Compiling Programs and Proofs: FoCaLiZe Internals

François Pessaux, Damien Doligez

Compiling Programs and Proofs: FoCaLiZe Internals

François Pessaux*

ENSTA ParisTech - U2IS, 828 bd des Maréchaux, 91120 Palaiseau, France

Damien Doligez

INRIA Paris - Gallium, 2 rue Simone Iff, 75589 Paris Cedex 12, France

Abstract

Designing a tool to ease the development of high-level security or safety systems must consider to facilitate not only design and coding but also formal demonstrations of correctness and compliance to standards. This entails some requirements on the tool as these demonstrations ask to link together computational and logical aspects of the development. These requirements are briefly considered and a solution is proposed: functions, statements and proofs are handled in a unique language, offering inheritance and parametrized modules. The FoCaLiZe environment implements this language, which remains simple enough to be used in a usual engineering process. The code generation produces an executable functional code (in OCaml) and a checkable term of a logical Type Theory (verified by Coq), close enough to truly ease traceability. It ensures that OCaml and Coq produced codes are error-free and provides compact generated code.

The main contribution of this paper is a detailed presentation of the compilation scheme, which is supported by an original treatment of the dependencies induced by the combination of computational and logical constructs. As the whole source code is translated to a logical term verified by Coq, we get a strong assurance in the correctness of the generated code, hence avoiding the need to prove correctness of the compiler itself.

Keywords: Formal Method, Compilation, Dependency Analysis, FoCaLiZe, Coq

1. Introduction

Formal methods have demonstrated their usefulness in software engineering, at least for high safety or security systems. But formal methods have not totally spread upon the software industry: while it is widely recognized that they decrease debugging

*Corresponding author

Email addresses: `francois.pessaux@ensta.fr` (François Pessaux), `damien.doligez@inria.fr` (Damien Doligez)

time by a large amount, they also have the reputation of increasing development time. A way to encourage the use of formal methods is to provide a dedicated development environment, which integrates formal methods all along the development process. An ideal development environment should at least address together specification, design, programming, and proving in a unified language to ease the formal development process. It should also provide checkable verifications/proofs to ease the assessment process, and obviously it should generate runnable code. There already are environments offering some of these features. One can cite Z [1], Maude [2], Dafny [3, 4], Why3 [5, 6, 7] and B [8], largely used in transports.

Theorem provers also provide such environments as they allow declaring specifications and functions and relating them, even by extracting functions from proofs. One can cite Isabelle [9, 10, 11], Nuprl [12], PVS [13, 14] and Coq[15]. For example, CompCert [16, 17, 18] is developed with Coq and the resulting compiler is obtained using extraction.

All these tools still require some real expertise and not all developers are really acquainted with them [19]. Of course, there is no hope of using formal methods without some knowledge of logics but one cannot expect that most developers know more than first-order logic. Thus an ideal environment should allow writing code, properties and their proofs in a FOL style, possibly with features easing programming (inheritance, parametrization, etc.). Paradoxically, it should also allow to link external code and to state some properties as axioms, for example contracts on external code or physical characteristics. It is clearly unsafe from a logical point of view, must be documented and submitted to a severe review process. But it is unavoidable in real life developments.

The FoCaLiZe [20] environment is an attempt to bring a common answer to two aspects of formal methods integration. The first one addresses developers needs, specially when a high level of safety is required. The second one is directly linked to the safety and security assessment process, as required by regulation authorities before commissioning. FoCaLiZe developments contain at the same level logical properties, programming features (pure functional, i.e. without imperative constructs) and mechanized proofs. The compiler emits both executable code and checked proofs together with automatic documentation facilities and ensures strong links between these outputs to increase confidence and to ease assessment. This is achieved by a common trunk of compilation until the concrete computational and logical syntaxes are emitted. Since some safety standards [21, 22] require source code and proof code review by assessors before commissioning, links between source code, object code, properties and proofs must remain very visible at the syntax level. An important point is that logical and computational parts of a program depend on each other. First, these dependencies must be controlled to avoid logical inconsistencies and production of code depending on proofs. Second, dependencies may serve to increase code sharing. Managing these two points is the cornerstone of the compilation process and is the main subject of the paper. They guarantee that the OCaml and Coq emitted codes contain no errors (i.e. any error detected by the target compilers is due to a bug of the FoCaLiZe compiler).

FoCaLiZe has already been used to develop a library of mathematical algebraic structures up to multivariate polynomial rings, a library of security policies [23] and to certify airport security regulation [24].

The remaining of this paper is organized as follows. Section 2 recalls our requirements on the design of the language and explains the choices underlying it. Section 3 gives a first approach of the dependency analysis. Section 4 introduces the syntactic elements of the language and some preliminary definitions (4.1), then presents the formalization of the normalization (4.5) and the typechecking (4.6) of species. Section 5 explores various code generation models and justifies the adopted one, which aims at maximizing code sharing and traceability between source and generated code and proofs. Then is sketched the compilation process, which deeply relies on λ-lifting techniques. This demonstrates the need for a more complex dependency calculus. Section 6 addresses the complete dependency calculus. The code generation is formalized in 7. Section 8 discusses some other possible choices and why we did not adopt them. Finally, a comparison with related works is done in Section 9.

Although definitions are given in a as formal as possible language, the paper intentionally contains no theorem to keep it shorter. The focus is put on a precise description of all the aspects of the compilation process, to ease some reuse in other projects. The study of dependencies was first presented in [25]. We give here a completely revised and extended presentation.

2. Analyzing the design of FoCaLiZe

FoCaLiZe is a "laboratory language" designed to study how to help the development of high integrity software from the double sight of developers and assessors. It is also conceived to be suitable for extensions, experiments, in order to test and progress toward constructs and analyses to make "formal development" easier. This section presents some design choices we considered and the adopted solutions.

The idea is to keep the language simple, pretty close to programming languages usually practiced by engineers, providing commonly used flavors like inheritance, modularity but also allowing to deal with formal properties and proofs. Logical aspects of the language are tried to be kept simple, with human-readable proofs (not only checkable by a machine), making possible interfaces with theorem provers to help the user in carrying on her/his proofs.

2.1. Choice of the Semantical Framework

High level security/safety standards such as EN-61508, CC2[21, 22] ask specifications to be as much as possible formally stated, that is, expressed in a logical language. And the assessment process requires a — as formal as possible — proof that these specification (or design) requirements are satisfied by the implementation.

If the semantics of the specification and programming languages differ too much, errors can arise when mapping logical concepts onto programming ones or vice-versa. These inconveniences can be avoided by choosing a single semantical framework for code, specifications and proofs. This choice is influenced by the one of the programming style which constrains a lot the logical style.

Targeting an imperative language usually leads to the use of Hoare logic style (separation logic, B logic…) to express properties. But proofs cannot freely unfold functions and procedures due to the possible side-effects. This is a real drawback.

Targeting a functional language can ease the choice if the needed side-effects (at least inputs-outputs) can be easily isolated into dedicated modules and separately handled. Then, in any first-order or higher-order logic, due to their referential transparency, functions (of the pure part) can be consistently unfolded when making proofs and thus, can be used without restriction to express properties and proofs. Properties of functions making side-effects can be separately demonstrated and rendered as contracts. For example, ERLANG and SCADE adopt this functional approach, however without logical counterpart. Note that the LAFOSEC study [26], ordered by the French National Security Agency (ANSSI), recommends the use of functional languages to develop high-level cyber-security tools.

Since static strong typing is known to ease error detection, we choose to target a strongly typed functional language for the generation of computational aspects. Now the kernel of any strongly typed functional language is a typed λ-calculus, which is a sub-language of any Type Theory, the correspondence between computations and logics being given by the Curry-Howard isomorphism. It relates programming types with types of the theory. Thus, we target a Type Theory as logical language. We choose OCaml (to get an executable program) and Coq (to check consistency of the global development).

2.2. Incremental Specification and Development

We now discuss the features to be offered to the user within the retained semantical framework. To illustrate coming discussions, we gradually introduce our running example. It is a "monitor" which inputs a value and outputs a validity flag indicating the position of the value with regards to two thresholds, and this value or a default one in case of problem. It is a simplification of a generic voter developed in FoCaLiZe for critical software [27].

Notation. Technical words denoting features and concepts have their first occurrence written in italic while keywords are written in bold.

To serve incremental development and reusability, FoCaLiZe allows inheritance, i.e. the ability of introducing, at any stage of the development, properties, functions and proofs. The rule of visibility is kept simple (no tags *private*, *friend*, *member*, etc.): all inherited elements are visible. Multiple inheritance is available and the choice of the last mentioned element is done in case of name conflict.

The formalized parts of specifications are rendered into (roughly speaking) first-order statements, called here *properties* (**property**), which will receive *proofs* (**proof of**). *Declarations* (**signature**) introduce names and types of functions. *Definitions* (**let**) give functions bodies. *Theorems* (**theorem**) are methods embedding a property and its proof at once.

Now, only the name of a function is needed to express a (specification) property on it and its body must be known only when unfolded in certain proofs. Thus declarations and definitions may be given separately. And a given function may receive different definitions, enhancing reusability (but function types, which are a true part of functions specification, must be kept). However, because definitions can be unfolded in proofs, each effective definition must be known at compile-time. This static resolution of methods is sometimes called *early-binding* and the proof management must consider this point (studied further).

Structured data types and pattern-matching are known to ease the management of complex algorithms. We choose a type language *à la* ML (product and union types, pattern-matching) with a slight different handling of polymorphism (see Section 2.3). We are now ready for a first example.

```
species Data =
  let id = "default" ;
  signature fromInt :
    int -> Self ;
end ;;
```

```
species OrdData =
  inherit Data ;
  signature lt : Self -> Self -> bool ;
  signature eq : Self -> Self -> bool ;
  let gt (x, y) =
    ~~ (lt (x, y)) && ~~ (eq (x, y)) ;
  property ltNotGt :
    all x y : Self, lt (x, y) -> ~ gt (x, y) ;
end ;;
```

The component `Data` (introduced by **`species`**) simply defines an "identifier" `id` and declares a function `fromInt` converting an integer to a value of the type `Self` which denotes a not yet known internal representation of `Data`. The *species* `OrdData` inherits from `Data`, it declares two functions (`lt` and `eq`), defines a derived function `gt` and states a property `ltNotGt` which uses both declared and defined names. Note the use of `~~`, the "not" operator on booleans, in contrast to `~` which applies to logical formulae.

Species are components of a modular development. They do not collect only types, declarations and definitions but also related properties and proofs. Inside a species, the manipulated data-types have a double role: a programming one used in type verification and a logical one used with the chosen Type Theory, as studied in the following. To simplify the model, all the data-types introduced in a species are grouped (by the way of product and union types) into a single data-type called the *representation* (**`rep`**). It can be just a type variable, whose name (`Self`) serves in declarations, and be instantiated by inheritance (but not redefined) to allow typing of definitions. It gives to species a flavor of Algebraic Data Type, a notion which has proved its usefulness in several formal frameworks (see [2], CASE tools).

```
species TheInt =
  inherit OrdData;
  rep = int ;
  let id = "native␣int" ;
  let fromInt (x) : Self = x ;
  let lt (x, y) = x < y ;
  let eq (x, y) = x = y ;
  proof of ltNotGt = by definition of gt property int_ltNotGt ;
end ;;
```

The species `TheInt` defines the representation (as an `int`, which is a type of FoCaLiZe standard library) and methods already declared in `OrdData`, then proves the property by unfolding `gt` and using a property found in the standard library (this proof, which could be done in `OrdData`, will be used later).

2.3. Parameterization

Parameterization is a more general way to use components material as exemplified by functors. Two kinds of parameters are often considered, the component ones and the value ones (e.g. integers modulo n). We adopt this feature with the notion of *parametrized species*. It has however to be adapted to reflect the link between a parametrized component and its parameters at the logical level. This is one of the roles of the *dependencies* of a Dependent Type Theory, hence it reinforces our choice of the target logical language.

The species IsIn below owns one *collection parameter* V (roughly speaking a component parameter) which provides the methods present in any component having at least those of OrdData. The species also owns two *entity parameters* minv and maxv whose type is the underlying **rep** of V. Calling a method of a collection parameter is done using the "bang notation": V!gt stands for the method (property) gt of the collection parameter V.

```
type statut_t = | In_range | Too_low | Too_high ;;

species IsIn (V is OrdData, minv in V, maxv in V) =
  rep = (V * statut_t) ;
  let getValue (x : Self) = fst (x) ;
  let getStatus (x : Self) = snd (x) ;

  let filter (x) : Self =
    if V!lt (x, minv) then (minv, Too_low)
    else
      if V!gt (x, maxv) then (maxv, Too_high)
      else (x, In_range) ;

  theorem lowMin :
    all x : V, getStatus (filter (x)) = Too_low -> ~ V!gt(x, minv)
  proof =
   <1>1 assume x : V,
        hypothesis H: snd (filter (x)) = Too_low,
        prove ~ V!gt (x, minv)
        <2>1 prove V!lt (x, minv) by definition of filter type statut_t
             hypothesis H
        <2>2 qed by step <2>1 property V!ltNotGt
   <1>2 qed by step <1>1 definition of getStatus ;
end ;;
```

The main question on the design of the modularity mechanism is about the representation of the component parameter: should its definition be exposed or encapsulated? There are two conflicting answers:

- Inheritance requires exposure as total encapsulation can make the development task tedious (permanent use of "getters" and "setters").
- A component "seeing" the data representation of its parameters can manipulate it without using the provided functions, hence breaking invariants, structural assumptions and theorems brought by parameters. Hence parameterization asks for abstraction. The abstraction of types, as it is usually done via module interfaces, is not sufficient since properties can still reveal the exact definition of the representation. Thus the compiler also has to forbid properties revealing it by a dependency analysis (c.f. Sections 3, 6).

Our solution is to introduce two notions of components, clearly identified at the syntax level by different names (not by attributes drown in code) with very different semantics:

- *Species*, which expose their representation, and which can be only used along inheritance during design and refinement.
- *Collections*, built from species by encapsulation of the representation, used as effective species parameters during the integration process.

To avoid link-time errors, any call to an effective species parameter imposes that all the functions exported by this parameter are already defined. To preserve consistency, all exported properties must have received proofs. Thus collections must be obtained by encapsulation of the so-called *complete species*, i.e. those which only contain definitions and theorems. The encapsulation builds an *interface* hiding the representation and only exposing the declarations of the functions and the properties of this complete species. The compiler must guarantee that all the corresponding definitions and theorems have been checked. The interface is then the only way to access them.

```
collection IntC = implement TheInt ; end ;;
collection In_5_10 =
  implement IsIn (IntC, IntC!fromInt (5), IntC!fromInt (10)) ; end ;;
collection In_1_8 =
  implement IsIn (IntC, IntC!fromInt (1), IntC!fromInt (8)) ; end ;;
```

The species TheInt, being *complete*, is submitted to encapsulation (**implement**) to create the collection IntC. IntC is used as the effective argument of IsIn's parameter V. Its method fromInt is used to provide effective values for the minv and max parameters. The species IsIn (IntC, IntC!fromInt (5), IntC !fromInt (10)) is then abstracted to create the collection In_5_10. Idem for In_1_8.

2.4. Parameterization Versus Polymorphism

Polymorphic method types are forbidden to prevent inconsistency coming from redefinition of methods along inheritance like in:

```
species Poly =
  let id (x) = x ;              species AsBool (P is Poly) =
end ;;                           rep = unit ;
                                 let elt = P!id (true) ;
species AsInt =                 end ;;
 inherit Poly ;
 rep = unit ;                   collection CInt = implement AsInt end ;;
 let id (x) = x + 1 ;           collection Err = implement AsBool (CInt) end ;;
end ;;
```

where AsInt inherits from Poly and redefines id with the type int → int. AsBool uses Poly assuming its id has type bool → bool. Both types are correct instances of the scheme $\forall\alpha.\alpha \rightarrow \alpha$. AsInt inheriting from Poly, it can be used to create a collection CInt. This collection then should be usable as an effective parameter of AsBool to create Err. However, in this latter Err!elt evaluates in true + 1 which is ill-typed.

Forbidding polymorphic methods is not a real restriction and is even wanted. A polymorphic method taking an argument of type α cannot rely on any property on this type (it can be any effective type). Conversely, let S be a species having a collection parameter C. Any method of S can rely on the properties and the functions of C. Now C

can be instantiated by any effective collection issued from a complete species, in which the representation has been defined and these properties have received proofs. Hence parameterization is a way to retrieve a kind of "polymorphism with proved properties".

For instance, a basic species implementing finite sequences of elements with a `mem` function and a theorem taking benefit from a property of its elements could be written as:

```
species FSeq (E is OrdData) =
  rep = list (E) ;
  let rec mem (e, l : Self) =
    match l with | [] -> false |  h :: q -> E!eq (h, e) || mem (e, q) ;
  theorem mem_eq_compatible :
    all l : Self, all e1 e2 : E,
      E!eq (e1, e2) -> mem (e1, l) -> mem (e2, l)
    proof = ... ;
end ;;
```

2.5. Properties and Proofs

We choose a usual first-order syntax to express properties, however built upon names of types, functions and other properties known in the logical context defined by the species construction. Our experience shows that this simple language is sufficient to express most requirements. It is indeed a sub-language of a dependent type theory, providing only certain forms of dependencies.

Proof writing is based on natural deduction as it is reminiscent of mathematical reasoning and is accessible to a non-specialist without too much effort. Then, the proof is conceived as a hierarchical decomposition into intermediate steps [28] unfolding definitions, introducing sub-goals and assumptions in the context until reaching a leaf, that is, a sub-goal which can be automatically handled by a prover. When all the leaves have received proofs, the compiler has to translate the whole tree to the target logical language and to build the context needed for checking this proof.

Solving leaves in FoCaLiZe is done by the prover Zenon [29]. It is a first-order automated theorem prover, based on the Tableaux method, developed by D. Doligez. Zenon can translate its proofs into Coq proofs, to be inserted in the appropriate context computed by the compiler.

We illustrate the decomposition with the following example. A list of steps is always ended by a `qed` step, whose goal is just the parent goal.

```
theorem t : all a b c : bool, a -> (a -> b) -> (b -> c) -> c
proof =
  <1>1 assume a b c : bool,
       hypothesis h1: a, hypothesis h2: a -> b, hypothesis h3: b -> c,
       prove c
    <2>1 prove b by hypothesis h1, h2
    <2>2 qed by step <2>1 hypothesis h3
  <1>2 qed by step <1>1
```

The proof has two outer steps `<1>1` and `<1>2`. Step `<1>1` introduces hypotheses `h1`, `h2`, `h3` and the sub-goal `c`. It is proved by a 2-steps sub-proof. Step `<2>1` uses `h1` and `h2` to prove `b`. Step `<2>2` uses `<2>1` and `h3` in order to prove `c`. Step `<1>2` ends the whole proof.

Proofs done in a given species are shared by all the species inheriting from this one. As early-binding allows redefinition of functions, proofs using an "old" definition are no longer valid. They must be detected by the compiler as soon as the redefinition is done, reverted to the property status and must be done again. This link between proofs and definitions is an example of *dependencies* between elements of the user development. Sections 3,4.4 and 6 study them.

Specification requirements such as the safety/security ones need to be proved early in the development cycle, assuming that some functional properties will be satisfied. This helps an early detection of specification errors [30]. Thus, FoCaLiZe allows to do proofs using properties not yet proved. The compiler guarantees that they will be proved later in a derived complete species. This is a way to do the proofs *just in time* and to maximize proof sharing [31].

2.6. Requirements and Choices on the Compilation Process

The target languages and the language features being chosen, it remains to conceive the *compilation process*.

Code generation has to translate the user code to a source code of the target computational language (called here simply *computational code*) and to a term of the target logical language (called here *logical code*). Computational code does not contain properties and proofs. In the logical code, data types and properties have to be translated to types of the Type Theory while definitions and proofs are translated to terms. Moreover, the user code, the computational and logical codes are to be scrutinized along the assessment process, as imposed by some safety and security standards. So the design of the compilation must ensure a good traceability between these three files. It has also to maximize code sharing between different parts of a development, to decrease the amount of code review.

The target languages' compilers typecheck the emitted codes. But on one hand these checks arrive too late. On the other hand, these errors found by the target compilers should be "de-compiled" to be conveyed to the developer. This is not an easy task. The best solution is that the target codes produced by the compiler contain no errors. We choose to include a typing pass during the compilation to early detect computational and logical typing errors and to emit comprehensive diagnostics. This is achieved by a common trunk of compilation for both target languages, the differentiation between them being done only in the very last pass of code generation, at concrete syntax emission. Logical aspects (methods, proofs, and logical dependencies "artifacts") which contribute to Coq code are simply discarded when producing OCaml code. The choice of not relying on an extraction mechanism is discussed in Section 8.

The management of inheritance and early-binding could rely on the internal mechanisms of a functional language providing these features [32] but most of Type Theories do not have them. Thus, an *ad hoc* translation into the logical target would be needed. But this choice would lead to two different compilation schemes, jeopardizing traceability. Moreover, preserving logical consistency requires a strict control of redefinitions. Therefore we decide to resolve inheritance and early-binding before code generation.

3. Dependencies in User Code

In this section we introduce a first notion of *dependencies*, one of the cornerstones of the compilation process. This notion is extended along the study of the code generation model in Section 5.2 then completed into the sections 4.4 and 6, which give a formal presentation of it. A first description of dependencies was done in [33].

In a Dependent Type Theory, terms and types may depend on terms or types. S. Boulmé [34] formalized the semantics of an early version of the language in Coq and identified two kinds of dependencies between methods. Either typing a method `m` may need to know the term corresponding to a method `n`. We call such a dependency a *def-dependency* of `m` on `n`. Or, typing `m` may only need to know the type of `n`. We call this kind of dependency a *decl-dependency* of `m` on `n`.

S. Boulmé also shown that def-dependencies, combined with other features, may introduce inconsistencies. Some of them can be avoided by syntactical restrictions: function bodies cannot contain property names nor keywords for proofs. Thus a function cannot def-depend on a proof. There remain only two possibilities. First, proofs with a `by definition of m` step (which unfolds the definition of `m`) def-depend on `m`. If `m` is redefined, these proofs must be invalidated. Second, functions and proofs can safely def-depend on the representation. But properties must not def-depend on it as explained by the following example.

The collection `Bad` encapsulates the complete species `Wrong`, its interface contains the statement of the theorem `theo`. When translated into logical code, this statement should be well-typed but typing `x + 1` in `theo` requires to know that `Self` is indeed (unifies with) `int`. The encapsulation of the representation, turning `Self` into an abstract type, prevents it. Thus, the species `Wrong` must be rejected by the compiler as typing `theo` would reveal the concrete type **`rep`** (a very bad point for security purposes, see [26]).

```
species Wrong =
  rep = int ;
  let inc (x) : Self = x + 1 ;
  theorem theo :
    all x : Self, inc (x) = x + 1 ;
collection Bad = implement Wrong ;
end ;;
```

Bad interface

```
rep : self
inc (x) : Self -> int
theorem theo :
  all x : Self, inc (x) = x + 1 ;
```

Note that function calls do not create def-dependencies and that encapsulation of collections prevents any def-dependency on methods of collection parameters. Thus, the analysis of def-dependencies must ensure that proofs remain consistent despite redefinitions and that properties have no def-dependencies on the representation (in other words, interfaces of collections should not reveal encapsulated information).

The different forms of decl-dependencies are the following. A function `m` decl-depends on a function `p` if `m` calls `p`, a property `m` decl-depends on a function `p` if the typing of `m` in the logical theory requires `p`'s type, a proof decl-depends on `p` if it contains a step `by property p` or an expression whose typing needs `p` and, recursively, `m` decl-depends on any method upon which `p` decl-depends and so on. Def-dependencies are also decl-dependencies. The method `p` can come either from the species itself or from a collection parameter. The following example gives another motivation for decl-dependencies analysis.

```
species S =
  signature odd : int -> bool ;
  let even (n) = if n = 0 then true
       else odd (n - 1) ;
end ;;
```

```
species T =
  inherit S ;
  let odd (n) = if n = 0 then false
       else even (n - 1) ;
end ;;
```

In S, even is at once declared and defined, so its type can be inferred by the typechecker, using the type of odd. Thus, even decl-depends on odd but odd does not depend on even. In T, defining odd creates a decl-dependency of odd on even and an implicit recursion between them. To keep the logical consistency, such an implicit recursion must be rejected. Recursion between entities must be declared (keyword rec). The compiler has to detect and forbid any cycle in dependencies through the inheritance hierarchy.

4. Dependencies and Normalization

A *well-formed* species must not have several methods of the same name and must not change the types of methods along inheritance. The *normal form* of a species is obtained by regrouping its explicitly given methods and its inherited ones, removing the **inherit** clauses. Computing this normal form is resolving the inheritance, however without unfolding methods (see Section 4.5). This inheritance resolution phase is important since it is required to check the consistency of the dependencies brought by inherited methods (c.f. Section 3).

Species typechecking, verification, normalization, computation of abstractions were first described by V. Prevosto [25]. We resume this presentation, adding clarifications and extensions. It serves as a basis for a complete study of dependencies and for the code generation model.

4.1. Syntax

In the following, we shortly introduce the syntax of the language, intentionally skipping constructs that do not add any specific points to the code generation model.

Definition 4.1. *Types*

$$\tau \quad := \quad \texttt{a} \mid \texttt{C} \mid \textbf{Self} \mid \textbf{prop} \mid \tau \to \tau \mid \alpha$$

◇

In addition to types *à la* ML, Self denotes the representation of the current species (defined or not), C the abstracted representation of a collection parameter and prop the type of logical statements. Basic type constructors are denoted by a. In the following, we identify the syntax of the types with the type algebra.

Definition 4.2. *Core Expressions*

$$\begin{array}{lcl} \text{long-ident} & := & \texttt{ident} \mid \texttt{upper-ident!ident} \\ \text{e} & := & \texttt{literal} \mid \text{long-ident} \mid \textbf{fun}\ \texttt{ident}(\texttt{,}\ \texttt{ident})^* \to \text{e} \\ & \mid & \textbf{let}\ [\textbf{rec}]\ \texttt{ident} = e\ (\textbf{and}\ \texttt{ident} = \text{e})^*\ \textbf{in}\ \text{e} \mid \text{e}\ (\text{e}(,\ \text{e})^*) \end{array}$$

◇

`literal` denotes the usual constants, `ident` represents basic lowercase identifiers, used to name methods, functions/properties bound variables, and `upper-ident` stands for uppercase identifiers used to name species and collections. Note that partial applications of functions should be named. The syntax forbids to nest a proposition into an expression.

Definition 4.3. *Logical Expressions (Propositions)*

$$\mathrm{p} \;\;:=\;\; \mathrm{e} \mid \mathrm{p} \vee \mathrm{p} \mid \mathrm{p} \wedge \mathrm{p} \mid \mathrm{p} \Rightarrow \mathrm{p} \mid \sim \mathrm{p} \mid \textbf{all}\ \texttt{ident}\texttt{:}\ \tau\texttt{,}\ \mathrm{p} \mid \textbf{ex}\ \texttt{ident}\texttt{:}\ \tau\texttt{,}\ \mathrm{p}$$

◇

`all` stands for the universal quantification and **`ex`** for the existential one. Note that logical expressions fully embed core expressions.

Definition 4.4. Proof Script

$$\begin{array}{lcl}
\text{proof} & ::= & \text{proof-step}^{*}\ \text{qed-step} \mid \text{leaf-proof} \\
\text{leaf-proof} & ::= & \textbf{by}\ \text{fact}^{+} \mid \textbf{conclude} \mid \textbf{assumed} \\
\text{qed-step} & ::= & \texttt{bullet}\ \ \textbf{qed}\ \text{proof} \\
\text{proof-step} & ::= & \texttt{bullet}\ \ \text{assumption}^{*}\ \textbf{prove}\ \text{p proof} \\
\text{assumption} & ::= & \textbf{assume}\ \texttt{ident:}\ \tau \mid \textbf{hypothesis}\ \texttt{ident:}\ \text{p} \\
\text{fact} & ::= & \textbf{definition of}\ \texttt{ident}^{+} \mid \textbf{hypothesis}\ \texttt{ident}^{+} \\
 & \mid & \textbf{property}\ \texttt{ident}^{+} \mid \textbf{type}\ \texttt{type-ident}^{+} \\
 & \mid & \textbf{step}\ \texttt{bullet}^{+}
\end{array}$$

◇

A script describes a hierarchical sequence of proof goals (i.e. logical expressions) with their related proof hints (combination of *assumption*s and *fact*s). A `bullet` is a syntactic element representing the nesting level and the name of a step. The **`assumed`** keyword represents an admitted proof (c.f. discussion in Section 8); **`assume`** serves to introduce typed variables as assumptions in the context of the proof.

Definition 4.5. *Species and Collection Expressions*

$$\begin{array}{lcl}
\text{sarg} & := & \text{e} \mid \texttt{upper-ident} \\
\text{se} & := & \texttt{upper-ident} \mid \texttt{upper-ident}\ (\text{sarg}^{+}) \\
\text{prm} & := & \texttt{upper-ident}\ \ \textbf{is}\ \texttt{upper-ident} \\
 & \mid & \texttt{upper-ident}\ \ \textbf{is}\ \texttt{upper-ident}\ \ (\texttt{upper-ident}^{*}) \\
 & \mid & \texttt{ident}\ \ \textbf{in}\ \texttt{upper-ident} \\
\text{field} & := & \textbf{inherit}\ \text{se}^{+} \mid \textbf{rep} = \tau \mid \textbf{signature}\ \texttt{ident:}\ \tau \\
 & \mid & \textbf{logical let}\ \texttt{ident} = \text{p} \mid \textbf{let}\ [\textbf{rec}]\ \texttt{ident} = \text{e}\ (\textbf{and}\ \texttt{ident} = \text{e})^{*} \\
 & \mid & \textbf{property}\ \texttt{ident}\ \textbf{:}\ \text{p} \mid \textbf{proof of}\ \texttt{ident}\ \textbf{:}\ = \text{proof} \\
 & \mid & \textbf{theorem}\ \texttt{ident}\ \textbf{:}\ \text{p}\ \textbf{proof}\ = \text{proof} \\
\text{s} & := & \textbf{species}\ \texttt{upper-ident}\ (\text{prm}^{*}) = \mathit{field}^{*} \\
\text{c} & := & \textbf{collection}\ \texttt{upper-ident} = \textbf{implement}\ \text{se}
\end{array}$$

◇

Note. A **logical let** definition names a function taking expressions only (not logical expressions) as arguments and returning a logical formula, moreover it **can be only applied to all its arguments**. It serves as a "macro" as shown by the definition of `reflexive`: **`logical let`** `reflexive (r)`**`=all`** `x:`**`Self,`** `r (x, x )` allows to state the properties `reflexive (leq)`, `reflexive (geq)`, ... No quantification is available over names introduced by **logical let**, which types are functions to `prop` (not a logical term). Thus, such definitions bring no higher-order logic features and their applications receive the same treatment as expressions of type `prop`. So they are not further considered and, as any logical method, they generate no code in OCaml.

Definition 4.6. *Names of a Field*

Let ϕ be a field, the set of bound names introduced by ϕ, written $\mathcal{N}(\phi)$, is defined by:

$$\begin{array}{lcl}
\mathcal{N}(\textbf{let } m \textbf{ = } e) = \mathcal{N}(\textbf{signature } m\textbf{: } \tau) = \mathcal{N}(\textbf{logical let } m \textbf{ = } p) & = & \{m\} \\
\mathcal{N}(\textbf{let rec } m_1 \textbf{ = } e_1 \textbf{ and } ... \; m_n \textbf{ = } e_n) & = & \{m_1; \ldots; m_n\} \\
\mathcal{N}(\textbf{rep}) = \mathcal{N}(\textbf{rep = } \tau) & = & \{\texttt{rep}\} \\
\mathcal{N}(\textbf{property } m \textbf{ : } p) = \mathcal{N}(\textbf{theorem } m \textbf{ : } p \textbf{ proof = } proof) & = & \{m\} \\
\mathcal{N}(\textbf{proof of } m \textbf{ = } proof) & = & \{m\}
\end{array} \quad \diamond$$

The difference between a field and a method appears with a **`let rec`** field which possibly introduces several methods. Note the constant name `rep` denoting the representation.

4.2. Preliminary Definitions

Definition 4.7. *Defined Names of a Field*

Let ϕ be a field, the set of defined names introduced by ϕ, written $\mathcal{D}(\phi)$, is defined by:

$$\begin{array}{l}
\mathcal{D}(\textbf{signature } m\textbf{: } \tau) = \mathcal{D}(\textbf{rep}) = \mathcal{D}(\textbf{property } m \textbf{ : } p) = \varnothing \\
\text{Otherwise, } \mathcal{D}(\phi) = \mathcal{N}(\phi)
\end{array} \quad \diamond$$

The complete typechecking process of a species S is the construction of a **normal form**. It first computes the **flat form** of S, then **typechecks** its fields and finally ensures its **well-formedness**. This global process can be refined into 8 consecutive steps detailed in the next sections:

1. Typechecking parameters;
2. Typechecking the **`inherit`** clause;
3. Typechecking the fresh methods of the species;
4. Appending inherited methods and methods defined in the species;
5. Flattening the species (resolve inheritance and early-binding to determine which are the effective methods present in the species, collapsing **`proof of`** fields with their related logical properties);
6. Ensuring that methods do not have a polymorphic ML-type;
7. Computing the def and decl-dependencies;
8. Ensuring that the obtained species is well-formed.

Definition 4.8. *Species in Flat Form*

A species S is said in *flat form* if all the following conditions are satisfied:

- It contains no **`inherit`** clause.
- It contains no **`proof of`** field (i.e. all of them have been combined with a related **`property`** to lead to a **`theorem`**, see definition 4.26).

$\diamond$

In such a species, inheritance and selection among available methods (early-binding) have been resolved (c.f. 4.5). This implies that its fields do not introduce several times a same name: $\forall i, j,\ i \ddagger j \Rightarrow \mathcal{N}(\phi_i) \cap \mathcal{N}(\phi_j) = \emptyset$.

Definition 4.9. *Normal Form of a Species*

A species S is said in *normal form* if all the following conditions are satisfied:

- It is in flat form;
- Its fields are well-typed and not polymorphic (however, they can contain "type variables" quantified at the species level);
- It is well-formed, i.e. a field only depends on names introduced in previous fields (c.f. 4.25). In other words, fields are ordered (by the compiler) according to their dependencies which cannot be circular (except between explicitly **`let rec`**-bound methods).

$\diamond$

4.3. Type of Fields, Species and Collections

We denote by Φ the field ϕ annotated by its (inferred) ML-type τ. The type of a species is defined only for a species in normal form.

Definition 4.10. *Type of Species*

$$ts \quad := \quad \{\overrightarrow{\Phi}\} \ \ | \ \ (C \ \textbf{is} \ \{\overrightarrow{\Phi}\}) \ ts \ \ | \ \ (C \ \textbf{in} \ \tau) \ ts$$

$\diamond$

The type of a species, ts, is represented by the type of its parameters (C's), followed by the list of the typed fields ({ $\overrightarrow{\Phi}$ }) contained in the normal form of the species. We call *atomic species type* a type without parameters, i.e. of the form $\{\overrightarrow{\Phi}\}$.

Notation. In the rest of this paper, we use a covering arrow to denote a finite ordered sequence of items: $\overrightarrow{\bullet}$ stands for a sequence of "$\bullet$" of the form $\bullet_1; \ldots; \bullet_n$.

When collection and entity parameters (c.f. 2.3) are not distinguished, we shorten C **is** t or C **in** τ by the notation $C \blacksquare t$. For brevity, cascading parameters $(C_1 \blacksquare t_1)((C_2 \blacksquare t_2)((C_3 \blacksquare t_3)ts))$ are shortened by a comma-separated notation: $(C_1 \blacksquare t_1, C_2 \blacksquare t_2, C_3 \blacksquare t_3)ts$.

Definition 4.11. *Type of Collection*

$$tc \quad := \quad \langle \overrightarrow{m : \tau} \rangle$$

$\diamond$

The type of a collection, tc, is the list of the names and types of the methods present in the **normal form** of its underlying complete species.

Definition 4.12. *Typechecking Environments*

- $\Gamma_S : (S \mapsto ts)$ or $(C \mapsto tc)$ maps a species name S onto its species type and a collection name C onto its collection type.
- $\Gamma_e : (x \mapsto \forall\overrightarrow{\alpha}.\tau)$ maps an identifier or a method x onto its type-scheme.

$\diamond$

4.3.1. Core Expressions Typechecking

The typechecking of expressions is based on a slightly modified Hindley-Milner algorithm [35, 36, 37]. As usual, the typing rules rely on two usual functions, Gen and $\leqslant$. The inference algorithm and fields fusion (c.f. Section 4.26) rely on a slight modification of the most general unifier of two types Mgu.

In the coming definitions, all the rules are given to provide a self-contained presentation, but those not differing from a usual type system *à la* ML are written in smaller fonts.

Definition 4.13. *Free Type Variable, Type Generalization and Type Scheme Instantiation*

- The set FV of free type variables in a type or a type scheme is defined by :

$$\begin{array}{lcl} FV(\mathtt{C}) = FV(\textbf{Self}) = FV(\textbf{prop}) & = & \varnothing \\ FV(\alpha) & = & \{\alpha\} \\ FV(\tau_1 \to \tau_2) & = & FV(\tau_1) \cup FV(\tau_2) \\ FV(\forall\alpha_1 \ldots \alpha_n.\tau) & = & FV(\tau) \backslash \{\alpha_1 \ldots \alpha_n\} \end{array}$$

- The set of of free type variables in a typing environment is defined by :

$$FV(\Gamma_e) = \bigcup_{x \in Dom(\Gamma_e)} FV(\Gamma_e(x))$$

- The type generalization creates a type scheme from a type τ by universally quantifying the unconstrained variables in the environment Γ_e. It is defined as usual by:

$$Gen(\tau, \Gamma_e) = \forall\alpha_1 \ldots \alpha_n.\tau \text{ with } \{\alpha_1 \ldots \alpha_n\} = FV(\tau) \backslash FV(\Gamma_e)$$

- τ' is an instance of the type scheme $\sigma = \forall\alpha_1 \ldots \alpha_k.\tau$ (and we write $\sigma \leqslant \tau'$) if there exists a substitution φ from type variables to types such as $Dom(\varphi) = \{\alpha_1 \ldots \alpha_k\}$ and $\tau' = \varphi(\tau)$.

$\diamond$

We modify the usual notion of most general unifier [38] in order to keep the representation visible for some steps of the unification process and hide it when types need to be explicited by the compiler (when building interfaces of collections for example).

We first introduce an auxiliary function, $Mg(\tau_1, \tau_2)$ to determine if τ_1 and τ_2 are unifiable under the hypothesis that the representation of the species is either τ_{rep} if defined or `Self` otherwise. Trying to unify two types may require to know what `Self` is equivalent to (i.e. what definition is assigned to the representation) as shown in the following definition:

$$[\textsc{MgId}] \qquad \tau_{rep} \vdash Mg(\tau, \tau) = \tau, id \ \text{ if } \tau \neq \tau_{rep}$$

$$[\textsc{MgSelfL}] \qquad \tau_{rep} \vdash Mg(\texttt{Self}, \tau_{rep}) = \texttt{Self}, id$$

$$[\textsc{MgSelfR}] \qquad \tau_{rep} \vdash Mg(\tau_{rep}, \texttt{Self}) = \texttt{Self}, id$$

$$[\textsc{MgVarL}] \qquad \tau_{rep} \vdash Mg(\alpha, \tau) = \tau, [\alpha \leftarrow\!\shortmid \tau] \ \text{ with } \alpha \notin FV(\tau)$$

$$[\textsc{MgVarR}] \qquad \tau_{rep} \vdash Mg(\tau, \alpha) = \tau, [\alpha \leftarrow\!\shortmid \tau] \ \text{ with } \alpha \notin FV(\tau)$$

$$[\textsc{MgArr}] \qquad \frac{\tau_{rep} \vdash Mg(\tau_1, \tau'_1) = \tau''_1, \theta_1 \qquad \tau_{rep} \vdash Mg(\theta_1(\tau_2), \theta_1(\tau'_2)) = \tau''_2, \theta_2}{\tau_{rep} \vdash Mg(\tau_1 \to \tau_2, \tau'_1 \to \tau'_2) = \tau''_1 \to \tau''_2, \theta_2 \circ \theta_1}$$

Note that no type apart τ_{rep} is unifiable with **Self**. The rules [MgSelfL] and [MgSelfR] return the type `Self` although they could return τ_{rep}. This choice allows to keep the representation hidden. As usual, the cases not handled by the rules fail.

Definition 4.14. *Most General Unifier and Unification*

$Mgu(\tau_1, \tau_2)$, the most general unifier of τ_1 and τ_2, is defined by using Mg as follows. If the computation $\Gamma_e(\textbf{rep}) \vdash Mg(\tau_1, \tau_2)$ returns τ_3, θ, then $Mgu(\tau_1, \tau_2) = \theta(\tau_3)$.

◇

Definition 4.15. *Typechecking Rules for Expressions*

Typechecking computational expressions follows the usual rules with only the call to (species or collection) methods added.

$$\frac{\Gamma_e(m) = \forall \overrightarrow{\alpha}.\tau' \qquad \forall \overrightarrow{\alpha}.\tau' \leqslant \tau}{\Gamma_S, \Gamma_e \vdash \texttt{Self}!m : \tau} [\textsc{SelfM}]$$

$$\frac{\Gamma_S(C) = \langle \overrightarrow{m_i : \tau_i} \rangle \qquad m : \tau \in \overrightarrow{m_i : \tau_i}}{\Gamma_S, \Gamma_e \vdash C!m : \tau} [\textsc{CollM}]$$

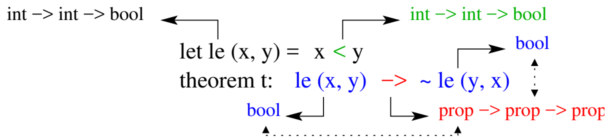
$$\frac{\Gamma_e(x) = \forall \overrightarrow{\alpha}.\tau' \qquad \forall \overrightarrow{\alpha}.\tau' \leqslant \tau}{\Gamma_S, \Gamma_e \vdash x : \tau} [\textsc{Var}] \qquad \frac{\Gamma_S, \Gamma_e \oplus (x_1 : \tau_1) \oplus \ldots \oplus (x_n : \tau_n) \vdash e : \tau}{\Gamma_S, \Gamma_e \vdash \textbf{fun}\ x_1, \ldots, x_n \ \textbf{->} e : \tau_1 \to \ldots \to \tau_n \to \tau} [\textsc{Fun}]$$

$$\frac{\Gamma_S, \Gamma_e \vdash e : \tau_1 \to \ldots \to \tau_n \to \tau \qquad \Gamma_S, \Gamma_e \vdash e_1 : \tau_1 \qquad \ldots \qquad \Gamma_S, \Gamma_e \vdash e_n : \tau_n}{\Gamma_S, \Gamma_e \vdash e(e_1, \ldots, e_n) : \tau} [\textsc{App}]$$

$$\frac{\Gamma_S, \Gamma_e \vdash e_1 : \tau_1 \qquad \Gamma_S, \Gamma_e \oplus (x : Gen(\tau_1, \Gamma_e)) \vdash e_2 : \tau_2}{\Gamma_S, \Gamma_e \vdash \textbf{let}\ x = e_1 \ \textbf{in}\ e_2 : \tau_2} [\textsc{Let}]$$

$$\frac{\Gamma_S, \Gamma_e \oplus (x_i : \forall \emptyset.\tau_i) \vdash e_i : \tau_i \qquad \Gamma_S, \Gamma_e \oplus (x_i : Gen(\tau_i, \Gamma_e)) \vdash e : \tau}{\Gamma_S, \Gamma_e \vdash \textbf{let rec}\ x_1 = e_1 \ \textbf{and} \ldots x_n = e_n \ \textbf{in}\ e : \tau} [\textsc{LetRec}]$$

Logical expressions are expected to have the type representing logical properties `prop`, which differs from the type of boolean values `bool`. Since logical expressions fully embed computational ones, a conversion from `bool` to `prop` is allowed by the rule [REGEXPR] where the inferred `bool` is turned into `prop` as illustrated by the following figure.



$$\frac{\Gamma_S, \Gamma_e \vdash e : \texttt{bool}}{\Gamma_S, \Gamma_e \vdash e : \texttt{prop}}[\textsc{RegExpr}] \qquad \frac{\Gamma_S, \Gamma_e \vdash p : \texttt{prop}}{\Gamma_S, \Gamma_e \vdash \sim p : \texttt{prop}}[\textsc{Not}]$$

$$\frac{\Gamma_S, \Gamma_e \vdash p_1 : \texttt{prop} \qquad \Gamma_S, \Gamma_e \vdash p_2 : \texttt{prop}}{\Gamma_S, \Gamma_e \vdash p_1 \otimes p_2 : \texttt{prop}}[\textsc{Or/And/Imply}] \quad (\text{for } \otimes = \vee, \wedge, \Rightarrow)$$

$$\frac{\Gamma_S, \Gamma_e \oplus (x : Gen(\tau, \Gamma_e)) \vdash p : \texttt{prop}}{\Gamma_S, \Gamma_e \vdash \textbf{all } x \textbf{ : } \tau, p : \texttt{prop}}[\textsc{All}]$$

$$\frac{\Gamma_S, \Gamma_e \oplus (x : Gen(\tau, \Gamma_e)) \vdash p : \texttt{prop}}{\Gamma_S, \Gamma_e \vdash \textbf{ex } x \textbf{ : } \tau, p : \texttt{prop}}[\textsc{Ex}]$$

Definition 4.16. *Typed Fields*

Let S be a species and ϕ a field of S. The *typed field* Φ is obtained by annotating ϕ with its inferred type. Judgments are of the form $\Gamma_S, \Gamma_e \vdash \phi : \Phi$. ◇

The rules for typechecking fields are presented below. As explained in Section 2.4, polymorphic methods are forbidden. However, typechecking rules do not enforce this restriction which is verified afterwards.

$$\vdash \textbf{rep} : \textbf{rep : } \texttt{Self} \qquad \vdash \textbf{rep} = \tau : \textbf{rep : } \tau$$

$$\vdash \textbf{signature } m \textbf{ : } \tau : \textbf{signature } m \textbf{ : } \tau \qquad \frac{\Gamma_S, \Gamma_e \vdash e : \tau}{\Gamma_S, \Gamma_e \vdash \ \textbf{let } m \textbf{ = } e : \textbf{let } m \textbf{ : } \tau \textbf{ = } e}$$

$$\frac{\Gamma_S, \Gamma_e \oplus \forall i \in [1 \ldots n], (m_i : Gen(\tau_i, \Gamma_e)) \vdash \ \forall i \in [1 \ldots n], e_i : \tau_i}{\Gamma_S, \Gamma_e \vdash \textbf{let rec } m_1 \textbf{ = } e_1 \ldots \textbf{and } m_n \textbf{ = } e_n : \textbf{let rec } m_1 \textbf{ : } \tau_1 \textbf{ = } e_1 \ldots \textbf{and } m_n \textbf{ : } \tau_n \textbf{ = } e_n}$$

The types of computational definitions, thanks to the Curry-Howard isomorphism, are directly embedded into the logical type theory. The type of a property (or a theorem) is definitively not a "ML-like" type but a logical formula: the statement itself of the property (or of the theorem).

$$\frac{\Gamma_S, \Gamma_e \oplus (\texttt{rep} : \texttt{Self}) \vdash p : \texttt{prop}}{\Gamma_S, \Gamma_e \vdash \textbf{property } m \textbf{ : } p : \textbf{property } m \textbf{ : } p}$$

As stated in Section 3, def-dependencies of properties and theorems on `Self` are forbidden by clearing the representation in the typechecking environment ($\Gamma_e \oplus (\texttt{rep} : \texttt{Self})$). The proof of a **`theorem`** is typechecked by checking all the expressions located in the proof tree although no type is issued. This ensures that all its sub-goals (which are expressions) are well-typed. No "logical typing", i.e. verification of the correctness of the proof is done at this stage.

$$\frac{\Gamma_S, \Gamma_e \oplus (\texttt{rep} : \texttt{Self}) \vdash p : \texttt{prop} \qquad \mathit{proof} \text{ is well-typed}}{\Gamma_S, \Gamma_e \vdash \textbf{theorem}\ m \ \textbf{:}\ p\ \textbf{proof =}\ \mathit{proof}\ :\ \textbf{theorem}\ m\ \textbf{:}\ p\ \textbf{proof =}\ \mathit{proof}}$$

The last rule typechecks the list of fields of a species. Each field is typechecked in the environment extended with the bindings issued by the previous typechecked fields.

$$\frac{\begin{array}{c}\Gamma_S, \Gamma_e \vdash \phi : \Phi \\ \Phi = \overrightarrow{m_i : \tau_i} \qquad \Gamma'_e = \Gamma_e \oplus (m_i : \mathit{Gen}(\tau_i, \Gamma_e) \mid m_i : \tau_i \in \Phi) \qquad \Gamma_S, \Gamma'_e \vdash \overrightarrow{\psi} : \overrightarrow{\Psi}\end{array}}{\Gamma_S, \Gamma_e \vdash \phi; \overrightarrow{\psi} : \Phi; \overrightarrow{\Psi}}$$

Definition 4.17. *Body of a Method*
Let S be a species in flat form and $m \in \mathcal{N}(S)$ a method name. The body of m, written $\mathcal{B}_S(m)$, is defined as follows:

$$\begin{array}{lcl}
\mathcal{B}_S(m) & = & \perp \text{ if } m \notin \mathcal{D}(S) \\
\mathcal{B}_S(\textbf{rep} = \tau) & = & \tau \\
\mathcal{B}_S(\textbf{let}\ m = e) & = & e \\
\mathcal{B}_S(\textbf{let rec}\ m_1 : \tau_1 = e_1\ \textbf{and} \ldots m : \tau = e\ \textbf{and} \ldots m_n : \tau_n = e_n) & = & e_i \\
\mathcal{B}_S(\textbf{theorem}\ m : p\ \textbf{proof} = \mathit{proof}) & = & \mathit{proof}
\end{array} \quad \diamond$$

Definition 4.18. *Logical Type of a Method*
Let S be a species in flat form, with fields well-typed according to rules of Section 4.3.1 and not polymorphic. We define the type $\mathcal{T}_S(m)$ of a method $m \in \mathcal{N}(S)$ by embedding computational types into logical ones thanks to the Curry-Howard isomorphism. Hence, logical types are logical statements (i.e. abstract syntax of p's).

$$\begin{array}{lcl}
\mathcal{T}_S(\textbf{let}\ m : \tau = e) = \mathcal{T}_S(\textbf{signature}\ m\textbf{:}\ \tau) & = & \tau \\
\mathcal{T}_S(\textbf{let rec}\ m_1 : \tau_1 = e_1\ \textbf{and} \ldots m : \tau = e\ \textbf{and} \ldots m_n : \tau_n = e_n) & = & \tau \\
\mathcal{T}_S(\textbf{rep} : \tau) & = & \tau \\
\mathcal{T}_S(\textbf{property}\ m\ \textbf{:}\ p) = \mathcal{T}_S(\textbf{theorem}\ m\ \textbf{:}\ p\ \textbf{proof =} \ldots) & = & p
\end{array} \quad \diamond$$

Note that if the representation is not defined, the typing rules of 4.3.1 naturally give it the type `Self`. Note also that the type of a **`proof of`** field is not addressed. Such fields are assumed to have been merged with a property in order to become a theorem as later described in the definition 4.26.

4.4. *Notion of Dependencies: First Stage*

The rules of 4.3.1 described the typing rules for species fields. In order to obtain the type of a species, the normal form of the species must be computed.

To introduce well-formed species and their normalization, a first notion of dependencies is needed : dependencies of methods on `Self` and on methods of the species, decl-dependencies of expressions, def-dependencies of a proof and dependencies obtained by transitive closure. This notion of dependencies will be extended in the Section 6.

Definition 4.19. *Dependencies of Methods on* `Self`

- A method m *decl-depends on the representation* if its type or the type of a subexpression of its body is `Self`.
- A method m *def-depends on the representation* if typechecking m uses the rules [MGSELFL] or [MGSELFR] in the unification process.

◇

A decl-dependency means that some parts of the method require the existence of a representation, whatever is its effective definition. A def-dependency means that the particular definition of **`rep`** is required to typecheck the method (changing the effective type of **`rep`** would cause typechecking to fail).

Technically, the detection of dependencies on the representation is done at two points of the analysis: once during typechecking and once during typed fields fusion (c.f. definition 4.26).

Definition 4.20. *Syntactic Decl-Dependencies of an Expression*

Let e be an expression, the set $\wr e \int$ is defined by:

$$\begin{array}{lcll}
\wr x \int & = & \varnothing & \text{(If } x \text{ is a variable)} \\
\wr C!m \int & = & \varnothing & \text{(If } C \neq \texttt{Self}\text{)} \\
\wr \texttt{Self}!m \int & = & \{m\} & \\
\wr \textbf{fun}\ x_1, \ldots, x_n \rightarrow\ e \int & = & \wr e \int & \\
\wr e\ (e_1, \ldots, e_n) \int & = & \wr e \int \cup \wr e_1 \int \cup \ldots \cup \wr e_n \int & \\
\wr \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2 \int & = & \wr e_1 \int \cup \wr e_2 \int & \\
\wr \textbf{let rec}\ x_1 = e_1\ \textbf{and} \ldots\ \textbf{and}\ x_n = e_n\ \textbf{in}\ e_{n+1} \int & = & \bigcup_{i=1\ldots n+1} \wr e_i \int & \\
 & & & \\
\wr p_1 \vee p_2 \int\ =\ \wr p_1 \wedge p_2 \int\ =\ \wr p_1 \Rightarrow p_2 \int & = & \wr p_1 \int \cup \wr p_2 \int & \\
\wr \sim\ p \int\ =\ \wr \textbf{all}\ x : \tau, p \int\ =\ \wr \textbf{ex}\ x : \tau, p \int & = & \wr p \int &
\end{array}$$

◇

$\wr e \int$ collects all the decl-dependencies of e on the methods of the current species by a syntactical walk along the Abstract Syntax Tree of e. As stated in Section 3, this is collecting the names of methods of the species called in e.

Definition 4.21. *Syntactic Def-Dependencies of a Proof*

Let p be a proof. The set of syntactic def-dependencies of p is defined by:
$\wr\wr\ p \iint = \{\ m \mid$ **by definition of** m is a step of $p\ \}$. ◇

Definition 4.22. *Names in a Same Field*

Let S be a species in flat form, L the list of its fields and m and n two methods of S. $m \infty n$ if it exists $\phi \in L$ such that $m \in \mathcal{N}(\phi)$ and $n \in \mathcal{N}(\phi)$. ◇

This –reflexive – relation is used to compute the effective dependencies on other methods of the species and the well-formedness of the species.

Definition 4.23. *Dependencies of a Method in a Species*
Let S be a species in flat form and $m \in \mathcal{N}(S)$. We define the decl-dependencies ($\wr m \int_S$) and the def-dependencies ($\wr\wr\, m \iint_S$) of a method m on the methods of S by cases on the form of m.

- If m is a **`signature`** or is **`rep`**, $\wr m \int_S = \wr\wr\, m \iint_S = \emptyset$.
- If m is a **`let`**, $\wr m \int_S = \wr \mathcal{B}_S(m) \int$ and $\wr\wr\, m \iint_S = \emptyset$.
- If m is a **`let rec`**, $\wr m \int_S = (\bigcup_n \wr \mathcal{B}_S(n) \int \mid n \infty\ m) \setminus \{ n \mid n \infty\ m \}$ and $\wr\wr\, m \iint_S = \emptyset$.
- If m is a **`property`**, $\wr m \int_S = \wr \mathcal{T}_S(m) \int$ and $\wr\wr\, m \iint_S = \emptyset$.
- If m is a **`theorem`**, $\wr m \int_S = \wr \mathcal{B}_S(m) \int \cup \wr \mathcal{T}_S(m) \int$ and $\wr\wr\, m \iint_S = \wr\wr\, \mathcal{B}_S(m) \iint$.

◇

If a field is a **`let rec`** `m1` **`=...`**, all the m_i have the same set of decl-dependencies, obtained by removing the names m_i from the union of the decl-dependencies of the m_i's bodies. Note that these rules do not consider dependencies on the representation which were already addressed in the definition 4.19.

Definition 4.24. *Transitive Dependencies*
Let S be a species in flat form. The method m_1 of S has a transitive decl-dependency on the method m_2 of S, denoted by $m_1 <_S^{decl} m_2$ (resp. a transitive def-dependency denoted by $m_1 <_S^{def} m_2$) if:

$$m_1 <_S^{decl} m_2 \Leftrightarrow \exists\ a, b \text{ such as } a \infty\ m_1 \wedge b \infty\ m_2 \wedge \exists \{ p_i \}_{i=1\ldots n} \text{ such as } \left\{ \begin{array}{l} a = p_1 \\ p_n = b \\ \forall j < n,\ p_j \in \wr p_{j+1} \int_S \cup \wr\wr\, p_{j+1} \iint_S \end{array} \right.$$

$$m_1 <_S^{def} m_2 \Leftrightarrow \exists \{ p_i \}_{i=1\ldots n} \text{ such as } \left\{ \begin{array}{l} p_1 \infty\ m_1 \\ p_n \infty\ m_2 \\ \forall j < n,\ p_j \in \wr\wr\, p_{j+1} \iint_S \end{array} \right.$$

◇

In other words, a and m_1 are in a same field (i.e. **`let-rec`** bound), so are b and m_2, and there exists a dependency path between m_1 and m_2, hence between a and b.

To check if $m_1 <_S^{decl} m_2$ (resp. $m_1 <_S^{def} m_2$), one builds the decl-dependencies (resp. def-dependencies) graph of all the methods of S, and searches a path from m_1 to m_2. These graphs are also used to properly order λ-liftings when building method generators and collection generators as described further.

4.5. Normalization

Definition 4.25. *Well-Formed Species*
A species S is said *well-formed* if $\forall m \in \mathcal{N}(S), \neg(m <_S^{decl} m)$. ◇

In a well-formed species no field has a circular dependency on itself by transitivity. The following examples show several errors detected by the analysis of well-formedness.

First, ∞ can reveal a *latent* dependency between a **`let rec`** bound name $m1$ having no direct dependency on a name $m3$. Indeed, $m1$ depends on $m2$ which depends on $m3$.

```
species A =
  let rec m1 = ... m2
  and m2 = ... m1 ;
end ;;
```

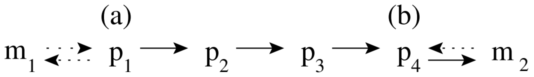
```
species B =
  inherit A ;
  let m3 = ... m1 ;
  let m2 = ... m3 ;
end ;;
```

Once in flat form, the species B would have the form:

```
species B =
  let m3 = ... m1 ;
  let rec m1 = ... m2
  and m2 = ... m3 ;
end ;;
```

where $m3$ depends on $m1$ and $m1$ depends on $m3$ via $m2$, mutually defined with $m1$. Hence, $m3$ depends on itself although it was not defined as recursive.

In the definition of $<_S^{decl}$, the conditions $a \infty\ m_1$ and $b \infty\ m_2$ are important since decl-dependencies of names introduced by a mutual **`let rec`** method exclude these names. In the figure below, omitting the condition $m_1 \infty\ a$ would prevent detecting the transitive dependency of m_1 on p_4 (hence on m_2).



Since ∞ is reflexive, we also ensure that two names do not have mutual dependencies (set $m_1 = p_1$ and $p_2 = m_2$ in the definition of $<_S^{decl}$) as shown the following example:

```
species A =
  signature m1 : ...
  let m2 = ... m1 ;
end ;;
```

```
species B =
  inherit A ;
  signature m2 : ... ;
  let m1 = ... m2 ;
  end ;;
```

Once in flat form, the species B would have this form

```
species B =
  let m1 = ... m2 ;
  let m2 = ... m1 ;
end ;;
```

which is not well-formed. Indeed, taking $p_1 = m1$, we have $m1 \infty\ m1$, taking $p_2 = m2$, we have $m2 \infty\ m2$. Then we have $m1 \in \llbracket m2 \rrbracket_B$ and $m2 \in \llbracket m1 \rrbracket_B$ hence $m1 <_B^{decl} m1$.

Definition 4.26. *Typed Fields Fusion*

Let Φ_1 and Φ_2 be two typed fields such as $\mathcal{N}(\Phi_1) \cap \mathcal{N}(\Phi_2) \neq \emptyset$. The fusion of two typed fields, Φ_1 ⧁ Φ_2, is defined by case. Note that ⧁ is not symmetric.

$$\begin{array}{lcl}
\textbf{signature}\ m : \tau_1 & ⧁ & \textbf{signature}\ m : \tau_2 & = & \textbf{signature}\ m : Mgu(\tau_1, \tau_2) \\
\textbf{signature}\ m : \tau_1 & ⧁ & \textbf{let}\ m : \tau_2 = e_2 & = & \textbf{let}\ m : Mgu(\tau_1, \tau_2) = e_2 \\
\textbf{signature}\ m : \tau_1 & ⧁ & \begin{array}{l}\textbf{let rec}\ m : \tau_2 = e_2 \\ \textbf{and}\ n_i : \rho_i = en_i\end{array} & = & \begin{array}{l}\textbf{let rec}\ m : Mgu(\tau_1, \tau_2) = e_2 \\ \textbf{and}\ n_i : \rho_i = en_i\end{array}
\end{array}$$

$$
\begin{array}{lcl}
\textbf{let } m : \tau_1 = e_1 & \ogreaterthan\ \textbf{signature } m : \tau_2 & = \textbf{let } m : Mgu(\tau_1, \tau_2) = e_1 \\
\textbf{let } m : \tau_1 = e_1 & \ogreaterthan\ \textbf{let } m : \tau_2 = e_2 & = \textbf{let } m : Mgu(\tau_1, \tau_2) = e_2 \\
\textbf{let } m : \tau_1 = e_1 & \ogreaterthan\ \begin{array}{l} \textbf{let rec } m : \tau_2 = e_2 \\ \textbf{and } n_i : \rho_i = en_i \end{array} & = \begin{array}{l} \textbf{let rec } m : Mgu(\tau_1, \tau_2) = e_2 \\ \textbf{and } n_i : \rho_i = en_i \end{array} \\
\begin{array}{l} \textbf{let rec } m : \tau_1 = e_1 \\ \textbf{and } n_i : \rho_i = en_i \end{array} & \ogreaterthan\ \textbf{signature } m : \tau_2 & = \begin{array}{l} \textbf{let rec } m : Mgu(\tau_1, \tau_2) = e_1 \\ \textbf{and } n_i : \rho_i = en_i \end{array} \\
\begin{array}{l} \textbf{let rec } m : \tau_1 = e_1 \\ \textbf{and } n_i : \rho_i = en_i \end{array} & \ogreaterthan\ \textbf{let } m : \tau_2 = e_2 & = \begin{array}{l} \textbf{let rec } m : Mgu(\tau_1, \tau_2) = e_2 \\ \textbf{and } n_i : \rho_i = en_i \end{array} \\
\begin{array}{l} \textbf{let rec } m_i : \tau_i = em_i \\ \textbf{and } n_i : \rho_i = en_i \end{array} & \ogreaterthan\ \begin{array}{l} \textbf{let rec } m_i : \sigma_i = em'_i \\ \textbf{and } o_i : \pi_i = eo_i \end{array} & = \begin{array}{l} \textbf{let rec } m_i : Mgu(\tau_i, \sigma_i) = em'_i \\ \textbf{and } n_i : \rho_i = en_i \\ \textbf{and } o_i : \pi_i = eo_i \end{array} \\
\textbf{property } m : p_1 & \ogreaterthan\ \textbf{property } m : p_2 & = \begin{array}{l} \textbf{property } m : p_2 \\ \text{if } p_1 \overset{\alpha}{=} p_2 \end{array} \\
\textbf{property } m : p_1 & \ogreaterthan\ \textbf{theorem } m : p_2 = pr & = \begin{array}{l} \textbf{theorem } m : p_2 = pr \\ \text{if } p_1 \overset{\alpha}{=} p_2 \end{array} \\
\textbf{theorem } m : p_1 = pr & \ogreaterthan\ \textbf{property } m : p_1 & = \begin{array}{l} \textbf{theorem } m : p_1 = pr \\ \text{if } p_1 \overset{\alpha}{=} p_2 \end{array} \\
\textbf{theorem } m : p_1 = pr_1 & \ogreaterthan\ \textbf{theorem } m : p_2 = pr_2 & = \begin{array}{l} \textbf{theorem } m : p_2 = pr_2 \\ \text{if } p_1 \overset{\alpha}{=} p_2 \end{array} \\
\textbf{property } m : p & \ogreaterthan\ \textbf{proof of } m = pr & = \textbf{theorem } m : p = pr \\
\textbf{theorem } m : p = pr_1 & \ogreaterthan\ \textbf{proof of } m = pr_2 & = \textbf{theorem } m : p = pr_2
\end{array}
$$

◇

$\overset{\alpha}{=}$ is the usual syntactical identity modulo α-conversion between expressions.

Because the fusion relies on $\overset{\alpha}{=}$ and Mgu which can fail, this function is partial. Moreover, because fields are not polymorphic, a successful application of Mgu means that the type of a field has not been modified by redefinitions.

As introduced in 4.19, dependencies on the representation are checked again during the fusion process which uses Mgu, possibly unifying `Self` with other types, adding decl and/or def-dependencies on the representation.

As inheritance allows re-definitions, a proof can be invalidated because the definitions it relies on have been changed. In such a case, the **`theorem`** owning the proof has to be reverted into a **`property`**. Detection of changes in a method, i.e. of *conflicts* between its different definitions is the starting point of proof invalidation (c.f. definition 4.29).

Definition 4.27. *Conflict Between Fields*

Let Φ_1 and Φ_2 be two typed fields sharing a same name m. We say that Φ_1 *conflicts* with Φ_2 in one of the following cases:

$$
\begin{array}{lll}
\Phi_1 \text{ is } \textbf{let } m : \tau = e_1 & \text{and} & \Phi_2 \text{ is } \textbf{let } m : \tau = e_2 \\
\Phi_1 \text{ is } \textbf{theorem } m : p \textbf{ proof} = \mathit{proof}_1 & \text{and} & \Phi_2 \text{ is } \textbf{theorem } m : p \textbf{ proof} = \mathit{proof}_2 \\
\Phi_1 \text{ is } \textbf{let } m : \tau = e_1 & \text{and} & \Phi_2 \text{ is } \textbf{let rec } m : \tau = e_2 \textbf{ and } \ldots \\
\Phi_1 \text{ is } \textbf{let rec } m : \tau = e_1 \textbf{ and } \ldots & \text{and} & \Phi_2 \text{ is } \textbf{let } m : \tau = e_2
\end{array}
$$

◇

No conflict exist between **`theorem`** and **`property`** (resp. **`let`** vs **`signature`**) since **`property`** and **`signature`** only show types, through which no def-dependency is possible.

Redefinitions can invalidate proofs, which must be erased if they have def-dependencies on the redefined method. Def-dependencies on the representation can arise in any kind of methods but since methods must keep their type along redefinitions, the **`representation`** cannot be changed, hence cannot require erasure.

Definition 4.28. *Erasure of Theorems*

$$\begin{array}{lcl} \mathcal{E}(\textbf{theorem}\ m : p\ \textbf{proof = }proof) & = & \textbf{property}\ m : p \\ \mathcal{E}(m) & = & m \text{ otherwise} \end{array}$$

◇

Definition 4.29. *Erasure in a Context*

Let N be a list of methods names, m be a typed field and l a list of typed fields. The erasure in the context of N is defined by:

$$\begin{array}{lcll} \mathcal{E}_N(\emptyset) & = & \emptyset & \\ \mathcal{E}_N(m\ ;\ l) & = & \mathcal{E}(m)\ ;\ \mathcal{E}_{N \cup \mathcal{N}(m)}(l) & \text{if } \llbracket\, m \,\rrbracket \cap N \neq \emptyset \\ \mathcal{E}_N(m\ ;\ l) & = & m\ ; \mathcal{E}_N(l) & \text{if } \llbracket\, m \,\rrbracket \cap N = \emptyset \end{array}$$

◇

In the second case of this definition, if m def-depends on names present in N, then m is erased first, and then all the proofs having a def-dependency on m in the remaining context are erased.

We now present the normalization algorithm *NF* which is used after the typechecking of all the fields. Let us consider S a species defined by:

species $S(\overrightarrow{C \blacksquare t})$ **= inherit** $S_1, \ldots, S_n = \phi_1 \ldots \phi_m$ **end**

The call to *NF* to normalize S is done under the hypotheses that the S_i are in normal form, the types of parameters methods are known in the environment and the fields of S are typed. The input, $\mathbb{W}_1$, of *NF* is a list containing all the typed fields $\overrightarrow{\Phi_{S_i}}$ of the inherited species S_i in normal form, in the same order as specified in the **`inherit`** clause, and ended by the fields directly defined in the species S. This ordering of $\mathbb{W}_1$ is an invariant of the algorithm.

$$\mathbb{W}_1 = \overrightarrow{\Phi_{S_1}} @ \ldots @ \overrightarrow{\Phi_{S_n}} @ [\Phi_1 \ldots \Phi_m]$$

$\mathbb{W}_1$ can contain several occurrences of a same name in different fields because of multiple inheritance and methods redefinition. The algorithm returns a list $\mathbb{W}_2$ of typed fields such that each name of $\mathbb{W}_1$ appears only once in $\mathbb{W}_2$ (i.e. resolution of inheritance). $\mathbb{W}_2$ represents the list of fields already processed by the algorithm and (possibly temporary) accepted as members of the normalized species. Hence, at any moment, they are "older" according to the inheritance order than the currently processed Φ.

To do that, the algorithm recursively extracts the first field, Φ, of $\mathbb{W}_1$ and appends it to $\mathbb{W}_2$, if no field of $\mathbb{W}_2$ shares a name with Φ. Otherwise, Φ is fusioned with the first field, Ψ_{i_0}, of $\mathbb{W}_2$ which shares a name with it. The result of fusion is re-inserted as the head of $\mathbb{W}_1$ to look for other possible conflicts with the fields of $\mathbb{W}_2$ following Ψ_{i_0}. The field Ψ_{i_0} is removed from $\mathbb{W}_2$ and, in case of conflict between Ψ_{i_0} and Φ, erasing is propagated in the remainder of $\mathbb{W}_2$. Doing that, all the theorems recorded in $\mathbb{W}_2$ which def-depend on a name of Φ are reverted into properties.

Definition 4.30. *Normalization Algorithm*

```
$\mathcal{NF}(\mathbb{W}_1) =$
  $\mathbb{W}_2 \leftarrow \varnothing$
  While $\mathbb{W}_1 \neq \varnothing$ do
    $\Phi \leftarrow hd(\mathbb{W}_1)$   and   $\mathbb{X} \leftarrow tl(\mathbb{W}_1)$
    If $\mathcal{N}(\Phi) \cap \mathcal{N}(\mathbb{W}_2) = \varnothing$ then
      $\mathbb{W}_1 \leftarrow \mathbb{X}$   and   $\mathbb{W}_2 \leftarrow \mathbb{W}_2 @ [\Phi]$
    Else
      Assume that $\mathbb{W}_2$ is of the form $[\Psi_1; \ldots; \Psi_m]$
      Let $i_0$ be the smallest index such as for $i \in [1 : m], \mathcal{N}(\Phi) \cap \mathcal{N}(\Psi_i) \neq \varnothing$
      $\mathbb{W}_1 \leftarrow (\Psi_{i_0} \circledgtr \Phi) :: \mathbb{X}$
      If $\Phi$ conflicts with $\Psi_{i_0}$ then
        $\mathbb{W}_2 \leftarrow [\Psi_1; \ldots; \Psi_{i_0-1}] @ \mathcal{E}_{\mathcal{N}(\Psi_{i_0})}(\Psi_{i_0+1}, \ldots, \Psi_m)$
      Else
        $\mathbb{W}_2 \leftarrow [\Psi_1; \ldots; \Psi_{i_0-1}] @ [\Psi_{i_0+1}; \ldots, \Psi_m]$
```

◇

4.6. Typechecking Species and Collections

Definition 4.31. *Sub-species Relation*

Let S_1 and S_2 be two species in normal form. S_1 is a sub-species of S_2, written $S_1 \preccurlyeq S_2$, if:

$$\mathcal{N}(S_2) \subset \mathcal{N}(S_1) \wedge \forall x \in \mathcal{N}(S_2),\ \mathcal{T}_{S_1}(x) = \mathcal{T}_{S_2}(x)$$

◇

By construction, if S_1 inherits from S_2, then $S_1 \preccurlyeq S_2$. This intuitively means that S_1 has at least all the methods of S_2, with the same types. This relation is used to check that the instantiation of a collection parameter of interface P by a collection of interface C is sound, that is C has at least the methods required by P.

Let ts be a species type and C a collection parameter name. We define $\mathcal{A}(ts, C)$ the abstraction operation turning an atomic species type (i.e. without parameters) ts into a collection type by replacing the occurrences of `Self` by C in the types of the methods of ts.

Definition 4.32. *Abstraction of a Species Type w.r.t. a Collection Parameter Name*

$$\mathcal{A}(\{\overrightarrow{\Phi}\}, C) = \langle m_i : \tau_i[\texttt{Self} \leftarrow\!\dashv C] \rangle \quad \forall m_i : \tau_i \in \{\overrightarrow{\Phi}\}$$

◇

The abstraction operation allows hiding the structure of the representation when creating a collection from a species.

Follow the rules for the typechecking of a species expression se, which issues a species type. We use the "overloaded" operator $\oplus$ standing for the extension of an environment Γ_e with the bindings coming from the names bound in a typed field Φ.

Definition 4.33. *Rules of Typechecking Species Expressions*

$$\frac{\begin{array}{c}\Gamma_S, \Gamma_e \vdash se_1 : \{\overrightarrow{\Phi}_1\} \quad \ldots \quad \Gamma_S, \Gamma_e \vdash se_i : \{\overrightarrow{\Phi}_i\} \qquad \Gamma'_e = \overrightarrow{\Phi}_1 \oplus \ldots \oplus \overrightarrow{\Phi}_i \\ \Gamma_S, \Gamma'_e \vdash \overrightarrow{\phi} : \overrightarrow{\Phi} \qquad \overrightarrow{\Phi''} = \mathit{NF}(\overrightarrow{\Phi}_1 @ \ldots @ \overrightarrow{\Phi}_i, \overrightarrow{\Phi}) \\ \forall m : \tau \in \Phi'', \tau \text{ is not polymorphic} \qquad \overrightarrow{\Phi''} \text{ is well-formed}\end{array}}{\Gamma_S, \Gamma_e \vdash \textbf{species } S \textbf{ = inherit } \overrightarrow{se_i} ; \overrightarrow{\phi} : \{\overrightarrow{\Phi''}\}}\,[\textsc{No-Prm}]$$

$$\frac{\begin{array}{c}\Gamma_S, \Gamma_e \vdash se : \{\overrightarrow{\Phi}\} \\ \Gamma_S \oplus (C : \mathcal{A}(\{\overrightarrow{\Phi}\}, C)) \vdash \textbf{species } S(prms) \textbf{ = inherit } \overrightarrow{se_i} ; \overrightarrow{\phi} : ts\end{array}}{\Gamma_S, \Gamma_e \vdash \textbf{species } S(C \textbf{ is } se, prms) \textbf{ = inherit } \overrightarrow{se_i} ; \overrightarrow{\phi} : (C \textbf{ is } \{\overrightarrow{\Phi}\})\ ts}\,[\textsc{Coll-Prm}]$$

$$\frac{\tau = C \qquad \Gamma_S, \Gamma_e \oplus (x : \tau) \vdash \textbf{species } S(prms) \textbf{ = inherit } \overrightarrow{S_i} ; \overrightarrow{\phi} : (x \textbf{ in } \tau)\ ts}{\Gamma_S, \Gamma_e \vdash \textbf{species } S(x \textbf{ in } C, prms) \textbf{ = inherit } \overrightarrow{S_i} ; \overrightarrow{\phi} : (x \textbf{ in } \tau)\ ts}\,[\textsc{Ent-Prm}]$$

$$\frac{\Gamma_S, \Gamma_e \vdash se : (x \textbf{ in } \tau)\ ts \qquad \Gamma_S, \Gamma_e \vdash e : \tau}{\Gamma_S, \Gamma_e \vdash se(e) : ts[x \mapsfrom e]}\,[\textsc{Ent-Inst}]$$

$$\frac{\Gamma_S, \Gamma_e \vdash se : (C_1 \textbf{ is } \{\overrightarrow{\Phi}\})\ ts \qquad \Gamma_S(C_2) = tc_2 \qquad tc_2 \preccurlyeq \mathcal{A}(\{\overrightarrow{\Phi}\}, C_2)}{\Gamma_S, \Gamma_e \vdash se(C_2) : ts[C_1 \mapsfrom C_2]}\,[\textsc{Coll-Inst}]$$

$$\frac{\Gamma_S(S) = ts}{\Gamma_S, \Gamma_e \vdash S : ts}\,[\textsc{Spec-Ident}] \qquad \diamond$$

The rule [COLL-PRM] requires that se has an atomic type: there is no higher-order species. This verification is performed during the typechecking. This restriction also appears in [COLL-INST] since the effective argument C_2 of se has to be a collection name (enforced by the syntax of the language) and not an arbitrary species expression.

In the rule [ENT-INST], τ can only be a name of a collection denoting the representation of this collection. This is syntactically enforced by the grammar of species parameters (c.f. prm in the definition 4.5).

Proposition. The important property of a typechecked species is that its type does not show anymore inheritance: the species is in normal form, with all the methods brought by inheritance and kept according to the early-binding mechanism (the most recent method is always kept in case of redefinition).

Definition 4.34. *Rule of Typechecking Collections*

$$\frac{\Gamma_S, \Gamma_e \vdash se : \{\overrightarrow{\Phi}\} \qquad se \text{ represents a complete species}}{\Gamma_S, \Gamma_e \vdash \textbf{collection } C \textbf{ = implements } se \textbf{ end} : \mathcal{A}(\{\overrightarrow{\Phi}\}, C)} \qquad \diamond$$

5. Code Generation Model

In this section we outline the chosen code generation model and show the advantages of the dependencies analysis for this model. In Section 8 other possibilities, not adopted, for the code generation are discussed.

Our first choice is to encapsulate the species and collection translations into OCaml and Coq modules to take benefit from a name-space, modularity and later, abstraction by module signature. We use a flat module model where species methods lead to module fields.

5.1. Generating Code for only Collections

As explained before, FoCaLiZe design makes a neat distinction between the code writer (of species) and the code user (of collections). This entails that collections are the only mean given to developers to ultimately create computational and logical codes. Thus we may decide to emit target code only when a collection is created. This is indeed possible as FoCaLiZe provides only static object features. But this choice has a major drawback explained below.

Let S be a complete species defined by `species S (C is S1)`. Let $C1$ created by `C1 implement S(D1)` and $C2$ created by `C2 implement S(D2)`. Let m be a method of S. $C1$ and $C2$, compiled independently, have their own copy ($m1$ and $m2$) of m. $m1$ and $m2$ share the body of m up to their calls to methods of $D1$ and $D2$. Moreover let SE be a complete species inheriting from S and let $C3$ a collection created from SE. $C3$ contains a copy $m3$ of $SE!m$. If m is not redefined in SE then $m1$, $m2$ and $m3$ share the body of $S!m$, up to the calls of methods of S redefined in SE and the calls of parameters' methods. This sharing cannot be capitalized, due to the independence of the compilation of collections.

Thus compiling only collections leads to the duplication of the methods bodies in each created collection. It increases the size of the generated code. It prevents separate compilation since the code is only generated at the "end of the development", requiring the re-examination of all the species underlying each collection. At last, it duplicates code review.

5.2. Incremental Code Generation

Our goal is to maximize code sharing between all the collections issued from species created along a given inheritance chain. In these collections, methods having the same name can differ either because some ones have been redefined or because they call methods which have been redefined or provided by different effective collection parameters. In this last case, the changes are located only at these calls: the core of the algorithm remains the same.

On the example of Section 2.3, whatever are the collections created from `IsIn`, the core of the `filter` method always applies comparisons. The only differences are the used parameters method `lt` and the parameters values `minv` and `maxv`. In the same way, the statements and the proofs of the theorem `lowMin` only differ by the effective used method `filter` of the species and those of its parameters `V`, `minv` and `maxv`.

5.2.1. Method Generator

We decide to generate the code of a method as soon as the method is defined in order to share it all along the inheriting species and the derived collections. Let m be a method whose body contains a call to a method p. Suppose that p is only declared or provided by a collection parameter. Then we λ-lift the name p in the body of m. On the opposite, calls to methods on which m has a def-dependency are directly done. Hence, we use def- and decl-dependencies to determine if a name should be λ-lifted.

If the target language requires explicit types representation (as in Coq), the only declared **rep**s (from the species itself or from its parameters) appearing in the type of the method are also λ-lifted (suffixed by _T in the following figure).

```
let gt (x, y) =
 ~~ lt (x, y) && ~~ eq (x, y)








let filter (x) : Self =
 if V!lt (x, minv) then
   (minv, Too_low)
 else
  if V!gt (x, maxv) then
   (maxv, Too_high)
  else (x, In_range)
```

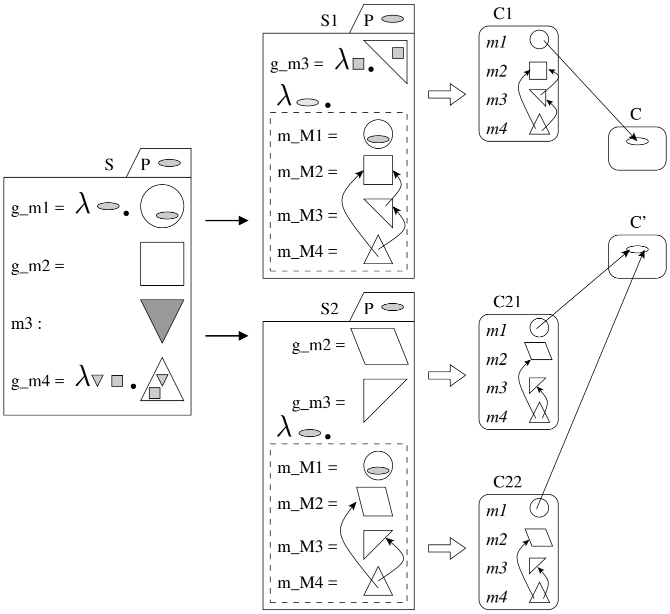
```
Let gt (abst_T : Set)
  (abst_eq: abst_T -> abst_T -> bool)
  (abst_lt: abst_T -> abst_T -> bool)
  (x: abst_T) (y: abst_T) : bool :=
 not (abst_lt (x, y)) &&
 not (abst_eq (x, y))

Let filter (V_T: Set)
  (V_lt: V_T -> V_T ->bool)
  (V_gt: V_T -> V_T -> bool)
  (minv: V_T) (maxv: V_T)
  (abst_T := V_T * statut_t)
  (x: V_T) : abst_T  :=
 if V_lt (x, minv) then (minv, Too_low)
 else
  if V_gt (x, maxv) then (maxv, Too_high)
  else (x, In_range)
```

The function generated from a given method m by λ-lifting its decl-dependencies is called the *method generator* of m. It is emitted as soon as m is (re-)defined in a species S, and is shared by all the children of S in the inheritance tree to avoid code duplication. Thus, introducing the notion of method generators allows maximal sharing of the methods bodies. Then, the value of m in a given species will be obtained by applying its method generator to the effective values of its decl-dependencies.

5.2.2. Collection Generator

If the species is not yet complete, its compilation produces only the set of method generators of the methods defined by this species. If the species is complete, it may be used to create collections and its compilation has a last step, which is the production of its *collection generator*.



In the above figure, the shapes representing the methods vary in size. The initial size represent the body of the method. As long as the code sharing increases, the shapes get smaller still keeping their contour to remind the original method they implement. Thin arrows represent in some sense pointers to the code of the related method. The dark gray method m_3 is only declared. λ-lifted symbols are in pale gray to remind they indeed represent unknown (a kind of declared) methods.

Let m be a method of a complete species S, thus a method of all the collections C_i created upon S. m may have decl-dependencies on a set M of methods of S and a set of decl-dependencies P on collection and entity parameters. Note that the values of the called methods of M by m are already known. Therefore the method generator g_m of m can be applied to these values before any creation of one C_i. Let us call m_M the value of this application. It only remains in m_M the λ-liftings of the names in P. Indeed the values of the different occurrences of m in the C_i differ now only by the values of the called methods from P.

All the m_M's values of methods of S are grouped into a record while their abstractions are lifted outside the record as abstractions on the whole set of collection parameters methods (dashed rectangles in the figure). This is the definition of the collection generator of S. Doing this way, all the C_i share the m_M's bodies.

The value m_M itself is obtained by first searching the method generator of m in the inheritance tree of S to find its most recent version, say in the species $S1$. Suppose that the method m_1 of M was λ-lifted from the body of m because m_1 was an only declared method in $S1$. Now m_1 has received a definition at some place in the inheritance tree (S is complete) so its method generator g_m_1 is known. To obtain the effective value of m_1, it remains to eliminate the λ-liftings of g_m_1 by applying them to the corresponding effective values. This process is well-founded as there is no circular dependencies.

Now let CC be a collection obtained from S. The values of CC are obtained by applying the collection generator of S to the effective values of the collection parameters methods. Their search is as follows. Let p_1 be a method of P. It was λ-lifted from the body of m because provided by a collection parameter C of $S1$. C has been instantiated by a collection D to create the complete species S, through a chain of parameter instantiations along the inheritance tree, which at a node gives the effective value of p_1. Note that lifting parameters at the record level instead doing it at each method level simplifies the generated code structure and increases its readability (a good point for assessment).

As a conclusion, providing material from the species parameters to create a collection is hence done once, at the point where the collection generator is called. The usage of the **method generators** ensures code sharing between all the species of a same inheritance tree. The usage of **collection generators** ensures code sharing between all the collections implementing a given species.

On the example of Section 2.3, the collection generator of the species `IsIn` (where some types are omitted) can be represented by:

```
Let collection_create (V_T : Set) (minv : V_T) (maxv : V_T)
    V_lt V_gt V_ltNotGt :=
  let t_filter := gen_filter V_T V_lt V_gt minv maxv in
  let t_getStatus := gen_getStatus V_T in
  let t_getValue := gen_getValue V_T in
  let t_lowMin :=
    gen_lowMin V_T V_lt V_gt V_ltNotGt minv maxv t_filter t_getStatus in
  { filter =  t_filter ;
    getStatus = t_getStatus ;
    getValue = t_getValue ;
    lowMin = t_lowMin }
```

where `gen_`*xxx* stands for the method generator of the method *xxx* and `t_`*xxx* for the temporary application of its method generator to its required arguments (either already `t_`*yyy* or λ-lifted collection / entity parameters material).

To now summarize, a species is compiled toward an OCaml and a Coq module made of the ordered sequence of method generators. Only declared or inherited methods do not lead to generated code. When a species has all its methods defined (complete species), this module also contains a record type definition and a collection generator returning, when applied, a value of this type.

To later create a collection, the collection generator is applied to the methods, types and values passed as effective arguments. As previously stated, a collection being opaque, it will be represented by a module with a constraint signature forcing to only export methods and an abstract type representing the collection's **`rep`**.

6. Dependency Calculus: Second Stage

As explained in this section, the notion of dependencies introduced in Section 4.4 is not sufficient to emit code that Coq can typecheck. We must now address the production of terms correctly typed at the logical level.

6.1. Dependencies on Methods of the Species

Let m be a well-typed method. Creating its method generator requires to know the types of the λ-lifted names. If m is a function of body e, only are needed the types of the methods appearing in the decl-dependencies $\wr e \int$ of e. Since these types are "ML-like" types, they cannot bring any other dependencies. Theorems bodies (i.e. proofs) can introduce def-dependencies, whose definitions must be well-typed in the logical target. Proofs may also introduce decl-dependencies on logical methods, whose types are logical statements: methods appearing in such types must also be well-typed in the logical target language. This is illustrated by the following example, which considers the theorem `ltNotGt` of Section 2.2.

Convention on generated code. : In the method generator of a method m, the methods n of the species on which m depends are always named `abst_`n. They are either λ-lifted (decl-dependency) or bound by the **:=** construct (def-dependency) to their effective generator. `abst_T` corresponds to the dependency on the representation (which is a type).

```
theorem ltNotGt :
  all x y : Self, lt (x, y) -> ~ gt (x, y)
  proof = by definition of gt property int_lt_not_gt ;
```

`ltNotGt` syntactically decl-depends on `gt`, `lt`, the representation and def-depends on `gt`. As stated in 5.2.1, def-dependencies are not λ-lifted and their corresponding definitions are directly used. Since the proof contains a def-dependency on `gt`, it means that the Coq term issued by the external prover contains `eq` (coming from the body of `gt`), assumed to be λ-lifted as `abst_eq`. Now λ-lifting only methods found in the syntactic def- and decl- dependencies (e.g. forgetting transitive dependencies) would lead to a generated code looking like:

```
Theorem ltNotGt (abst_T : Set) (abst_lt := lt)
  (abst_gt := OrdData.gt abst_T abst_eq abst_lt) :
    forall x y : abst_T, Is_true (abst_lt x y) -> ~Is_true (abst_gt x y).
  apply "Large Coq term generated by Zenon";
```

In this generated code, `abst_eq` which is unbound represents `eq`. But, `eq` has a def-dependency on the representation: it should also be propagated to `ltNotGt`.

Thus only considering dependencies coming from the syntax is not sufficient. The same incompleteness of syntactic dependencies arises for collection parameters methods. Hence, omitted until now, a process of "completion" of these dependencies has to be applied before really λ-lifting them. This point is carried out by computing:

1. the *visible universe* of a method;
2. its *minimal logical typing environment*;
3. dependencies on the parameters methods (indeed, these methods are not defined until an effective collection instantiates the parameter: they will be λ-lifted as well – they are decl-dependencies).

The visible universe of a method m is the set of other methods of the species that are needed to ensure that the translation of the method generator of m will be well-typed by the two target languages. The minimal logical typing environment of a method m is obtained by picking the methods of the visible universe and abstracting them or not, i.e. determining if one just needs to keep their type (hence λ-lifting) or also their body (hence "**:=**-binding"). They are formally defined below.

The visible universe of a method m contains all the methods p on which m has a transitive def-dependency and the methods on which the p decl-depend.

Definition 6.1. *Visible Universe of a Method*

The *visible universe* of a method m, $\mid m \mid$ is defined as follows:

$$\frac{m_1 \in \lbag m \rbag_S}{m_1 \in \mid m \mid} \qquad \frac{m_1 <_S^{def} m}{m_1 \in \mid m \mid}$$

$$\frac{m_2 <_S^{def} m \qquad m_1 \in \lbag m_2 \rbag_S}{m_1 \in \mid m \mid} \qquad \frac{m_2 \in \mid m \mid \qquad m_1 \in \lbag \mathcal{T}_S(m_2) \rbag_S}{m_1 \in \mid m \mid} \qquad \diamond$$

From the notion of visible universe, one defines the minimal logical typing environment of a method m of a species S. It contains the other methods of S needed to have the method generator of m well-typed in the target languages.

- Methods not present in the visible universe are not required.
- Methods present in the visible universe on which m doesn't def-depend are required but only their type is needed.
- Methods present in the visible universe on which m def-depends are required with both their type and body.

Definition 6.2. *Minimal Logical Typing Environment of a Method*

Let S be a species in normal form of type: $\overrightarrow{(C \blacksquare t)}\ \{n_i : \tau_i = e_i\}$. Let m be a method name $\in \mathcal{N}(S)$. The minimal typing environment of m is obtained by the $\Cap$ operation defined by:

$$\varnothing \Cap m = \varnothing \qquad \frac{n \notin \mid m \mid \qquad \{n_i : \tau_i = e_i\} \Cap m = \Sigma}{\{n : \tau = e \ ; \ n_i : \tau_i = e_i\} \Cap m = \Sigma}$$

$$\frac{n \in \mid m \mid \qquad n <_S^{def} m \qquad \{n_i : \tau_i = e_i\} \Cap m = \Sigma}{\{n : \tau = e \ ; \ n_i : \tau_i = e_i\} \Cap m = \{n : \tau = e \ ; \ \Sigma\}}$$

$$\frac{n \in \mid m \mid \qquad n \not<_S^{def} m \qquad \{n_i : \tau_i = e_i\} \Cap m = \Sigma}{\{n : \tau = e \ ; \ n_i : \tau_i = e_i\} \Cap m = \{n : \tau \ ; \ \Sigma\}} \qquad \diamond$$

6.2. Dependencies on Parameters Methods

We now compute the dependencies of an expression on parameters methods.

Definition 6.3. *Dependencies of an Expression on a Species Parameter*

Let C be a collection parameter of a species S in normal form. $\lfloor e \rfloor_{S,C}$ the set of direct dependencies of an expression e of S on methods of C is defined by:

$$
\begin{array}{lcl}
\lfloor C \rfloor_{S,C} \quad \text{(If } C \text{ is an entity parameter)} & = & \{C\} \\
\lfloor m \rfloor_{S,C} \quad \text{(If } m \text{ is a variable)} & = & \varnothing \\
\lfloor C'!m \rfloor_{S,C} & = & \varnothing \quad \text{(If } C' \neq C\text{)} \\
\lfloor C!m \rfloor_{S,C} & = & \{m\} \\
\lfloor \textbf{fun}\ x_1, \ldots, x_n \rightarrow\ e \rfloor_{S,C} & = & \lfloor e \rfloor_{S,C} \\
\lfloor e\ (e_1, \ldots, e_n) \rfloor_{S,C} & = & \lfloor e \rfloor_{S,C} \bigcup_{i=1\ldots n} \lfloor e_i \rfloor_{S,C} \\
\lfloor \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \rfloor_{S,C} & = & \lfloor e_1 \rfloor_{S,C} \cup \lfloor e_2 \rfloor_{S,C} \\
\lfloor \textbf{let rec}\ x_1 = e_1 \ldots\ \textbf{and}\ x_n = e_n\ \textbf{in}\ e_{n+1} \rfloor_{S,C} & = & \bigcup_{i=1\ldots n+1} \lfloor e_i \rfloor_{S,C}
\end{array}
$$

$$
\begin{array}{lcl}
\lfloor p_1 \vee p_2 \rfloor_{S,C} \ = \ \lfloor p_1 \wedge p_2 \rfloor_{S,C} \ = \ \lfloor p_1 \Rightarrow p_2 \rfloor_{S,C} & = & \lfloor p_1 \rfloor_{S,C} \cup \lfloor p_2 \rfloor_{S,C} \\
\lfloor \sim p \rfloor_{S,C} \ = \ \lfloor \textbf{all}\ x \textbf{ : } \tau, p \rfloor_{S,C} \ = \ \lfloor \textbf{ex}\ x \textbf{ : } \tau, p \rfloor_{S,C} & = & \lfloor p \rfloor_{S,C} \qquad \diamond
\end{array}
$$

This definition is similar to 4.20 except it tracks calls on methods of a parameter. Note that if C is an entity parameter, the identifier of the parameter itself is considered as its only method.

Let C be a parameter of the species S. We now address computing the minimal set of methods of C required to typecheck the method generator of the method m of S.

$$
{}^{[Body]}\lfloor m \rfloor_{S,C} = \lfloor \mathcal{B}_S(m) \rfloor_{S,C} \qquad {}^{[Type]}\lfloor m \rfloor_{S,C} = \lfloor \mathcal{T}_S(m) \rfloor_{S,C}
$$

$$
{}^{[Def]}\lfloor m \rfloor_{S,C} = \bigcup \lfloor \mathcal{B}_S(n) \rfloor_{S,C} \quad \text{for all } n \text{ such as } n <_S^{def} m
$$

$$
{}^{[Univ]}\lfloor m \rfloor_{S,C} = \bigcup \lfloor \mathcal{T}_S(n) \rfloor_{S,C} \quad \text{for all } n \text{ such as } n \in \mid m \mid
$$

The rule [BODY] (resp. [TYPE]) collects the syntactical dependencies on methods in the body (resp. type) of a definition. Note that the rule [TYPE] collects dependencies of properties and theorems upon only computational methods of C since properties cannot depend on properties.

The rule [DEF] collects dependencies of a method m on C searching them in the set of transitive def-dependencies of m's body in S. The rule [UNIV] collects dependencies of a method m on C searching them in the types of the methods belonging to the visible universe of m. These first four rules can be used before the following rule [PRM].

$$
\frac{\begin{array}{c}\Gamma_S(S) = (\ldots, C_p\ \textbf{is}\ ^{\cdots}, \ldots, C_{p'}\ \textbf{is}\ S'(\ldots, C_p, \ldots)) \\ \Gamma_S(S') = (\ldots, C'_k\ \textbf{is}\ I'_k, \ldots) \\ o \in {}^{[Type]}\lfloor m \rfloor_{S,C_{p'}} \ \vee\ o \in {}^{[Body]}\lfloor m \rfloor_{S,C_{p'}} \qquad (n : \tau_n) \in {}^{[Type]}\lfloor o \rfloor_{S',C'_k}\end{array}}{(n : \tau_n[C'_k \longmapsfrom C_p]) \in {}^{[Prm]}\lfloor m \rfloor_{S,C_p}}
$$

The rule [PRM] takes into account dependencies of a method on a previously introduced parameter, C_p, used as argument to build the current parameter $C_{p'}$. [PRM] returns a set of names with their type. The type τ_n coming from the rule ${}^{[Type]}\lfloor o \rfloor_{S',C'_k}$ is the type computed during the typechecking and the rule [PRM] modifies it. We explain it with the following example.

```
species Sprim (K is Base) =
  theorem prm_reflex: all z : K, K!equal (z, z) proof = ...
end ;;

species S (Cp is Base, Cpprim is Sprim (Cp)) =
  theorem prm_reflex2 : ... proof = by property Cpprim!prm_reflex ;
end ;;
```

In `Sprim`, the type of `K!equal` in the dependency of `prm_reflex` is its inferred one: `K` → `K` → `bool`. In the species `S`, the method `prm_reflex2` depends on `Cpprim!prm_reflex` which, in turn, depends on `equal`, but which `equal`? It does not depend on the one of `K` since this parameter is the formal one of `Sprim`. It depends on `equal` coming from the effective collection used to instantiate `K`, i.e. `Cp`. Hence, the type of this `equal` is no more `K` → `K` → `bool` as above, but `Cp` → `Cp` → `bool`. A substitution is required to replace `K` by `Cp`. Indeed, $^{[Type]}\lfloor\, o \,\rfloor_{S',C'_k}$ computes the dependencies in S', hence relatively to the parameters of S'. There is a dependency of `prm_reflex` in `Sprim` on `K!equal`. However, in `S`, the effective dependency is on the collection parameter `Cp` used to instantiate the formal parameter `K` of `Sprim` . This information is especially important because the code generation needs to emit typed identifiers to λ-lift dependencies.

Note that the methods n added as new dependencies of m by [PRM] only come from the type of o. Indeed C_p being a collection parameter, only the types of its methods are visible. These types can be "ML-like" ones, which contain no method names. They can also be logical formulae and the only names appearing in them are those of computational methods (since properties cannot depend on properties) which have "ML-like" types. Hence there is no risk that (logically) typing such an n needs to transitively add other methods.

None of these rules take into account decl-dependencies that methods of parameters have inside their own species and that are visible through their types. This is the role of the rule [CLOSE] which is applied once the five previous ones have been applied. The following example shows that, using `P!th0` to prove `th1` (which only makes reference to `P!f`) however needs to add `P!g` in the logical typing context, which is done by [CLOSE].

```
species A =
  signature f : Self -> int ;
  signature g : Self -> int ;
  property th0: all x : Self, f (x) = 0 /\ g (x) = 1 ;
end ;;

species B (P is A) =
  theorem th1 : all x : P, P!f (x) = 0 proof = by property P!th0 ;
end ;;
```

Omitting the application of this rule would lead to the following incorrect generated code where `_p_P_g` is unbound:

```
Theorem th1 (_p_P_T : Set) (_p_P_f : _p_P_T -> int)
  (_p_P_th0 :
    forall x : _p_P_T, Is_true (_p_P_f x = 0) /\ Is_true (_p_P_g x = 1)) :
  forall x : _p_P_T, Is_true (_p_P_f x  = 0).
```

although `_p_P_g` should also be λ-lifted to get a well-typed term like in the following (and correct) generated code:

```
Theorem th1 (_p_P_T : Set) (_p_P_f : _p_P_T -> int)
  (_p_P_g :  _p_P_T -> int)
  (_p_P_th0 :
    forall x : _p_P_T, Is_true (_p_P_f x = 0) /\ Is_true (_p_P_g x = 1)) :
  forall x : _p_P_T, Is_true (_p_P_f x  = 0).
```

The following and last rule serves to build the complete set of dependencies $\mathcal{D}^+$ upon an initial set of dependencies $\mathcal{D}$.

$$\frac{\begin{array}{c}\Gamma_S(S) = (\ldots, C_p \textbf{ is } I_p, \ldots) \\ o \in \mathcal{D}(S, C_p)[m] \qquad (n : \tau_n) \in \lbag \mathcal{T}_{I_p}(o) \rbag_{I_p}\end{array}}{(n : \tau_n[\texttt{Self} \hookleftarrow C_p]) \in \mathcal{D}^+(\mathcal{D}, S, C_p)[m]} \ \textsc{Close}$$

This rule does not consider def-dependencies because only types of parameters are visible (encapsulation of collections) and they have no def-dependencies. Moreover, for the same reasons given for [PRM], new dependencies brought by this rule cannot themselves require applying this rule again to make them logically typecheckable.

7. Code Generation

Code generation strongly relies on the computation of dependencies, done after the normalization of the species. Species lead to code only through their defined methods. For collections, the produced code comes from the aggregation of the methods of species, taking into account instantiations of parameters.

As stated in Section 2.6, the code generation model is the same for both the computational and the logical target languages. A common intermediate form is elaborated from which concrete syntaxes are finally emitted. Dependencies are indeed tagged to determine whether they come from only logical or also computational methods. The emission of computational code simply ignores logical methods, dependencies they induce and explicit polymorphism. We only present the logical code generation (Coq as target language) since the computational code (OCaml as target language) is just obtained by forgetting logical parts and minor syntactical adjustments.

Remember that a species is compiled into a module containing the generators of its methods. If the species is complete, this module also hosts a record type and the collection generator. A collection is also compiled into a module, containing a call to the collection generator of its underlying species, followed by the definitions of the methods coming from the application of the collection generators to the effective values of its parameters.

We present code generation for species and then, for collections. We adopt a top-down approach to first present the global shape of the code before refining details at each step. All along the code generation rules, we use the grayed `typewriter font` to denote code emitted in Coq syntax.

7.1. Species

The code emitted for a species S is a module (also named S) which possibly contains a record type definition ($\mathcal{R}$), then the code ($\mathcal{M}$) issued from the methods of S and possibly a collection generator ($\mathcal{C}$).

Definition 7.1. *Species Generation*

$$\frac{\mathcal{R} = \mathit{GenRecord}(S) \qquad \mathcal{M} = \mathit{GenMethsGens}(S) \qquad \Gamma_S(S) = (\overrightarrow{C \blacksquare t})\{\overrightarrow{\Phi}\} \qquad \mathcal{C} = \text{if } \forall m \in \mathcal{N}(\Phi),\ m \in \mathcal{D}(\Phi) \text{ then } \mathit{GenCollGen}(S) \text{ else } \varnothing}{S \longrightarrow \texttt{Module}\ S\texttt{.}\ \mathcal{R}\ \mathcal{M}\ \mathcal{C}\ \texttt{End}\ \mathrm{S}\texttt{.}}$$ ◇

The procedures $GenRecord$, $GenMethsGens$ and $GenCollGen$ are given in the following, according to the chosen top-down presentation.

As exposed in Section 5.2.2, the record contains all the applications already done of the method generators of the complete species being compiled. The record type exposes their types rf_m_k : τ_k. Only the representations C_i of the parameters followed by those of the methods of the parameters m_j remain λ-lifted in this record type, as follows:

Record me_as_species (C_i_T : Set) (_p_C_j_m_j: τ_j) : Type :=
 mk_record {
 rf_T : Set ;
 rf_m_k : τ_k ;
}

Note that the representation of the species (rf_T : Set) is always the first field of the record since it cannot depend on any other method of the species.

7.1.1. *Record Type Generation*

Definition 7.2. *Lifting Parameters in the Record Type*

Let m be a method of a species S. Let C be a parameter of S. We define :
${}^{[Rtype]}\lfloor m \rfloor_{S,C} = \mathcal{D}^+({}^{[Type]}\lfloor m \rfloor_{S,C})$ ◇

${}^{[Rtype]}\lfloor m \rfloor_{S,C}$ is computed from the rule [TYPE] completed by application of the rule [CLOSE] for this parameter C.

It is indeed the set of dependencies of m on the methods of C which have to be λ-lifted in the record type definition. Since the record type only shows the types of methods, the dependencies collected by the rule [BODY] are not needed. Moreover, def-dependencies do not appear in types, hence dependencies from the rules [DEF] and [UNIV] are also not relevant. Finally, the rule [PRM] is also not relevant since in the types present in the record, the methods of a parameter, even built by a species expression "application", are always abstracted relative to this parameter's name : the provided effective argument never appears. Let's consider the following example:

```
species Base =
  signature one : Self ;
  signature eq : Self -> Self -> bool ;
  property eq_spec: all x : Self, eq (x, x) ;
end ;;

species Sprim (K is Base) =
  let param_one = K!one ;
  theorem prm_refl: all z : K, K!eq (z, z) proof = by property K!eq_spec ;
end ;;

species S (Cp is Base, Cpprim is Sprim (Cp)) =
  rep = unit ;
  theorem prm_refl2 : Cp!eq (Cpprim!param_one, Cpprim!param_one)
  proof = by property Cpprim!prm_refl ;
end ;;
```

in which `prm_refl2` depends on `Cpprim!param_one`. There is no need to know that `Cpprim!param_one` is `Base!one`: it is lifted as `_p_Cpprim_param_one`:

```
Module S.
  Record me_as_species
    (Cp_T : Set) (Cpprim_T : Set) (_p_Cp_eq : Cp_T ->Cp_T -> bool)
    (_p_Cpprim_param_one : Cp_T) : Type :=
  mk_record {
    rf_T : Set ;
    rf_prm_refl2 :
      Is_true (_p_Cp_eq _p_Cpprim_param_one _p_Cpprim_param_one)
  }.
```

Later, when a collection will be built from `S`, the parameters `Cp` and `Cpprim` will be instantiated to determine the effective methods to apply to the collection generator. At this moment, `_p_Cpprim_param_one` will be instantiated by the effective value `one` of the collection used to instantiate `Cp`.

Definition 7.3. *Representations of Species Parameters*

Let S be a species such that $\Gamma_S(S) = (\overrightarrow{C \blacksquare t})\{\overrightarrow{\Phi}\}$, the set of parameters representations of S is defined by:

$$PReprs(\overrightarrow{C \blacksquare t}) = \begin{cases} \varnothing & \text{if } \overrightarrow{C \blacksquare t} = \ \varnothing \\ C_0 \oplus PReprs(\overrightarrow{C' \blacksquare t'}) & \text{if } \overrightarrow{C \blacksquare t} = \ C_0 \texttt{ is } se \ ; \ \overrightarrow{C' \blacksquare t'} \\ PReprs(\overrightarrow{C' \blacksquare t'}) & \text{if } \overrightarrow{C \blacksquare t} = \ c \texttt{ in } C_0 \ ; \ \overrightarrow{C' \blacksquare t'} \end{cases} \quad \diamond$$

This set represents the first λ-lifted material of the record type definition. In case of an entity parameter, no representation is recorded. Indeed if this parameter has a type containing a collection parameter of the species, then this latter inevitably appeared before the entity parameter and was already recorded. Otherwise, the entity parameter inevitably has a type containing a toplevel collection. The same remark applies on species expressions se.

Definition 7.4. *Record Type Creation: GenRecord*

The generation of the record type, done as follows and presented with some abuses of notation, is only done for a complete species. This operation relies on $[\![\tau]\!]^t$ the translation of types in the target language (c.f. definition 7.17).

- Let S be a species in normal form.
- Let $\Gamma_S(S) = (\overrightarrow{C_n \blacksquare t_n})\{\overrightarrow{\Phi_m}\}$
- Let $\mathcal{C} = \texttt{(}C_j\texttt{\_T : Set)}$ for all $C_j \in PReprs(\overrightarrow{C_n \blacksquare t_n})$, the code operating the λ-liftings of parameters representations.
- Let $pm = \bigcup_{j=1}^{n}(C_j \times \bigcup_{i=1}^{m}({}^{[Rtype]}\lfloor\ m_i\ \rfloor_{S,C_j}))$ an association list between a parameter name C_j and the list of its methods upon which at least one method of S depends, according to the rule [RTYPE], thus the list of methods of C_j which must be λ-lifted in the record type. The generated code for theses λ-liftings is given by $\mathcal{M} = \forall (C_j \times ms) \in pm, \forall m \in ms,\ \texttt{(\_p\_}C_j\texttt{\_}x \texttt{ : } [\![\mathcal{T}_{C_j}(m)]\!]^t\texttt{)}$.

- Let $\mathcal{F}$ be the list of the names of methods of S, with their logical types.

The type of the record is built as follows:

$$\dfrac{\begin{array}{c}\Gamma_S(S) = (\overrightarrow{C_n \blacksquare t_n})\{\overrightarrow{\Phi_m}\} \qquad \mathcal{C} = \forall C_j \in PReprs(\overrightarrow{C_n \blacksquare t_n}),\ \texttt{(}C_j\texttt{\_T : Set)}\\ pm = \bigcup_{j=1}^{n}(C_j \times \bigcup_{i=1}^{m}({}^{[Rtype]}\lfloor\, x_i \,\rfloor_{S,C_j}))\\ \mathcal{M} = \forall (C_j \times ms) \in pm, \forall m \in ms,\ \texttt{(\_p\_}C_j\texttt{\_}m\texttt{:}[\![\mathcal{T}_{C_j}(x)]\!]^t\texttt{)}\\ \mathcal{F} = \forall m : \tau \in \{\overrightarrow{\Phi_m}\},\ \texttt{rf\_}m\texttt{:}[\![\tau]\!]^t\texttt{.}\end{array}}{\begin{array}{c}S \longrightarrow \texttt{Record me\_as\_species}\ \mathcal{CM}\ \texttt{(abst\_T: Set): Type :=}\\ \texttt{mk\_record \{}\mathcal{F}\texttt{\}.}\end{array}}$$

◇

Note that by $\mathcal{C}$, the record type is λ-lifted by all the species parameters representations, even if some do not effectively appear inside the record's fields. It is mostly uncommon to have species not using their parameters, hence we choose to have a simplified abstraction computation.

Definition 7.5. *Dependencies of a Method on Methods of a Parameter: Last Stage*
Let m be a method of a species S. Let C be a parameter of S. ${}^{[Gen]}\lfloor\, m \,\rfloor_{S,C}$ is the complete set of dependencies of m on the methods of C. It is defined by:

$$\begin{aligned}{}^{[Gen]}\lfloor\, m \,\rfloor_{S,C} = {}& {}^{[Body]}\lfloor\, m \,\rfloor_{S,C} \cup {}^{[Type]}\lfloor\, m \,\rfloor_{S,C} \cup \\ & \mathcal{D}^{+}({}^{[Def]}\lfloor\, m \,\rfloor_{S,C} \cup {}^{[Univ]}\lfloor\, m \,\rfloor_{S,C} \cup {}^{[Prm]}\lfloor\, m \,\rfloor_{S,C}, S, C)[m]\end{aligned}$$

◇

${}^{[Gen]}\lfloor\, m \,\rfloor_{S,C}$ gathers the dependencies of m on C's methods found by applying the rules [TYPE], [BODY] and the rule [CLOSE] on the union of dependencies obtained by [DEF], [UNIV] and [PRM]. This function is used to compute the λ-liftings for method generators (c.f. 7.8), the arguments of the methods they def-depend on (c.f. 7.9), and for local applications of method generators in collection generators (c.f 7.11). It is someway the largest set of dependencies.

7.1.2. *Parameters Instantiation*

Let S be a species, parametrized by C_p, containing a method generator for a method m having dependencies on some methods of C_p. The following rules recursively determine the instantiation of C_p along the inheritance tree of S, i.e. what the formal parameter C_p refers to.

The following example illustrates the need to follow the instantiations of parameters to know the right effective methods to provide a method generator with.

```
species P0 =
  rep = int ;
  let m = 5 ;
end ;;

species S1 (P is P0) =
  rep = int ;
  let v = P!m ;
end ;;

species S2 (Q is P0) =
  inherit S1 (Q) ;
end ;;
```

The species P0 defines a method m. S1 has a parameter P of interface P0 and defines a method v depending on its parameter P's method m. Then S2 also has a parameter Q of interface P0 and inherits from S1. During this instantiation, the parameter P of S1 is instantiated by the parameter `Q` of `S2`. Since `S2` inherits from `S1`, `S2` has a method `v`.

In `S1` from where `v` comes, `v` depends on the parameter `P`. However, once in `S2`, it now depends on "by what" `P` has been instantiated during the inheritance: the parameter `Q` of `S2`. Hence, in the collection generator of `S2`, the method generator of `v` must be applied to the λ-lifting of `Q!m` (i.e. `_p_Q_m` in the produced code) and not to something related to (the unbound in `S2`) parameter `P`.

```
species P1 =
  inherit P0
  let m = 25 ;
end ;;

collection C0 = implement P0 ;;
collection C1 = implement P1 ;;
collection D0 = implement S1 (C0) ;;
collection D1 = implement S1 (C1) ;;
```

Following the parameters instantiations is also required when creating collections. The species `P1` inherits from `P0` and redefines its method `m`. The collections `C0` and `C1` are respectively created upon `P0` and `P1`. When creating the collections `D0` and `D1` upon `S1`, the parameter `P` of `S1` is instantiated by two different collections, the first one where `m = 5`, the second one where `m = 25`.

There are three possible kinds of instantiations of parameters: by a collection parameter (**CP**), by a toplevel collection (**TC**) and, for entity parameters, by an expression (**EP**).

Intuitively, the search for instantiations starts in the species S_0 containing the definition of the method m that depends on a parameter C_p. Then, it recursively walks along the inheritance tree toward more recent species S_i inheriting from S_{i-1}. At each step the formal parameter (initially C_p) of S_{i-1} is replaced by the effective argument provided in S_i.

Definition 7.6. *Provenance of a Method*

Let S be a species of the form **species** S **= inherit** $\overrightarrow{se}\,\overrightarrow{\phi}$ and a method $m \in \mathcal{D}(S)$. The provenance of m in S (i.e. along the inheritance tree) is defined as follows.

$$\frac{m \in \mathcal{D}(S)}{m \Rsh S = S} \qquad \frac{m \notin \mathcal{D}(S) \qquad m \Rsh se_i = S' \ \text{ with } i \text{ the highest index in } \overrightarrow{se}}{m \Rsh S = S'} \qquad \diamond$$

These rules search the definition of m in S first, otherwise in the latest inheritance node defining m (in case of multiple inheritance, the rightmost species in the **`inherit`** clause is considered first as stated in 2.2).

Definition 7.7. *Parameter Instantiation*

Depending on the kind of expression instantiating a parameter, the methods of this latter are accessed differently. Note that when instantiating by a toplevel collection, substitutions have no effect. When instantiating by a collection parameter, the substitution trivially consists in returning this collection parameter. When instantiating an entity parameter, the substitution takes all its sense since the effective expression used to instantiate has to replace each occurrence of the formal parameter.

$$\frac{m \Rsh S = S}{\mathit{PInst}(S, C_p, m) = \texttt{CP}(C_p)}$$

$$\frac{\begin{array}{c} m \Rsh S = S' \text{ with } S' \ddagger S \\ S(\ldots, C_p \blacksquare t_p, \ldots) = \textbf{inherit } S'(\ldots, e_p, \ldots) \qquad \Gamma_S(S') = (\ldots, C'_p \blacksquare t'_p, \ldots)\{\overrightarrow{\Phi}\} \end{array}}{\mathit{PInst}(S, C_p, m) = \mathit{IKind}_S(e_p)}$$

$$\frac{\begin{array}{c} m \Rsh S = S'' \text{ with } S'' \ddagger S' \ddagger S \\ S(C_1 \blacksquare t_1, \ldots, C_p \blacksquare t_p, \ldots) = \textbf{inherit } S'(e_1, \ldots, e_p, \ldots) \\ \Gamma_S(S') = (C'_1 \blacksquare t'_1, \ldots C'_p \blacksquare t'_p, \ldots)\{\overrightarrow{\Phi}\} \qquad \mathit{PInst}(S', C_p, m) = \mathit{ik} \end{array}}{\mathit{PInst}(S, C_p, m) = \mathit{ik}[C_p \mapsfrom e_p]} \qquad \diamond$$

$$\mathit{IKind}_S(e) = \begin{cases} \texttt{CP}(e) & \text{if } e \text{ is a collection parameter of } S \\ \texttt{TC}(e) & \text{if } e \text{ is a toplevel collection of } S \\ \texttt{EP}(e) & \text{if } e \text{ is an expression (for an entity parameter)} \end{cases}$$

The following functions *GenRepType* and *GenPrmMeth* serve to instantiate the λ-liftings induced by the representations of parameters and the methods, according to the parameters' instantiations (by other parameters, toplevel collection, or expression for entity parameters). The effective definition of the record field `me_as_carrier` is provided by the forthcoming definition 7.14.

$$\mathit{GenRepType}(\mathit{ik}) = \begin{cases} \texttt{\_p\_}e\texttt{\_T} & \text{if } \mathit{ik} = \texttt{CP}(e) \\ e\texttt{.me\_as\_carrier} & \text{if } \mathit{ik} = \texttt{TC}(e) \\ \epsilon & \text{if } \mathit{ik} = \texttt{EP}(e) \end{cases}$$

and:

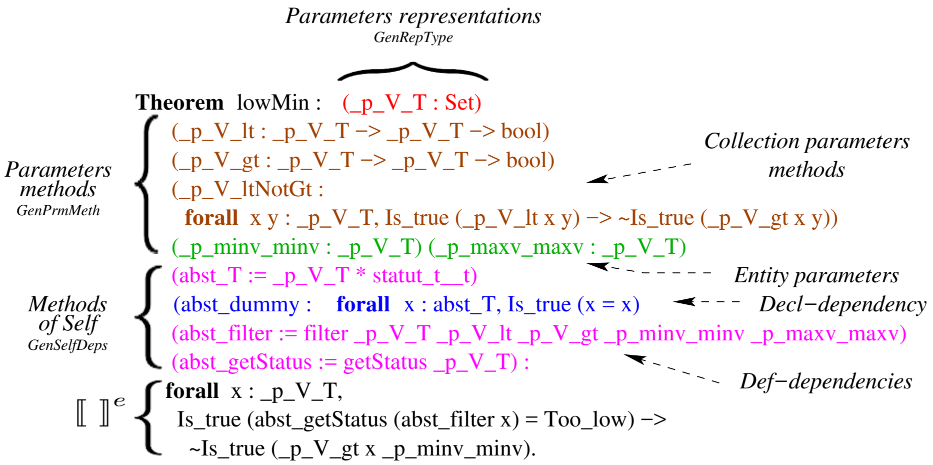
$$\mathit{GenPrmMeth}(\mathit{ik}, \overrightarrow{ms}) = \begin{cases} \texttt{\_p\_}e\texttt{\_}m & \forall m \in \overrightarrow{ms} \text{ if } \mathit{ik} = \texttt{CP}(e) \\ e\texttt{.}m & \forall m \in \overrightarrow{ms} \text{ if } \mathit{ik} = \texttt{TC}(e) \\ e & \text{if } \mathit{ik} = \texttt{EP}(e) \end{cases}$$

7.1.3. Method Code Generation

In order to help the intuitive understanding of the relations between the rules, we illustrate the shape of a generated code for a method in the following figure.

We consider the method `lowMin` of the example introduced in Section 2.3. In order to also illustrate decl-dependencies, we voluntarily use an unnecessary method `dummy` (a property simply stating $\forall x, x = x$) as a hint of a proof step as shown is the following listing.

```
theorem lowMin :
  all x : V, getStatus (filter (x)) = Too_low -> ~ V!gt(x, minv)
proof =
 <1>1 assume x : V,
      hypothesis H: snd (filter (x)) = Too_low,
      prove ~ V!gt (x, minv)
      <2>1 prove V!lt (x, minv) by definition of filter type statut_t
           hypothesis H
      <2>2 qed by step <2>1 property V!ltNotGt, dummy
 <1>2 qed by step <1>1 definition of getStatus ;
```



```
Theorem lowMin : (_p_V_T : Set)
  (_p_V_lt : _p_V_T -> _p_V_T -> bool)
  (_p_V_gt : _p_V_T -> _p_V_T -> bool)
  (_p_V_ltNotGt :
    forall x y : _p_V_T, Is_true (_p_V_lt x y) -> ~Is_true (_p_V_gt x y))
  (_p_minv_minv : _p_V_T) (_p_maxv_maxv : _p_V_T)
  (abst_T := _p_V_T * statut_t__t)
  (abst_dummy : forall x : abst_T, Is_true (x = x)
  (abst_filter := filter _p_V_T _p_V_lt _p_V_gt _p_minv_minv _p_maxv_maxv)
  (abst_getStatus := getStatus _p_V_T) :
  forall x : _p_V_T,
   Is_true (abst_getStatus (abst_filter x) = Too_low) ->
    ~Is_true (_p_V_gt x _p_minv_minv).
```

The function $GenMethGen$ is the entry point for generating the code of the method generator of a method m belonging to a species S. It relies on three intermediate functions ($GenRepType$, $GenPrmMeth$ and $GenSelfDeps$).

The first two rules of $GenMethGen$ state that methods only declared or defined in an inherited species do not lead to code.

Definition 7.8. *Code Generation for Methods*

$$\frac{m \notin \mathcal{D}(S)}{GenMethGen(S, m) = \epsilon}\,[\text{MG-DECL}] \qquad \frac{m \in \mathcal{D}(S) \qquad m \Rsh S \neq S}{GenMethGen(S, m) = \epsilon}\,[\text{MG-INH}]$$

Note that in the following rule [MG-METH], depending on the kind of method, the binder may change: `Definition` for a computational method, `Theorem` for a theorem (not written in the rule).

- Let S be a species in normal form.
- Let $\Gamma_S(S) = (\overrightarrow{C \blacksquare t})\{\overrightarrow{\Phi}\}$
- Let $(pr_i, ms_i) = (C_i, {}^{[Gen]}\lfloor m \rfloor_{S,C_i})$ for all $C_i \in \overrightarrow{C \blacksquare t}$, the list of couples recording each parameter C_i and the list of dependencies of m on C_i.
- Let $insts = (PInst(S, C_i, m) \times ms_i)$, the list of couples recording the instantiation of each parameter C_i and the list of dependencies of m on C_i.
- Let $\mathcal{C} = GenRepType(ik)$ for all $(ik, ms) \in insts$, the code operating the λ-liftings of parameters representations.
- Let $\mathcal{P} = GenPrmMeth(ik, ms)$ for all $(ik, ms) \in insts$, the code operating the λ-liftings of parameters methods.
- Let $\mathcal{S} = GenSelfDeps_S(\{\overrightarrow{\Phi}\} \Cap m)$, the code operating the λ-liftings or bindings for decl- or def-dependencies on other methods of the species S.

$$\frac{\begin{array}{c}\Gamma_S(S) = (\overrightarrow{C \blacksquare t})\{\overrightarrow{\Phi}\} \qquad \forall C_i \in \overrightarrow{C \blacksquare t}, (pr_i, ms_i) = (C_i, {}^{[Gen]}\lfloor\, m \,\rfloor_{S,C_i}) \\ insts = \forall C_i \in pr_i,\ (\mathit{PInst}(S, C_i, m) \times ms_i) \\ \mathcal{C} = \forall (ik, ms) \in insts,\ \mathit{GenRepType}(ik) \\ \mathcal{P} = \forall (ik, ms) \in insts,\ \mathit{GenPrmMeth}(ik, ms) \\ \mathcal{S} = \ \mathit{GenSelfDeps}_S(\{\overrightarrow{\Phi}\} \Cap m)\end{array}}{\begin{array}{c}GenMethGen(S, m) = \\ \texttt{Definition}\ \mathrm{m}\,\mathcal{C}\ \mathcal{P}\ \mathcal{S}\,\texttt{:}\ [\![\mathcal{T}_S(m)]\!]^t\ \texttt{:=}\ [\![\mathcal{B}_S(m)]\!]^e\ \texttt{.}\end{array}}\,[\text{MG-M}\textsc{eth}]$$

Finally, the last rule, [MG-SPE] iterates on all the methods of the species.

$$\frac{\Gamma_S(S) = (\overrightarrow{C \blacksquare t})\{\overrightarrow{\Phi}\}}{GenMethGens(S) = \forall m \in \{\overrightarrow{\Phi}\}, GenMethGen(S, m)}\,[\text{MG-S}\textsc{pe}]$$

$\diamond$

Emitting λ-Liftings and Bindings. The following function *GenSelfDeps* emits the λ-liftings due to decl-dependencies and the definitions (`:=`-bindings) due to def-dependencies. If m has a def-dependency on n, then the method generator of n is applied to its own dependencies. These dependencies of n are methods either already λ-lifted or defined (`:=`-bound) as n is known when m is introduced.

Definition 7.9. *λ-lifting Dependencies on Methods of* `Self`

$$\mathit{GenSelfDeps}_S(\varnothing) = \epsilon$$

$$\mathit{GenSelfDeps}_S(n : \tau; l) = \texttt{(abst\_n :}\ [\![\tau]\!]^t\texttt{)}\ \mathit{GenSelfDeps}_S(l)$$

$$\mathit{GenSelfDeps}_S(\texttt{rep}; l) = \texttt{(abst\_T : Set)}\ \mathit{GenSelfDeps}_S(l)$$

$$\mathit{GenSelfDeps}_S(\texttt{rep} = \tau; l) = \texttt{(abst\_T :=}\ [\![\tau]\!]^t\texttt{)}\ \mathit{GenSelfDeps}_S(l)$$

- Let S be a species in normal form.
- Let $\Gamma_S(S) = (\overrightarrow{C \blacksquare t})\{\overrightarrow{\Phi}\}$
- Let $S' = n \Rsh S$ the species in which n is defined.
- Let $\mathcal{N}$ be the code emitted to access the method generator of n, depending on its provenance.
- Let $(pr_i, ms_i) = (C_i, {}^{[Gen]}\lfloor\, n \,\rfloor_{S,C_i})$ for all $C_i \in \overrightarrow{C \blacksquare t}$, the list of couples recording each parameter C_i and the list of dependencies of n on C_i.
- Let $insts = (\mathit{PInst}(S, C_i, n) \times ms_i)$, the list of couples recording the instantiation of each parameter C_i and the list of dependencies of n on C_i.
- Let $\mathcal{C} = \mathit{GenRepType}(ik)$ for all $(ik, ms) \in insts$, the code operating the λ-liftings of parameters representations.
- Let $\mathcal{P} = \mathit{GenPrmMeth}(ik, ms)$ for all $(ik, ms) \in insts$, the code operating the λ-liftings of parameters methods.
- Let $\mathcal{S} = \texttt{(abst\_}m_i\texttt{ :}\ [\![\tau]\!]^t\texttt{)}$ for all $(m_i : \tau_i) \in \{\overrightarrow{\Phi}\} \Cap n$, the code operating the application of the method generator of n to the dependencies it has on other methods of S.

$$\frac{\begin{array}{c} S' = n \Rsh S \qquad \mathcal{N} = \text{ if } S' = S\text{, then } n \text{ else S'}\texttt{.}n \\ \Gamma_S(S) = (\overrightarrow{C \blacksquare t})\{\overrightarrow{\Phi}\} \qquad \forall C_i \in \overrightarrow{C \blacksquare t}, (pr_i, ms_i) = (C_i, {}^{[Gen]}\lfloor\, n \,\rfloor_{S,C_i}) \\ insts = \forall C_i \in pr_i,\ (\mathit{PInst}(S, C_i, n) \times ms_i) \\ \mathcal{C} = \forall (ik, ms) \in insts,\ \mathit{GenRepType}(ik) \\ \mathcal{P} = \forall (ik, ms) \in insts,\ \mathit{GenPrmMeth}(ik, ms) \\ \mathcal{S} = \forall (m_i : \tau_i) \in \{\overrightarrow{\Phi}\} \Cap n,\ \texttt{(abst\_}m_i\texttt{ : } [\![\tau]\!]^t\texttt{)} \end{array}}{\mathit{GenSelfDeps}_S(n : \tau = e; l) = \texttt{(abst\_}n\texttt{ := }\mathcal{N}\mathcal{C}\ \mathcal{P}\ \mathcal{S}\texttt{)}\ \mathit{GenSelfDeps}_S(l)} \qquad \diamond$$

7.1.4. *Collection Generator Code*

If a species is complete, its collection generator is emitted by the following function $\mathit{GenCollGen}$. $\mathcal{C}$ is the code for λ-lifting the representations of parameters. pm is an association list between a parameter name C_j and the list of its methods upon which at least one method of S depends. Note that $\mathcal{C}$ and pm are the same as those in $\mathit{GenRecord}$ (technically, they are returned by this latter). $\mathcal{P}$ is the set of names used to λ-lift the methods in pm.

The applications of each method generator to its arguments are grouped inside a record. The code generation for these applications is the same as when these generators are used for **:**=-bindings in $\mathit{GenMethGen}$ (c.f. definition 7.1.3). In these applications, the methods coming from the collection parameters are still not defined hence must be λ-lifted. Hence the collection generator is the function parametrized by these λ-lifted methods, which returns a value of the record.

Definition 7.10. *Collection Generator Code Generation*

- Let S be a species in normal form.
- Let $\Gamma_S(S) = (\overrightarrow{C_n \blacksquare t_n})\{\overrightarrow{\Phi_m}\}$
- Let $\mathcal{C} = \texttt{(}C_j\texttt{\_T : Set)}$ for all $C_j \in \mathit{PReprs}(\overrightarrow{C_n \blacksquare t_n})$, the code operating the λ-liftings of parameters representations.
- Let $pm = \bigcup_{j=1}^{n}(C_j \times \bigcup_{i=1}^{m}({}^{[Rtype]}\lfloor\, m_i \,\rfloor_{S,C_j}))$ an association list between a parameter name C_j and the list of its methods upon which at least one method of S depends, according to the rule [RTYPE], thus the list of methods of C_j which must be λ-lifted in the collection generator.
- Let $\mathcal{P} = \texttt{\_p\_}C_j\texttt{\_}m$ for all $(C_j \times ms) \in pm$ and $m \in ms$, the code operating the λ-liftings of parameters methods.
- Let $\mathcal{G}$ be the code creating the local applications of the methods generators to their dependencies (from other methods of the species S or from parameters methods).
- Let $\mathcal{M}$ be the code emitted to apply the record constructor to the local applications built by $\mathcal{G}$.

$$\frac{\begin{array}{c}\Gamma_S(S) = (\overrightarrow{C_n \blacksquare t_n})\{\overrightarrow{\Phi_m}\} \qquad \mathcal{C} = \forall C \in PReprs(\overrightarrow{C_n \blacksquare t_n}),\ \texttt{(}C\texttt{\_T : Set)} \\ pm = \bigcup_{j=1}^{n}(C_j \times \bigcup_{i=1}^{m}({}^{[Rtype]}\lfloor m_i \rfloor_{S,C_j})) \\ \mathcal{P} = \forall (C_j \times ms) \in pm, \forall m \in ms,\ \texttt{\_p\_}C_j\texttt{\_}m \\ \mathcal{G} = \forall \Phi_i \in \overrightarrow{\Phi_m},\ \forall m \in \mathcal{D}(\Phi_i),\ GenLocal_S(m) \\ \mathcal{M} = \forall \Phi_i \in \overrightarrow{\Phi_m},\ \forall m \in \mathcal{D}(\Phi_i) \setminus \texttt{rep},\ \texttt{local\_}m\end{array}}{\begin{array}{c}GenCollGen(S) = \texttt{Definition collection\_create}\ \mathcal{C}\ \mathcal{P}\ \texttt{:=} \\ \mathcal{G}\ \texttt{mk\_record}\ \mathcal{C}\ \mathcal{P}\ \texttt{local\_rep}\ \mathcal{M}\ \texttt{.}\end{array}} \qquad \diamond$$

Since the representation is the first method of the species λ-lifted in the record type, `local_rep` is explicitly processed first in $GenCollGen$. Note that $\mathcal{C}$ and pm are the same as in $GenRecord$ (technically, they are returned by this latter).

Definition 7.11. *Local Applications in a Collection Generator*

$$\frac{\begin{array}{c}S' = m \Rsh S \qquad \mathcal{N} = \text{ if } S' = S, \text{then } m \text{ else S'}\texttt{.}m \\ \Gamma_S(S) = (\overrightarrow{C \blacksquare t})\{\overrightarrow{\Phi}\} \qquad \forall C_i \in \overrightarrow{C \blacksquare t}, (pr_i, ms_i) = (C_i, {}^{[Gen]}\lfloor m \rfloor_{S,C_i}) \\ insts = \forall C_i \in pr_i,\ (PInst(S, C_i, m) \times ms_i) \\ \mathcal{C} = \forall (ik, ms) \in insts,\ GenRepType(ik) \\ \mathcal{P} = \forall (ik, ms) \in insts,\ GenPrmMeth(ik, ms) \\ \mathcal{S} = \forall (m_i : \tau_i) \in \{\overrightarrow{\Phi}\} \Cap m,\ \texttt{local\_}m_i\end{array}}{GenLocal_S(m : \tau = e) = \texttt{let local\_}m \texttt{ = } \mathcal{N}\mathcal{C}\ \mathcal{P}\ \mathcal{S}\ \texttt{in}} \qquad \diamond$$

The rule $GenLocal_S$ is very similar to $GenSelfDeps_S$. It mostly forgets type annotations and prefixes each method name by `local_` instead of `abst_`. This is not surprising since the rule $GenLocal_S(m)$ provides m with the effective methods on which it depends: these methods have previously be λ-lifted by $GenSelfDeps_S$.

7.2. Collections

The following $InstCollGenPrms$, in addition to generate code, returns the number of arguments generated to instantiate the λ-liftings of the record type. This returned number is used to know how many placeholders must be generated (for Coq) when generating records access in $GenMeths$ (c.f. definition 7.14).

Definition 7.12. *Collection Generation*

$$\frac{\begin{array}{c}(\mathcal{P}, n\_rtype\_args) = InstCollGenPrms(S, \overrightarrow{e}) \\ \mathcal{M} = GenMeths(C, S, n\_rtype\_args) \qquad \overrightarrow{m'} = SubstInMeths(S, \overrightarrow{e})\end{array}}{\begin{array}{c}\textbf{collection } C \textbf{ implement } S(\overrightarrow{e}) \longrightarrow \texttt{Module}\ C\ \texttt{.} \\ \texttt{Let effective\_collection :=}\ C\texttt{.collection\_create}\ \mathcal{P}\ \texttt{.}\mathcal{M} \\ \texttt{End}\ C\ \texttt{.}\end{array}} \qquad \diamond$$

Let pm be the list of methods per parameter computed in $GenRecord$ (c.f. pm in the definition 7.4) (technically it is recorded in an environment, not computed again). This information represents the arguments expected by the collection generator. The instantiation of a collection generator's parameters structurally applies the substitution σ. In the following rule, C' and P' represent a collection (toplevel or parameter) name.

Definition 7.13. *Collection Generator Parameters Instantiation*

- Let S be a species (in normal form).
- Let $\Gamma_S(S) = (\overrightarrow{C_n \blacksquare t_n})\{\overrightarrow{\Phi_m}\}$
- Let $\sigma = [\overrightarrow{C_i \hookleftarrow e_i}]$, the substitution mapping, for each C_i in $\overrightarrow{C_n \blacksquare t_n}$, its effective value e_i.
- Let $pr = PReprs(\overrightarrow{C_n \blacksquare t_n})$ be the list of the parameters representations.
- Let $\mathcal{C}$ be the code emitted to instantiate the λ-liftings corresponding to the types of parameters representations by collections representations if the parameters are collection parameters. In case of entity parameter no code is emitted.
- Let $\mathcal{M}$ be the code emitted to instantiate the λ-liftings corresponding to the methods of the parameters.

$$\frac{\begin{array}{c}\Gamma_S(S) = (\overrightarrow{C_n \blacksquare t_n})\{\overrightarrow{\Phi}\} \qquad pr = PReprs(\overrightarrow{C_n \blacksquare t_n}) \\ \sigma = [\overrightarrow{C_i \hookleftarrow e_i}] \qquad \mathcal{C} = \forall C_j \in pr,\ \text{if } \sigma(C) = C' \text{ then } C' \texttt{.me\_as\_carrier} \\ \mathcal{M} = \forall (P, ms) \in pm,\ \text{if } \sigma(P) = P' \text{ then } \forall m \in ms,\ P' \texttt{.} m \text{ else } [\![\sigma(C)]\!]^e\end{array}}{InstCollGenPrms(S, \overrightarrow{e_i}) = \mathcal{C}\ \mathcal{M}} \quad \diamond$$

Note that the application of σ can only return either a collection identifier (denoted in the above rule by P' and C') or an expression. The first case corresponds to the instantiation of a collection parameter. The second case is an instantiation of an entity parameter. In case of entity ("else" case), the instantiation simply translates the obtained expression.

Definition 7.14. *Collection's Methods Generation*

$$\frac{\begin{array}{c}\Gamma_S(S) = (\overrightarrow{C \blacksquare t})\{rep = \tau;\ \overrightarrow{\Phi}\} \qquad \mathcal{M} = \forall (m_i : \tau_i) \in \overrightarrow{\Phi}, \\ \texttt{Definition } m_i \texttt{ := effective\_collection.(} m_i \underbrace{\texttt{\_ } \ldots \texttt{ \_}}_{n\_rtype\_args} \texttt{).}\end{array}}{\begin{array}{c}GenMeths(C, S, n\_rtype\_args) = \\ \texttt{Definition me\_as\_carrier :=} [\![\tau]\!]^t \texttt{.}\ \mathcal{M}\end{array}} \quad \diamond$$

The record type has as many arguments as the collection generator has. The collection generator λ-lifts the dependencies of the record type. This number of arguments is recorded when the parameters of the collection generator are instantiated (c.f. definition 7.13).

Due to explicit polymorphism in Coq, field accesses in records require to explicitly mention the arguments of the record that encode polymorphism. We could have explicitly emitted the type expressions, however hopefully, the Coq type inference is able to recover them as long as their number is explicit, which is done by providing anonymous arguments (the placeholders denoted by _ above the horizontal brace in m_i _ ... _).

7.3. Translation of Types and Core Expressions

Several rules generate target code for both types and expressions.

For expressions, the main difficulty toward Coq is the generation of explicit polymorphism. Any polymorphic definition has an extra fresh parameter of type `Set` to denote each type variable. Hence, each instantiation of the definition must be provided with as many arguments of type `Set` as the definition has polymorphic type variables. Again, Coq is able to infer these types as long is it provided with the right number of placeholders (_). However, we need to maintain a mapping $\mathcal{V}$ between fresh generated parameters and their related type variables.

Emitting code for methods names needs to discriminate on their provenance which can be the species itself, a toplevel collection, a collection parameter or an entity parameter. Entity parameters fall into the regular case of identifiers. We omit the case where the toplevel collection belongs to another compilation unit, in which case the name of this unit should be added as prefix with a "dot notation".

In $\llbracket\ \rrbracket^e$, the translation of expressions, we voluntarily omit the code generation of recursive functions. It requires a very special processing due to the need of termination proofs, which is addressed in a dedicated paper [39]. Presenting it here should increase a lot the length of this paper.

Definition 7.15. *Translation of Expressions*

$$\dfrac{\Gamma_e(x) = \forall\overrightarrow{\alpha_n}.\tau}{\Gamma_e, \mathcal{V} \vdash \llbracket x \rrbracket^e = \mathtt{(}\,x\ \underbrace{\ \mathtt{\_}\ }_{n \text{ times}}\ \mathtt{)}} \qquad \Gamma_S, \Gamma_e, \mathcal{V} \vdash \llbracket \mathtt{Self}!m \rrbracket^e = \mathtt{abst\_}m$$

$$\dfrac{\Gamma_S \vdash \mathtt{C} \text{ is a toplevel collection}}{\Gamma_S, \Gamma_e, \mathcal{V} \vdash \llbracket C!m \rrbracket^e = C\,\mathtt{.}\,m} \qquad \dfrac{\Gamma_S \vdash \mathtt{C} \text{ is a collection parameter}}{\Gamma_S, \Gamma_e, \mathcal{V} \vdash \llbracket C!m \rrbracket^e = \mathtt{\_p\_}C\mathtt{\_}m}$$

$$\dfrac{\Gamma_e \vdash x_1 : \tau_1 \quad \dots \quad \Gamma_e \vdash x_n : \tau_n \qquad \Gamma_e \vdash e : \tau \qquad \Gamma_S, \Gamma_e, \mathcal{V} \vdash \llbracket e \rrbracket^e : e'}{\Gamma_S, \Gamma_e, \mathcal{V} \vdash \llbracket \textbf{fun } x_1, \dots, x_n \texttt{ ->} e \rrbracket^e = \mathtt{(}x_1 \llbracket \tau \rrbracket^t{}_1\mathtt{)} \dots \mathtt{(}x_n \llbracket \tau \rrbracket^t{}_n\mathtt{)}\ \mathtt{:}\ \llbracket \tau \rrbracket^t\ \mathtt{:=}\ e'}$$

$$\Gamma_S, \Gamma_e, \mathcal{V} \vdash \llbracket e\ (e_1, \dots, e_n) \rrbracket^e = \mathtt{(}\llbracket e \rrbracket^e\ \llbracket e_1 \rrbracket^e\ \dots\ \llbracket e_n \rrbracket^e\mathtt{)}$$

$$\Gamma_S, \Gamma_e, \mathcal{V} \vdash \llbracket \textbf{let } x = e_1 \textbf{ in } e_2 \rrbracket^e = \mathtt{let}\ x\ \mathtt{:=}\ \llbracket e_1 \rrbracket^e\ \mathtt{in}\ \llbracket e_2 \rrbracket^e$$

$$\dfrac{\begin{array}{c}\Gamma_e \vdash e : \tau \qquad Gen(\tau, \Gamma_e) = \forall\alpha_1 \dots \alpha_n.\tau \\ v_1 \dots v_n \text{ fresh variables} \qquad \Gamma_S, \Gamma_e, \mathcal{V} \oplus (\alpha_i \mapsto v_i) \vdash \llbracket e \rrbracket^e = e'\end{array}}{\Gamma_S, \Gamma_e, \mathcal{V} \vdash \llbracket e \rrbracket^e = \mathtt{(}v_1\mathtt{:\ \ Set)} \dots \mathtt{(}v_n\mathtt{:\ \ Set)}\ e'} \qquad \diamond$$

Definition 7.16. *Translation of Logical Expressions*

Logical expressions of type `bool` need to be explicitly injected in the type of logical properties (`Prop`) using Coq's construct `Is_true`. In effect, expressions of type `bool` are computable terms: their value can be computed although logical properties are not always decidable.

$$\dfrac{\Gamma_S, \Gamma_e, \mathcal{V} \vdash e : \mathtt{bool}}{\Gamma_S, \Gamma_e, \mathcal{V} \vdash \llbracket e \rrbracket^p = \mathtt{(Is\_true}\ \llbracket e \rrbracket^e\mathtt{)}} \qquad \dfrac{\Gamma_S, \Gamma_e, \mathcal{V} \vdash e : \mathtt{prop}}{\Gamma_S, \Gamma_e, \mathcal{V} \vdash \llbracket e \rrbracket^p = \llbracket e \rrbracket^e}$$

$$\frac{}{\Gamma_S, \Gamma_e, \mathcal{V} \vdash [\![\sim p]\!]^p = \texttt{\~{}}\,[\![p]\!]^p} \qquad \frac{}{\Gamma_S, \Gamma_e, \mathcal{V} \vdash [\![p_1 \vee p_2]\!]^p = [\![p_1]\!]^p \bigvee [\![p_2]\!]^p}$$

$$\frac{}{\Gamma_S, \Gamma_e, \mathcal{V} \vdash [\![p_1 \wedge p_2]\!]^p = [\![p_1]\!]^p \bigwedge [\![p_2]\!]^p}$$

$$\frac{}{\Gamma_S, \Gamma_e, \mathcal{V} \vdash [\![p_1 \Rightarrow p_2]\!]^p = [\![p_1]\!]^p \;\texttt{->}\; [\![p_2]\!]^p}$$

$$\frac{}{\Gamma_S, \Gamma_e, \mathcal{V} \vdash [\![\textbf{all}\; x \textbf{ : } \tau, p]\!]^p = \texttt{forall}\; x \;\texttt{:}\; [\![\tau]\!]^t \texttt{,}\; [\![p]\!]^p}$$

$$\frac{}{\Gamma_S, \Gamma_e, \mathcal{V} \vdash [\![\textbf{ex}\; x \textbf{ : } \tau, p]\!]^p = \texttt{exists}\; x \;\texttt{:}\; [\![\tau]\!]^t \texttt{,}\; [\![p]\!]^p} \qquad \diamond$$

Definition 7.17. *Translation of Types*

For types, the same particularity as for expressions arises. The translation of the representations depends on the context: are they collection parameters or toplevel collections? This changes the way to access the type constructor denoting the representation. We also omit the case where the toplevel collection belongs to another compilation unit, in which case the name of this unit should be added as prefix. Note that the type `prop` is ultimately translated into the type `Prop` of Coq. However, this may only be required in case of **`logical let`** signatures, the only place where the user may mention this type, since it is impossible for him/her to create values of type **`prop`**.

$$\mathcal{V} \vdash [\![\texttt{a}]\!]^t = \texttt{a} \qquad \frac{\mathcal{V} \vdash \forall i, [\![\tau_i]\!]^t = t_i}{\mathcal{V} \vdash [\![\texttt{p}(\overrightarrow{\tau_i})]\!]^t = \texttt{p}\,\texttt{(}\,\overrightarrow{t_i}\,\texttt{)}} \qquad \mathcal{V} \vdash [\![\alpha]\!]^t = \mathcal{V}(\alpha)$$

$$\frac{\mathcal{V} \vdash {[\![\tau]\!]^t}_1 = t_1 \qquad \mathcal{V} \vdash {[\![\tau]\!]^t}_2 = t_2}{\mathcal{V} \vdash [\![\tau_1 \to \tau_2]\!]^t = t_1 \;\texttt{->}\; t_2} \qquad \frac{\texttt{C} \text{ is a toplevel collection}}{\mathcal{V} \vdash [\![\texttt{C}]\!]^t = \texttt{C.me\_as\_carrier}}$$

$$\frac{\texttt{C} \text{ is a collection parameter}}{\mathcal{V} \vdash [\![\texttt{C}]\!]^t = \texttt{C\_T}} \qquad \mathcal{V} \vdash [\![\texttt{Self}]\!]^t = \texttt{abst\_T} \qquad \diamond$$

Despite this precise description of the compilation, several precise technical details have still be untold, like the following examples.

As it would need a deep presentation of Zenon we do not detail the compilation of proofs and the interaction with Zenon. Technically, it rests upon the dependency calculus and all the needed material is already computed for the core of the code generation presented here. The idea is to embed proofs steps invoking Zenon into Coq `Sections` to transform the λ-liftings into `Parameters` to stay with first-order formulae manipulated by Zenon. The compiler must generate the prelude of the section and its postlude to restore the λ-liftings. It must also give to Zenon the definitions which are used as Zenon hints.

The λ-lifting of dependencies requires a special care when two collection parameters get instantiated by a same collection. Weak polymorphic type variables must be handled slightly differently from polymorphic ones.

The compilation relies on different environments between passes to avoid computing several times some same or similar information: this requires slight modifications in a few rules (mostly substitutions in methods for the code generation environment in order to explicitly provide arguments for record accesses instead of placeholders).

The dependencies are computed once for all the target languages. To ensure that dependencies coming from logical methods do not flow into the computational code generated, dependencies must be tagged to remind their provenance.

On a purely practical point of view, the user's identifiers must not conflict with the keywords of the target languages, which is ensured by a convenient name mangling.

8. Discussion

The requirements and choices discussed in sections 2 and 5 were considered at the early beginning of the FoCaLiZe project. First, the intended semantics was specified in Coq by S. Boulmé [34]. It revealed the need of a careful treatment of dependencies. Then, a first prototype of the language, developed by V. Prevosto [25], designed the first concrete syntax and the first FoCaL compiler, whose correctness against FoCaL's semantics was proved (by hand) [40]. After these pioneer studies, the compiler has been fully redesigned and extended to lead to the current FoCaLiZe system with a common code generation model, by the author. The interface with the Zenon automated theorem prover, already present in the first compiler, has also been extended a lot.

λ-lifting Parameters Instead of Methods. We considered also another possibility to obtain a collection generator. Instead of λ-lifting one by one the parameter method names from the record of collection generators, we could have lifted only the collection parameters names. Then, a collection generator should be directly applied to an effective collection. But typing this application needs to attribute a record type to collection parameters, so to collections and then to species. Since species define a form of subtyping (allowing a collection with more methods than expected to instantiate a collection parameter) we would need a kind of subtyping relation on record types. Although this feature exists in some languages, we prefer to require the minimum of constructs for target languages, hence leaving our code generation model open to a wider range of potential targets.

Theorems Versus Axioms. In the Coq model of a collection, since theorems have been proved should they remain theorems (choice 1) or should they be added as axioms (choice 2)?

The major drawback of choice 2 is that instead of containing a complete Coq source file, any FoCaLiZe compilation unit should be split into several intermediate Coq files, incrementally proving one theorem in the context where previously proved ones are replaced by axioms. For instance, instead of having Coq verifying one complete model:

```
Theorem t1 : ... proof = ...
Theorem t2 : ... proof = ... t1 ...
Theorem t3 : ... proof = ... t1, t2 ...
```

one should split this model in three parts:

```
Theorem t1 : ... proof = ...
```

```
Axiom t1 : ...
Theorem t2 : ... proof = ... t1 ...
```

```
Axiom t1 : ...
Axiom t2 : ...
Theorem t3 : ... proof = ... t1, t2 ...
```

This would clearly seriously degrade readability, traceability and would require for each compilation unit to analyze several Coq files instead of one.

Using functors. An orthogonal model is also possible, keeping method generators but avoiding collection generators. Since a species is represented by a module, it may look convenient to represent parametrized species by functors and inheritance by module inclusion. This presents two major drawbacks. First, notions of functor and module inclusion are required in the target languages, hence may restrict target candidates. Second, even if the inheritance mechanism could be (possibly) simplified, we anyway still need to have it explicitly resolved by the compiler, at least for typechecking, normal form computation and automatic documentation generation. Hence, this intuitive idea of simplifying by using functors would indeed lead to more work than the one already done by the compiler.

Code Extraction. Since FoCaLiZe generates Coq and OCaml code, one may wonder why not directly develop in Coq and use the code extraction mechanism to get executable code.

First, the high-level modeling structures provided by FoCaLiZe are complex to manually express in Coq. It is just the work of the compiler to do this job: checking well-formedness of the components, resolving inheritance, early-binding and parameterization, generating the effective code, interfacing with the prover(s). Nobody would appreciate doing this job by hand.

Second, we decided not to impose an extraction mechanism on the target logical language in order to minimize requirements on it. Moreover, one can imagine to also emit computational code toward other languages than ML, which would require the logical target language to also have an extraction mechanism toward several languages. For instance, a back-end toward Dedukti [41, 42] is currently in progress, taking benefit from some evolutions of Zenon to handle deduction modulo [43].

External Code. Aside species and collections which serve to make an incremental development, some primitive notions like booleans, integers, etc. are needed and given by the standard library of the language. The library can also provide proved properties on these primitives. It is possible to call FoCaLiZe collections methods from OCaml code, being aware of their proved properties. Conversely, external OCaml or Coq libraries can also be accessed via a binding mechanism. Their properties can be granted by other means (contracts, verification tools, etc). If no formal proof evidence can be issued for the properties of these external libraries, it is still possible to assume them as holding, giving them a proof reduced to the keyword **`assumed`**.

This is a needed but dangerous feature as admitted properties can lead to logical inconsistencies. Depending on the context of the system, some “only nearly true” but “well-known theorems” may be safely considered as holding to simplify the problem and directly deal with “more functional” properties. For instance, $\forall x \in N, x + 1 > x$ doesn’t hold since arithmetic is modulo the number of bits of machine integers, however some practical considerations can guarantee that no overflow will occur [44].

The use of this keyword is recorded in the automatically done documentation file provided by the compilation process. The assessment process must ultimately check that any occurrence of this keyword is harmless.

Contracts. Using contracts to take benefits from external code can be done in two ways. First, the contract may simply be blindly assumed as holding. In this case its property must be artificially proved using the keyword **`assumed`** as presented in the previous paragraph.

The second solution is to obtain a real proof, i.e. a λ-term of the target logical language, then embed it in the global logical model created during the compilation. Doing this way, the logical verifier is also able to check the proofs of the contracts. This implies that the contract has already been formally verified externally (for instance with a WP tool). The aim is then to rebuild a complete proof from the proved pre/post-conditions. However, it is not yet clear if a deep embedding of the WP logic (and the foreign language) is unavoidable.

Proofs Writing. To write proofs, another solution could be to use the syntax and the tools of the chosen prover. We tried the first choice (it is still possible in FoCaLiZe to insert Coq proofs) and found several drawbacks.

- The user must have a deep knowledge of the prover to powerfully use it and to understand error messages.
- The user must be aware of the mapping done by the compiler between the concrete language and the target logical one.
- The proofs too deeply depend on this target logical language. Indeed, the proof format should allow translation of proofs to several theorem provers (based on type theory).

Recursive Functions. Termination proofs for recursive functions have recently been integrated in FoCaLiZe [39]. The user can provide a measure (function returning an integer) or a well-founded relation. In both cases, the compiler automatically generates proof obligations that must be proved by the user with the usual proof style (i.e. helped by Zenon). For a measure, the core of the obligations requires a strict decreasing of the measure at each recursive call. For a well-founded relation, the argument of the recursive calls must be smaller than the initial argument according to this relation. In both cases, the user has to write proofs only taking care of the recursive parameter, the compiler ensures the transformation of the obligations and their proofs to cope with all the parameters of the function.

Pattern-Matching. The pattern-matching must still be handled in a finer way to hide the facts that Zenon only handles head-most patterns and that Coq requires expansion of nested patterns.

9. Related Work

Our choice was to consider a pure functional language, leaving side effects outside the model. Other tools made the choice of directly addressing an imperative language.

Spec♯ [45] is such a language, a subset of C♯ with an stronger static typing and a pre/post mechanism to specify properties of the manipulated entities. The compiler adds runtime verifications for contracts. The language is compiled toward Boogie [46] which serves as an intermediate verification language. This latter is able to perform loop-invariants inference and uses the Spec♯ program annotations and type properties to try to statically prove that the inserted runtime checks are always satisfied (hence useless). Dafny [3] follows the same scheme, also compiling toward Boogie, but extends the language with built-in sets and sequences, algebraic datatypes and termination metrics. Differently from us where termination proofs are written like the others, termination is checked thanks to an annotation (lexicographic tuple). In both cases, proofs are also done using an external theorem prover [47]. However, no logical term is produced as witness.

Why3 [7] relies on WhyML, a language with functional features similar to FoCaLiZe but also providing references, loops and exceptions. It does not have inheritance and parametrization but a system of theories allowing modularity. Like for Spec♯ and Dafny, the approach is different from ours, not only by the use of Hoare logic required to handle imperative programs. Proofs are done from pre/post-conditions and invariants given by the programmer and not from "lemmas" written outside the implementation of the functions. Wh3 is able to generate proof obligations and submit them to a large number of external provers, much more than FoCaLiZe currently does. No witness term is either produced.

Unfortunately, as far as we know, there is no available details on the implementation of B. Even if the logical framework differs from ours, it is clear that refinement and abstract machines have common points with our inheritance and parametrization, hence may induce questions similar to those presented in this paper.

In [48] D. Leijen presents a type system with effects to separate functional and imperative components of a program. Such a technique may be of interest since we also rely on an ML-like type system. This would not solve proof writing for imperative parts of a program but this could detect and forbid unfolding functions with side effects in our proofs.

It is clear that our work does not target the compilation of general object languages. We compile toward high-level target languages and not native code. Hence we do not care of register allocation, memory allocation, etc. Also, our object-oriented features are static (no dynamic dispatch). For these reasons, it is difficult to compare our compilation scheme with those of these languages.

Finally, our approach does not aim at proving the compiler unlike, for example, [49, 50]. The generated code is submitted to an external verifier (Coq). The guarantee of correctness for the computational target code relies on the common trunk of the compilation, as long as the logical generated code is accepted by Coq.

10. Conclusion and Future Work

The main objective of this paper is the presentation of the compiler which aims rather differ from those of most compilers of programming languages. This compiler produces both logical and computational codes tightly linked together. We try to give a very detailed, and as formal as possible, presentation for several reasons.

First, we consider important to explain the way the compiler produces error-free Coq and OCaml codes, without requiring the user to manipulate these languages. Errors are handled at the source language level. The proofs are done by the user, helped by Zenon, and Coq is used as both an "assembly" formal language and a guarantor which double-checks the complete development, including proofs. Both target codes are very close to each other, as they are emitted from the same intermediate form, by only adapting the syntax. The OCaml code is mostly an erasing of the logical parts of the Coq code. This closeness eases the assessment process when specifications and code must be compared.

Second, there are few papers on compilers with similar target languages. The system B [8] generates proof obligations and C/ADA code. DAFNY [3, 4] allows building executable or DLL libraries. It features a language with dynamic allocation and generic classes. WHY3 [5, 6, 7] is able to generate OCaml code, generating proof obligations and submitting them to various external provers. These languages allow imperative programming and Hoare logic styles. The user states his/her specifications using pre/-post conditions, and more generally, invariants. Their goal is not to issue a complete formal model of the generated code to submit it to a prover.

Third, reading the source code of a compiler is generally not the good way to reuse some of its features. We hope that this presentation can help other projects with similar needs, and we write it with this aim in mind. For instance, developments with strong safety/security concerns can ask for Domain Specific Languages issuing proofs of conformance. All these details may also be enjoyed by anybody wishing to extend FoCaLiZe.

To justify such a compilation process and the induced difficulties, a preliminary presentation of the design choices was needed. These choices come from the will to avoid dissociating the computational and logical aspects in a formal development, while keeping the language palatable. The mix of inheritance, early-binding, encapsulation, inductive data types and unfolding of definitions in proofs creates complex dependencies, which have to be analyzed to ensure programs correctness. This analysis gives the basis of the compilation process through the notions of method and collection generators, that we introduced to maximize code sharing. These choices brought the main difficulties of the compilation process.

Some work remains to be done in order to enhance FoCaLiZe and its compiler. The Zenon prover greatly contributes to the proof automation. Some works are currently extending it to handle arithmetic, and other provers than Coq. This may allow to experiment FoCaLiZe with other target type theories and other kinds of properties.

Temporal properties (*à la* "always", "never", "finally" ...) cannot currently be expressed in FoCaLiZe. This must be a point to address since many programs requiring a high level of confidence are embedded, real-time ones and require such properties to be proved. Zenon is able to handle TLA formulae, hence this direction has to be investigated.

Acknowledgments

I first want to strongly thank Thérèse Hardin, who greatly contributed to enhance this paper. I also want to thank people who gave me their comments (and also strongly participated to the FoCaLiZe adventure), David Delahaye, Damien Doligez, Catherine Dubois, Mathieu Jaume, Virgile Prevosto, Renaud Rioboo, Pierre Weis.

[1] J. Spivey, An introduction to Z and formal specifications, Softw. Eng. J. 4 (1).

[2] Maude, http://maude.cs.uiuc.edu/ (2015).

[3] K. M. Leino, Dafny: An automatic program verifier for functional correctness, in: Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 348–370.

[4] K. M. Leino, Specification and verification of object-oriented software, in: In Marktoberdorf International Summer School 2008, Lecture Notes, 2008.

[5] Why3, http://why3.lri.fr/ (2015).

[6] F. Bobot, J. Filliâtre, C. Marché, A. Paskevich, Why3: Shepherd your herd of provers, in: Boogie 2011: First International Workshop on Intermediate Verification Languages, Wrocław, Poland, 2011, pp. 53–64.

[7] J. Filliâtre, A. Paskevich, Why3 – Where Programs Meet Provers, in: ESOP'13 22nd European Symposium on Programming, Vol. 7792, Springer, Rome, Italy, 2013.

[8] Atelier-B, http://www.atelierb.eu/ (2015).

[9] Isabelle Team, The Isabelle/Isar reference manual (2015).
URL http://www.cl.cam.ac.uk/research/hvg/Isabelle/

[10] L. Paulson, The foundation of a generic theorem prover, Tech. Rep. UCAM-CL-TR-130, University of Cambridge, Computer Laboratory (Mar. 1988).

[11] T. Nipkow, L. Paulson, M. Wenzel, Isabelle/HOL – A Proof Assistant for Higher-Order Logic, Vol. 2283 of LNCS, Springer, 2002.

[12] R. L. Constable, T. Knoblock, J. L. Bates, Writing programs that construct proofs, Journal of Automated Reasoning 1 (3) (1984) 285–326.

[13] S. Owre, J. Rushby, N. Shankar, D. Stringer-Calvert, PVS: an experience report, in: D. Hutter, W. Stephan, P. Traverso, M. Ullman (Eds.), Applied Formal Methods—FM-Trends 98, Vol. 1641 of Lecture Notes in Computer Science, Springer-Verlag, Boppard, Germany, 1998, pp. 338–345.

[14] S. Owre, N. Shankar, Writing PVS proof strategies, in: M. Archer, B. D. Vito, C. Muñoz (Eds.), Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003), no. CP-2003-212448 in NASA Conference Publication, NASA Langley Research Center, Hampton, VA, 2003, pp. 1–15.

[15] Coq Team, The Coq proof assistant reference manual, LogiCal Project, version 8.4 (2015).
URL http://coq.inria.fr

[16] S. Blazy, X. Leroy, Formal verification of a memory model for C-like imperative languages, in: International Conference on Formal Engineering Methods (ICFEM 2005), Vol. 3785 of Lecture Notes in Computer Science, Springer, 2005, pp. 280–299.

[17] S. Blazy, Z. Dargaye, X. Leroy, Formal verification of a C compiler front-end, in: FM 2006: Int. Symp. on Formal Methods, Vol. 4085 of Lecture Notes in Computer Science, Springer, 2006, pp. 460–475.

[18] X. Leroy, Formal certification of a compiler back-end, or: programming a compiler with a proof assistant, in: 33rd ACM symposium on Principles of Programming Languages, ACM Press, 2006, pp. 42–54.

[19] J. Etienne, M. Maarek, F. Anseaume, V. Delebarre, Improving predictability, efficiency and trust of model-based proof activity, in: Proceedings of the 37th International Conference on Software Engineering - Volume 2, ICSE '15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 139–148.

[20] FoCaLiZe, http://focalize.inria.fr/ (2015).

[21] IEC-61508-3, Functional safety of electrical/electronic/programmable electronic safety-related systems, International Electrotechnical Commission, 2011.

[22] Common Criteria, Common Criteria for Information Technology Security Evaluation, Norme ISO/IEC 15408 – Version 3.1 Rev 4 (2012).

[23] L. Habib, M. Jaume, C. Morisset, Formal definition and comparison of access control models, Journal of Information Assurance and Security (JIAS) 4 (4) (2009) 372–381.

[24] D. Delahaye, J. Étienne, V. Viguié, Certifying Airport Security Regulations using the Focal Environment, in: FM06, Vol. 4085 of LNCS, Springer-Verlag, 2006, pp. 48–63.

[25] V. Prevosto, Conception et implantation du langage FoC pour le développement de logiciels certifiés, Ph.D. thesis, Université Paris 6 (sep 2003).

[26] Lafosec, Sécurité et langages fonctionnels, https://www.ssi.gouv.fr/publication/lafosec-securite-et-langages-fonctionnels/ (2013).

[27] P. Ayrault, T. Hardin, F. Pessaux, Development of a generic voter under focal, in: TAP'09, Vol. 5608 of LNCS, Springer-Verlag, 2009, pp. 10–26.

[28] L. Lamport, How to write a proof, Research report, Digital Equipment Corporation (1993).

[29] Zenon, http://zenon-prover.org/ (2015).

[30] P. Ayrault, T. Hardin, F. Pessaux, Development life-cycle of critical software under focal, Electron. Notes Theor. Comput. Sci. 243 (2009) 15–31.

[31] V. Prevosto, M. Jaume, Making proofs in a hierarchy of mathematical structures, in: Proceedings of the 11^{th} Calculemus Symposium, 2003.

[32] T. Hardin, R. Rioboo, Les objets des mathématiques, RSTI - L'objet.

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

[34] S. Boulmé, Spécification d'un environnement dédié à la programmation certifiée de bibliothèques de calcul formel, Thèse de doctorat, Université Paris 6 (2000).

[35] L. Damas, R. Milner, Principal type-schemes for functional programs, in: POPL'82, ACM, 1982, pp. 207–212.

[36] R. Milner, A theory of type polymorphism in programming, J. Comput. Syst. Sci. 17 (3) (1978) 348–375.

[37] M. Tofte, Operational semantics and polymorphic type inference, Thèse de doctorat CST-52-88, University of Edinburgh (1988).

[38] J. Robinson, A machine-oriented logic based on the resolution principle, J. ACM 12 (1) (1965) 23–41.

[39] C. Dubois, F. Pessaux, Termination Proofs in FoCaLiZe, in: Draft Proceedings of the 2015 Symposium on Trends in Functional Programming, 2015.

[40] V. Prevosto, S. Boulmé, Proof contexts with late binding, in: Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005, 2005, pp. 324–338.

[41] M. Boespflug, Design and implementation of a proof verifying kernel for the $\lambda\Pi$-calculus modulo, Theses, Ecole Polytechnique X (Jan. 2011).

[42] M. Boespflug, Q. Carbonneaux, O. Hermant, The lambda-pi-calculus modulo as a universal proof language, in: Proceedings of the Second International Workshop on Proof Exchange for Theorem Proving, 2012.

[43] R. Cauderlier, P. Halmagrand, Checking Zenon Modulo Proofs in Dedukti, in: Fourth Workshop on Proof eXchange for Theorem Proving (PxTP), Berlin, Germany, 2015.

[44] M. Clochard, J.-C. Filliâtre, A. Paskevich, How to avoid proving the absence of integer overflows, in: A. Gurfinkel, S. A. Seshia (Eds.), 7th Working Conference on Verified Software: Theories, Tools, and Experiments, 7th Working Conference on Verified Software: Theories, Tools, and Experiments, San Francisco, CA, United States, 2015.

[45] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, K. R. M. Leino, Boogie: A modular reusable verifier for object-oriented programs, in: Proceedings of the 4th International Conference on Formal Methods for Components and Objects, FMCO'05, Springer-Verlag, Berlin, Heidelberg, 2006, pp. 364–387. doi:10.1007/11804192_17.

[46] K. R. M. Leino, This is boogie 2, Tech. Rep. MSR-TR-2008-194 (June 2008).

[47] L. De Moura, N. Bjørner, Z3: An efficient smt solver, in: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 337–340.

[48] D. Leijen, Koka: Programming with row-polymorphic effect types, Tech. Rep. MSR-TR-2013-79 (August 2013).

[49] A. Chlipala, A certified type-preserving compiler from lambda calculus to assembly language, in: Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007, 2007, pp. 54–65. doi:10.1145/1250734.1250742.

[50] X. Leroy, A formally verified compiler back-end, J. Autom. Reason. 43 (4) (2009) 363–446. doi:10.1007/s10817-009-9155-4.