Contents

Chapter 1

Changes in Virgile's PhD

1.0.1 Type unification (1)

Section 3.3, definition 9, page 27.
Rule [SELF1] should be: $mg(t, \mathbf{Self}, t) = \mathbf{Self}, id$
Rule [SELF2] should be: $mg(t, t, \mathbf{Self}) = \mathbf{Self}, id$

1.0.2 Type unification (2)

Is the "preference of `Self`" (see above) rule really needed ? It seems it can lead to absence of principal type. C.f. the section 13.

1.0.3 Normal form algorithm

Section 3.7.1, page 36.
In the algorithm, line 10 should be: $\mathbb{W}_1 \leftarrow ((\psi_{i_0} \ominus \phi), \mathbb{X})$.

In the algorithm, line 13 should be: $\mathbb{W}_2 \leftarrow (\mathbb{W}_2, \phi)$.

In the running text, page 37, line 8, the same modification must be done to get "on garde $\psi_{i_0} \ominus \phi$ dans $\mathbb{W}_1 \ldots$".

1.0.4 Typing rules for parametrised species

Section 3.8, figure 3.2, page 43.
Rule [COL-PRM] should be:

$$\frac{\mathcal{C}, \Omega \vdash e^S : a \qquad \mathcal{C} + C : \mathcal{A}(a, C), \Omega \vdash \mathbf{species}\ S(prms)\ \mathbf{inherits}\ e_1^S, \ldots e_{h_f}^S = \Phi_1 \ldots \Phi_n : t_S}{\mathcal{C}, \Omega \vdash \mathbf{species}\ S(C\ \mathbf{is}\ e^S, prms)\ \mathbf{inherits}\ e_1^S, \ldots e_{h_f}^S = \Phi_1 \ldots \Phi_n : (C\ \mathbf{is}\ a)t_S}$$

1.0.5 Dependency on the carrier

Section 3.9.4, definition 28, page 50.
The definition should be: "Soit une expression e, si une sous-expression de e ~~a le type~~ **fait référence à ("contient")** `Self`, il y a une decl-dépendance vis-à-vis du type support.".
In English: "Let's have e an expression, if a sub-expression of e ~~has type~~ **makes reference to ("contains")** `Self`, then there is a decl-dependency on the carrier".

This is more accurate since an expression having type `int` $\rightarrow$ `Self` does not have type `Self`, but when we state its type, `Self` occurs in the type and must be bound somewhere (and the decl-dependency on the carrier is just there for this purpose).

1.0.6 Dependencies in a species

Section 3.9.5, definition 31, page 53.
Lines 3 and 4 should be:

$$\forall j < n,\ y_j \in \wr\ y_i \int_S \cup \mathbb{I}\ y_{j+1}\ \mathbb{I}_S$$

$$x_1 <_S^{def} x_2 \mathrel{\widehat{=}} \exists \{y_i\}_{i=1\ldots n}\ tel\ que\ y_1 \circlearrowleft_S x_1, y_n \circlearrowleft_S x_2, \forall j < n,\ y_j \in \mathbb{I}\ \ y_{j+1}\ \mathbb{I}_S$$

1.0.7 Parameters used by a method

Section 3.9.5, definition 66, page 124.
To understand the rules `[BODY]`, `[TYPE]`, `[DEF-DEP]`, `[UNIVERS]` and `[PRM]`, it should be stated that implicitly the parameter $C_{p'}$ has the form:

$$C_{p'}\,\mathtt{is/in}\,\tau_{p'}$$

.

1.0.8 Instanciation of species parameters

Section 3.9.5, definition 67, page 124.
Second rule should be:

$$\frac{x \upharpoonleft S = S_h \qquad \mathcal{E}(S_h) = (C_1 \bullet \tau_1, \ldots, C_{p_f} \bullet \tau_{p_f}) \qquad l_h = e_1 \ldots\ e_{p_f}}{\mathtt{Inst}_S(x) = \{\mathtt{Inst}_{C_p}(e_p)_{C_p \in \mathcal{U}_{S_h}(x)}\}}$$

1.0.9 Translation example in OCaml

Section 8.2, page 152.

The code sample shown and the explanation about `create` at the top of the page is wrong or at least not complete. In effect, if presented this way, we don't know from where `create` comes. In fact, we must used the one coming from the species we "implement". So a qualified notation (i.e. module name + function name) is required.

1.0.10 Dependencies of a method

Section 8.3.1, definition 72, page 153.

Missing notion of order

All along the rules, dependencies are stated as a set of names. In fact this is incomplete since there can be dependencies between these names for a collection parameter. So they must be ordered according to their own dependencies (i.e. according to def-dependencies inside the hosting species that is that collection parameter).
For instance:

```
species S1 ... =
  let eq = ... ;
  theorem th1 : all x in ..., !eq (...) ...
    proof = ... ;
end ;;

species S2 (P is S1) ... =
  theorem th2 : all x in P, ...
    proof = ... property P!th1 ... ;
end ;;
```

In `S1`, `th1` decl-depends on `eq`. In `S2`, method `th2` has a dependencies on its collection parameter `P` methods. Especially, on `P!th1`, and completions rules require to add `P!eq` (in order to express the "type" of `th1`, i.e. its statement). Since in `P`, `th1` decl-depends on `eq`, when making λ-liftings to abstract `P`'s methods in the dependencies on methods of parameters in `S1`, we must ensure that `eq` is λ-filter before `th1` otherwise `eq` will be unbound in `th1`.

This order is given be the dependencies of the methods inside the species used as collection parameter.

Missing rule

The rules [DEF-DEP], [UNIVERS] and [PRM] can add dependencies on parameters after rule [BODY] and [TYPES]. However, the added methods can have decl-dependencies via their "types". An this is not taken into account by the current set of rules. To circumvent, a new rule is added, [DIDOU] (better name to be found, but the day I thought to this rule, I was very poor in naming schemes ☺). This rule intuitively takes the all dependencies found by [BODY], [TYPES], [DEF-DEP], [UNIVERS] and [PRM] as initial set and performs a fixpoint by adding for each method of dependencies, its decl-dependencies coming from its "type" (i.e. ML-like type for computational methods, and statement for logical methods). Of course, when tracking decl-dependencies of a method of parameter, we address the method type in its species. But this species is a collection parameter. So before adding the found method to the set of dependencies on collection parameters of the analysed species, we must replace in the method, occurrences of `Self` by the by the species parameter from where this method comes.

Missing substitution in rule [PRM]

First, the rule `[PRM]` (page 153, definition 73) needs further explanations to understand its presentation. It must first be understood that in this rule, the species `S` has the following form:

`species` $S(C_p$ `is` $\ldots, C_{p'}$ `is` $S'(C_p))$

Moreover, implicitly $i_{p'}$ is the interface of $C_{p'}$. And C_p is a valid implementation of the parameter C'_k (having the interface i'_k) of the species S'.

Now, the rule says:

$$\frac{z \in Deps(S, C_{p'})[x] \qquad i_{p'} = S'(e_1, \ldots, C_p, \ldots) \qquad \mathcal{E}(S') = (C'_1 \texttt{.} i'_1, \ldots, C'_k \texttt{ is } i'_k, \ldots) \qquad y \in Deps(S', C'_k)[z]}{y \in Deps(S, C_p)[x]}$$

This rule forget to show that we must instantiate the formal parameter of `S'` by the effective argument provided. In effect, in the bodies/types of the methods of `S'` (those methods the conclusion adds to the currently computed dependencies set), parameters are those of `S'`, not our current ones we use to instantiate the formal ones of `S'` ! To prevent those of `S'` to remain in the expressions and be unbound, we do the instanciation here.

Inconsistency between inherited/re-computed dependencies

When computing dependencies on collection parameters of a method, it is never clearly stated about how to compute them when the method is inherited.

One way is to compute from scratch the dependencies from the body of the inherited method. The second is to recover the inherited dependencies and to perform substitution of formal parameters of the inherited species by effective arguments used in the `inherits` species expression.

This last process is in fact very difficult due to the amount of information recorded in the parameters/methods descriptions (moreover, making severe usage of sharing).

So we really prefer to use the first method that naturally create the data-structures and information to record. The only problem is that during inheritance, some dependencies present in the original inherited species may "disappear" due to parameters instantiations.

For example:

```
species Simple =
   signature equal : Self -> Self -> bool ;
end ;;

species Couple (S is Simple, T is Simple) =
  signature morph: S -> T;
  let equiv(e1, e2) = T!equal(!morph(e1), !morph(e2));
end ;;

species Bug (G is Simple) inherits Couple (G, G) =
  theorem theo: true
    proof =
```

```
      <1>1
         prove true
         <2>f qed assumed {* *}
      <1>2 qed by definition of equiv ;
end ;;
```

In fact, the problem is that when we compute dependencies on collection parameters, in the case where we inherit from a species having 2 parameters instantiated by the same argument, we get into a fusion of the methods we depend on.

But, the method generator coming from the originally inherited method expects to be applied to as many arguments that *lambda*-lifting were created.

In the above example, `S` and `T` are instantiated both by `G`. So we get 1 dependency on `G!rep` (the carrier) and 1 on `G!equal` although the method generator expects 3 arguments : twice `G!rep` and one `G!equal` because in `Couple` was abstracted on the carrier of `S`, the carrier of `T` and `T!equal`. Because we work with sets to represent dependencies, twice `G!rep` is ... 1 `G!rep`. And same thing for `G!equal`.

We described the problem here via a dependency on the carrier, but its is the same thing with dependencies on other methods: we would just need to make `equiv` depending for instance on `S!equal` and `S!equal`.

The solution is to make a mix between the two initial solutions. First, we compute the abstractions due to dependencies on collection parameters in the body of the method once inherited. This way, we naturally let data-structures be created and sharing what they need. In fact, this gives us a "skeleton" of dependencies where some λ-liftings may have disappeared compared to the number of required in the inherited species. But, we know that all the methods involved in the dependencies are present, may be not with the right number of occurrences. Then we take the dependencies scheme of the inherited method and we rebuild a final dependencies structure by replacing in the inherited one all the occurrences of dependencies on the formal parameter by the corresponding effective argument (used during instanciation) ones. This way, we just "remap" the computed dependencies on the inherited ones' scheme.

In the example above, this means that we compute in `Bug` that we have dependencies on `G!rep` and `G!equal`. We look back in the inherited `Couple` dependencies, we find `S!rep`, `S!rep` and `T!equal`. So we construct the final dependencies as `S!rep[S<-G] S!rep[S<-G]` and `T!equal[T<-G]`

Dependencies on collection parameters for record type

Computing real dependencies on collection parameters for record type is not clearly stated. For OCaml, it is quite trivial since we don't have any logical methods, hence we can only have dependencies via "types". For Coq, the situation is more tricky. In fact, due to logical methods (theorems and properties), dependencies may be more complex, involving types and methods found in the expressions forming the logical statements.

The same kind of problems arises than above (missing rule): an extra rule is needed (the [DIDOU] rule), but the only difference is that since the record type only shows "types", the initial set of dependencies to close is the one obtained by the rule [TYPE].

It is not yet formally clear that computing dependencies required by the record type is complete. Experiment seems to show that yes, but further theorical investigations should be performed.

Taking a wider set of dependencies (for instance the same than those computed method per method, all grouped in a single big union) would lead to extra arguments (by λ-lifting) to the record type that would not be used. This is unwanted for 2 reasons: efficiency/readability of the generated code, and more importantly, the risk to have variables not bound to a type, infered as polymorphics, and for which Coq would say that it "can't infer a type for this placeholder".

Dependencies on collection parameters for collection generator

For the λ-liftings to do for the collection generator, only taking into account remapped dependencies is wrong in case where some dependencies really present in the inheriting species are not mapped onto inherited ones. This is the bug n°13.

```
species Comparable =
  signature one : Self ;
end ;;
```

```
species Compared (V is Comparable, minv in V) =
  representation = V ;
  let my_param = minv ;
end ;;

species Buggy (VV is Comparable) =
  inherit Compared (VV, VV!one) ;
end ;;
```

In effect, `Buggy!my_param` has a dep on `VV!one` (its own param). but in the inherited, `Compared!my_param` instead has a dep on its entity parameter `minv`, not on its collection param `V`. Hence, during the "remapping", the dependency in `Buggy` is dropped and the collection generator is missing a λ-lifting:

```
  (* Fully defined 'Buggy' species's collection generator. *)
  let collection_create () =
    (* From species bugggg_avec_damien#Compared. *)
    let local_my_param = Compared.my_param (_p_VV_one) in
    { rf_my_param = local_my_param ;
      }
```

In fact, the remapping process is correct since it really lead to the correct set of arguments to pass to the inherited method generator. What is wrong is that to compute the arguments of the ***collection generator*** we make a *"big union"* of the extra arguments of the ***method generators***. But, in our case, these extra arguments of the ***method generators*** are correct, simply in one of them, there is some stuff coming from the collection parameters (in the expression used to instantiate the in-param) and this stuff is **NOT** a method on which the inherited method generator depends on. So, the stuff to λ-lift in the ***collection generator*** should be computed not on the mapped dependencies but on the **NON**-mapped dependencies ! This should be the *"big union"* of non-remapped dependencies.

In addition to the fact that λ-liftings to generate for collection generator must use *non*-remapped deps, the rule `[PRM]` must be applied to deps on params found from rule `[TYPE]` otherwise one has too many (and useless) λ-liftings. This last point caused some extra argument for which Coq didn't succeed in finding the type. Note that there still have examples where it arises, but in this case arguments are usefull (see bug n° 15 in the tracker).

1.0.11 Coq code generation model

The problem

The initial Coq code generation model appeared to have a strong weakness. Moreover, it was strongly different from the OCaml one. Let's understand the weakness on a simple example... The problem is:

```
species IntModel inherits Basic_object =
  representation = basics#int ;
  let one in Self = 1 ;
  let modulo (a, b) = if true then a else (if false then b else one) ;
end ;;

species Me (Naturals is IntModel, n in Naturals) =
  representation = Naturals ;

  theorem lookatme : all x in Self, basics#base_eq (n, Naturals!one)
    proof : assumed (* *) ;

  let reduce (x in Naturals) in Self = Naturals!modulo (x, n) ;
end ;;
```

The generated Coq code follows (at least, for the interesting part of our problem, i.e. the `lookatme` theorem):

```
Chapter Me.
  Record Me (Naturals_T : Set) (_p_n_n : Naturals_T)
            (_p_Naturals_one : Naturals_T) : Type :=
    mk_Me {
      Me_T :> Set ;
```

```
      (* From species ok__in_example#Me. *)
      Me_lookatme :
        forall x : Me_T, Is_true ((basics.base_eq _ _p_n_n _p_Naturals_one)) ;
      (* From species ok__in_example#Me. *)
      Me_reduce : Naturals_T -> Me_T
      }.

  (* Variable abstracting the species parameter [Naturals]. *)
  Variable Naturals_T : Set.
  (* Variable abstracting the species parameter [n]. *)
  Variable n_n : Naturals_T.

  (* Carrier representation. *)
  Let self_T : Set := Naturals_T.

  (* Variable(s) induced by dependencies on methods from species
     parameter(s). *)
  Variable Naturals_modulo : Naturals_T -> Naturals_T -> Naturals_T.
  Variable Naturals_one : Naturals_T.


  (* From species ok__in_example#Me. *)
  Section lookatme.
    (* Due to a decl-dependency on species parameter carrier type 'Naturals'. *)
    Variable _p_Naturals_T : Set.
    (* Due to a decl-dependency on method 'one' of species parameter'Naturals'. *)
    Variable _p_Naturals_one : _p_Naturals_T.
    (* Due to a decl-dependency on method 'n' of species parameter 'n'. *)
    Variable _p_n_n : _p_Naturals_T.
    Theorem Me__lookatme :
      forall x : self_T, Is_true ((basics.base_eq _ _p_n_n _p_Naturals_one)).
    (* Artificial use of type 'Naturals_T' to ensure abstraction of it's
    related variable in the theorem section. *)
    assert (___force_abstraction_p_Naturals_T := _p_Naturals_T).
    (* Artificial use of method '_p_Naturals_one' to ensure abstraction of
    it's related variable in the theorem section. *)
    assert (___force_abstraction__p_Naturals_one := _p_Naturals_one).
    (* Artificial use of method '_p_n_n' to ensure abstraction of it's
    related variable in the theorem section. *)
    assert (___force_abstraction__p_n_n := _p_n_n).
    apply basics.magic_prove.
    Qed.
    End lookatme.

  Let self_lookatme :
    forall x : self_T, Is_true ((basics.base_eq _ n_n Naturals_one)) :=
    Me__lookatme Naturals_T Naturals_one n_n.

  ...
  End Me.
```

We see that there is a `Section` for the species `Me`. The theorem `lookatme` is in a nested `Section` (`Chapter` is a synonym for `Section` in Coq).

It depends on `one` coming from `Natural` (which is a collection parameter), and on `n` which is an entity parameter.

Hence, naturally, we will need to abstract these 3 things (`one`, `n` and the type `Natural`). This really performed by the 3 `Variables` at the beginning of the `Section lookatme`. Hence, looking at the type of the **theorem generator** `Me__lookatme` **inside** the `Section lookatme`, we see that is has type:

```
Me__lookatme:
  self_T -> Is_true (basics.base_eq _p_Naturals_T _p_n_n _p_Naturals_one)
```

since the `Section` is not closed, because the `Variables` are not yet abstracted by the Coq's `Section` mechanism.

We now close the `Section`, then naturally the **theorem generator** `Me__lookatme` turns having the type:

```
Me__lookatme :
  forall (_p_Naturals_T : Set) (_p_Naturals_one _p_n_n : _p_Naturals_T),
    self_T -> Is_true (basics.base_eq _p_Naturals_T _p_n_n _p_Naturals_one)
```

Great ! the `Variables` have been abstracted for us by Coq. Now, the idea is that this generator will be used to create a collection like:

```
collection ConcreteInt implements IntModel ;;
collection ConcreteMe implements Me (ConcreteInt, ConcreteInt!one) ;;
```

Let's continue and close the `Chapter` (≃ a `Section`) of species `Me`. And then, here the theorem generator `Me__lookatme` turns unfortunately to have type:

```
Me__lookatme:
  forall (Naturals_T _p_Naturals_T : Set)
    (_p_Naturals_one _p_n_n : _p_Naturals_T),
      Naturals_T ->
        Is_true (basics.base_eq _p_Naturals_T _p_n_n _p_Naturals_one)
```

We see that we have an extra `Naturals_T : Set` in
forall (*Naturals_T* `_p_Naturals_T : Set`) ...
So now, we have 2 arguments of type `Set`. Why ?

Let's have a look in the outer `Section` related to the species `Me`. We had a `Variable` already abstracting the type of the collection parameter `Natural`, namely **Variable** `Natural_T : Set`. And so, by closing the outer `Section` of the species `Me`, Coq abstracted once again. Then, it is the outer `Section` that brings the problem, not the inner one.

And obviously, this typing problem makes so that when we try to create collection like above, we do not apply the method generator to the number of arguments that Coq expects.

A solution could be to say "let's remove the `Variable Natural_T : Set` from the outer `Section`". Right, no ! Since we do not λ-lift in properties (generated as `Hypothesis` in Coq), if a FoCaLize property needs to make reference to this type, we need a way to speak of if. And if we remove it, then we are not able anymore to speak of it...

A few remarks

- The annoying `Section` is then the outer one, the one of the species. Indeed, when we leave the inner `Section`, the theorem gets right abstracted on the inner `Variable`.
- This only arises on theorems because they are the only ones to be in a nested `Section`.
- Moreover, we can note in this code generation model that the `Local self_xxx` are always generated even when the method `xxx` is inherited. The reason is that since we do not λ-lift in properties (leading to Coq `Hypothesis`), if a property depends on another method, it really need to have a way to speak of it in the property's statement. And namely, that's via this `self_xxx`. However, in the OCaml code generation model, inherited methods are not generated again, hence do not lead to `local_xxx` definitions. This is not really homogeneous.

Conclusion

In order to solve this problem and to make Coq and OCaml code generation models (hence Coq and OCaml generated codes), we decided to use the same model, without `Sections` and `Chapter`'s, and to manage abstractions ourselves via explicit λ-liftings even for properties and theorems.

This raises a technical problem however since Zenon does not support yet higher order. And in fact, with this generation model, out theorems get parametrised by all the λ-lifted definitions. To circumvent this problem, we decided to only reintroduce `Sections` in the code dedicated to be sent to Zenon, then map back the proved theorems onto regular λ-lifted definitions. This means that on Zenon's the point of view, there is no λ-lifts, all is first order,

and it can exhibit a proof. Once we get the proof done, we apply it to the stated theorem in the `Section` to get a temporary version of the theorem. And finally, after closing the `Section` for Zenon, we transform this temporary version into a fully λ-lifted definition.

Chapter 2

Code generation model

The code generation model is the now closely the same for both OCaml and Coq generated source files. The main difference comes from the fact that in OCaml, logical methods are discarded. However, except for the case of Zenon proofs where we introduce Coq `Section`, the generation model for "computation" and logical methods are exactly the the same. This especially means that dependencies are "manually" abstracted by explicit λ-lifting instead of using (like in the previous compiler) the Coq's `Section` mechanism. In the same order of idea, in order to have a common model, both OCaml and Coq code are based on a record-oriented structure for the species and collection, with explicit record fields accesses.

2.1 Species

2.1.1 Species header

The generated code for a species is hosted by a module whose name is the species' name. This way, it is possible to have species having the same names of methods without conflict. The module hence defines the name-space of "things" contained in and induced by a species.

2.1.2 Carrier representation

In OCaml, if the structure of the carrier of the species is known (i.e. if the method `representation` was defined), then we generate a type definition whose name is `me_as_carrier` and body is the type translation of the corresponding FoCaLize type expression.

In such a type definition, carriers of species parameters appearing in the species' carrier are abstracted by type variables. The naming scheme of these variable is "'" + the species parameter's name un-capitalised + "`_as_carrier`" + an integer stamp that is unique (inside this type definition). Since species names in FoCaLize are capitalised and capitalised identifiers in OCaml are reserved for modules and sum type value constructors, we need to un-capitalise the name when generating OCaml code. For the stamp, it is required to prevent several type variable from having the same name in case the species is parameterised by both a collection parameter and an entity parameter whose names differ only by the capitalisation of the first letter.

For instance, the following header of a species definition:

```
species Cartesian_product(A is Setoid, B is Setoid) =
  representation = A * B;
  ...
```

will generate the OCaml type definition:

```
module Cartesian_product =
  struct
  (* Carrier's structure explicitly given by "rep". *)
```

```
type ('a0_as_carrier, 'b1_as_carrier) me_as_carrier =
  'a0_as_carrier * 'b1_as_carrier
...
```

In fact, *a posteriori*, I think now that it's useless in case of a species, even closed. This type definition is only used in the case of a collection.

In Coq, no definition generated, the knowledge of the structure of the carrier being reflected directly is needed in the methods (see later).

2.1.3 The record type

The type of data representing a species is a record type. We first examine its header, i.e. stuff before this type definition's body, then it's body.

The record header

The name of this type is always `me_as_species`. It can be parametrised due to various abstraction requirements induced by the late-binding feature of FoCaLize. OCaml requires a parameter not needed for Coq: this is the only fundamental difference. We will see that abstractions (hence, parameters) required by Coq can involve methods but this is only because some of the dependencies that are present in the Coq code are always trivially absent in the OCaml code.

1. First come all the species parameter carriers appearing in the **types** of the methods of the species. Each carrier will be abstracted by one type variable. This allows to "late-bind" the `representation` of the species parameters. We say here "appearing in the **types**...": be aware that the type of "computational" methods are ML-like types and the ones of logical methods are their **statement** ! We do not explain here how these parameters are found: this is the role of dependency computation on species parameters.

   The naming scheme of these type variable in OCaml is the same than described above for the carrier representation.

   In Coq these parameters of the type are not "really type variables" but arguments of type `Set` (simply a technical question). Their naming scheme is the species parameter's name + "`_T`".

2. Next, **only in the** OCaml **code**, the record type is always parametrised by a type variable representing the carrier of (i.e. the internal representation of the type encapsulated in this) species. By convention, this type variable is always named `'me_as_carrier` (don't confuse with `me_as_carrier` that is the name of the type definition representing the effective structure of the carrier when it is known).

   In Coq we don't have this mandatory type parameter, but instead of it, we will have one extra field in the body of the record. This variable enable to "late-bind" the `representation` of the species.

3. Finally come all the entity parameters then the methods of the species parameter appearing in the **types** of the methods of the species. Since in OCaml the type of a method can only involve type constructors (a ML-like type), it is clear that we won't have any such parameters. In effect, in OCaml the logical methods are discarded. However, since for the type of a logical method is its logical statement, to Coq's side, we can have any expression inside their type. In particular, we can have calls to some collection parameters' methods. Having these parameters in the record type allows the "late-binding" on the collection parameter itself (i.e. on by which effective collection will be used to instantiate the parameter). The naming scheme for the parameters induced by these dependencies is "`_p_`" + the species parameter's name + _ + the method's name.

   There is a special a case: the entity parameters. We will explain this after the following example.

Let's now change our previous example to illustrate the header of our record type in OCaml and Coq:

```
species Cartesian_product(A is Setoid, B is Setoid) =
  representation = A * B;
  let make (x in A, y in B) in Self = (x, y) ;
  let equiv (x in Self) = ... A!equal (...) && B!equal (...) ;
  theorem thm : all x in A, A!tst (x) -> ...
    proof = ... by property A!commutes ... ;
  ...
```

We can see that we have:

- the carrier defined,
- a method `make` of type `A -> B -> Self`
- a method `equiv` having type `Self -> bool`, having dependencies on the methods `equal` of the collection parameters `A` and `B`, but in its **body**, not in its type (`equal` doesn't appear in `equiv`'s type). Hence, the record type (in Coq and in OCaml) won't have any parameter to abstract the dependencies on these methods.
- a theorem `thm` of "type" `all x in A, A!tst (x) -> ...` and "body" (proof) `... by property A!commutes ...`. Hence, it has dependencies on `A!tst` in its type and on `A!commutes` in its body. This means that in Coq the record type will have a parameter to abstract the dependency found in the type of the theorem, i.e. `A!tst` but not for `A!commutes`

The record type for OCaml will then look like:

```
module Cartesian_product =
  struct
  (* Carrier's structure explicitly given by "rep". *)
  type ('a0_as_carrier, 'b1_as_carrier) me_as_carrier =
    'a0_as_carrier * 'b1_as_carrier
  type ('a0_as_carrier, 'b1_as_carrier, 'me_as_carrier) me_as_species =
  ...
```

In Coq a record is introduced by a constructor. By convention, we always name it `mk_record`. The record type for Coq will then look like:

```
Module Cartesian_product.
  Record Cartesian_product (A_T : Set) (B_T : Set)
    (_p_A_tst : A_T -> basics.bool__t) : Type :=
    mk_record
  ...
```

The entity parameters In OCaml they can never appear in the record type since in ML-like types, we can't have expressions (however, their type can appear – imagine simply a method returning the entity parameter). However, in Coq it is possible to have dependency on an entity parameter in a theorem or property statement (i.e. in the type of a logical method). For example:

```
species Me (Naturals is IntModel, n in Naturals) =
  representation = Naturals ;

 theorem myth : all x in Self,
    basics#syntactic_equal (n, Naturals!un)
  proof = assumed (* *) ;
  end ;;
```

The theorem `myth` shows a dependency on the entity parameter `n`. In this case, the record type will be parametrised by this entity parameter like if it was a collection parameter's method. Obviously, an entity parameter doesn't have methods since it is a "value" and not a species. So we don't have any notion of method in the naming scheme. We choose to name these entity parameters by "_p_" + the entity parameter's name + _ + the the entity parameter's name again.

```
Module Me.
  Record Me (Naturals_T : Set) (_p_n_n : Naturals_T)
    (_p_Naturals_un : Naturals_T) : Type :=
    mk_record
  ...
```

The record type for OCaml will then look like:

```
module Cartesian_product =
  struct
  (* Carrier's structure explicitly given by "rep". *)
  type ('a0_as_carrier, 'b1_as_carrier) me_as_carrier =
    'a0_as_carrier * 'b1_as_carrier
  type ('a0_as_carrier, 'b1_as_carrier, 'me_as_carrier) me_as_species =
  ...
```

The record type for Coq will then look like:

```
Module Cartesian_product.
  Record Cartesian_product (A_T : Set) (B_T : Set)
    (_p_A_tst : A_T -> basics.bool__t) : Type :=
    mk_record
  ...
```

Fields and their types

Now we saw the header of the record type definition, we must address its body, i.e. its fields. Roughly speaking, the fields will be all the methods with their types hosted in the species in normal form. By "all" we mean the methods declared, defined in the species and those inherited. Because the species is in normal form, this means that we do not have several times a method: inheritance has been resolved and chose the right version of each method to keep.

The type accompanying each method is, like we previously said, a ML-like type for "computational" methods and a logical statement for logical methods. This especially means that since logical methods are discarded in OCaml, in this target language, we will only have ML-like types.

The only important difference between OCaml and Coq is that in Coq we always have an extra (and first) field representing the carrier of the species (remember that in OCaml, instead, we had a type definition that we didn't have in Coq). This field always appears as `rf_T :> Set ;` and represents the type encapsulated in the species. In OCaml, the field corresponding to a method is straight the method's name. In Coq, the field's name is "`rf_`" + the method's name ("rf" for **r**ecord **f**ield).

Let's now take a simple example and see the record types in OCaml and in Coq.

```
species Me (Naturals is IntModel, n in Naturals) =
  representation = Naturals ;

 theorem daube : all x in Self,
    basics#syntactic_equal (n, Naturals!un)
  proof = assumed (* *) ;

  let junk (x in Self) in int = 1 ;

  let reduce (x in Naturals) in Self =
    Naturals!modulo (x, n) ;
end ;;
```

```
module Me =
  struct
  (* Carrier's structure explicitly given by "rep". *)
  type ('naturals0_as_carrier, 'n1_as_carrier) me_as_carrier =
    'naturals0_as_carrier
  type ('naturals0_as_carrier, 'n1_as_carrier, 'me_as_carrier) me_as_species = {
    (* From species ok__in_example#Me. *)
    junk : 'me_as_carrier -> Basics._focty_int ;
    (* From species ok__in_example#Me. *)
```

```
    reduce : 'naturals0_as_carrier -> 'me_as_carrier ;
    }
```

```
Module Me.
  Record Me (Naturals_T : Set) (_p_n_n : Naturals_T)
    (_p_Naturals_un : Naturals_T) : Type :=
  mk_record {
    rf_T :> Set ;
    (* From species ok__in_example#Me. *)
    rf_daube :
      forall x : rf_T,
        Is_true ((basics.syntactic_equal _ _p_n_n _p_Naturals_un)) ;
    (* From species ok__in_example#Me. *)
    rf_junk : rf_T -> basics.int__t ;
    (* From species ok__in_example#Me. *)
    rf_reduce : Naturals_T -> rf_T
    }.
```

Note that in the generated Coq code, the method `daube` (in fact, the theorem) contains an application of `basics.syntactic_equal`. We can see the mechanism of explicit polymorphism and the interest of having kept in the Coq code generation environment the number of extra arguments (_s) that must be added to identifiers in applicative position.

2.1.4 Methods

Once the record type is defined, it is time to generate the definitions corresponding to the various methods of the species. Several cases exist: a method can be declared, defined and in each case either at the current inheritance level or inherited from an ancestor.

Inherited, declared, defined ?

First of all, the point is that methods only declared or **inherited** are **never** leading to generated code, neither in OCaml, nor in Coq. This means that only methods freshly defined in the species are leading to code.

Defined methods

In term of generation model, there is no difference between OCaml and Coq. The point is because in OCaml we don't have logical methods, all what we will explain about theorems and proof is trivially out of the subject for OCaml. Hence we won't make any difference in our explanation here, simply considering both kinds of methods independently of the target language.

There are 2 kinds of methods: "computational" and logical. The generation model makes only a difference for theorems because they have a proof, but the final definition of both kinds of method uses the same mechanism: making explicit abstractions (λ-lifts) for all the types and **declared** methods (from `Self` or from the species parameters) the defined method depends on. Moreover, an entity parameter (since it's in fact a value) will appear as an argument of a method if it is used by it: it becomes hence an argument of the method.

To help us, we need 3 notions. We don't examine here how they are computed. This will be investigated later and is mostly described in Virgile Prevosto's PhD.

- The carriers present in the methods from parameters (and in the type of entity parameters) and methods of `Self` the method depends on. Since they are atomic types, there is no ordering issue in this set.
- The "minimal typing environment". It represents the **ordered** set of methods of `Self` a method depends on. It must be ordered because some of the dependencies can depend on some others. Because of the well-formation property, we are sure this order exists.
- The **ordered** set of methods from parameters the method depends on.

A method will lead to a `let` of `theorem` definition depending on its kind. The generated name is the same than in the FoCaLize source. Only a "stringification" is done when the method is an operator (e.g. `=`, `+`, `+=e`, ...). This stringification is done on the fly using a very simple mechanism (check function `pp_vname_with_operators_expanded` in the source file `focalize/focalizec/src/basement/parsetree_utils.ml`.

The generated definition is in fact a **method generator**, not the method itself. Its is a function that is parametrised by all the things the method depends on, and whose body is the method's body. This mechanism serves the late-binding feature and allows to really create a method once things it depends on are defined by applying the generator to the effective definitions of the methods the current method depends on. Hence, until the methods we depend on are not yet defined, one can still work with our method generator. Moreover, this allows to make several effective methods from a same generator, by applying different effective definitions for λ-lifted parameters of the generator.

Next come all the λ-lifts that represent the dependencies of the method: In the following order come:

1. The parameters representing the carriers of species parameters appearing in the method (i.e. used in its whole definition). Their name is "`_p_`" + the species parameter's name + "`_T`.

2. The parameters representing the methods of species' parameters the current method depends on. They are ordered in 2 directions. First, all the methods of a same parameter are consecutive and we λ-lift following the order of apparition of the species parameters. Second, for each species parameter, the consecutive list of its methods is ordered according to their own dependencies together.

   The first point naturally ensure that following the species parameters' order, definitions of methods of a parameter can only depend on former parameters (otherwise, scoping and type-checking would have told "Unbound ..."). The second point is not obtained for free. We must really order the methods of a species parameter according to their own dependencies "on Self's methods" in their hosting species. For instance, let's imagine that in a species, we need to abstract `P1!eq_refl : all x in Self, !equal (x, x)` and `P1!equal : Self -> Self -> bool`, clearly to have `P1!eq_refl` well typed, `P1!equal` must be known, hence appear sooner (i.e. must be λ-lifted before `P1!eq_refl`).

   Methods are named by "`_p_`" + the species parameter's name + "`_`" + the method's name.

   The translation mechanisms of expressions is not studied in this section since it isn't really part of the "species" compilation model. We can however note that in the body of a species parameter's method, calls performed to other methods of **this** parameter (i.e. so, obviously, on which the method of the parameter depends on) are done using the naming scheme: "`_p_`"+ the species parameter's name + "`_`" + called method's name (flag `SMS_from_param` used when calling the function `Species_record_type_generation.generate_logical_expr`).

3. The parameters representing the methods of ourselves (i.e. of `Self`) we **decl**-depend on. They are named by "`abst_`" + the method's name. The only exception is in case where the method is the `representation`, the name will be `abst_T`.

   Methods on which we **def**-depend are not abstracted (i.e. not represented by λ-lifting). In effect, since we depend on their **def**inition, they are defined (the compiler ensures that) and their effective definition must be used in the body of the method that depends on. Otherwise, there would be no link between the fact the method depends on a definition, and a λ-lifting would represent any definition that will be provided one day, nobody know when and where ! This deals with the first sentence of the last paragraph of page 116 (below definition 58) in Virgile Prevosto's PhD.

Finally comes the translation of the method definition itself, i.e. it's parameters (if the definition is functional) and its body. As above, we leave for later the translation mechanisms of expressions. We can however note that in the body of a method, calls performed to other methods of `Self` (i.e. so obviously on which we depend) are done using the naming scheme: "`abst_`"+ the called method's name (flag `SMS_abstracted` used when calling the function `Species_record_type_generation.generate_logical_expr`).

Attention, theorems require some intermediate cooking before one can directly build their method generator. In effect, their proof may involve a script for Zenon and in this case, a complex process must be inserted in order to get the proof done to finally get the method generator. This will be explained in section **??**odo1.

Sample code to help to summarize

We take a part of the example given in Virgile Prevosto's Phd, section 2.2.2 starting page 14 to illustrate the model we exposed until now. Attention, we explicitly skipped (removed in the generated listings) code dealing with collection generator we will explain in the next section.

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

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

species Monoide inherits Setoide =
  signature ( * ) : Self -> Self -> Self ;
  signature un : Self ;
  let element = Self! un * !un ;
end ;;

species Setoide_produit (A is Setoide, B is Setoide) inherits Setoide =
  representation = (A * B) ;

  let ( = ) (x, y) =
     and_b
       (A!( = ) (basics#fst (x), basics#fst (y)),
        B!( = ) (snd (x), snd (y))) ;
  let creer (x, y) in Self = basics#pair (x, y) ;
  let element = Self!creer (A!element, B!element) ;
  let print (x) =
     "(" ^ A!print (fst (x)) ^ "," ^ B!print (snd (x)) ^ ")" ;

  proof of refl = (* by definition of ( = ) *) assumed {* *} ;
  proof of symm = assumed {* *} ;

end ;;
```

```
module Setoide =
  struct
  type 'me_as_carrier me_as_species = {
    (* From species ok__phd_sample#Setoide. *)
    element : 'me_as_carrier ;
    (* From species ok__phd_sample#Setoide. *)
    _equal_ : 'me_as_carrier -> 'me_as_carrier -> Basics._focty_bool ;
    (* From species basics#Basic_object. *)
    parse : Basics._focty_string -> 'me_as_carrier ;
    (* From species basics#Basic_object. *)
    print : 'me_as_carrier -> Basics._focty_string ;
    (* From species ok__phd_sample#Setoide. *)
    different : 'me_as_carrier -> 'me_as_carrier -> Basics._focty_bool ;
    }
  let different abst__equal_ (x : 'me_as_carrier) (y : 'me_as_carrier) =
    (Basics.not_b (abst__equal_ x y))
  end ;;

module Monoide =
  struct
  type 'me_as_carrier me_as_species = {
```

```
    (* From species ok__phd_sample#Monoide. *)
    un : 'me_as_carrier ;
    (* From species ok__phd_sample#Monoide. *)
    _star_ : 'me_as_carrier -> 'me_as_carrier -> 'me_as_carrier ;
    (* From species ok__phd_sample#Setoide. *)
    _equal_ : 'me_as_carrier -> 'me_as_carrier -> Basics._focty_bool ;
    (* From species basics#Basic_object. *)
    parse : Basics._focty_string -> 'me_as_carrier ;
    (* From species basics#Basic_object. *)
    print : 'me_as_carrier -> Basics._focty_string ;
    (* From species ok__phd_sample#Monoide. *)
    element : 'me_as_carrier ;
    (* From species ok__phd_sample#Setoide. *)
    different : 'me_as_carrier -> 'me_as_carrier -> Basics._focty_bool ;
    }
  let element abst_un abst__star_ = (abst__star_ abst_un abst_un)
  end ;;

module Setoide_produit =
  struct
  (* Carrier's structure explicitly given by "rep". *)
  type ('a0_as_carrier, 'b1_as_carrier) me_as_carrier =
    'a0_as_carrier * 'b1_as_carrier
  type ('a0_as_carrier, 'b1_as_carrier, 'me_as_carrier) me_as_species = {
    (* From species ok__phd_sample#Setoide_produit. *)
    creer : 'a0_as_carrier -> 'b1_as_carrier -> 'me_as_carrier ;
    (* From species basics#Basic_object. *)
    parse : Basics._focty_string -> 'me_as_carrier ;
    (* From species ok__phd_sample#Setoide_produit. *)
    print : 'me_as_carrier -> Basics._focty_string ;
    (* From species ok__phd_sample#Setoide_produit. *)
    _equal_ : 'me_as_carrier -> 'me_as_carrier -> Basics._focty_bool ;
    (* From species ok__phd_sample#Setoide_produit. *)
    element : 'me_as_carrier ;
    (* From species ok__phd_sample#Setoide. *)
    different : 'me_as_carrier -> 'me_as_carrier -> Basics._focty_bool ;
    }
  let creer (x : 'a0_as_carrier) (y : 'b1_as_carrier) = (Basics.pair x y)
  let print _p_A_print _p_B_print (x : 'me_as_carrier) =
    (Basics._hat_ "("
      (Basics._hat_ (_p_A_print (Basics.fst x))
        (Basics._hat_ "," (Basics._hat_ (_p_B_print (Basics.snd x)) ")"))))
  let _equal_ _p_A__equal_ _p_B__equal_ (x : 'me_as_carrier)
    (y : 'me_as_carrier) =
    (Basics.and_b (_p_A__equal_ (Basics.fst x) (Basics.fst y))
      (_p_B__equal_ (Basics.snd x) (Basics.snd y)))
  let element _p_A_element _p_B_element abst_creer =
    (abst_creer _p_A_element _p_B_element)

  <<<< ATTENTION:                                                    >>>>
  <<<< SKIPPED THE COLLECTION GENERATOR STUFF THAT WE EXPLAIN LATER >>>>
 end ;;
```

```
Module Setoide.
  Record Setoide : Type :=
    mk_record {
    rf_T :> Set ;
    (* From species ok__phd_sample#Setoide. *)
    rf_element : rf_T ;
    (* From species ok__phd_sample#Setoide. *)
    rf__equal_ : rf_T -> rf_T -> basics.bool__t ;
    (* From species basics#Basic_object. *)
    rf_parse : basics.string__t -> rf_T ;
    (* From species basics#Basic_object. *)
    rf_print : rf_T -> basics.string__t ;
```

```
    (* From species ok__phd_sample#Setoide. *)
    rf_different : rf_T -> rf_T -> basics.bool__t ;
    (* From species ok__phd_sample#Setoide. *)
    rf_refl : forall x : rf_T, Is_true ((rf__equal_ x x)) ;
    (* From species ok__phd_sample#Setoide. *)
    rf_symm :
      forall x  y : rf_T,
        Is_true ((rf__equal_ x y)) -> Is_true ((rf__equal_ y x))
    }.

  Definition different (abst_T : Set)
    (abst__equal_ : abst_T -> abst_T -> basics.bool__t) (x : abst_T)
    (y : abst_T) : basics.bool__t := (basics.not_b (abst__equal_ x y)).

End Setoide.

Module Monoide.
  Record Monoide : Type :=
    mk_record {
    rf_T :> Set ;
    (* From species ok__phd_sample#Monoide. *)
    rf_un : rf_T ;
    (* From species ok__phd_sample#Monoide. *)
    rf__star_ : rf_T -> rf_T -> rf_T ;
    (* From species ok__phd_sample#Setoide. *)
    rf__equal_ : rf_T -> rf_T -> basics.bool__t ;
    (* From species basics#Basic_object. *)
    rf_parse : basics.string__t -> rf_T ;
    (* From species basics#Basic_object. *)
    rf_print : rf_T -> basics.string__t ;
    (* From species ok__phd_sample#Monoide. *)
    rf_element : rf_T ;
    (* From species ok__phd_sample#Setoide. *)
    rf_different : rf_T -> rf_T -> basics.bool__t ;
    (* From species ok__phd_sample#Setoide. *)
    rf_refl : forall x : rf_T, Is_true ((rf__equal_ x x)) ;
    (* From species ok__phd_sample#Setoide. *)
    rf_symm :
      forall x  y : rf_T,
        Is_true ((rf__equal_ x y)) -> Is_true ((rf__equal_ y x))
    }.

  Definition element (abst_T : Set) (abst_un : abst_T)
    (abst__star_ : abst_T -> abst_T -> abst_T) : abst_T :=
    (abst__star_ abst_un abst_un).

End Monoide.

Module Setoide_produit.
  Record Setoide_produit (A_T : Set) (B_T : Set) : Type :=
    mk_record {
    rf_T :> Set ;
    (* From species ok__phd_sample#Setoide_produit. *)
    rf_creer : A_T -> B_T -> rf_T ;
    (* From species basics#Basic_object. *)
    rf_parse : basics.string__t -> rf_T ;
    (* From species ok__phd_sample#Setoide_produit. *)
    rf_print : rf_T -> basics.string__t ;
    (* From species ok__phd_sample#Setoide_produit. *)
    rf__equal_ : rf_T -> rf_T -> basics.bool__t ;
    (* From species ok__phd_sample#Setoide_produit. *)
    rf_element : rf_T ;
    (* From species ok__phd_sample#Setoide. *)
    rf_different : rf_T -> rf_T -> basics.bool__t ;
    (* From species ok__phd_sample#Setoide_produit. *)
```

```
    rf_refl : forall x : rf_T, Is_true ((rf__equal_ x x)) ;
    (* From species ok__phd_sample#Setoide_produit. *)
    rf_symm :
      forall x  y : rf_T,
        Is_true ((rf__equal_ x y)) -> Is_true ((rf__equal_ y x))
    }.

  Definition creer (_p_A_T : Set) (_p_B_T : Set)
    (abst_T := ((_p_A_T * _p_B_T)%type)) (x : _p_A_T) (y : _p_B_T) :
    abst_T := (basics.pair _ _ x y).
  Definition print (_p_A_T : Set) (_p_B_T : Set) (_p_A_print :
    _p_A_T -> basics.string__t) (_p_B_print : _p_B_T -> basics.string__t)
    (abst_T := ((_p_A_T * _p_B_T)%type)) (x : abst_T) : basics.string__t :=
    (basics._hat_ coq_builtins.___a_string
      (basics._hat_ (_p_A_print (basics.fst _ _ x))
        (basics._hat_ coq_builtins.___a_string
          (basics._hat_ (_p_B_print (basics.snd _ _ x))
            coq_builtins.___a_string)))).
  Definition _equal_ (_p_A_T : Set) (_p_B_T : Set) (_p_A__equal_ :
    _p_A_T -> _p_A_T -> basics.bool__t) (_p_B__equal_ :
    _p_B_T -> _p_B_T -> basics.bool__t) (abst_T := ((_p_A_T * _p_B_T)%type))
    (x : abst_T) (y : abst_T) : basics.bool__t :=
    (basics.and_b (_p_A__equal_ (basics.fst _ _ x) (basics.fst _ _ y))
      (_p_B__equal_ (basics.snd _ _ x) (basics.snd _ _ y))).
  Definition element (_p_A_T : Set) (_p_B_T : Set) (_p_A_element : _p_A_T)
    (_p_B_element : _p_B_T) (abst_T : Set)
    (abst_creer : _p_A_T -> _p_B_T -> abst_T) : abst_T :=
    (abst_creer _p_A_element _p_B_element).

  (* From species ok__phd_sample#Setoide_produit. *)
  Theorem refl  (_p_A_T : Set) (_p_B_T : Set) (abst_T : Set)
    (abst__equal_ : abst_T -> abst_T -> basics.bool__t):
    forall x : abst_T, Is_true ((abst__equal_ x x)).
  (* Proof assumed because " ". *)
  apply coq_builtins.magic_prove.
  Qed.

  (* From species ok__phd_sample#Setoide_produit. *)
  Theorem symm  (_p_A_T : Set) (_p_B_T : Set) (abst_T : Set)
    (abst__equal_ : abst_T -> abst_T -> basics.bool__t):
    forall x  y : abst_T,
      Is_true ((abst__equal_ x y)) -> Is_true ((abst__equal_ y x)).
  (* Proof assumed because " ". *)
  apply coq_builtins.magic_prove.
  Qed.

    <<<< ATTENTION:                                                  >>>>
  <<<< SKIPPED THE COLLECTION GENERATOR STUFF THAT WE EXPLAIN LATER >>>>
End Setoide_produit.
```

Defined recursive methods

The compilation scheme of a recursive function is pretty different and will be explained in the dedicated section **??**. It however finally uses the same abstraction mechanism (i.e. λ-lifted things). The main difference is induced by the use of the Coq construct "`Function`" and the need for a termination proof.

2.1.5 Fully defined species

In case a species is fully defined, i.e. all its methods are defined, no more remaining only declared, the species can be turned into a collection by an `implements`. To allow creating collections, we must then add to this species a **collection generator**. The intuitive view of such a generator is that it is a function that takes the parameters required

by the method generators and feed them with their related parameters to produce a bunch of effective **methods**. Hence, the collection generator takes as many parameters as the method generators of the species need to abstract the dependencies of the methods (i.e. in fact, the λ-lifts of each method generator) and apply each method generator to the set of parameters it needs. Hence applying the collection to effective arguments will create a bunch of effective methods of the species by applying the method generators. Technically, this bunch of methods is stored in a value of ... the record type representing the species. Hence, a collection is just a value of this record type, storing functions (methods) provided to process value whose type is the carrier of the species.

You may note that this carrier type is not recorded in the record structure. However, methods of the record will obviously have traces of this type in their own type schemes. Turning the carrier abstract (i.e. not exporting its internal structure), it becomes impossible to manipulate it except via the provided methods, i.e. functions stored in the record value representing one collection.

Collection generator function's header

As presented above, the collection generator is a function. Its name is always `collection_create`. As we said, this functions take arguments that represent all the things abstracted in the method generators of the species.

Attention: Because to have a collection generator, the species must be fully defined, it is clear that the only things that can remain abstracted are species parameters' carriers, species parameters' methods and entity parameters ! Never some methods of "`Self`" since all the methods of "`Self`" are ... defined !

Local functions

For each method of the species, we will create a "real method", i.e. create a local function of the collection generator, applying the method generator to its required arguments taken among the effective arguments of the collection generator. Hence, for each method of the species, we build (locally to the collection generator) a function. These local functions will be named: "`local_`" + method's name. We have 2 possible cases to find the method generator to use to create the collection generator:

- Either the method generator belongs to the current inheritance level (i.e. the method was defined at this level in the species). In this case, this generator is simply the name of the method because a local function in the module was generated with this name. In the following sample code, that's the case for the methods `creer`, `print`, `element`, ...
- Or the method generator belongs a previous inheritance level (i.e. the method was defined previously, in an ancestor). In this case, the name of the method generator is qualified by the module hosting the ancestor. This means that is the ancestor belongs to another compilation unit, we need to also specify the module on which the compilation unit is mapped. This gives a name like: file as module + "`.`" + hosting species name + "`.`" + name of the method corresponding to this generator. In the following sample code, that's the case for the method `parse` defined in the species `Basic_object` of the compilation unit "`basics.fcl`".

Creating the record value

Now we have our bunch of functions representing the methods of the species, we just need to create a value of the record type by feeding each record fields with its related function we locally created.

And then, the return value of the collection generator is the record value. Hence, this shows clearly that a collection is in fact a value whose type is the record type we created to model the species.

Sample code to help to summarize

Like we did to explain the code generation model of species in 2.1.4, we use the same sample we used in 2.1.4 and complete the parts about collection generators we previously snipped in the species `Setoide_produit` (the other species, not being fully defined, don't have a collection generator).

```
<<<< ATTENTION:                                                  >>>>
<<<< STUFF BEFORE, PREVIOUSLY SEEN IN PREVIOUS EXPLANATIONS >>>>

module Setoide_produit =
  struct
  <<<< ATTENTION:                                                  >>>>
  <<<< STUFF BEFORE, PREVIOUSLY SEEN IN PREVIOUS EXPLANATIONS >>>>

  (* Fully defined 'Setoide_produit' species's collection generator. *)
  let collection_create () _p_B_element _p_B__equal_ _p_B_print _p_A_element
    _p_A__equal_ _p_A_print =
    (* From species ok__phd_sample#Setoide_produit. *)
    let local_creer = creer in
    (* From species basics#Basic_object. *)
    let local_parse = Basics.Basic_object.parse in
    (* From species ok__phd_sample#Setoide_produit. *)
    let local_print = print _p_A_print _p_B_print in
    (* From species ok__phd_sample#Setoide_produit. *)
    let local__equal_ = _equal_ _p_A__equal_ _p_B__equal_ in
    (* From species ok__phd_sample#Setoide_produit. *)
    let local_element = element _p_A_element _p_B_element local_creer in
    (* From species ok__phd_sample#Setoide. *)
    let local_different = Setoide.different local__equal_ in
    { creer = local_creer ;
      parse = local_parse ;
      print = local_print ;
      _equal_ = local__equal_ ;
      element = local_element ;
      different = local_different ;
       }

  end ;;
```

In the above OCaml you may notice that the collection generator takes a "spurious" `()` ("`unit`") parameter. This is not a mistake and is only used to prevent a collection generator that does not need any parameter (because there is no collection and entity parameter for this species) from having no argument.

In effect, in this case, for sake of non-expansivity, ML type system doesn't allow to generalize type variables appearing in values that are not functional (roughly, very roughly speaking, since there are other cases ... Let's say that functional value can have their type generalized).

This is a problem since type variables appearing in the module representing a species won't be generalizable, then, as soon we create a collection, we instance these type variables by the carrier representation and by parameters' carriers. And if we want to create another collection, since the variables are now instantiated (they are not polymorphic) we can't instantiate them by other types. And we get a type error on OCaml side. For a full example, see the sample code in section 2.1.5.

Obviously, we could add this extra parameter only if the collection generator has no parameter, but for sake of simplicity and homogeneity, we prefer to add it in all the cases.

```
<<<< ATTENTION:                                                  >>>>
<<<< STUFF BEFORE, PREVIOUSLY SEEN IN PREVIOUS EXPLANATIONS >>>>

Module Setoide_produit.
  <<<< ATTENTION:                                                  >>>>
  <<<< STUFF BEFORE, PREVIOUSLY SEEN IN PREVIOUS EXPLANATIONS >>>>

  (* Fully defined 'Setoide_produit' species's collection generator. *)
  Definition collection_create (_p_A_T : Set) (_p_B_T : Set) _p_A_element
    _p_A__equal_ _p_A_print _p_B_element _p_B__equal_ _p_B_print :=
    let local_rep := ((_p_A_T * _p_B_T)%type) in
    (* From species ok__phd_sample#Setoide_produit. *)
    let local_creer := creer _p_A_T _p_B_T in
    (* From species basics#Basic_object. *)
    let local_parse := basics.Basic_object.parse local_rep in
```

```
    (* From species ok__phd_sample#Setoide_produit. *)
    let local_print := print _p_A_T _p_B_T _p_A_print _p_B_print in
    (* From species ok__phd_sample#Setoide_produit. *)
    let local__equal_ := _equal_ _p_A_T _p_B_T _p_A__equal_ _p_B__equal_ in
    (* From species ok__phd_sample#Setoide_produit. *)
    let local_element := element _p_A_T _p_B_T _p_A_element _p_B_element
      local_rep local_creer in
    (* From species ok__phd_sample#Setoide. *)
    let local_different := Setoide.different local_rep local__equal_ in
    (* From species ok__phd_sample#Setoide_produit. *)
    let local_refl := refl _p_A_T _p_B_T local_rep local__equal_ in
    (* From species ok__phd_sample#Setoide_produit. *)
    let local_symm := symm _p_A_T _p_B_T local_rep local__equal_ in
    mk_record (_p_A_T : Set) (_p_B_T : Set) local_rep local_creer local_parse
    local_print local__equal_ local_element local_different local_refl
    local_symm.

End Setoide_produit.
```

Sample code for extra () parameter for OCaml

As described in 2.1.5, here is a full example showing the need to have an extra `()` in OCaml for collection generators.

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


let print_bool =
   internal bool -> string
   external | caml -> {* string_of_bool *} | coq -> {* (* [Unsure] *) *}
;;
let ext_nil =
   internal list ('a)
   external | caml -> {* [] *} | coq -> {* (* [Unsure] *) *}
;;
let ext_cons =
   internal 'a -> list ('a) -> basics#list ('a)
   external | caml -> {* (fun e l -> e :: l) *} | coq -> {* (* [Unsure] *) *}
;;
let ext_head =
  internal list ('a) -> 'a
  external | caml -> {* List.hd *} | coq -> {* (* [Unsure] *) *}
;;
let ext_tail =
  internal list ('a) -> list ('a)
  external | caml -> {* List.tl *} | coq -> {* (* [Unsure] *) *}
;;


species Concrete_list (E is Basic_object) =
  representation = basics#list (E) ;
  let equal (x in Self, y in Self) in bool = syntactic_equal (x, y) ;
  let nil in Self = ext_nil ;
  let cons (e, l) in Self = ext_cons (e, l) ;
  let head (l in Self) in E = ext_head (l) ;
  let tail (l in Self) in Self = ext_tail (l) ;
   let rec map (f, l) =
     if equal (l, nil) then nil
     else
       let h = head (l) in
       let q = tail (l) in
       let h2 = f (h) in
       let q2 = map (f, q) in
       cons (h2, q2) ;
```

```
end ;;
```

Once compiled to OCaml we get the following code:

```
let print_bool =  string_of_bool ;;
let ext_nil = [] ;;
let ext_cons = (fun e l -> e :: l) ;;
let ext_head = List.hd ;;
let ext_tail = List.tl ;;

module Concrete_list =
  struct
  (* Carrier's structure explicitly given by "rep". *)
  type 'e0_as_carrier me_as_carrier = 'e0_as_carrier Basics._focty_list
  type ('e0_as_carrier, 'me_as_carrier) me_as_species = {
    (* From species test#Concrete_list. *)
    cons : 'e0_as_carrier ->
             'e0_as_carrier Basics._focty_list -> 'me_as_carrier ;
    (* From species test#Concrete_list. *)
    equal : 'me_as_carrier -> 'me_as_carrier -> Basics._focty_bool ;
    (* From species test#Concrete_list. *)
    head : 'me_as_carrier -> 'e0_as_carrier ;
    (* From species test#Concrete_list. *)
    nil : 'me_as_carrier ;
    (* From species test#Concrete_list. *)
    tail : 'me_as_carrier -> 'me_as_carrier ;
    (* From species test#Concrete_list. *)
    map : ('e0_as_carrier -> 'e0_as_carrier) ->
            'me_as_carrier -> 'me_as_carrier ;
    }
  let cons (e : 'e0_as_carrier) (l : 'e0_as_carrier Basics._focty_list) =
    (ext_cons e l)
  let equal (x : 'me_as_carrier) (y : 'me_as_carrier) =
    (Basics.syntactic_equal x y)
  let head (l : 'me_as_carrier) = (ext_head l)
  let nil = ext_nil
  let tail (l : 'me_as_carrier) = (ext_tail l)
  let rec map abst_cons abst_equal abst_head abst_nil abst_tail
    (f : 'e0_as_carrier -> 'e0_as_carrier) (l : 'me_as_carrier) =
    if (abst_equal l abst_nil) then abst_nil else let h = (abst_head l)
      in
      let q = (abst_tail l) in
      let h2 = (f h)
      in
      let q2 = (map abst_cons abst_equal abst_head abst_nil abst_tail f q)
      in
      (abst_cons h2 q2)
  (* Fully defined 'Concrete_list' species's collection generator. *)
  let collection_create !!! WE REMOVED THE EXTRA () !!! =
    (* From species test#Concrete_list. *)
    let local_cons = cons in
    (* From species test#Concrete_list. *)
    let local_equal = equal in
    (* From species test#Concrete_list. *)
    let local_head = head in
    (* From species test#Concrete_list. *)
    let local_nil = nil in
    (* From species test#Concrete_list. *)
    let local_tail = tail in
    (* From species test#Concrete_list. *)
    let local_map = map local_cons local_equal local_head local_nil
      local_tail in
    { cons = local_cons ;
      equal = local_equal ;
      head = local_head ;
      nil = local_nil ;
```

```
      tail = local_tail ;
      map = local_map ;
       }

  end ;;
```

If we check the interface of the OCaml compilation unit, we can see that the module `Concrete_list` has type:

```
module Concrete_list :
  sig
    type 'a me_as_carrier = 'a Basics._focty_list
    type ('a, 'b) me_as_species = {
      cons : 'a -> 'a Basics._focty_list -> 'b;
      equal : 'b -> 'b -> Basics._focty_bool;
      head : 'b -> 'a;
      nil : 'b;
      tail : 'b -> 'b;
      map : ('a -> 'a) -> 'b -> 'b;
    }
    val cons : 'a -> 'a Basics._focty_list -> 'a list
    val equal : 'a -> 'a -> bool
    val head : 'a list -> 'a
    val nil : 'a list
    val tail : 'a list -> 'a list
    val map :
      ('a -> 'b -> 'b) ->
      ('c -> 'b -> bool) ->
      ('c -> 'a) -> 'b -> ('c -> 'c) -> ('a -> 'a) -> 'c -> 'b
    val collection_create : ('_a, '_a Basics._focty_list) me_as_species
  end
```

where the function `collection_create` has a non generalised type variable `'_a`. Hence, to continue the example, we just need to create two collections, one with `int` as carrier, the other with `bool` and to create 2 collections of lists using these 2 collections as argument.

```
species Contrete_int inherits Basic_object =
  representation = basics#int ;
  let print = string_of_int ;
end ;;
collection Int implements Contrete_int ;;

species Contrete_bool inherits Basic_object =
  representation = basics#bool ;
  let print = print_bool ;
end ;;
collection Bool implements Contrete_bool ;;
```

When creating the last collection implementing list of booleans, the `'_a` type variable was already instantiated by `int`, hence leading to OCaml complaining:

```
File ".....", line 210, characters 13-33:
This expression has type
  int -> int Basics._focty_list -> int Basics._focty_list
but is here used with type
  bool -> Bool.me_as_carrier Basics._focty_list -> me_as_carrier
```

Hence, adding a dummy parameter to the collection generator, it can now be generalised (i.e. become polymorphic) and there is no more instanciation issue.

2.2 Collection

Collections are compiled differently as species but they start by exactly the same kind of record type definition. This record type will represent the type of data in the target language collections are mapped onto.

Things differ after. In effect, we do not need anymore to create the methods since all were already defined in the fully defined species we "`implements`. The aim to get a collection, it to get a value of the record type, where fields are filled with the functions representing methods of the species we "`implements`". In this species we created a "collection generator" that was a function taking arguments representing dependencies on the species parameters (collection and entity) and returning a value of the record type ...a collection. So, to compile a collection, we will apply the collection generator to things it need to give us material to finally create the collection value that will always be named `effective_collection`.

Once done, we will get from the record generated by the collection generator of the species we "`implements`", each field's value and put it in a record whose type is **our** (i.e. the collection) type. It is then simply a verbatim copy since collection never add fields; so the species the collection implements and the collection have the same methods, so the records have the same fields. With this process, we ensure that the collection will be only type-compatible with itself and won't be type-compatible with the species it "`implements`" and also not with other collections extracted from this species (with the same arguments).

Now, the question is to know what to apply to the collection generator of the species we "`implements`". In fact, we need to apply this generator to the methods of the collections used to instantiate the collection parameters of the species we "`implements`". For instance, going on with our example started in 2.1.4, we add a few FoCaLize code to create a `Monoide_produit` some fully defined species to represent integers to finally build a collection representing couples of integers:

More FoCaLize code to create collections

```
species Monoide_produit (C is Monoide, D is Monoide)
  inherits Monoide, Setoide_produit (C, D) =
...
end ;;

species Entiers_concrets inherits Monoide =
  representation = basics#int ;
...
end ;;

collection Les_entiers implements Entiers_concrets ;;

collection Couple_d_entiers implements
  Monoide_produit (Les_entiers, Les_entiers)
;;
```

We get interested directly by the way to get the collection `Couple_d_entiers` since it is more interesting because the implemented species has collection parameters. For the collection `Les_entiers`, the process is the same, except there is no problem of instanciation because there is no parameter. Here we see that the collection parameters `C` and `D` of `Monoide_produit` were instantiated by the collections `Les_entiers` and `Les_entiers`. The collection generator of `Couple_d_entiers` is parametrised (due to dependencies on species parameters) by several methods of the species parameters (amongst others, trust me ☺, `un`, `print`, `element`, ...). We then must apply the collection generator to the corresponding methods of "by what C" was instantiated and "by what D" was instantiated, i.e from `Les_entiers` and from `Les_entiers`.

Now, where can we get these methods of `Les_entiers` ? Since it is a collection, the module hosting the collection contains a `effective_collection` value that has a type being a record. Then we just need to pick in the fields of this value to have the arguments we want to give to the collection generator.

Note: It appears that in this case, since the 2 parameters are "`is` the same collection", the dependencies will be the same twice. But that's just for this particular case. By the way, this also means that we do not "optimise" telling "Oh, I see the dependencies are exactly on the same functions of the same species, so let's keep only one occurrence of the parameter...". No we keep the model without exception.

We then get the generated OCaml code for the 2 create collections:

OCaml code for collections

```
module Les_entiers =
  struct
  (* Carrier's structure explicitly given by "rep". *)
  type me_as_carrier = Basics._focty_int
  type 'me_as_carrier me_as_species = {
    (* From species ok__phd_sample#Entiers_concrets. *)
    parse : Basics._focty_string -> me_as_carrier ;
    (* From species ok__phd_sample#Entiers_concrets. *)
    print : me_as_carrier -> Basics._focty_string ;
    (* From species ok__phd_sample#Entiers_concrets. *)
    un : me_as_carrier ;
    (* From species ok__phd_sample#Entiers_concrets. *)
    _star_ : me_as_carrier -> me_as_carrier -> me_as_carrier ;
    (* From species ok__phd_sample#Entiers_concrets. *)
    _equal_ : me_as_carrier -> me_as_carrier -> Basics._focty_bool ;
    (* From species ok__phd_sample#Monoide. *)
    element : me_as_carrier ;
    (* From species ok__phd_sample#Setoide. *)
    different : me_as_carrier -> me_as_carrier -> Basics._focty_bool ;
    }
  let effective_collection =
    let t =
      Entiers_concrets.collection_create () in
    { parse = t.Entiers_concrets.parse ;
      print = t.Entiers_concrets.print ;
      un = t.Entiers_concrets.un ;
      _star_ = t.Entiers_concrets._star_ ;
      _equal_ = t.Entiers_concrets._equal_ ;
      element = t.Entiers_concrets.element ;
      different = t.Entiers_concrets.different ;
       }
  end ;;

module Couple_d_entiers =
  struct
  (* Carrier's structure explicitly given by "rep". *)
  type me_as_carrier = Les_entiers.me_as_carrier * Les_entiers.me_as_carrier
  type 'me_as_carrier me_as_species = {
    (* From species ok__phd_sample#Setoide_produit. *)
    creer : Les_entiers.me_as_carrier ->
              Les_entiers.me_as_carrier -> me_as_carrier ;
    (* From species basics#Basic_object. *)
    parse : Basics._focty_string -> me_as_carrier ;
    (* From species ok__phd_sample#Setoide_produit. *)
    print : me_as_carrier -> Basics._focty_string ;
    (* From species ok__phd_sample#Monoide_produit. *)
    _star_ : me_as_carrier -> me_as_carrier -> me_as_carrier ;
    (* From species ok__phd_sample#Setoide_produit. *)
    _equal_ : me_as_carrier -> me_as_carrier -> Basics._focty_bool ;
    (* From species ok__phd_sample#Setoide_produit. *)
    element : me_as_carrier ;
    (* From species ok__phd_sample#Monoide_produit. *)
    un : me_as_carrier ;
    (* From species ok__phd_sample#Setoide. *)
    different : me_as_carrier -> me_as_carrier -> Basics._focty_bool ;
    }
  let effective_collection =
    let t =
      Monoide_produit.collection_create ()
      Les_entiers.effective_collection.Les_entiers.un
      Les_entiers.effective_collection.Les_entiers._star_
      Les_entiers.effective_collection.Les_entiers._equal_
```

```
      Les_entiers.effective_collection.Les_entiers.print
      Les_entiers.effective_collection.Les_entiers.element
      Les_entiers.effective_collection.Les_entiers.un
      Les_entiers.effective_collection.Les_entiers._star_
      Les_entiers.effective_collection.Les_entiers._equal_
      Les_entiers.effective_collection.Les_entiers.print
      Les_entiers.effective_collection.Les_entiers.element in
    { creer = t.Monoide_produit.creer ;
      parse = t.Monoide_produit.parse ;
      print = t.Monoide_produit.print ;
      _star_ = t.Monoide_produit._star_ ;
      _equal_ = t.Monoide_produit._equal_ ;
      element = t.Monoide_produit.element ;
      un = t.Monoide_produit.un ;
      different = t.Monoide_produit.different ;
       }
  end ;;
```

The Coq code follows exactly the same scheme. It appears to be more tricky and bigger for 2 reasons: first logical methods are in the model, second the record access notation is much more complex in Coq that it is in OCaml (where one just need to say `val.field`).

In effect in Coq to access a field of a value of type record, one must provide, of course the value and the field name, but also all the effective arguments that were provided when the type record was created from the `mk_record` that represents the species. For instance, our collection `Couple_d_entiers "implements" a Monoide_produit`. Looking at the record type of `Monoide_produit`, we see:

```
  Record Monoide_produit (C_T : Set) (D_T : Set) : Type :=
    mk_record {
    rf_T :> Set ;
    (* From species ok__phd_sample#Setoide_produit. *)
    rf_creer : C_T -> D_T -> rf_T ;
    ...
```

that is we have 2 parameters that are the carriers of the collection parameters. When we want to pick values from the fields of the value returned by the collection generator (hence from the one of `Monoide_produit`), we will need to make explicit by what these parameters were instantiated when we used the generator. And here, we instantiated twice with the carrier of `Les_entiers`. Then, accessing the field `rf_creer` (corresponding to the method `creer` of the species `Monoide_produit`) of the value `v` returned by the collection generator:

```
let t :=
  Monoide_produit.collection_create
  Les_entiers.effective_collection.(Les_entiers.rf_T)
  Les_entiers.effective_collection.(Les_entiers.rf_T)
  .... in
```

will look like:

```
 t.(Monoide_produit.rf_creer
      Les_entiers.effective_collection.(Les_entiers.rf_T)
      Les_entiers.effective_collection.(Les_entiers.rf_T))
```

So the full generated Coq code will be:

Coq code for collections

```
Module Les_entiers.
  Record Les_entiers : Type :=
    mk_record {
    rf_T :> Set ;
    (* From species ok__phd_sample#Entiers_concrets. *)
    rf_parse : basics.string__t -> rf_T ;
    (* From species ok__phd_sample#Entiers_concrets. *)
    rf_print : rf_T -> basics.string__t ;
```

```
    (* From species ok__phd_sample#Entiers_concrets. *)
    rf_un : rf_T ;
    (* From species ok__phd_sample#Entiers_concrets. *)
    rf__star_ : rf_T -> rf_T -> rf_T ;
    (* From species ok__phd_sample#Entiers_concrets. *)
    rf__equal_ : rf_T -> rf_T -> basics.bool__t ;
    (* From species ok__phd_sample#Monoide. *)
    rf_element : rf_T ;
    (* From species ok__phd_sample#Setoide. *)
    rf_different : rf_T -> rf_T -> basics.bool__t ;
    (* From species ok__phd_sample#Entiers_concrets. *)
    rf_refl : forall x : rf_T, Is_true ((rf__equal_ x x)) ;
    (* From species ok__phd_sample#Entiers_concrets. *)
    rf_symm :
      forall x  y : rf_T,
        Is_true ((rf__equal_ x y)) -> Is_true ((rf__equal_ y x))
    }.

  Let effective_collection :=
    let t :=
      Entiers_concrets.collection_create in
    mk_record t.(Entiers_concrets.rf_T) t.(Entiers_concrets.rf_parse)
      t.(Entiers_concrets.rf_print) t.(Entiers_concrets.rf_un)
      t.(Entiers_concrets.rf__star_) t.(Entiers_concrets.rf__equal_)
      t.(Entiers_concrets.rf_element) t.(Entiers_concrets.rf_different)
      t.(Entiers_concrets.rf_refl) t.(Entiers_concrets.rf_symm).

End Les_entiers.

Module Couple_d_entiers.
  Record Couple_d_entiers : Type :=
    mk_record {
    rf_T :> Set ;
    (* From species ok__phd_sample#Setoide_produit. *)
    rf_creer : Les_entiers.effective_collection.(Les_entiers.rf_T) ->
                Les_entiers.effective_collection.(Les_entiers.rf_T) -> rf_T ;
    (* From species basics#Basic_object. *)
    rf_parse : basics.string__t -> rf_T ;
    (* From species ok__phd_sample#Setoide_produit. *)
    rf_print : rf_T -> basics.string__t ;
    (* From species ok__phd_sample#Monoide_produit. *)
    rf__star_ : rf_T -> rf_T -> rf_T ;
    (* From species ok__phd_sample#Setoide_produit. *)
    rf__equal_ : rf_T -> rf_T -> basics.bool__t ;
    (* From species ok__phd_sample#Setoide_produit. *)
    rf_element : rf_T ;
    (* From species ok__phd_sample#Monoide_produit. *)
    rf_un : rf_T ;
    (* From species ok__phd_sample#Setoide. *)
    rf_different : rf_T -> rf_T -> basics.bool__t ;
    (* From species ok__phd_sample#Setoide_produit. *)
    rf_refl : forall x : rf_T, Is_true ((rf__equal_ x x)) ;
    (* From species ok__phd_sample#Setoide_produit. *)
    rf_symm :
      forall x  y : rf_T,
        Is_true ((rf__equal_ x y)) -> Is_true ((rf__equal_ y x))
    }.

  Let effective_collection :=
    let t :=
      Monoide_produit.collection_create
      Les_entiers.effective_collection.(Les_entiers.rf_T)
      Les_entiers.effective_collection.(Les_entiers.rf_T)
      Les_entiers.effective_collection.(Les_entiers.rf_un)
      Les_entiers.effective_collection.(Les_entiers.rf__star_)
```

```
      Les_entiers.effective_collection.(Les_entiers.rf__equal_)
      Les_entiers.effective_collection.(Les_entiers.rf_print)
      Les_entiers.effective_collection.(Les_entiers.rf_element)
      Les_entiers.effective_collection.(Les_entiers.rf_un)
      Les_entiers.effective_collection.(Les_entiers.rf__star_)
      Les_entiers.effective_collection.(Les_entiers.rf__equal_)
      Les_entiers.effective_collection.(Les_entiers.rf_print)
      Les_entiers.effective_collection.(Les_entiers.rf_element) in
    mk_record
      t.(Monoide_produit.rf_T
         Les_entiers.effective_collection.(Les_entiers.rf_T)
         Les_entiers.effective_collection.(Les_entiers.rf_T))
      t.(Monoide_produit.rf_creer
         Les_entiers.effective_collection.(Les_entiers.rf_T)
         Les_entiers.effective_collection.(Les_entiers.rf_T))
      t.(Monoide_produit.rf_parse
         Les_entiers.effective_collection.(Les_entiers.rf_T)
         Les_entiers.effective_collection.(Les_entiers.rf_T))
      t.(Monoide_produit.rf_print
         Les_entiers.effective_collection.(Les_entiers.rf_T)
         Les_entiers.effective_collection.(Les_entiers.rf_T))
      t.(Monoide_produit.rf__star_
         Les_entiers.effective_collection.(Les_entiers.rf_T)
         Les_entiers.effective_collection.(Les_entiers.rf_T))
      t.(Monoide_produit.rf__equal_
         Les_entiers.effective_collection.(Les_entiers.rf_T)
         Les_entiers.effective_collection.(Les_entiers.rf_T))
      t.(Monoide_produit.rf_element
         Les_entiers.effective_collection.(Les_entiers.rf_T)
         Les_entiers.effective_collection.(Les_entiers.rf_T))
      t.(Monoide_produit.rf_un
         Les_entiers.effective_collection.(Les_entiers.rf_T)
         Les_entiers.effective_collection.(Les_entiers.rf_T))
      t.(Monoide_produit.rf_different
         Les_entiers.effective_collection.(Les_entiers.rf_T)
         Les_entiers.effective_collection.(Les_entiers.rf_T))
      t.(Monoide_produit.rf_refl
         Les_entiers.effective_collection.(Les_entiers.rf_T)
         Les_entiers.effective_collection.(Les_entiers.rf_T))
      t.(Monoide_produit.rf_symm
         Les_entiers.effective_collection.(Les_entiers.rf_T)
         Les_entiers.effective_collection.(Les_entiers.rf_T)).

end Couple_d_entiers.
```

2.3 Toplevel values

At toplevel, since we are not in a species, there is no possible dependencies on species parameters (since there is no parameter) and no possible dependencies on methods of `Self` (since there is no notion of `Self`). For this reason, we do not have to wonder about how naming methods od `Self` and of species parameters.

Hence, code generation for toplevel value definitions (either constants or functions) is a simple translation into the target language with no particular analysis to do before-hand. The name of the generated definition is simply the name given in the FoCaLize source file and the definition is generated at toplevel.

Hence, we are in the case where there is no more problems than those intrinsic to the target language. For instance, in Coq, we will have to be careful with polymorphic identifiers and handle the explicit polymorphism by providing extra type arguments like seen in 2.1.3.

For instance:

FoCaLize toplevel value definition

```
let x = 42 ;;
```

```
let fst (x) =
  match x with
  | (v, _) -> v
;;
```

Generated OCaml toplevel value definition

```
let x = 42 ;;

let snd (x : 'a * 'b) =
  match x with
   | (_, v) ->
     (begin
     v
     end)
;;
```

In the Coq code, we see again here a hack to speak of the type of tuples. In effect, we must explicitly tell Coq that the * used in the type expression `(__var_a * __var_b)` is the * dealing with **types**. That's the reason of the presence of the `%type` notation post-fixing the type expression of the tuple.

Generated Coq toplevel value definition

```
Let x : basics.int__t := 42.

Let snd (__var_a : Set) (__var_b : Set) (x : ((__var_a * __var_b)%type)) :
    __var_b :=
  match x with
   | (_, v) =>
       v
  end.
```

2.4 Toplevel theorems

Toplevel theorems, like toplevel values do not have any dependencies since they are not in a species. Hence they are generated like other theorems, with the `Section` mechanism for the proof part (if the proof is done in FoCaL Proof Language), but with no λ-lifting stuff.

For instance:

FoCaLize toplevel theorem definition

```
theorem beq_trans :
  all x y z in 'a, x = y -> y = z -> x = z
  proof = assumed {* Import trusted external code. *} ;;
```

Generated Coq toplevel theorem definition

```
Theorem beq_trans :
  forall __var_a : Set, forall x  y  z : __var_a,
    Is_true ((_equal_ _ x y)) ->
      Is_true ((_equal_ _ y z)) -> Is_true ((_equal_ _ x z)).
(* Proof assumed because " Import trusted external code. ". *)
apply coq_builtins.magic_prove.
Qed.
```

2.5 Type definitions

Type definitions always appear at toplevel, i.e. outside a species. For a given type definition named "myty" in FoCaLize, the corresponding generated names are historically a few different in OCaml and in Coq. In OCaml the generated type will be named "tt _focty_" + mtty. In Coq, the generated type will be named mtty + "__t".

We have 4 kinds of type definitions:

- Alias : the definition doesn't create a new type (i.e. a new set of values inhabiting a type) but is a shortcut for naming a (complex ?) type expression. For instance: `type t = alias (int * char * int);;` The type constructor `t` is only an alias for the type of tuples with 3 components whose first and third components are an integer and second one is a character. In particular, `t` will be unifiable with any occurrence of its alias.

- Union : the definition create a new sum type and gives its value constructors. This new type will be compatible with only itself. For instance: `type t = |A |B (int * char);;` The type constructor `t` denotes the type whose 2 value constructors are `A` and `B`. The constructor `B` is parametrised by a pair of an integer and a character.

- Record: the definition create a new record type and gives its fields names and types. This new type will be compatible with only itself. For instance: `type t = { name :string ;birth :int } ;;` The type constructor `t` denotes the record type whose 2 fields are named `name` and `birth` and types are respectively `string` and `int`.

- External : the definition establishes the link between a type of an external language and its representation in FoCaLize. Because they are quite complex, they will be studied in more details in a next dedicated section (see 2.6.1).

2.5.1 Type alias

As stated before, such a **type definition** introduces a new type name (i.e. type constructor) to represent a **type expression**. Fortunately, both OCaml and Coq have the notion of type alias. Hence, we simply need to translate the FoCaLize type expression in the corresponding target type expression. This doesn't pose any particular problem. The only thing to remind is that since in Coq polymorphism is explicit, if a type definition is parametrised by a type variable, then one must explicitly give this variable the type `Set`. In OCaml there is no need to give the type of this variable, the syntax only requires to bind this variable name in the type definition's name (i.e. `type 'a t = ... ;;`).

Below follow a few simple alias type definitions examples with their corresponding generated source code in Coq and OCaml.

Generated Coq type definition

```
open "basics" ;;

type t = alias int ;;
type u = alias (t * (char * bool * string)) ;;
type poly_pair ('a) = alias ('a * 'a) ;;
type inv_prod ('a, 'b) = alias ('b * 'a) ;;
```

Generated Coq type definition

```
type _focty_t = Basics._focty_int ;;
type _focty_u =
  Basics._focty_int *
    (Basics._focty_char * Basics._focty_bool * Basics._focty_string) ;;
type 'a _focty_poly_pair = 'a * 'a ;;
type ('a, 'b) _focty_inv_prod = 'b * 'a ;;
```

Generated Coq type definition

```
Require basics.
Definition t__t := basics.int__t.
Definition u__t :=
  ((basics.int__t
    * (((basics.char__t * basics.bool__t * basics.string__t)%type)))%type).
Definition poly_pair__t (__var_a : Set) := ((__var_a * __var_a)%type).
Definition inv_prod__t (__var_b : Set) (__var_c : Set) :=
  ((__var_c * __var_b)%type).
```

The only technical thing is to make sure that type variables are correctly identified all along the unifications to be sure that they are properly link between their declaration (with the type name) and their usage. This is illustrated by the `inv_prod` definition where we want to have variables inverted between the binding order and their usage order.

Be careful that the aliases we made here are **tuples**. They use the type constructor $*$ (star) to create **one** type expression that is a tuple compound of several type (sub-)expressions. This will not have to be confused with the problem of parametrising sum type value constructors with **one** tuple of several components or several values as explained later in 2.5.2.

2.5.2 Type union (sum type)

As stated before, such a sum type definition introduces a new type name (i.e. type constructor) with its value constructors. Again fortunately, both OCaml and Coq have the notion of sum type. Hence, we do not need to encode sums, we need to translate them into the corresponding constructs in the target languages. Each value constructor in FoCaLize will lead to a value constructor in the target languages.

In OCaml the return type of a constructor doesn't need to be explicit. Conversely, in Coq it needs to be. Hence for the simple sum type definition:

Simple sum type in FoCaLize

```
type t =
  | A
  | B
 ;;
```

we get the following OCaml code where only the value constructors names `A` and `B` are emitted:

Simple sum type generated in OCaml

```
type _focty_t =
  | A
  | B
 ;;
```

Whereas in Coq, in addition to the constructors names, one must explicitly say that they are values of this type:

Simple sum type generated in Coq

```
Inductive t__t : Set :=
  | A : (t__t)
  | B : (t__t).
```

Parametrised value constructors are a bit more complex. We first introduce the simplest case, where value constructors have one parameter. In this case, in OCaml code, we have the quite obvious translation that tells "`of`" the argument's type. In Coq the constructor will simply be considered to have a functional type whose argument is the type of the value constructor's argument and whose return type is the defined type itself. Hence, even for recursive and/or parametrised type definitions, we easily generate the code of the example:

Sum types with simply parametrised value constructors

```
type t_poly ('a) =
```

```
  | None
  | Some ('a)
;;

type t_rec =
  | TR1 (int)
  | TR2 (t_rec)
;;

type t_poly_rec ('a) =
  | TPR1 ('a)
  | TPR2 (t_poly_rec ('a))
;;
```

in OCaml to get:

Sum type generated in OCaml

```
type 'a _focty_t_poly =
  | None
  | Some of ('a)
 ;;
type _focty_t_rec =
  | TR1 of (Basics._focty_int)
  | TR2 of (_focty_t_rec)
 ;;
type 'a _focty_t_poly_rec =
  | TPR1 of ('a)
  | TPR2 of ('a _focty_t_poly_rec)
 ;;
```

and in Coq to get:

Sum type generated in Coq

```
Inductive t_poly__t (__var_a : Set) : Set :=
  | None : (t_poly__t __var_a)
  | Some : (__var_a -> (t_poly__t __var_a)).

Inductive t_rec__t : Set :=
  | TR1 : (basics.int__t -> t_rec__t)
  | TR2 : (t_rec__t -> t_rec__t).

Inductive t_poly_rec__t (__var_b : Set) : Set :=
  | TPR1 : (__var_b -> (t_poly_rec__t __var_b))
  | TPR2 : ((t_poly_rec__t __var_b) -> (t_poly_rec__t __var_b)).
```

in both of which see see that the generated type constructor's name is consistently the same between the code of the name of the definition and the generated type expressions (so, recursive) where this type constructor's name appears.

The translation of sum type is quite complete, but we still need to deal with a choice of representation to do about value constructors that "look having" several parameters, like in

Value constructor parameterised by "several" arguments

```
type t_prm_val_cstr =
  | C (int * int * int)
;;
```

In such a case, is the value constructor `C` parametrised by **three** integers or by **one** tuple of 3 integers ? This makes a particular difference in Coq because value constructors are curried. For Coq unless an explicit tuple is stated, the value constructor for the above example must have the functional type `int -> int -> int -> int -> t_prm_val_cstr`.

Of course, it would be possible to always group the arguments into one unique tuple,but making proofs using such a value constructor would not be tractable (dixit Renaud). In OCaml, since we do not have this concern of proof, this choice would not be a real problem.

In fact, in the syntax, FoCaLize propose two value constructor argument expressions different: one for "several" parameters and one for 1 tuple parameter of "several" components. Hence, translation into the target languages are not ambiguous. In a sum type definition, for a value constructor, arguments separated by `,` ("comma") are considered as "several arguments", and arguments separated by `*` ("star") are considered as grouped inside one unique argument that is a tuple. Hence, for the following FoCaLize example:

Value constructor parameterised by "several" arguments (2)

```
type t_prm_val_cstr =
  | C (int * int * int)
  | D (int, int, int)
;;
```

we get the generated OCaml code that follows, where we do make any difference between one and several arguments: all is every considered as a tuple

Value constructor parameterised by "several" arguments in OCaml (2)

```
type _focty_t_prm_val_cstr =
  | C of ((Basics._focty_int * Basics._focty_int * Basics._focty_int))
  | D of (Basics._focty_int * Basics._focty_int * Basics._focty_int)
 ;;
```

and the generated Coq code where the types of `C` and `D` are clearly different:

Value constructor parameterised by "several" arguments in Coq (2)

```
Inductive t_prm_val_cstr__t : Set :=
  | C :
    ((((basics.int__t * basics.int__t * basics.int__t)%type)) ->
      t_prm_val_cstr__t)
  | D :
    (basics.int__t -> basics.int__t -> basics.int__t -> t_prm_val_cstr__t).
```

2.5.3 Record type

As stated before, such a record type definition introduces a new type name (i.e. type constructor) with its fields labels and types. Again fortunately, both OCaml and Coq have the notion of record type. Hence, we do not need to encode record, we need to translate them into the corresponding constructs in the target languages. Each field in FoCaLize will lead to a field in the record generated in the target languages.

The translation process is simple since it map a each field label of the FoCaLize definition onto a field label of the same name in the target language: The type of the field is generated like any type expression in the target language.

Record type definitions in FoCaLize

```
type r0 = { x0 = int ; y0 = float } ;;
type r1 ('a) = { x1 = 'a } ;;
```

Generated record type definitions in OCaml

```
type _focty_r0 = {
   x0 : Basics._focty_int ;
   y0 : Basics._focty_float ;
} ;;

type 'a _focty_r1 = {
    x1 : 'a ;
} ;;
```

The only difference for Coq is (like already encountered when dealing with the record type representing species and collections, in 2.1.3 and 2.2) that a record type definition requires a "record constructor". By convention, it will always be named by "mk_" + the record type's name + "__t".

Generated record type definitions in Coq

```
Record r0__t : Type :=
  mk_r0__t {
  x0 : basics.int__t ;
  y0 : basics.float__t
  }.

Record r1__t (__var_c : Set) : Type :=
  mk_r1__t {
  x1 : __var_c
  }.
```

2.6 External definitions

External definitions are intended to make the interface between code imported from foreign target languages to be able to use it on FoCaLize's side. There are 2 kinds of external definitions: type definitions and value definitions.

Type definitions are used either to import basic types and to allows FoCaLizeto map its internal types on them (for instance, `int`, `bool`, etc that are built-in type in FoCaLize and that must be mapped onto their OCaml and Coq counterparts), or to specify the type of values developed outside FoCaLize and that we want to use from FoCaLize.

Value definitions are used to provide support (i.e. primitives to manipulate) and inhabitants to external types or to provide values of a type known in FoCaLize but whose construction was done outside the FoCaLize source code.

Any external definition contains 2 aspects: its "internal" view that says how the defined entity must be seen by FoCaLize(during its analyses) and its "external" view that says how this entity must be mapped onto target languages (during code generation) when it is used in a FoCaLize source code.

2.6.1 External type definitions

An external type definition starts like a regular type definition, i.e. by the type constructor's name and possibly parameters: **`type`** `('a, 'b)t =`). The body of the definition shows that it is an external by having the following shape:

```
type t  =
  internal ...
  external ...
and ...
and ...
```

Attention: Be careful that the `and ...` are **not** other type definitions: they are optional and belong to the current external type definition. We will see later the meaning.

The `internal` clause

For a type definition, the `internal` clause shows at which FoCaLize type definition the body of the external definition corresponds. In fact this is equivalent to establish an alias between the structure of the external type and the structure of the type definition inside FoCaLize.

However, since we do not always want to really have this alias (i.e. an equivalence between the internal and external representations), this clause can be empty. In this case, the created type constructor will be fully abstract on FoCaLize's side and the only way to make and manipulate values of this type will have to be provided via other external (values) definitions or via built-ins of the compiler.

So, when the `internal` clause is not empty, we said that is represented a type definition. Hence this clause can have 3 shapes: an alias, a sum or a record (the 3 kinds of type definitions in FoCaLize). For example (with the first case representing a fully abstract type as described above):

External type definitions

```
type int =
  internal (* Internal#int *)
  ...
;;

type foc_diag ('a) =
  internal alias ('a * 'a)
  ...
;;

type list ('a) =
  internal
  | []
  | ( :: ) ('a, list ('a))
  ...
;;

type foc_record ('a, 'b) =
  internal  { hcode = int ; contents = ('b * 'a) }
  ...
;;
```

The first definition creates a type `int` that is fully abstract. Obviously, this one will be used magically by the internals of the compiler, so no need to have external definitions to manipulate its representation, this will be built-in in the compiler. Consult 2.6.1 for a discussion about this mechanism.

The second definition creates a parametrised type `foc_diag` that will internally be compatible with a pair of elements of the same type.

The third definition creates a parametrised type `list` that is made of 2 value constructors: `[]` with no argument and `::` with 2 arguments (not a tuple). Hence this corresponds to an internal representation of a sum type. We can see here how to define the lists like they are in Coq and OCaml to directly map them onto those of Coq and OCaml, without making any internal hack inside the compiler. We could have defined our custom lists by a regular type definition like

```
type list ('a) =
  | []
  | ( :: ) ('a, list ('a))
;;
```

but we wouldn't have been able to make so that constructors directly map onto their counterparts in Coq and OCaml (in effect, these constructors being non-regular identifiers, they would have been "stringified" like we saw in 2.1.4).

The fourth and last definition creates a parametrised type `foc_record` that is a record type with 2 fields: `hcode` and `contents`.

So, a question: but why to use external definitions to create such types ? It should be possible to define these sum and record types by a regular definition ! The answer, partially given about the `list` example is that this allows to control manually and explicitly the mapping of the type and of its components (fields and value constructors) into the target languages. We will see a clear example once we have ended the case of `foc_diag`.

To summarize, the `internal` clause controls how the type is seen on FoCaLize's side.

The `external` clause

The external clause tells how the **type constructor** (not it's components like fields, value constructors : this will be done by the extra `and ...` clauses described below in 2.6.1) must be **defined** into the target languages. In other words, what to emit to define the type constructor in the target languages.

Like in any `external clause`, we have an enumeration of mapping "language $\mapsto$ external code": we call this an "external expression". Languages can syntactically be `coq`, `caml` or a string for other languages not internally handled by the FoCaLize compiler. External code is an arbitrary string enclosed by `{*` and `*}` that will be emitted verbatim at code generation pass when the type constructor will have to be defined.

At code generation time, the shape of the definition in OCaml will look like: `type` followed by the enumeration of polymorphic type variables "`_focty_`" + type's name = followed by the verbatim copy of the external code. The polymorphic type variables are named as usual in OCaml. If there are several then they comma-separated and enclosed between parentheses. Their name are `, 'a`, then `, 'b`, then `, 'c` and if there are more than 26, it goes on with `'aa`, `'ab` etc.... In fact it is like printing their numbers in base 26, using lowercase letters as digits.

In Coq, it will look like: `Definition` followed the the type's name + "`_t`" followed by the enumeration of the polymorphic type variables `:=` followed by the verbatim copy of the external code. The polymorphic type variables are named by "`__var_`" + "a", then "b", then "c". Like for OCaml the suffix is the variable's number written in base 26. Hence the name of the 27^{th} variable will be `__var_ab`. In Coq we must explicitly annotate variables with the type `Set`.

Hence, with the definitions we started in our example, the beginning of the generated code in OCaml and Coq will look like:

OCaml code for external definitions

```
type _focty_int = ...
type 'a _focty_foc_diag = ...
type 'a _focty_list = ...
type ('a, 'b) _focty_foc_record = ...
```

Coq code for external definitions

```
Definition foc_diag__t (__var_a : Set) := ...
Definition list__t (__var_a : Set) := ...
Definition foc_record__t (__var_a : Set) (__var_b : Set) :=
```

Continuing our previous example, we state the FoCaLize external type definitions:

External type definitions (2)

```
coq_require "coq_stubs" ;;

type int =
  internal (* Internal#int *)
  external
  | caml -> {* int *}
  | coq -> {* Z *}
;;

type foc_diag ('a) =
  internal alias ('a * 'a)
  external
  | caml -> {* ('a * 'a) *}
  | coq -> {* ((__var_a * __var_a)%type) *}
;;

type list ('a) =
  internal
  | []
  | ( :: ) ('a, list ('a))
  external
  | caml -> {* 'a list *}
  | coq -> {* (list __var_a) *} (* From the List module. *)
  ...
;;

type foc_record ('a, 'b) =
  internal  { hcode = int ; contents = ('b * 'a) }
```

```
  external
  | caml -> {* ('b, 'a) Ml_stubs.bbt_record *}
  | coq -> {* coq_stubs.foc_record ['b, 'a] *}
  ...
;;
```

In the first definition we see that to define the `int` **type constructor** in OCaml, we will emit "int" and in Coq, we will emit "Z". That's because we want the `int` **type constructor** to be mapped onto these existing types of the target languages.

The second definition shows that we want `foc_diag` to be defined as a pair in OCamland in Coq.

In the third definition, we say that lists are implemented in OCaml by its native lists and in Coq, by an existing data-type (the one provided my Coq's module "List").

Finally, in the last definition, we want to say that the type `foc_record` is implemented by a user-defined type. For instance, a record type in both OCaml and Coq. Due to the way the code is emitted we can't write directly in the external code a record definition in each of the target languages: it would be syntactically incorrect. So, instead, we say that the type `foc_record` is implemented by a type name given in a other source file (one file for OCaml and one for Coq). We must remember that the generated definition in the target languages will correspond to an alias (i.e. of the form `type` name = external-code in OCaml, and `Definition` name `:=` external-code in Coq), so we must put in the external code only things that will comply the target language's syntax. For instance, specifying for the Coq language a definition like:

```
type foc_diag ('a) =
  internal alias ('a * 'a)
  external
  | caml -> {* ('a * 'a) *}
  | coq -> {* Record : Type :=
                mk_foc_diag (alpha : Set) {
                  fst : alpha ;
                  snd : alpha } *}
;;
```

would lead to a syntactically incorrect garbage rejected by Coq like:

```
Definition foc_diag__t (__var_a : Set) :=
   Record : Type :=
                mk_foc_diag (alpha : Set) {
                  fst : alpha ;
                  snd : alpha } .
```

Attention: In the sample code in FoCaLize we are studying, when we extended it, we added a first line that reads **`coq_require`** `"coq_stubs";;` This is needed to make so Coq can see definitions of the source file Coq`_stubs.v` in which we wrote our stubs for Coq. Since the FoCaLize compiler doesn't analyse inside the external code, it can't see that there is a dependency for Coqonto this source file, hence can't add itself in the generated Coq file the `Require` directive. By adding this `coq_require` directive in the FoCaLize source code, we tell the compiler to add the corresponding `Require` directive in the generated Coq code.

Finally, one may note in this last definition that a FoCaLize `foc_record ('a, 'b)` is implemented by a `('b, 'a) Ml_stubs.bbt_record`. We inverted the parameters ! This means that so they will be in the generated definition (have a look in the sample generated code below) !

We now continue to inspect the generated code in both OCamland Coq:

OCaml code for external definitions (2)

```
type _focty_int =  int ;;
type 'a _focty_foc_diag =  ('a * 'a)  ;;
type 'a _focty_list =  'a list ;;
type ('a, 'b) _focty_foc_record =  ('b, 'a) Ml_stubs.bbt_record ;;
```

and:

Coq code for external definitions (2)

```
Require coq_stubs.

Definition int__t :=  Z.
Definition foc_diag__t (__var_a : Set) :=  ((__var_a * __var_a)%type) .
Definition list__t (__var_a : Set) :=  (list __var_a) .
Definition foc_record__t (__var_a : Set) (__var_b : Set) :=
  coq_stubs.foc_record __var_b __var_a .
```

To fully understand the mapping mechanism, we must now see the stubs written for each target language.

Stub file ml_stubs.ml

```
type ('c, 'd) bbt_record = {
  bbt_hashing_code : int ;
  bbt_contents : ('c * 'd)
} ;;
```

Stub file coq_stubs.v

```
Require Export ZArith. (* To have type Z. *)

Record foc_record (param_a : Set) (param_b : Set) : Type :=
  mk_foc_record {
    hc : Z;
    conts : ((param_a * param_b)%type)
}.
```

The extra `and` ... clauses

Finally, there are still two things to describe... We know how the type constructor must be seen internally, how it must be mapped onto target languages but we don't know yet on what to map the sum value constructors and the record fields names !

The extra `and` directive are here for this purpose and will be used only in case of sum or record type definitions. We now complete our external definitions:

External type definitions (3)

```
type list ('a) =
  internal
  | []
  | ( :: ) ('a, list ('a))
  external
  | caml -> {* 'a list *}
  | coq -> {* (list __var_a) *} (* From the List module. *)

and [] =
  | caml -> {* [] *}
  | coq -> {* List.nil *}

and ( :: ) =
  | caml -> {* ( :: ) *}
  | coq -> {* List.cons *}
;;

type foc_record ('a, 'b) =
  internal  { hcode = int ; contents = ('b * 'a) }
  external
  | caml -> {* ('b, 'a) Ml_stubs.bbt_record *}
  | coq -> {* coq_stubs.foc_record __var_b __var_a *}

  and hcode =
```

```
  | caml -> {* Ml_stubs.bbt_hashing_code *}
  | coq -> {* coq_stubs.hc *}

  and contents =
  | caml -> {* Ml_stubs.bbt_contents *}
  | coq -> {* coq_stubs.conts *}
;;
```

We must note that for `foc_record`, we mapped the record fields onto the related fields present in our stubs source files. Obviously, an external type definition must consistently map the type constructor, its field label or value constructor to a definition existing in the target language. For instance, if we mapped `foc_record` to a type `t1`, and its field labels to labels of a type `t2` or to field names that do not exist in the target language, then the compilation of the generated code would fail. It is important to see that the compilation process on FoCaLize's side wouldn't point out any error: however Coq or/and OCaml would complain.

This last set of clauses, since they are involved in how to map value constructors and record fields **when they are encountered** do not lead to any code when the type definition is emitted. So we do not have any more generated code related to the type definition to examine.

External type definitions and built-in types.

As stated before, external type definition are also used to setup the basic types internally known by the compiler :`int`, `unit`, `string`, `bool` and `char`. These definitions are in the file `stdlib/basics.fcl`. We could have fully hidden these definition by directly hard-coding these types and their components (value constructor for `unit`) in the pervasive environments. But this would have been doing manually a job the compiler knows how to make. So, instead, we used these external definitions. They magically work with the inner of the compiler because this one is made so that when it has to infer the type of an integer value, it will generate a type whose name is ...justly "int" ...exactly the name we gave in our external type definition. This means that internally, the compiler knows that an integer value has type "int", even if it doesn't know what is an integer. This especially means that if no external definition is given for `int`, the compiler is still able to infer the type of integer expressions. However, is the user writes "int", then he will be rejected because the scoper won't find any definition of this type constructor.

The case of `unit` is slightly different since the external definition also gives the (only) value inhabiting this type. So, the compiler doesn't have built-in mechanism to synthesise a type `unit`: to get this type, the only ways are either the user wrote "unit" or he wrote "()" and looking-up in the environment for the value constructor's type we will normally get `unit`. The difference with the case of `int` is really that in `unit`, the definition gives the values of the type although in `int`, they are not given by the definition but rather "hard-wired" in the compiler (it knows syntactically what is an integer).

Hence, considering this difference point, `int`, `char` and `string` are handled the same way, and `unit` and `bool` are handled the same way.

2.6.2 External value definitions

External value definitions follow the same idea than types, they have an `internal` and an `external` clause. The `internal` clause deals with how the compiler must consider the value during its analyses: it deals with the value's **type**. The `external` clause deals with how the compiler must map this value at code generation time in the target languages.

Hence, an external definition is compound of the `internal` clause followed by a type expression and tne `external` clause followed by an "external expression". For instance:

External value definitions

```
let int_max =
  internal int -> int -> int
  external
  | caml -> {* Ml_builtins.bi__int_max *}
  | coq -> {* coq_builtins.bi__int_max *}
```

```
;;

let ( || ) =
  internal bool -> bool -> bool
  external
  | caml -> {* Ml_builtins.bi__or_b *}
  | coq -> {* coq_builtins.bi__or_b *}
;;

let physical_equal =
  internal 'a -> 'a -> bool
  external
  | caml -> {* Ml_builtins.bi__physical_equal *}
  | coq -> {* coq_builtins.bi__syntactic_equal __var_a *}
    (* Wrong, but don't know how to do better. *)
;;
```

Such a definition is simply compiled into a definition having the same name in the target language than in FoCaLize (except in case where stringification is required like in `||`) whose body is the verbatim copy of the external expression corresponding to the target language. In OCaml, we do not need to make the type of the definition explicit. In Coq, we need to and also, as usual, need to handle the explicit polymorphism with the extra arguments of type `Set` to represent the polymorphic type variables (like in `physical_equal`).

For the above FoCaLize sample code, we then simply get the generated code:

OCaml code for external value definitions

```
let int_max =  Ml_builtins.bi__int_max ;;
let _bar__bar_ =  Ml_builtins.bi__or_b ;;
let physical_equal =  Ml_builtins.bi__physical_equal ;;
```

and:

Coqcode for external value definitions

```
Let int_max : int__t -> int__t -> int__t :=  coq_builtins.bi__int_max.
Let _bar__bar_ : bool__t -> bool__t -> bool__t := coq_builtins.bi__or_b.
Let physical_equal (__var_a : Set) : __var_a -> __var_a -> bool__t :=
  coq_builtins.bi__syntactic_equal __var_a .
```

Hence, with this code generation model, since the identifier bound to an external definition receives, in the generated code, an effective definition, we do not need to replace each occurrence of this identifier of the FoCaLize source code by its related external expression in the generated source code. This makes the code generation simpler since when we "see" the identifier in the FoCaLize source code, we generate exactly the same identifier in the generated source code. Otherwise we should have reminded for each identifier whether it is bound to an external stuff to see by what to replace its occurrence. Too heavy, and may be could lead to incorrect syntactic forms in the generated source !

Chapter 3

Compiler sources architecture

3.1 focalizec source tree

The compiler source tree is split into several directories and files described below. The root of the compiler sources is located in `focalize/focalizec`. All directories and files given below are relative to this root. In some of the directories is a sub-directory called `odoc`. It can be safely ignored and is only used when invoking verb+make doc+ to store the ocamldoc output (HTML documentation extracted from the source code of FoCaLize). In the below enumeration (d x) stands for "directory" whose content is at x nesting level of the root (i.e. of `focalize/focalizec`) and (f) stands for "file".

- `Makefile` (f) : The toplevel Makefile building the compiler, the standard library, the extra libraries, the documentation generator, the FoCaLize documentations and the contributions. This Makefile indeed trigger the build process in all the deeper directories.
- `Makefile.common` (f) : Defines once for all the implicit rules and suffixes that will be used in deeper Makefiles.
- `.config_var` (f) : File generated during the "configure" process (invoked by the file `configure` below and that records the various things among installation paths, installed commands and tools, ...
- `Makefile.config` (f) : Defines once for all the commands according to the information the "configure" process recorded. It especially include the `.config_var` file that kept trace of what the "configure" process defined.
- `TAGS` (f) : Documentation file where developers are invited to note all the tags added to the development tree under CVS, with a description of the status of the development tree when the tag was set (or the reason why to put a tag).
- `TODO` (f) : More or less describes points that are still pending.
- `configure` (f) : Script used to detect available commands and tools, to ask the user where to install FoCaLize components, ... Parts of its output is stored in the above file `.config_var`.
- `doc_src` (d 1) : The directory containing all the documentations at destination of the users.
  - `Makefile` (f) : Makefile triggering the documentations build.
  - `html` (d 2) : The FoCaLize WEB site in HML format.
    * `Includes` (d 3)
      · `aftertitle-eng.html` (f)
      · `aprestitre-fra.html` (f)

        - avanttitre-fra.html (f)
        - basdepage-fra.html (f)
        - beforetitle-eng.html (f)
        - bottomofpage-eng.html (f)
        - copyright-eng.html (f)
        - copyright-fra.html (f)
        - doctype (f)
        - endofpage-eng.html (f)
        - findepage-fra.html (f)
        - hautdepage-fra.html (f)
        - htmlc-version.html (f)
        - maquette-eng.html (f)
        - maquette-fra.html (f)
        - powered_by_caml.html (f)
        - topofpage-eng.html (f)
    * Makefile (f)
    * Makefile.html (f)
    * images (d 3) : Directory containing images for the WEB site.
        - focal_picture.jpg (f) : A niiiice 3D picture done with Povray ☺.
- man (d 2) : Directory containing the "man" manual.
    * Makefile (f)
    * focalizec.env (f)
    * focalizec.man (f) : "man" for focalizec.
    * focalizedep.man (f) : "man" for focalizedep.
- tex (d 2) : Documentations in LaTeX format.
    * Makefile (f)
    * refman (d 3) : Directory containing the reference manual.
        - Makefile (f)
        - basic_concepts.tex (f)
        - bibli.bib (f) : Contains bibliography references.
        - building_species.tex (f)
        - compiler_err_msgs.tex (f) : About the focalizec compiler error messages.
        - compiler_opts.tex (f) : About the focalizec compiler command line options.
        - constructs_syntax.tex (f)
        - doc_gen.tex (f) : About the FoCaLize automated documentation generation (option focalize-doc of focalizec).
        - fullpage.sty (f)
        - glimpse.tex (f) : Short presentation of the language to give a first an rapid taste.
        - header_html_snapshot.ps (f)
        - introduction.tex (f)
        - lexical_conventions.tex (f) : About FoCaLize lexical conventions, description of the tokens and syntax.
        - macros.hva (f)
        - macros.tex (f) : LaTeXmacros used in the reference manual.
        - mathml_snapshot.ps (f)

        - more_on_meths.tex (f)
        - motivations.tex (f)
        - output.tex (f) : About what the focalizec compiler generates.
        - pres_requir.tex (f)
        - proofs.tex (f)
        - refman.html (f)
        - refman.hva (f)
        - refman.pdf (f)
        - refman.tex (f) : Entry point of the reference manual.
        - syntaxdef.hva (f)
        - syntaxdef.sty (f)

- src (d 1) : Root of the sources of focalizec and focalizedep.
  - attic (d) : Contains miscellaneous interesting parts of code that have been written one day, that are not used anymore, but that we didn't want to trash in case it could serve in the future.
    * ast.mli, lib_pp.ml, lib_pp.mli, misc.ml, new2old.ml, new2old.mli, oldsourcify.ml, oldsourcify.mli, printer.ml, printtree.ml, string_search_stuff.ml
    * May be more...
  - basement (d 2) : Contains the early basic bricks of the compiler.
    * Makefile (f) : Trigger build in this directory.
    * configuration.ml (f) : Manages version number (and by side effect the -help option output text) and the command line options flags.
    * configuration.mli (f)
    * files.ml (f) : Contains primitives dealing with files, taking into account search path, files suffixes, read/write object files, ...
    * files.mli (f)
    * handy.ml (f) : Various general functions, on lists, pretty-printers, font setting...
    * handy.mli (f)
    * installation.ml (f) : File automatically generated by the "configure" process and record as OCaml source, the various installation paths and commands the focalizec compiler need to know when dealing with FoCaLize source files.
    * location.ml (f) : Defines the type of "locations", i.e. point in a source file and gives primitives to work with them.
    * location.mli (f)
    * miscHelpers.ml (f) : Mostly contains 1 function used during several passes to bind formal names to their types from a type scheme and a list of formal names. May be could somewhere else to save one file...
    * miscHelpers.mli (f)
    * parsetree.mli (f) : The description of the AST.
    * parsetree_utils.ml (f) : General utilities to process the AST.
    * parsetree_utils.mli (f)
    * types.ml (f) : The description of the types structure and operations to work with them. Because types are a complex structure with invariants, they are exported as opaque to prevent breaking these invariants. Hence, any function dealing with the intimate representation of a type must be in this file since his is the only location where this representation is visible.
    * types.mli (f) :

- `ccodegen` (d) : Dedicated to the C code generation back-end. Maintained by Julien Blond and is not currently included in FoCaLize.
  * ... not stable.
- `commoncodegen` (d 2) : Contains processing common to all target languages that is performed in any case before emitting the target code.
  * `Makefile` (f) :
  * `abstractions.ml` (f) : Performs the synthesis of λ-lifted things, summarising def and decl-dependencies and dependencies on collection parameters methods. It especially performs the final (complete) computation of dependencies on collection parameters, applying the various completion rules on the dependencies found previously by the rules [TYPE] and [BODY].
  * `abstractions.mli` (f)
  * `context.mli` (f) : The structure of the "context", i.e. the structure inductively passed to each function during this pass and that group into one single record type all the information needed by the various functions. This a handy way to prevent from having numerous parameters to pass each time to the functions.
  * `externals_generation_errs.ml` (f) : Exceptions that can be raised when generating code for "external" definitions, i.e. definitions that are not written in native FoCaLizelanguage and used to interface with other programming languages.
  * `externals_generation_errs.mli` (f)
  * `minEnv.ml` (f) : Compute the "Coqminimal typing environment".
  * `minEnv.mli` (f)
  * `misc_common.ml` (f) : Various type definitions and functions used in several points during this pass.
  * `misc_common.mli` (f)
  * `recursion.ml` (f) : Deals the the recursive functions processing.
  * `recursion.mli` (f)
  * `visUniverse.ml` (f) : Compute the "visible universe".
  * `visUniverse.mli` (f)
- `contribs` (d 2) : Contains contribution source codes written by users that are not developers of the compiler and its framework.
  * `Makefile` (f)
  * `automata` (d 3) : Hierarchical automata. Maintained by Philippe Ayrault.
    · `Makefile` (f)
    · `gen_def.fcl` (f)
    · `main.fcl` (f)
    · `request.fcl` (f)
    · `switch_automata.fcl` (f)
    · `switch_recovery_automata.fcl` (f)
    · `switch_recovery_normal_automata.fcl` (f)
    · `switch_recovery_reverse_automata.fcl` (f)
  * `utils` (d 3) : Various utilities. Maintained by Philippe Ayrault.
    · `Makefile` (f)
    · `pair.fcl` (f)
    · `peano.fcl` (f)
  * `voter` (d 3) : Model of a generic voter. Maintained by Philippe Ayrault.
    · `Makefile` (f)

· etat_vote.fcl (f)
· main.fcl (f)
· num_capteur.fcl (f)
· value.fcl (f)
· vote.fcl (f)

– coqcodegen (d 2) : Contains Coq code generation back-end.
  * FOCAL_COQ_MAPPINGS (f) : Short (incomplete) description of how FoCaLize names are mapped onto Coq names in the generated source code.
  * Makefile (f)
  * main_coq_generation.ml (f) : Entry point of Coq code generation. Start code generation for a compilation unit, deals with toplevel entities (let, theorems, expressions and type definitions) and triggers processing of species, collections.
  * rec_let_gen.ml (f) : Manages parts of the code generation for recursive functions with stuff dedicated to Coq.. Especially, generates the termination lemmas and transforms arguments of a recursive into a tuple.
  * rec_let_gen.mli (f)
  * species_coq_generation.ml (f) : Main part of code generation for species and collections.
  * species_coq_generation.mli (f)
  * species_record_type_generation.ml (f) : Deals with the generation of the record type representing species and collections. Also generates code for expressions.
  * species_record_type_generation.mli (f)
  * type_coq_generation.ml (f) : Deals with code generation for (toplevel) type definitions.
  * type_coq_generation.mli (f)
– devdocs (d 2) : The current documentation explaining the compiler's architecture and mechanisms.
  * foc2ocaml.tex, pending.txt (f) : Oldies. Should disappear.
  * legacy.tex (f) : The main LaTeXfile. Compile it to get the current documentation.
  * macros.tex (f) : Miscellaneous macros to make life easier ☺.
  * mathpartir.sty : Inference rules package.
  * phd_changes.tex : Things we changed, corrected, enhanced since Virgile Prevosto's PhD Thesis.
– docgen (d 2) : Source for focalizec automated documentation generation (-focalize-doc option of the compiler) as XML files.
  * Makefile (f) :
  * doc_lexer.mll : Lexer to scan "@"-markups in a FoCaLize source code.
  * env_docgen.ml : Contains the documentation environment mechanisms. This environment maps methods names onto some optional MathML and LaTeX code.
  * env_docgen.mli (f)
  * focdoc.css (f) : FoCaLizeDoc-XML style sheet.
  * focdoc.dtd (f) :
  * focdoc.rnc (f) :
  * focdoc.xsd (f) :
  * focdoc2html.xsl : Transforms some FoCaLizeDoc-XML into HTML + MathML.
  * focdoc2tex.xsl (f) : Transforms some FoCaLizeDoc-XML into LaTeX.
  * main_docgen.ml (f) : Engine extracting information from a FoCaLize source code to produce FoCaLizeDoc-XML.
  * main_docgen.mli (f)

    * mmlctop2_0.xsl (f) : Transforms HTML + MathML into HTML.
    * proposition.xsl (f) : : Processing of logical expressions from FoCaLizeDoc-XML to HTML + MathML.
    * proposition2tex.xsl (f) : Processing of logical expressions from FoCaLizeDoc-XML to LaTeX.
    * utils_docgen.ml (f) : Various helpers used for XML production.
    * utils_docgen.mli (f)
- extlib (d 2) : Libraries higher level than the basic standard library. Especially contains formal calculus structures.
    * Makefile (f)
    * access_control (d 3) : Access control policies. Maintained by Lionel Habib and Mathieu Jaume.
        · Makefile (f)
        · access_control.fcl (f)
        · ensembles_finis.fcl (f)
        · hru.fcl (f)
        · rbac.fcl (f)
        · ticket.fcl (f)
        · tm (d 4)
        · graph.ml (f)
        · trust_management.fcl (f)
        · unix.fcl (f)
    * algebra (d 3) : Formal calculus library. Maintained by Renaud Rioboo.
        · Makefile (f)
        · additive_law.fcl (f)
        · arrays.fcl (f)
        · arrays_externals.v (f)
        · big_integers.fcl (f)
        · constants.fcl (f)
        · integers.fcl (f)
        · iterators.fcl (f)
        · multiplicative_law.fcl (f)
        · parse_poly.fcl (f)
        · polys_abstract.fcl (f)
        · product_structures.fcl (f)
        · quotient_structures.fcl (f)
        · randoms.fcl (f)
        · randoms_externals.ml (f)
        · randoms_externals.v (f)
        · rings_fields.fcl (f)
        · small_integers.fcl (f)
        · weak_structures.fcl (f)
        · weak_structures_externals.ml (f)
        · weak_structures_externals.v (f)
- focalizedep (d 2) : The dependencies generator to create Makefiles.
    * Makefile (f)

    * directive_lexer.mll (f) : Lexer to scan open and use directives in a FoCaLize source.
    * make_depend.ml (f) : Source of the generator.

- mlcodegen (d 2) : Contains OCaml code generation back-end.
    * FOCAL_ML_MAPPINGS (f) : Short (incomplete) description of how FoCaLize names are mapped onto OCaml names in the generated source code.
    * Makefile (f)
    * base_exprs_ml_generation.ml (f)
    * base_exprs_ml_generation.mli (f)
    * main_ml_generation.ml (f)
    * main_ml_generation.mli (f)
    * misc_ml_generation.ml (f)
    * misc_ml_generation.mli (f)
    * species_ml_generation.ml (f)
    * species_ml_generation.mli (f)
    * type_ml_generation.ml (f)
    * type_ml_generation.mli (f)
- parser (d 2) : Contains the lexical and syntactic analysers to process FoCaLize source code.
    * Makefile (f)
    * dump_ptree.ml (f) : Dumps the AST structure of a file in raw text mode. Used for debugging purpose.
    * dump_ptree.mli (f)
    * lex_file.ml (f) : Initial attempt to have a small "main" to check the behaviour of the lexer on a file at the early stages of development. This is not used anymore.
    * lexer.mll (f) : The lexer description in ocamllex.
    * lexer.spec (f) : Trace of thoughts we had all along the time. Initially contained ideas about the language syntax, (at lexical level) to make it smooth.
    * parse_file.ml (f) : Initial attempt to have a small "main" to check the behaviour of the parser on a file at the early stages of development. This is not used anymore.
    * parse_file.mli (f)
    * parser.mly (f) : The parser description in ocamlyacc.
    * parser.spec (f) : Trace of thoughts we had all along the time. Initially contained ideas about the language syntax, (at syntactic level) to make it smooth. Next other ideas were added, beyond syntax, more oriented toward the AST structure.
    * sourcify.ml (f) : Dumps the AST structure of a file in FoCaLize syntax. Gives back a source code from a parsed code. This is mostly useful for debugging purpose. May be used bu customers to pretty-print their source, but I don't recommand. This more specially serves to report pieces of code in a good-looking form in error messages. Each time we need to tell the user something involving a piece of code, we pretty-print it with functions provided here.
    * sourcify.mli (f)
    * sourcify_token.ml (f) : Not used anymore. Was used to test the lexer at the early stages of development. It was used to print the found token.
    * test (d 3) : Oldies originally used to test the lexer/parser in the early times. No used anymore.
        · sets_orders.fcl (f) : First translation of a file of the standard library used ti test the lexer/-parser. Not used anymore.
- scoper (d 2) : Old stuff. Was discarded a long time ago. The directory is empty by the way.

- stdlib (d 2) : The standard library. Most of these sources are maintained by Renaud Rioboo. A few ones (basics and built-ins) are low-level code close to the compiler and maintained by the core compiler development team.
  * Makefile (f)
  * basics.fcl (f) : Basic bricks and internal-related stuff. Contains the primitive type definitions and some very basic functions for FoCaLize.
  * coq_builtins.v (f) : Basic bricks and internal-related stuff especially dedicated to Coq code.
  * generic_proof_cases.v (f) : Contains built-in stuff for Coq about built-in lists and some tactics.
  * lattices.fcl (f)
  * ml_builtins.ml (f) : Basic bricks and internal-related stuff especially dedicated to OCaml code.
  * orders.fcl (f)
  * orders_and_lattices.fcl (f)
  * products.fcl (f)
  * quotients.fcl (f)
  * sets.fcl (f)
  * sets_externals.ml (f) : External definitions to link for OCaml related to sets.fcl.
  * sets_orders.fcl (f)
  * sets_orders_externals.ml (f) : External definitions to link for OCaml related to sets_orders.fcl.
  * strict_orders.fcl (f)
  * sums.fcl (f)
  * wellfounded.fcl (f) : Define well foundation, termination orders for recursive function definitions for Coq.
  * wellfounded_externals.v (f) : External definitions to link for OCaml related to wellfounded.fcl.
- toplevel (d 2)
  * Makefile (f)
  * exc_wrapper.ml (f) : Firewall designed to recover all the exceptions that can be raised during a source compilation and issues the related error message. Roughly, this code wraps the main call to the compilation engine.
  * focalizec.ml (f) : The compiler entry point.
  * focalizec.mli (f)
  * fodump.ml (f) : For debug only, dumps (partially) the content of a FoCaLize object file (".fo" file).
- typing (d 2) : Performs the type-checking of the language, normal form of species, checks well-formation of species, compute def and decl-dependencies, initiate computation of dependencies on collection parameters' methods, resolves inheritance.
  * Makefile (f)
  * ast_equal.ml (f) : Implements equality test between logical expressions (indeed, between expressions since logical statements fully embed expressions).
  * ast_equal.mli (f)
  * depGraphData.mli (f) : Defines the structure of the dependencies graph used to compute well-formation from def and decl-dependencies.
  * dep_analysis.ml (f) : Handles the dependencies analyses on methods of Self, i.e. def and decl-dependencies.
  * dep_analysis.mli (f)

    * `env.ml` (f) : Structure of the environments used all along the compilation. Contains a generic environment structure, the specialised scoping environment the specialised typing environment, the specialised OCaml code generation environment and the specialised Coq code generation environment. All these environments are instantiations of the generic environment structure. Also contains data-structures used from type-checking up to code generation to record the various information about analysed species, collections, methods, types, ... Some of these data-structures will appear in later passes, picked-up from these environments.
    * `env.mli` (f)
    * `infer.ml` : Core engine performing type-checking, inheritance resolution, normal form of species, check of well-formation, def and decl-dependencies computation and triggers the initial computation of dependencies on collection parameters' methods.
    * `infer.mli` (f)
    * `param_dep_analysis.ml` (f) : Handles the initial computation of dependencies on collection parameters' methods. The computed dependencies will have to be completed later in the next pass.
    * `param_dep_analysis.mli` (f)
    * `scoping.ml` (f) : Performs scoping analysis.
    * `scoping.mli`
    * `substColl.ml` (f) : Performs substitution of a collection (name/carrier) by another one or `Self` in an AST.
    * `substColl.mli` (f)
    * `substExpr.ml` (f) : Performs substitution of an expression (i.e. `Parsetree.expr`) by another in an AST.
    * `substExpr.mli` (f)

- `tests` (d 1) : Various little tests we wrote to shake the compiler or to ensure that some particular features/processing required by the compiler were really working. This code is most of the time meaningless and only design to target a particular point of the compilation process. The naming scheme is: a source file name starting by "ko__" must lead to an error (an error message, not a failure of the compiler), a source file name starting by "ok__" must be accepted by the compiler and lead to effective code.
  - `Makefile` (f)
  - `ko__bad_self_use.fcl` (f)
  - `ko__param_toy.fcl` (f)
  - `ko__test_error.fcl` (f)
  - `ko__test_rec.fcl` (f)
  - `ok__baby_toy.fcl` (f) : A bit of everything, without any order.
  - `ok__baby_toy_externals.ml` : External code to link for OCaml related to `ok__baby_toy.fcl`.
  - `ok__caveat.fcl` (f) : Illustrates the pitfall of sum type constructors that have **1** argument that is a tuple or **several** arguments but are hardly different from the syntax point of view.
  - `ok__coll_outside.fcl` (f) : Example showing how now use a collection defined outside the current one (i.e. the used one not a parameter) in a collection or a species.
  - `ok__definition_72_rule_PRM.fcl` (f) : Illustrates the need for rule PRM in definition 72 page 153 in Virgile Prevosto's PhD.
  - `ok__in_example.fcl` (f) : Example showing how now use the `in` parameter without allowing their type to be anymore a "ml type".
  - `ok__in_example2.fcl` (f) : Same purpose than for `ok__in_example.fcl`.

- `ok__list.fcl` (f) : Example showing how to make lists as species in FoCaLize. It also illustrates the need for the extra unit argument to the collection generator "`collection_create`" in case it has no arguments, what would prevent OCaml from generalising the type of this generator. This example must pass in OCaml but is not yet accepted in Coq.
- `ok__multiple_inherit.fcl` (f) : Used to track the provenance of methods in case of multiple inheritance.
- `ok__need_inspect_self.fcl` (f) : Example showing how now use the need to inspect the structure of `representation` while computing the "`used_species_parameters_ty`" in the case where a method has a def-dependency on `Self`.
- `ok__need_re_ordering.fcl` (f) : Example showing the need of the final re-organisation of methods due to the collapsing procedure of properties and `proof of`s.
- `ok__odd_even.fcl` (f) : Example showing simple mutually recursive methods that must pass for OCaml code generation. Note that Coq code generation does not yet handle mutual recursion.
- `ok__phd_def_deps.fcl` (f) : Example of Virgile Prevosto's Phd, section 3.9.6, page 56. Illustrates the need for the erasing procedure.
- `ok__phd_meths_gen.fcl` (f) : Example from Virgile Prevosto's Phd, section 6.4.3, page 115.
- `ok__phd_sample.fcl` (f) : The initial example given in Virgile Prevosto's Phd, section 2.2.2 starting page 14.
- `ok__scoping_tricky.fcl` (f) : Example showing scoping occurrences of a same name in case where this name is not recursively defined. Combines the stuff with late-binding.
- `ok__term_measure.fcl` (f) : Is designed to be a successful use of termination proofs. Still doesn't pass.
- `ok__toplevel_odd_even.fcl` (f) : Example showing simple mutually recursive toplevel functions that must pass for OCaml code generation. Note that Coq code generation does not yet handle mutual recursion.
- `ok__torture_params.fcl` (f) : Example showing how to torture parametrised species. It exhibits various non-trivial cases that must pass and lead to effective OCamlCoq code. These examples make heavy use of `in`, `is` parameters and inheritance to shake the compiler's instanciation of parameters during inheritance.

3.2 Other tools

Aside focalizec itself and its library, 2 others tools explicitly dedicated to the FoCaLize packages exist.

3.2.1 zenon

Maintained by Damien Doliger, he is the man in the place to describe ☺.

3.2.2 zvtov

Maintained by Damien Doliger, he is the man in the place to describe ☺.

3.3 Passes and directories

Each directory described above roughly corresponds to one pass of the compiler except the directory `src/basement` which contains basic stuff used a bit everywhere. The order of the tasks performed at compile-time can be mapped on the source directories as follows:

1. `src/toplevel` : Entry point of the compiler, performing I/O and administrative tasks to initiate the compilation process.

2. `src/parser` : Lex and parse the source code

3. `src/typing` : Scopes the parsed source (i.e. the primary AST, hence leading to a "scoped" AST), then type-check the scoped AST (hence leading to a "typed" AST), resolves inheritance, compute def and decl-dependencies, ensure well-formation and performs a first partial computation of dependencies on collection parameters' methods.

4. `src/commoncodegen` : Compute final abstractions (i.e. λ-liftings) due to def and decl-dependencies and dependencies on collection parameters' methods. This process is in fact done once for OCaml code generation and once for Coq code generation (obviously if code generation for each language is enabled via the command line options of the compiler). The OCaml code production is done first is the Coq one is also requested.

5. `src/mlcodegen` : Emits the OCaml code if requested.

6. `src/commoncodegen` : As describe above but for Coq code generation if requested.

7. `src/coqcodegen` : Emits the Coq code if requested.

Libraries, standard (`src/stdlib`) and other libraries (`src/extlib`) are compiled once, when the compiler is built. Then they get installed according to the user's configuration and not touched anymore. However, when compiling a FoCaLize program, they will be used if references are done to entities they host.

During the compilation of the FoCaLize source, only object files of the libraries are needed. OCaml and Coq files of the libraries will be used once focalizec has output the generated code, in order to produce the whole OCaml executable or Coq model.

Chapter 4

Lexing / parsing

4.1 Lexing

The lexing process is performed as usual, by a lexer created by ocamllex. The description of the lexer is located in `src/parser/lexer.mll`. The structure of this file is like any regular lexer for a realistic language.

The keywords of the language is stored in a hash-table to prevent the automaton from being too big. Each time a stream of alpha-numeric characters that can be either an identifier or a keyword is found, we lookup in this hash-table. If the word make of this stream of characters belongs to the hash-table, then we return the corresponding token, else we return an identifier based on the form of the word (capitalised or not).

As any lexer, each time it is called (by the lexer), it returns one and only one lexem.

4.2 Parsing

The lexing process is performed as usual, by a lexer created by ocamlyacc. The description of the parser is located in `src/parser/parser.mly`. The structure of this file is like any regular parser for a realistic language. The parser doesn't try to implements recovery on error, hence, it describes the syntax of the language without any addition (conversely to OCaml or gccś parsers).

The important rule applied in the parser is that at each AST node it creates, it record the corresponding location (extend) in the source file and the corresponding (optional) documentation (i.e. special comments kept in the AST). This mechanism is performed by a set of basic functions with names derivated from `mk` (`mk_loc`, `mk_doc_elem`, `mk_doc`, `mk_local_ident`, `mk`, ...). This especially means that one never must create an AST node by hand. Any created AST node must be created via these basic functions.

The parser, returns a complete AST corresponding to the parse of the whole submited source file. It currently doesn't have an entry rule returning only an AST for one definition. This especially means that the result of the parsing is always an AST node of the form `Parsetree.file`.

Chapter 5

The environments structure

All along the compilation pass, we need to keep trace of various information about compiled entities (methods, types, identifiers, ...). This is done by a kind of association list, mapping a name of entity onto an information.

Depending on the compilation passe, the nature of the recorded information is different. However, the basis of name mapping is always the same. For this reason, instead of building several times the name mapping system, we decided to create a structure of generic environments which can be instantiated by specific information to record for each compilation pass.

Environments implementation is located in the file `src/typing/env.ml`.

5.1 The generic environment

5.1.1 Environment "recording" structure

As previously stated, the core of an environment is to map names onto information. In FoCaLize names can designate several types of entities:

- Sum type value constructors (e.g. `None`, `Some`).
- Record type field labels.
- Types constructors (e.g. `int`, `option`).
- Values.
- Species.

To ensure there is no conflicts between these different types of entities, i.e. to be allowed to give a same name to a record types field and to a type constructor or a value identifier, en environment will keep separate name-spaces for each type of entity.

As we stated before, an environment maps a name (the **key**) onto an information (the **value**). Hence, the **key** of an environment being a name, it will be mapped onto a `Parsetree.vname`. Since we want environments to record different values (to be polymorphic in fact), the **value** will be a different type variable for each name-space.

Hence, the generic environments have type:

To create a module system, we need to wrap this core structure inside a module providing access to this structure, i.e. search the **value** bound to a **key**: `find` and add binding of a **key** to a **value** : `add`. Obviously, we need these 2 functions for each name-space.

```
type ('a, 'b, 'c, 'd, 'e) generic_env = {
  constructors : (Parsetree.constructor_name * 'a binding_origin) list ;
  labels : (Parsetree.vname * 'b binding_origin) list ;
  types : (Parsetree.vname * 'c binding_origin) list ;
  values : (Parsetree.vname * 'd binding_origin) list ;
  species : (Parsetree.vname * 'e binding_origin) list
}
```

5.1.2 Principle of making an environment

We would like to directly create a functor taking as argument a module whose signature implements the types of bound information in each name-space for an environment and gives back a module providing the functions `find` and `add` for each name-space, in this environment. The idea is then to apply this functor to a set of various modules to get the scoping environment, the typing environment, the OCaml and the Coq code generation environments.

This simplistic view fails because of the notion of "modules", i.e. in FoCaLize compilation units.

In effect, since compilation units host definitions, they can be seen as modules. Hence we need an extra function to search inside "modules". This will especially serve to look-up names qualified by the #-notation . For example the identifier `basics!int` is bound in the module `basics` by the name `int`. This `find_module` function doesn't need to be visible in our generic environments since it only serves for internal purpose. It must conceptual take a module name, an environment and return the "sub"-environment composed of all the definitions hosted by the module. This `find_module` function is then dependant of the type of the the name-space in which to look-up in the managed environment. On must have 1 `find_module` for when we look-up for values, one for when we look-up for types, one for sum type constructors, one for record field labels, etc... The problem is that "creating" the "sub"-environment from the in formations contained in a module depend on the structure of information recorded in the environment. This means that we can't consider these in formations as anymore polymorphic since we know to know their structure to build the "sub"-environment.

For this reason, we proceed in two steps, i.e. two functors. One, `EMAccess`), will provide the "way to access" the information kept as **values** in the environment. The second (`Make`) will take a module produced by the previous one and glue the structure of the environment (i.e. in fact the association list) with the access primitives provided.

5.1.3 `EMAccess` : accessing functions for an environment

module is required to have the signature:

The fields type definitions `constructor_bound_data`, `label_bound_data`, `type_bound_data`, `value_bound_dat` and `species_bound_data` respectively represent the information the environment maps onto names of sum type value constructors, record field labels, type constructors, values and species.

Next come the `find_module` function that looks for a "module" name (the `Types.fname option`, we will see later the reason of the `option`) in the current environment and returns the environment composed of all the definitions present in this module (compilation unit). For technical reason, some extra arguments are passed to this function and we will explain their presence later.

Next come the `pervasives` value that is the initial (may non-empty) environment containing built-in bindings that are needed. Note that this environment may be the empty environment (i.e. containing no binding) is nothing has to exist in a built-in way.

Next come a the function `make_value_env_from_species_methods` whose purpose can be understood once the module system is understood. Basically, this function must create an environment, populating the values name-space with information extracted from the methods of a species.

Finally, the function `post_process_method_value_binding` can also be understood via the module system mechanism. This function takes a collection name, an information bound to a value and is applied on this information. This function is called by `find_value` to post-process the information bound to a value name if needed.

```
module type EnvModuleAccessSig =
  type constructor_bound_data
  type label_bound_data
  type type_bound_data
  type value_bound_data
  type species_bound_data
  val find_module :
    loc: Location.t -> current_unit: Types.fname -> Types.fname option ->
      (constructor_bound_data, label_bound_data, type_bound_data,
       value_bound_data, species_bound_data)
      generic_env ->
        (constructor_bound_data, label_bound_data, type_bound_data,
         value_bound_data, species_bound_data)
      generic_env
  val pervasives : unit ->
    (constructor_bound_data, label_bound_data, type_bound_data,
     value_bound_data, species_bound_data)
      generic_env
  val make_value_env_from_species_methods :
    Parsetree.qualified_vname -> species_bound_data ->
      (constructor_bound_data, label_bound_data, type_bound_data,
       value_bound_data, species_bound_data)
        generic_env
  val post_process_method_value_binding :
    Parsetree.qualified_vname -> value_bound_data -> value_bound_data
  end
```

Table 5.1: Sample code for EMAccess signature

For sake of simplicity, we decide to group the types definitions representing information bound for each name-space of an environment in a module. This is not mandatory but allows a better structure of the code.

5.1.4 `Make`-ing the environment

This functor finally create the final environment structure, linking the access functions provided by its argument (of interface `EMAccess`) and the basic structure of generic environment (`('a, 'b, 'c, 'd, 'e) generic_env`).

This functor must create a module whose signature contains an abstract type representing the environment, an abstract value representing the empty environment (i.e. with no binding), an abstract value representing the "pervasives" environment and finally, for each name-space a search function and an extension function (i.e. a function that adds a binding to an environment, then returning this environment with the binding "in front"). This means that an environment (dedicated to a "Blabla" processing) will have a signature like (forget the noisy arguments we spoke about, like `loc`, `current_unit`, we will check them in detail later) :

We address now some remarks about the `Make` functor. This functor provides the environment access functions that do not depends on the structure of data stored in the environment. However, it uses the functions that depends on that are provided by its argument module.

As a general scheme, for each name-space `xxx`, we have 3 functions:

- `add_xxx` : Adds a binding to the environment (exported function).
- `find_xxx` : Entry point to find the information bound to an identifier (exported function).
- `find_xxx_vname` : Find the information bound to a `Parsetree.vname` (not exported). This function is just an internal used by `find_xxx_vname` once it decomposed the initially looked-up identifier into a primitive `Parsetree.vname`.

```
module BlablaEnv :
  sig
    type t
    val empty : unit -> t
    val pervasives : unit -> t

    val add_value :
      Parsetree.vname -> BlablaInformation.value_binding_info -> t -> t
    val find_value :
      loc: Location.t -> current_unit: Types.fname ->
        current_species_name: string option -> Parsetree.expr_ident -> t ->
          BlablaInformation.value_binding_info

    val add_constructor : Parsetree.constructor_name -> Types.fname -> t -> t
    val find_constructor :
      loc: Location.t -> current_unit:Types.fname ->
        Parsetree.constructor_ident -> t -> Types.fname

    val add_label : Parsetree.vname -> Types.fname -> t -> t
    val find_label :
      loc: Location.t -> current_unit: Types.fname -> Parsetree.label_ident ->
        t -> Types.fname

    val add_type :
      loc: Location.t -> Parsetree.vname ->
        BlablaInformation.type_binding_info ->
      t -> t
    val find_type :
      loc: Location.t -> current_unit: Types.fname -> Parsetree.ident -> t ->
        BlablaInformation.type_binding_info

    val add_species :
      loc: Location.t -> Parsetree.vname ->
        BlablaInformation.species_binding_info -> t -> t
    val find_species :
        loc: Location.t -> current_unit: Types.fname -> Parsetree.ident ->
      t -> BlablaInformation.species_binding_info
  end
```

Adding a binding

All the environment extension functions (`add_xxx`) are quite simple, they just add a binding to the association list in field of the environment related to the right name-space. One may note that an added binding is always tagged `BO_absolute`. This means that the binding comes in the environment from the current compilation unit, opposite as the tag `BO_opened` that serves to identify bindings present in the environment due to an `open` directive.

Looking for a binding (except for values)

Look-up functions are more complex, especially the one looking for values. We now describe the general look-up mechanism that is exactly the one used for `find_label`, `find_constructor` and `find_type`.

The first and simplest case is when the looked-up identifier doesn't have any qualification. In this case, it must trivially be searched in the related association list of the current environment.

The second case involves a qualified identifier and is more complex.The principle is to split the looked-up name into a scope specifier (a qualification) and a simple name. For instance, `foo` has no scope specifier and a simple name equal to "foo". Conversely, `basics#bar` has the scope specifier "basics" and the simple name "bar". Then, from the scope specifier (i.e. in fact an "module" name), we try to load the ("sub")-environment of this module. If the specifier is `None`, then we will get back the current environment. It is in this environment that we will perform the search (like in the first case described above since the obtained identifier doesn't have anymore qualification) . Now, the point is to

check if we are allowed to accept to find a binding induced by an `open` directive. This is usually the case except if the identifier has an explicit qualification equal to `None` or `Some (file)` where `file` is the current compilation unit.

The first point allows to recover an identifier defined in the **current compilation unit** even if some opened modules imported identifiers wearing the same name. This features strongly relies on the fact that the parser **must** parse qualified identifiers like `#foo` as a **global** one with `None` as qualifier. In other words, such an identifier means "the related definition being at toplevel in the current compilation unit".

The second point simply states that if we are looking for a qualified identifier being in the current compilation unit, then is must not be searched among identifiers imported by opened modules.

Based on this information, we simply look-up in the association list to find the related binding.

```
(* ****************************************************************** *)
(** {b Descr} : Looks-up for an [ident] inside the types environment.

    {b Rem} : Exported outside this module.                           *)
(* ****************************************************************** *)
let rec find_type ~loc ~current_unit type_ident (env : t) =
  match type_ident.Parsetree.ast_desc with
   | Parsetree.I_local vname ->
       (* No explicit scoping information was provided, hence *)
       (* opened modules bindings are acceptable.             *)
       find_type_vname ~loc ~allow_opened: true vname env
   | Parsetree.I_global qvname ->
       let (opt_scope, vname) = opt_scope_vname qvname in
       let env' = EMAccess.find_module ~loc ~current_unit opt_scope env in
       (* Check if the lookup can return something *)
       (* coming from an opened module.            *)
       let allow_opened = allow_opened_p current_unit opt_scope in
       find_type_vname ~loc ~allow_opened vname env'


(* ****************************************************************** *)
(** {b Descr} : Looks-up for a [vname] inside the types environment.

    {b Rem} : Not exported outside this module.                       *)
(* ****************************************************************** *)
and find_type_vname ~loc ~allow_opened vname (env : t) =
  try env_list_assoc ~allow_opened vname env.types with
  | Not_found -> raise (Unbound_type (vname, loc))
```

Table 5.2: Sample code for simple look-up function

Looking for a binding for values

This case is more subtle since it involves more kinds of identifiers, especially because of the "!-notation". Processing for local and global identifiers (`Parsetree.I_local` and `Parsetree.I_global`) are the same as previously described.

We must then address the case of finding a binding for a method with the "!"-notation (i.e. `Parsetree.EI_method`). There are two different cases.

1. There is no species qualification before the !, or the specified species is `Self`. In this case then the searched identifier must belong to the inheritance of `Self`. First, this means that opened stuff is forbidden. Next, because the `values` bucket is so that inherited methods and our methods belong to it, we just have to search for the `vname` inside the current environment.

2. There remain the case where these is an explicit species qualifier different of `Self`. We still have two cases.

(a) The specified species is the current species (not `Self`, the real name of the current species). This may arise because of substitution performed during typechecking in species signatures. In this case, the search is obviously the same than the previous case since it is really like the case where the qualifier is `Self`.

(b) In the other cases, we must recover the environment in where to search (i.e. the environment compound of the definitions present in the species), according to if the species is qualified by a module name. Note that in this environment, all the imported bindings are tagged `BO_absolute`. This tag allows to make the difference between bindings introduced by the definitions of the current compilation unit and those brought by opened modules. So if the species is hosted in a module (i.e. compilation unit), we first get this module's environment otherwise we keep the current environment. Then we look for the species/collection definition. We transform all its definitions into an environment via the function `EMAccess.make_value_env_from_species_methods`. Finally in this environment, we look for the binding of the method name.

This should be sufficient, but in case of scoping environment, we still have to modify the value bound to our identifier. For this reason, we have an extra function provided by the `EMAccess` module that will be finally applied on the obtained value. Currently, in the other environments, this function is the identity (do nothing on the argument) since we only found a job to do in case of the scoping environment. We will describe what this function really does in the case of the scoping environment in its dedicated section.

5.2 Scoping environment

The scoping environment provides information required to compute the scope of an identifier. In other words, it enables to state the extend in the program where an identifier is visible. The aim is to prevent name confusion and to ensure that names are identified by (i.e. related to) only one definition.

5.2.1 The `ScopeInformation` module

We first look at the information bound to each name in the different name-spaces.

Sum type value constructors

Since these identifiers are always introduced by type definitions that are at toplevel, we only need to remind in which compilation unit they were introduced, hence hosted. Then a constructor will be bound in the environment to a compilation unit name.

Record type field names

For the same reason than for the sum type value constructors, Record type field names are bound to a compilation unit name.

Values

Information bound to value identifiers is more complex since it must depict the different cases of identifiers.

- `SBI_file` The identifier is at toplevel of a file (including the current file). We record the file name.
- `SBI_method_of_self` The identifier is a method implicitly of `Self`.
- `SBI_method_of_coll` The identifier is a method explicitly of a collection. We record the fully qualified name of the collection.
  Attention: while inserting a method at its definition point in the environment, it must always be tagged with `SBI_method_of_self`. The tag `SBI_method_of_coll` can only be returned by `find_value` who may perform a change on the fly if required.
- `SBI_local` The identifier is locally bound (`let` or function parameter).

Types

Type constructors may be separated in two categories:

- `TBI_builtin_or_var` The type identifier is either a type variable name (`'a` for instance) or a built-in type. In these two cases, the constructor is not hosted in a type definition in a particular compilation unit. In effect, type variables arise during the type-checking process and are not attached to a particular type definition. In fact, a type variable **is not** a type constructor and only this justifies the fact is doesn't belong to a particular compilation unit.
  On the other side, we allow the possibility to have built-in type constructors in the FoCaLize language. This means that we can have type constructors related to no type definition written as FoCaLize. In this case, the compiler must deal internally with this type constructor. Currently this feature is not used since even basic types constructors like `int`, `bool`, `char`, ... have a regular FoCaLize (external) definition.

- `TBI_defined_in` The identifier is a type constructor name defined at toplevel by a type definition in a file. We record the hosting compilation unit.

Species

Data recorded for species is the most complex. It must depicts the scope of the species name itself, but also give information about the methods and parameters available through this species. For this reason, we have a record grouping several things.

First, (`spbi_methods` records the list of **all** the methods names owned by the species, **including those added by inheritance**. Methods however appear only once and are ordered from the most recent ancestor are in head of the list. In case of multiple inheritance, we consider that ancestors are older from left to right.

Next, we record information about the species's parameters (`spbi_params_kind`). For each parameter we simple record if it is a collection (`SPK_is`) or entity (`SPK_in`) parameter.

Finally (`spbi_scope`), we record if the current species is defined at toplevel in a compilation unit (`SPBI_file`) or is a collection parameter of another species (`SPBI_parameter`).

5.2.2 The `ScopingEMAccess` module

This module provides the functions we previously stated to help the generic environment module to access the information stored in the environment. We have 2 functions to inspect.

The function `make_value_env_from_species_methods` simply takes the list of the method names of the species and add them as methods **"of the species"** passed as argument (usage of `SBI_method_of_coll`) in the **values** name-space. Here the insertion explicitly shows that the methods inserted in the environment are those of a particular species, not of `Self`. Moreover, the methods are tagged as `BO_absolute` to say that by default they do not come from an "opened" module since we consider that may be the species comes from an "opened" module, but not its methods who are hosted **in** the species.

The function `post_process_method_value_binding` is important for the scoping environment (and in fact, currently we have such a function in the interface of `EMAccess` only because we need to make a special job in the case of the scoping environment). Its job consists in changing the tag of the methods of the species from `SBI_method_of_self` to `SBI_method_of_coll`. In effect when we create a species, its methods are tagged as being "of `Self`". But when we load the methods of this species in an environment we must say that these methods are not belonging to the currently processed species but are belonging to the species from which we loaded the methods in our environment.

For example, let's examine the following source code: When we are scoping in the Species `S` the method `meth1` we encounter the expression (identifier) `P!meth2`. To scope this identifier we need to get the list (i.e. a "sub"-environment) containing the methods of `P` in order to search for the name `meth2` among them. When `P` was "created" (in fact when it was scoped) we inserted in its scoping description the fact that it had a method `meth2` ... of `Self` (yep, inside `P`, `meth2` is really a method of `Self`). So it was tagged with `SBI_method_of_self`. So if we insert

```
species S (P is ...) =
  let meth1 (x) = ... P!meth2 ... ;
end ;;
}
```

straight-line this information in our environment, while scoping our species `S` we will see that there exists a method `meth2` ... of `Self`, i.e. of `S` which is wrong. For this reason, we must change the tag of `meth2` before inserting the binding in our environment.

5.3 Typing environment

The typing environment provides information required to type-check the program, i.e. to infer types where they are not written, to check that explicitly written types are correct and to annotate the AST with the type found type of each AST node. Moreover, FoCaLize requires more than type-checking "à la" ML. We will also record information about dependencies.

5.3.1 The `TypeInformation` module

We first look at the information bound to each name in the different name-spaces. We will speak here of *type* and *type scheme*. The different between them will be exposed later while presenting the type-checking process. Let's intuitively say that a *type scheme* is a "template" from which one can extract a *type* by "instanciation" of the scheme. So in **expressions**, only *types* are present, *type schemes* are bound to **definitions**.

Sum type value constructors

A sum type value constructor is considered as a function taking as many parameters as they have arguments and returning a value whose type is the one hosting the constructor (i.e. the current type definition). For example: leads

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

to 3 value constructors of types: `A: t`, `B : (int * bool) -> t` and `C: int -> bool -> t`. Note the difference between `B` and `C`: the first one take 1 argument that is a pair, the second take 2 arguments.

So, for each constructor of a sum type, we record its type scheme (`cstr_scheme`) and if it has arguments or not (`cstr_arity`, that can be `CA_zero` or `CA_some`).

Later, to type-checking an expression using a constructor applied to some arguments, we will simply simulate a regular function application and type-check this application like any other one. That the reason why we also need to know if the constructor has argument(s) or not since if it has none, there is no application to simulate: the constructor in not a function but a value with the right type directly.

Record type field names

In the same idea than for sum type value constructors, a record type field label will be considered as a function taking 1 argument whose type is the type of the field an returning a value value whose type is the one hosting the field label (i.e. the current type definition). For example: leads to 2 value field labels of types: `lbl1: int -> t` and `lbl2 : string -> t`.

```
type t = {
  lbl1 : int ;
  lbl2 : string }
;;
```

So, for each field label, we simply record its type scheme (`field_scheme`) . We also record if the field is mutable or not (`field_mut`) but this features is not used (for extension purpose) since FoCaLize doesn't handle mutability (i.e references).

Values

Values are simply bound to their type scheme.

Types

Type constructors must be bound to more complex information since they must exhibit the kind of definition they are introduced by. We have 4 kinds of definitions:

- `TK_abstract` The constructor represents an abstract definition, i.e. a type whose values are not introduced by the definition itself. For instance, `int` is abstract since its values do not appear in the type definition: they are built-in in the compiler. Type abbreviations are also handled this way. This means that defining `type t = (int * string)` create a type `t` that is abstract. Since there is no value and no structure information in the definition, we do not need to record anything special.

- `TK_external` The constructor represents a type whose representation is explicitly provided for external languages. In other words, we import in FoCaLize types from (an)other language(s). We record on what this constructor must be mapped in the various foreign languages, the value constructors or field labels if some exist that can be used on FoCaLize's side and on what to map them in the various foreign languages.

- `TK_variant` The constructor represents a sum type definition. Hence values are given in the definition's structure itself. Hence we record the list of value constructor names with for each its arity and its type scheme (in fact these 2 things are the same than recorded for each value constructor in the related name-space).

- `TK_record` The constructor represents a record type definition. Hence, field labels are given in the definition's structure itself. Hence we record the list of these fields names with their type scheme and their mutability (like for sum types value constructors, these 2 things are those recorded for each field label in the related name-space).

Species

The structure of information recorded about species is pretty complex since it must depict the complete species structure, most importantly the parameters and all the methods stuff, but some other additional things used during type-checking stage.

- `spe_kind` Describes if the species is a toplevel collection (`SCK_toplevel_collection`), a toplevel species (`SCK_toplevel_species`) or a collection parameter (`SCK_species_parameter`).

- `spe_is_closed` This boolean tells if the species if fully defined (even if not turned into a collection). This information will be useful to known when collection generators must be created.

- `spe_sig_params` The list of descriptions of the parameters the species has, ordered the same way they appear in the species definition (i.e. the left-most in head of the list). We will examine the structure of this information a bit later.

- `spe_sig_methods` The list of description of all the methods of the species. We will examine the structure of this information a bit later. One may note that like for scoping information, all the methods are present, i.e. the inherited also, once and in the inheritance resolution order. This way, the methods represent the "normal form" of the species as described in the theory.

- `spe_dep_graph` We record here the dependency graph of the species' methods. Hence, this deals with methods "of `Self`" and represents def and decl-dependencies. This doesn't include dependencies on collection parameters !

Species parameter information Since FoCaLize provides 2 kinds of parameters we have 2 types of information to record:

- For entity parameters, `SPAR_in`, we keep the parameter's name, the collection representing its type and the flag telling if this collection is a parameter, a toplevel one (like for the field `spe_kind` of a species description seen above).

  For example, `species S (Nat is ..., n in Nat)` will record for `n` that its name is "`in`", its type is `Nat` and that this type is from a collection that is a collection parameter (i.e. `SCK_species_parameter`) and not a toplevel one.

- For collection parameters, `SPAR_is` we record more information, especially things that have already be computed about the species the parameter is "`is`" in order to have a quick access to information instead of computing it each time we encounter a collection parameter of a certain "type". To base our presentation let's take the case of the species taken just above for the entity parameters.

  - We record the name of the parameter with its compilation unit name. In fact, this will be the "type" of the collection this parameter makes available in the hosting species. In our example, `Nat` is both the name of the parameter and the type that uniquely represents the collection is bring. In effect, to use methods of this parameter, we will use `Nat!xxx`. By adding the compilation unit name, we exactly build a "type of species" like we use anywhere else in the compiler.
  - Like for entity parameters we record the flag telling if the species the parameter is "`is`" a parameter, a toplevel collection or a toplevel species. In our example, assuming that `Nat` is a species defined somewhere else, we would have `SCK_toplevel_species`.
  - Next come the list of all the methods descriptions the parameter has. This list is exactly of the same type than the list that describes a species methods (whose structure will be examined just after). It is in fact the normalised form of the species methods the parameter has.
  - Since a collection parameter "is" a species expression, not only a simple identifier (for example `P2` and `P3` in `species S (P1 is ..., P2 is T (P1), P3 is U (P1, P2)) = ... ;;` who are built by applying arguments to parametrised species, we must record the complete species expression denoting what "is" the parameter. Instead of directly recording the AST part representing the expression, we translate it into a custom type.

    In effect, in the AST, expressions used to represent species expressions are general expressions for sake of parsing technical issues. But in fact, due to the structure of the syntax, only a few kinds of AST nodes can be created. In particular, expressions coming out from the parser have restricted forms (invariants due to the syntax). Hence, instead of always assuming these invariants hold in the expressions representing species expressions, we prefer to convert them in a dedicated structure, more restricted than the general expressions, to enforce these invariants to hold directly by the type of the structure. For this reason, we record the species expression under the form of a `Parsetree_utils.simple_species_expr`.

Method information Depending on the kind of the methods we have different strucutures to record. In any case, we always record the history of the method, i.e. where it was defined and from where and how it was inherited until it arrives in the current species (structure `from_history`) and its name. The kinds of method are basically sumarised by the type `spe_sig_methods` and can be:

- `SF_sig` represents a signature. since we do not have anything more than a type for this method, we keep it as a type scheme.
- `SF_let` represents a `let` definition, either "computational" or "logical". We record the list of parameter names of the definition (if it's a constant, then this list is empty), the type scheme of the definition, its the body ("computational" or "logical"), a termination proof if it has some, a structure telling if during type-checking we detected def and decl-dependencies on the carrier (on `representation`) and finally the flags found in the AST telling if the definition was `logical` and was recursive.
- `SF_let_rec` Here is a list of information identical to what we store for `SF_let`, one for each recursively defined functions (i.e. only 1 element in the list if there is no mutually recursive functions).
- `SF_theorem` represents a theorem definition. We record the mapping of type variables found in the `forall` and `exists`" in the property's logical expression onto their name. This is a technical point. In fact, theorems and properties do not have a type scheme. Their "type" is their logical statement. However, it is possible to have universally quantified type variables appearing in a theorem statement. Without a type scheme we are not able to technically instanciate these variables in a consistent way. To we patch this leak of scheme by recording for each universally quantified type variables, what identifier in the statement has this type. Next, we record the logical statement of the theorem, its proof, and like for `SF_let`, information on dependencies found on the carrier.
- `SF_property` represents a property definition. We record the same information than for a theorem, except the proof since a property doesn't have any proof.

5.3.2 The `TypingEMAccess` module

The access module for the typing environments is quite trivial. The function `make_value_env_from_species_methods` works simply inserting the methods of a species in the value bucket like it's done for the scoping environments. The only point is that since theorems and properties no do have a type scheme, we insert them with a trivial type scheme whose body (core type) is simply `prop`.

The function `post_process_method_value_binding` is trivial and is the identity. In effect, we do not need any post-processing like we needed for scoping environments.

5.4 OCaml code generation environment

The OCaml code generation environment provides information required to generate the target OCaml source code.

5.4.1 The `MlGenInformation` module

We first look at the information bound to each name in the different name-spaces.

Sum type value constructors

In this bucket, we only record on which string a sum type value constructor that has been introduced by a type definition involving an external representation must be translated into OCaml.

For instance, in OCaml, `Nil` will be mapped onto `[]` and `Cons` to `( :: )`. It may be clearly noticed that for OCaml, only constructors coming from **external** sum types are entered in the generation environment. Hence, if a constructor is not found, then this means that it comes from a regular FoCaLize type definition, not dealing with any external material, hence must be simply translated into OCaml using the name given in FoCaLize side.

Record type field names

FoCaLize's records are generated as real records in the target languages. Hence, we only need to remind the name on which to map each record field name of the FoCaLize type definition. Conversely to the sum type value constructors, all the record field labels are recorded in the environment.

Values

For OCaml code generation, we do not need any information about values. Hence, the type of the bound values is trivially `unit`. But in fact, we will never enter values not look for values in such an environment.

Types

Types also do not need any information to be generated. Hence, they are handled like values above in term of environment.

Species

For species (and collections) we must find a way to know their methods and other information coming from the dependency computation. All this is recorded in the `species_binding_info` structure below:

- The list of parameters (collection and entity) of the species with their kind. This information is the same than the one stored in the type-checking environment.
- The list of methods the species has. This information is the same than the one stored in the type-checking environment. Remember that is represents the methods present in the normal form of the species, i.e. with not double, with inheritance resolved and in the right order (to prevent dependency issues).
- An optional `collection_generator_info` describing, if available, the structure of the species' collection generator. This generator is optional because species that are non fully defined do not have any collection generator although they are entered in the environment. Generator data is used to prevent computing several times (in case where several collections are built from a same closed species) and contains:
  - The list of species parameters names and kinds the species whose collection generator belongs to has. This list is positional, i.e. that the first name of the list is the name of the first species parameter and so on. The kind of a parameter is the same thing than the one recorded in the scoping environment.
  - The list mapping for each parameter name, the set of methods the collection generator depends on, hence must be provided an instance to be used. Note that the list is not guaranteed to be ordered according to the order of the species parameters names (that's why we have the information about this order given in the first component of the `species_binding_info` structure we are globally describing).
- Finally, a flag telling if this information is bound to a species (`COS_species` or a collection (`COS_collection`).

5.4.2 The `MlGenEMAccess` module

The access module for the OCaml code generation environments is really trivial. The function `make_value_env_from_species_` has no meaning for OCaml code generation environments because we do not provide any `find_value` function and that's this function that requires `make_value_env_from_species_methods`.

The function `post_process_method_value_binding` is trivial and is the identity. In effect, we do not need any post-processing like we needed for scoping environments.

5.5 Coq code generation environment

The Coq code generation environment provides information required to generate the target Coq source code. Its structure will be pretty close to the OCaml environment. Most of the differences are induced by the explicit polymorphism in Coq (that imposes to keep trace of polymorphic type variables) and by the presence of logical methods (i.e. theorems, that were discarded in OCaml).

5.5.1 The `CoqGenInformation` module

We first look at the information bound to each name in the different name-spaces.

Sum type value constructors

The idea behind information bound to sum type value constructors is about the same than in OCaml. We want to know if needed on what string to map a constructor name if it comes from an external type definition. But we always need another information to handle polymorphic constructors.

In effect, in Coq a polymorphic function (because constructors are functions in fact) must be provided one extra argument for each polymorphic type variable appearing in the type of the function. This argument must have type `Set` and be provided at application-time, like the other remaining (regular) explicit arguments of the function definition. In fact, Coq can infer the value of these arguments and we can simply use a _ (underscore) in place of these extra arguments at application-time. However, we need to know how many _s must be generated (i.e. how many extra arguments the constructor has due to its polymorphism). For instance, the type `t` defined in Coq by: introduces 1

```
Inductive t (alpha : Set) : Set :=
 | A : (alpha -> (t alpha)).
```

value constructor `A` with not 1 argument, but 2 ! If we want to create a `t` value parametrised by the integer `1`,

```
Coq < Check (A 1).
Toplevel input, characters 9-10:
> Check (A 1).
>          ^
Error: The term "1" has type "nat" while it is expected to have type "Set".
```

To have a valid application, we must add one _ before our `1`, like in:

```
Coq < Check (A _ 1).
A nat 1
     : t nat
```

We could be more explicit by directly providing the value of the first argument that is in our case `nat`:

```
Coq < Check (A nat 1).
A nat 1
     : t nat
```

but since Coq knows to infer it, we prefer let Coq doing the work (type-checking get simpler on FoCaLize's side).

For this reason any sum type value constructor is recorded in the environment with an optional mapping to a name on Coq side and the number of extra arguments it must be apply to due to the type generalised variables appearing in its type.

Record type field names

Record type field names are handled like in OCaml.

Values

In the same order of idea than for sum type value constructors we need to record the number of extra arguments to apply (when the value is functional) due to polymorphic type variables present in the value's type.

Moreover, for technical reasons due to Zenon, we also need to remind if the value is a toplevel property (`VB_toplevel_property` with its logical expression body), a toplevel let-bound value (`VB_toplevel_let_bound`) or something else (`VB_non_toplevel`).

Types

Like in OCaml, types also do not need any information to be generated.

Species

Species are bound to exactly the same information than they are in OCaml code generation environments.

5.5.2 The `CoqGenEMAccess` module

The access module for the OCaml code generation environments is pretty simple and works exactly like for type-checking environments.

Chapter 6

Scoping

The scoping process aims to link each identifier occurrence to its related definition according to the rules that drive the visibility of identifiers. It avoids name confusion and ensure that each occurrence of identifier refers to one and one unique effective definition.

Ideally, once scoped, a program should have unique names for each identifier (i.e. a name plus a stamp that ensure the unicity). For example in the following piece of code:

```
let y = 0 ;;
let x =
  let x = 1 in
  let x = x + 1 in
  let y = x + x in
  y + x in
y + x ;;
```

scoping rules allow to uniquely identify identifiers by renaming them in:

```
let y_0 = 0 ;;
let x_0 =
  let x_1 = 1 in
  let x_2 = x_1 + 1 in
  let y_1 = x_2 + x_2 in
  y_1 + x_2 in
y_0 + x_0 ;;
```

This way, each occurrence has a clearly identified definition since if two identifiers have the same name then they refer to the same definition. Moreover, this ensure that all the occurrences have one and one unique related definition, hence preventing unbound identifiers.

Moreover, since compilation unit can be "`open`-ed", some identifiers can be used without explicit qualification, then looking like belonging to the current compilation unit although they belong to another one. Scoping also allow to "rename" identifiers belonging to an "`open`-ed" unit to make their hosting unit clear by adding a qualification.

In FoCaLize, we do not rename the identifiers with indices. Instead, we make they qualification explicit. The case where two identifiers with the same qualification are present is handled by the environment mechanism that will hide the oldest identifier definition by the newest according to their order in the source code. In fact, in the case of toplevel definitions in a same compilation unit and methods in a same species, the environment mechanism refuses to have several times the same names. This is not a technical problem, this is only a choice to prevent the programmer from masking these fundamental kind of definitions.

Hence the output of the scoping pass is an AST where all the identifiers occurrences received a qualification if they are not locally bound. This means that by default, the parser must parse identifiers that are not qualified (i.e. with no #-notation and/or no !-notation) as **local identifiers** (i.e. as `Parsetree.I_local`). This means that during scoping, only `Parsetree.I_local` identifiers will be affected by the scoping transformation. Local identifiers will be looked-up to determine whether they are really local or are method names or toplevel (of a file) names. The

transformation is not performed in place. Instead, we return a fresh AST (still possibly having sharing with the original one) that will be suitable for the typechecking phase.

For identifiers already disambiguated by the parser, there are 2 cases: "#-ed" and "!-ed" identifiers. The scoper will still work by ensuring that these identifiers are really related to an existing definition.

- For "#-ed" identifiers, the look-up is performed and they are always explicitly replaced with the name of the hosting file where they are bound. Hence in a current compilation unit "Kikoo"", then `#test ()` will be replaced by `Kikoo#test ()` if the function `test` was really found inside this unit. If it was not found, then an exception is raised.
- For "!-ed" identifiers, the look-up is performed but no change is done. If it is like `!test()`, then it is **not** changed to `Self!test` !!! Only a verification is done that the method exists in `Self`. If it is like `Coll!test`, then also only a verification is done that the method exists in `Coll`.

The scoping heavily uses the "scoping environment" structure described in 5.2. Scoping is performed on each phrase of the source text, in the order of apparition of these phrases. Hence we have to scope phrases among documentation title, `use` directive, `open` directive, `coq_require` directive, species definition, collection definition, type definition, toplevel value definition, toplevel theorem definition and toplevel expression. For each scoped phrase, the **new environment** made of the initially received one extended by the new scoped definitions is returned. We don't return only a delta: we return a complete usable new environment.

Most of the scoping functions use a parameter named a "context". This structure is intended to group into 1 unique parameter various values (that would otherwise be as many parameters) the scoping functions will mostly always use.

```
type scoping_context = {
  (** The name of the currently analysed compilation unit. *)
  current_unit : Types.fname ;
  (** The optional name of the currently analysed species. *)
  current_species : string option ;
  (** The list of "use"-d (or open-ed since "open" implies "use") modules.
      Not file with paths and extension : just module name (ex: "basics"). *)
  used_modules : Types.fname list
} ;;
```

6.0.3 Scoping an `use` directive

Scoping a `use` directive returns an unchanged environment. It simply adds the "used" module to the list modules allowed to be used of the current context and return a new context with this extended list. The point is only to mention for the rest of the scoping passe that qualified identifiers with this module as qualification is now allowed.

6.0.4 Scoping an `open` directive

This directive has no to be really scoped. Instead, it has an impact on the scoping process and more accurately on the scoping environment. In load in the environment the scoping information of the identifiers contained in the opened compilation unit, tagging then as `BO_opened` like seen in 5.1.4. Hence all the imported identifiers will be known as possible definitions to use as "origin" of an identifier occurrence, according to the scoping rules.

6.0.5 Scoping a species definition (`scope_species_def`)

Before scoping a species, the first thing is to pass a modified context in which we record that we are inside this species (field `current_species` on the context.

The scoping environment of a species will be gradually extended as long as we process its components: we must first add the parameters of collection and entity in their order of apparition then import the bindings from the inheritance tree, and finally local bindings will be added while scoping the species' body (fields). When scoping involves searches in the environment, searches must be done in the following order:

1. Try to find the identifier in local environment.

2. Check if it's a parameter of entity ("`in`") or collection ("`is`").

3. Try to find the identifier throughout the hierarchy.

4. Try to find if the identifier is a global identifier..

5. "Else" ... not found !

So the order in which the identifiers are inserted into the environment used to scope the species must respect extensions in the reverse order in order to find the most recently added bindings first.

Hence we first scope the species parameters. Once they are scoped, we get an environment where they are bound to their scoping information.

Using this new environment, we now scope the `inherits` clause, i.e. the inheritance. In addition to the scoped inheritance expression, this will give us bake the names of the methods we inherits. This is especially useful because we must add them into our environment before scoping the methods defined in this species. Indeed, the bodies of these defined methods may make reference to identifiers corresponding to inherited methods.

Once these inherited methods are added to the current environment, we can scope the defined methods in this new environment. Each scoped method is added to the environment used to scope the remaining methods.

Once all is scoped, we create the scoping information for the species itself and add it to our initial environment (not to the one where we gradually added parameters, methods etc, since they do not need to be visible outside the species). And finally, in the type bucket of this environment, we add the type corresponding the the species carrier.

We then return the scoped species definition and the new environment where the species and it carrier are bound. This new environment is not a delta, it is a complete "all-in-one" environment suitable to scope the remaining phrases of the source file.

Scoping the species parameters (`scope_species_params_types`)

The species parameters (i.e. collection and entity parameters) are scoped in their apparition order, adding the scoping information of each one to scope the next ones. We have two cases of parameters.

- Entity parameter `a in C(...)`: we must scope the collection expression `C(...)` the parameter `a` is "`in`". To do so, we use the current scoping environment. Once this collection expression is scoped, we must insert in the environment the name of the parameter (here, `A`. Since it is a **value** (whose type will be the carrier of the collection expression), we add it in the values bucket of the environment.

- Collection parameter `P is C(...)` we must scope the collection expression `C(...)` . This gives back the scoping information (with methods names list) of this expression. Because collections and species are not first-class values, the environment extension will not be done in the values bucket. Instead, we must extend the environment with a "locally defined" collection having the same methods than those coming from the collection expression `C(...)`. So from the obtained scoping information (with methods list) of this expression, we create a fresh species scoping information that we bind to the parameter name `P` in the environment. Then, because a species induces a type by its carrier, we insert in the type bucket of the environment a type constructor representing the carrier of the parameter (here a type `P`).

The scoping of a parameter returns the extended environment and the scoped version of the list of parameters.

Scoping the inheritance clause (`scope_inheritance`)

Scoping the inheritance is done by scoping each collection expression in their order of apparition. Like we saw for species parameters scoping, scoping a collection expression returns the scoped expression and the list of the methods names this expression has. Hence, for each parent, we collect the obtained scoped methods. They will be later inserted in the environment to scope the remaining of the species. Note that we don't need to add the methods found of a parent to scope the next parent in the inheritance list. When all these collected will have to be inserted in the environment, we must take care to insert first those of the "left-most" parent in the `inherits` clause, going on from left to right. This ensure searches in the environment will comply the inheritance resolution order of FoCaLize. During insertion in the

environment, because these methods are inherited, they now are methods of ourselves, hence methods of `Self` and they must be inserted as `SBI_method_of_self` ! The change needed to say that one of these methods is a method of the "current collection"'s name, i.e. toggling the flag to `SBI_method_of_coll` will be done during accesses in the environment (by `find_value`).

Scoping the defined methods (`scope_species_fields`)

To scope all the defined methods of the species, we scope each field in its order of apparition, then add the obtained information about it in the environment used to scope the remaining methods.

Scoping a method is done by scoping its body (a `Parsetree.expression`) then returning this scoped method and the scoping environment extended by a binding between this method and the computed scoping information. One must note that depending on if the method is recursive, we pre-insert its name in the scoping environment before scoping its body or not. If the method is recursive, we pre-insert, otherwise no.

Scoping in general, scoping other constructs

For each used "identifier" occurrences (i.e. structures that represent identifiers since we have several kinds in FoCaLize), we look-up in the scoping environment for information about this identifier.

Depending on the form of the identifier, we either re-build explicitly a scoped identifier with explicit qualification (case of `Parsetree.expr_ident`) or we simply check that the identifier is really bound (case of identifiers where qualification is already built-in in its structure).

Depending on the class of the identifier (value, type, record field...) we look-up in the related bucket in the scoping environment to get the required information.

The interesting function in scoping is `scoped_expr_ident_desc_from_value_binding_info` that in fact re-build an `expr_ident_desc` from the inner simple name (i.e. `vname`) found in an `expr_ident` and the scoping information bound to this name in the scoping environment.

Chapter 7

Type-checking

The type-checking pass performs in fact several important tasks. It obviously infer the type of each expression and construct, but it also performs inheritance resolution, compute dependencies on methods of `Self` (def and decl-dependencies), ensure that the species are well-formed and sort the methods in order they are properly ordered. Once the type-checking pass is ended, a processed species gets in normal form, i.e. with all its methods present once, inherited and defined ones having been consistently put together.

This especially means that at each inheritance step, any species issued by the type-checker has all its methods: the inheritance disappears from the species structure. Obviously, we still have means to know about the inheritance history somewhere, but all the methods, inherited, defined, declared are always all together in a species in normal form.

In other words, considering only the bunch of methods a species has, there is no difference between a species having them via inheritance and a species having them in its own current body with no inheritance.

This point is especially important since it allows to inductively be sure that if one inherits of a species, then in just one shot we know which methods we have inherited: there is no need to walk again along all the inheritance steps. Then, we can says that we inductively build the normal form of species all along the inheritance tree. This point allows faster searches and prevent from having information disseminated in several place which would be more difficult to maintain.

In fact, the scoping pass already used a similar way to proceed, keeping for each species the list of all the methods it had, either in its own body or by arbitrary inheritance.

The output of the typing pass is a `Infer.please_compile_me` structure in which we have both the AST of the definitions and information computed during typing. This structure will then be passed to the "abstraction" stuff (directory `src/commoncodegen`) to compute extra dependency information and group all the dependencies together and factorise some computations required by code generation (more especially, what to lambda-lift and to instanciate). This "abstraction" pass is called before calling the code generation by each code generator back-end (in fact, called by each target code generator).

We will now examine various points of this typing pass.

7.1 Type inference

7.1.1 Where to record type information in the AST ?

Type inference is the process of guessing the type of each expression, each definition of the source code. In fact, in FoCaLize, types are partly inferred, partly given by the programmer. Signatures are a way to make types explicit by giving annotations. However, at each node of the AST, types must be infer-ed to finally label the node. In effect, the first output of the type-checking pass it a "typed AST", i.e. the initial AST with each node now having its type recorded in the node itself.

As defined in the source file `basement/parsetree.mli`, an AST node is a generic data-structure containing a specific description:

Generic AST node

```
type 'a ast = {
   (** The location in the source of the AST node. *)
   ast_loc : Location.t;
   (** The description of the node. *)
   ast_desc : 'a;
   (** The support for documentation. *)
   ast_doc : documentation;
   (** The type of the node. *)
   mutable ast_type : ast_node_type_information;
}
```

Hence, a node containing an expression (hence of type `Parsetree.expr`) will be built by something like:

An "expression" AST node

```
type expr_desc =
  | E_self
  | E_const of constant
  | E_fun of vname list * expr
  | ...

type expr = expr_desc ast
```

In the generic data-structure of the node, the field `ast_type` is used to record the type inferred for this node. Since we want to keep the same AST structure all along the compilation process, we use a mutable field for the type because initially, after lexing/parsing and scoping, the type is not yet known. So, the type-checking pass will modify the value contained in this field for each node of the AST.

According to the source file `basement/parsetree.mli`, values for this field can be:

An "expression" AST node

```
type ast_node_type_information =
   | ANTI_non_relevant   (** The node has no meaningful type information.
          However, it was processed by the type-checker. *)
   | ANTI_none      (** The node was not yet processed by the type-checker.
           Clearly, after the type-checking pass, no AST node
      should remain with this tag in the [ast_type] field of
      the node ! *)
   | ANTI_type of Types.type_simple  (** The type information is a type. Mostly
          used to label expressions. *)
   | ANTI_scheme of Types.type_scheme (** The type information is a type scheme.
           Mostly used to label definitions. *)
```

Before type-checking is done, all the node of the AST have their `ast_type` field worthing `ANTI_none`. Once the type-checking pass is done, no AST node should remain with this tag. If some do, then is must be considered as a bug (node forgotten during processing) of the compiler. At least, when a node does not require a type information (for instance, the AST node of a `open` directive), it must however be traversed by the type-checker and must be updated with the `ANTI_non_relevant` value.

7.1.2 Types and type schemes

Like in regular ML-like type-checkers, expressions are assigned "types" although definitions are assigned "type schemes". The difference is due to the ability for definitions to be polymorphic and to be instantiated differently at each usage occurrence. In effect, an expression exists in only one point. So it has one **type**, that's all. A definition leads to a "template" of types, where polymorphic type variables car be instantiated as wished each time the identifier bound by the definition is used. For this reason, a definition is bound to a "model" of types, a "family" of type, that is usually called a **type scheme**.

Then a type scheme is in fact a list of polymorphic type variables and a body that is a type expression. For instance, the ML-like type scheme (possibly bound to a `List.map` function) `('a -> 'b) -> 'a list -> 'b list`

will be represented by the list of it's 2 type variables `'a` and `'b` and its body that is the expression `('a -> 'b) -> 'a list -> 'b list`. In fact, in a type scheme all the polymorphic type variables are **implicitly quantified universally**. The "implicitly" is the reason why it is so difficult for the programmer to really see the difference between a type and a type scheme. For the above type scheme, we should be more explicit and better write:

`∀'a, 'b . ('a -> 'b) -> 'a list -> 'b list`

If we now have a look at the following expression: `List.map (fun x -> x + 1)`, then the expression (i.e. the identifier node) `List.map` will have the **type** `('t -> 'u) -> 't list -> 'u list` (in which we intentionally changed the names of the types variables to show that they are **not the same** that those of the type scheme). This type expression contains 2 variables `'t` and `'u` that will be unified during the type-checking of the whole application expression (unified with `int` in the current example).

In FoCaLize, type schemes are compound of the list of the polymorphic type variables that are its parameters and the type expression that is its body:

An "expression" AST node

```
type type_scheme = {
  ts_vars : type_variable list ;          (** Parameters in the scheme. *)
  ts_body : type_simple    (** Body of the scheme where generalized types
                               have a level equal to [generic_level]. *)
}
```

7.1.3 Working "in place" with substitutions

The type-checking algorithm of ML-like languages is often stated using the notion of MGU and using substitutions. In FoCaLize, the effective inference algorithm uses techniques more efficient in practice than regular substitutions to manually apply and combine on the type terms. Instead, we work "in place", by taking benefits of sharing between type sub-terms to simulate the substitutions by direct physical modifications inside the terms.

The full description of this technique is outside the scope of the present document, but a clear and efficient explanation can be found in "Le langage Caml" written by Pierre Weis and Xavier Leroy.

The idea is to represent the types by terms that can be physically shared, with type variables that can be directly assign a value. Hence, as described in the source file `basement/type.ml`, our type algebra is:

An "expression" AST node

```
type type_simple =
  | ST_var of type_variable                    (** Type variable. *)
  | ST_arrow of (type_simple * type_simple)    (** Functional type. *)
  | ST_tuple of type_simple list               (** Tuple type. *)
  | ST_sum_arguments of type_simple list       (** Type of sum type value
                                                   constructor's arguments. To
                                                   prevent them from being
                                                   confused with tuples. *)
  | ST_construct of
      (** Type constructor, possibly with arguments. Encompass the types
          related to records and sums. Any value of these types are typed as
          a [ST_construct] whose name is the name of the record (or sum)
          type. *)
      (type_name * type_simple list)
  | ST_self_rep     (** Carrier type of the currently analysed species. *)
  | ST_species_rep of
      (** Carrier type of a collection hosted in the specified module. *)
      (fname * collection_name)


and type_variable = {
  (** Binding level of the type. *)
  mutable tv_level : int ;
  (** Value of the type variable. *)
```

```
  mutable tv_value : type_variable_value
}


and type_variable_value =
  | TVV_unknown
  | TVV_known of type_simple
```

The algebra describes the built-in type constructors and more interestingly for our explanation, the **type variables**. We can see that in the type algebra, all the variables look pretty structurally the same: the constructor `ST_var` and a `type_variable` containing a few information. Hence, for instance 2 variables whose value are unknown will look exactly the same. To make the difference, we must consider physical equality. Hence, if 2 variables are physically equal, "they are **the** same" variable(s?), otherwise they are really different. The aim of this mechanism is to share the same physical data to represent all the occurrences of a variable in a term, so that when we want to assign it a value, we just need to modify it in place and all the shared occurrences will be updated for free.

A type variable is initially an unknown of the unification equation induced by the type-checking process. Hence it starts with its field `tv_value` worthing `TVV_unknown`. If during unification, a variable needs to be assigned a value (i.e. a constraint was found on this variable), then its `tv_value` that is mutable will be assigned `TVV_known` "of something". This "something" is itself a type and this allows indeed to instantiate a type variable by a type expression.

Obviously, this mechanism to represent instantiations will create "strings" of links between variables and their effective value. To be sure that the value of a variable is know "equal to something" or really unknown, we must use a mechanism that returns the canonical representation of a type. For instance, let's imagine that in our inference problem, we arrived to have 4 variables α, β, γ and δ, with $\alpha = \beta$, $\beta =$ `int`, $\delta = \gamma$ and $\gamma = \beta$. Hence, we have the following picture:

$$\begin{array}{ccccc} \alpha & \rightarrow & \beta & \rightarrow & \texttt{int} \\ & & \uparrow & & \\ \delta & \rightarrow & \gamma & & \end{array}$$

representing the system: where `'a = ST_var (TVV_known (...))` where `...` represents `'b` and has the structure `ST_var (TVV_known (ST_construct (``int'', [])))`. We have the same kind of thing for $\delta$ and $\gamma$, with $\gamma$ worthing `TVV_known (...)` with `...` representing the structure of $\beta$. In fact, despite all the variables we see above, all are equal and are instantiated by `int`. This means than must not trust the first value constructor seen for a type to know what it is equal to. One must "follow" the links.

Moreover, to avoid the loss of efficiency induced by walking each time along these "strings" of links, the operation of getting the canonical representation of a type will use the "path compression" operation in order to suppress indirections a soon as they are encountered a first time.

The operation returning the canonical representation is the guardian of the correct structure of the types. **Any operation working / relying / walking on the type structure must call this operation to be sure that the structural view of the type is has is really the canonical view of the type**. This operation called `repr` is located in the `basement/types.ml` source file of the compiler. The presence of such a strong invariant is the reason why the types are exported as **abstract**. This ensures that outside the module manipulating the type algebra, nobody will forget to get the canonical representation of a type.

Basically, this function receives a type. If this type is not a variable, then it returns it directly. This means that the type constructor is already known to be something else than a variable for which we should investigate further. On the other side, if the received type is a variable whose "value" (i.e. field `tv_value` is "known to be equal to something" (i.e. is `TVV_known (...)`, then we will ask to get the canonical representation of this "something". This is typically a recursive call on this "something". This way, if this "something" is itself a variable "known to be equal to something else", then we will inductively know each step of indirection. So, once we get our canonical representation of our "something", **this is** in fact the canonical representation of our type, since it was a variable "known to be equal to **this** something". By the way, before returning, since now we know that the variable is in fact "pointing" onto a type (i.e. is not anymore a variable, was instantiated), we take benefit to cut the string of indirections by directly establishing a link between the variable and the canonical representation we obtained. Hence, next time we will access this variable via a type sub-term shared somewhere else, we won't have anymore to walk along the whole "string" of links to know the variable's value. Then, the `repr` function simply looks like:

Getting the canonical representation of a type

```
let rec repr = function
  | ST_var ({ tv_value = TVV_known ty1 } as var) ->
      let val_of_ty1 = repr ty1 in
      var.tv_value <- TVV_known val_of_ty1 ;
      val_of_ty1
  | ty -> ty
```

7.1.4 Unification

The computation of the most general type of an expression, called type inference, strongly rely on unification of type terms. We will say that 2 terms τ_1 and τ_2 can be *unified* if there exists a substitution ϕ so that $\phi(\tau_1) = \phi(\tau_2)$. The substitution ϕ is then called *unifier*. of the terms τ_1 and τ_2. Hence, two terms can be unified if it is possible to instantiate all or part of their variables by a same substitution, so that they become structurally equal.

Based on the representation of our types, since the unification tends to return a substitution to apply on the 2 unified types in order to make them equal, instead of getting this substitution to later apply it to each type (and combine this substitution with the other substitutions the types may be subject to), we will directly make the type terms equal by instantiating their type variables. The "instanciation" is then made in place, by changing in place the mutable field `tv_value` of the unknown variables from `TVV_unknown` to `TVV_known` "of" the type required to have equality.

This is especially fast since all the occurrences of a variable share the same physical location. Hence, changing the value of the field `tv_value` at this location is equivalent to simultaneously instantiate all the occurrences of this variable (past and future) in type terms.

Hence, ideally unification doesn't return any result and makes the 2 unified type equal by side effect or fails because there exist no unifier for the 2 types. And then, in the type-checking algorithm, instead of using one of the type on which we apply its related substitution, we can directly use any one of the 2 types once unified since they are now equal.

In fact, in our case this is not completely the case since we have an additional problem that forces us to return a type. This is due to the fact that when unifying a type and `Self`, FoCaLize's rules require to have `Self` as unification result rather than any of one the two types. This is a problem when the unification used the known representation of `Self` to achieve finding the mgu. In effect, in this case, one of the 2 type is `Self` and this other is a type expression that is compatible with `Self`'s representation. And in this fact, choosing any of the 2 types as result is wrong: the result must always chose (prefer) `Self`. Hence our unification routine returns the preferred unifier in addition to make the physical modifications in place when required. We then have a unification function described in the compiler's source file `basement/types.ml` looking like (explanations follow):

The unification algorithm

```
0  let unify ~loc ~self_manifest type1 type2 =
1    let rec rec_unify ty1 ty2 =
2      let ty1 = repr ty1 in
3      let ty2 = repr ty2 in
4      if ty1 == ty2 then ty1 else
5      match (ty1, ty2) with
6       | (ST_var var, _) ->
7           (* BE CAREFUL: [occur_check] performs the setting of decl-dependencies
8              on the carrier ! In effect, if [ty2] involved Self then we have a
9              dependency on the carrier and that must be taken into account !
10             The interest to make [occur_check] doing this work is that it
11             walk all along the type so it's a good idea to take benefit of this
12             walk to avoid one more walk. *)
13          occur_check ~loc var ty2 ;
14          lowerize_levels var.tv_level ty2 ;
15          var.tv_value <- TVV_known ty2 ;
16          ty2
17      | (_, ST_var var) ->
18          (* BE CAREFUL: Same remark than above for [occur_check]. *)
19          occur_check ~loc var ty1 ;
```

```
20          lowerize_levels var.tv_level ty1 ;
21          var.tv_value <- TVV_known ty1 ;
22          ty1
23      | ((ST_arrow (arg1, res1)), (ST_arrow (arg2, res2))) ->
24          let arg3 = rec_unify arg1 arg2 in
25          let res3 = rec_unify res1 res2 in
26          ST_arrow (arg3, res3)
27      | ((ST_sum_arguments tys1), (ST_sum_arguments tys2)) ->
28          let tys3 =
29            (try List.map2 rec_unify tys1 tys2 with
30            | Invalid_argument "List.map2" ->
31                (* In fact, that's an arity mismatch on the types. There is a
32                   strange case appearing when using a sum type constructor that
33                   requires arguments without arguments. The type of the
34                   constructor's arguments is an ampty list. Then the conflict is
35                   reported as "Types and ... are not compatible". Hence one of
36                   the type is printed as nothing (c.f. bub report #180).
37                   In this case, we generate a special error message. *)
38                if (List.length tys1) = 0 || (List.length tys2) = 0 then
39                  raise (Arity_mismatch_unexpected_args (loc))
40                else raise (Conflict (ty1, ty2, loc))) in
41          ST_sum_arguments tys3
42      | ((ST_sum_arguments _), (ST_tuple _))
43      | ((ST_tuple _), (ST_sum_arguments _)) ->
44          (* Special cases to handle confusion between sum type value
45             constructor's that take SEVERAL arguments and not 1 argument that
46             is a tuple. *)
47          raise (Arity_mismatch_unexpected_args (loc))
48      | ((ST_tuple tys1), (ST_tuple tys2)) ->
49          let tys3 =
50            (try List.map2 rec_unify tys1 tys2 with
51            | Invalid_argument "List.map2" ->
52                (* In fact, that's an arity mismatch on the tuple. *)
53                raise (Conflict (ty1, ty2, loc))) in
54          ST_tuple tys3
55      | (ST_construct (name, args), ST_construct (name', args')) ->
56          (if name <> name' then raise (Conflict (ty1, ty2, loc))) ;
57          let args'' =
58            (try List.map2 rec_unify args args' with
59            | Invalid_argument "List.map2" ->
60                (* In fact, that's an arity mismatch. *)
61                raise
62                  (Arity_mismatch
63                     (name, (List.length args), (List.length args'), loc))) in
64          ST_construct (name, args'')
65      | (ST_self_rep, ST_self_rep) ->
66          (begin
67          (* Trivial, but anyway, proceed as everywhere else. *)
68          set_decl_dep_on_rep () ;
69          ST_self_rep
70          end)
71      | (ST_self_rep, _) ->
72          (begin
73          match self_manifest with
74           | None -> raise (Conflict (ty1, ty2, loc))
75           | Some self_is_that ->
76              ignore (rec_unify self_is_that ty2) ;
77              set_def_dep_on_rep () ;
78              (* Always prefer Self ! *)
79              ST_self_rep
80          end)
81      | (_, ST_self_rep) ->
82          (begin
83          match self_manifest with
84           | None -> raise (Conflict (ty1, ty2, loc))
```

```
85            | Some self_is_that ->
86                ignore (rec_unify self_is_that ty1) ;
87                set_def_dep_on_rep () ;
88                (* Always prefer Self ! *)
89                ST_self_rep
90          end)
91       | ((ST_species_rep c1), (ST_species_rep c2)) ->
92           if c1 = c2 then ty1 else raise (Conflict (ty1, ty2, loc))
93       | (_, _) -> raise (Conflict (ty1, ty2, loc)) in
94    (* ****************** *)
95    (* Now, let's work... *)
96    rec_unify type1 type2
97  ;;
```

First of all, we see that as previously said, since we intend to work on the structure of the types, we start by computing their canonical representation by calling `repr` (lines 2 and 3).

If the 2 types are already the same (physically, note the usage of == and not =), then there is nothing more to to and we can return any one of the 2 types as unifier. We can really return any one since they are really equal and the problem of preferring `Self` doesn't apply here: either the 2 types are both `Self` or they are both something else.

Then the algorithm considers all the cases of two types. When one is a variable (lines 6 and 17), we assign to the variable the other type hence telling that the variable is not anymore unknown (lines 15 and 21).

Before assigning the variable, we perform an "occur check" (lines 13 and 19). This ensures that the variable we assign doesn't appear in the type is it assigned. This is to prevent types from being cyclic. In effect, if we try to unify α with $\alpha \to \alpha$, we get in the following situation:

$$\begin{array}{ccc} \alpha & \to & \alpha \\ \uparrow & & \downarrow \\ & \leftarrow & \end{array}$$

with the arrow on the same line than the αs represent the functional type constructor and the arrows below the link the unification creates between variables and types. Hence, if the unification falls in this case, an error is raised telling that the type of the expression leads to cyclic types and that this expression is rejected. Below is the code of the occur check in which the line 12 and 13 can be skipped (they will be examined in 7.3.7).

Occur check routine

```
0  let occur_check ~loc var ty =
1    let rec test t =
2      let t = repr t in
3      match t with
4       | ST_var var' ->
5           if var == var' then raise (Circularity (t, ty, loc))
6       | ST_arrow (ty1, ty2) -> test ty1 ; test ty2
7       | ST_sum_arguments tys -> List.iter test tys
8       | ST_tuple tys -> List.iter test tys
9       | ST_construct (_, args) -> List.iter test args
10      | ST_species_rep _ -> ()
11      | ST_self_rep ->
12          (* There is a dependency on the carrier. Note it ! *)
13          set_decl_dep_on_rep () in
14   test ty
```

Let's go back to our unification routine. For the moment, the lines 14 and 20 (and the comment lines 7-12) can be forgotten since they will be explained when we will be dealing with polymorphism.

When unifying with one type being a variable, the returned type is always the type assigned to the variable (lines 16 and 22). We could also choose to return the variable but this would be less efficient since no `repr` is yet applied since we assigned it a value, then to use this variable as type, one `repr` will immediately be required. Choosing to return the other type that has been "`repr`-ed" we can save one call. This may seems a tiny advantage, but unification in place is efficient for several tiny advantages put together ! An this one is part of them.

In term of difficulty, we now have the unification of `Self` and a type. The simplest case (line 65) is the unification of `Self` and itself. Trivially, the unification succeed. Forget the line 66 for the moment. There remain 2 symmetric

cases: unifying `Self` and another type that is not `Self` and not a variable (lines 71 and 81). In these cases, we are in the rules [`Self1`] and [`Self2`] of Virgile Prevosto's PhD, page 27, definition 9. These are the cases where, if the structure of the carrier (i.e. of the `representation` method) is visible then an occurrence of `Self` is allowed to be unified with a type having a structure compatible with the one given by the `representation` method.

To be able to make so, in the typing context, we record if the structure of the carrier is known of not. This information if passed to the unification function via the parameter `~self_manifest`. This parameter has type `simple_type option`. If the value is `None`, this means that the structure of the carrier is not visible (hence `Self` can only be unified with `Self`). If the value is `Some ...`, then this means that we know that the carrier was defined as the type "..." (hence unifying `Self` with a type expression compatible with "..." must be successful).

So, when unifying `Self` with a type, if `~self_manifest` is `None` we raise an error because the carrier is abstract. If `~self_manifest` is `Some (`τ`)`, then we must ensure that τ can be unified (and if so, the possible side effects induced by this unification must be done) with the other type (lines 76 and 96): it is then simply a recursive call. Since if the unification succeed we want to return `Self` as unifier, we throw the result an return `ST_self_rep`, meaning the type "`Self`".

Attention: With this mechanism, we directly unify the type that represents the structure of the carrier. This means that we directly apply modification in place on it. This especially means that, because this is not a type scheme but a type, there is a cumulative effect of all the unifications that are made between `Self` and other types. Hopefully, this is not a problem since methods are not allowed to be polymorphic. And `representation`, defining the structure of `Self` is a method. So it can't be polymorphic. This especially means that the structure of `Self` will never have remaining variables that could be instantiated by a unification. Hence, there is no risk that a unification pollutes the type used as reference for the structure of `Self` by instantiating a type variable by a type that would be incompatible for a later unification of this variable by another type. Sooooo, this means that `~self_manifest` is a type and not a type scheme because we don't have polymorphism on methods. If we had some, we would need another mechanism to "prefer `Self`" during unification (problem of specialisation, generalisation, putting the right binding level, and so on...).

The other cases of unification are simpler and structural. The unification can now only be successful if the 2 types have the same constructor. So we check the types 2 by 2 with each time the same constructor and if the matching is right, we recurse structurally on the types structures.

Finally remains the all cases where the 2 types do not have the same constructor (line 93). This leads to a type-checking error by raising an exception.

7.1.5 Polymorphism

Like introduced in 7.1.2, type schemes are used to implement the polymorphism in type inference. They act as "models" of types. But the question is how to make such "models" and how extract types from such "models". The first point is handled by the `generalize` function and the second by the `specialize` function of the source file `basement/types.ml`.

We must start our explanation by introducing the notion of **binding level**. The aim is to keep trace of type variables introduced in let-definitions that are deeper than our level. The binding level counts the number of nested let-definitions. Hence, the current level must be incremented each **before** every potentially generalisable definition. This increase will enable the generalisation once we will go back to a lower level. And it must be decreased at the end of this definition. To increase the current binding level, the function `begin_definition` must be used; to decrease it the function `end_definition` must be used.

We said "potentially generalisable" definition since all let-definitions can't be generalised (non-expansivity problem).

There remains a little problem with the level of variables. The higher the level, the closest is the definition that introduced this type variable. When a variable is put in contact (i.e. unified) with a type containing variables with a lower level (i.e. created in outer let-definitions), hence that may not be generalisable when we will leave the current definition, we must reflect this onto our variable ! In effect, unification means equality. So if our variable is "equal" to a variable than can be generalised, so it must be for our variable. Hence, when unifying a variable with a type, we must hunt in this type for variables that have a lower level and if we find some, we must lower the level of our variable

to this found level. That's what the function `lowerize_levels` does. And in effect, this function is called when unifying a type variable with something (lines 14 and 20 in the `unify` function listed above).

So, the generalisation process consists in considering as generalisable, all the variables having a level strictly greater than the current binding level. We then search for all the generalisable variables, and remind them in a list. By the way, we change in place the level `tv_level` field) of the found variables by setting a level meaning "generalised" (technically, we choose a value to big, that there is no chance that anybody nests so many let-definitions to reach this level). Finally, to build the type scheme, we simply create a `type_scheme` structure with the list of found variables and the type itself as body.

Now, taking an instance of a type scheme is done by the function `specialize` that simply create a fresh type variable for each generalised variable of the type scheme, then copies the body of the type scheme, replacing each generalised variable by the corresponding fresh one.

7.1.6 Type inference among other typing things to do

The type inference we spoke about deals with expressions and definitions. Hence it is only one part of the analyses performed during the "typing pass". More accurately, it is the first analysis carried out after the scoping. Each expression of a program must be type-checked anyway where it appears, in entity parameters, in methods, in toplevel definitions. Some other constructs of the FoCaLize language require some other kinds of type-checking: that is the case of species, collections, collection parameters, inheritance species expressions. However, during these other kinds of type-checking, we use the type-checking of expressions to make the "glue" ensuring consistency. Rules given in Virgile' PhD in figure 3.2, page 43, section 3.8 are example of this idea (these rules deal with with species parameters).

Aside this notion of type-checking, we also have other analyses to ensure a program is sound and also to extract the basic shape of a species by resolving inheritance and late binding. All these things are performed during the typing pass, the type-checking being the first step.

The typing pass is driven by the source file `typing/infer.ml` and more especially by the entry point function `typecheck_file`. This function triggers the type-checking of each phrase of the compilation unit. Be aware that in the source of the compiler, "type-check" is widely overloaded and denotes the process of inferring type for entities and then ensuring they are well-formed and building the first pieces of information that will ease the code generation. In particular, the fact that each species is given a normal form containing all its methods is handy to recover the methods of parameters or inheritance species expressions since there is no need to walk along all the inheritance level (point already mentioned in the introduction of this chapter, at 7).

7.2 Environment and structures for the typing pass

The structure of the environment used during the typing pass has been presented in 5.3 and is coded in typing/env.ml.

The only other structure used during this pass is the typing context that group various information to pass to the functions instead of having to pass them individually hence preventing from having tons of arguments for each call. This structure, shown below is local to the `typing/infer.ml` and passed to each function under the parameter name `ctx`.

Typing context

```
type typing_context = {
  (** The name of the currently analysed compilation unit (i.e. the name
      of the file without extension and not capitalized). *)
  current_unit : Types.fname ;
  (** The name of the current species if relevant. *)
  current_species : Parsetree.qualified_species option ;
  (** Optional type Self is known to be equal to. *)
  self_manifest : Types.type_simple option ;
  (** Mapping between 'variables [vname]s and the [simple_type] they are
      bound to. Used when creating a polymorphic type definition. *)
  tyvars_mapping : (Parsetree.vname * Types.type_simple) list
```

```
}
```

7.3 Typing a species definition

The process of typing a species will be now examined step by step by step. This is done by typing its parameters, resolving inheritance, type-checking the methods, merging properties and proofs, normalising the species, computing its dependencies graph and then inserting it in the environment. A few other administrative tasks are performed and will be detailed below.

7.3.1 Dealing with the species parameters

For each parameter, this process builds the species type of the parameter and get the list of methods it has. Each parameter will be inserted in the environment as species and type (the type representing the carrier of this parameter).

There are 2 cases: entity and collection parameters. Both of them lead to a "parameter description" that will be recorded in the `Env.TypeInformation.species_param` that will appear in the hosting species `spe_sig_params` field.

7.3.2 Typing a species expression

A species expression is either a simple species/collection identifier like `Setoid` or an application of a species/collection identifier to one or more **species/collection identifiers**. This means that we have no expressions like `A(B(C))`.

The first thing is to find in the environment the name of the "main" species (i.e. the unique only if there is no application, or the name in "applicative" position if there is an application). From this search, we get a species description of a previously existing species.

The aim is to finally get (among other things, but this is the most obvious result we want) the list of methods this species expression has. If there is no parameter, then the list of methods is trivially the one obtained in the structure found in the environment. The most interesting point is when the expression is an application. We then must apply the species to its effective argument(s) before being able to know the methods it has. In effect, let's imagine we have:

```
species A (B is Setoid) =
  let eq (x in B, y in B) = B!bla (x, y) && B!bli (y, x) ... ;;

species C inherits A (D) ... ;
```

`C` inherits `eq` but not with the body `let eq (x in B, y in B)= B!bla (x, y)&&B!bli (y, x)...` but with the body `let eq (x in D, y in D)= D!bla (x, y)&&D!bli (y, x)...` where `B` was replaced by `D`. This is done by `apply_species_arguments` that returns the correct (substituted) list of methods but also the list of these substitutions and the list of species types `Self` must be compatible with. This is used in case of a species expression parametrised used applied to `Self` (this point is a bit more detailed in the following section 7.3.3).

The most tricky point is to build and use this list of substitutions during typing. In effect, we obviously build it while applying the arguments, **but we also use it** (in fact, the substitutions already existing in the list accumulator at the point we process an effective argument). In effect, this list contains the substitutions to apply to each effective parameter before processing it. This is the way to represent the fact that in rule [`COLL-INST`, page 43, figure 3.2, section 3.8 in Virgile Prevosto's Phd, the substitution $[C_1 \leftarrow C_2]$ is performed on t_s, i.e. in it's methods types **but also in the remaining effective species parameters** (yep, indeed, the t_s signature of a species contains both the methods and the parameters). Because these parameters will be "evaluated" after the current one, we need to delay the substitution until they are really processed. Be careful that by construction, this list contains the substitutions in **reverse** order of the application order. This means that the first required substitution is in tail !

Hence, for each species parameter, when we encounter a collection parameter, we type the expression it is "`is`" and we then must apply the already seen substitutions to this type to ensure that it is transformed if required to collections of possible previous "`is`" parameters. For instance, let's take the following code:

```
species Me (Naturals is Intmodel, n in Naturals) = ...
collection ConcreteInt implements Intmodel ;;
collection Foo implements Me (ConcreteInt, ConcreteInt!un) ;;
```

while typechecking the `ConcreteInt!un`, we get a type `Naturals`. But since in the `Foo` collection, `Naturals` is instantiated by the effective `ConcreteInt`, `Naturals` appears to be incompatible with `ConcreteInt`. However, in `Foo`, with the first instanciation, we said that `Naturals` **is** a `ConcreteInt`, and we substituted everywhere `Naturals` by `ConcreteInt`. So idem must we do in the type inferred for the effective argument `ConcreteInt!un`.

7.3.3 Inheritance resolution

The expected result of inheritance resolution is to load the inherited methods in the environment and get their signatures and methods information. Hence, combined with the defined methods, we will be able to know all the methods of the species. We also want to get a possibly new context where the fact that `Self` is now manifest is updated, in case we inherited a `representation` method.

By the way, since in the inheritance clause, we can have parametrised collection applied to `Self` as effective argument, we remind all the species type that `Self` must be compatible with (called `self_must_be` in the source code). In effect, since we are building the current species, the species type of `Self` (i.e. the currently built species) is not yet known. Then to ensure consistency we must check afterwards, i.e. once we got all the methods of the species if it compliant with all the species types expected as effective argument of the used collections on which we applied `Self`. **Note that currently the test is not performed and to remind we have to do this, the compiler emits a warning**.

When processing each inherited species expression, we type it like we did for the collection parameters. This gives us the list of methods the species has (hence the list of methods the current species inherits via the processed species expression), the list of species types that `Self` must be compatible with, and finally the substitutions applied on the formal parameters of parametrised species used in the `inherits` clause. Such substitutions are those replacing formal parameters by effective collection expressions when the species expression used in the `inherits` clause uses application. This information will be recorded in the history information of the inherited methods for later use.

For instance, considering the FoCaLize sample code:

```
species Foo0 (A0 is Sp0) inherits ... = let v = 1 end ;;
species Foo1 (A1 is Sp1) inherits Foo0 (A1) = let v = 2 end ;;
species Foo2 (A2 is Sp1) inherits Foo1 (A2) = end ;;
species Foo3 (A3 is Sp1, A4 is A3) inherits Foo2 (A4) = end ;;
```

we will remind that in `Foo3`, we inherited from `Foo2` applied to `A4` and that in `Foo2`, we inherited from `Foo1` applied to `A2`.

As described above in 7.3.2, these substitutions will be used to represent the fact that in rule [`COLL-INST`], page 43, figure 3.2, section 3.8 in Virgile Prevosto's PhD, the substitution $[C_1 \leftarrow C_2]$ is performed on t_s.

7.3.4 Type-checking methods

This is done by `typecheck_species_fields`. This function infers the types of the species fields contained in the list of methods definitions. The typing environment is incrementally extended with the found methods and used to type-check the next methods.

The function returns a 5-uplet whose 3 first components are suitable to be inserted in the structure of a species's type, and the last ones are the `proof of` and `termination proof of` fields that have been found among the fields. These `proof of`'s must be collapsed with their related property **but** at the inheritance level where the proof is found (not at the one where the property was stated), to lead to a theorem before the formalisation process starts. The `termination proof of`'s must be collapsed with their related `let rec` definitions also before the normalisation process starts.

Type-checking of methods need some comments in some field types cases.

The first one is for the signature `representation`. Once `representation` is found, the typing context will be modified by setting the field `self_manifest` to `Some` of the type expression. Then we first type-check the provided type expression for the carrier. But we can't directly set the inferred type into `self_manifest`. We must make a copy of the inferred type in order to keep the originally inferred type aside any further modifications that could arise while unifying anywhere `Self` with "its known representation". In effect, unification in place would establish a link by side effect from the representation to the type `Types.ST_self_rep`, hence fooling the explicit structure of what is initially `representation`. This first would prevent us from being able to generate code finally relying on the representation of `representation`. Furthermore, because of how `Types.unify` handles unification with `Types.ST_self_rep` to prevent cycles, unification of this **mangled** representation would succeed with any types, even those incompatible with the original **correct** representation of `representation`'s type.

Next, we need to add a bit of explanation about type-checking properties and theorems. For the same reason that in external definition, type variables present in a type expression in a property or a theorem are implicitly considered as universally quantified. In effect, there no syntax to make explicit the quantification. Then we first create a variable mapping from the type expression to prevent variables from being "unbound". We must increase the binding level because when processing `Pr_forall` and `Pr_exists`, the function `typecheck_logical_expr` needs to store the type scheme of the identifiers introduced. And since generalisation if done in `typecheck_logical_expr` with a binding level of + 1 compared to our current one, generalisation could not be done otherwise.

Not forgetting constraints from inheritance

It is roughly described in Virgile's work that during fusion, methods inferred types and possibly inherited ones are matched against together to determine the final type of methods. This is not sufficient. In effect, let's consider the following code:

```
species S1 =
  representation = int ;
  signature to_int : Self -> int ;
end ;;

species S2 =
  inherit S1 ;
  let to_int (x) : int = x ;
  let join (x) = to_int (x) ;
end ;;
```

While in `S2`, typing `join`, we already typed `to_int` but with type `int` $\rightarrow$ `int` ! In effect, the fusion pass is not yet done for species `S2` since it will be done once the species has all its methods typed. Hence, when typing the method `join`, we see it calls `to_int`, so `join` also has type `int` $\rightarrow$ `int`.

Then comes the fusion that really gives `to_int` the good type: `Self` $\rightarrow$ `int`. But, it's too late for `join` which is not impacted by the fusion (no signature for it defined above in the inheritance), and remains with its previous type.

It is clear that's not a question of missing dependency on `Self` in the dependencies calculus causing wrong abstractions (λ-liftings). In effect, the type found for `join` is `int` $\rightarrow$ `int` which is totally wrong since `to_int` has type `Self` $\rightarrow$ `int`, so the argument `x` of `join`, passed to `to_int` **must** be `Self` an not `int` !

The solution is to notice that while inferring the type of `to_int`, we didn't "compared" the found type to the already existing type due to signature brought by inheritance. So, when typing a method, trying to match it agains a possibly already existing type constraint can solve the issue. That's what is done: if a scheme is found in the environment (the initial environment, not the one with pre-bindings), then we take an instance of this scheme and unify the inferred type with it, retaining the result of the unification as final type for the method. If none is found, we simply keep the inferred type.

Dependencies on the carrier

We have a special case for computing dependencies on the carrier `Self` introduced by using the type introduced by `representation`.

In effect, we have a decl-dependency on the carrier if the type of a method contains a reference to `Self` (i.e. if a type sub-term is `Self`). One could only perform this check for each field as soon as we inferred its type. However, without any type annotation, the type of a method could hide the fact that it refers to `Self`. For instance, let's have the case:

```
species A =
  representation  = int ;
  sig m: Self ;
end ;;

species B inherits A =
  let m = 5 ;
end ;;
```

inferring the type of `m` in `B` would simply lead to `int`, hence telling that there is no decl-dependency no `Self`. In fact, after the species is put in normal form, it appears that indeed there was since the type of `m` is not `int` but rather `Self`. For this reason, the detection of this dependency is also performed at "fields fusion" stage, when building the normal form of the species. More accurately, the take benefits that at fusion stage, we will make a unification. Hence, during unification, we take care of the 3 cases where such a dependency can appear: unifying `Self` and `Self` (line 68) and the cases where we unify a type variable with something (lines 6 and 17) : in these 2 last cases, we know that wee need to make an occur check that will walk all along the type structure. So to avoid one more complete descent on this structure and because the occur check is called in this case, we made so it set itself a global hidden flag telling if there is or not a decl-dependency on `Self` by calling if needed the function `set_decl_dep_on_rep` in case it encounters the type constructor `ST_self_rep` (see line 12-13 in occur check routine in 7.1.4).

On the other side, we have a def-dependency on the carrier if the rules [`SELF1`] or [`SELF2`] of the section 3.3, definition 9, page 27 in Virgile Prevosto's PdD are used. To detect this, in the function `unify` we examined before (7.1.4) in lines 77 and 87, since we detect we are using the known structure of the carrier, we call `set_def_dep_on_rep` that is a function recording in a global hidden flag that a def-dependency was found. In fact for similar reason that for decl-dependency, the effective recording of presence of this dependency is also done at "fields fusion" stage. And hopefully, at this stage, we make a unification between all the types of the found occurrences of a same-name method !

As previously stated, in fact, the recording of these dependencies is also done as soon as we typed a field since doing it only at the fusion stage can be non sufficient in some case since at fusion we do not descent on the whole expressions of methods.

Hence, these global flags reminding the presence of decl/def-dependencies must obviously be reset before addressing a new method. That's done by calling `reset_deps_on_rep`.

7.3.5 Proof collapsing

Since it is possible to provide the proof of an existing property after this property, we must transform properties having a proof collapsing the two into one theorem. In effect, a `proof` field alone is not a correct field in the normal form of a species: it must be related to a property. This process is carried out by the function `collapse_proofs_of` This function tries to find among the list of methods it receives as argument, property fields whose proofs are separately given in the list of proofs also passed as argument.

Each time the search succeeds, the property and the related proof are merged in a new theorem field, hence discarding the property fields. Because this process is performed before the normalisation pass, we still require to have 2 separate lists of methods:

- the inherited ones,
- those defined at the current inheritance level.

For this reason, the search will be done first on the methods defined at the current inheritance level (in order to find the "most recent") and only if the search failed, we will try it again on the inherited methods.

Attention: Such a merge now requires a re-ordering of the final list of fields. In effect, by moving the `proof of` field when merging it as a `SF_theorem` located where the initial `SF_property` field was, if the proof (that was originally "later") uses stuff defined between the `SF_property` and the original location of the proof, then this stuff will now appear "after" the proof itself. And this is not well-formed. The example file found in `focalizec/tests/ok__need_re_ordering.fcl` illustrates this need. This re-ordering will be done after the normal form of the species is computed.

A question on the fly: While writing the document, I wonder if this collapsing process wouldn't be carried out in the fusion algorithm used to create the normal form, exactly like when we pair signatures and `let`-definitions...

7.3.6 Normal form

The normalisation is done in `normalize_species`, applying the algorithm described in Virgile Prevosto's PhD in Section 3.7.1, page 36 plus its extension to properties and theorems in Section 3.9.7, page 57. Type "equality" test is performed as usual using the unification, with no unification error meaning "true" and error meaning "false".

The only thing is that we make clearly explicit the "silently described" notion of conflict detection mentioned in Virgile Prevosto's PhD page 57 line 6.

In effect, if a field is inherited several times via the **same** parent, erasing must not be performed ! That's like if we did as if there was no erasing to do ! Such a situation can arise if 2 species `B` and `C` inherit from a species `A` (with a method `m`) and another species `D` inherits from from `B` and `C`. The method `m` then appears twice in `D`, once from `A` via `B`, once from `A` via `C`. And this is not a conflict that requires erasement: in effect, there is no "redefinition". For this reason, before erasing a field, we check with the function `non_conflicting_fields_p`.

During normalisation, an operation of "fusion" is used. This is mostly the once described in Virgile Prevosto's PhD in section 3.6, page 35 and completed in Section 3.9.7, page 56, definition 35. The only difference is that we made explicit the notion of "to be equal", "to have the same types" for 2 properties or 2 theorems. In effect, the type of a logical method is its logical statement. Hence, it is an expression of the FoCaLize language. This means that we must compare 2 AST structures. A simple structural equality doesn't have any sense since at least the 2 structures will differ by their source location, hence always returning "different". So, the comparison we adopt here is performed thank to the `typing/ast_equal.ml` module. Hence we implement a custom structural equality to determine if 2 expressions have the same form modulo α-conversion of the bound identifiers. In particular, we are only interested in comparing only the `Parsetree.ast_desc` field. We also try to consider that parentheses are non significant is case an `E_paren` is found not equal to another expression. Currently, the comparison is done on the skeleton of the expression without tricky heuristic like considering that $A \Rightarrow B \Rightarrow C$ is equivalent to $(A \wedge B) \Rightarrow C$ and so on...

During this fusion operation, when we "fusion" a signature and a `let`-definition, if the unification succeeds, then we **keep the type given by the signature** to assign it to the method. We don't keep the type infered for the method. In effect, the inference algorithm may find a type that is more general than the signature, or may leave some type occurrences expanded althgough they are in fact occurrences of `Self` (obviously because there was previously de definition of `representation` equal to type. For instance, if we have:

```
species A0 =
  sig f : Self ->Self ;
end ;;
species A1 inherits A0 =
  let f (x) = x ;
end ;;
```

, the inference algorithm will find in `A1` that `f` has type $\forall\alpha.\alpha \to \alpha$ since this is the most general unifier. And because polymorphic methods are not allowed, if we do not perform signature matching for `f`, then first, we won't see that it must be considered to be `Self` $\to$ `Self`, then second we will reject the method. Hence this point shows that verification that methods are not polymorphic must be done **after** the definitions are assigned the type of their possible signatures.

Another example, that doesn't involve polymorphism would be:

```
species B0 =
  representation = int
```

```
  sig f : int -> Self ;
end ;;
species B1 inherits B0 =
  let f (x) = x + 1 ;
end ;;
```

Since nowhere in the definition of `f` in `B1 Self` appears, the type inference algorithm will see that `1` has type `int`, that + as type `int` → `int` → `int` and then will deduce that `f` has type `int` → `int`. Unfortunately, the developer wanted to see `f` has a function injecting an integer into the species data-type, hence wanted to consider it with type `int` → `Self`. So if we don't keep the type of the signature to assign it to the definition, we forget this constraint information.

7.3.7 Computing def/decl-dependencies

As previously stated, def and decl-dependencies reflect dependencies of methods of the species on methods of this species. In other words, they are related to dependencies of methods of `Self` on methods of `Self`. We already examined how to compute theses dependencies for the particular case of the carrier (the method `representation`) in . We now address this problem for all the other methods. In fact, dependencies computation is done in two passes. The first one is to create a simple dependency graph and is performed at typing stage. The second will be to exploit this graph to compute for each method of `Self`, the set of methods of `Self` that must be abstracted because of dependencies. This will be done via the "minimal coq typing Coq environment" in a next stage.

So, for the moment, we concentrate on building the dependency graph of each field. In effect, each **field** will have its own dependency graph (note, a field can contain several functions like in the case of mutually recursive methods). The graph of each method is built sharing node information between methods. In other words, like any graph, we have nodes, if a same node is required to build the graph of `m1` and `m2`, then these 2 graphs will physically share the node. Again, in other words, the dependency graph of each species is a sub-graph of a global graph.

The data-structure for such graphs can be found in the source file `typing/depGraphData.mli`.

In this graph, a node is a method an edge between 2 nodes n_1 and n_2 is a direct dependency relation telling that n_1 depends on n_2. Hence, $meth_{foo} \rightarrow meth_{bar}$ means that $meth_{foo}$ depends on $meth_{bar}$. Each edge is tagged with information telling if the dependency is a decl (`DK_decl`) or a def (`DK_def`) dependency. In case of decl-dependency, we have a tag telling if this dependency comes from the type (`DcDK_from_type`, the body (`DcDK_from_body`) or the termination proof (`DcDK_from_term_proof`) of the field. In case of def-dependency, we have a tag indicating if this dependency comes from a termination proof (`DfDK_from_term_proof`) or not (`DfDK_not_from_term_proof`).

The method `representation` on which we can have dependencies via dependencies on `Self` as previously see, is handled like the other methods by the graph building algorithm. The only difference is that because we recording at type-checking stage directly if a method has dependencies on `Self` and recorded this information in flags of the methods, we will have hard-wired cases to check for a dependency on the carrier since this information is obtained just consulting the flags in opposition with others methods where we need now to inspect their code.

A few things to remark:

- Signatures can't introduce decl or def-dependencies.
- `let` methods can only introduce decl-dependencies via their computational body.
- Properties can only introduce decl-dependencies via their logical statement (i.e. their type in fact).
- Theorems can introduce both kinds of dependencies: decl by their statement (i.e. type in fact) and decl/def by their proof (i.e. body in fact).

The basic process to build the graph is to process each field in turn. Each name defined in the field will lead to a node in the graph (i.e. "somebody" that may depend on "somebody else" or "somebody" on which "somebody else" may depend on). Hence, each time we need to create a node, we use a special function that first look among the already created nodes, if it find a node having a right name, it returns this node, otherwise it really creates the node. This ensure that node will be create once and will be shared between every occurrence of apparition.

The starting point of graph construction is the function `build_dependencies_graph_for_fields` hosted in the source file `typing/dep_analysis.ml`. This function contains 2 local function that deal respectively with `let`-definitions and theorems/properties. They basically behave the same way:

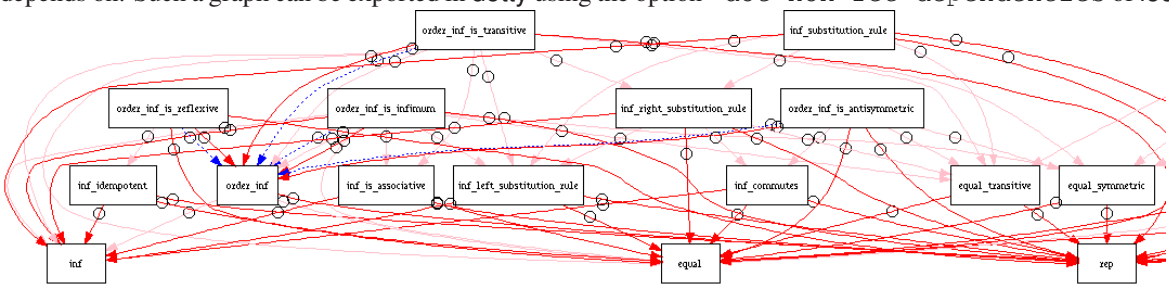
1. Check in the field's tag if there is a a decl-dependency on `representation`. If so, manually add an edge from the current name's node to the node of `representation`. Note that this edge is always tagged as "coming from type" (`DcDK_from_type`) since even in a theorem, a decl-dependency on the carrier can only come from the type because one can't say in the body, i.e. in the proof, **`by definition of Self`** or **`by property Self`**.

2. Find the names of decl and defs dependencies for the current name (trivially, for `let-definitions` def-dependencies are trivially always empty). This gives us the names of methods we decl-depend from our type, those we decl-depend from our body and those we def-depend. This computation is a simple descent over the AST of the method, hunting for calls to methods identified to belong to use. Once we got these 3 sets of names, for each of them, we create a node (create or get if it already exists). Finally, for all these nodes ("dependencies nodes"), we add an edge properly tagged from the current name's node to each of the dependencies names' nodes.

3. **Only for `let`-definitions**: we process the optional termination proof. By making a descent on the AST of the proof, we get the names of methods we decl-depend and those we def-depend. Exactly like in the above point, we create nodes for these names and add edges properly tagged (refining the decl or def tag by the tags telling "coming from a termination proof", i.e. `DcDK_from_term_proof` or `DfDK_from_term_proof`).

4. Check in the field's tag if there is a a def-dependency on `representation`. If so, manually add an edge from the current name's node to the node of `representation`. Note that this edge is always tagged as "coming not from a termination proof" (`DfDK_not_from_term_proof`) since properties and theorem do not have such proof and in the case of recursive `let`-definitions, anyway if the termination proof refers to `Self`, then the parts outside the proof obviously also. Hence, the carrier will always need to be lambda-lifted.

At the end of this process, we have a dependency graph telling for each method of the species, which method is depends on. Such a graph can be exported in dotty using the option `-dot-non-rec-dependencies` of focalizec.



Attention : However, this does not give us yet what we need to λ-lift in term of methods of `Self` ! We still need to perform some analyses, especially to compute the "visible universe of each method to finally get the "minimal Coq typing environment" to have this information. But this is done in another pass.

7.3.8 Miscellaneous

When finally the species is built, after having checked that it is well-formed, it has no doubles, it is fully defined or not and so on, we finally add the species description in the environment. We also must add in the environment a type that represents the species' carrier. However, this must be done only the species is really fully defined. In effect, in this case, it will really have a carrier for the species. This especially avoid such things:

```
species A =
  signature f : int
end ;;
species B =
  signature b : A -> A
end ;;
```

for which, if we generate the code for this, in OCaml we won't have any type definition for `me_as_carrier` in `A`. And in `B`, `g` will have the type `A.me_as_carrier ->A.me_as_carrier` where `A.me_as_carrier` unbound. This is moral since `A` doesn't have a carrier.

Attention: While writing this note, I wonder if the point that the species must have a method "representation" would be in fact sufficient...

7.4 Typing a collection definition

Typing a collection definition is pretty close to typing a species. The main difference is that we don't have inheritance, hence the normal form of the collection is directly the normal form of the species it "implements".

We have to ensure that the species expression we "implements" is fully defined before being able to accept the collection.

Type-checking the collection's fields is in fact just performing the abstraction, replacing `Self` by the collection name in the fields (i.e. in the bodies and in the types). Then we do this substitution by applying `SubstColl.subst_species_fiel` on each fields. This gives us the list of fields of the collection. This list will be later saved as part of the information bound to a collection in the type-checking environment.

Note that we do not use the dependency graph of the implemented species: we compute it in a regular way. This prevent us from having to manage ourselves substitutions due to abstractions all over the graph. Using the regular process, we get a graph with the right types and so on directly.

In effect, creating a collection implies abstracting `Self`, i.e. replace it in types and bodies by the carrier of the collection. If we reused the dependency graph of the species the collection implements, we would have to perform ourselves the substitutions all over the information contained in the graph. Since this information represents a lot of stuff and involved sharing and cycles, doing this manually would be very hard and error prone. So we prefer to let the regular mechanism working for us.

7.5 Typing a type definition

Type-checking a type definition serves to introduce in the typing environment a type constructor and possibly elements induced by the type, like value constructors in case of sum type definition or field labels in case of record type definition. We basically have 2 kinds of type definitions: "regular" and "external". Both of them can introduce 3 kinds of types: aliases, sum types and record types. The main difference between "regular" and "external" definition is in the way the type and its components are mapped onto the target languages. In a "regular" definition, mapping is handled by the compiler model scheme. In an "external" one, the mapping is guided by the user's annotations. Anyway, both kinds of definitions can introduce the same kinds of elements in the typing environment.

Because type definitions are implicitly recursive, before type-checking a type definition, we always pre-insert in the typing environment the currently typed type constructor. At this point, anyway its real definition, it is considered a abstract (i.e. we don't know its body). In particular, if it is a sum or a record we don't insert (know) anything about its possible value constructors or field labels. In fact, that's not a problem since these 2 kinds of information can never arise inside a definition body. After the type definition is fully type-checked, we will forget this primary insertion and prefer the new one coming from the type-checking.

In type definitions, type variables are implicitly universally quantified and are bound before the definition's body. In a definition like **`type`** `t ('a)=` **`alias`** `('a 'a)*`, we see that the `'a` appears just after the name `t` and before the body of the definition. Hence, before type-checking the definition, we previously insert in the environment the bound variables to that they be known while analysing the body. The mapping between type variable names and their related type (in term of type in our algebra) is recorded in the type-checking context in the field `tyvars_mapping`.

7.5.1 Regular type definitions

Type alias

A type alias simply introduces an equivalence between a name (the type constructor) and a type expression. For instance, in `type t = alias (int int)*`, the name `t` becomes compatible with `(int * int)`. This means that we do not create a type having new values: the values are only made of gluing existing values. Moreover, this type `t` is not really new in the sense it is not only compatible with itself but with any type compatible with `(int * int)`.

To type-check such a definition, we simply type-check the aliased type expression. This gives us a type (`Types.type_simple`) that is the canonical representation of the type expression. Then we simply bind the new constructor to this type. In other word, in the environment we bind the new type constructor to its identity in our algebra.

Since types can be polymorphic (i.e. parametrised by type variables), like for values, we do not really bind the name to a type but to a type scheme where polymorphic type variables are generalised. Hence, a type constructor can be seen like a function taking types as arguments and returning a type. For instance, considering the type of pairs `type t ('a)= alias ('a 'a)*`, the constructor `t` is considered internally by the compiler like a "function of types". For this polymorphic stuff, we find as for values, the usual story of binding level and generalisation.

Then, once the body of the definition is type-checked, we generalise it and add the new type constructor bound to this type scheme in the environment. Hence, each the type `t` will appear in a type expression, we will know its effective structure via the environment. For instance, using the above type `t`, if we encounter a type expression `t (int)`, we will ask for the scheme of `t`, get $\forall\alpha.\alpha \to \mathtt{t}(\alpha)$, we take an instance of this scheme and get $\alpha' \to \mathtt{t}(\alpha')$ and unify the leftmost α' with `int` to simulate an application, and finally we get the type `int * int` for the initial expression.

Type sum (A.K.A variant)

In addition to insert in the environment a type constructor, it also introduces value constructors. In this sense, such a definition creates a new type, with new values. This values may be parametrised by values of existing types, but they are anyway **new** values. In FoCaLize, a sum type is only compatible with itself. This means that anywhere a value of this type is expected, the provided expression will have to reduce on one of the values introduced by this type.

The difference with aliases previously examined is that we now must type-check, not a type expression, but an enumeration of constructors expressions as body of the definition. For each constructor of a sum type definition `t`, we have 2 cases: either it has no argument and is considered to have the type `t`, or it has arguments and is considered as a function taking values whose types are those of the enumerated arguments and returning a value of type `t`. Be aware that a value constructor parametrised by several arguments is a function taking several arguments, not a tuple of all the arguments ! Hence, on the following type definition:

```
type t =
  | A of int
  | B of (int, int)
  | C of (int * int)
;;
```

we will get: `A: int -> t`, `B : int -> int -> t` and C : (int * int) -> t. One may notice the difference between `B` that is parametrised by 2 arguments and `C` that is parametrised by 1 argument that is a tuple of 2 components. As usual, in the environment we bind each constructor not to a type but to a type scheme to be able to use polymorphic constructors with various instantiations.

Hence, to synthesise the type of an expression involving a sum value constructor of `t`, we will get in the environment its type scheme. If the constructor is used alone (i.e. with no expression as argument), then the type of the whole expression will be by construction `t`. Note that if the constructor is used alone although it shouldn't, we wont return a functional type but an error because the compiler always checks that value constructors are used with the correct arity.

There remains the problem of what to bind the type constructor to ? In effect, by opposition to alias types, here we don't have any type expression giving the "identity" of the type. In fact, a sum type is a new type only compatible with itself. Hence, we will bind it to a new type whose name is the name of the constructor. And to make values of this type, the only way will be to manipulate its value constructors.

Once the type scheme of each value constructor is inferred, we just have to add them into the environment, as well as the type constructor.

Type record

The principle of type-checking of records is very similar to the one of sum types. The only difference is that instead of value constructors, we deal with field label. In the same spirit that value constructors were considered like function, so are field labels. At programming level, a field label has one type. So, internally a field label of a record type `t` is considered to be a function taking an argument whose type is the type of the label and returning a value of type `t`. Note that to have an effective complete value of type `t`, of course we need all the fields to be assigned a value, but at typing stage, this is not our concern.

Hence, we infer the type of each field of the record and finally insert them in the environment as well as the type constructor itself.

7.5.2 External type definitions

Like seen in 2.6.1, external type definitions have two sides: the "internal" and "external". In fact, the "internal" part describes exactly the same thing than a regular type definition: "how the type is seen inside the compiler". For this reason, is it type-checked exactly the same way.

The "internal" part, since it contains some external code out of reach of FoCaLize, doesn't need any type-checking analysis: we simply record the various mappings it contains into the type definition for further usages.

7.6 All other toplevel constructs

For the remaining constructs, i.e. toplevel expression, toplevel theorems and toplevel `let`-definitions, the same principles than previously presented apply. The only difference is that in the typing context it will be recorded that we are not inside a species. For toplevel definitions, the test of non-polymorphism will not be performed since toplevel definitions can be polymorphic.

Directives don't need any type-checking. The `open` directive will act exactly like during the scoping pass, dealing this time with the typing environment instead of the scoping one (c.f. 6.0.4).

Chapter 8

Intermediate form

Once type-checking pass is ended, we saw that in addition to have the type of each expression computed and screwed in each AST node, we put the species and collections in normal form, having resolved inheritance and ordering problems between the methods. More over we computed the dependency graph of each method, hence indicated for each method which other method of `Self` is directly depends on (decl or def, via type, via body or via termination proof).

However this is not sufficient yet to known exactly what to abstract (i.e. λ-lift) in our method generators, then collection generators. We still need to find the complete set of methods of `Self` a method depends on and the set of collection parameters' methods a method depends on.

The first point will be carried out by computing the "visible universe" of a method and its "minimal Coq typing environment". The "visible universe" is the set of methods of Self that are needed for the analysed methods; the "minimal Coq typing environment" is just picking in the visible universe and abstracting or not its methods (i.e. just determining if we just need keeping their type or also their body). The second point is achieved by applying rules of page 153, definition 72 + the additionnal DIDOU rule.

Once these sets are known, the current pass will create a compact form of several data useful for code generation, hence preventing from having to compute several time the same things and to ensure that the same structure (in fact, most often, the order dependencies are abstracted) will be used every time needed. For instance, if in a method `m` we λ-lifted `m2`, then `m1`, extra arguments of this method will appear and will need to be always used consistently with this order (i.e. one must sure that we instantiate `m2` by a a method implementing the signature of `m2` and idem for `m1`. Moreover, since this λ-lifts information will be used at various points of the code generation, it is better to record it once for all instead of compute it again and again.

All this work is performed by stuff located in the source directly `src/commoncodegen` whose "entry" point is mostly the function `compute_abstractions_for_fields` of the source file `src/commoncodegen/abstraction.ml` (note that we have a dedicated function to process toplevel theorems because they are not hosted in species although they may require abstractions).

As explained in 7 the call to this pass is triggered by each target code generator (i.e. once by the OCaml code generation back-end, and once by the Coq one, obviously only if the code generation is requested for these target languages via the command line options). Conversely to previous passes, this one does not enrich any environment. However it takes a code generation environment. Since we have 2 target languages, we have 2 code generation environments (c.f. 5.4 and 5.5). Hence, the entry point of abstractions computation must be able to work with the 2 kinds of environments. That the reason why the environment is passed as a sum type:

```
type environment_kind =
  | EK_ml of Env.MlGenEnv.t
  | EK_coq of Env.CoqGenEnv.t
```

to allow to have only one set of functions to do this pass instead of duplicating the code and adapting it's behaviour in the few cases where one are interested in accessing the environment. The output of this pass is directly used by the code generation that called it to produce its final output (i.e. target language source code).

8.1 "Computing abstractions"

As stated in introduction the aim is to fully build the set of methods of `Self` and the set of collection parameters' methods a method depends on. At the end of this process, we want to get for each definition a structure grouping both the information present in the typing environment and the one synthesised about abstractions. Such a structure will then be suitable to be sent to a code generation back-end and looks like:

```
type field_abstraction_info =
  | FAI_sig of
     (Env.TypeInformation.sig_field_info * abstraction_info)
  | FAI_let of
     (Env.TypeInformation.let_field_info * abstraction_info)
  | FAI_let_rec of
     (Env.TypeInformation.let_field_info * abstraction_info) list
  | FAI_theorem of
     (Env.TypeInformation.theorem_field_info * abstraction_info)
  | FAI_property of
     (Env.TypeInformation.property_field_info * abstraction_info)
```

As we said, the second component of each parameters of the constructors is a `abstraction_info` that summarises all the things we will compute. Nothing very special about the constructors of this type: it is clear that we have one for each kind of method (just note that, as we presented before in the type-checking section, there is no more methods `proof of` since they have been collapsed with their respective property into theorems). This structure groups the results of various abstractions computation passes:

```
type abstraction_info = {
  ai_used_species_parameter_tys : Parsetree.vname list ;
  (** Dependencies on species parameters' methods. They are the union of:
        - dependencies found via [BODY] of definition 72 page 153 of Virgile
          Prevosto's Phd,
        - dependencies found via [TYPE] of definition 72 page 153 of Virgile
          Prevosto's Phd,
        - other dependencies found via [DEF-DEP], [UNIVERSE] and [PRM] of
          definition 72 page 153 of Virgile Prevosto's Phd + those found
          by the missing rule in Virgile Prevosto's Phd that temporarily
          named [DIDOU]. *)
  ai_dependencies_from_params :
    ((** The species parameter's name and kind. *)
     Env.TypeInformation.species_param *
     Env.ordered_methods_from_params)  (** The set of methods we depend on. *)
      list ;
  (* Dependencies used to generate the record type's parameters. It only
     contains dependencies obtained by [TYPE] and [DIDOU]. *)
  ai_dependencies_from_params_for_record_type :
    ((** The species parameter's name and kind. *)
     Env.TypeInformation.species_param *
     Env.ordered_methods_from_params)  (** The set of methods we depend on
                                           only through types and completion. *)
      list ;
  ai_min_coq_env : MinEnv.min_coq_env_element list
}
```

In effect, knowledge of what to λ-lift is acquired along different steps (corresponding to rules of definition 72 page 153 in Virgile Prevosto's PhD + one new rule that didn't exist and appeared to be mandatory). Hence this structure reminds the state of computed dependencies at some key steps. In fact, before being able to create so a summarising (!!☺) view of the information, we need to internally remind more key steps and use a more detailed structure where the results of the consecutive steps are not yet collapsed:

```
type internal_abstraction_info = {
  iai_used_species_parameter_tys : Parsetree.vname list ;
  (** Dependencies found via [BODY] of definition 72 page 153 of Virgile
      Prevosto's Phd. *)
  iai_dependencies_from_params_via_body :
```

```
    ((** The species parameter's name and kind. *)
     Env.TypeInformation.species_param *
     Parsetree_utils.ParamDepSet.t)  (** The set of methods we depend on. *)
      list ;
  (** Dependencies found via [TYPE] of definition 72 page 153 of Virgile
      Prevosto's Phd. *)
  iai_dependencies_from_params_via_type :
    ((** The species parameter's name and kind. *)
     Env.TypeInformation.species_param *
     Parsetree_utils.ParamDepSet.t)  (** The set of methods we depend on. *)
      list ;
  (** Dependencies found via only [PRM]. Obviously they are all present in
      the set below ([iai_dependencies_from_params_via_completions]). *)
  iai_dependencies_from_params_via_PRM :
    ((** The species parameter's name and kind. *)
     Env.TypeInformation.species_param *
       Parsetree_utils.ParamDepSet.t)
      list ;
  (** Other dependencies found via [DEF-DEP], [UNIVERSE] and [PRM] of definition
      72 page 153 of Virgile Prevosto's Phd + [DIDOU] applied on the rules
      [DEF-DEP], [UNIVERSE] and [PRM]. *)
  iai_dependencies_from_params_via_completions :
    ((** The species parameter's name and kind. *)
     Env.TypeInformation.species_param *
     Parsetree_utils.ParamDepSet.t)  (** The set of methods we depend on. *)
      list ;
  iai_min_coq_env : MinEnv.min_coq_env_element list
}
```

As already said, the steps correspond to rules and we will a bit later explain how they are implemented. The rules deal with dependencies on species parameters' methods. The only point dealing with dependencies on methods of `Self` is the "minimal Coq typing environment stored in the field `iai_min_coq_env` (respectively in `ai_min_coq_env`).

At the end of each rule, the compiler will record the state of dependencies on the species parameters for further usage. Finally, once all the key steps are no more needed, we merge all the computed dependencies, keeping only the 3 different sets:

- `used_species_parameter_tys` that records the list of collection parameters' names that are used by the species and hence that must be abstracted.
- `ai_dependencies_from_params` that records the dependencies on species parameters' methods that need to be abstracted to "write" (emit the code so that it is well-typed in the target language) the definition of the method.
- `ai_dependencies_from_params_for_record_type` that records the dependencies on species parameters that impose abstractions when "writing" the record type representing the species. In effect, the record type doesn't mandatorily requires all the dependencies required by the definition of a method. Note that in OCaml, since the record type only makes visible types, there is never dependencies on species parameters' methods).
- `ai_min_coq_env` the minimal Coq typing environment that describes the set of methods of `Self` that must be abstracted because of dependencies in the method.

Hence, the abstractions computation is done in 2 shots. The inner one that processes fields and create the internal abstraction structure (function `__compute_abstractions_for_fields` returning a list of `internal_field_abstractio`
We may note that this function is a "fold" since at some point, to recover the already computed dependencies from parameters on previous fields since this info will possibly used to apply rules [`DEF-DEP`], [`UNIVERSE`] and [`PRM`] of definition 72 page 153 from Virgile Prevosto's Phd. next, the outer one (the only exported outside) that is in fact a wrapper around the inner one, and that only merge the abstraction information **and sort it** (that was missing in

Virgile Prevosto's PhD) to get the "compact" representation of the abstraction for each method (i.e. it then returns a `field_abstraction_info list`).

Since the core of the computation is hosted in the inner function, we will investigate its work in detail. The outer function doesn't present any special difficulty and will be explained in a shorter way.

8.1.1 The inner computation

The basic process to apply to a method is always the same. Only `signatures` are a bit simpler because some rules do not apply (i.e. trivially lead to empty sets of dependencies). Hence, we will expose the general case, more specifically presenting the case of theorems when we need an example (since for them, no rule lead to trivially empty dependencies sets since theorems can induce both def and decl-dependencies). The function taking care of this job is `Abstractions.__compute_abstractions_for_fields`.

`compute_lambda_liftings_for_field`

The first step is to compute the dependencies on species parameters appearing in the body of the method. This is basically the rule [`BODY`]. This process is done by a structural descent on the body of the method, looking for identifiers of the form `C!meth`.

By the way, we recover the decl and def-children of the current method. The idea is simply to split the list of children of the node representing the current method in 2 parts, the ones whose edge is tagged `DepGraphData.DK_decl` and the ones whose edge is tagged `DepGraphData.DK_def`. We do this at this stage because these 2 lists will be useful later and this allows to directly compute the species parameters' carriers appearing in the type of the method. This could be done in a separate part, but that's simply our historic choice. May be the reason of this history is that before, dependency computation for OCaml and for Coq each used a pretty different algorithm. And in the OCaml code generation, since there was less things to compute, we did all in the same pass. Later, when the compilation process was better understood, we identified the common algorithm and rules and made to that now abstractions computation is exactly the same for both target languages. And then, the part dealing with def and decl-dependencies splitting remained here.

Then we really walk along the AST to find species parameters' methods called in the body of the method. This is done parameter per parameter, we don't look for methods of all the parameters in one shot. In fact, we process one parameter at time, in the order of apparition of the parameters. This is very important because this gives a particular structure (we rely on it everywhere) to our dependency on parameters information. This information is a list of parameters and for each of them the set of methods we depend on. This set structure is described in `basement/parsetree_utils.ml`. The parameters appear in the list in the same order they appear in the species definition. Hence if a method has no dependency on the species parameters, the dependency information **won't** be an empty list, but a list with all the parameters and for each an empty set. Not comply this invariant will straight lead to break the compiler (assert failure will occur in various places). Since we added termination proof, we must also walk along these proofs to find the dependencies.

During this process, we also hunt types representing species parameters **carriers** appearing in the type method. **Note on the fly**: I see that for computational methods, I inspect the ML-like type and for logical ones, I inspect the logical statement (which is really the "type" of the method. But what is a bit strange is that I do this in **types** while dealing with species parameters' methods in **bodies**. The point is not so the fact that I mix "body" and "type", but more the fact that I wonder if I also hunt later in the bodies...).

Once we are done with parameters' carriers appearing in the type of the method, we do the same thing on the methods of `Self` we decl-depend on. Note that if we have a decl-dependency on `representation`, then we do not need to inspect its structure to know if it contains references to some species parameter types since this means that the `representation` is still kept abstract.

We then do the same process for the methods of `Self` we def-depend on. Attention, if we have a def-dependency on `representation`, we must inspect its structure to know if it contains references to some species parameter since `representation`'s structure will appear in clear, possibly using these species parameters carrier types. So, conversely to just above, we don't make any difference between `representation` and other methods of ourselves.

In fact, technically, to get the set of species parameters' carriers, we get the set of carriers, and afterward, we filter those that are among our parameters. This is more efficient that testign each time before inserting or not a carrier in the set.

Finally, we return the species parameters' carriers used in the method, the dependencies from parameters found in the body of the method, the decl-children of the method in the dependency graph and the def-children of it.

`VisUniverse.visible_universe`

The next step is to compute the "visible universe" of the method. This is done by calling the function of `commoncodegen/visUniverse.ml`. This universe describes which methods of `Self` must be λ-lifted for the current method, according to the definition 57 page 116 section 6.4.4 in Virgile Prevosto's PhD. The algorithm mostly implement the rules of the definition, without any special extra comment.

The structure of the universe is simply a map of method names. If a name belongs to the keys of the map, then it is in the visible universe. The bound key is then the way method arrived into the visible universe (needed later to be able to λ-lift). A method can arrive in th iniverse either by a decl-dependency and **no** transitive def-dependency (tag `IU_only_decl`), or by at least a transitive def-dependency (tag `IU_trans_def`) and in this case, no matter if it also arrives thanks to a decl-dependency.

Completion of the dependency on parameter's methods

It is now time to apply the rules [`TYPE`], [`DEF-DEP`], [`UNIVERSE`] and [`PRM`] of the definition 72 page 153 of Virgile Prevosto's Phd. They contribute to extend the visibility a method must have on those of its parameters. Note that in term of implementation, we **don't** return the initial set of dependencies extended by the freshly found one ! We always return separate sets of dependencies related to each rule (or set of rules in the case of [`DEF-DEP`], [`UNIVERSE`] and [`PRM`] that are returned together in one set because we never need to differentiate their provenance). The rules are computed in the following order: [`TYPE`], [`DEF-DEP`], [`UNIVERS`] and then [`PRM`]. This work is done by the function `Abstractions.complete_dependencies_from_params`.

1. Rule [`TYPE`]. This rule says that we must search for dependencies on species parameters' methods among the "type" of the currently examined method of `Self`. This rule is possible only if a logical expression is provided. In effect, in a type scheme, species parameters' methods can never appear since it is a ML-like type. Furthermore, even in case of termination proof, we have nothing to do since expressions appearing have ML-like types and proofs are not considered as "type". This rule simply walk along the "type" of the currently examined method of `Self`, searching occurrences of identifiers having the form `param!meth`.

2. Rule [`DEF-DEP`]. This rule is implemented by first recovering all the abstraction infos of the methods of `Self` we def-depend. Because species are well-formed, there is no cycle in its dependencies, and because it is in normal form, the methods we depend on have already be processed and their abstraction infos are known. This rule tells if the method def-depends on a method `z` and in the body of `t` we find a dependency on a parameter's method, then this parameter's method must be added to the dependencies of the current method. Instead of implementing this rule this way, we read it like "add to the dependencies of the current method all the dependencies on parameters computed on methods we def-depend". Then, instead of looking for each method if there is individually a dependency on each species parameter's method to add, we make a big union in one shot.

   By the way we recover all the abstraction infos of the methods of `Self` we directly start reminding the species parameters' carriers appearing in the type of the methods we def-depend. In effect, by definition, the methods we def-depend belong to our visible universe. And because the rule [`UNIVERS`] just below will deal with methods of species parameters appearing in "types" of methods of `Self` belonging to the visible universe, recording these carriers soon will serve for the next rule.

3. Rule [`UNIVERS`]. This rule says that if a method z belongs to the visible universe of the method, if m has a dependency on a species parameter method y via the type of z, then we must add this dependency to the current method. To make thing faster and simpler, instead of checking if there exists a y, we directly do so that for

each z in the visible universe, we must add its `iai_dependencies_from_params_via_type` (i.e. the dependencies on species parameters we already computed for the method z. Since the methods are well-ordered, we are sure that we will find this information for z since it was mandatorily processed before.

By the way, we go on reminding the species parameters' carriers appearing in the type of the methods added in the dependencies.

4. Rule [PRM]. This one is really trickier and suffered of small typos and implicit stuff in the original PhD. The correction is presented in 1.0.10. It deals with the fact that a parameter $C_{p'}$ using a previous parameter C_p may induce dependencies on methods of C_p via its own dependencies on its own parameters (those instantiated by C_p). Hence, we are interested in computing dependencies on species parameters of a species having the shape:

$$\texttt{species}\ S(C_p\ \texttt{is}\ \ldots, C_{p'}\ \texttt{is}\ S'(C_p))$$

First, we look for `is` parameters themselves parametrised. We hunt in the `species_parameters`, to get some `Env.TypeInformation.SPAR_is` whose `simple_species_expr` has a non empty list `sse_effective_args`.

Then, we get for each parametrised parameter of the species, which other parameters it uses as effective argument in which species and at which position. This is the typical shape already given:
$\texttt{species}\ S(C_p\ \texttt{is}\ \ldots, C_{p'}\ \texttt{is}\ S'(C_p))$ We want to know that $C_{p'}$ uses C_p as first argument for the species S'. So we want to get the pair $(C_{p'}, (S', [(C_p, 1)]))$. If $C_{p'}$ used another C_q as third argument, we would get the pair: $(C_{p'}, (S', [(C_p, 1); (C_q, 3)]))$.

Now, we know that $C_{p'}$ is a species parameter built from S' applying C_p at position 0. We must find the name of the formal parameter in S' corresponding to the position where C_p is applied. Let's call it K. We have now to find all the dependencies (methods y) of K in S' and we must add them to the dependencies of C_p. This is done by checking for each affective argument and position if it is an entity or a collection parameter. in the first case, since an entity doesn't have "call-able" method we do not have any dependency to add, then we have nothing to do. In the second case, we have in our hand the effective argument used. We then get the name of the formal parameter (C_p) of S' at the position where the effective argument was used. Now, get all the z in `Deps` $(S, C_{p'})$ (that can be found in the `starting_dependencies_from_params`). Now, for all z, we must search the set of methods, y, on which z depends on in S' via the formal parameter's name. So, first we get z's dependencies information y. In these dependencies, we try to find the one corresponding to formal name . If none is found in the assoc list that because there no method in the dependencies on this parameter and then we "add" and empty set of dependencies. If we found some, before adding the dependencies, we must instantiate the formal parameter of S' by the effective argument provided. In the code, this means that we must replace `formal_name` by `eff_arg_qual_vname` in the dependencies y. In effect, in the bodies/types of the methods of S', parameters are those of S', not our current ones we use to instantiate the formal ones of S' ! To prevent those of S' to remain in the expressions and be unbound, we do the instanciation here. And finally, we add the substituted dependencies in the current dependencies accumulator.

Extra completion of the dependency on parameter's methods

Here is a point I identified as missing in Virgile's PhD. The related rule is called [DIDOU] because I didn't have any better idea while I was working on this, and it remained until somebody finds a better name ☺

This rule performs a transitive closure on the species parameters' methods appearing in types of methods already found by the previous completion rules. An open question is does this rule must also compute the fix point taking into account the methods `it` adds ? In practice I never saw such a need, but... a bit of theory wouldn't hurt☺The important point here is to understand that for each method of species parameters, we will look for it's own dependencies on methods of "its `Self`". Since in the parameter, these methods are "from `Self`", in effect, in the species declaring the parameter, they will look as methods of the species parameter.

This processing is performed by the function `Abstractions.complete_dependencies_from_params_rule_didou`. It starts by making the union of all the dependencies found by the previous rules. Then it creates a fresh empty set of dependencies that will serve as

dependencies accumulator all along the fix point iterations. Then, for each species parameter, for each method already found of this parameter, we compute the dependencies the decl-dependencies coming from **the type** of the method of this species parameter, we must add it as a dependency on this species parameter method. In fact, we only add it if it was not already present, which allows to detect the fix point reached. But, since we computed the decl-dependencies of the method (i.e. dependencies related to other methods of this species parameter which are methods of `Self` in this parameter), we must replace the occurrences of `Self` in this method by the species parameter from where this method comes.

The process iterates until no more method have been added for any of the species parameters.

Completion of the dependency on parameter's carriers

This work is done by the function `Abstractions.complete_used_species_parameters_ty`. Now, we complete the species parameters carriers seen by taking into account types of methods obtained by the completion of the dependencies on parameters achieved by the previous rules. This is simply a scan of the previously built dependencies. The only hack is that we scan the types and remind all the species carriers appearing inside. Then we finally filter to only keep the carriers that are really coming from parameters, forgetting those coming from other toplevel species or collection.

End of the inner computation

Once all the dependencies from the rules, the visible universe, the carriers we depend on are computed, we simply store them in a `Abstractions.internal_abstraction_info` structure to further create the more compact form of the dependencies. This compact form is computed as described in "The outer computation (wrapper)" (C.f. 8.1.2.

8.1.2 The outer computation (wrapper)

The wrapper is the function `compute_abstractions_for_fields`. Basically it's big `List.map` on the temporary abstractions info we computed above. Its aim is to process this temporary information to make it more compact and adjust some things due to inheritance and order of dependencies between them and to apply the rule [`DIDOU`] on dependencies that will appear in the **record type** of the species.

The first thing to do is to merge all the dependencies found by the rules [`TYPE`] and all the completion rules.

Next, we compute the dependencies used to generate the record type parameters, i.e. [`TYPE`]+[`PRM`]+([`DIDOU`] on [`TYPE`] +[`PRM`]). This is simply done like previously, using the function `complete_dependencies_from_params_rule_didou` but providing it an empty dependency set for the parameter ∼`via_body`, hence, the completion will not take any body dependency into account.

Once we get the dependencies needed for the method definition and for the record type field definition, we must order them. In effect, nothing guaranties that the order the dependencies are stored in our data-structures are consistent with the dependencies of the methods in their own species (that are, remind, our species parameters). Then we sort the methods like we did when we computed dependencies on methods of `Self`.

The final process deals with inheritance and instantiations of species parameters. If the method is inherited, it consists in mapping the computed dependencies on parameters in the current species on the dependencies on parameters previously computed in the species we inherited the method. Note that if the method is not inherited, we do nothing, returning directly the dependencies on parameters we computed until now.

We need this process to correctly compile code like:

```
species Couple (S is Simple, T is Simple) =
  signature morph: S -> T ;
  let equiv (e1, e2) =
    let _to_force_usage = S!equal in
    T!equal (!morph (e1), !morph (e2)) ;
end ;;
species Bug (G is Simple) inherits Couple (G, G) = ... end ;;
```

The formal parameters S and T are instantiated by inheritance both by G. In Couple, equiv depends on the types S and T and on the methods S!equal and T!equal. So, the application of the method generator of equiv in Bug must have twice __p_G_T and __p_G_equal provided: once for the λ-lift of S and once for the one of T in Couple (yep, remember that in Couple, the method equiv depends on 2 species parameter types, S and T and the methods S!equal and T!equal). Unfortunately, when we compute directly the dependencies in the species Bug, since we work with sets, the 2 occurrences of __p_G_T are reduced into 1, and same thing for __p_G_equal. Hence the dependencies information we have for the method equiv inherited from Couple in the species Bug gives us a wrong number of dependencies (hence of λ-lifts) compared to those required to use the method generator which lies in Couple.

One solution would have to take the dependencies information directly in the species we inherit (i.e. Couple here) and to perform the substitution replacing the formal parameters by the effective arguments provided in the inherits clause. Technically, because the data-structure representing dependency information is very complex, involves sharing, I didn't dare to do this to avoid errors, forgetting parts of the data-structure and so on, and also to preserve the sharing. In effect, the substitution returns a copy of the term. Hence, all the shared parts in the data-structure would be freshly copied, then separated of their other occurrences somewhere else in the data-structure.

So, if we have a deeper look at our problem, by re-using directly the dependency information in the species we inherits, we have the right number of abstractions (the right abstractions scheme), but without the substitutions induced by the inherits clause. On the other side, by computing directly the dependency information in the current species, we have the right substitutions, but without the correct number of abstractions (i.e. without the right abstractions scheme). One may note however that in the second point, only the number of abstractions (and their positions) are incorrect. However, nothing completely disappear: if in the inherited species we had a dependency on a parameter method, so we have in the current species. The only thing that could happen is that several methods initially identified as differentiate got merged into one same method after instantiation.

The solution is then simply to trust the abstractions scheme of the inherited species. Next, we compute the dependency information in the current species. This will give us the "dependency bricks" with the right substitution applied. And then we will just map our bricks onto the scheme according to the formal by effective parameter instantiations.

For instance, on our previous example, the dependencies (and hence the λ-lifts) computed in Couple are: S, T, S!equal, T!equal, so the method generator of equal has 4 extra parameters (and in this order).

Now, computing the dependencies in Bug gives us G, G!equal.

We see that S was instantiated by G, so in the inherited scheme we replace the dependencies information related to S by those computed in Bug, i.e. by those related to G. We see that T was instantiated by G, so we do the same thing in the inherited scheme for the dependencies information related to T. Hence, the final dependencies information we get for the method equal in Bug is G, G, G!equal, G!equal which is right according to what the method generator expects.

Note this impacts only collection parameters since entity parameters do not provide methods during the inheritance.

Once all this job ends, we have all our dependencies computed and we store all the information in a abstraction_info structure that will be exported toward the code generation back-ends.

Chapter 9

OCaml code generation

The code generation starts from the `Infer.please_compile_me` structure returned by the type-checking pass. It will examine each phrase of the program, call the abstractions computation previously described (c.f. 8) if needed, before starting generating some target code.

The most interesting parts of the code generator are those dealing with species and collection. Toplevel functions do not pose particular problems, as well as type definitions since OCaml has similar type definitions. One may note that because OCaml doesn't have logical features, toplevel theorems trivially lead to no produced code. Generation for `use` and `open` directives doesn't produce any code, only `open` has a significant effect, loading the definitions of the related module in the environment (exactly like for the other passes).

9.1 Species generation

9.1.1 The collection carrier mapping

To be able to properly map the species parameters' carriers to type variables in the generated code, we start by creating a "collection carrier mapping". This mapping is the correspondance between the collection type of the species definition parameters and the type variables names to be used later during the OCaml translation. For a species parameter `A is/in ...`, the type variable that will be used is "'" + the lowercased name of the species parameter + an integer unique in this type + "`_as_carrier`".

We need to add an extra integer (in fact, a stamp) to prevent a same type variable from appearing several time in the tricky case where a `in` and a `is` parameters wear the same lowercased name. For instance in **`species`** `A (F` **`is`** `B , f` **`in`** `F)` where `F` and `f` will lead to a same name of OCaml type variable: "'`f_as_carrier`".

Hence, each time we will need to generate a type expression involving a species parameter carrier, by a simple look-up in the mapping we will get the type variable's name to emit. This mapping primarily serves to the type pretty-print function. It also have few minor usages that will be explained when we will encounter the case.

The mapping gets stored in a compilation context (a bit like the context we already saw for type-checking) with various other structures that will be always passed to the compilation functions. This way, grouping all in one unique argument makes the code clearer.

9.1.2 The record type

As described in , a species starts by a record type definition. This record contains one field per method. This type is named "`me_as_species`" to reflect the point that it represents the OCaml structure representing the FoCaL species. Depending on whether the species has parameters, this record type also has parameters. In any case, it at least has a parameter representing "self as it will be once instanciated" once "we" (i.e. the species) will be really living as a collection.

If the carrier is defined, then before the record definition, we generate the type definition "me_as_carrier" that shown the constraints due to the representation definition (c.f. 2.1.2). This is done by the function generate_rep_constraint_in_record_type.

Next, we must start the generation of the record type itself. Be careful that it will have at least one type parameter (the one representing our carrier and named 'me_as_carrier). It can have several parameters if the species has species parameters. So, we start generating the 'me_as_carrier, and we use the collection carriers mapping to generate the other type parameters corresponding to the carriers of the species parameters.

Now comes the point where we must generate the record fields. The first weird thing is that we must extend the collections carrier mapping with ourselve known. This is required when representation is defined. Hence, if we refer to our representation (i.e. me_as_carrier), not to Self, I mean to a type-collection that is "(our compilation unit, our species name)" (that is the case when creating a collection where Self gets especially abstracted to "(our compilation unit, our species name)", we will be known and we wont get the fully qualified type name, otherwise this would lead to a dependency with ourselve in term of OCaml module.

Indeed, we now may refer to our carrier explicitely here in the scope of a collection (not species, really collection) because there is no more late binding: here when one say "me", it's not anymore "what I will be finally" because we are already "finally". Before, as long a species is not a collection, it always refers to itself's type as "'me_as_carrier" because late binding prevents known until the last moment who "we will be". But because now it's the end of the species specification, we know really "who we are" and "'me_as_carrier" is definitely replaced by "who we really are" : "me_as_carrier".

We can now iterate through the list of fields of the species, to create the record's fields. We take care to not generate let fields whose type involves prop since they can only be created by logical let that are discarded in OCaml. We also skip theorem and property fields.

Once done, we close the record definition.

9.1.3 Abstraction computation

As previously explained, each back-end triggers the computation of abstractions (i.e. things – carriers, methods – to abstract due to dependencies). From this computation we get the list of fields of the species with the information explaining what to λ-lift.

9.1.4 Definitions' code generation

We then iterate code generation on each field, depending on its kind. Signatures, properties and theorem are purely discarded in OCaml since they have no mapping and no use.

It then remains the let-definitions. The generation consists in 2 parts: the generation of the extra parameters due to λ-lifts and the translation of the function's body into OCaml. This last part is quite straightforward, so we will take more time on the first one.

The first thing is that only methods defined in the current species must be generated. Inherited methods **are not** generated again. So we start generating the the let and name of the function.

Next come the λ-lifts that abstract according to the species's parameters the current method depends on. We process each species parameter. For each of them, aach abstracted method will be named like "_p_", followed by the species parameter name, followed by "_", followed by the method's name. We don't care here about whether the species parameters is in or IS.

Next come the extra arguments due to methods of ourselves we depend on. They are always present in the species under the name "abst_...". These λ-lifts are only done for methods that are in the minimal coq environment because they computational and only declared.

Next come the parameters of the let-binding with their type. We ignore the result type of the let if it's a function because we never print the type constraint on the result of the let. We only print them in the arguments of the let-bound identifier. We also ignore the variables used to instanciate the polymorphic ones of the scheme because

in OCaml polymorphism is not explicit. Note by the way thet we do not have anymore information about `Self`'s structure...

Becareful that while printing the type of the function's arguments, since they belong to a same type scheme they may share variables together. For this reason, we first purge the printing variable mapping and after, activate its persistence between each parameter printing. This allows the type pretty-print function to "remember" the variables already seen and print the same name if it see one of them again. And this, until we release the persistence.

Finally, we can dump the code corresponding to the translation of the function's body. The only tricky part is to generate code for identifiers because we must take care of wether the identifier represents a local variable, an entity parameter, a method of `Self`, a toplevel identifier or a collection parameter's method. We must also be able to detect occurrences of recursive calls to be sure that at application time, we will really provide the arguments for the λ-lifts.

9.1.5 If the species is complete

...then we must manage the fact that a collection generator must be created. The generator is created from the list of compiled fields. The idea is to make a function that will be parametrised by all the dependencies from species parameters and returning a value of the species record type.

The generic name of the collection generator: "`collection_create`". Be careful, if the collection generator has no extra parameter then the "'`collection_create`" will not be a function but directly the record representing the species. In this case, if some fields of this record are polymorphic, OCaml won't generalize because it is unsound to generalise a value that is expansive (and record values are expansives). So, to ensure this won't arise, we always add one `unit` argument to the generator. We could add it only if there is no argument to the generator, but it is pretty boring and we prefer just to **K**eep **I**t **S**imple and **S**tupid ☺.

After the name of the generator, we must generate the parameters the collection generator needs to build the each of the current species's local function (functions corresponding to the actual method stored in the collection record). These parameters of the generator come from the abstraction of methods coming from our species parameters we depend on. By the way, we want to recover the list of species parameters linked together with their methods we need to instanciate in order to apply the collection generator. To do so, we first build by side effect the list for each species parameter of the methods we depend on (using the dependency information previously computed). Then we simply dump this list, using the naming scheme "`_p_`" + the species parameter name + "_" + the called method name. Since that also this way species parameters methods are called in the bodies of the local functions of `Self` (see below), this will really lead to bind this abstracted identifiers in these bodies.

At this point comes the moment to generate the local functions that will be used to fill the record value. We then iterate on the list of compiled fields of the species, skipping the `logical let`s, to find the method generator of the field and apply it to all it needs. "All it needs" means stuff abstracted because of dependencies on species parameters and things abstracted because of dependencies on other methods of ourselves.

To get the method generator, we must check at which inheritance level it is, in other words in which species its code was defined. To do so, since we know that the species is complete, we just need to look in the `from_history` of the field and the generator is defined in the `fh_initial_apparition` (since by construction, we record the first declaration is no definition or the definition of a method here, and subsequent apparition due to inheritance are recorded somewhere else).

We first start by abstractions for species parameters. Depending on if the method generator is inherited or directly defined in the current species, we have 2 behaviours.

In the simples case, it is defined in the current species and we just need to apply the method generator to each of the extra arguments induced by the various lambda-lifting we previously performed for species parameters: here we will not use them to λ-lift them this time, but to apply them ! The name used for application is formed according to the same scheme we used at λ-lifting time: "`_p_`" + the species parameter name + "_" + the called method name.

However, if the method is inherited, the things are more complex. We must apply the method generator to each of the extra arguments induced by the various lambda-lifting we previously in the species from which we inherit, i.e. where the method was defined. During the inheritance, parameters have been instanciated. We must track these instanciations to know to what apply the method generator.

Following parameters instantiations

This work is performed by `species_ml_generation.instanciate_parameter_through_inheritance`. We search to instanciate the parameters (`is` and `in`) of the method generator of one `field_memory` (i.e. the data-structure that represents what we know about a method that was previously generated). The parameters we deal with are those coming from the λ-lifts we did to abstract dependencies of the method described by the `field_memory` on species parameters of the species where this method is **defined**. Hence we deal with the species parameters of the species where the method was **defined** ! It must be clear that we do not matter of the parameters of the species who inherited !!! We want to trace by what the parameters of the original hosting species were instanciated along the inheritance.

So we want to generate the OCaml code that enumerates the arguments to apply to the method generator. These arguments are the methods coming from species parameters on which the current method has dependencies on. The locations from where these methods come depend on the instanciations that have be done during inheritance.

This function trace these instanciations to figure out exactly from where these methods come. Process sketch follows:

1. Find at the point (i.e. the species) where the method generator of the `field_memory` was defined, the species parameters that were existing at this point.
2. Find the dependencies the original method had on these (its) species parameters.
3. For each of these parameters, we must trace by what it was instanciated along the inheritance history, (starting from oldest species where the method appeared to most recent) and then generate the corresponding OCaml code.
   (a) Find the index of the parameter in the species's signature from where the method was **really** defined (not the one where it is inherited).
   (b) Follow instanciations that have been done on the parameter from past to now along the inheritance history.
   (c) If it is a `in` parameter then we must generate the code corresponding to the FoCaL expression that instanciated the parameter. This expression is built by applying effective-to-formal arguments substitutions.
   (d) If it is a IS parameter, then we must generate for each method we have dependencies on, the OCaml code accessing the OCaml code of the method inside its module structure (if instanciation is done by a toplevel species/collection) or directly use an existing collection generator parameter (if instanciation is done by a parameter of the species where the method is found inherited, i.e. the species we are currently compiling).

Ending the collection henerator

Finally, we must apply the method generator to each of the extra arguments induced by the methods of our inheritance tree we depend on and that were only declared when the method generator was created. These methods leaded to "local" functions defined above (by the same process we describe here). Hence, for each method only declared of ourselves we depend on, its name is "`local_`" + the method's name. Since we are in OCaml, we obviously skip the logical methods.

We were at the point of generating the "local" functions corresponding to our methods. This is now done, and we only have to create now the record value representing the collection returned by the collection generator. To do this, we only assign each record fields corresponding to the current species's method the corresponding "local" function we defined just above. Remind that the record field's is simply the method's name. The local function corresponding to the method is "`local_`" + the method's name.

9.1.6 Ending the code of a species

We ar enow done with the process handling the case a species is fully defined, hence has a collection generator. We just now need to create the data that will be recorded in the OCaml code generation environment. Remember that while creating the collection generator, we returned the list of arguments it need. This info will be part of what is stored in

the environment. We also build the list of the `compiled_field_memory`'s of the methods and the information about the species parameters.

9.2 Collection generation

Chapter 10

Coq code generation

10.0.1 Theorems and proofs

10.0.2 Recursive functions

Currently, a recursive function is handled in 3 ways in Coq depending on whether it is a:

- recusive method: usage of `Function`.
- local recursive function: usage of `fix` with assumption that the function is structurally recursive on its **first** argument.
- toplevel recursive function: usage of `Fixpoint` with assumption that the function is structurally recursive on its **first** argument.

The entry point for recursive function code generation in Coq is `generate_recursive_let_definition` from module `Species_coq_generation` .

Recursive method

```
type t = | D | C (t) ;;

species A =
  signature toto : int ;

  let rec foo(x, y) =
    match x with
    | D ->
        if syntactic_equal(y, 0) then 1
        else 1 + foo(D, (y + 1))
    | C(D) -> 2
    | C(a) -> foo(a, (y - toto)) ;
end ;;
```

Entry point: `generate_defined_recursive_let_definition_With_Function`. In this configuration, the generated code uses `Function`, creates a `Module` and a `Section` to embed the termination proof. Note that logical recursive methods are not yet handled.

```
Module Termination_foo_namespace.
  Section foo.
```

Then, is creates `Variables` for methods on which the recursive function depends (like always, using `generate_field_definition_prelude`). Since we are in a `Section` for Zenon, we tell instead of abstracting dependencies by adding extra arguments to the current definition, we generate `Variable`, `Let` and `Hypothesis`.

This prelude inserts abstractions due to the types of the species parameters, then dues to methods of the species parameters the function depends on, then finally methods of `Self` on which the function depends on.

```
Variable abst_toto : basics.int__t.
```

A `Variable` always called "`__term_order` is now generated to represent the order. It always has 2 arguments having the same type. This type is a **tuple** if the method has several arguments. We see below that our function takes 2 parameters, one of type `t` the other of type `int`. So the order takes 2 tuples of type `(t * int)`, i.e. will have to compare 2 "sets" of call arguments. It always return `Prop`.

```
(* Abstracted termination order. *)
Variable __term_order :
  (((t__t) * (basics.int__t))%type) -> (((t__t) * (basics.int__t))%type) -> Prop.
```

Then the proof obligation that this order is well-founded come. The `Variable` representing the termination proof obligation is always called "`__term_obl`". It's now time to generate the lemmas proving that each recursive call decreases. Each of then will be followed by a `/\` to make the conjunction of all of them. And the latest one will be used to add the final "`well_founded __term_order`" to this big conjunction. This is big job is performed by `generate_termination_lemmas` of the module `Rec_let_gen`.

This function takes the list (initial variable of the function * expression provided in the recursive calls). The expression must hence be $<$ to the initial variable for the function to terminate. In fact that's the tuple of initial variables that must be $<$ to the tuple of expressions provided in each recursive call. Basically, the algorithm is the following:

For each recursive call:

```
(* For each binding, bind the variable by a forall.
Done by Rec_let_gen.generate_variables_quantifications *)
    For each binding
      match binding with
        | Recursion.B_let (name, scheme) ->
          "forall %a : %a, " name (specialize scheme)
        | Recursion.B_match (_, pattern) ->
          (* Get all bound variable of the pattern. *)
          (* Generate a forall for each bound variable. *)
          "forall %a : %a, " var_name pat-var-ty
        | Recursion.B_condition (_, _) ->
          (* No possible variable bound, so nothing to do. *)
(* We must generate the hypotheses and separate them by ->. *)
List.iter
  (function
    | Recursion.B_let ->
      (* A "let x = e" induces forall x: ..., x "=" e. *)
      Rec_let_gen.generate_binding_let
    | Recursion.B_match (expr, pattern) ->
      (* Induces "forall variables of the pattern, pattern = expr". *)
      Rec_let_gen.generate_binding_match
    | Recursion.B_condition (expr, bool_val) ->
      (* Induces "Is_true (expr)" if bool_val is true,
      else "~ Is_true (expr)" if bool_val is false. *)
  )
  bindings ;
(* Now, generate the expression that tells the decreasing by applying
the "__term_order" Variable or the really defined order if we were
provided one by the argument [ explicit_order] with its arguments to
apply due to lambda-liftings. *)
```

(* Now, generate the arguments to provide to the order. They are 2 tuples,
 the first one being the one containing arguments of the recursive call,
 the second one being the arguments of the original call. *)
(* Separate by a /*)

```
  Variable __term_obl :
    (forall x : t__t, forall y : basics.int__t,
     (@D = x) -> ~ Is_true ((basics.syntactic_equal _ y 0)) ->
       __term_order ((@D ), ((basics._plus_ y 1))) (x, y))
    /\
    (forall x : t__t, forall y : basics.int__t,
     forall a : t__t,
     (@C  a = x) ->
       __term_order (a, ((basics._dash_ y abst_toto))) (x, y))
```

Above, we the 2 cases of recursive calls with in one, `__term_order` applies to $(\texttt{D}, (y+1))$ and (x, y) and in the other case, applied to $(a, (y - \texttt{toto}))$ and (x, y).

```
    /\
     (well_founded __term_order).
```

Now comes the translation of the real body of the recursive function, using the Coq `Function` construct. We generate its arguments as a tuple if there are several. Before the return type is inserted the verbatim code "`{wf __term_order __arg}`". We need then to recover the individual arguments of the function. Unfortunately, we can't simply generate `let (x, y, ..) := __arg` because Coqonly allows pairs as let-binding pattern. So instead, we generate a `match`.

```
  Function foo (__arg: (((t__t) * (basics.int__t))%type))
    {wf __term_order __arg}: basics.int__t
    :=
    match __arg with
      | (x,
      y) =>
```

Before being able to generate the effective body of the function, we must transform the recursive function's body so that all the recursive calls send their arguments as a unique tuple rather than as several arguments. This is because we "tuplified" the arguments of the recursive function in order to be able to exhibit a lexicographic order if needed. This job is done by `transform_recursive_calls_args_into_tuple` of the module `Rec_let_gen`. Once we got this modified body expression, we can now generate the code, telling that we must not apply recursive calls to the extra arguments due to lambda-liftings because we are in the scope of a `Section` containing `Variables` representing dependencies. Code generation for this body is as usual, using of the module `generate_expr Species_record_type_generation`.

```
      match x with
       | D  =>
           (if (basics.syntactic_equal _ y 0) then 1
            else (basics._plus_ 1 (foo ((@D ), ((basics._plus_ y 1))))))
       | C D  => 2
       | C a => (foo (a, ((basics._dash_ y abst_toto))))
       end
      end.
```

Now comes the termination proof. We first enforce `Variables` to be used to prevent Coqfrom removing them (we generate `assert` for this sake). Anf finally we apply `magic_prove` for each recursive call + once for the `(well_founded __term_order)`. In our example, this is 2 recursive calls (so 2 proof goals) + 1 = 3.

```
    Proof.
      assert (__force_use_abst_toto := abst_toto).
      apply coq_builtins.magic_prove.
      apply coq_builtins.magic_prove.
      apply coq_builtins.magic_prove.
      Qed.
```

We must now create the curried version of the function, i.e. having several arguments and not 1 being a tuple. Remember that to be able to create an order, we transformed the initial (and "real") function by making it taking a tuple of arguments instead of several arguments. It's then now time to put thing back in place because code using this function relies on it having several arguments and not one tuple of arguments ! Finally we close the open `Section` and `Module`.

```
      Definition A__foo x  y := foo (x, y).
      End foo.
    End Termination_foo_namespace.
```

TO BE CONTINUED.

```
  Definition foo (abst_toto : basics.int__t) :=
    Termination_foo_namespace.A__foo abst_toto coq_builtins.magic_order.
```

Local recursive function

```
species B =
  let bar(a, b : int) =
    let rec rec_bar(x, y) =
      match x with
      | D -> 0
      | C(z) -> 1 + rec_bar(z, y) in
    rec_bar(a, b) ;
end ;;
```

They are trivially compiled into a `let fix` construct, hard-wiring that the the structurally decreasing argument is the first one.

```
Definition bar (a : t__t) (b : basics.int__t) : basics.int__t :=
  let fix rec_bar (__var_a : Set) (x : t__t) (y : __var_a) {struct x} :
    basics.int__t :=
    match x with
     | D  => 0
     | C z => (basics._plus_ 1 (rec_bar _ z y))
     end
  in (rec_bar _ a b).
```

Toplevel recursive function

The function:

```
let rec gee(x, y) =
    if syntactic_equal(x, 0) then 1
    else 1 + gee((x - 1), (y + 1))
;;
```

into:

```
Fixpoint gee (x : basics.int__t) (y : basics.int__t) {struct x} : basics.int__t :=
  (if (basics.syntactic_equal _ x 0) then 1
   else (basics._plus_ 1
              (gee ((basics._dash_ x 1)) ((basics._plus_ y 1))))).
```

This fails since this function is not recursive structural. Instead, consider the following recursive structural function:

```
let rec buzz(x) =
  match x with
  | A -> 0
  | C(y) -> 1 + buzz(y)
;;
```

which is compiled into:

```
Fixpoint buzz (x : t__t) {struct x} : basics.int__t :=
  match x with
   | A  => 0
   | C y => (basics._plus_ 1 (buzz y))
   end.
```

Chapter 11

Doc generation

Chapter 12

focalizedep

Chapter 13

Cadavers in the cupboard

Here is the holly section of things that are not yet done and for which we know there is still something to do. Currently, most of them are not very intricate and do not impact the general sanity of the compiler. However, it is possible to create some cases of programs falling in these open points. Here are the cadavers we hide under the carpet... ☺

- The verification that `Self` is really compatible with the list of species types encountered during species expression typing is not currently done. This deals with inheritance from parametrised species applied to `Self` as described in 7.3.3. Most of the points in the code are tagged by a comment `[Unsure]` and involve an identifier named `self_must_be` to represent the unused list of these species types `Self` must be compatible with. In fact, the function performing compatibility test already exists and is called `is_sub_species_of` in the source file `typing/infer.ml` but we do not use it for this task.
- Recursive functions with termination proofs using Coq's `Function` construct are to be finished. I have some notes about this and must scan them instead of rewriting all. *Apriori*, there is not tons of work to do for this and major identified points are:
  - Adding α conversion in some parts of the generated code to prevent name conflicts in Coq.
  - Rewrite the `LTac` of William that should provide a mean to prove well foundation of orders from simple basic orders.
  - Instead of using `magic_order` at the end of the proof, use the defined order (can be done only when the above `LTac` will work since it will provide the well foundation of the order.
  - Verify the `measure` kind of proof and implement the `structural` kind based on the primary brick `order`.
  - Provide a way for the user to see the proof he has to do for termination by showing him the theorems generated by the compiler for this termination proof.
- Check record label exhaustivity when dealing with expressions of type record. Currently the compiler only issues a warning to say that it is not done.
- Have an automated and transparent renaming mechanism to prevent identifiers used in FoCaLize to make syntax errors in the target languages if they are token of these languages. For instance, defining an identifier "`module`" will make a syntax error in the generated OCaml code since "`module`" is a keyword in OCaml.
- Local collections are not yet re-implemented. One must understand the semantics of the structure we want before.
- Implement an effective API to allow passes to be added by users after scoping and typechecking and re-analyse the possibliy (by these user-passes) modified code. To prevent a useless re-analyse if no pass was inserted, the API could provide a function toggling an internal boolean telling if the code must be analysed again for example.

- And most generally, the few points in the code tagged by a comment `[Unsure]`. These are the points where I strongly wondered without having yet found a solution in which I have a strong, indubitable confidence.
- The notion of "`local`" `let` is not implemented. Some questions about its semantics... Should it be inherited, i.e. visible in the children ? but not when the species is used as parameter ? as collection ? Does it appear in the signature ? What are the usage restrictions (ok or not in proofs) ? Does it lead to code ? Depending on these points, one must be careful to prevent dependencies on a `local` that would not visible. What does it means to have a "`local`" `signature`, `theorem`, `property` ?
- Change the patterns so that they are as simple as those that Coq can handle. We don't want to keep our OCaml-like patterns and decompose them into simpler patterns to prevent the user from trying to make proofs on a code that is not really the one he wrote.
- Records expressions are not generated in Coq.
- Unification currently implements the "`Self` preference" rule (rules `[Self1]` and `[Self2]` in Virgile's Phd section 3.3, definition 9 page 27. It seems to be a hack to tend to introduce the maximum of `Self` but doesn't seem to lead to a general type. In fact, this problem is circumvented by the notion of signature matching that allows to verify that the infered type of a method is compatible with the given signature of the method and keep as result the given signature. It may be possible to do so that the unification doens't return any type, hence leading to a regular ML-like unification, the signature matching process acting alone to abstract types (that are compatible with the known definition of the carrier) into `Self`.
- We should generate ".mli" interface files to prevent visibility of internal structure of species if manually hacking (interfacing) OCaml code with source generated by focalizec.
- Question: `proof by type Self`. What does it do ? What does it mean ?
- Feature allowing to use a fully defined species instead of a collection ? Wanted or not ? What's about the abstraction ? Remove ?!
- Error: Unexpected error: "Failure("Instantiation of collection parameter by Self - Configuration currently not available.")" (`from commoncode_gen/misc_common.ml`).

```
open "basics" ;;

species B =
  let x = 1 ;
end ;;

species S (A is B) =
  representation = A ;
  let x = 6 ;
end ;;

species C inherits S (Self) = end ;;
```

or

```
species Basic_object =
  let print = "" ;
end ;;

species One (A is Basic_object) =
  representation = A;
  let b = 1 ;
end;;

species Two =
  inherit One (Self);
end;;
```

- Species parametrised by `Self` are currently handled by delaying the verification that `Self` is really compatible with all the interfaces encountered in the `inherits` clause involving `Self` as effective argument of a collection. May be this verification shoud be done on the fly, considering only the interface of `Self` we can build with the methods we currently know for `Self`. **Question**: Ok, rrrrright, but is it possible to check interface compliance although we do not have yet a normal form of ourselves ?
- The focalizec command should be enhanced upon producing an executable instead of stopping only on object files (.cmo).
- On Eric's request: what to do of such a program (that compiles in FoCaL, OCaml and Coq) but is a bit weird.

```
species Subset(Val is Superset) =
  signature empty : Self ;
end ;;

species Subset (Val is Superset) =
  inherit Subset_Comp (Val) ;
  representation = Val -> bool ;
  let empty(v in Val) = false ;
end ;;
```

  In effect, `empty` is first declared as a constant, then it is implemented as a function, just thanks to the fact that `Self` is known to be equal to a functional type.

Index