# FoCaLize

# Reference Manual

1.0.0

January 2009

**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), Damien Doligez (GALLIUM-INRIA), David Delahaye (CPR-CEDRIC), Catherine Dubois (CPR-CEDRIC), Jean-Frédéric Etienne (CPR-CEDRIC), Stéphane Fechter (SPI-LIP6), Eric Jaeger (SPI-LIP6), Mathieu Jaume (SPI-LIP6), Lionel Habib (SPI-LIP6), Thérèse Hardin (SPI-LIP6), Charles Morisset (SPI-LIP6), Ivan Noyer (SPI-LIP6), François Pessaux (SPI-LIP6), 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]

# 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 EDEMOI ANR project and with 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`

# Contents

# Introduction

## Motivations

The Foc project was launched in 1998 by T. Hardin and R. Rioboo [**?**] [5] 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[6]. 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 [**?**]), a point which enlightened the borders of this approach,

[5] 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

[6] 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 [**?**, **?**] used the OCaml language to try different solutions, recorded in [**?**].

### 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 [**?**, **?**, **?**]) and the correction of the compiler (mostly written by V. Prevosto) against Focal's semantics is proved (by hand) [**?**], a point which brings a satisfactory confidence in the language's correctness. Then R. Rioboo [**?**] 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 [**?**] 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[**?**] 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 [**?**, **?**, **?**], 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 [**?**, **?**, **?**]. This last work led to the development of a translator [**?**] 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 [**?**] and the study of some traps within formal methods by E. Jaeger and T. Hardin[**?**].

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 of FoCaLize 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 a 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 to process it. Since in FoCaLize we don't only address data type and operations, among these "things" we also find the declaration (specification) of these operations, the properties (which may represent requirements) and their proofs.

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

- The `representation` gives the data representation of entities manipulated by the *species*. It is a type defined by a type expression. The *representation* definition may be deferred, meaning that the real structure of the data-type the *species* embeds does not need to be known at this point. In this case, it is simply a type variable. However, to obtain an implementation, the *representation* has to be defined later either by setting `representation = exp` where `exp` is a type expression or by inheritance (see below). Type expressions in FoCaLize are roughly ML-like types (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, hence the data structure of the manipulated entities by the program/object/module. In such a case, the *representation* is simply the tuple grouping all these variables that were disseminated all along the 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 is given but not yet the implementation). Declaration can however be used in definition of other methods: the type provided by the *signature* allows FoCaLize to ensure that the method is used in contexts compatible with this type. The late-binding and the collection mechanisms, further introduced, ensure that the definition of the method will be effectively known when needed.

- Definitions are composed of the keyword `let`, followed by a name, a type (optional) 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`.

- Statements are composed of the keyword `property` followed by a name and a first-order formula. A *property* may serve to express requirements (i.e. facts that the system must hold to conform to the Statement of Work) and then can be viewed as a specification purpose *method*, like *signature*s were for `let`-*method*s. They induce 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. Formulae are written with usual logical connectors, universal and existential quantifications over a FoCaLize type, and name of *methods* known within the *species*' context. For instance, a *property* telling that for any vehicle, if the speed is non-null, then doors can't 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 proof of other properties or theorems. 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* together with the formal proof that their statement holds in the context of the *species*. The proof accompanying the statement will be processed by FoCaLize Zenon and ultimately checked with the theorem prover Coq.

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 [**?**] 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 be able to express functions and properties our "setoid" requires. However,

we can refer to it via `Self` (in this case a type variable). In the same way, we specify a *signature* for the equality (operator `=`). We introduce the three properties that an equality (an equivalence relation) must conform to.

We complete the example by the definition of the function `different` which use the name = (`basics#not_b` stands for the the boolean negation function defined in the FoCaLize source file `basics.fcl`). It is possible right now to 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).

It is possible to use *methods* only declared before they get a real *definition* thanks to the *late-binding* feature provided by FoCaLize. In the same idea, redefining a *method* is allowed in FoCaLize the last version being always kept as the effective *definition* inside the species.

## 1.2 Type of Species, Interfaces and Collections

The *type* of a *species* is obtained by removing definitions and proofs. Thus, it is a kind of record type, made of all the method types of the species. If the `representation` is still a type variable say $\alpha$, then the *species* type is prefixed with an existential binder $\exists\alpha$. This binder will be eliminated as soon as the `representation` will be instantiated (defined) and must be eliminated to obtain 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 interfaces in the concrete syntax, they are simply denoted by the name of the species underlying them.* 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 *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 declarations have received a definition and all properties have received a proof.

When *complete*, a species can be submitted to an abstraction process of its representation to create a *collection*. Thus the *interface* of the collection is just the *interface* of the complete species underlying it. A collection can hence be seen as an abstract data type, only usable through the methods of its interface, but having the guarantee that all declarations are defined and all statements are 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 **inheriting** of a previously 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, in FoCaLize inheritance serves two kinds of evolutions. In the first case the evolution aims making a *species* with more operations while keeping those of its parents (possibly redefining some of them if required). In the second case, the *species* only tends to be closer to a "runnable" implementation, providing explicit definitions to *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 inherits Setoid =
  signature ( * ) : Self -> Self -> Self ;
  signature one : Self ;
  let element = one * one ;
end ;;
```

*Monoid* are "things" that are *Setoids* but also have an operation `*` 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 `*` 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`.

Multiple inheritance is available in FoCaLize. For sake of simplicity, the above example uses simple inheritance. In case of inheriting a *method* from several parents, the order of parents in the `inherits` 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 retained inside it being those of the *methods* present in the *species* after inheritance resolution.

A strong constraint in inheritance is that the type of inherited, and/or redefined *methods* must not 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 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 $S!m$ has the most general unifier of these two types as its own type.

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: `inherits Setoid` gives to `Setoid_product` all the methods of `Setoid`.

```
species Setoid_product (A is Setoid, B is Setoid) inherits 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 **??**.

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 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* working on natural numbers modulo a certain value. In the expression $5$ `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 in which we are interested by its **value**, not only by its *representation* and the methods acting on it. We call those *entity parameters*, their introduction rests upon the introduction of a *collection parameter* and they denote a *value* having the type of the *representation* of this *collection parameter*.

Let us first have a *species* representing natural numbers:

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

Note that `IntModel` 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)...

We now build our *species* "working modulo ...", embedding the value of this modulo:

```
species Modulo_work (Naturals is IntModel, n in Naturals) =
  let job1 (x in Naturals) in ... =
    ... Naturals!modulo (x, n) ... ;
  let job2 (x in Naturals, ...) in ... =
    ... ... Naturals!modulo (x, n) ... ... ;
end ;;
```

Using the *entity parameter* `n`, we ensure that the *species* `Modulo_work` 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 **??**, 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* `IntModel` 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 MachineNativeIntColl implements MachineNativeInt ;;
```

Next, to get a *collection* implementing arithmetic modulo 8, we could extract from the *species* `Modulo_work` the following *collection*:

```
collection Modulo_8_work implements Modulo_work
   (MachineNativeIntColl, MachineNativeIntColl!from_string (''8'') ;;
```

As seen by this example, a species can be applied to effective parameters by giving their values with the usual syntax of parameter passing.

As said before, to ensure modularity and abstraction, the *representation* of a *collection* is hidden (as well as the definitions). This means that any software component dealing with a *collection* will only be able to manipulate it through the operations (*methods*) its interface provides. This point is especially important since it prevents other software components from possibly breaking invariants required by the internals of the *collection*.

## 1.6 Properties, Theorems and Proofs

FoCaLize aims not only to write programs, it intends to encompass both the executable model (i.e. program) and properties this model must satisfy. For this reason, "special" *methods* deal with logic instead of purely behavioural aspects of the system: *theorems*, *properties* and *proofs*.

Stating a *property* indicates that a *proof* that it **holds** will be given at some stage of the development. For *theorems*, the *proof* is directly provided with the statement. Such proofs must be done by the developer and will finally be sent to the formal proof assistant Coq who will automatically check that the demonstration of the *property* is consistent. Writing a proof can be done in several ways.

It can be written in FoCaLize's proof language, a hierarchical proof language that allows to give hints and directions for a proof. This language will be sent to an external theorem prover, Zenon[**?**, **?**] developed by D. Doligez. This prover is a first order theorem prover based on the tableau method incorporating implementation novelties such as sharing. Zenon will attempt, from these hints, to automatically generate the proof and exhibit a Coq term suitable for verification by Coq. Basic hints given by the developer to Zenon are: "prove by definition of a *method*" (i.e. looking inside its body) and "prove by *property*" (i.e. using the logical statement of a *theorem* or *property*). Surrounding this hints mechanism, the language allows to build the proof by stating assumptions (that must obviously be demonstrated next) that can be used to prove lemmas or parts for the whole property. We show below an example of such demonstration.

```
  theorem order_inf_is_infimum: all x y i in Self,
    !order_inf(i, x) -> !order_inf(i, y) ->
      !order_inf(i, !inf(x, y))
    proof =
      <1>1 assume x in Self, assume y in Self,
           assume i in Self, assume H1: !order_inf(i, x),
           assume 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 the proof itself**, the developer does not have to elaborate it formally "from scratch".

Like any automatic theorem prover, Zenon may fail finding a demonstration. In this case, FoCaLize allows to write verbatim Coq proofs. In this case, the proof is not anymore automated, but this leaves the full power of expression of Coq to the developer.

Finally, the `assumed` keyword is the ultimate proof backdoor, telling that the proof is not given but that the property must be admitted. Obviously, a really safe development should not make usage of this feature since it bypasses the formal verification of software's model. However, such a functionality remains needed for various reasons. For example, a development may be linked with external code; properties of the FoCaLize code may depends on properties of the external code that have to be stated (for example using the documentation of this code) but also assumed.

## 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 code to a specific representation (for example, there exists a generic implementation of integer addition modulo n but it can be redefined in arithmetics modulo 2 if boolean values are used to represent the two values). It 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 the consequences of any modification will be managed consistently and propagated to those depending of it.

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 properties of the body of the redefined function. All the proofs which truly depend of the definition are then invalidated by the compiler and must be done again in the context updated with the new definition. Thus the main difficulty is to choose the best level in the hierarchy to do a proof. In [**?**], Prevosto and Jaume propose a *coding style* to minimise the number of proofs to be redone in the case of a redefinition, by a certain kind of modularisation of the proofs.

### 1.7.2 Code Generation

FoCaLize currently compiles programs toward two languages, OCaml to get an executable piece of software, and Coq to have a formal model of the program, with theorems and proofs.

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

Conversely, in Coq, all the *methods* are compiled, i.e. "computational" *methods* and logical *methods* with their proofs. This allows Coq to check the entire consistence of the system developed in FoCaLize.

### 1.7.3 Tests

FoCaLize incorporates a tool named *FocalTest* [**?**] for Integration/Validation testing. It allows to confront automatically a property of the specification with an implementation. It generates automatically 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. Note that this indentifies 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, else the test case is rejected. It can be an expensive process for some kind of preconditions. To overcome this drawback, a constraint based generation is under development: it allows to produce directly test cases for which the precondition is satisfied.

### 1.7.4 Documentation

The tool called FoCaLizeDoc [**?**] automatically generates documentation, thus the documentation of a component is always coherent with respect to its implementation.

This tool uses its own XML format that contains information coming not only from structured comments (that are parsed and kept in the program's abstract syntax tree) and FoCaLize concrete syntax but also from type inference and dependency analysis. From this XML representation and thanks to some XSLT stylesheets, it is possible to generate HTML files or LaTeX files. Although this documentation is not the complete safety case, it can helpfully contribute to its elaboration. In the same way, it is possible to produce UML models [**?**] as means to provide a graphical documentation for 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).

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

Note that some distributions of FoCaLize includes these tools and are automatically installed during the FoCaLizeinstallation 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 **??** and **??**. 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 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 **??**, 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 model. This model can then be sent to the Coq proof assistant 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, i.e. a file containing Coq syntax plus "holes" in place of proofs written in the FoCaLize Proof Language. This kind of files is suffixed by ".zv" instead of directly ".v". When sending this file to Zenon these "holes" will be filled by effective Coq code automatically generated by Zenon (if it succeed in finding a proof), hence leading to a pure Coq code file that can be compiled by Coq.

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

1. FoCaLize **source compilation**. This step takes the FoCaLize source code and generates the OCaml and/or "pre-"Coq code. You can disable the code generation for one of these languages (see page **??**), or both, 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 both OCaml and Coq you will get two generated files: `source.ml` (the OCaml code) and `source.zv` (the "pre-"Coq 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. Hence this steps acts like a manual 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 takes the generated `.zv` file and attempts to produce a real Coq `.v` source file by replacing proofs written in FoCaLize Proof Language by some effective Coq proofs found by the Zenon theorem prover. Note that if Zenon fails in finding a proof, a hole will remain in the final Coq `.v` file. Such a hole appears as the text "`TO_BE_DONE_MANUALLY.`" in place of the effective proof. In this case, Coq will obviously fail in compiling the file, so the user must do the proof by hand or modify his original FoCaLize source file to get a working proof. This step acts like a manual invocation:

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

   For more about the Zenon options, consult section **??**.

4. 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.

   ```
   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.

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 Comments

Comments (possibly spanning) on several lines are introduced by the two characters `(*`, with no intervening blanks, and terminated by the characters `*)`, with no intervening blanks. Comments are treated as blanks. Comments can occur inside string or character literals (provided the `*` character is escaped) and can be nested. They are discarded during the compilation process. Example:

```
(* Discarded comment *)
species S =
  ...
  let m (x in Self) = (* Another discarded comment *)
  ...
end ;;
(* Another discarded comment at end of file *)
```

Comments spanning on a single line start by the two characters `--` and end with the end-of-line character. Example:

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

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

### 3.1.3 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 analyze 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 **??**. 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 ;;
```

### 3.1.4 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.

#### 3.1.4.1 Introduction

Sorting words to find out which kind of meaning they may have is a very common conceptual categorization 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.

#### 3.1.4.2 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 properties.

We further explain those concepts below.

#### 3.1.4.3 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 *mixfix* is written *among* its arguments, as $if\ \ldots then\ \ldots else\ \ldots$ in $if\ c\ then\ 1\ else\ 2$.

In FoCaLize, as in maths, ordinary identifiers are always prefix and binary operators are always infix.

#### 3.1.4.4 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 maths.

#### 3.1.4.5 Categorization 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 these categories are

- *lowercase*: the identifier starts with a lowercase letter and 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.
- *uppercase*: the identifier starts with an uppercase letter and 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.

We distinguish identifiers using their first "meaningful" character: the first character that is not an underscore.

#### 3.1.4.6 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.

#### 3.1.4.7 Regular identifiers

Regular lower case identifiers are used to designate the names of variables, functions, and labels of records.

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

A regular identifier is a sequence of letters, digits, and _ (the underscore character), starting with a letter or an underscore.

The identifier is lowercase if its first letter is lowercase.

The identifier is uppercase if its first letter is uppercase.

Letters contain at least the 52 lowercase and uppercase letters from the standard ASCII set. In an identifier, all characters are meaningful. Examples: `foo`, `bar`, `_20`, `__gee_42`.

#### 3.1.4.8 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, in the above examples, `+`, `++` and `=_mod_5` will be infix operators and `~+zero` will be a prefix one.

Note that some of these symbols, such as `:`, are considered as uppercase identifiers, and others, such as `+`, as lowercase identifiers. The consequence is that for example a notation starting with a `:` can be used as a constructor name for a union type, but not as a method name.

#### 3.1.4.9 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).

#### 3.1.4.10 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 go 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.

### 3.1.5 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 characters ‘ (backquote) and end by two characters ’ (quote). 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 delimitor characters and contain symbolic characters.

The first meaningful character at the beginning of a delimited ident/symbol is used to find its associated token.

### 3.1.6 Species and collection names

Species, collection names and collection parameters are uppercase identifiers.

### 3.1.7 Integer literals

$$
\begin{array}{rcl}
\textit{binary-digit} & ::= & \texttt{0} \mid \texttt{1} \\
\textit{octal-digit} & ::= & \texttt{0} \ldots \texttt{7} \\
\textit{hexadecimal-digit} & ::= & \texttt{0} \ldots \texttt{9} \mid \texttt{A} \ldots \texttt{F} \mid \texttt{a} \ldots \texttt{f} \\
\textit{sign} & ::= & \texttt{+} \mid \texttt{-} \\
\textit{unsigned-binary-literal} & ::= & \texttt{0}\ (\texttt{b} \mid \texttt{B})\ \textit{binary-digit}\ \{\textit{binary-digit} \mid \texttt{\_}\}^* \\
\textit{unsigned-octal-literal} & ::= & \texttt{0}\ (\texttt{o} \mid \texttt{O})\ \textit{octal-digit}\ \{\textit{octal-digit} \mid \texttt{\_}\}^* \\
\textit{unsigned-decimal-literal} & ::= & \textit{digit}\ \{\textit{digit} \mid \texttt{\_}\}^* \\
\textit{unsigned-hexadecimal-literal} & ::= & \texttt{0}\ (\texttt{x} \mid \texttt{X})\ \textit{hexadecimal-digit}\ \{\textit{hexadecimal-digit} \mid \texttt{\_}\}^* \\
\textit{unsigned-integer-literal} & ::= & \textit{unsigned-binary-literal} \\
 & \mid & \textit{unsigned-octal-literal} \\
 & \mid & \textit{unsigned-decimal-literal} \\
 & \mid & \textit{unsigned-hexadecimal-literal} \\
\textit{integer-literal} & ::= & [\textit{sign}]\ \textit{unsigned-integer-literal}
\end{array}
$$

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 `00`. 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`, `-0xFF_FF`.

### 3.1.8 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:

| Sequence | Character | Comment |
|---|---|---|
| \b | \008 | Backspace. |
| \n | \010 | Line feed. |
| \r | \013 | Carriage return. |
| \t | \009 | Tabulation. |
| \␣ | ␣ | Space character. |
| \" | " | Double quote. |
| \' | ' | Single quote. |
| * | * | Allows e.g. for insertion of "(*" in a string |
| \( | ( | See comment above for * |
| \) | ) | |
| \[ | [ | |
| \] | ] | |
| \{ | { | |
| \} | } | |
| \\ | \ | Backslash character. |
| \` | ` | Backquote character. |
| \- | - | 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. |
| *digit digit digit* | | The character whose ASCII code in **decimal** is given by the 3 digits following the \. This sequence is valid for all ASCII codes. |
| \0x *hex hex* | | The character whose ASCII code in **hexadecimal** is given by the 2 characters following the \. This sequence is valid for all ASCII codes. |

### 3.1.9 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 **??**).

### 3.1.10 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`, `0x5a.a3eef`, `0x5a.a3e-ef`.

### 3.1.11 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
```

### 3.1.12 Name qualification

Name qualification is done according to the compilation unit status.

As precisely described in section (**??**), 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 inherits basics#Basic_object = ... end ;;
```

can be transformed with no explicit qualification into:

```
open "basics";;
species S inherits 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 inherits 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 inherits #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’’.

### 3.1.13 Reserved keywords

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

```
alias all and as assume assumed
begin by
caml collection conclude coq coq_require
definition
else end ex external
false function
hypothesis
```

```
if in inherits internal implements 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 Coqor OCamlare also reserved (For example Set), and may cause problems at later stages of the FoCaLizecompilation.

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 **??**).

#### 3.2.1.1 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 OCamltypes; 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.

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

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.