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), 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].

ESTIME and GALLIUM are teams of INRIA Rocquencourt[4]

[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`

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

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.

Contents

Introduction

Motivations

The Foc project was launched in 1998 by T. Hardin and R. Rioboo [11] [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 Integrated Development Environment (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. 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 sceptical 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 mechanism such as modularity to help invariant maintaining. 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-5,7 or SIL 3,4) 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 and fault avoidance, fault detection by validation testing, vulnerability and safety analysis.

Initial application testbed

When the Foc project was launched by Hardin and Rioboo, only one specific domain was considered, the one of Computer Algebra. 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

[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 several different representations of polynomials according to the treatment to be done

S. Boulmé (see his thesis[3]), a point which enlightened the borders of this approach, regarding the logical background. On the other hand, before designing the syntax, it was needed to study the development style in such a language. R. Rioboo [4, 11] used the OCaml language to try different solutions which are recorded in [11].

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[17, 20, 19]) and the correction of the compiler (mostly written by V. Prevosto) against Focal's semantics is proved (by hand) [18], a point which brings a satisfactory confidence in the language's correctness. Then Rioboo [**?**] began the development of a huge computer algebra library, which offers full specification and implementation of usual algebraic structures up to multivariate polynomial rings with complex algorithms, first as a way to extensively test the language and (quite satisfactory) efficiency of the produced code and then to provide a standard library of mathematical backgrounds. And 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[15] 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[12, 13, 16], 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 [6, 7, 8]. This last work led to the development of a translator[5] from Focal to UML for documentation purposes.

The FoCaLize system

The FoCaLize development effort started in 2006: it was clearly 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 friendyness. The addition of powerful data structure definitions together with the corresponding pattern matching facility, lead to new expressing power.

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

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. Thus 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.

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 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.

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[10].

The FoCaLize system in short

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 written 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.

Chapter 1

Overview

Before entering the precise description of FoCaLize we give an informal presentation of near all its features, to help further reading of the reference manual. Every construction or feature of FoCaLize will be entirely 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 *method* introduced by the keyword `representation` gives the data representation of entities manipulated by the *species*. It is a type called the *representation* (or the representation type when emphasising on the fact that it is a type) and defined by a type expression. The *representation* may be not-yet-defined in a *species*, 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) plus *species representation types*, denoted by keyword `Self` inside the species and by the name of their *species* outside of them.

  Each *species* has a unique method *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* can simply be 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. It serves to announce a *method* to be defined later, i.e. to only specify its type, without implementation yet. Such *methods* are especially dedicated for specification or design purposes since declared names may

be used to define others *methods* while delaying their definition. The type provided by the *signature* allows FoCaLize to ensure via type-checking that the method is used in contexts compatibles 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 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. It will lead to a proof obligation later 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 names of *methods* known within the *species*'s context. For instance, a *property* telling that if the speed is non-null, then doors can't be opened could look like:

  `all v in Speed, v <> Speed!zero -> ~ doors_open`

  In the same way as *signatures*, even if no proof is yet given, the name of the *property* can be used to express other ones and its statement can be used as an hypothesis in proofs. 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 and ultimately checked with the theorem prover Coq.

  Like in any formal development, one severe difficulty before proving is obviously to state a true interesting and meaningful statement. For instance, claiming that a piece of software is "formally proved" as respecting the safety requirements `system_ok` "since **its** property is demonstrated" is a lie if this property was, for instance, `1 = 1 -> system_ok`. This is obviously a non-sense since the text of the property is trivial and does not link `system_ok` with the rest of the software (see [10] for less trivial examples).

We now make concrete these notions on an example we will 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) ;

end ;;
```

In this *species*, the *representation* is not explicitly given (no keyword `representation`), 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` and it is in fact a type variable. In the same way, we specify a *signature* for the equality (operator =). We introduce the three properties that an equality (equivalence relation) must conform to.

We complete the example by the definition of the function `different` which use the name = (here `basics#not_b` stands for the function `not_b`, the boolean `and` coming from 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 each implementation of = given further 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 and, it is always the last version which is 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 α, 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* and this abstraction is permanent.

Beware! *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* if all declarations have received definitions and all properties have received proofs.

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 methods/theorems are defined/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 to a complete implementation. On the previous example, forgetting `different`, we typically presented a kind of *species* for "specification" since it expressed only *signatures* of functions to be later implemented and properties to which, later, give *proofs*.

We can now create a new *species*, may be more complex, by **inheritance** of a previously defined. We say here "may be more complex" because it can add new operations and properties, but it can also only bring real definitions to *signatures* and *proofs* to *properties*, adding no new *method*.

Hence, in FoCaLize inheritance serves two kinds of evolutions. In the first case the evolution aims making a *species* with more operations but keeping those of its parents (or redefining some of them). In the second case, the *species* only tends to be closer to a "run-able" 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 ;;
```

We see here that we added new *methods* but 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*.

The *type* of a *species* built using inheritance is defined like for other *species*, the *methods* types retained inside it being those of the *methods* present in the *species* after inheritance is resolved.

A strong constraint in inheritance is that the type of inherited, and/or redefined *methods* must not change. This is required to ensure consistence 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.

1.4 Combining Bricks by Parametrisation

Until now we are only able to enrich *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 pair of setoids is a new structure, using the previous *species* as the "ingredient" to create the structure of the pair. Indeed, the structure of a pair is independent of the structure of each component it is made of. A pair can be seen as *parametrised* by its two components. Following this idea, FoCaLize allows two flavors of parametrisation.

1.4.1 Parametrisation by Collection Parameters

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 an interface I. The name C serves to call the *methods* of C which figure in I. C can be instantiated by an effective parameter CE of interface IE. CE is a collection and its interface IE must contain 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 *properties* 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 pair of setoids has two components, hence a *species* for pairs of setoids will have two *collection parameters*. It is itself a setoid, 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 =
  representation = (A * B) ;

  let ( = ) (x, y) =
     and_b
       (A!( = ) (first (x), first (y)),
        B!( = ) (scnd (x), scnd (y))) ;
  let create (x, y) in Self = basics#crp (x, y) ;
  let element = Self!create (A!element, B!element) ;

  proof of refl = by definition of ( = ) ;
end ;;
```

We express the *representation* of the product of two setoids as 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*.

Next, we add a definition for `=` of `Setoid_product`, relying on the methods `=` of `A` (`A!( = )`) and `B` (which are not yet defined). Similarly, we introduce a definition for `element` by building a pair, using the function `create` (which calls the predefined function `basics#crp`) and the methods `element` of respectively `A` and `B`. And we can 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.

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 inherits of `Setoid` and is parametrised by `Setoid`.

Why such *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 parametrisation 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 parametrised *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 parametrisation too weak to be usable. We choose to always hide the *representation* of a *collection parameter* to the parametrised hosting *species*. Hence the introduction of the notion of

collection, obtained by abstracting the representation from a complete species.

1.4.2 Parametrisation by Entity Parameters

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 such parameters *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 modulo : Self -> Self -> Self ;
end ;;
```

Note that `IntModel` can be later implemented in various ways, using Peano's integers, machine integers, arbitrary-precision arithmetic ...

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 1.2, a *species* needs to be fully defined to lead to executable code for its functions and checkable proofs for its theorems. When a *species* is fully defined, 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` turned 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* turns hidden. 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* expects that a *proof* that it **holds** will finally be given. For *theorems*, the *proof* is directly embedded in the *theorem*. 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 [**?**, 9] 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 body 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>9 qed
        by step <2>1
           property inf_is_associative, equal_transitive
           definition of order_inf
    <1>2 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 such "proofs"

since they bypass the formal verification of software's model. However, such a functionality remains needed since some of "well-known" properties can never be proved for a computer. For instance, $\forall x \in \mathbb{N}, x+1 > n$ does not hold in a computer with native integers. However, in a mathematical framework, this property holds and is needed to carry out other proofs. Thus the developer may prove either that all manipulated values remain in an interval where this property holds or may admit this property or may add code to detect overflow ... On another side, a development may be linked with external code, trusted or not, but for which properties cannot be proved inside the FoCaLize part since it does not belong to it. Expressing properties of the FoCaLize part may need to express properties on the imported code, that cannot be formally proved, then must be "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 development cycle of a FoCaLize program, the compiler keeps trace of dependencies between *species*, their *methods*, the *proofs*, ... to ensure that modifications of one of them will be detected those depending of it.

FoCaLize considers 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 erased 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 [21], 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* [15] 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 cases, it also serves as an oracle. When a test case fails, it means a counterexample of the property has been found: the implantation does not match the property; it can also indicate an error in the specification.

The tool *FocalTest* automatically produces the test environment and the drivers to conduct the tests. We benefit 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. *FocalTest* proposes a pure random test cases generation: it generates test cases until the precondition is satisfied, the verdict of the test case is given by executing the post-condition. 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 [14] 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 dependence 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 [5] 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.

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 suit 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.x.x.tgz
```

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 on the following.

2.5.1 Outputs

A FoCaLize development contains both "computational code" (i.e. code performing operations that lead 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 7 for details.

2.5.2 Compiling a source

Compiling a FoCaLize program involves several steps 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 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 7), 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

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 **??**). 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 (c.f **??**), 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 inherits 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 (c.f. 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 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* `implements` *name-species*

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 in Self = random_foc#random_int (42) ;
  let double (x in Self) = x + x ;
  let print (x in Self) = print_int (x) ;
end ;;

collection MyFull_Instance implements Full ;;

let v = Full.create_random ;;
Full.print (v) ;;
let dv = Full.double (v) ;;
Full.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 (c.f. 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 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** called `Elem` in the example:

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

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 (c.f 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 our 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 (c.f 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 in the example the species `MyCouple`) 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 ourselves (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 implements AddModN
  (ACollImplentingInts, ACollImplentingInts!from_int (42)) ;;
```

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

In this section, we 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 ordered. `OrderedIntCouple` has all the methods of `IntCouple`, except `create` which is redefined and the property `is_ordered`) stating 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 inherits (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 `inherits` clause. The inheriting species inherits of all the methods of inherited species. In case of a same name appears in several 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 inherits S1, S2 = ... end;;
species S4 inherits 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 must be identical, properties must be identical. If some of these methods have already received definitions, if they have the same type, then the definition which is retained for the inheriting species is the one coming from the rightmost defined parent in the `inherits` 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 one which is inherited.

```
species Foo inherits 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 ...) inherits 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 inherits (IntCouple) = ... ;;
```

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

```
species ListUnique (UElem is ...) inherits 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 (c.f. 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 (c.f 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 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 than for properties, the FoCaLize **system rejects theorems which 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 **??**). 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 inherits 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 inherits 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 (c.f. 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 **??**, 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, proofs written as direct Coq scripts will be addressed in 9.0.6. And 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.

$$\begin{array}{rcl} \textit{proof} & ::= & \{\textit{proof\_step}\}\ \textit{qed\_step} \\ & | & \textbf{by}\ \{\textit{fact}\}^{+} \\ & | & \textbf{conclude} \end{array}$$

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.

$$\textit{proof\_step} \quad ::= \quad \textit{proof\_bullet}\ \textit{statement}\ \textit{proof}$$

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 foo : A -> (B -> A)
proof =
  <1>1 assume h1: A,
       prove B -> A
    <2>1 assume h2: B,
         prove A
```

```
    by hypothesis h1
  <2>2 qed
    by step <2>1
<1>2 qed
  conclude
```

In this proof, 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, composed of steps `<2>1` and `<2>2`. These 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 5.1.1 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_bullet* **qed** *proof*
| *proof_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
```

statement ::= {**assume** *assumption* ,} [**prove** *logical_expr*]

A statement must be non-empty: at least one **assume** 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_expr* 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 ::= *ident* **in** *type_expr*
| *ident* : *logical_expr*

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

fact ::= **definitionof** [[*ident*] #] *ident* {, [[*ident*] #] *ident*}
| **hypothesis** *ident* {, *ident*}
| (**property** | **theorem**) [[[[*ident*] #] *ident*] !] *ident* {, [[[[*ident*] #] *ident*] !] *ident*}
| (**property** | **theorem**) {[[[[*ident*] #] *ident*] !] *ident*}$^{+}_{,}$
| **step** *proof_bullet* {, *proof_bullet*}

A fact used in a leaf proof can be a definition, a hypothesis, a property, a theorem, 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 *proof_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 5.1.1).

5.1.1 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.

Chapter 6

Recursive function definitions

In the current alpha-release, the logical counterpart of recursive functions is not completely handled (Coq code generation). We are still working on the point: recursive functions are planed to be fully supported as soon as possible, in addition with new material to help writing the required termination proofs.

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 Coqcompiler. 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.
- ".zv" files are compiled by Zenon via zvtov. The generated ".v" file is then compiled by Coq as describe above.
- ".fcl" files are compiled by focalizec, generating both the ".ml" OCaml source and the ".zv" pre-Coqsource. The ".ml" file is then sent to OCamland the ".zv" file is sent to Zenonto finally get a ".v" file that is sent to Coq.

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

* −**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 **??** 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 standard library directory first (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-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.

* −**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 passing 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-zenon**. When Coq code generation is activated, stops the compilation process before passing the generated file to Zenon. The produced file is then a pre-Coq source file, ended by the suffix ".zv". This option has no effect if −**no-coq-code** is 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.

* −**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** (c.f **??**) and that were kept during the compilation process. Moreover, these annotations can contain special tags used by the documentation generator of FoCaLize.

8.0.2 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.

8.0.2.1 @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.

8.0.2.2 @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.

8.0.2.3 @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

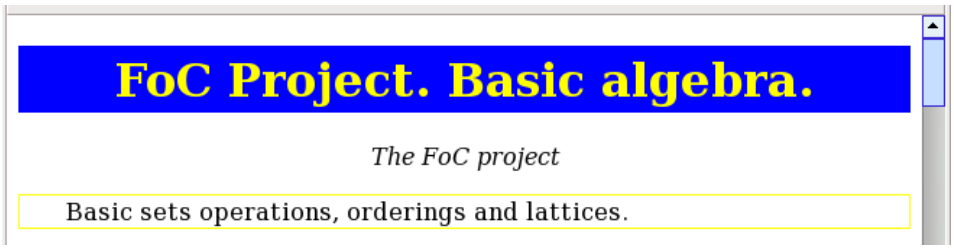
implements) and will appear in the header of the final document.

For example:

```
(***************************************************************************)
(*                          FoCaL 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.

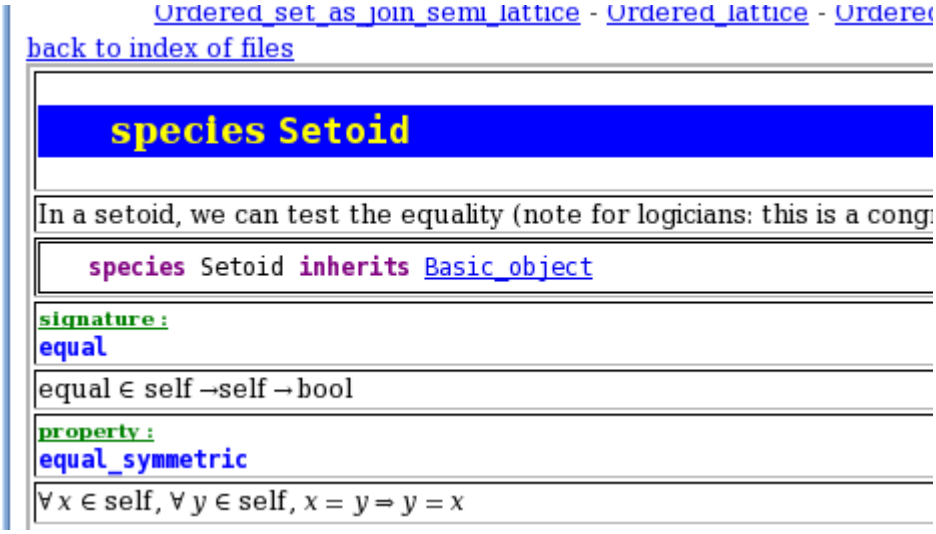
8.0.2.4 @mathml

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 inherits 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.



8.0.3 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.

8.0.3.1 XML to HTML

Transformation from ".fcd" to a format that can be read by a WEB browser is performed in two passes.

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/focalize`
   with the command xsltproc.
   For example:

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

8.0.4 XML to LaTeX

Currently not officially available.

Chapter 9

Hacking deeper

9.0.5 Interfacing FoCaLize with other languages

9.0.6 Dealing with hand-written Coq 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. FoCaL is cowardly and gives 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 `inherits` 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 $/\backslash$ properly.

Description: A logical expression contains a $/\backslash$ (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 $\backslash/$ properly.

Description: A logical expression contains a $\backslash/$ (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.

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 field 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).

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.

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. Method '$fname$' is not defined.

Description: A collection is built out of a completely defined species (c.f. 4.1.5), i.e. a species where **all** the methods are **defined** and not only declared. In the species $sname$, the method $mname$ is only declared, hence the species is not complete and no collection can be extracted from it.

Hints: Add an effective definition of the method, either by writing it code or by inheritance, according to your program model.

Species '$sname$' cannot be turned into a collection. Method '$fname$' does not have a termination proof.

Description: A collection is built out of a completely defined species (c.f. 4.1.5), i.e. a species where **all** the methods are **defined** and in particular proofs of properties are done. This also applies to recursive functions which must have a termination proof provided. The recursive function $fname$ of the species $sname$ doesn't have its termination proof.

This error message only arises if the `-impose-termination-proof` option is used on the command line. Otherwise, it is turned into a warning and the compiler will automatically generate an assumed proof.

Hints: Add an effective termination proof to the function or do not invoke the `-impose-termination-proof` option when compiling the source file.

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).

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 $file$.foc without the `--no-ocaml-code` option.

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 $file$.foc without the `--no-coq-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.

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 mutual recursive functions. At least the two 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

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, and R. Rioboo. Some hints for polynomials in the Foc project. In *Calculemus 2001 Proceedings*, June 2001.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] D. Doligez. Zenon, version 0.4.1. http://focal.inria.fr/zenon/, 2006.

[10] E.Jaeger and T.Hardin. A few remarks about developing secure systems in b. In IEEE, editor, *HASE 2008*, 2008. .

[11] T. Hardin and R. Rioboo. Les objets des mathématiques. *RSTI - L'objet*, 2004.

[12] M. Jaume and C. Morisset. A formal approach to implement access control. *Journal of Information Assurance and Security*, 2:137–148, 2006.

[13] 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.

[14] M. Maarek and V. Prevosto. Focdoc: The documentation system of foc. In *Proceedings of the 11th Calculemus Symposium*, Rome, sep 2003.

[15] 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.

[16] C. Morisset. *Sémantique des systèmes de contrôle d'accès*. PhD thesis, Université Pierre et Marie Curie - Paris 6, 2007.

[17] V. Prevosto. *Conception et Implantation du langage FoC pour le développement de logiciels certifiés*. PhD thesis, Université Paris 6, sep 2003.

[18] 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.

[19] V. Prevosto and D. Doligez. Algorithms and proof inheritance in the Foc language. *Journal of Automated Reasoning*, 29(3-4):337–363, dec 2002.

[20] V. Prevosto, D. Doligez, and T. Hardin. Algebraic structure and dependent records. In *TPHOLs'2002*, volume 2410 of *LNCS*. Springer-Verlag, 2002.

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

Index