been possible to prove the soundness of this transformation (semantic preservation).

The FoCaLiZe compiler's architecture is designed to easily plug third-parties analyses that can benefit from the internal structures elaborated by the compiler from the source code. This allows, for instance, to make dedicated documentation tools for custom purposes, just exploiting the information stored in the FoCaLiZe program's abstract syntax tree, or the extra information added by some external processes or specialized analyses.

Chapter 2

Installing and Compiling

2.1 Required software

To be able to develop with the FoCaLiZe environment, a few third party tools are required. All of them can be freely downloaded from their related website.

- The Objective Caml compiler (version $\geq$ 3.10.2).
  Available at `http://caml.inria.fr`. This will be used to compile both the FoCaLiZe system at installation stage from the tarball and the FoCaLiZe compiler's output generated by the compilation of your FoCaLiZe programs.
- The Coq Proof Assistant (version $\geq$ 8.1pl4).
  Available at `http://coq.inria.fr`. This will be used to compile both the FoCaLiZe libraries at installation stage from the tarball and the FoCaLiZe compiler's output generated by the compilation of your FoCaLiZe programs.
- The Dedukti Logical Framework (version $\geq$ 2.5).
  Available at `https://github.com/Deducteam/Dedukti`. This will be used to compile both the FoCaLiZe libraries at installation stage from the tarball and the FoCaLiZe compiler's output generated by the compilation of your FoCaLiZe programs. The Sukerujo parser is used, it is available on the `sukerujo` branch of Dedukti git repository.

Note that some distributions of FoCaLiZe includes these tools and are automatically installed during the FoCaLiZe installation process.

2.2 Optional software

The FoCaLiZe compiler can generate dependencies graphs from compiled source code. It generates them in the format suitable to be processed and displayed by the dotty tools suite of the "Graphwiz" package. If you plan to examine these graphs, you also need to install this software from `http://www.graphviz.org/`.

2.3 Operating systems

FoCaLiZe was fully developed under Linux using free software. Hence, any Unix-based operating system should support FoCaLiZe. The currently tested Unix are: Fedora, Debian, Suse, BSD.

Windows users can run FoCaLiZe via the Unix-like environment Cygwin providing both users and developers tools. This software is freely distributed and available at `http://www.cygwin.com/`.

From the official Cygwin web site: "*Cygwin is a Linux-like environment for Windows. It consists of two parts: A DLL (cygwin1.dll) which acts as a Linux API emulation layer providing substantial Linux API functionality. A collection of tools which provide Linux look and feel. The Cygwin DLL currently works with all recent, commercially released x86 32 bit and 64 bit versions of Windows, with the exception of Windows CE. Cygwin is not a way to run native linux apps on Windows. You have to rebuild your application from source if you want it to run on Windows.*

Cygwin is not a way to magically make native Windows apps aware of UNIX ® functionality, like signals, ptys, etc. Again, you need to build your apps from source if you want to take advantage of Cygwin functionality."

Under Cygwin, the required packages are the same as those listed in 2.1 and 2.2. As stated in Cygwin's citation above, you need to get the sources packages of this software and compile them yourself, following information provided in these packages.

The installation of FoCaLiZe itself is the same for all operating systems and is described in the following section (2.4).

2.4 Installation

FoCaLiZe is currently distributed as a tarball containing the whole source code of the development environment. You must first deflate the archive (a directory will be created) by:

```
tar xvzf focalize-x.y.z.tgz
```

Where $x.y.z$ is the version number. Next, go in the sources directory:

```
cd focalize-x.x.x/
```

You now must configure the build process by:

```
./configure
```

The configuration script then asks for directories where to install the FoCaLiZe components. You may just press enter to keep the default installation directories.

```
latour:~/src/focalize$ ./configure ~/pkg
Where to install FoCaLiZe binaries ?
Default is /usr/local/bin.
Just press enter to use default location.

Where to install FoCaLiZe libraries ?
Default is /usr/local/lib/focalize.
Just press enter to use default location.
```

After the configuration ends, just build the system:

```
make all
```

And finally, get root priviledges to install the FoCaLiZe system:

```
su
make install
```

2.5 Compilation process and outputs

We call *compilation unit* a file containing source code for toplevel-definitions, species, collections. Visibility rules, described in section 3.1.13, are defined according to compilation units status. From a compilation unit, the compiler issues several files described thereafter.

2.5.1 Outputs

A FoCaLiZe development contains both computational code (i.e. code performing operations leading to an effect, a result) and logical properties.

When compiled, two outputs are generated:

- The "computational code" is compiled into OCaml source that can then be compiled with the OCaml compiler to lead to an executable binary. In this pass, logical properties are discarded since they do not lead to executable code.
- Both the "computational code" and the logical properties are compiled into a Coq or Dedukti model. This model can then be sent to the Coq proof assistant or the Dedukti logical framework who will verify the consistency of both the "computational code" and the logical properties (whose proofs must be obviously provided) of the FoCaLiZe development. This means that the Coq code generated is not intended to be used to generate an OCaml source code by automated extraction. As stated above, the executable generation is preferred using directly the generated OCaml code. In this idea, Coq acts as an assessor of the development instead of a code generator.
  More accurately, FoCaLiZe first generates a pre-Coq code and a pre-Dedukti code, i.e. a file containing Coq or Dedukti syntax plus "holes" in place of proofs written in the FoCaLiZe Proof Language. This kind of files is suffixed by ".zv" or ".sk.zv" instead of directly ".v" or ".sk". When sending this file to the prover these "holes" will be filled by effective Coq or Dedukti code automatically generated by the prover (if it succeed in finding a proof), hence leading to a pure code file that can be compiled by Coq or Dedukti.

In addition, several other outputs can be generated for documentation or debug purposes. See the section 7 for details.

2.5.2 Compiling a source

Compiling a FoCaLiZe program involves several steps (numbered here 1, 2, 3 and 4) that are automatically handled by the focalizec command. Using the command line options, it is possible to tune the code generations steps as described in 7.

1. FoCaLiZe **source compilation**. This step reads the FoCaLiZe source code and generates the OCaml and/or "pre-"Coq code and/or "pre-"Dedukti code. You can disable the code generation for one of these languages (see page 7), or several, or even all languages, in this case, no code is produced and you only get the FoCaLiZe object code produced without anymore else output and the process ends at this point. If you disable one of the target languages, then you won't get any generated file for it, hence no need to address its related compilation process described below.

   Assuming you generate code for OCaml, Coq, and Dedukti you will get three generated files: `source.ml` (the OCaml code), `source.zv` (the "pre-"Coq code) and `source.sk.zv` (the "pre-"Dedukti code).

2. OCaml **code compilation**. This step takes the generated OCaml code (it is an OCaml source file) and compile it. This is done like any regular OCaml compilation, the only difference is that the search path containing the FoCaLiZe installation path and your own used extra FoCaLiZe source files directories are automatically passed to the OCaml compiler. This step acts like the direct invocation:

   ```
   ocamlc -c -I /usr/local/lib/focalize -I mylibs
      -I myotherlibs source.ml
   ```

   This produces the OCaml object file `source.cmo`. Note that you can also ask to use the OCaml code in native mode, in this case the `ocamlopt` version of the OCaml compiler is selected (see OCaml reference manual for more information) and the object files are `.cmx` files instead of `.cmo`. ones.

3. **"Pre-"Coq code compilation**. This step reads the generated `.zv` file and produces a real Coq `.v` source file. The proofs written in the FoCaLiZe Proof Language are replaced by the effective Coq proofs found by the Zenon theorem prover. Note that if Zenon fails to find a proof, a *hole* appears in the final Coq `.v` file: the text "`TO_BE_DONE_MANUALLY.`" is written in place of an effective proof. The Coq compiler then obviously fails to compile the file, and the user must modify his original FoCaLiZe source file to provide a tractable proof script for Zenon or insert a direct Coq proof either in the FoCaLiZe or in the generated Coq source file. This step acts like the direct invocation:

   ```
   zvtov -new source.zv
   ```

   For more about the Zenon options, consult the section 5.1.3.

4. **"Pre-"Dedukti code compilation**. This step reads the generated `.sk.zv` file and produces a real Dedukti `.sk` source file. The proofs written in the FoCaLiZe Proof Language are replaced by the effective Dedukti proofs found by the Zenon Modulo theorem prover. Zenon Modulo [9] is an extension of Zenon that implements a powerful proof technique, Deduction Modulo [15], and is able to produce Dedukti proof terms [8]. This step acts like the direct invocation:

   ```
   zvtov -new source.sk.zv
   ```

   All Zenon options are also understood by Zenon Modulo.

5. Coq **code compilation**. This step takes the generated `.v` code and compiles it with Coq. This is done like any regular Coq compilation. The only difference is that the search path containing the FoCaLiZe installation path and your own used extra FoCaLiZe source files directories are automatically passed to the Coq compiler. This step acts like the direct invocation:

   ```
   coqc -I /usr/local/lib/focalize -I mylibs
     -I myotherlibs source.v
   ```

   Once this step is done, you have the Coq object files and you are sure that Coq validated you program model, properties and proofs. The final "assessor" of the tool-chain accepted your program.

6. Dedukti **code compilation**. This step takes the generated `.sk` code and compiles it with Dedukti. This is done like any regular Dedukti compilation. The only difference is that the search path containing the FoCaLiZe installation path and your own used extra FoCaLiZe source files directories are automatically passed to the Dedukti compiler. This step acts like the direct invocation:

   ```
   skcheck -e -I /usr/local/lib/focalize -I mylibs
     -I myotherlibs source.sk
   ```

   Once this step is done, you have the Dedukti object files and you are sure that Dedukti validated you program model, properties and proofs. The final "assessor" of the tool-chain accepted your program.

Once all separate files are compiled, to get an executable from the OCaml object files, you must link them together, providing the same search path than above and the `.cmo` files corresponding to all the generated OCaml files from all your FoCaLiZe `.foc` files. You also need to add the `.cmo` files corresponding to the modules of the standard library you use (currently, this must be done by the user, next versions will automate this process).

```
ocamlc -I mylibs -I myotherlibs
  install_dir/ml_builtins.cmo install_dir/basics.cmo
  install_dir/sets.cmo ...
  mylibs/src1.cmo mylibs/src2.cmo ...
  myotherlibs src3.cmo mylibs/src3.cmo ...
  source1.cmo source2.cmo ...
  -o exec_name
```

Chapter 3

The core language

3.1 Lexical conventions

3.1.1 Blanks

The following characters are considered as blanks: space, newline, horizontal tabulation, carriage return, line feed and form feed. Blanks are ignored, but they separate adjacent identifiers, literals and keywords that would otherwise be confused as one single identifier, literal or keyword.

3.1.2 Escaped characters

3.1.3 Comments

Comments are treated as blanks and discarded during the compilation process. FoCaLiZe features two kinds of comments, *uni-line comments* introduced by `--` and *general comments* enclosed between `(*` and `*)`.

Note: The three character sequences `(*`, `*)`, and `--` are named *comment delimiters*. Due to their particular lexical role the comment delimiters must be escaped in other regular tokens, in particular string and character literals.

3.1.3.1 General comments

The two characters `(*`, with no intervening blanks, start a *general comment*; the two characters `*)`, with no intervening blanks, close a general comment. General comments may span on any number of lines and may be arbitrarily nested. In addition, any legal FoCaLiZe program may be commented out via a general comment.

Note: Almost arbitrary text can be written inside a general comment, therefore general comments are used to add explanations in the code to help the reader.

Example:

```
(* The main species of the development: S.
   S contains only one method, m, since m is general enough to perform the
   entire work under any circumstance. *)
species S =
  ...
  let m (x in Self) = (* Another useful comment *)
  ...
```

```
end
;;
(* Another discarded comment at end of file *)
```

3.1.3.2 Uni-line comments

The two characters `--`, with no intervening blanks, start a *uni-line comment*; a uni-line comment always spreads to the end of the line.

Note: No general comment marker may appear in a uni-line comment.

Example:

```
-- Discarded uni-line comment
species S =
  let m (x in Self) = -- The powerful m method.
  ...
end
;;
```

Note that double quotes (symbol ") should not appear in comments, and that a spanning comment should not start with uni-line comment mark. A uni-line comment should also always be terminated by a carriage return (an unclosed uni-line comment cannot end a file).

3.1.4 Annotations

Annotations are introduced by the three characters `(**`, with no intervening blanks, and terminated by the two characters `*)`, with no intervening blanks. Annotations cannot occur inside string or character literals and cannot be nested. They must precede the construct they document. In particular, a **source file cannot end by an annotation**.

Unlike comments, annotations are kept during the compilation process and recorded in the compilation information ("`.fo`" files). Annotations can be processed later on by external tools that could analyse them to produce a new FoCaLiZe source code accordingly. For instance, the FoCaLiZe development environment provides the FoCaLiZeDoc automatic production tool that uses annotations to automatically generate documentation. Several annotations can be put in sequence for the same construct. We call such a sequence an **annotation block**. Using embedded tags in annotations allows third-party tools to easily find out annotations that are meaningful to them, and safely ignore others. For more information, consult 8. Example:

```
(** Annotation for the automatic documentation processor.
    Documentation for species S. *)
species S =
  ...
  let m (x in Self) =
    (** {@TEST} Annotation for the test generator. *)
    (** {@MY_TAG_MAINTAIN} Annotation for maintainers. *)
    ... ;
end ;;
```

3.1.5 Identifiers

FoCaLiZe features a rich class of identifiers with sophisticated lexical rules that provide fine distinction between the kind of notion a given identifier can designate.

3.1.5.1 Introduction

Sorting words to find out which kind of meaning they may have is a very common conceptual categorisation of names that we use when we write or read ordinary English texts. We routinely distinguish between:

- a word only made of lowercase characters, that is supposed to be an ordinary noun, such as “table”, “ball”, or a verb as in “is”, or an adjective as in “green”,
- a word starting with an uppercase letter, that is supposed to be a name, maybe a family or Christian name, as in “Kennedy” or “David”, or a location name as in “London”.

We use this distinctive look of words as a useful hint to help understanding phrases. For instance, we accept the phrase ”my ball is green” as meaningful, whereas ”my Paris is green” is considered a nonsense. This is simply because ”ball” is a regular noun and ”Paris” is a name. The word ”ball” as the right lexical classification in the phrase, but ”Paris” has not. This is also clear that you can replace ”ball” by another ordinary noun and get something meaningful: ”my table is green”; the same nonsense arises as well if you replace ”Paris” by another name: ”my Kennedy is green”.

Natural languages are far more complicated than computer languages, but FoCaLiZe uses the same kind of tricks: the “look” of words helps a lot to understand what the words are designating and how they can be used.

3.1.5.2 Conceptual properties of names

FoCaLiZe distinguishes 4 concepts for each name:

- the *fixity* assigns the place where an identifier must be written,
- the *precedence* decides the order of operations when identifiers are combined together,
- the *categorisation* fixes which concept the identifier designates.
- the *nature* of a name can either be symbolic or alphanumeric.

Those concepts are compositional, i.e. all these concepts are independent from one another. Put is another way: for any fixity, precedence, category and nature, there exist identifiers with this exact set of properties.

We further explain those concepts below.

3.1.5.3 Fixity of identifiers

The fixity of an identifier answers to the question “where this identifier must be written ?”.

- a *prefix* is written *before* its argument, as sin in $sin\ x$ or $-$ in $-y$,
- an *infix* is written *between* its arguments, as $+$ in $x\ +\ y$ or mod in $x\ mod\ 3$.
- a *mix-fix* is written *among* its arguments, as *if ... then ... else ...* in *if c then 1 else 2*.

In FoCaLiZe, all the ordinary identifiers are prefix and all the binary arithmetics operators are infix as they are in mathematics.

3.1.5.4 Precedence of identifiers

The precedence rules out where implicit parentheses take place in a complex combination of symbols. For instance, according to the usual mathematical conventions:

- $1 + 2 * 3$ means $1 + (2 * 3)$ hence 7, it does not mean $(1 + 2) * 3$ which is 9,
- $2 * 3^{4} + 5$ means $(2 * (3^{4})) + 5$ hence 167, it does not mean $((2 * 3)^{4}) + 5$ which is 1301, nor $2 * (3^{(4+5)})$ which is 39366.

In FoCaLiZe, all the binary infix operators have the precedence they have in mathematics.

3.1.5.5 Categorisation of identifiers

The category of an identifier answers to the question "is this identifier a possible name for this kind of concept ?". In programming languages categories are often strict, meaning that the category exactly states which concept attaches to the identifier.

For FoCaLiZe there are two categories of identifiers, the *lowercase* and the *uppercase* identifiers.

- a *lowercase identifier* designates a simple entity of the language. It may name some of the language expressions, a function name, a function parameter or bound variable name, a method name, a type name, or a record field label name.
- an *uppercase identifier* designates a more complex entity in the language. It may name a sum type constructor name, a module name, a species or a collection name.

Roughly speaking, the first letter of an identifier fixes its category: if the first letter is lowercase the identifier is lowercase, and conversely if an identifier starts with an uppercase letter it is indeed an uppercase identifier.

More precisely, we classify identifiers by looking at their *starter character*: the starter character is the first character of the identifier that is not an underscore. Considering underscores as some sort of spacing marks into names, the starter character is the first "meaningful" character of the identifier,

Recall that the category of identifiers is orthogonal to the rest of their properties. For instance, the fixity of an identifier does not tell if the identifier is lowercase or not. Put it another way: as stated above, + is infix but this does not say if + is a lowercase or uppercase identifier! (In fact + is lowercase, since as stated below its `'+'` starter character is a lowercase symbolic character.)

3.1.5.6 Nature of identifiers

In FoCaLiZe identifiers are either:

- *symbolic*: the identifier contains characters that are not letters. `+`, `:=`, `->`, `+float` are symbolic.
- *alphanumeric*: the identifier only contains letters, digits and underscores. `x`, `_1`, `Some`, `Basic_object` are alphanumeric.

3.1.5.7 Alphanumeric identifiers

An alphanumeric identifier is a sequence of letters, digits, and _ (the underscore character). Letters contain at least the 52 lowercase and uppercase letters from the standard ASCII set. In an identifier, all characters are meaningful.

Alphanumeric lowercase identifiers designate the names of variables, functions, types. and labels of records.

Alphanumeric uppercase identifiers designate the names of constructors, species, and collections.

$$
\begin{array}{rcl}
\textit{digit} & ::= & \texttt{0} \ldots \texttt{9} \\
\textit{lower} & ::= & \texttt{a} \ldots \texttt{z} \\
\textit{upper} & ::= & \texttt{A} \ldots \texttt{Z} \\
\textit{letter} & ::= & \textit{lower} \mid \textit{upper} \\
\textit{lident} & ::= & \{\texttt{\_}\}^* \ \textit{lower} \ \{\textit{letter} \mid \textit{digit} \mid \texttt{\_}\}^* \\
\textit{uident} & ::= & \{\texttt{\_}\}^* \ \textit{upper} \ \{\textit{letter} \mid \textit{digit} \mid \texttt{\_}\}^* \\
\textit{ident} & ::= & \textit{lident} \mid \textit{uident}
\end{array}
$$

Roughly speaking, an alphanumeric lowercase identifier is a sequence of letters, digits, and _ (the underscore character), starting with a lowercase alphanumeric letter (a lowercase letter, a digit, or an underscore).

More precisely, an alphanumeric identifier is lowercase if its starter character (or first "meaningful" letter) is lowercase.

Examples: `foo`, `bar`, `_20`, `___gee_42` are lowercase alphanumeric identifiers; `foo`, `bar`, `_20`,

Roughly speaking, an alphanumeric uppercase identifier is a sequence of letters, digits, and _ (the underscore character), starting with an uppercase letter.

More precisely, an alphanumeric identifier is uppercase if its starter character (or first "meaningful" letter) is uppercase.

Examples: `Some`, `None`, `_One_`, `Basic_object`, `___GEE_42` are uppercase alphanumeric identifiers.

3.1.5.8 Infix/prefix operators

FoCaLiZe allows infix and prefix operators built from a "starting operator character" and followed by a sequence of regular identifiers or operator characters. For example, all the following are legal operators: `+`, `++`, `~+zero`, `=_mod_5`.

The position in which to use the operator (i.e. infix or prefix) is determined by the position of the first operator character according to the following table:

Prefix	Infix
` ~ ? $! #	, + - * / % & | : ; < = > @ ^ \

$$
\begin{array}{rcl}
\textit{prefix-char} & ::= & \texttt{`} \mid \texttt{\~{}} \mid \texttt{?} \mid \texttt{\$} \mid \texttt{!} \mid \texttt{\#} \\
\textit{infix-char} & ::= & \texttt{,} \mid \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{\&} \mid \texttt{|} \mid \texttt{:} \mid \texttt{;} \mid \texttt{<} \mid \texttt{=} \mid \texttt{>} \mid \texttt{@} \mid \texttt{\^{}} \mid \texttt{\textbackslash} \\
\textit{prefix-op} & ::= & \textit{prefix-char} \ \{\textit{letter} \mid \textit{prefix-char} \mid \textit{infix-char} \mid \textit{digit} \mid \texttt{\_}\}^* \\
\textit{infix-op} & ::= & \textit{infix-char} \ \{\textit{letter} \mid \textit{prefix-char} \mid \textit{infix-char} \mid \textit{digit} \mid \texttt{\_}\}^* \\
\textit{operator} & ::= & \textit{infix-op} \mid \textit{prefix-op}
\end{array}
$$

Hence, `+`, `++` and `=_mod_5` are infix symbolic operators and `~+zero` is a prefix symbolic one.

Note that symbolic character starters are classified into disjoint sets of uppercase and lowercase characters.

```
| '*'
| '+'  | '-'
| '/'  | '%'  | '&'  | '|'  | '<'  | '='  | '>'  | '@'  | '^'  | '\\'
```

are lowercase infix starters; `,` `?,` `$` are lowercase prefix starters; `:,` `` ` `` are uppercase infix starters; `(,` `[,` `{` are uppercase prefix starters.

For instance, `+` and `@++` are lowercase infix identifiers. Since its starter is `:`, `::` is an uppercase infix identifier; hence, as any other uppercase identifier, `::` could designate a value constructor name for a union type. Similarly `[]` is an uppercase prefix identifier that also could designate a value constructor. On the contrary, no notation starting with a `:` can designate a method name.

3.1.5.9 Defining an infix operator

The notion of infix/prefix operator does not mean that FoCaLiZe defines all these operators: it means that the programmer may freely define and use them as ordinary prefix/infix operators instead of only writing prefix function names and regular function application. For instance, if you do not like the FoCaLiZe predefined `^` operator to concatenate strings, you can define your own infix synonym for `^`, say `++`, using:

```
let ( ++ ) (s1, s2) = s1 ^ s2 ;
```

Then you can use the `++` operator in the usual way

```
let hw = "Hello" ++ "␣world!" ;
```

As shown in the example, at definition-time, the syntax requires the operator to be embraced by parentheses. More precisely, you must enclose the operator between **spaces** and parentheses. You must write `( + )` with spaces, not simply `(+)` (which leads to a syntax error anyway).

3.1.5.10 Prefix form notation

The notation `( op )` is named the *prefix form notation* for operator `op`.

Since you can only define prefix identifiers in FoCaLiZe, you must use the prefix form notation to define an infix or prefix operator.

When a prefix or infix operator has been defined, it is still possible to use it as a regular identifier using its prefix form notation. For instance, you can use the prefix form of operator `++` to apply it in a prefix position as a simple regular function:

```
( ++ ) ("Hello", "␣world!") ;
```

Warning! A common error while defining an operator is to forget the blanks around the operator. This is particularly confusing, if you type the `*` operator without blanks around the operator: you write the lexical entity `(*)` which is the beginning (or the end) of a comment!

The FoCaLiZe notion of symbolic identifiers goes largely beyond simple infix operators. Symbolic identifiers let you assign sophisticated names to your functions and operators. For instance, instead of creating a function to check if integer `x` is equal to the predecessor of integer `y`, as in

```
let is_eq_to_predecessor (x, y) = ... ;
... if is_eq_to_predecessor (5, 7) ... ;
```

it is possible to directly define

```
let ( =pred ) (x, y) = ... ;
... if 5 =pred 7 ... ;
```

Attention : since a comma can start an infix symbol, be careful when using commas to add a space after each comma to prevent confusion. In particular, when using commas to separate tuple components, always type a space after each comma. For instance, if you write `(1,n)` then the lexical analyser finds only two words: the integer `1` as desired, then the infix operator `,n` which is certainly not the intended meaning. Hence, following usual typography rules, always type a space after a comma (unless you have define a special operator starting by a comma).

Rule of thumb: The prefix version of symbolic identifiers is obtained by enclosing the symbol between spaces and parens.

3.1.6 Extended identifiers

Moreover, FoCaLiZe has special forms of identifiers to allow using spaces inside or to extend the notion of operator identifiers.

- **Delimited alphanumerical identifiers**. They start by two ` (backquote) characters and end by two ' (quote) characters. In addition to usual alpha-numerical characters, the delimited identifiers can have spaces. For example: ``` ``equal is reflexive'' ```, ``` ``fermat conjecture'' ```.
- **Delimited symbolic identifiers**. They are delimited by the same delimiter characters and contain symbolic characters.

As usual, the first meaningful character at the beginning of a delimited identifier rules out its conceptual properties. For instance, ``` ``equal is reflexive'' ``` has `e` as its first meaningful character; hence ``` ``equal is reflexive'' ``` is alphanumeric, prefix, lowercase, and has the same precedence as any other regular function. Similarly, ``` ``+ for matrices'' ``` starting with the + symbol is symbolic, infix, lowercase, and has the same precedence as the + operator.

3.1.7 Species and collection names

Species, collection names and collection parameters are uppercase identifiers.

3.1.8 Integer literals

binary-digit ::= `0` | `1`

octal-digit ::= `0` . . . `7`

hexadecimal-digit ::= `0` . . . `9` | `A` . . . `F` | `a` . . . `f`

sign ::= `+` | `-`

unsigned-binary-literal ::= `0` (`b` | `B`) *binary-digit* {*binary-digit* | `_`}*

unsigned-octal-literal ::= `0` (`o` | `O`) *octal-digit* {*octal-digit* | `_`}*

unsigned-decimal-literal ::= *digit* {*digit* | `_`}*

unsigned-hexadecimal-literal ::= `0` (`x` | `X`) *hexadecimal-digit* {*hexadecimal-digit* | `_`}*

unsigned-integer-literal ::= *unsigned-binary-literal*
| *unsigned-octal-literal*
| *unsigned-decimal-literal*
| *unsigned-hexadecimal-literal*

integer-literal ::= [*sign*] *unsigned-integer-literal*

An integer literal is a sequence of one or more digits, optionally preceded by a minus or plus sign and/or a base prefix. By default, i.e. without a base prefix, integers are in decimal. For instance: `0`, `-42`, `+36`. FoCaLiZe syntax allows to also specify integers in other bases by preceding the digits by the following prefixes:

- **Binary**: base 2. Prefix is `0b` or `0B`. Digits are [0-1].
- **Octal**: base 8. Prefix is `0o` or `0O`. Digits are [0-7].
- **Hexadecimal**: base 16. Prefix is `0x` or `0X`. Digits are [0-9] [A-F] [a-f].

Here are various examples of integers in various bases: `-0x1Ff`, `0B01001`, `+0o347`, `-OxFF_FF`.

3.1.9 String literals

string-literal ::= `"` {*plain-char* | `\` *char-escape*}* `"`

plain-char ::= *any printable character except backslash* (`\`) *and double quote* (`"`)

char-escape ::= `b` | `n` | `r` | `t`
| `␣` | `"` | `'` | `*` | `\` | `` ` `` | `-`
| `(` | `)` | `[` | `]` | `{` | `}` | `%`
| *digit digit digit*
| *hexadecimal-digit hexadecimal-digit*

String literals are sequences of any characters delimited by `"` (double quote) characters (*ipso facto* with no intervening `"`). Escape sequences (meta code to insert characters that can't appear simply in a string) available in string literals are summarised in the table below:

<table>
<tr><th>Sequence</th><th>Character</th><th>Comment</th></tr>
<tr><td>\b</td><td>\008</td><td>Backspace.</td></tr>
<tr><td>\n</td><td>\010</td><td>Line feed.</td></tr>
<tr><td>\r</td><td>\013</td><td>Carriage return.</td></tr>
<tr><td>\t</td><td>\009</td><td>Tabulation.</td></tr>
<tr><td>\␣</td><td>␣</td><td>Space character.</td></tr>
<tr><td>\"</td><td>"</td><td>Double quote.</td></tr>
<tr><td>\'</td><td>'</td><td>Single quote.</td></tr>
<tr><td>*</td><td>*</td><td>Allows e.g. for insertion of “(*” in a string</td></tr>
<tr><td>\(</td><td>(</td><td rowspan="6">See comment above for *</td></tr>
<tr><td>\)</td><td>)</td></tr>
<tr><td>\[</td><td>[</td></tr>
<tr><td>\]</td><td>]</td></tr>
<tr><td>\{</td><td>{</td></tr>
<tr><td>\}</td><td>}</td></tr>
<tr><td>\\</td><td>\</td><td>Backslash character.</td></tr>
<tr><td>\`</td><td>‘</td><td>Backquote character.</td></tr>
<tr><td>\-</td><td>−</td><td>Minus (dash) character. As for multi-line comments, uni-line comments can’t appear in strings. Hence, to insert the sequence “−−” use this escape sequence twice.</td></tr>
<tr><td>\digit digit digit</td><td></td><td>The character whose ASCII code in decimal is given by the 3 digits following the \. This sequence is valid for all ASCII codes.</td></tr>
<tr><td>\0x hex hex</td><td></td><td>The character whose ASCII code in hexadecimal is given by the 2 characters following the \. This sequence is valid for all ASCII codes.</td></tr>
</table>

3.1.10 Character literals

character-literal ::= ' (*plain-char* | \ *char-escape*) '

Characters literals are composed of one character enclosed between two “'” (quote) characters. Example: 'a', '?'. Escape sequences (meta code to insert characters that can’t appear simply in a character literal) must also be enclosed by quotes. Available escape sequences are summarised in the table above (see section 3.1.9).

3.1.11 Floating-point number literals

decimal-literal ::= [*sign*] *unsigned-decimal-literal*

hexadecimal-literal ::= [*sign*] *unsigned-hexadecimal-literal*

scientific-notation ::= `e` | `E`

unsigned-decimal-float-literal ::= *unsigned-decimal-literal* [`.` {*unsigned-decimal-literal*}*] [*scientific-notation decimal-literal*]

unsigned-hexadecimal-float-literal ::= *unsigned-hexadecimal-literal* [`.` {*unsigned-hexadecimal-literal*}*] [*scientific-notation hexadecimal-literal*]

unsigned-float-literal ::= *unsigned-decimal-float-literal*
| *unsigned-hexadecimal-float-literal*

float-literal ::= [*sign*] *unsigned-float-literal*

Floating-point numbers literals are made of an optional sign ('+' or '-') followed by a non-empty sequence of digits followed by a dot ('.') followed by a possibly empty sequence of digits and finally an optional scientific notation ('e' or 'E' followed an optional sign then by a non-empty sequence of digits. FoCaLiZe allows floats to be written in decimal or in hexadecimal. In the first case, digits are [0-9]. Example: `0.`, `-0.1`, `1.e-10`, `+5E7`. In the second case, they are [0-9 a-f A-F] and the number must be prefixed by "0x" or "0X". Example `0xF2.E4`, `0X4.3A`, `Ox5a.a3eef`, `Ox5a.a3e-ef`.

3.1.12 Proof step bullets

proof-step-bullet ::= `<` {*digit*}+ `>` {*letter* | *digit*}+

A proof step bullet is a non-negative non-signed integer literal (i.e. a non empty sequence of [0-9] characters) delimited by the characters < and >, followed by a non-empty sequence of alphanumeric characters (i.e. [A-Z a-z 0-9]). The first part of the bullet (i.e. the integer literal) stands for the depth of the bullet and the second part stands for its name. Example:

```
<1>1 assume ...
     ...
     prove ...
  <2>1 prove ... by ...
  <2>f qed by step <2>1 property ...
<1>2 conclude
```

3.1.13 Name qualification

Name qualification is done according to the compilation unit status.

As precisely described in section (4.1.1), toplevel-definitions include species, collections, type definitions (and their constitutive elements like constructors, record fields), toplevel-theorems and toplevel-functions. Any toplevel-definition (thus outside species and collections) is visible all along the compilation unit after its apparition. If a toplevel-definition is required by another compilation unit, you can reference it by referencing the external compilation unit (with a `use` or a `open` command) and then **qualifying** its name, i.e. making explicit the compilation unit's name before the definition's name using the '#' character as delimiter. Examples:

- `basics#string` stands for the type definition of `string` coming from the source file "basics.fcl".
- `basics#Basic_object` stands for the species `Basic_object` defined in the source file "basics.fcl".
- `db#My_db_coll!create` stands for the method `create` of a collection `My_db_coll` hosted in the source file "db.fcl".

The qualification can be omitted by using the `open` directive that loads the interface of the argument compilation unit and make it directly visible in the scope of the current compilation unit. For instance:

```
use "basics";;
species S = inherit basics#Basic_object; ... end ;;
```

can be transformed with no explicit qualification into:

```
open "basics";;
species S = inherit Basic_object; ... end ;;
```

After an `open` directive, the definitions of loaded (object files of) compilation units are added in head of the current scope and mask existing definitions wearing the same names. For example, in the following program:

```
(* Redefine my basic object, containing nothing. *)
species Basic_object = end ;;
open "basics";;
species S = inherit Basic_object; ... end ;;
```

the species `S` inherits from the last `Basic_object` in the scope, that is the one loaded by the `open` directive and not from the one defined at the beginning of the program. It is still possible to recover the first definition by using the "empty" qualification `#Basic_object` in the definition of `S`:

```
(* Redefine my basic object, containing nothing. *)
species Basic_object = end ;;
open "basics";;
species S = inherit #Basic_object; ... end ;;
```

The qualification starting by a '#' character without compilation unit name before stands for "the definition at toplevel of the current compilation unit".

3.1.14 Reserved keywords

The identifiers below are reserved keywords that cannot be employed otherwise:

```
alias all and as assume assumed
begin by
caml collection conclude constructor coq coq_require
definition
else end ex external
false final function
hypothesis
if in inherit internal implement is
```

```
let lexicographic local logical
match measure
not notation
of on open or order
proof prop property prove
qed
rec representation
Self signature species step structural
termination then theorem true type
use
with
```

Keywords of Coq, Dedukti, or OCaml (for example `Set`, `class`), can be safely used in a FoCaLiZe source code.

Some symbols (such as `:`) are also reserved, and cannot be used to name methods. It is still possible to use such symbols as first character of a symbolic identifier.

3.2 Language constructs and syntax

3.2.1 Types

Before dealing with expressions and in general, constructs that allow to compute, let us first examine data-type definitions since, to emit its result, an algorithm must manipulate data that are more or less specific to the algorithm. Hence we must know about type definitions to define data that have a convenient shape and carry the necessary information to model the problem at hand.

Type definitions allow to build new types or more complex types by combining previously existing types. They always appear as toplevel-definitions, in other words, outside species and collections. Hence a type definition is visible in the whole compilation unit (and also in other units by using the `open` directive or by qualifying the type name as described in section 3.1.13).

3.2.1.1 Type constructors

A **type constructor** is, roughly speaking, a type name.

FoCaLiZe provides the basic built-in types (constructors):

- `int` for signed machine integers,
- `bool` for boolean values (`true` and `false` that are hardwired in the syntax or `True` and `False` that are defined in "basics.fcl"),
- `float` for floating point numbers,
- `unit` for the trivial type whose only value is `()`,
- `char` for characters literals,
- `string` for strings literals.

Note that these types are translated to OCaml types; for example `int` on 32-bits architecture encode values between -2^{30} and $+2^{30}-1$.

New type constructors are introduced by **type definitions**. Types constructors can be parameterised by **type expressions** separated by commas and between parentheses.

3.2.1.2 Type expressions

Type definitions require **type expressions** to build more complex data-types.

$$
\begin{array}{rcl}
\textit{type-exp} & ::= & \textit{lident} \\
& | & \textit{uident} \\
& | & \texttt{Self} \\
& | & \texttt{'}\textit{lident} \\
& | & \textit{lident}\ \texttt{(}\ \{\textit{type-exp}\}(\texttt{,})^{+}\ \texttt{)} \\
& | & \textit{type-exp}\ \texttt{->}\ \textit{type-exp} \\
& | & \texttt{(}\ \{\textit{type-exp}\}(\texttt{*})^{+}\ \texttt{)} \\
& | & \texttt{(}\ \textit{type-exp}\ \texttt{)}
\end{array}
$$

A type expression can be a type constructor.

A type expression can denote the representation of a species or a collection by using their name, thus a capitalized name. The special case of `Self` denotes the representation of the current species. Hence, obviously `Self` is only bound in the scope of a species.

Type expressions representing function types are written using the arrow notation (`->`) in which the type of the argument of the function is the left type expression and its return type is the right one. As usual in functional languages, a function with several (say n) arguments is considered as a function with **1** argument returning a function with $n - 1$ arguments. Hence, `int -> int -> bool` is the type of a function taking 2 integers and returning a boolean.

FoCaLiZe provides native tuples (generalisation of pairs). The type of a tuple is the type of each of its components separated by a * character and surrounded by parentheses. Hence, `(int * bool * string)` is the type of triplets whose first component is an integer, second component is a boolean and third component is a string, e.g. `(-3, true, "test")`.

Finally, type expressions can be written between parentheses without changing their semantics.

3.2.1.3 Type definitions

A type **definition** introduces a new type constructor (the name of the type), which becomes available to build new type expressions. Hence, defining a type is the way to give a name to a new type structure. FoCaLiZe proposes 3 kinds of type definitions: aliases, sum types and record types.

Aliases

Aliases provide a way to create type abbreviations. It is not handy to manipulate large **type expressions** like for instance, a tuple of 5 components: `(int * int * int * int * int)`. Moreover, several kind of information can be represented by such a tuple. For instance, a tuple with x, y, z (3D-coordinates), temperature and pressure has the same type as a tuple with year, month, day, hours, minutes. In these two cases, the manipulated type expression is the same and the two uses cannot be easily differentiated. Type aliases allows to give a name to a (complex) type expression, for sake of readability or to shorten the code. Example:

```
type experiment_conditions = alias (int * int * int * int * int) ;;
type date = alias (int * int * int * int * int) ;;
```

alias-type-def ::= `type` *ident* `=` `alias` *type-exp*

In the remaining of the development, the type names `experiment_conditions` and `date` will be known to be tuples of 5 integers and will be compatible with any other type being also a tuple of 5 integers. This especially means that a *type alias does not create a really "new" type, it only gives a name to a type expression and this name is type-compatible with any occurrence of the type expression it is bound to*. Obviously, it is possible to use aliases with and in any type expression or type definition.

Sum types

Sum types provide the way to create new **values** that belong to the same **type**. Like `1` or `42` are **values** of **type** `int`, one may want to have `Red`, `Blue` and `Green` as the **only** values of a new type called `color`. The **only** means that the created type `color` is inhabited only by these 3 values. To define such a type, we itemize its value names (that are always capitalized identifiers) by preceeding them by a `|` character :

```
type color =
  | Red
  | Blue
  | Green
;;
```

Note that the first `|` character is required: it is not a separator. This especially means that when writing a sum type definition on a single line, the first `|` must be written:

```
type color = | Red | Blue | Green ;;
```

Values of a sum type are built from the **value constructors**, i.e. from the names enumerated in the definition (that must not be confused with the **type constructor** which is the name of the type. For, instance, `Red` is a **value** of the type constructor `color`.

Value constructors of sum types can be **parameterised** by a type expressions, corresponding values being obtained by applying the value constructor to values of the parameters types. For instance, let's define the type of playing cards as king, queen, jack and simply numbered cards:

```
type card =
  | King
  | Queen
  | Jack
  | Numbered (int)
;;
```

Hence, the `Numbered` constructor "carries" the integer value written on the card. Some values of type `card` are: `King`, `(Numbered 4)`, `(Numbered 42)`. The `Numbered` constructor has **parameter**.

An important attention must be taken for constructors having "several" arguments. FoCaLiZe provides 2 different ("type-incompatible") ways to make a value constructor carrying several values.

- Either the constructor has **1** argument that is a tuple, i.e. a type expression involving the `*` constructor. The corresponding type definition for such a type would be:

  ```
  type t =
    | Cstr (bool * int * string)
  ;;
  ```

- Or the constructor has **several** arguments, i.e. several type expressions separated by a comma.

```
type t2 =
  | Cstr2 (bool, int, string)
;;
```

means that the constructor `Cstr2` has **3 arguments**.

This especially important since when matching on such value constructors, confusing the arguments of `Cstr2` with one and unique tuple with 3 components will result in a type error. Below are shown several pieces of source code with valid/invalid mixes between these concepts.

```
type mytuple = alias (bool * int * string) ;;

type t =
  | Cstr (bool * int * string)
;;

let fct_t1 (x) =
  match x with
   | Cstr (a, b, c) -> ()
;;
```

leads to:

```
Error: Types
(basics#bool * basics#int * basics#string), '_a, '_b and
(basics#bool * basics#int * basics#string) are not compatible.
```

because `Cstr` expects 1 tuple argument and not 3 arguments.

```
let fct_t1 (x) =
  match x with
   | Cstr ((a, b, c)) -> ()
;;
```

is accepted since it makes explicit that the pattern matches the unique argument of `Cstr` that is a tuple and by the way de-structurates this tuple.

```
let fct_t2 (x) in mytuple =
  match x with
   | Cstr (x) -> x
;;
```

is accepted since the type `mytuple` aliases a 3 components tuples and `Cstr` is really parametrised by 1 argument that is a 3 components tuple.

```

let fct_t2 (x) =
  match x with
   | Cstr2 (a, b, c) -> ()
;;
```

NOTA: TO BE COMPLETED.

Any type expression, even recursive, can be used as a parameter of value contructors. For instance, the type of lists of boolean $\times$ integer pairs could be defined like:

```
type b_i_list =
  | Empty
  | Cons ((bool * int) * b_i_list)
;;
```

From this type definition, a value of type `b_i_list` is either empty (constructor `Empty`) or has a head (the first component of the `Cons` constructor) and a tail list (the second component of this constructor): `Cons ((false, 2), (Cons ((true, 1), Empty)))`. The length of this list is 2 and its elements are `(false, 2)` followed by `(true, 1)`.

$$
\begin{aligned}
\textit{type-params} &::= \texttt{(}\ \{\texttt{'}\ \textit{lident}\}(\texttt{,})^{+}\ \texttt{)} \\
\textit{type-args} &::= \texttt{(}\ \{\textit{type-exp}\}(\texttt{,})^{+}\ \texttt{)} \\
\textit{constructor} &::= \texttt{|}\ \textit{uident}\ [\textit{type-args}] \\
\textit{sum-type-def} &::= \texttt{type}\ \textit{lident}\ [\textit{type-params}]\ \texttt{=}\ \{\textit{constructor}\}^{+}
\end{aligned}
$$

Record types

Record types provide a way to aggregate data of various types, a bit like tuples, but naming the components of the group, instead of differentiating them by their position like in tuples. A record is a sequence of names and types between braces. For example:

```
type experiment_conditions = {
  x : int ;
  y : int ;
  z : int ;
  temperature : int ;
  pressure : int
} ;;

type identity = {
  name : string ;
  birth : int ;
  living : bool
} ;;
```

$$
\begin{aligned}
\textit{field} &::= \textit{lident}\ \texttt{=}\ \textit{type-exp}\ \texttt{;} \\
\textit{record-type-def} &::= \texttt{type}\ \textit{lident}\ \textit{type-params}\ \texttt{=}\ \texttt{\{}\ \{\textit{field}\}^{+}\ \texttt{\}}
\end{aligned}
$$

To create a **value** of a record type, a value must be provided for each field of the record.

```
{ name = "Benjamin" ; birth = 2003 ; living = true }
```

Like in tuples, records can mix types of fields.

Parameterised type definitions

It is possible, *at toplevel*, to **parameterise a type definition** with a **type variable** that can be instantiated by any type expression. A type variable is written as an identifier preceded by a `'` (quote) character.

For instance, the type definition of generic (polymorphic) lists may be defined by:

```
type list ('a) =
  | Empty
  | Cons ('a,  list ('a))
;;
```

The value constructor `Cons` carries a value of type "variable" and a tail of type `list` with its parameter instantiated by the same type variable. This explicitly says that all the elements of such a list have the same type. It is now possible to use the `list` type in type **expressions** by providing a type **expression** as argument of the **type constructor** `list`. For instance, `list (int)` is the type of lists containing integers, `list (list (char))` is the type of lists containing lists of characters.

Parameterised record types can also be introduced, as in the following example:

```
type pair ('a, 'b) = {
  first : 'a ;
  second : 'b
} ;;

type int_bool_pair = pair (int, bool) ;;
```

3.2.2 Mutually Recursive Types Definitions

Only sums. Syntax: **`type`** ldots **`with`** ...

3.2.3 Type-checking

The type-checking process is roughly similar to ML type-checking. Polymorphic types are allowed at top-level. However, methods are not allowed to be polymorphic. This means that their types cannot contain variables. But they may contain collection parameters as stated in section 4.2.1.

A type t_1 is an **instanciation** of a type t_2 if t_1 is obtained by replacing some type variables of t_2 by a "more defined type expression".

For example, $'a \rightarrow$ `int` $\rightarrow$ `bool` is an instanciation of $'a \rightarrow$ `int` $\rightarrow' c$ since we replaced the variable $'c$ by the type `bool`.

Two types t_1 and t_2 are said **compatible** if they have a **common** instanciation. For the intuition, this means that there is an instanciation of the variables in t_1 and an instanciation of the variables in t_2 such that these instanciations make t_1 and t_2 the same type. Note that type variables appearing in different type expressions are different variables, that is $'a \rightarrow$ `int` and `bool` $\rightarrow' a$ are compatible.

For example, we consider the two following types:

- $t_1 =' a \rightarrow \texttt{int} \rightarrow' b \rightarrow' c$
- $t_2 = \texttt{bool} \rightarrow' d \rightarrow' d \rightarrow' e$

In t_1 we replace: $'a$ by `bool`, and we leave the others variables unchanged. We get the new type $t'_1 = \texttt{bool} \rightarrow \texttt{int} \rightarrow' b \rightarrow' c$.

In t_2, we replace $'d$ by `int`, $'e$ by $'c$. We get the new type $t'_2 = \texttt{bool} \rightarrow \texttt{int} \rightarrow \texttt{int} \rightarrow' c$.

The type t'_1 is an instanciation of t_1. The type t'_2 is an instanciation of t_2. The two types t'_1 and t'_2 are structurally the same. Hence t_1 and t_2 are **compatible**.

As it can be seen, an instanciation does not need to give a value to all the type variables.

For the sake of intuition, compatibility is a generalisation of the notion of types being “equal”. The most trivial instanciation appears when the two types do not have any type variables; in this case they are compatible iff they are structurally equal. This illustrates the common view of “being a good type” when for instance providing an argument to a function according to the type of the expected argument in the function’s prototype.

3.2.4 Representations

As further explained (see section 4.1.2) the representation is a method of a species that describes the internal data structure that the species manages. Hence, it is a kind of **type definition**, more accurately an **alias type definition**. This means that a representation does not introduce a new type, it only “assigns” to the representation a **type expression** defining the type of the manipulated entities of the species. Moreover, like for any other methods (cf. section 4.1.2), *the representation must not be a polymorphic type*. Thus its definition cannot contain type variables (but may contain collection parameter names). Defining a species’ representation is simply done by adding the `representation` method:

```
open "basics" ;;

species IntPair =
  representation = (int * int) ;
end ;;
```

Recall that the type introduced by the method `representation` is denoted by `Self` within the species.

representation ::= `representation =` *type-exp*

3.2.5 Expressions

Expressions are constructs of the language that are evaluated into a **value** of a certain **type**. Hence values and types are not at the same level. Types serve to classify values into categories. Although proofs may contain expressions, we describe them in Sec. 5. Indeed proofs are not expressions, they do not lead to FoCaLiZe values thus live at another level.

$$
\begin{array}{rcl}
\textit{qualified-uident} & ::= & [[\textit{ident}]\ \texttt{\#}]\ \textit{uident} \\
\textit{qualified-lident} & ::= & [[\textit{ident}]\ \texttt{\#}]\ \textit{lident} \\
\textit{lident-or-operator} & ::= & \textit{lident} \mid \texttt{(}\ \textit{operator}\ \texttt{)} \\
\textit{method-ident} & ::= & [\texttt{Self}]\ \texttt{!}\ \textit{lident-or-operator} \\
 & \mid & [\textit{lident}\ \texttt{\#}]\ \textit{uident}\ \texttt{!}\ \textit{lident-or-operator} \\
\textit{exp} & ::= & \textit{integer-literal} \\
 & \mid & \textit{string-literal} \\
 & \mid & \textit{character-literal} \\
 & \mid & \textit{float-literal} \\
 & \mid & \texttt{true} \mid \texttt{false} \\
 & \mid & \textit{qualified-uident} \\
 & \mid & \textit{method-ident} \\
 & \mid & \texttt{let rec}\ \{\textit{let-binding}\}(\texttt{and})^{+}\ \texttt{in}\ \textit{exp} \\
 & \mid & \texttt{if}\ \textit{exp}\ \texttt{then}\ \textit{exp}\ \texttt{else}\ \textit{exp} \\
 & \mid & \texttt{match}\ \textit{exp}\ \texttt{with}\ \{\textit{match-binding}\}^{+} \\
 & \mid & \textit{exp}\ \texttt{(}\ \{\textit{exp}\}(\texttt{,})^{+}\ \texttt{)} \\
 & \mid & \textit{prefix-op}\ \textit{exp} \\
 & \mid & \textit{exp}\ \textit{infix-op}\ \textit{exp} \\
 & \mid & \texttt{\{}\ \{\textit{record-field-value}\}(\texttt{;})^{+}\ \texttt{\}} \\
 & \mid & \texttt{\{}\ \textit{exp}\ \texttt{with}\ \{\textit{record-field-value}\}(\texttt{;})^{+}\ \texttt{\}} \\
 & \mid & \textit{exp}\ \texttt{.}\ \textit{qualified-lident} \\
 & \mid & \texttt{(}\ \textit{exp}\ \texttt{)} \\
\textit{record-field-value} & ::= & \textit{qualified-lident}\ \texttt{=}\ \textit{exp} \\
\textit{let-binding} & ::= & \textit{lident}\ [\texttt{in}\ \textit{type-exp}]\ \texttt{=}\ \textit{exp} \\
 & \mid & \textit{lident}\ \texttt{(}\ \{\textit{lident}\ \texttt{in}\ \textit{type-exp}\}(\texttt{,})^{+}\ \texttt{)}\ \texttt{=}\ \textit{exp} \\
\textit{match-binding} & ::= & \texttt{|}\ \textit{pattern}\ \texttt{->}\ \textit{exp} \\
\textit{pattern} & ::= & \textit{integer-literal} \\
 & \mid & \textit{string-literal} \\
 & \mid & \textit{character-literal} \\
 & \mid & \textit{float-literal} \\
 & \mid & \texttt{true} \mid \texttt{false} \\
 & \mid & \textit{lident} \\
 & \mid & \textit{qualified-uident}\ [\texttt{(}\ \{\textit{pattern}\}(\texttt{,})^{+}\ \texttt{)}] \\
 & \mid & \texttt{\_} \\
 & \mid & \texttt{\{}\ \{\textit{record-field-pattern}\}(\texttt{;})^{+}\ \texttt{\}} \\
 & \mid & \texttt{(}\ \{\textit{pattern}\}(\texttt{,})^{+}\ \texttt{)} \\
 & \mid & \texttt{(}\ \textit{pattern}\ \texttt{)} \\
\textit{record-field-pattern} & ::= & \textit{qualified-lident}\ \texttt{=}\ \textit{lident}
\end{array}
$$

3.2.5.1 Literal expressions

The literal expressions of type integer, string, character, float and boolean are evaluated into the constant represented by the literal. The expression `25` denotes the value 25 of type `int`.

3.2.5.2 Sum type value constructor expressions

We presented in section 3.2.1.3 the way to define sum types. We saw that **values** of such a **type** are built using its **value** constructors.

Hence, for **value** constructors with no argument, the constructor itself is an expression that gets evaluated in a value wearing the same name.

For **value** constructors with parameters, a value is created by evaluating an expression applying the constructor to as many expressions as the constructor's arity. Obviously, sub-expressions used as arguments of the constructor must be well-typed (compatible) according to the type of the constructor. The resulting value is denoted by the name of the constructor followed by the tuple of values given as arguments. For instance, with the following type definition:

```
type t =
  | A
  | B (int * bool)
;;
```

the expression `A` is evaluated into A, the expression `B ((2 + 3), true)` is evaluated into the value $B(5, true)$.

3.2.5.3 Identifier expressions

An identifier expression is either a basic identifier, an extended identifier or a qualified identifier (see section 3.1.13), which denotes the value of this identifier in the scope of the expression. The identifier is said to be **bound** to this value.

The value bound to an identifier can be of any type. A value having a functional type, that is a **functional value** (also called a **closure**), is created by a function definition. Such a value, obtained by the evaluation of the body of the function, is slightly different from other ones since it embeds both the code of the function (i.e. a kind of evaluation of its body expression) and its environment (i.e. bindings between identifiers occuring in the body of the function and their value in the definition scope). This closure will be kept untouched until it appears in a functional application expression as described further in 3.2.5.9.

There are several possibilities to bind an identifier. Definitions introduce a basic or extended identifier and **binds** it to the value of the expression stated in the definition. There are three ways to introduce and directly bind an identifier:

- by a `let-in` construct,
- by a toplevel-`let`-definition,
- by a method definition (`let`).

Each of these cases will be described in their related section.

There are two ways to introduce basic identifiers as parameters:

- in a function definition
- by a pattern inside a `match-with` construct

Then the binding of the parameter is deferred until the application of the function or the pattern-matching mechanism. Each of these cases will be described in their related section.

Suppose that an expression exp contains several occurrences of an identifier `my_var`. Assume that, in the scope of exp, `my_var` is bound to a **value** v, then each occurrence of `my_var` in exp is substituted by v during the evaluation of exp. This is basically the principle of the so-called **eager** or **call by-value** evaluation regime.

Identifier resolution Remember that identifiers forms differ depending on the syntactic class of entity they refer to, capitalized identifiers being used for species and collections. To evaluate an identifier expression, the FoCaLiZe compiler tries to find its definition from the current scoping context. It searches for the closest (latest) definition with this name, starting by the parameters present in the current definition (i.e. formal parameters in a function and in a `match-with` construction and `let-in` bound identifiers). If no identifier definition with this name is found, the search goes on among the methods of the current species. If a method is found with this name, it will be retained, otherwise the identifier is looked in the preceding toplevel-definitions of the current compilation unit. If no suitable definition is found, then the ones imported by the `open` directives are examined to find one with the searched name. Finally if no definition is found, the identifier is **reported unbound** by an error message.

Note that an `open` directive may arise anywhere at toplevel in the source code. Hence, the order of search between the current file's toplevel-definitions and the imported ones by `open` is not really separated: the name resolver looks for the most recent definition considering that the toplevel-definitions and the imported ones are ordered according to the apparition of the effective definitions in the file themselves and the imported ones. In other words, if a toplevel-definition exists for an entity `foo`, if later an `open`directive imports another `foo`, then this last one will be the retained one.

Identifier qualification

Identifiers can manually be disambiguated in term of compilation unit location using the sharp (#) notation as explained in section 3.1.13.

As further presented in section 4.2.1, species methods identifiers are made explicit using the "!" notation. The notation `Spe!meth` stands for "the method `meth` of the species `Spe`". By extension, `!meth` stands for the method `meth` of the current species. It is possible to explicit `Self` in the naming scheme using `Self!meth`. This is useful when a more recently defined identifier hides a method of the species at hand:

```
species S =
  let m (x in ...) = ... ;
  let n (y in ...) =
    ...
    let m = ... in
    (* Want to call the *method* "m" with argument the local "m" !!! *)
    !m (m) ;
end ;;
```

Hence, the name resolution mechanism allows to omit the "!" but making it explicit can help for conflicts resolution. Moreover, when invoking species parameters' methods, the name resolution never searches among methods of collection parameters, hence the explicit "!" notation is required.

As the grammar shows, name qualification by compilation unit and hosting species can be freely mixed. We can build identifiers like `my_file#My_species!my_method` to refer to the method `my_method` hosted in the species `My_species` located in the FoCaLiZe source file "my_file.fcl". These disambiguation methods are indeed orthogonal.

Extended identifier expressions

Finally, infix/postfix operators can be used as regular identifiers. Usually, an operator is syntactically used according to it prefix or infix nature. For instance, the binary + operator is used between its arguments as in `x + 4`, the unary operator `~` is used before its argument as in `~x`. FoCaLiZe allows to refer to those operators as regular identifiers (for instance as function parameters). This allows to use operators as any other identifiers, and

- using them as regular function (i.e. in functional position),
- bind them as arguments of functions,
- use them as regular identifiers in expressions, for example to pass them as arguments of other functions.

The following example illustrate the second point:

```
let twice(( + ), x, y, z) = (x + y) + z ;;
```

To get an identifier from an operator, its symbol (cf. 3.1.6) must be delimited by spaces and enclosed into matching parentheses. For example: `( + )` is the regular identifier corresponding to the infix symbol `+`.

Note that spaces around the operator symbol are mandatory and part of the syntax. If spaces are omitted, the parens get their usual meaning and the interpretation can be completely different. A specially puzzling error is to write `(*)` to mean `( * )`:

```
...
let (*) (x, y) = ...
```

Now, `(*` is evidently parsed as the beginning of comment, leading to a syntax error or any other cryptic error long after the faulty `(*` occurrence. Conversely `*)` is always considered as an end of comment by the lexical analyzer.

3.2.5.4 `let-in` expression

`let-in` expression binds an identifier to a value to evaluate a trailing expression (the "`in-part`" of the "`let-in`" or "body") where this ident may appear. During the evaluation of the trailing expression, any occurrence of the bound identifier is "replaced" by the value bound to this identifier. For instance:

```
let x = 3 + 2 in (x, x)
```

binds `x` to the evaluation of the **expression** `(3+2)` (i.e. the integer **value** 5) and then, the evaluation of the trailing expression returns the tuple **value** $(5, 5)$. From the syntax, it is clear that `let-in` constructs can be nested. For instance,

```
let x = 3+2 in
  let y = (x, x) in
  let z = true in
  (y, z, y, z)
```

returns the value $((5,\ 5),\ true,\ (5,\ 5),\ true)$ of type `((int * int) * bool * (int * int) * bool)`.

Note that the notion of "binding an identifier to a value" is essentially different from the notion of assignment in imperative languages. In such languages (like C, Java, Pascal,...) a variable is first *declared*, then a value is *assigned* to the variable. It is thus possible to assign a variable several times to different values. For example in C:

```
...
{
  int i ;
  ... ;
  i = 10 ;
  while (i > 0) i = i-- ;
}
...
```

The variable `i` is declared, then assigned the initial value 10, then the `while` loop makes it decreasing by successive assignments.

In a `let-in` binding construct, an identifier is given a value once and for all: it is impossible to change its value, once it has been bound. Each new definition, binding an already bound identifier will just hide the old definition. For instance:

```
let x = 5 in
  let y = (x, x) in
  let x = true in
  let z = (x, x) in
  (y, x, y, x)
```

leads to the value $((5,\ 5),\ true, (5,\ 5),\ true)$ of type `((int * int) * bool * (int * int) * bool)`. Clearly the first value bound to `x` holds until `x` is bound again: 5 is used to define `y` but not to define `z`, since the value of `x` is then the boolean $true$.

The `let-in` construct serves to bind an identifier to a value of any type. As a consequence, it can also bind an identifier to a functional value. This lead to the natural way to define **functions**. For instance:

```
let f (x, y) = x + y in
f (6, 7)
```

The **let** construct binds `f` to a function which has 2 parameters `x` and `y`, and the body of `f` is the addition of these 2 parameters. Then the body of the `let-in` construct applies `f` to 2 effective arguments `6` and `7` (we obviously expect the result of this *application* to be 13). (Function application is explained below in 3.2.5.9).

It is possible to provide a type constraint to precise the type of the return value of a function, or the type of the `let`-bound variable or parameters:

```
let f (x : int, y) = x + y in
f (6, 7)
```

```
let f (x : int, y) in int = x + y in
f (6, 7)
```

```
let a in int = 3 in
(a, a)
```

It is possible to define several identifiers at the same time separating each definition by the keyword `and`.

```
...
let f = exp_1
and g  = exp_2
and h = exp_3 in exp;
```

All the definitions are separately evaluated "in parallel". As a consequence, the identifiers introduced by a `let ... and` cannot be used in the right members of this construction (in the `exp_i`). Do not confuse this construct with nested `let-in` as the followig one, where `exp_2` can contain `f` and `exp_3` can contain `f` and `g`.

```
let f = exp_1 in
 g  = exp_2 in
 h = exp_3 in exp
```

Mutually recursive functions need to know each other because their bodies call these other functions and their definition require a non-nested evaluation of each function. In this case, the keyword `let` must be followed by the keyword `rec`.

```
...
let rec even (x) =
   if x = 0 then true else odd (x - 1)
and odd (y) =
   if y = 0 then false else even (y - 1) in
...
```

Warning: in the current version of FoCaLiZe mutually recursive functions cannot be compiled into Coq code. Only OCaml code generation is available. Moreover, for Coq, recursive functions imply termination proofs. This last point will be covered in the section 6 especially dedicated to (non-mutually) recursive function definitions.

3.2.5.5 `logical let`

As seen above, the `let-in` construct is used to bind computational expressions. We would sometimes also like to have parameterised logical expressions, i.e. a kind of functions returning a logical proposition. For example we may want, for a certain value of x and y, to use the statement "$x < y$ and $x + y < 10$" (which holds or not) to build more complex logical expressions.

To allow functional bindings in logical expressions FoCaLiZe provide the `logical let` construct. It serves to introduce a parameterised logical expression, which can be applied to effective arguments to obtain a logical proposition. Our example would be expressed by:

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

species S =
  ...
  logical let f (x in int, y in int) = x < y /\ x + y < 10 ;
  ...
end ;
```

Since `logical let` binds an identifier to a logical expression, the body of the definition **must** obviously **be of type** `bool`. Once defined, `f` can be used as a regular function, but only in properties and theorems statements. For instance:

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

species S =
  ...
  let m (x in Self) in int = ... ;
  logical let f (x in int, y in int) = x < y /\ x + y < 10 ;
  ...
```

```
  property p : all a in Self, all b, c in int, f (c, b) => f (m (a), b) ;
end ;
```

See other examples in the standard library where this construction is used to define associativity, commutativity, ...

3.2.5.6 `final (logical) let`

Cannot be redefined by inheritance.

3.2.5.7 Conditional expression

A conditional expression has the form: `if` exp_1 `then` exp_2 `else` exp_3

Its evaluation starts by the evaluation of the exp_1 expression which must be of type boolean. If its value is $true$ then the result value of the whole expression is the value of exp_2, otherwise (i.e. if its value is $false$) the value of exp_3. This obviously implies that exp_2 and exp_3 must have the same type. This construct is then a binary conditional expression (i.e. with 2 branches).

```
let f (x) = if x then 1 else 0 in ...
```

The function `f` will return 1 if the effective argument provided for `x` is $true$, otherwise it will return 0.

```
let is_too_small (x) = ... in
let y = ... in
let y_corrected = if is_to_small (y) then 0 else y in ...
```

In this example, we assume we have a function `is_too_small` checking if a value is "too small" and an identifier `y` bound to a certain value. The result of the conditional expression bound to `y_corrected` will be either 0 if the condition is met or y otherwise.

3.2.5.8 Match expression

The `match-with` construct is a generalised conditional construct with pattern-matching. By "generalised", we mean that unlike the `if-then-else` which has only 2 branches, the present expression can have several branches. The notion of condition here is not anymore a boolean value. Instead, the construct allows to discriminate on the different values an expression is evaluated into. The basic structure of a `match-with` consists in a discriminating expression followed by an enumeration of cases (called **patterns**). The dicriminating expression is evaluated and its value is matched against the patterns, following the textual ordering of these patterns, until a match succeeds. Then the expression associated with the matching pattern is evaluated to obtain the value of the whole expression `match-with`.

```
let a = ... ;
let x =
  match a + 5 with
   | 0 -> "zero"
   | 5 -> "five"
   | 1 -> "one"
   | 10 -> "ten"
   | _ -> "other" in
...
```

The discriminated expression in this case is `a + 5` of type `int`. We can then react to each (or some of the) values of this expression. When `a + 5` is equal to 0 the result of the whole `match-with` expression (bound to the identifier `x`) is the string “zero”. When `a + 5` is equal to 1, the result is the string “one”, and so on. The final pattern _ stands for “anything that was not in the previous cases” (also called “catch-all pattern”). Hence, the order of the patterns is important. If the case `|_ ->` was put before the case `|1 ->`, then this last case would never be reached since the _ pattern would have caught the discriminated value.

As a consequence of the structure of this construct, type constraints must be respected in order to have the whole expression well-typed:

- The type of the discriminating expression must be compatible with the type of the patterns.
- thus all the patterns must have compatible types.
- The types of all the result expressions in the rightmost parts of the cases must be compatible.
- The patterns have to be mutually exclusive (except for the catch-all pattern, see below).
- The different cases have to capture every possible pattern.

In the example above, the patterns were constant. A value matches a constant pattern if and only if it is equal to this constant. In addition, the `match-with` construct provides true **pattern matching**. That is, patterns may be built from constants, value constructors, variables and the catch-all symbol _. Any value matches any variable pattern and the _ pattern. For general patterns built from value constructors, variables, constants, _, roughly speaking, a value matches a pattern if this pattern can be seen as a prefix of this value. Then, the variables of the pattern get bound to the parts of the discriminating expression that are “at the same place” than those variables. For example:

```
let e = ... in
...
let x =
  match e with
   | (0, 0, 0) -> 1
   | (0, x, y) -> x + y
   | (1, 1, x) -> x
   | (x, y, z) -> x + y + z
...
```

According the the type-checking mechanism, the examined expression `e` must have here type `(int * int * int)`. The first pattern will be chosen if `e` is equal to the tuple $(0, 0, 0)$. We say here “equal” since there is no variable in the pattern, hence the only way to fit the pattern is to simply be equal. If this pattern is not fitted, the we examine the second pattern. It will be chosen if `e` has a 0 as first component and any integer for the second and the third ones. In this case, the result value will be the evaluation of the expression `x + y` where x will be bound to the effective second component of the value of `e` and `y` will be bound to its third component. We can notice that no “catch-all pattern” is needed since the enumerated patterns cover all the possible values of tuples with 3 components (the last pattern does not put any constraint on the tuple components, hence will match all the remaining cases).

The previous example used tuples as matched expression and patterns, but patterns also contain sum type value constructors, hence allowing to “match” on any sum type structure. For example:

```
type t =
  | A
  | B (int)
  | C (int, int)
;;
...
let e = A ;;
let x =
  match e with
```

```
    | A -> 0
    | B (3) -> 4
    | B (_) -> 10
    | C (x, 10) -> 5
    | C (_, y) -> y
;;
```

This example shows different cases following the structure of the type `t`. Note the use of the "catch-all" pattern inside patterns. In fact, the "catch-all" pattern acts like a variable unused in the rightmost part of the case. It is however preferable to use "_" instead of a variable since OCaml generates warning for unused variables and the generated OCaml code generated by FoCaLiZe will not change unused variables into "_"s.

Patterns also allow to match record values (cf. 3.2.5.11), i.e. to match on values of the fields:

```
type t = { name : string ; birth : int } ;;
let r = ... in
let x =
  match r with
    | { name = "Alexandre" } -> ...
    | { name = n ; birth = 2003 } -> ...
    | { name = n } -> ...
```

In such a pattern, fields not specified are considered as "catch-all" patterns. Hence, the last case catches all the record values not caught before since the field `name`'s value is bound to a variable (so, any value can match it) and the field `birth` is absent (so, considered as `birth = _`).

3.2.5.9 Application expression

We previously saw that the `let-in` construct allow for the definition of functions by binding an identifier to a functional value. Using a function by providing it with effective arguments to get its result value is called **application**. Hence, in an application there are 2 distinct parts: the applicative part that must be an expression leading to a functional value and the effective arguments that are expressions whose values will be provided to the function to make its computation. The syntax for application is simply the juxtaposition of the applicative expression and the comma-separated expressions used as arguments embraced by parentheses:

```
let f (x) = ... in
let g (x, y) = ... f (y) ... in
g (f (3), 4)
...
```

As described in 3.2.5.3, the evaluation of an application of a function to its effective arguments start by the evaluation of these arguments (the order of the evaluation of several arguments is left unspecified). Then these effective values are substituted to the corresponding parameters inside the body of the function and the so-obtained expression (the substituted body) is evaluated. For instance, having the following function and application:

```
let g (x, y) = (y, x) in
g (true, 1)
```

The evaluation of the `let-in` expression first binds the identifier `g` to a **functional value** also called **closure**. Then the application expression `g (true, 1)` is evaluated. So the values of `g` and of the expression `(true,1)` are elaborated: the evaluation of `g` returns a closure, `true` is evaluated into the boolean **value** $true$, 1 into the integer **value** 1. The next step is to evaluate the body of the **closure** of g, replacing the formal parameter x by the effective argument $true$ and y by 1. The body of g creates a tuple from its 2 arguments, putting y in the first component and x in the second. Hence, the result of the application is the tuple **value** $(1,\ true)$.

3.2.5.10 Operator application expression

Since operators are designed to be used in infix or prefix position, application of operators consists simply in providing arguments according to the operator infix/prefix nature. For infix operators, arguments are on left and right sides. For prefix operators, the operator is in front of the argument expression.

3.2.5.11 Record expression

As stated in 3.2.1.3, record types are defined by a list of labels with their types. As usual a record expression follows the same structure, replacing the type expressions of the definition by values of these types. For instance, assuming the given record type definition, the following example shows a possible record value:

```
type identity = {
  name : string ;
  birth : int ;
  living : bool
} ;;

...
{ name = "Nobody" ; birth = 42 ; living = false }
...
```

If the record type definition is in a different compilation unit, you may qualify the record fields by the "#" notation:

```
{ my_file#name = "Nobody" ; my_file#birth = 42 ; my_file#living = false }
```

3.2.5.12 Cloning a record expression

It is sometimes needed to create a new value of a record type by modifying a few fields of an existing record, leaving the other fields unchanged. If the record type definition contains numerous fields, manually copying the old fields values to create the new record value appears boring and error prone:

```
type t = { a : int ; b : int ; c : int ; d : int ; e : int ; f : int } ;;
...
let v1 = { a = 1 ; b = 2 ; c = 3 d = 4 ; e = 5 ; f = 6 } in
...
let v2 = {
  a = v1.a ; b = v1.b ;
  c = 5 ;  (* Changed value. *)
  d = v1.c ;  (* an error since the requested value was  "v1.d". *)
  e = 6 ;  (* Changed value. *)
  f = v1.f } in
...
```

Instead of manually copy the unchanged fields, FoCaLiZe provides a way to clone a record value, that is to create a **new**, a **fresh** value from an existing one, only by specifying the fields whose values differ from the old record value:

```
type t = ... (* Like above. *)
let v1 = ... (* Like above. *)
...
let v2 = { v1 with c = 5 ; e = 6 } in
...
```

As for other record value expressions, if the record type definition is in a different compilation unit, you may qualify the record fields by the "#" notation.

3.2.5.13 Record field access expression

Once a record value is created by aggregating values of its fields, it is possible to recover the value of one field by a dot notation. For instance, assuming the type definition and record values of the previous example:

```
... v1.a ...
... v2.c ...
```

respectively get the value of the fields `a` of `v1` and `c` of `v2`, that is, 1 and 5. If the record type definition is in a different compilation unit, you may qualify the record fields by the "#" notation: `t1.my_source#a`.

3.2.5.14 Parenthesised expression

The parentheses can be used around any expression, to enforce the associativity or evaluation order of expressions. Simple expressions (i.e. atomic) can also be parenthesised without changing their values.

3.2.6 Core language expressions and definitions

In the previous sections, we described the syntax of expressions. Expressions rarely appear outside any definition but it is still possible to have top-level expressions. They will be directly evaluated and not bound to any identifier, but this implies that these expressions use previously written definitions.

As further explained in (cf. 4.1.2) species are made of methods. Some methods contain expressions (functions, properties, theorems). Function-methods are introduced by the `let` keyword, using the same syntax (hence expressions) that the `let-in` construct except the fact they do not have a "`in`" expression. The idea is that the "`in`" expression is implicitly the remaining of the species. Properties and theorems are respectively introduced by the keywords `property` and `theorem` and may contain expressions. The section 3.2.8 is dedicated to their detailed explanation.

```
open "basics" ;;
species My_Setoid inherits Basic_object =
  signature ( = ) : Self -> Self -> bool ;
  signature element : Self ;
  let different (x, y) = basics#not_b (x = y) ;

  property refl : all x in Self, x = x ;
  property symm : all x y in Self, Self!( = ) (x, y) -> y = x ;
end ;;
```

Toplevel-definitions are definitions introduced outside of any species. General functions and general theorems, i.e. that do not depend on a particular species can be introduced as toplevel-definitions. Toplevel-functions are introduced by the `let` keyword and don't have a "`in`" expression, this part being implicitly the remaining of the program (i.e. the current compilation unit and those using the current). Toplevel-theorems are introduced by the `theorem` keyword. These definitions must be ended by a double semi ("`;;`").

```
let is_failed (x) =
  match x with
  | Failed -> true
  | Unfailed (_) -> false
;;

theorem int_plus_minus: all x y z in int,
   (*   x + y = z -> y = z - x *)
  #base_eq (#int_plus (x, y), z) -> #base_eq (y, #int_minus (z, x))
  proof =
    coq proof {*
      intros x y z;
      unfold int_plus, int_minus, base_eq, syntactic_equal in |- *;
      intros H;
      unfold bi__int_minus;
      apply EQ_base_eq; apply Zplus_minus_eq;
      symmetry  in |- *;
      apply (decidable _ _ _ (Z_eq_dec (x + y) z) H).
      Qed.
    *}
;;
```

3.2.7 Files and uses directives

FoCaLiZe provides 3 directives that are not expressions. This means that they do not lead to values or computation.

All these directives deal with searching for files in the available search paths. The path of the compilation unit is never specified since it is always searched first. Hence, the file will be searched first in the local directory, then in the stardard library directory and finally in the library search path specified with the `-I` option (cf. 7).

Note that if several files at different locations have the same name, the directives will use the first fiound in the search path set in the compilation command-line.

3.2.7.1 The `use` directive

This directive is followed by the name of the file to open between double quotes without the ".fo" extension (compiled version of a ".fcl"). Before being allowed to use the qualified notation for an identifier, (i.e. the "#"-notation), the qualifying compilation must be declared with a directive `use`. In other terms, "using" a compilation units allows to access its entities from the current compilation unit.

3.2.7.2 The `open` directive

This directive is followed by the name of the file to open between double quotes without the ".fcl" extension. As previously introduced (cf. 3.2.5.3 and 3.1.13) the `open` directive loads in the current name resolution (scoping) environment the definitions of the compilation unit named in the `open` directive. This prevents the user from having to explicitly qualify definitions of this unit by the "#" notation. Definitions imported by the directive hide ("mask") those wearing the same name already defined in the current compilation unit from the point the directive appears. Remember that it is however possible to recover the hidden definitions, using the "#" notation without compilation unit name.

Note that the `open` directive implicitly implies the `use` directive. This means that it is not useful to add a `use` together with an `open` directive.

```
open "sets";;
```

This directive loads the definitions of the compilation unit “sets.fo” in the current name resolution (scoping) environment.

3.2.7.3 The `coq_require` directive

Some source files of a development may be directly written in Coq to provide external definitions (detailed further in 9.0.1) to import and use in the FoCaLiZe source code. In this case, the Coq code generated for the FoCaLiZe source code must be aware of the need to import the external definitions from the manually written Coq file. For this reason, the FoCaLiZe source must explicitly indicate by the `coq_require` directive that it makes references to definitions hosted in this Coq source file. For example, the file “wellfounded.fcl” of the standard library needs “wellfounded_externals.vo” (the compiled version of “wellfounded_externals.v”) and signals this fact in its early lines of code:

```
...
open "basics";;
open "sets_orders";;
coq_require "wellfounded_externals";;
...
```

3.2.8 Properties, theorems and proofs

Properties are first order logic propositions and theorems are properties with their proofs. We will study here first the structure of logical expressions used to express the statements, show properties and theorems forms and shortly present the 3 available ways to write proofs.

3.2.8.1 Logical expressions

Logical expressions are those used to write first order logic formulas.

logical-exp	::=	`all` {*lident*}$^+$ `in` *type-exp* `,` *logical-exp*
	|	`ex` {*lident*}$^+$ `in` *type-exp* `,` *logical-exp*
	|	*logical-exp* `->` *logical-exp*
	|	*logical-exp* `<->` *logical-exp*
	|	*logical-exp* `/\` *logical-exp*
	|	*logical-exp* `\/` *logical-exp*
	|	`~` *logical-exp*
	|	*exp*
	|	`(` *logical-exp* `)`

Logical expressions contain the usual logical connectors “imply” ($\Rightarrow$), “and” ($\wedge$), “or” ($\vee$), “there exists” ($\exists$), “for all” ($\forall$), “is equivalent” ($\Leftrightarrow$) and “not” ($\neg$). Moreover, logical expressions embed the FoCaLiZe expressions used in computational methods (i.e. identifiers, conditionals, application, . . .). This allows to have connected propositions using the previously defined functions and species methods.

```
species S ... =
  signature gt : Self -> Self -> bool ;   (* Greater than... *)
  signature geq : Self -> Self -> bool ;   (* Greater or equal... *)
```

```
  signature equal : Self -> Self -> bool ;  (* Equal to... *)
  signature different : Self -> Self -> bool ;  (* Different of... *)

  property gt_is_lt : all x y in Self,
    (!gt (x, y) -> (!geq (x, y) /\ !different(x, y)))
    /\
    (!geq (x, y) -> (!gt (x, y) \/ !equal(x, y))) ;
end ;;
```

Since propositions in logical expressions are truth values, this obviously imply that the arbitrary expressions used between connectors must have type `bool`.

3.2.8.2 Properties

A property is a logical expression bound to an identifier. Its form is the name of the property, a colon character (":") and the logical expression being its statement. See the example given in 3.2.8.1.

property ::= `property` *lident* `:` *logical-exp*

3.2.8.3 Proofs

FoCaLiZe currently provides 3 ways to write proofs. We only give here a simple description of these 3 means without going deeply in the technical mechanisms since this problem will be especially addressed in section 5 and 9.0.2.

- **Consider the proof as "assumed"**. This way is the simplest but also the weakest one since it consists in saying that no proof is given and the system must accept the statement as an axiom.

  ```
  species S =
    representation = int;
    let equal = (=0x);
    theorem symmetry : all x y in Self, equal (x, y) -> equal (y, x)
    proof = assumed
      (* Machine integers equality admitted to be symmetric  *) ;
    ...
  end ;;
  ```

  Although such a proof can introduce inconsistencies if the "theorem" is not a tautology and thus decrease confidence in the correctness of the FoCaLiZe program, there are several cases where using this keyword may help.

  - The first case is simply that the developer doesn't know (yet ?) how to make the proof, doesn't have time yet to write it, or is not interested in proofs but still wants his program to compile to get the executable code.
  - Second case deals with import of external code, i.e. code not written in FoCaLiZe and considered as external. In this case, since the imported code does not fit the FoCaLiZe model and more accurately, does not have formal properties, it is impossible to make any proof on FoCaLiZe's side based on the structure of this code and its non-existing implementation properties. In other terms, things coming outside FoCaLiZe universe can not be modeled by FoCaLiZe. The developer can only import them providing a binding is given and must trust them.

– Last case addresses "well-known" mathematical properties that do not actually hold in computers since they are finite machines, working on bounded arithmetics. The most obvious example is the fact that since an integer is coded on a machine word (e.g. 32-bit values), the mathematical property $\forall x \in \mathrm{N}, x + 1 > x$ does not hold anymore.

  However, conceptually, except when dealing with boundaries, this property holds and we need to achieve further proofs. For this reason, assuming that the proof holds is legitimate, if the developer is able to guarantee that the integer computations never overflow. If he cannot guarantee non-overflow, then this is a true problem of specification or design which should be reconsidered.

In any case, we advice the reader to use the test tool (or another mean) to comfort the confidence in the statement of the theorem, when such statement is admitted.

- **Write an automated proof script**. FoCaLiZe provides a syntax, the FoCaLiZe Proof Language, to split proofs into steps that may be proved by the theorem prover. Without entering deeply into the syntax further described in chapter 5, the main features are the following. The user may state hypotheses, demonstrate subgoals that will serve as lemmas for a higher level goal and may give hints about definitions or declarations of methods. Then the prover tries to automatically guess a proof of this goal, then tries to prove those lemmas, hence building a proof tree until the top goal (i.e. the theorem) is proved. Here is an example of such a proof.

```
theorem zero_is_unique : all o in Self,
 (all x in Self, !equal (x, !plus (x, o))) -> !equal (o, !zero)
 proof =
       <1>1 assume o in Self,
            assume H1: all x in Self, !equal (x, !plus (x, o)),
            prove !equal (o, !zero)
           <2>1 prove !equal (!zero, !plus (!zero, o))
              by hypothesis H1
           <2>3 prove !equal (o, !zero)
              by step <2>1
              property zero_is_neutral, equal_transitive, equal_symmetric
           <2>4 conclude
      <1>2 conclude
 ;
```

- **Write a Coq or Dedukti script** This way is the most difficult since it means to directly write Coq or Dedukti code. It requires the understanding of both the proof checker and the mapping the FoCaLiZe compiler does to generate Coq/Dedukti code from FoCaLiZe source code. The section 9.0.2 describes how FoCaLiZe definitions are mapped onto Coq and Dedukti names.
  The script is introduced by the keywords `coq proof` or `dedukti proof` and surrounded by $\{*$ and $*\}$. Below follows an example of such a proof.

```
theorem int_minus_plus: all x y z in int,
  (* x - y = z -> x = y + z *)
  #base_eq (#int_minus (x, y), z) -> #base_eq (x, #int_plus (y, z))
  proof =
    coq proof {*
      intros x y z; unfold int_plus, int_minus, base_eq,
        syntactic_equal in |- *;
      intros H;
      unfold bi__int_minus;
      apply EQ_base_eq; rewrite <- (Zplus_minus y x);
      apply Zplus_eq_compat; trivial; apply decidable.
      apply Z_eq_dec. assumption.
      Qed.
```

```
    *} ;;
```

3.2.8.4 Theorems

Now we know how to write a logical statement and how to write a proof, the structure of a theorem appears simple since it contains both the statement and the proof inside the same construct. The theorem is introduced by the keyword `theorem` and the proof by the keyword `proof` followed by an equal character ("=").

theorem ::= `theorem` *lident* `:` *logical-exp* `proof` `=` *proof*

For instance:

```
species Meet_semi_lattice inherits Setoid =
  ...
  theorem inf_right_substitution_rule : all x y z in Self,
     equal(y, z) -> equal(!inf(x, y), !inf(x, z))
     proof =
       by property
          inf_left_substitution_rule,
          inf_commutes,
          equal_transitive ;
  ...
end ;;
```

The kind of proof used here is written in FoCaLiZe Proof Language and must not be a matter of understanding at this point since this particular point will be addressed with more details in chapter 5.

Notice that theorems can be hosted in a species or can be toplevel-theorems. Unlike theorems, properties cannot appear at toplevel since there is no way to inherit at toplevel, hence no way to give a proof after the property declaration in a "parent".

Chapter 4

The FoCaLiZe model

As stated in section 1, the FoCaLiZe language is designed to build an application step by step, going from very abstract specifications to the concrete implementation through a hierarchy of structures. At first sight species seem quite similar to classes in an Object-Oriented context. *However, despite of inheritance and late-binding features,* FoCaLiZe *is definitively not an Object-Oriented language as C++, Java, etc. are.*

In the following we focus on the basic concepts underlying a FoCaLiZe development, that is:

- Top-level definitions
- Species
- Collections
- Parametrisation
- Inheritance
- Late-binding

To ensure that this part can be read independently of the section 1, we duplicate some explanations.

4.1 Basic concepts

4.1.1 Top-level Definitions

We call **toplevel-definition** (just one word) a definition which appears outside species and collections. Such definitions can only be:

- Species
- collections,
- type definitions,
- general theorems (not depending on a species)
- general functions (not depending on a species),
- expressions to be directly evaluated (but there is no way to bind their value to an identifier).

Any toplevel-definition is terminated by a double semi-character (";;").

4.1.2 Species

Species are the nodes of the FoCaLiZe hierarchy. A species is a sequence of **methods** or **fields**, each one being terminated by a semi character (";"). Hence, a basic species looks like:

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

Species names are always **capitalised**. As any toplevel-definition, a species ends with a double semi-character (";;"). There are several kinds of methods:

- The **representation**. It defines the type of the entities manipulated in the species and is a kind of alias type (see section 3.2.4). The representation can be a type variable and then is said to be "not yet defined" or "only declared" and is not explicitly introduced . It can be bound to a type defined by a more complex type expression possibly containing type variables (introduced via collection parameters). Either, this type value is obtained by inheritance or is introduced by the keyword `representation` followed by = followed by a type expression. Ultimately to get a *complete* (fully defined) species, the representation must be a fully instantiated type (directly or by 4.3.1).
  In the context of a species, the representation is denoted by `Self`.
  Note that a representation is never a polymorphic type. When it is only declared, it is a type variable, which can receive only one instantiation. In other words, this type variable is not universally quantified, as are the type variables of polymorphic types.
- **Signatures**. They introduce names of constants and functions, uniquely providing their type as a type expression. A signature begins with the keyword `signature` followed by the introduced name followed by **:** followed by a type expression. For instance:

  ```
  species IntStack =
    signature push : int -> Self -> Self ;
  end ;;
  ```

  As we saw above, `Self` represents the representation (thus a type) of the current species. Hence an operation pushing an integer onto a stack takes as parameter the integer to push, the stack on which to push and give back a new stack, that is, an entity of type `Self`.

- **Functions**. They are implementations of signatures, providing effective code. A function is introduced by the `let` keyword followed by the name followed by = followed by a definition, which is similar to ML definitions. Recursive functions are introduced by `let rec` to make explicit the recursivity.

  ```
  species IntStack =
    representation = int list ;
    let push (v in int, s in Self) = v :: s ;
  end ;;
  ```

  Function parameters can be entities (that is, values) of the species itself (which type is the representation, thus denoted by `Self`), entities of known collections, values of known types.
  Functions can use in their body other methods of the species, toplevel-definitions of functions, methods of collections (described further in 4.1.5), or methods of collections parameters (see 4.2.1).
  When we say "other methods of the species", this includes functions only introduced by their signatures. This means that it is possible to use something only declared, without yet effective implementation. We will address this point later in detail in section 4.4.1.

Although FoCaLiZe is a functional language, function application must always be total. This means that any function call must be provided all the effective arguments of the function. As previously described in the core syntax (cf. 3.2.5.9), function application is “à la C”, that is with arguments comma separated and enclosed by parentheses.

- **Properties**. They are first order formulae containing names already introduced. When stating a property, the proof that it holds is not yet provided (but will have to be ultimately provided). A property can be viewed as a declaration.

```
species IntStack =
  ...
  property push_returns_non_empty :
    all v in int, all s in Self, push (v, s) -> ~ is_empty (s) ;
end ;;
```

Proofs of properties can be **delayed**, that is, done afterwards using a `proof` field in a species. The way to give proofs will be seen further.

```
species IntStack2 =
  inherit IntStack;
  proof of push_returns_non_empty = ... ;
end ;;
```

- **Theorems**. They are properties with their proofs. In fact, when defining a property, we only give the statement of a theorem, leaving its proof for later. A theorem can be viewed as a definition.

```
species IntStack =
  ...
  theorem push_returns_non_empty :
    all v in int, all s in Self, push (v, s) -> ~ is_empty
    (s)
  proof = ... ;
end ;;
```

One important restriction on the type of the methods is that it cannot be polymorphic. However, FoCaLiZe provides another mechanism to circumvent this restriction, the parametrisation as explained further (cf. 4.2).

4.1.3 Complete species

A species is said *complete* if all its methods are *defined*, i.e. have an implementation. In other words this means that there is no more methods only *declared*. This notion implies that:

- The representation has been associated with a type definition.
- Every declaration is associated to a definition.
- A proof is given for every property.

Obviously, it is possible to build a species without signatures and properties, only providing functions and theorems directly. In this case, if the representation is also defined, then the obtained species is trivially complete.

The important point for a species to be complete is that it can be turned into effective executable OCaml code and effective checkable Coq and Dedukti code, since all the components are known.

Important: Although we said that only a complete species can lead to effective executable code, of course species even not complete are compiled ! This means that you do not need to have a complete species to compile your source code ! It is very common to have species not complete in source files since programs are written in a modular fashion, in several files. Moreover, a library may provide species with methods not defined, leaving the user the freedom to chose an effective implementation for some algorithms.

4.1.4 Interfaces

The **interface** of a species is the list of the declarations of its methods. It corresponds to the end-user point of view, who wants to know which functions he can use, and which properties these functions have, but doesn't care about the details of the implementation.

The interface of a species is obtained by keeping the signatures and properties and retaining only the signatures of the let methods and the statement of the theorems. The representation is hidden thus abstract (only unifiable with itself). Hence, getting the interface of a species can roughly be seen as erasing the representation, turning the functions into signatures and the theorems into properties.

While this abstraction is easy within programming languages, it is not always possible when dealing with proofs and properties. Such problematic species are rejected by FoCaLiZe and will be described later in 4.4.2.

An interface has a **name**, *which is the name of the underlying species*. There should be no confusion between species names and interface names as interface names are only used to declare formal collection parameters (see section 4.2.1) and to apply methods of collection parameters.

4.1.5 Collections

A **collection** is a kind of "grey box", built from a *complete* species by abstraction of the representation. A collection has exactly the same sequence of methods than the complete species underlying it, apart the representation which is hidden. Note that creating a collection from it is the only way to turn methods of a complete species into executable code. This point is emphasised by the syntax:

`collection` *name-collection* = `implement` *name-species* `; end`

The interface of a collection is the one of the complete species it implements. The interface I_1 of a collection C_1 is *compatible* with an interface I_2 if I_1 contains all the components of I_2.

Thus, implementing a complete species creates a collection, which is a kind of abstract data-type. This especially means that entities of the collection cannot be directly created or manipulated as their type is not accessible. So they can only be manipulated by the methods of the *implemented* species.

```
species Full =
  rep = int ;
  let create_random : Self = random_foc#random_int (42) ;
  let double (x : Self) : Self = x + x ;
  let print (x : Self) = print_int (x) ;
end ;;

collection MyFull_Instance =
  implement Full;
end ;;

let v = MyFull_Instance!create_random ;;
MyFull_Instance!print (v) ;;
let dv = MyFull_Instance!double (v) ;;
MyFull_Instance!print (dv) ;;
```

In this example, we define a complete species `Full`. Then we create the collection `MyFull_Instance`. And we use methods of this collection to create entities of this collection. We print the result of the evaluation of the top-level definitions of `v` and `dv`.

Note that two collections created from a same species are not type-compatible since their representation is abstracted making impossible to ensure a type equivalence.

As a conclusion, collections are the only way to get something that can be executed since they are the terminal items of a FoCaLiZe development hierarchy. Since they are "terminal", this also means that no method can be added to a collection. Moreover, a collection may not be used to create a new species by inheritance (as explained in the next section).

4.2 Parametrisation

This section describes a first mechanism to incrementally build new species from existing ones: the parametrisation.

4.2.1 Collection parameters

Remember that methods cannot be polymorphic (cf. 4.1.2). For example, how to implement the well-known polymorphic type of lists ? Grouping elements in a list does not depend of the type of these elements. The only constraint is that all elements must have the same type. Hence, a ML-like representation of lists would be like:

```
type 'a list =
  | Nil
  | Cons of ('a * 'a list)
```

The `'a` is a parameter of the constructor type `list`, which is indeed a polymorphic ML type.

In FoCaLiZe we would like to create a species looking like:

```
species List =
  signature nil : Self ;
  signature cons : 'a -> Self -> Self ;
end ;;
```

Instead of abstracting the type parameter and leaving it free in the context of the species, in FoCaLiZe we *parametrise* the species by a **collection parameter**, the parameter named `Elem` in the example:

```
species List (Elem is Basic_object) =
  signature nil : Self ;
  signature cons : Elem -> Self -> Self ;
end ;;
```

The collection parameters are introduced by their name, followed by the `is` keyword, followed by an **interface name** (remember that an interface has the same name as its underlying species). In the example, `Basic_object` is a pre-defined species from the standard library, containing only few methods and this name is used here to denote the interface of this species. A collection parameter can be instantiated by any collection which interface is *compatible* with the one required by the parametrised species (cf. 4.1.4). In the example, any effective parameter instantiating `Elem` is a collection which interface contains at least the methods listed in the interface of `Basic_object`.

In the example, we use the parameter `Elem` to build the signature of the method `cons`. Note that collection names can be used in type expressions to denote the "abstracted" representation of the collection. Here "abstracted" means that the representation is not visible but we can refer to it as an abstract type. In other words, `Elem -> Self -> Self` stands for the type of a function:

- taking a first argument whose type is the representation of a collection having a compatible interface with the interface `Basic_object`. (This especially means that such an argument is created using methods of the compatible collection),
- taking a second argument whose type is the representation of the current species,
- and returning a value whose type is the representation of the current species.

Why a collection parameter and not a species parameter?

The answer to this question is especially important to understand the programming model in FoCaLiZe. It is a **collection parameter** because ultimately, at the terminal nodes of the development, this parameter will have to be instantiated by an entity where everything is defined, so at least a complete species. Imagine how to build an executable code if a parameter can be instantiated by a species with some methods only declared... This is the first reason.

Remember that properties mentioned in the collection interface have been proved in the underlying complete species. Indeed in the hosting species, these theorems can be used as lemmas to do current proofs. If the collection representation was not abstracted, then some methods of the hosting species would have the ability to directly manipulate entities of the collection parameter, with the risk of breaking some invariants of the collection parameter. This is the second reason. Thus the representation of a collection parameter is abstract for the hosting, exactly as is the representation of a collection (cf. 4.1.5).

To summarize, declaring a collection parameter for a parametrised species means providing two things: the (capitalized) name of the parameter and the interface (denoted by a species name) that the instantiation of this parameter must satisfy.

It is important at this point to note that FoCaLiZe deals with dependent types, and therefore that *the order of the parameters is important*. To define the type of a parameter, one can use the preceding parameters. For instance, assuming that a parametrised species `List` declares the basic operations over lists, one can specify a new species working on couples of respectively values and lists of values like:

```
species MyCouple (E is Basic_object, L is List (E)) =
  representation = (E * L) ;
  ... ;
end ;;
```

The representation of this species represents the type `('a * ('a list))`. This means that the type of the values in the first component of the couple is the same than the type of the elements of the list in the second component of the couple.

A parametrized species (like `MyCouple` in the example) cannot be only partially instantiated. An instantiation for **all** its parameters is required.

The previous example used a parameter to build the representation of the species. Collection parameters can also be used via their other methods, i.e. signatures, functions, properties and theorems, denoted by the parameter's name followed by the "!" character followed by the method name.

To create a species describing a notion of generic couple, it suffices to use two collection parameters, one for each component of the couple. To define a printing (i.e. returning a string, not making side effect in our example) method, it suffices to require each collection parameter to provide one. Now the printing method has only to add parentheses and comma around and between what is printed by each parameter's printing routine.

```
(* Minimal species requirement : having a print routine. *)
species Base_obj =
  signature print : Self -> string ;
end ;;

species Couple (C1 is Base_obj, c2 is Base_obj) =
  representation = (C1 * C2) ;
  let print (c in Self) =
    match (c) with
     | (component1, component2) ->
       "(" ^ C1!print (component1) ^
       ",␣" ^
       C2!print (component2) ^")" ;
end ;;
```

Hence, `C1!print (component1)` means "call the collection `C1`'s method `print` with the argument `component1`".

The qualification mechanism using "!" is general and can be used to denote the method of any available species/collection, even those of species being defined (i.e. `Self`). Hence, in a species instead of calling:

```
species Foo ... =
  let m1 (...) = ... ;
  let m2 (...) = if ... then ... else m1 (...) ;
end ;;
```

it is allowed to explicitly qualify the call to `m1` by "!" with no species name, hence implicitly telling "from myself":

```
species Foo ... =
  let m1 (...) = ... ;
  let m2 (...) = if ... then ... else !m1 (...) ;
end ;;
```

In fact, without explicit "!", the FoCaLiZe compiler performs the name resolution itself, allowing a lighter way of writing programs instead of always needing a "!" character before each method call.

4.2.2 Entity parameters

There is a second kind of parameter: the **entity-parameter**. Such a parameter can be instantiated by an **entity of a certain collection**.

For example, to obtain a species offering addition modulo an integer value, we need to parametrise it by an entity of a collection implementing the integers and to give a way to build an entity representing the value of the modulo. Such a parameter is called an **entity parameter** and is introduced by the keyword `in`.

```

species AddModN (Number is InterfaceForInts, val_mod in Number) =
  representation = Number ;
  let add (x in Self, y in Self) =
    Number!modulo (Number!add (x, y), val_mod) ;
end ;;

species
```

Hence, any collection created from `AddModN` embeds the addition modulo the effective value instantiating `val_mod`. It is then possible to create various collections with each a specific modulo value. For instance, assuming that the species AddModN is complete and have a method from_int able to create a value of the representation from an integer, we can create a collection implementing addition modulo 42. We also assume that we have a collection ACollImplentingInts having at least InterfaceForInts as interface.

```
collection AddMod42 =
  implement
    AddModN
      (ACollImplentingInts,
       ACollImplentingInts!from_int (42));
end ;;
```

Currently, entity parameters must live "in" a collection. It is not allowed to specify an entity parameter living in a basic type like int, string, bool... This especially means that these basic types must be embedded in a collection if we want to use their values as entity parameters.

4.3 Inheritance and its mechanisms

We now address the second mechanism to build complex species based on existing ones. It will cover the notion of *inheritance* and its related feature, the *late-binding*.

4.3.1 Inheritance

FoCaLiZe *inheritance* is the ability to create a species, not from scratch, but by integrating methods of other species. The inheritance mechanism also allows to redefine methods already existing as long as they keep the same type expression. For theorems to have the same type is simply to have the same statement (but proofs can differ).

During inheritance, it is also possible to replace a signature by an effective definition, to redefine a property by a theorem and in the same idea, to add a proof of to a property in order to conceptually redefine it as a theorem. Moreover new methods can be added to the inheriting species.

Since inherited methods are owned by the species that inherits, they are called exactly like if they were defined "from scratch" in the species.

For instance, assuming we have a species IntCouple that represent couples of integers, we want to create a species OrderedIntCouple in which we ensure that the first component of the couple is lower or equal to the second. Instead of inventing again all the species, we will take advantage of the existing IntCouple and "import" all its methods. However, we will have to change the creation function since it must ensure at creation-time of a couple (so at run-time) that it is indeed properly ordered. OrderedIntCouple has all the methods of IntCouple, except create which is redefined and the property is_ordered that states that the couple is really ordered.

```
species IntCouple =
  representation = (int * int) ;
  let print (x in Self) = ... ;
  let create (x in int, y in int) = (x, y) ;
  let first (c1, c2) = c1 ;
  ...
end ;;

species OrderedIntCouple =
  inherit IntCouple;
  let create (x in int, y in int) =
```

```
    if x < y then (x, y) else (y, x) ;

  property is_ordered : all c in Self, first (c) <= scnd (c) ;
end ;;
```

Multiple inheritance, i.e. inheriting from several species is allowed by specifying several species separated by comma in the `inherit` clause. The inheriting species inherits of all the methods of inherited species. When a name appears more than once in the inherited species, the compiler proceeds as follows.

If all the inherited species have only declared representations, then the representation of the inheriting species is only declared, unless it is defined in this inheriting species. If some representations are declared, the other ones being defined, then the totally defined representations of inherited species must be the same and this is also the one of the inheriting species. In the following example, species `S3` will be rejected while species `S4` has `int` as representation.

```
species S0; -- no defined representation
end;;
species S1 =
representation = int ; .. end ;;
species S2 =
representation = bool; ... end;;
species S3 = inherit S1, S2; ... end;;
species S4 = inherit S0, S1; ... end;;
```

If some methods of inherited species have the same name, if they are all signatures or properties, if these species have no parameters, then signatures and properties must be identical. If some of these methods have already received a definition and if they have the same type, then the definition which is retained for the inheriting species is the one coming from the rightmost defining parent in the `inherit` clause. For instance below, if species `A`, `B` and `C` provide a method `m` which is defined in `A` and `B` but only declared in `C`, then `B!m` is the method inherited in `Foo`.

```
species Foo = inherit A, B, C, D;
  ... m (...) ... ;
end ;;
```

Inheritance and parametrisation If a species `S1` inherits from a parametrised species `S0`, it must instantiate all the parameters of `S0`. Due to the dependent types framework, if `S1` is itself parametrised, it can use its own parameters to do that.

Assume we have a species `List` parametrised by a collection parameter representing the kind of elements of the list. We want to derive a species `ListUnique` in which elements are present at most once. We build `ListUnique` by inheriting from `List`.

```
species List (Elem is ...) =
  representation = Elem list;
  let empty = ... ;
  let add (e in Elem, l in Self) = ... ;
  let concat (l1 in Self, l2 in Self) = ... ;
end ;;

species ListUnique (UElem is ...) =
  inherit List (UElem);
  let add (e in UElem, l in Self) =
    ... (* Ensure the element e is not already present. *) ;
  let concat (l1 in Self, l2 in Self) =
    ... (* Ensure elements of l1 present in l2 are not added. *) ;
end ;;
```

UElem is a formal collection parameter of ListUnique which acts as an effective collection parameter in the expression ListUnique. The representation of ListUnique is UElem list. The representation of UElem is hidden: it denotes a collection. But, the value constructors of the type list are available, for instance, for pattern-matching.

As a consequence, if two methods in inherited species have the same name and if at least one of them is itself a parametrised one, then the signatures of these methods are no longer required to be identical but their type must have a common instance after instanciation of the collection parameters.

Species inheriting species parametrised by Self A species can also inherit from a species parametrised by itself (i.e. by Self). Although this is rather tricky programming, the standard library of FoCaLiZe shows such an example in the file *weak_structures.fcl* in the species Commutative_semi_ring. Indeed this species specifies the fact that a commutative semi-ring is a semi-ring on itself (as a semi-ring of scalars). In such a case, this implies that the current species must finally (when inheritance is resolved) have an interface compatible with the interface required by the collection parameter of the inherited species. The FoCaLiZe compiler collects the parts of the interface of Self obtained either by inheritance or directly in the species body. Then it checks that the obtained interface is indeed compatible with the required interfaces of the parametrised inherited species. if so, the compiler is able to build the new species. Thus the compiler tries to build a kind of fix-point but this process is always terminating, issuing either the new species or rejecting it in case of interface non-compliance.

4.3.2 Species expressions

We summarize the different ways of building species. The first way is to introduce a simple collection parameter, requiring that the effective parameter can offer all the methods listed in the associated interface.

```
species List (Elem is Basic_object) = ... ;
```

Then, we can iterate the process and build a species parametrised by a parametrised species, like in the example:

```
species MyCouple (E is Basic_object, L is List (E)) = ... ;;
```

Going on, we can inherit from species that are referenced only by their name, like in:

```
species OrderedIntCouple = inherit IntCouple; ... ;;
```

And finally, we mix the two possibilities, building a species by inheritance of a parametrised species, like in:

```
species ListUnique (UElem is ...) = inherit List (UElem); ... ;;
```

Hence, we can now define more accurately the notion of **species expression** used for both inheritance and parametrisation. It is either a simple species name or the application of a parametrised species to as many collection expressions as the parametrised species has parameters.

4.4 Late-binding and dependencies

4.4.1 Late-binding

When building by multiple inheritance (cf. 4.3.1) some signatures can be replaced by functions and properties by theorems. It is also possible to associate a definition of function to a signature (cf. 4.1.2)or a proof to a property. In the same order, it is possible to redefine a method even if it is already used by an existing method. All these features are relevant of a mechanism known as *late-binding*.

During compilation, the selected method is always the **most recently defined** along the inheritance tree. This especially means that as long as a method is a signature, in the children the effective implementation of the method will remain undefined (that is not a problem since in this case the species is not complete, hence cannot lead to a collection, i.e. code that can really be executed yet). Moreover, if a method `m` previously defined in the inheritance tree uses a method `n` freshly **re**defined, then this **fresh redefinition** of `n` will be used in the method `m`.

This mechanism enables two programming features:

- The mean to use a method known by its type (i.e. its prototype in term of Software Engineering), but for which we do not know, or we don't need or we don't want yet to provide an implementation.
- To provide a new implementation of a method while keeping the initial implementation for the inherited species. For example, the inheriting species can provide some new information (representation, functions, ..) which allow a more efficient implementation of a given function.

4.4.2 Dependencies and erasing

We previously saw that methods of a species can use other methods of this species and methods from its collection parameters. This induce what we call **dependencies**. There are two kinds of dependencies, depending on their nature:

- **Decl-dependencies**
- **Def-dependencies**

In order to understand the difference between, we must inspect further the notion of representation, function, and theorem.

4.4.2.1 Decl-dependencies

When defining a function, a property or a theorem it is possible to use another functions or signatures. For instance:

```
species Bla =
  signature test : Self -> bool ;
  let f1 (x in string) = ... ;
  let f2 (y in Self) = ... f1 ("Eat␣at␣Joe's") ... ;
  property p1 : all x in Self, test (f2 (x)) <-> test (f1 ("So␣what")) ;
  theorem t1 : all x in Self, p1 <->  test (f1 ("Bar"))
  proof = ... ;
end ;;
```

In this cases, knowing the type (or the logical statement) of the used methods is sufficient to ensure that the using method is well-formed. The type of a method being provided by its **declaration**, we will call these induced dependencies **decl-dependencies**.

Such dependencies also arise on the representation as soon as the type of a method makes reference to the type `Self`. Hence we can have dependencies on the representation as well as on other methods.

Hence, in our example, `test`, `f2`, `f1` (since it is used in `p1` and `t1` as the argument of `test` which expects an argument of type `Self`), `p1` and `t1` have a decl-dependency on the representation. Moreover, `f2` has one on `f1`. The property `p1` has decl-dependencies on `test`, `f1` and `f2` and `Self`. And finally `t1` decl-depends on `p1`, `test`, `f1` and `Self`.

4.4.2.2 Def-dependencies

A method m has a **def-dependency** over another one p if the system needs to know the **definition** of p to ensure that m is well-formed.

A definition of function can create only decl-dependencies on methods differing from the representation since the type system of FoCaLiZe only needs the types of the names present in the body of this function. Note also that when **using** a signature in another method, since signature only contain types, no def-dependencies can arise.

Now remember that `representation` is also a method and there is no syntactical way to forbid constructions like `if representation = int ..` in function or properties. Such definitions would have a **def-dependency** on the representation. For consistency reasons going beyond the scope of this manual but that will be shortly presented below in 4.4.3.2, the FoCaLiZe system rejects functions and properties having def-dependencies on the representation.

There remains the case of theorems. This case is the most complex since it can lead to def-dependencies in proofs. For the same reasons as for properties, the FoCaLiZe system rejects theorems whose statements have def-dependencies on the representation. Other def-dependencies are accepted. These dependencies must be introduced by the statement of the proof (with a syntax given in section 5.1). Now, what does mean for a theorem to def-depend on a method ? This basically means that to make the proof of the theorem statement, one must use not only the declaration of a method, but also its definition, its body. This is a needed and powerful feature.

4.4.2.3 Erasing during inheritance

As a consequence of def-dependencies and late-binding, if a method is redefined, all the proofs of theorems having def-dependencies on these methods are erased. This means that since the body of the method changed, may be the proof is not correct anymore and must be done again. In practice, it can happen that the proof still holds, but the compiler can't ensure this, hence will turn the theorem into a property in the species where the redefinition occurred. The developer will then have to provide a new proof of the inherited theorem thanks to the `proof of` field. For example, any sorting list algorithm must satisfy the invariant that its result is a sorted list with the same elements as its effective argument but the proof that indeed this requirement is satisfied depends on the different possible implementations of sort. It is perhaps possible to decompose this proof into different lemmas to minimize erasing by redefinition, some lemmas needing only decl-dependencies over the redefined method.

4.4.2.4 Dependencies on collection parameters

Since collection parameters always have their representation abstracted, hidden, only **decl-dependencies** can appear in the parametrised species using them. Hence they can never lead to erasing. These dependencies are only used internally by the FoCaLiZe compiler in order to generate the target code. For this reason, we will not focus anymore on them.

4.4.3 More about methods definition

We will now examine more technical points in methods definitions.

4.4.3.1 Well-formation

FoCaLiZe providing late-binding, it is possible to **declare** a method `m0` and use it in another **defined method** `m1`.

```
species S0 =
  signature m0 : Self ;
  let m1 = m0 ;
end ;;
```

In another species `S1`, it is also possible to **declare** a method `m1` and use it in another **defined method** `m0`.

```
species S1 =
  inherit S0;
  signature m1 : Self ;
  let m0 = x ;
end ;;
```

As long as these two species have no interactions no problem can arise. Now, we consider a third species `S2` inheriting from both `S0` and `S1`.

```
species S2 =
  inherit S0, S1;
  ...
end ;;
```

The inheritance mechanism will take each method **definition** from its hosting species: from `S0` for `m1` and from `S1` for `m2`. We have hence a configuration where `m0` calls `m1` and `m1` calls `m0`, i.e. the two methods are now mutually recursive although it was not the case where each of them was **defined**.

To avoid this situation, we will say that a species is well-formed if and only if, once inheritance is resolved, no method initially not recursive turns to become recursive. The FoCaLiZe compiler performs this analysis and rejects any species that is not compliant to this criterion. In the above example, an error would be raised, explaining how the mutual recursion (the cycle of dependencies) appears, i.e. from `m1` to `m0` (and implicitly back to `m1` from `m0`).

```
Species 'S2' is not well-formed.  Field 'm1' involves a non-declared recursion
for the following dependent fields:  m1 -> m0.
```

4.4.3.2 Def-dependencies on the representation

As we previously said (cf. 4.4.2.2) def-dependencies on the representation are not allowed in properties and theorems. The reason comes from the need to create consistent species interfaces. Let's consider the following species with the definitions:

```
species Counter =
  representation = int ;
  let inc (x in Self) = x + 1 ;
  theorem inc_spec : all x in Self, inc (x) >= x + 1
    proof = ... ;
end ;;
```

The statement of `inc_spec` contains a def-dependency on the representation since to type-check this statement, one need to know that the representation is `int`. To create the species' interface, we must make the representation abstract, hence hiding the fact that it is `int`. Without this information it it now impossible to type-check `inc_spec` body since it makes explicit reference to `+`, `<=`, `1` that are operations about `int`.

In practice, such an error is reported as a typechecking error telling that `representation` “is not compatible with type” `t` where `t` is the type expression that was assigned to the representation (i.e. `int` in our example).

Chapter 5

The FoCaLiZe Proof Language

5.1 Proofs of theorems

As presented in 3.2.8.3, FoCaLiZe proposes 3 ways to make proof of properties. We will only deal here with proofs written in the FoCaLiZe Proof Language. As a reminder, the proofs written as direct Coq or Dedukti scripts will be addressed in 9.0.2; the last kind of proof, by `assumed` doesn't need anymore description since it consists in bypassing the formal proof mechanism.

The syntax of proofs is as follows.

$$
\begin{array}{lcl}
\textit{proof} & ::= & \{\textit{proof-step}\}^{*}\ \textit{qed-step} \\
& | & \texttt{by}\ \{\textit{fact}\}^{+} \\
& | & \texttt{conclude} \\
& | & \texttt{coq proof}\ \textit{enforced-dependencies}\ \ \textit{external-code} \\
& | & \textit{enforced-dependencies}\ \ \texttt{assumed}\ \textit{external-code}
\end{array}
$$

$$
\textit{enforced-dependencies} \quad ::= \quad \{\textit{enforced-dependency}\}^{*}
$$

$$
\begin{array}{lcl}
\textit{enforced-dependency} & ::= & \\
& | & \texttt{definition of}\ \{\textit{definition-name}\}^{+} \\
& | & \texttt{property}\ \{\textit{property-name}\}^{+}
\end{array}
$$

A proof is either a leaf proof or a compound proof. A leaf proof (introduced with the `by` or `conclude` keywords) invokes Zenon with the assumptions being the given facts and the goal being the goal of the proof itself (i.e. the statement that is proved by this leaf proof). See below for the kinds of facts that can be given.

The `conclude` keyword is used to invoke Zenon without assumptions.

A compound proof is a sequence of steps that ends with a `qed` step. The goal of each step is stated in the step itself, except for the `qed` step, which has the same goal as the enclosing proof.

$$
\textit{proof-step} \quad ::= \quad \textit{proof-step-bullet}\ \ \textit{statement}\ \ \textit{proof}
$$

A proof step starts with a proof bullet, which gives its level of nesting. The top level of a proof is 0. In a compound proof, the steps are at level one plus the level of the proof itself.

For example, consider the following proof:

```
theorem implications : all a b : bool, a -> (b -> a)
proof =
  <1>1 assume a : bool, b : bool,
       hypothesis h1 : a,
       prove b -> a
       <2>1 hypothesis h2 : b,
            prove a
            by hypothesis h1
       <2>2 qed
            by step <2>1
  <1>2 conclude
      (* or: qed conclude
         or: qed by step <1>1 *) ;;
```

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

After the proof bullet comes the statement of the step. This is the statement that is asserted and proved by this step. At the end of this step's proof, it becomes available as a fact for the next steps of this proof. In our example, step `<2>1` is available in the proof of `<2>2`, and `<1>1` is available in the proof of `<1>2`. Note that `<2>1` is not available in the proof of `<1>2`: see section 5.1.2 for the scoping rules.

After the statement is the proof of the step. See below (under Statements) for a description of what is the current goal for this proof.

qed-step ::= *proof-step-bullet* `qed` *proof*
| *proof-step-bullet* `conclude`

A `qed` step is similar to a normal step, except that its statement is the goal of the enclosing proof. It may be reduced to the word `conclude` when its proof is reduced to `conclude`. In our example, we could have replaced `<1>2` with:

```
<1>2 conclude
```

In other words, a `qed` step serves ending the enclosing proof. In the former example, the `<2>2 qed` ends the proof of `<1>1` whose aim was `b -> a`.

Similarly, the `<1>2 conclude` (equivalent to `<1>2 qed by step <1>1` or `<1>2 qed conclude`) ends the complete proof, i.e. the one implicitly at level 0, i.e. the whole theorem itself.

statement ::= {*assumption* `,` }* [`prove` *logical-exp*]

A statement must be non-empty: at least one assumption or the `prove` part must be present.

A statement appearing in a step has two readings: internal and external. The external reading is for the rest of the proof: the current step proves that the assumptions imply the conclusion (i.e. the *logical-exp* that appears after `prove`). The internal reading is for the proof of the step: the current goal is the `prove` expression, and the assumptions are available as facts.

assumption ::= `assume` *ident* `:` *type-exp*
| `hypothesis` *ident* `:` *logical-exp*
| `notation` *ident* `=` *exp*

An assumption can either introduce a new (universally quantified) variable with its type (first form), or a new named hypothesis (second form), or a *named notation* (third form).

A named notation can be unfold in the goal or in other facts.

fact ::= `definition of` {[[*ident*] `#`] *ident*}(`,`)$^+$
| `hypothesis` {*ident*}(`,`)$^+$
| (`property` | `theorem`) {[[[[*ident*] `#`] *ident*] `!`] *ident*}(`,`)$^+$
| `type` {*type-ident*}(`,`)$^+$
| `step` {*proof-step-bullet*}(`,`)$^+$

A fact used in a leaf proof can be a definition, a hypothesis, a property, a theorem, a type name, or a step.

Giving a definition as a fact allows Zenon to unfold this definition in the goal and in the other facts.

Giving a hypothesis/property/theorem as a fact allows Zenon to use this hypothesis/property/theorem to prove the goal.

Giving a type name as a fact allows Zenon to use this type definition to prove the goal.

Giving a *proof-step-bullet* as a fact allows Zenon to use the (external reading of the) corresponding step as an assumption to prove the goal. Note that even if several steps are labelled with this proof bullet, only one of them is in scope at any point, so there is no ambiguity (see section 5.1.2).

5.1.1 Other simple proofs examples

Here are given some trivial theorems proven using the FoCaLiZe Proof Language to get familiar with the syntax and the way to think when making proofs. It must be understood that the prover aims at helping by combining intermediate proved properties. A proof is no more to be split up to a "trivial" level like it is the case in Coq. For instance, in the following theorem, the fact that from `~ b` and `(b \/ c)` one can deduce `c` is automatically handled by the prover.

```
theorem and_ors :
  all a b c : bool, ((a \/ b) /\ (b \/ c)) -> (~ b -> c)
  proof =
  <1>1 assume a : bool, b : bool, c : bool,
       hypothesis h1 : (a \/ b) /\ (b \/ c),
       prove ~ b -> c
       <2>1 hypothesis h2: ~ b,
            prove c
            by hypothesis h1, h2   (* Indeed, (~ b /\ (b \/ c)) -> c *)
       <2>2 qed
            by step <2>1
  <1>2 conclude ;;
```

The following example illustrates how to split a proof at the lowest level, since in this case, we just need Zenon to use hypotheses we introduced, only forcing it to deduce `a` and `b` from `a /\ b`.

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

```
  (* Sketch, assume a /\ b, then prove a and prove b, each of them being
     trivial consequence of a /\ b. *)
  <1>1 assume a : bool, b : bool,
       hypothesis h1: a /\ b,
       prove a \/ b
       <2>1 prove a
            by hypothesis h1
       <2>2 prove b
            by hypothesis h1
       <2>3 qed
            by step <2>1, <2>2
  <1>2 conclude
            (* or: qed conclude
            or even: qed by step <1>1 *) ;;
```

In fact, such a theorem is so simple that Zenon is able to automatically prove it:

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

also pass. However, some more complex proofs will need explicit intermediate steps that Zenon can't automatically guess. More advance cases can be found in the tutorials provided in the `focal` distribution.

5.1.2 Scoping rules

The scope of a step bullet extends from the end of the proof of that step to the end of the proof of the enclosing step (i.e. the end of the proof of the `qed` step that has the same level as this step). This means that proof bullets can be reused in other branches of the proof to name different steps.

The scope of an assumption is the proof of the step where this assumption appears.

5.1.3 Zenon options

The list of Zenon options and their meaning is given by typing: `zenon -h`.

Chapter 6

Recursive function definitions

When defining a recursive function FoCaLiZe, as Coq, expects a proof that this function terminates. A termination proof is stated immediately after the body of its related recursive function definition:

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

Termination proofs apply to recursive methods of species as well as to toplevel recursive functions with the limitations presented in the next section. Recursive functions or methods can be used as Zenon hints in proofs like any other ones (**`by definition of`**).

As usually, proving that a function terminates implies showing that each recursive call is made on an argument which is "smaller" than the initial one. The notion of "smaller" can be stated by different means.

6.1 Limitations

Termination proofs are not supported for some forms of recursion. In such cases, it may be possible to omit the proof of the related functions, letting the compiler generating an assumed proof. The following shapes of recursion or recursive functions do not currently support termination proofs:

- Local recursive functions (nested in functions or more generally expressions).
- Nested recursive calls, i.e. of the form `f (f (e))`.
- Mutually recursive functions other than those structural on their first argument.
- Termination proofs using a *measure* can only refer to **one** parameter of the function among all the ones it has.
- In this early state, **polymorphic** toplevel recursive functions may be incorrectly generated, leading to a Coq error.

6.2 Kinds of termination proofs

FoCaLiZe proposes 3 ways to prove the termination of a function: by a structural decreasing of an argument, using a well-founded relation or using a measure. Each possibility is detailed in the following sections.

From a technical point of view, the difference in the generated code only affects the Coq code. Recursive structural functions are compiled to the construct `Fixpoint` of Coq. Functions that are not recursive structural or that do not have termination proofs are compiled to the `Function` construct of Coq.

6.2.1 Structural termination

A structural recursive function is characterized by recursive calls performed on a subterm of the initial decreasing argument. Hence this argument must be of an inductive type. The termination proof is stated by: `termination proof = structural on` *arg*.

6.2.1.1 Example

This function is a simple computation of the length of a list. The recursive call is performed on the tail of the list thanks to the pattern-matching. We have `l = _ :: q`, hence `q` is structurally smaller than `l`.

```
open "basics" ;;

let rec length (l : list (int)) =
  match l with
  | [] -> 0
  | _ :: q ->  1 + (length (q))
termination proof = structural l ;;
```

6.2.1.2 Example

This function tests if a value `x` belongs to the list `l`. The decreasing is the same than in the previous example. This example illustrates the possibility to have a function with several arguments and state the chosen one as decreasing.

```
open "basics" ;;

let rec mem (x, l) =
  match l with
  | [] -> false
  | h :: q ->  if x = x then true else mem (x, q)
termination proof = structural l ;;
```

6.2.1.3 Wrong Example

```
open "basics" ;;

let rec zero (x) =
  match x with
  | 0 -> 0
  | n -> zero (n - 1)
termination proof = structural x ;;
```

In this case $(n-1)$ is not a subterm of n. This latter has type `int` which is not an inductive type. Currently, the compiler does not ensure that the type is inductive, leading to a Coq error. This is a known limitation and weakness of the compiler which should be fixed in a future release.

6.2.1.4 Example

```
open "basics" ;;

type pint_t =
 | Z
```

```
  | S (pint_t)
;;

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

This new function uses a representation of the integers being an inductive type (Peano's integer indeed). The structural decreasing is now ensured and this definition is correct.

6.2.2 Termination by a well-founded relation

Note: some explanations of this section come from [6].

The core of terminating recursion is that there are no infinite chains of nested recursive calls. This intuition is commonly mapped to the mathematical idea of a well-founded relation.

We illustrate this section with a function `div` computing the quotient in the Euclidien division of two integers. We made this function total by returning the (wrong) value 0 in order to only focus on termination.

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

let rec div (a, b) =
  if a <= 0 || b <= 0 then 0
  else ( if (a < b) then 0 else 1 + div ((a - b), b) )
termination proof = order pos_int_order on a ... ;
```

From the user's point of view, despite `div` has two arguments, only the first one `a` is of interest for termination. The well-founded relation used here, `pos_int_order` (of type `int` → `int` → `bool`) is the usual ordering on positive integers provided by the standard library. The well-foundedness obligation of this relation, is stated by `is_well_founded (pos_int_order)`, which also provided by the FoCaLiZe standard library (in the module `wellfoundation.fcl`).

The function `div` having only one recursive call, the only decreasing proof obligation will be:

$\forall a : int, \forall b : int, \neg(a \leq 0 \vee b \leq 0) \rightarrow \ \neg(a < b) \rightarrow pos\_int\_order(a - b, a)$

where the conditions on the execution path leading to the recursion must be accumulated as hypotheses.

The termination proof consists in as many steps as there are recursive calls, each one proving the ordering (according to the relation) of the decreasing argument and the initial one, then one step proving that the termination relation is well-founded and an immutable concluding step (`<1>e` **`qed coq proof`** `{* wf_qed*}`) telling to the compiler to assemble the previous steps, generate some stub code using a built-in Coq script to close the proof. The complete termination proof for `div` is:

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

let rec div (a, b) = ...
  termination proof = order pos_int_order on a
    <1>1 prove all a : int, all b : int,
      ~ (a <= 0 || b <= 0) ->
      ~ (a < b) -> pos_int_order (a - b, a)
      <2>1 assume  a : int, b : int,
           hypothesis H1: ~ (a <= 0 || b <= 0),
           hypothesis H2: ~ (a < b),
           prove pos_int_order (a - b, a)
```

```
        <3>1 prove b <= a
             by property int_not_lt_ge, int_ge_le_swap hypothesis H2
        <3>2 prove 0 <= a
             by property int_not_le_gt, int_ge_le_swap, int_gt_implies_ge
             hypothesis H1
        <3>3 prove 0 < b
             by property int_not_le_gt, int_gt_lt_swap hypothesis H1
        <3>4 prove (a - b) < a
             by step <3>1, <3>2, <3>3 property int_diff_lt
        <3>e qed by step <3>4, <3>2 definition of pos_int_order
   <2>e conclude
 <1>2 prove is_well_founded (pos_int_order)
    by  property pos_int_order_wf
<1>e qed coq proof {*wf_qed*} ;;
```

Proof obligations can be printed by the compiler to prevent the user from guessing them. For this, the termination proof must be stated as `order on ...` with an `assumed` Zenon proof:

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

let rec div (a, b) = ...
  termination proof = order pos_int_order on a
  assumed ;;
```

Then invoke the focalizec compiler with the option `-show-term-obls` (or shorter `-sto`). During the compilation, the obligations get printed on the standard output like:

```
---------------------------------------------------------------
Termination proof obligations for the recursive function 'div':
<1>1 prove all a : basics#int, all b : basics#int,
  (basics#|| (basics#<= (a, 0), basics#<= (b, 0)) = false) ->
  ((basics#< (a, b)) = false) -> wellfoundation#pos_int_order
  ((basics#- (a, b)), a)
<1>2 prove is_well_founded (wellfoundation#pos_int_order)
<1>e qed coq proof {*wf_qed*}
---------------------------------------------------------------
```

To summarize, a recursive function whose termination relies on a well-founded relation is given by the four following points:

1. the relation (which can be an arbitrary expression, not only an identifier),
2. the theorem stating that this relation is well-founded,
3. a theorem for each recursive call, stating that the arguments of the recursive calls are smaller than the initial arguments according to the given relation,
4. the recursive function with a termination proof of the shape (the order of the obligations does not matter):

```
<1>x proofs of right ordering for each recursive call
     (the same statements than corresponding theorems in point 3,
      even if it is possible to directly inline the proofs instead)
<1>x+1 proof of the relation being well-founded
     (same statement than in point 2, same remark than in steps <1>x)
<1>x+2 qed coq proof {*wf_qed*}
```

6.2.3 Termination by a measure

Note: some explanations of this section come from [6].

We consider here the particular case when the termination relies on a *measure* – a function that returns a natural number – which must decrease at each recursive call. We want to ease such termination proofs even if it would be possible for the user to use the previous approach, by constructing himself a well-founded relation from the measure. Precisely, the compiler does this job for him.

A measure has to be positive, which will be a proof obligation. However, the relation built from the measure being internalized, its well-foundedness is no more asked to the user. From the user's point of view, the proof obligation for each recursive call must now show that the measure decreases on the argument of interest between each call.

We illustrate this section with a function `mem` checking if an element belongs to a list. Although a structural argument could also be used, we chose to rely on the decreasing *length* of the list where the element (whose "type" is a parameter of the hosting species) is recursively searched. We first write the method `length` whose termination is simply structural on its argument `l`. Then we write the method `mem`, stating a termination proof using the measure `length` on its argument `l`.

```
species Ex_mes (A is Basic_object) =
  let rec length (l : list(A)) =
    match l with
    | [] -> 0
    | h :: q ->  1 + length (q)
  termination proof = structural l ;

let rec mem (l, x: A) =
    match l with
    | [] -> false
    | h :: q ->  h = x || mem (q, x)
  termination proof =
    measure length on l ... ;
end ;;
```

From the user's point of view, despite `mem` has two arguments, only `l` is of interest for the termination. The measure being `length`, the first proof obligation is:

$\forall l : list(A), 0 \leq length(l)$

Then, the method `mem` having only one recursive call, the only decreasing proof obligation is:

$\forall l : list(A), \forall q : list(A), \forall h : A,\ l = h :: q \rightarrow length(q) < length(l)$

where variables bound on the execution path leading to the recursion must be accumulated as hypotheses. Here, the recursion being in a pattern-matching case, `h` and `q` must be related to the matched value `l`. The core of the decreasing fact is the $<$ relation between the argument of the recursive call and the one in the current call.

For readability, instead of inlining the proofs of these obligations, the user can state two related properties or theorems before the function `mem` itself.

```
  property length_pos : all l: list (A), 0 <= length (l) ;

  theorem mes_decr : all l : list(A) , all q : list(A), all h : A,
    l = h :: q -> length (q) < length (l)
  proof = ... ;
```

Note that `length_pos` is a *property*, not a theorem: it is not yet proved. Now the termination proof consists in as many steps as there are recursive calls, each one proving the strict decreasing of the measure, then one step proving that the measure is positive and an immutable concluding step (`<1>$x+2$` **`qed coq proof`** `{*mf_qed*}` telling to the compiler to assemble the previous steps and generate some stub code to close the proof. Note that `mf_qed` and `wf_qed` are equivalent. Both keywords are provided for users willing to make a difference between termination by a well-founded relation and a measure in their source code.

```
  let rec mem (l, x: A) = ...
  termination proof = measure length on l
    <1>1 prove all l : list(A) , all q : list(A), all h : A,
           l = h :: q -> length (q) < length (l)
         by property mes_decr
    <1>2 prove all l: list (A), 0 <= length (l)
         by property length_pos
    <1>e qed coq proof {*mf_qed*} ;
```

To summarize, from the user's point of view, a recursive function whose termination relies on a measure is given by the four following points:

1. The measure function returning a regular integer (which raises the issue that $<$ is well-founded on naturals, not integers). The measure can be an arbitrary expression, not only an identifier (c.f. Section 6.2.3.1 for a more detailled discussion).
2. The theorem stating that the measure is always positive or null.
3. A theorem for each recursive call, stating that the measure on the argument of interest decreases.
4. The recursive function with a termination proof of the shape:

```
<1>x proofs of decreasing for each recursive call
     (the same statements than corresponding theorems in point 3,
      even if it is possible to directly inline the proofs instead)
<1>x+1 proof of the measure being always positive or null
     (same statement than in point 2, same remark than in steps <1>x)
<1>x+2 qed coq proof {*mf_qed*}
```

6.2.3.1 Using Other Function Parameters or Arbitrary Expression

As previously stated, although the user usually states his relation or his measure by giving the name of the corresponding function, any arbitrary expression can be used. In the case of a measure, this makes possible to also use some other parameters of the function than the decreasing one.

In the (non-meaningful and dummy) following example, the function `mem` (whose termination could be proved in a simpler way since it is structural) has twos parameters and the non-decreasing one, `x`, is also used in the measure expression.

```
let dummy (x : int) = 0 ;;
let rec mem (l, x: int) =
  match l with
  | [] -> false
  | h :: q ->  h = x || (mem (q, x))
termination proof = measure (function ll -> dummy (x) + length (ll)) on l
...
```

This obviously impacts the proof obligations since `x` must be bound in their statements, leading to an additional **`all`** `x :int` to bind `x`. The compiler takes care of this point while printing the obligations to the user.

```
  (* Obligation 1: strict decreasing at each recursive call. *)
  <1>1 prove all l : list(int) , all x : int, all q : list(int), all h : int,
         l = h :: q -> dummy (x) + length (q) < dummy (x) + length (l)
       ...
  (* Obligation 2: measure always >= 0. *)
  <1>2 prove all x : int, all l: list (int), 0 <= dummy (x) + length (l)
```

```
    ...
<42>e qed coq proof {*wf_qed*} ;;
```

6.2.3.2 Recursive Functions in Dedukti

The Dedukti backend is not able to check termination. The translation of recursive functions to Dedukti is detailed in [7].

Chapter 7

Compiler options

When invoking the FoCaLiZe compiler with the focalizec command, various command line options can be provided. The compiler can process several files in their order of apparition in the command line. Several types of files are handled. By default, if no option is specified, the default behaviour is of the compiler is:

- ".ml" and ".mli" files are compiled with the OCaml compiler producing bytecode. It is possible to customise the compiler code generation using the `-ocaml-comp-mode` option. The version of OCaml used is automatically selected from the configuration options selected during FoCaLiZe's installation. The FoCaLiZe standard library path is implicitly passed to OCaml.
- ".v" files are compiled with the Coq compiler. The version of Coq used is automatically selected from the configuration options selected during FoCaLiZe's installation. The FoCaLiZe standard library path is implicitly passed to Coq.
- ".sk" files are compiled with the Dedukti compiler using an alternative parser named Sukerujo. The FoCaLiZe standard library path is implicitly passed to Dedukti.
- ".zv" files are compiled by Zenon via zvtov. The generated ".v" file is then compiled by Coq as describe above.
- ".sk.zv" files are compiled by Zenon Modulo via zvtov. The generated ".sk" file is then compiled by Dedukti as describe above.
- ".fcl" files are compiled by focalizec, generating the ".ml" OCaml source and the ".sk.zv" and ".zv" pre-checker source. The ".ml" file is then sent to OCaml, the ".zv" file is sent to Zenon to finally get a ".v" file that is sent to Coq and the ".sk.zv" file is sent to Zenon Modulo to finally get a ".sk" file that is sent to Dedukti.

It is possible to control the kind of files generated by focalizec (no Coq, no Dedukti, no OCaml, ".zv", ".sk.zv", ".v", ".sk" using options described bellow.

* −**coq-version** *version name*. Enables Coq code and compilation compatibility for a specific version. If the option is not set, by default the latest version of Coq supported by FoCaLiZe is assumed (at least 8.5pl1). Since Coq version 8.5, the −**I** option is replaced by the −**Q** one to specify the loadpath.
* −**dot-non-rec-dependencies** *directory name*. Dumps non-let-rec dependencies of the species present in the compiled source file. The output format is suitable to be graphically displayed by dotty (free software available via the graphviz package). Each species will lead to a dotty file into the argument directory. Files are names by "deps_" + the source file base name (i.e. without path and suffix) + the species name + the suffix ".dot".

* −**focalize-doc** Generates documentation. The result file gets located in the same directory than the compiled file, replacing the suffix ".fcl" by ".fcd". This file contains XML in plain ASCII text and need to be processed before being read. Consult section 8 for more details.
* −−**experimental** Reserved for development purpose. Never use. Invoking the compiler with this option may trigger unpredictable results.
* −**i**. Prints the interfaces of the species present in the compiled source file. Result is sent to the standard output.
* −**I** *directory name*. Adds the specified directory to the path list where to search for compilation units. Several −I options can be used. The search order is in the local directory, then in the standard library directory (unless the −no-stdlib-path option is used, see below), then in the directories specified by the −I options in their apparition order on the command line.
* −**impose-termination-proof**. Make termination proofs mandatory for recursive functions. If a recursive function doesn't have its termination proof, then the field will be considered as not fully defined and no collection will be built on the species hosting the function. By default this option is not enabled and if a recursive function does not have any termination proof, a warning is printed during compilation when trying to make a collection from this species.
* −**methods-history-to-text** *directory name*. Dumps the methods' inheritance history of the species present in the compilation unit. The result is sent as plain text files into the argument directory. For each method of each species a file is generated wearing the name made of "history_" + the source file base name (i.e. without path and suffix) + "_" + the hosting species name + the suffix ".txt".
* −**no-ansi-escape**. Disables ANSI escape sequences in the error messages. By default, when an error is reported, bold, italic, underline fonts are used to make easier reading the message. Using this option removes all these text attributes and may be used if your terminal doesn't support ANSI escape sequences or, for example, if compiling under emacs.
* −**no-coq-code**. Disables the Coq code generation. By default Coq code is always generated.
* −**no-dedukti-code**. Disables the Dedukti code generation. By default Dedukti code is generated if Dedukti, Sukerujo, and Zenon Modulo are all found at FoCaLiZe configuration time.
* −−**no-ocaml-code**. Disables the OCaml code generation. By default OCaml code is always generated.
* −**no-stdlib-path**. Does not include the standard library installation directory in the libraries search path. This option is rarely useful and mostly dedicated to the FoCaLiZe compiler build process.
* −**ocaml-comp-mode** *file name*. Specifies the OCaml compiler code generation mode. This option is folowed by a string that can be "byt" for bytecode compilation, "bin" for native code compilation, or "both" for bytecode and native code compilation. This option has no effect if −−**no-ocaml-code** is used.
* −**pretty** *file name*. (Undocumented: mostly for debug purpose). Pretty-prints the parse tree of the FoCaLiZe file as a FoCaLiZe source into the argument file.
* −**pmatch-err-as-warn**. Toggles pattern-matching issues (uncomplete matching and useless pattern) as warnings instead of errors (default behaviour) that would abort compilation.
* −**raw-ast-dump**. (Undocumented: mostly for debug purpose). Prints on stderr the raw AST structure after parsing stage.
* −**scoped_pretty** *file name*. (Undocumented: mostly for debug purpose). Pretty-prints the parse tree of the FoCaLiZe file once scoped as a FoCaLiZe source into the argument file.
* −**stop-before-coq** When Coq code generation is activated, stops the compilation process before pass-

ing the generated file to Coq. The generated pre-Coq source is sent to Zenon then the compilation process stops. The produced file is hence ended by the suffix ".v". This option has no effect if −**no-coq-code** or −**stop-before-zenon** is used.

* −**stop-before-dedukti** When Dedukti code generation is activated, stops the compilation process before passing the generated file to Dedukti. The generated pre-Dedukti source is sent to Zenon Modulo then the compilation process stops. The produced file is hence ended by the suffix ".sk". This option has no effect if −**no-dedukti-code** or −**stop-before-zenon** is used.
* −**stop-before-zenon**. When Coq or Dedukti code generation is activated, stops the compilation process before passing the generated file to Zenon or Zenon Modulo. The produced file is then a pre-Coq source file or a pre-Dedukti source file, ended by the suffix ".zv" or ".sk.zv". This option has no effect if both −**no-coq-code** and −**no-dedukti-code** are used.
* −**verbose**. Sets the compiler in verbose mode. It will then generate the trace of the steps and operations is does during the compilation. This feature is mostly used for debugging purpose but can also explain the elaboration of the model during compilation for people interested in FoCaLiZe's compilation process.
* −**v**. Prints the FoCaLiZe version then exits.
* −**version**. Prints the full FoCaLiZe version, sub-version and release date, then exits.
* −**where**. Prints the binaries and libraries installation directories then exits.
* −**zvtovopt** *"options"*. Set options to pass to zvtov. zvtov is anyway always called with options *"-zenon (path to Zenon/Zenon Modulo) -new"* in head. It is possible to use this option to pass options to Zenon but be careful of correctly quoting options to make sure they are interpreted by their respective recipient.

  For instance, telling Zenon to change its maximum memory size can be done in the 3 equivalent ways:

  ```
  focalizec -zvtovopt '-zopt '\''-max-size 2G'\' foo.fcl
  focalizec -zvtovopt "-zopt \"-max-size 2G\"" foo.fcl
  focalizec -zvtovopt "-zopt '-max-size 2G'" foo.fcl
  ```
* −**help** −−**help**. Prints the summary of command line options (i.e. this documentation) on the standard output.

Chapter 8

Documentation generation

When invoked with the `−focalize-doc` option, the command focalizec generates an extra file (with the ".fcd" suffix) containing "documentation" information extracted from the compiled source file.

This information describes the different elements found in the source file (species, collections, methods, toplevel definitions, type definitions) with various annotations like type, definition/inheritance locations. It also contains the special comments previously called **annotations** (cf. 3.1.4) and that were kept during the compilation process. Moreover, these annotations can contain special tags used by the documentation generator of FoCaLiZe.

8.0.1 Special tags

FoCaLiZe's documentation system currently supports 5 kinds of tags. They impact the content of the final generated document, either in its content or in the way information is displayed depending on the output format. These tags start with the "@" character and the content of the tag follows until the end of the line. It is then possible in an annodation to mix regular text that will not be interpreted and tags.

8.0.1.1 @title

This tag must appear (i.e. is only taken into account) in the first annotations block of the source file. The following text is considered to be the title of the source file and will appear in the header of the final document.

See example provided for the `@description` tag below.

8.0.1.2 @author

This tag must appear (i.e. is only taken into account) in the first annotations block of the source file. The following text is considered to be the author of the source file and will appear in the header of the final document.

See example provided for the `@description` tag below.

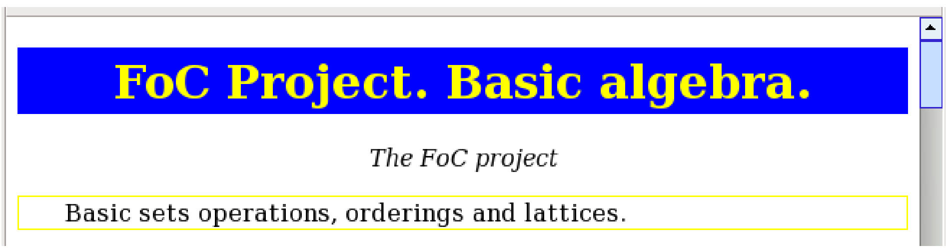
8.0.1.3 @description

This tag must appear (i.e. is only taken into account) in the first annotations block of the source file. The following text is considered to be the description of the content of the source file (what services it implements) and will appear in the header of the final document. For example:

```
(***********************************************************************)
(*                                                                     *)
(*                        FoCaLiZe Compiler                            *)
(*                                                                     *)
(*  Copyright 2007 LIP6 and INRIA                                      *)
(*  Distributed only by permission.                                    *)
(***********************************************************************)

(**
 @title FoC Project. Basic algebra.
 @author The FoC project
 @description Basic sets operations, orderings and lattices.
*)
...
```

will lead to a document header like (displayed in HTML format):



You may notice in the above source code example that the header information is located in an annotation that is not the **first** one. In effect, the top-most banner starting by:

```
(***********************************************************************)
```

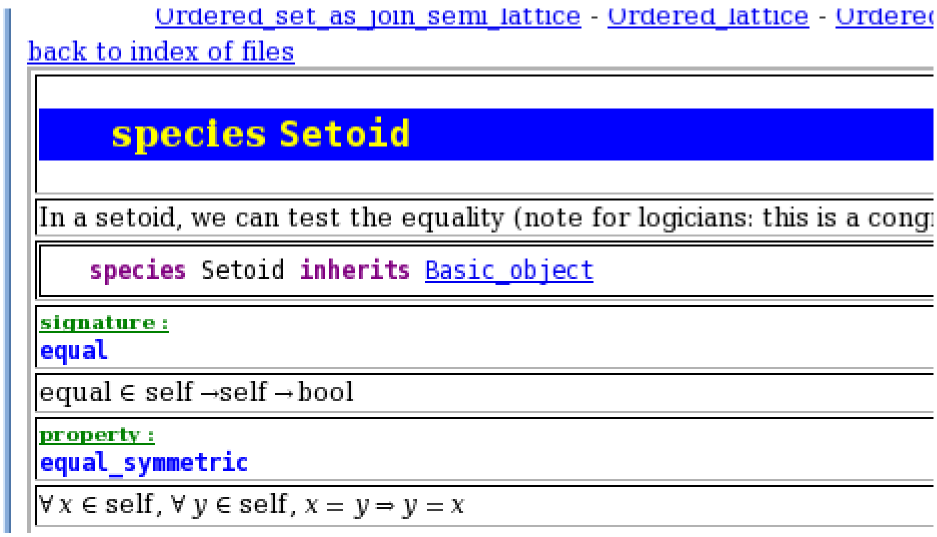
is in fact also an annotation since it starts by the sequence “(**”. However all these annotation belong to the same annotations block as requiered.

8.0.1.4 @mathml

This tag must appear in the document comment preceding a method definition. It indicates the sequence of MathML code to use to replace the name of the method everywhere in the current document. This tag only affects the HTML display since it allows to show more usual symbols rather than identifiers in a browser. This is expecially useful for mathematical formulaes where one prefer to see the sign = rather than an identifier “`equal`”. For example:

```
(** In a setoid, we can test the equality (note for logicians: this is
   a congruence). *)
species Setoid =
  inherit Basic_object;
  (** @mathml <eq/> *)
  signature equal : Self -> Self -> bool ;
  property equal_transitive : all x y z in Self,
    equal (x, y) -> equal (y, z) -> equal (x, z) ;
  ...
```

will replace any occurrence of the method `equal` by the “`<eq/>`” MathML sequence that displays a = sign when displayed by an HTML browser.



8.0.2 Transforming the generated documentation file

The generated documentation file is a plain ASCII text containing some XML compliant with FoCaLiZe's DTD (`focalize/focalizec/src/docgen/focdoc.dtd`). Like for any XML files processing is performed thank to the command xsltproc with XSL stylesheets (".xsl" files).

You may write custom XSL stylesheets to process this XML but the distribution already provides 2 stylesheets to format this information.

8.0.2.1 XML to HTML

Transformation from ".fcd" to a format that can be read by a WEB browser is performed in two passes. These 2 steps can be invoked into one unique Unix-shell command (all on the same line without carriage return) as provided by the script `fcd2html.sh` installed in the default binaries installation directory:

```
xsltproc ''directory to the stylesheet''/focdoc2html.xsl mysrc.fcd |
xsltproc ''directory to the stylesheet''/mmlctop2_0.xsl - > mysrc.xml
```

More in details, the above command-line performs the 2 following explicit steps:

1. Convert the ".fcl" file to HTML with MathML annotations. This is done applying the stylesheet `focalize/focalizec/src/docgen/focdoc2html.xsl` with the command xsltproc. For example:

   ```
   xsltproc ''directory to the stylesheet''/focdoc2html.xsl mysrc.fcd > tmp
   ```

2. Convert the HTML+MathML temporary file into HTML. This is done applying the stylesheet `focalize/focalizec/src/docgen/focdoc2html.xsl` with the command xsltproc. For example:

   ```
   xsltproc ''directory to the stylesheet''/mmlctop2_0.xsl tmp > mysrc.xml
   ```

> **Attention:** You may note that the final result file name must be ended by the suffix "**.xml**" otherwise your browser won't be able to interpret it correctly and won't display symbols ($\Rightarrow, \in, \exists, \rightarrow, \ldots$) correctly.

Attention: By default, `xsltproc` validates the input files against DTDs. Depending on your system configuration, some of these DTDs may be fetched from the net, resulting in a long processing time (in fact, connexion latency). You may save time deactivating these checks using the option `-novalid` of `xsltproc`. Files generated by focalizec are anyway assumed to be compliant to these DTDs.

8.0.3 XML to LaTeX

Currently not officially available.

Chapter 9

Hacking deeper

9.0.1 Interfacing FoCaLiZe with other languages

9.0.2 Dealing with hand-written Coq and Dedukti proofs

Chapter 10

Compiler error messages

Unable to find file '$name$' in the search path.

Description: The source file made reference to a FoCaLiZe compilation unit $name$ (by the `open` or `use` directives, or by explicit qualification with the "#" notation) but the related FoCaLiZe file was not found in the current libraries search path.

Hints: Locate in which directory the missing file is and add this directory to the libraries search path with the `-I` compiler option.

Invalid or corrupted compilation unit '$name$'. May be it was compiled with another version of the compiler.

Description: The source file made reference to a FoCaLiZe compilation unit $name$ (by the `open` or `use` directives, or by explicit qualification with the "#" notation but the related FoCaLiZe file was found with an incorrect format.

Hints: May be the compilation unit was compiled with another version of FoCaLiZe or was mangled and you must compile it again with your current version.

Invalid file extension for '$name$'.

Description: The FoCaLiZe compiler expects compilation units to be ended by the suffix ".fcl", ".ml", ".mli", ".zv"or ".v". If the submitted input file doesn't end by one of these suffixes, this error message arises with the name, $name$ of the involved file.

Hints: Change the extension of the input file name or ensure the submitted input file name is the correct one.

System error - $sysmsg$.

Description: During the compilation process an error related to the operating system occurred (I/O error, permission error, file-system error, ...). The original message $sysmsg$ of the system explaining the problem follows the FoCaLiZe's message.

Hints: Consult the original message of the system and get an appropriate solution depending on this message.

Invalid OCaml compiler kind ”$string$” for option -ocaml-comp-mode. Must be ”byt”, ”bin” or ”both”.

Description: By default, if some OCaml code was generated, the FoCaLiZe compiler sends the generated code to the OCaml compiler. The default compilation mode is bytecode production. It is possible to select the native code production using the option `-ocaml-comp-mode` followed by the string “bin” or to select both code production modes by the string “both”. The argument string “byt” is not required since it is the default mode. Any other string is invalid and leads to the present error message.

Hints: Select “byt”, “bin” or “both” as argument to the `-ocaml-comp-mode` option.

No input file. focalizec is cowardly giving up...

Description: The FoCaLiZe compiler needs one input file to compile. If none is supplied, this error message arises.

Hints: Add the input source file to compile on the command line.

Lexical error str

Description: In the currently submitted source file, a sequence of characters is not recognised as legal according to the FoCaLiZe programming language legal words structure. The involved character str follows in the error message.

Hints: Change the source code at the indicated location.

Syntax error

Description: In the currently submitted source file, a phrase of the program doesn’t follow FoCaLiZe’s syntax.

Hints: Change the source code at the indicated location. It sometimes happens that the location gets fuzzy due to the parsing process. If the error is not immediate to you, explore the neighbours of the specified location. If you still can’t find out the error, have the following emergency process: comment your code and incrementally uncomment it to find the point where the error appears without having to search in the whole file. Once the error appears, have a look at the part of code you uncommented since the previous successful compilation and try to guess the syntactic cause.

Unclear syntax error msg.

Description: An error occurred during the syntactic analysis but was not reported to be due to a syntax non-compliance. This error is not clearly identified and this message is displayed as post-mortem report with the exception msg that caused the error.

Hints: None

Compilation unit 'm' was not declared as "use"

Description: It not possible to use a qualified notation for a compilation unit name (i.e. using an entity from this compilation unit by explicitly specifying the unit with the "#"-notation) before this compilation unit is declared "use" or "open". This error message indicates the location where an identifier refers to a compilation unit that was not qualified either by the `use` or `open` directive. Note that the `open` directive implicitly implies `use`.

Hints: Use the `use` directive on the compilation detected unit.

Parameterised species expected n_1 arguments but was provided n_2.

Description: A species expression (used in species parameter expression or `inherit` clause) applies a species with n_1 argument(s) although its definition declared it as using n_2 argument(s).

Hints: None.

Non-logical let must not bind '$ident$' to a property.

Description: A `let` construct (not a `logical let`) attempts to bind the identifier $ident$ to a logical expression although it can only bind it to a computational expression.

Hints: Source program to fix. May be the `let` should be turned into a `logical let` if the body of the binding is really a logical expression.

Delayed termination proof refers to an unknown method '$ident$' of the species.

Description: A `proof of` clause was found in a species for the property $ident$ but this property was not found in the species.

Hints: None.

Ambiguous logical expression. Add explicit parentheses to associate the $side$ argument of the `/\` properly.

Description: A logical expression contains a `/\` (logical "and") with at least one argument being a `->` (logical "implication") or a `<->` (logical "equivalence") without parentheses around the $side$ argument ("left" or "right"). Since this is not clear of how to associate, we ask the user to explicitly add parentheses.

Hints: Explicitly add the parentheses to make the association non-ambiguous.

Ambiguous logical expression. Add explicit parentheses to associate the $side$ argument of the `\/` properly.

Description: A logical expression contains a `\/` (logical "or") with at least one argument being a `->` (logical "implication") or a `<->` (logical "equivalence") without parentheses around the $side$ argument ("left" or "right"). Since this is not clear of how to associate, we ask the user to explicitly add parentheses.

Hints: Explicitly add the parentheses to make the association non-ambiguous.

Hypothesis, notation or variable name '$name$' already bound in the current scope of the proof.

Description: In the current proof, an hypothesis, a notation or a variable is introduced but there already exists one with this name.

Hints: Rename the mentionned hypothesis, notation or variable.

The property '$name$' doesn't belong to a collection but to a species.

Description: A proof fact attempts to use a property hosted in a species instead of in a collection. Since only collections are ensuring that properties they contain hold, only properties from collections can be used to make proofs. In effect, in a species, a property may only be stated without being provable. Relying on an unproved property may allow to write wrong proofs.

Hints: Either you have a collection implementing the species from whom you are trying to use the proof and then use the property from this collection. Or have a collection parameter whose interface is the species from whom you are trying to use the proof and then use the property from this collection parameter.

Termination proof considers '$name$' as decreasing but it is not an argument of the function.

Description: A termination proof (either `by measure` or `by order` of a recursive function uses the identifier '$name$' to pinpoint the decreasing parameter. However, this identifier is not a parameter of the function.

Hints: You may got confused in the name of the parameter. Ony one of the function's parameter can appear at this place.

Types definition mixes regular and external kinds ...

Description: A types definition contains both regular types and externally mapped ones. Mutually recursive type definitions must only contains regular (sum) types.

A mutually recursive types definition must only contain sum types ...

Description: Mutually recursive type definitions must only contains regular sum types to be compiled to Coq which requires this constraints.

Hints: If you mix alias and unions, consider transforming the aliases into sums by adding a dummy constructor. For instance, instead of:

```
type t = (int * u)
with u = | A | B (t) ;;
```

you may write:

```
type t = Dummy (int * u)
with u = | A | B (t) ;;
```

Note that if the Coq code generation is disabled (option `-no-coq-code`), the check is relaxed and the error message will not appear. However, there will be no Coq material produced.

Types definition contains several external types.

Description: Externally mapped types cannot currently be mutually recursively defined.

Unbound sum type value constructor '$name$'.

Description: An identifier representing a sum type value constructor was not found among the available sum type definitions.

Hints: Source program to fix. Since in core expressions capitalized identifiers are considered as sum type value constructors, may be you tried to use a capitalized name for one of your variables. In this case, as any variables, make it starting with a lowercase letter. Otherwise, may be your type definition is missing or not reachable in the current scope (missing explicit qualification with the "#" notation or `open` directive if your type definition is hosted in another source file).

Unbound record type label '$name$'.

Description: An identifier representing a record type label was not found among the available record type definitions.

Hints: Source program to fix. May be your type definition is missing or not reachable in the current scope (missing explicit qualification with the "#" notation or `open` directive if your type definition is hosted in another source file).

Unbound identifier '$name$'.

Description: An identifier (expected to be bound by a `let`, a pattern of a function parameter declaration) was not found.

Hints: Source program to fix. May be your definition should be toplevel and is missing or not reachable in the current scope (missing explicit qualification with the "#" notation or `open` directive if your definition is hosted in another source file).

Unbound type '$name$'.

Description: The definition of an identifier expected to be a type constructor was not found.

May be your type definition is missing or not reachable in the current scope (missing explicit qualification with the "#" notation or `open` directive if your type definition is hosted in another source file).

Unbound compilation unit '$name$'.

Description: A `open` or `use` directive or an explicit qualification by the "#" notation makes reference to a compilation unit that was not found in the current libraries search path.

Hints: Locate in which directory the missing file is and add this directory to the libraries search path with the `-I` compiler option.

Unbound species '$name$'.

Description: The definition of the species $name$ was not found in the current scope.

Hints: May be your species definition is missing or not reachable in the current scope (missing explicit qualification with the "#" notation or `open` directive if your species definition is hosted in another source file).

Species '$name$' is not a collection. Its carrier can't be used in type expressions.

Description: A type expression makes reference to the carrier of either a collection that doesn't exist or a species (which is not a collection).

Hints: The common confusion is to consider that a complete species is like a collection. However, only carriers of collections are allowed in type expressions. Ensure you didn't make reference to a species instead of an effective collection.

Type name '$name$' already bound in the current scope.

Description: In a source file it is not allowed to redefine a type definition. This means that each type name definition must be unique inside a file. However, it is possible to have several type definitions with the same names as long as they are in different source files (even if they are used together via `open` directives of explicit qualification by the "#" notation).

Hints: None.

Species name '$name$' already bound in the current scope.

Description: In a source file it is not allowed to redefine a species definition. This means that each species name definition must be unique inside a file. However, it is possible to have several species definitions with the same names as long as they are in different source files (even if they are used together via `open` directives of explicit qualification by the "#" notation).

Hints: None.

Types t_1 and t_2 are not compatible.

Description: The typechecking system detected a type conflict between two expressions t_1 and t_2 that were expected to be type-compatible.

Hints: Source program to fix. This is mostly due to an attempt to use the type of a `representation` although it is turned abstracted by the collection or parametrisation mechanisms. In this case, ensure that you are not trying to make assumptions on the type of a collection parameter or a collection.

Type t_1 occurs in t_2 and would lead to a cycle.

Description: The FoCaLiZe type system does not allow cyclic types. This especially means that a type expression must not be a sub-part of itself to prevent cycles.

Hints: None.

Type constructor '$name$' used with conflicting arities: n_1 and n_2.

Description: A type expression applies a type constructor $name$ to n_1 argument(s) although its definition declared it as using n_2 argument(s) (or in the other order, depending on the way the error was detected: in any way the definition and the usage of the type involve 2 different numbers of arguments).

Hints: None.

No expected argument(s).

Description: A type expression applies a type constructor to arguments although this constructor needs none.

Hints: None.

In method '$name$', type scheme sch contains free variables.

Description: As presented in 4.1.2, species methods cannot be polymorphic. The method $name$ has a type scheme shown by sch which is polymorphic.

Hints: You may explicitly add type annotations (constraints) on the arguments or/and return type of your method definition. If you need some kind of such polymorphism, use the collection parameter mechanism.

Sum type value constructor '$name$' expected n_1 arguments but was used with n_2 arguments.

Description: The sum type constructor $name$ is used with a bad number of arguments. It was declared to use n_1 arguments but is used with n_2.

Hints: None.

Unbound type variable $name$.

Description: In a type expression, a type variable $name$ is not bound.

Hints: Source program to fix. May be the type expression appears in a parametrised type definition where you forgot to specify the type constructor's parameter in head of the definition.

Method '$mname$' multiply defined in species '$sname$'.

Description: Like for toplevel definitions, method definitions inside a species must not bind several times the same name. In the species $sname$, the method $mname$ is defined several times.

Hints: Source program to fix. May be you defined several times the same method and in this case, remove one of the definitions. Or if the different occurrences of $mname$ refer to different conceptual functions, change the names to make them different.

Delayed proof of '$name$' was found several times in the species. Other occurrence is at: loc.

Description: A delayed proof of the property $name$ was found several times in the same species (i.e. not via inheritance but directly in the species body). Only one must be kept.

Hints: None.

In species '$sname$', proof of '$pname$' is not related to an existing property.

Description: In the species $sname$ a delayed proof of the property $pname$ was found but the statement of this property doesn't exist in the current species even via inheritance.

Hints: May be you forgot to write the property, or you mistook on the property name the proof is related to or you forgot to inherit from a species having this property.

Representation is multiply defined.

Description: In a species, the method `representation` is multiply defined in the body of the species although at most one definition must be provided.

Hints: Source program to fix. Remove the spurious definitions.

If the `representation` method is not directly present in the body, that is because the species inherits from a parent where the representation is already defined. In this last case, since the parent's structure is already established, you must remove the `representation` method in the species where the error was reported.

Representation is multiply defined by multiple inheritance and was formerly found of type t_1 and newly found of type t_2.

Description: In the species, several parents brought by inheritance several incompatible definitions of the representation. The error message reports t_1 and t_2, two incompatible types found for the representation definition.

Hints: None.

’Self’ can’t be parametrised by itself.

Description: This error appears when `Self` appears as a species identifier used in a species expression that is a parameter of the current defined species.

Hints: None.

A ”is” parameter can only be instantiated by an identifier of a collection.

Description: In a species expression, a parametrised species by an entity parameter (`is`-parameter) is provided an effective argument that is not a collection identifier.

Hints: None.

Illegal use of toplevel species ’S’ as effective argument.

Description: Effective parameter in species expressions (for inheritance or collection implementation) can only be collections or collection parameters. In the present case, a toplevel species was used instead which is inconsistent.

Hints: None.

Collection ’s_1’ is not compatible with ’s_2’. In method ’$name$’, types t_1 and t_2 are not compatible.

Description: During collection parameter instantiation, the interface of the provided collection s_1 is not compatible with the interface s_2, because it doesn’t have a signature containing at least s_2’s methods with compatibles types. The wrong field $name$ is reported with the two types t_1 and t_2 expected and actually found.

Hints: None.

Collection ’s_1’ is not compatible with ’s_2’. In method ’$fname$’, type t_1 occurs in t_2 and would lead to a cycle.

Description: During collection parameter instantiation, the interface of the provided collection s_1 is not compatible with the interface s_2, since type compatibility check detected a cyclic type. This means that the type t_1 is a sub-part of itself via the type t_2.

Hints: None.

Collection ’s_1’ is not compatible with ’s_2’. In method ’$fname$’, the type constructor ’$tname$’ is used with the different arities n_1 and n_2.

Description: During collection parameter instantiation, the interface of the provided collection s_1 is not compatible with the interface s_2, since the type constructor (not sum type constructor) $tname$ is used with an improper number of arguments n_1 versus n_2.

Hints: None.

Collection 's_1' is not compatible with 's_2'. Method '$name$' is not present in 's_1'.

Description: During collection parameter instantiation, the interface of the provided collection s_1 is not compatible with the interface s_2, because it doesn't have a signature containing at least s_2's methods and especially not the method $name$.

Hints: None.

Parameterised species is applied to n arguments.

Description: A parameterised species is applied to a wrong number n of effective arguments.

Hints: None.

Species '$sname$' cannot be turned into a collection. Following field(s) is(are) not defined: ...

Description: A collection is built out of a completely defined species (cf. 4.1.5), i.e. a species where **all** the methods are **defined** and not only declared. In the species $sname$, the listed methods are only declared (or missing a termination proof in case of recursive functions and usage of the `-impose-termination-proof`). Hence the species is not complete and no collection can be extracted from it.

Note that missing termination is by default only a **warning** and is turned into an error by using the `-impose-termination-proof` option on the command line. In effect, the notion of to "be defined" applies to recursive functions which must have a termination proof provided in addition to their computational body.

Hints: Add an effective definition of the method, either by writing it code or by inheritance, according to your program model.

Add an effective termination proof to the function or do not invoke the `-impose-termination-proof` option when compiling the source file.

Warning: In species '$sname_1$', proof of '$pname$' could be done earlier in '$sname_2$'.

Description: This message is only a **warning**. It states that the property $pname$ whose proof was done in the species $sname_1$ could be proved earlier in the inheritance (mor eaccurately, in the species $sname_2$ from which $sname_1$ inherits) because all the material used in the proof was already available in the species $sname_2$. This espacially means that all the methods the proof **def**-depends on were alrady **defined**, and all the methods the proof **decl**-depends on were already **declared**.

Hints: Move the proof of the property directly in the species $sname_2$. If the property is also hosted in $sname_2$, then it can directly be turned into a theorem, merging the 2 fields `property` and `proof of`.

In the delayed termination proof, parameter '$name$' does not refer to a parameter of the original function.

Description: As any proof, termination proofs can be made later after the function definition. However it must refer to the original function's parameters names. In the current proof, the identifier $name$ doesn't exist among the original function's parameters.

Hints: Change the parameter name in the proof to make it matching the function definition's ones.

Method '$mname$' was found with incompatible types during inheritance. In species 's_1': τ_1, in species 's_2': τ_2.

Description: During inheritance, a method $nmane$ was found with 2 incompatible types. Remind that all along the inheritance tree, methods must not change their type. The two found types and the species hosting the definitions having these types are provided by 's_1'and τ_1 (resp. 's_2'and τ_2).

Hints: None.

Logical method '$mname$' appearing in species 's_1' should have the same statement than in species 's_2' at $source - location$.

Description: During inheritance, a theorem or a property $nmane$ was redefined but with a different statement. As described at the beginning of 4.3.1, the inheritance mechanism also allows to redefine methods already existing as long as they keep the same type expression. For theorems to have the same type is simply to have the same statement. A same property can be written in several semantically equivalent ways. For instance, transitivity of an operation $\odot$ can be written by: $\forall x, y, z \in S, x \odot y \Rightarrow y \odot z \Rightarrow x \odot z$ or $\forall x, y, z \in S, (x \odot y \wedge y \odot z) \Rightarrow x \odot z$. FoCaLiZe does not try to establish the equality of these two expressions. It only compares syntactically the statements modulo variables renaming (i.e. α-conversion) and non-significant parentheses.

Hints: The simplest way is to rewrite the logical statement of the inheriting species as it was written in the inherited species.

Definition '$name$' is considered as both logical and non-logical.

Description: In the inheritance tree of the current species, a method $name$ was previously found a "logical" and is now found no more "logical".

Hints: Ensure that you did not define 2 methods with the same name but for different purposes (one to help in stating logical expressions and the other for your computational behaviour).

Definition '$name$' is redefined although marked final.

Description: In the inheritance tree of the current species, a method $name$ was previously defined with the keyword `final` forbidding any later redefinition while the current species try to redefine it.

Hints: If the previous definition was flagged as to be never changed, there may have good reasons. Try to understand while the software architecture imposed this constraint (before – may be dangerously– thinking at removing the `final` qualifier!).

Species 'sname' is not well-formed. Method '$name$' involves a non-declared recursion for the following dependent methods: ...

Description: The species $sname$ doesn't respect the well-formation rule presented in 4.4.3.1. The chain of functions involved in the cycle is given in the error message as a sequence of methods names $m_1 \rightarrow m_2 \rightarrow \ldots \rightarrow m_n$ with the implicit final path $m_n \rightarrow m_1$.

Hints: None.

No $lang$ mapping given for the external value definition '$name$'.

Description: The external value definition allowing to link FoCaLiZe code to foreign languages doesn't specify how to map the value identifier $name$ in the language $lang$.

Hints: Supply a binding for this language in the external definition.

No $lang$ mapping given for the external type definition '$name$'.

Description: The external type definition allowing to link FoCaLiZe code to foreign languages doesn't specify how to map the type identifier $name$ in the language $lang$.

Hints: Supply a binding for this language in the external definition.

No $lang$ mapping given for the external sum type value constructor '$name$'.

Description: The external sum type definition allowing to link FoCaLiZe code to foreign languages doesn't specify how to map the sum type constructor $name$ in the language $lang$.

Hints: Supply a binding for this language in the external definition.

No $lang$ mapping given for the external record field '$name$'.

Description: The external record type definition allowing to link FoCaLiZe code to foreign languages doesn't specify how to map the record field $name$ in the language $lang$.

Hints: Supply a binding for this language in the external definition.

Unable to find OCaml generation information for compiled file '$file$'. Compilation unit may have been compiled without OCaml code generation enabled.

Description: The FoCaLiZe compilation unit file $file$.fcl was compiled but the object file doesn't contain information about OCaml code generation. The FoCaLiZe compiler allows to disable the OCaml code production by the `--no-ocaml-code` option. May be this option was used.

Hints: Invoke the compiler on the source file $file$.fcl without the `--no-ocaml-code` option.

Record type definition contains a mutable field '$name$' that can't be compiled to Coq.

Description: Never raised in the current version since mutable record fields are not yet available.

Unable to find Coq generation information for compiled file '$file$'. Compilation unit may have been compiled without Coq code generation enabled.

Description: The FoCaLiZe compilation unit $file$.fcl was compiled but the object file doesn't contain information about Coq code generation. The FoCaLiZe compiler allows to disable the Coq code production by the `--no-coq-code` option. May be this option was used.

Hints: Invoke the compiler on the source file $file$.fcl without the `--no-coq-code` option.

Unable to find Dedukti generation information for compiled file '$file$'. Compilation unit may have been compiled without Dedukti code generation enabled.

Description: The FoCaLiZe compilation unit $file$.fcl was`--no-dedukti-code` compiled but the object file doesn't contain information about Dedukti code generation. The FoCaLiZe compiler allows to disable the Dedukti code production by the `--no-dedukti-code` option. May be this option was used.

Hints: Invoke the compiler on the source file $file$.fcl without the `-no-dedukti-code` option and with the `-dedukti-code` option.

Using a collection parameter's method ($name$) in a Zenon proof with "by definition" is not allowed.

Description: The current proof tries to used the definition of a method $name$ of a species parameter. Since species parameters are always abstracted, **definitions** (i.e. "bodies") of their methods are **not** available in the parametrised species. For this reason, it is impossible to provide this definition to Zenon.

Hints: None.

Using an only declared method of Self ($name$) in a Zenon proof with "by definition" is not allowed.

Description: The current proof tries to used the definition of a method $name$ **only declared** in the current species. Since the definition is not available, it is impossible to provide it to Zenon.

Hints: None.

Using a local identifier ($name$) in a Zenon proof with ”by definition” is not allowed.

Description: The current proof tries to used a local variable $name$, i.e. an identifier not representing a method, hence meaningless for Zenon.

Hints: None.

Using a local identifier ($name$) in a Zenon proof with ”by property” is not allowed.

Description: The current proof tries to used a local variable $name$, i.e. an identifier not representing a method, hence meaningless for Zenon.

Hints: None.

Assumed hypothesis 'hyp' in a Zenon proof was not found.

Description: The current proof makes a reference to an hypothesis hyp that was not found in the current proof tree.

Hints: None.

Step '$<\ldots>\ldots$' in a Zenon proof was not found.

Description: The current proof makes a reference to an proof step that was not found in the current proof tree.

Hints: None.

Termination proof is stated as structural on the parameter '$name$' which does not syntactically decrease.

Description: A termination proof was given for a recursive function by a structural criterion. However, the syntactical analysis of the argument used for the recursive call does not show that the given argument stricly decreases.

Hints: May be you stated a wrogn parameter name, or your recursive call(s) really do not strictly decrease(s).

Mutual recursion is not yet supported for Coq code generation. At least functions '$name_1$' and '$name_2$' are involved in a mutual recursion.

Description: The current version of FoCaLiZe does not yet handle Coq code generation for non-structural mutual recursive functions. At least the two such functions $name_1$ and $name_2$ were found as mutually recursive but may be the recursion involves more functions. It is then impossible to produce Coq source code.

Hints: Until this feature is available in FoCaLiZe, do not try to generate the Coq code for the source file containing these functions by using the `--no-coq-code` option.

Recursive call to '$name$' contains nested recursion.

Description: The function contains a recursive call to $name$ inside a recursive call. The current version of FoCaLiZe doesn't support the Coq code generation for nested recursive calls.

Hints: Try to rewrite your function with the nested call performed before the outer recursive call. For instance:

```
let rec f (x) =
  ...
  f (f (bla))
  ...
```

should be turned into:

```
let rec f (x) =
  ...
  let tmp = f (bla) in
  f (tmp)
  ...
```

Recursive call to '$name$' is incomplete.

Description: The function contains a recursive occurrence of $name$ with an incomplete number of parameters. Since application syntactically requires all the arguments to be present, this can arise if the recursive identifier is used in non-applicative position. However the error message is more general since future extensions may involve partial applications. Below follows an example of such invalid usage of a recursive function identifier:

```
let rec f (x) =
  ...
  let tmp = f in
  let ...  = tmp (...) ... in
  f (...)
  ...
```

Hints: None

Pattern-matching is not exhaustive.

Description: The pointed pattern-matching expression doesn't handle all the shapes of possible incoming values according to the type of the matched expression. If such a non-handled value arises at runtime, execution will be aborted. Moreover, Coq will reject generated code.

Hints: Complete the pattern-matching by adding cases covering all the possible shapes of values. Inspect the type definition related to the matched expression to find the missing constructors. The ultimate *bullet-proof* solution is to add a "catch-all" pattern, although it is not always the suitable solution: study if all the remaining cases require the same process.

Useless case in pattern-matching.

Description: The pointed pattern-matching expression contains a case already caught by a previous one. Hence, this useless case will never be followed.

Hints: Remove the spurious case. But before, make sure you didn't write above a pattern using a variable instead of a value constructor. Another common error is to use in a pattern, a variable wearing the same name than a previously bound one, thinking that pattern-matching will test equality of the matched value with the value of this previously bound variable. For instance:

```
let var = 3 in
match (var + 0) with
  | var -> false
  | _ -> true
```

will **not** check that the matched value (4) is equal to the value of `var`. In the pattern, `var` is a new identifier that will be bound to the value of the matched expression, here to 3. Hence, in this case the first branch is always taken, leaving the second one useless.

Unexpected error: "msg". Please report.

Description: An error was raised and not expected during a normal execution of the compiler. This is a failure of the compiler and must be fixed by the FoCaLiZe development team. The error message display the internal reason of the failure and must be reported to the FoCaLiZe development team.

Hints: `http://focal.inria.fr/`, link "Bug tracking".

Bibliography

[1] P. Ayrault, T. Hardin, and F. Pessaux. Development life cycle of critical software under FoCal. In ENTCS-Elsevier, editor, *Harnessing Theories for Tool Support in Software-TTSS'08*, 2008.

[2] R. Bonichon, D. Delahaye, and D. Doligez. Zenon: An Extensible Automated Theorem Prover Producing Checkable Proofs. In *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, volume 4790 of *LNCS/LNAI*, pages 151–165, Yerevan (Armenia), Oct. 2007. Springer.

[3] S. Boulmé. *Spécification d'un environnement dédié à la programmation certifiée de bibliothèques de Calcul Formel*. Thèse de doctorat, Université Paris 6, 2000.

[4] S. Boulmé, T. Hardin, V. Ménissier-Morain, and R. Rioboo. On the way to certify computer algebra systems. In *Calculemus 99*, volume 23. Elsevier, 1999.

[5] S. Boulmé, T. Hardin, and R. Rioboo. Some hints for polynomials in the Foc project. In *Calculemus 2001 Proceedings*, June 2001.

[6] F. P. C Dubois. Termination Proofs for Recursive Functions in FoCaLiZe. In *Trends in Functional Programming (to be published)*, pages –, 2015.

[7] R. Cauderlier and C. Dubois. ML Pattern-Matching, Recursion, and Rewriting: From FoCaLiZe to Dedukti. In A. Sampaio and F. Wang, editors, *Theoretical Aspects of Computing - ICTAC 2016 - 13th International Colloquium, Taipei, Taiwan, ROC, October 24-31, 2016, Proceedings*, volume 9965 of *Lecture Notes in Computer Science*, pages 459–468. Springer Berlin Heidelberg, 2016.

[8] R. Cauderlier and P. Halmagrand. Checking Zenon Modulo Proofs in Dedukti. In Kaliszyk, Cezary and Paskevich, Andrei, editor, *Proceedings Fourth Workshop on Proof eXchange for Theorem Proving, PxTP 2015, Berlin, Germany, August 2-3, 2015.*, volume 186 of *Electronic Proceedings in Theoretical Computer Science*, pages 57–73, Berlin, Germany, 2015. Open Publishing Association.

[9] D. Delahaye, D. Doligez, F. Gilbert, P. Halmagrand, and O. Hermant. Zenon Modulo: When Achilles Outruns the Tortoise using Deduction Modulo. In K. McMillan, A. Middeldorp, and A. Voronkov, editors, *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, volume 8312 of *LNCS/ARCoSS*, pages 274–290. Springer Berlin Heidelberg, dec 2013.

[10] D. Delahaye, J.-F. Étienne, and V. Viguié Donzeau-Gouge. A Formal and Sound Transformation from FoCaLiZe to UML: An Application to Airport Security Regulations. In *UML and Formal Methods (UML&FM)*, Innovations in Systems and Software Engineering (ISSE) NASA Journal, Kitakyushu-City (Japan), Oct. 2008. Springer.

[11] D. Delahaye, J.-F. Étienne, and V. Viguié Donzeau-Gouge. Formal Modeling of Airport Security Regulations using the FoCaLiZe Environment. In *Requirements Engineering and Law (RELAW)*, Barcelona (Spain), Sept. 2008. IEEE CS Press.

[12] D. Delahaye, J.-F. Étienne, and V. Viguié Donzeau-Gouge. Certifying Airport Security Regulations using the FoCaLiZe Environment. In *Formal Methods (FM)*, volume 4085 of *LNCS*, pages 48–63. Springer, Aug. 2006.

[13] D. Delahaye, J.-F. Étienne, and V. Viguié Donzeau-Gouge. Reasoning about Airport Security Regulations using the FoCaLiZe Environment. In *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, pages 45–52. IEEE CS Press, Nov. 2006.

[14] D. Doligez. Zenon, version 0.4.1. http://focal.inria.fr/zenon/, 2006.

[15] G. Dowek, T. Hardin, and C. Kirchner. Theorem Proving Modulo. *Journal of Automated Reasoning (JAR)*, 31, 2003.

[16] T. Hardin and R. Rioboo. Les objets des mathématiques. *RSTI - L'objet*, 2004.

[17] É. Jaeger and T. Hardin. A few remarks about formal development of secure systems. In *HASE*, pages 165–174. IEEE Computer Society, 2008.

[18] M. Jaume and C. Morisset. A formal approach to implement access control. *Journal of Information Assurance and Security*, 2:137–148, 2006.

[19] M. Jaume and C. Morisset. Towards a formal specification of access control. In *Joint Workshop on Foundations of Computer Security and Automated Reasoning for Security Protocol Analysis FCS-ARSPA'06 (Satellite Workshop to LICS'2006)*, 2006.

[20] M. Maarek and V. Prevosto. Focdoc: The documentation system of foc. In *Proceedings of the 11th Calculemus Symposium*, Rome, sep 2003.

[21] M.Carlier and C.Dubois. Functional testing in the focal environment. In B.Beckert and R.Hähnle, editors, *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, volume 4966 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 2008.

[22] C. Morisset. *Sémantique des systèmes de contrôle d'accès*. PhD thesis, Université Pierre et Marie Curie - Paris 6, 2007.

[23] V. Prevosto. *Conception et Implantation du langage FoC pour le développement de logiciels certifiés*. PhD thesis, Université Paris 6, sep 2003.

[24] V. Prevosto and S. Boulmé. Proof contexts with late binding. In *Typed Lambda Calculi and Applications*, volume 3461 of *LNCS*, pages 324–338. Springer, 2005.

[25] V. Prevosto and D. Doligez. Algorithms and proof inheritance in the Foc language. *Journal of Automated Reasoning*, 29(3-4):337–363, dec 2002.

[26] V. Prevosto, D. Doligez, and T. Hardin. Algebraic structure and dependent records. In *TPHOLs'2002*, volume 2410 of *LNCS*. Springer-Verlag, 2002.

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

Index