FoCaLiZe

Programming and Proving

A Bit Under the Hood

François Pessaux - ENSTA ParisTech (U2IS)

francois.pessaux@ensta-paristech.fr

DEDUCTEAM Seminar
11 April 2014

Topics and Short Outline

- `FoCaLiZe`: a language to express **code**, **properties** and **formal proofs**.
- Outline:
  - Short presentation of `FoCaLiZe`,
  - How **design & features** choices drive the **semantics** and the **compilation model**,
  - Sketch of compilation scheme focusing on **dependencies**.
  - … Dependency analysis rules in spare just in case … ☺

Started more than 10 years ago (T. Hardin and R. Rioboo) …

FoCaLiZe *Credo*

- Why ?
  - Standards require usage of formal methods to ensure high level assurance of critical systems.
  - Formal methods ? Runtime verification, UML … For us: mechanically checked proofs.
  - Ideally should be within any computer science engineer skills: our long term goal.

- How ?
  - Basis: wedding `OCaml` and `Coq` **avoiding** too complex features.
  - Features **mixing logical and programming** aspects: inheritance, late-binding, abstraction, parametrisation, **properties** and **proofs**.
  - Mixing **computational**/**logical** features: risk of **inconsistencies** (S. Boulmé PhD) .
  - Our claim: **Accepted** by `FoCaLize` compiler ⇒ **No** `OCaml` or `Coq` error!

`FoCaL`: first compiler by V. Prevosto … `FoCaLize`: Darwinian evolution

Semantical Framework

- **Requirements** / **implementation**: a single language and a single semantics for **logical** / **programming** features.

- **Pure functional** declarations and definitions, **first-order** (like) formulae, proofs written in *FPL*.
- Properties can use function names only, proofs can unfold function definitions **not the inverse**.
- Thus a kind of **dependent type theory**, however some **dependencies** are forbidden: don't want/need the whole Coq's power

- `FoCaLiZe` source: compiled to `OCaml` and `Coq` **source** files.
- Proofs sent to `Zenon` returning a `Coq` **term** to **embed** in final Coq source.
- **Curry-Howard** isomorphism. Logical aspects **discarded** in `OCaml`.

Species

- Structure grouping **signatures**, **properties**, **functions** and **proofs** related to an underlying data-type: the `representation`.

```
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 ;;
```

- **Inheritance**: to enhance **reusability**.
- **Late-binding**: introduces a **name** and a **type**, deferring definition (`representation` also).
- Allows to **incrementally** introduce new items.
- Progression from a **specification** to **implementation**.
- At each step: use new items to prove **conformance** with **previously** stated **requirements**.

Parameterization

- Parameterized module ? We need **parameterized species**.
- Two kinds of parameters:
  - Use **methods** & **properties** of other species: **collection parameter**.
  - Use **values** of other species: **entity parameter**.

```
species IsIn (V is OrdData, minv in V, maxv in V) =
 representation = (V * statut_t) ;
 let filter (x) : Self =
   if V!lt (x, minv) then (minv, Too_low)
   else if V!gt (x, maxv) then (maxv, Too_high) ... ;
 theorem lowMin: all x: V,
   getStatus (filter (x)) = Too_low -> ~ V!gt(x, minv)
 proof = ... ;
```

Abstracted or not (to be) Abstracted

- Definition of `representation` exposed or encapsulated ?
  - **Inheritance** & **late-binding** require **exposure**.
  - Parameterization requires **abstraction**.

➡ Visibility driven by 2 structures:
  - Species: total **transparency** of **definitions**.
  - Collection: representation **abstracted**, only **types** (hence also **properties**) visible.

Collection

- To provide **effective arguments** to collection parameters.
- No link-time errors ➡ all exported **functions** must be **defined**.
- No inconsistencies ➡ all **properties** must be **proved**.
- Abstracted « instance » of a **complete** species.
- The **only** form of **proved run-able** code.

```
species TheInt =
 inherit OrdData ;
 ... (* Complete species. *)
end ;;
collection IntC = implement TheInt ; end ;;
collection In_5_10 =
  implement IsIn (IntC, IntC!fromInt (5), IntC!fromInt (10)) ;
end ;;
```

Properties and Proofs

- Be **independent** from any particular proof checker.
- **Own** proof language, **natural deduction** style.
- **Proof** = **hierarchical** decomposition into intermediate steps introducing **subgoals** and **assumptions**.
- **Leaf**: **subgoal** which can be **automatically** handled by **Zenon** automated prover using **facts** given by the **user**.

```
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
```

- **Zenon** returns a **Coq term** plugged by the compiler in the context.
- **Only** acceptable Zenon errors: « *out of memory* », « *time out* », « *no proof found* ».

Outline of Coming Technical Points

Reminders about FoCaLiZe ended!

Coming next…

- **Dependencies** on **own** species methods
- **Dependencies** on **collection parameters** methods
- Code generation: **method generators**
- Code generation: **collection generators**
- Initial work: V. **Prevosto** dependency analysis, rules modified and extended.

Notion of Dependencies (1/3)

- A method **depending** on the **definition** of `m` has a **def-**dependency on `m`.
- Only two possible def-dependencies:
  - **Proof** with a **`by definition`** `of m` (unfolds the definition of `m`)

    ➡ If `m` redefined, proof must be invalidated.
  - **Functions** and **proofs** can def-depend on the **`representation`**.
- By **syntax**, functions cannot def-depend on proofs.
- By **encapsulation**, no possible def-dependencies on parameters methods.
- Analysis required to prevent def-dependencies on the **`representation`** in **properties** and theorems **statements**.

```
species Sample =
  representation = bool ;
  signature decldep_on_me : Self -> int;
  property things_hold: all x : int, bla (i) ;
  let defdep_on_me (x : Self) = … if (x) decldep_on_me (x) else … ;
  theorem prove_me: all x : Self, all i : int, bla (i) \/ defdep_on_me (x) = i
    proof = by definition of defdep_on_me property things_hold ;
end ;;
```

Notion of Dependencies (2/3)

- A method **depending** on the **definition** of `m` has a **def-**dependency on `m`.
- Only two possible def-dependencies:
  - **Proof** with a **`by definition`** `of m` (unfolds the definition of `m`)

    ➡ If `m` redefined, proof must be invalidated.
  - **Functions** and **proofs** can def-depend on the **`representation`**.
- By **syntax**, functions cannot def-depend on proofs.
- By **encapsulation**, no possible def-dependencies on parameters methods.
- Analysis required to prevent def-depend on the **`representation`** in **properties** and theorems **statements**.

```
species Sample =
  representation = bool ;
  signature decldep_on_me : Self -> int;
  property things_hold: all x : int, bla (i) ;
  let defdep_on_me (x : Self) = … if (x) decldep_on_me (x) else … ;
  theorem prove_me: all x : Self, all i : int, bla (i) \/ defdep_on_me (x) = i
    proof = by definition of defdep_on_me property things_hold ;
end ;;
```

Notion of Dependencies (3/3)

- Method **depending** on the **declaration** of `m` has a **decl-dependency** on `m`.
- **Decl-**dependencies: a matter of **typechecking**.

```
species Sample =
  representation = bool ;
  signature decldep_on_me : Self -> int;
  property things_hold: all x : int, bla (i) ;
  let defdep_on_me (x : Self) = … if (x) decldep_on_me (x) else … ;
  theorem prove_me: all x : Self, all i : int, bla (i) \/ defdep_on_me (x) = i
    proof = by definition of defdep_on_me property things_hold ;
end ;;
```

- Dependencies: the **key** to ensure **no** `OCaml/Coq` **errors**!

Finding Dependencies on Methods of *Self*

- **Cyclic** dependencies only allowed between (mutually) **recursive** functions.
- Through **proofs**, **def-**dependencies force keeping **definitions** in the context to be typecheck-able (fact by `definition of`).

➡ These definitions **themselves** have to be **typecheck-able**.

- Through **proofs**, **decl-**dependencies on **logical** methods (expressions).

➡ Methods in such « ***types*** » **also** have to **typecheck-able**.

```
property ltNotGt: all x y: Self, lt (x, y) -> ~gt (x, y) ;
```

Coq ⇒

```
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".
```

- Keep methods $\in$ **transitive closure** of the **def-**dependency relation + methods on which these latter **decl-**depend: the **visible universe**.

Visible Universe

$$\frac{y \in \wr x \wr_S}{y \in | x |} \qquad \frac{y <_S^{def} x}{y \in | x |}$$

$$\frac{z <_S^{def} x \qquad y \in \wr z \wr_S}{y \in | x |} \qquad \frac{z \in | x | \qquad y \in \wr \mathcal{T}_S(z) \wr_S}{y \in | x |}$$

- $x <_S^{def} y$: « y **def-**depends on x by transitivity »
- $\mathcal{T}_S(x)$: « the **type** of x in the species S ».

Minimal Typing Environment

$$\varnothing \Cap x = \varnothing \qquad \frac{y \notin |\, x \,| \qquad \{y_i : \tau_i = e_i\} \Cap x = \Sigma}{\{y : \tau = e \;;\; y_i : \tau_i = e_i\} \Cap x = \Sigma}$$

$$\frac{y \in |\, x \,| \qquad y <_S^{def} x \qquad \{y_i : \tau_i = e_i\} \Cap x = \Sigma}{\{y : \tau = e \;;\; y_i : \tau_i = e_i\} \Cap x = \{y : \tau = e \;;\; \Sigma\}}$$

$$\frac{y \in |\, x \,| \qquad y \not<_S^{def} x \qquad \{y_i : \tau_i = e_i\} \Cap x = \Sigma}{\{y : \tau = e \;;\; y_i : \tau_i = e_i\} \Cap x = \{y : \tau \;;\; \Sigma\}}$$

- Methods $\notin$ visible universe: **not** required.
- Methods $\in$ visible universe on which x **doesn't def-**depend: only their type required.
- Methods $\in$ visible universe on which x **def-**depends: their **type and body** required.

Dependencies Summary

- type t (‘a) = …
- … (S * int) …
- all x : t (int), y : S, f (x, S) …

- by type definition of …
- On the representation:
  <2>1 assume x : Self, prove x = 0

Peut dépendre de	Type	Preuve	Définition
Type	✓		
Preuve	✓		✓
Définition	✓		✓

- by type u
- all x : t (int), f (x) …
- by property …

- let f (x : S) = …
- let g (x : Self) = …

- On the representation:
  let h (x : Self) = if x …

Dependencies on Methods of Collection Parameters

- Similar problem than methods of `Self`: track dependencies on **collection parameters methods**.

```
theorem too_low_not_gt_min:
  all x : V, get_status (filter (x)) = Too_low -> ~ V!gt (x, minv)
  proof = <…> … bla … prove ~ V!gt (x, minv) … property V!lt_not_gt … ;
```

Coq ⇒

```
Theorem too_low_not_gt_min  (_p_V_T : Set) (_p_V_lt : _p_V_T -> _p_V_T -> basics.bool__t)
  (_p_V_gt : _p_V_T -> _p_V_T -> basics.bool__t)
  (_p_V_lt_not_gt : 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)%type))
  (abst_filter := filter _p_V_T _p_V_lt _p_V_gt _p_minv_minv _p_maxv_maxv) … := … ;
```

- Again, AST traversal is **not** sufficient.
- Consider there are dependencies on all the methods of all the collection parameters?

  ➡ Cumbersome, unreadable, inefficient!
- Challenge: find the **minimal set** of required methods.

Computing Deps on Methods of Collection Parameters

- Four kinds of rules, collecting dependencies a method as on a parameter method…
  - (2) **explicitly** stated in the body (resp. type) of a definition,
  - (2) induced by the dependencies **the** method has inside **its** hosting species (for decl and def),
  - (1) because this parameter is used as **effective argument** to build the **current** parameter,
  - (1) due to decl-dependencies that methods **of parameters** have inside their **own** species and that are visible through **types**.
- Entity parameters: no extra dependencies since no methods. Are « *themselves the dependency* ».

Rules for Deps. on Parameters Methods (1/4)

$$\mathcal{D}o\mathcal{P}_{[\textsc{Body}]}(S, C)[x] = \mathcal{D}o\mathcal{P}_{[\textsc{Expr}]}(S, C)[\mathcal{B}_S(x)]$$

$$\mathcal{D}o\mathcal{P}_{[\textsc{Type}]}(S, C)[x] = \mathcal{D}o\mathcal{P}_{[\textsc{Expr}]}(S, C)[\mathcal{T}_S(x)]$$

- `[Body]`: harvest dependencies on a method **explicitly stated** in the **body** of a definition.
- `[Type]`: harvest dependencies on a method **explicitly stated** in the **type** of a definition.

Rules for Deps. on Parameters Methods (2/4)

$$\mathcal{D}o\mathcal{P}_{[\textsc{Def}]}(S,C)[x] = \mathcal{D}o\mathcal{P}_{[\textsc{Expr}]}(S,C)[\mathcal{B}_S(z)] \qquad \text{for all } z \text{ such as } z <_S^{def} x$$

$$\mathcal{D}o\mathcal{P}_{[\textsc{Univ}]}(S,C)[x] = \mathcal{D}o\mathcal{P}_{[\textsc{Expr}]}(S,C)[\mathcal{T}_S(z)] \qquad \text{for all } z \text{ such as } z \in \mid x \mid$$

- `[Def]` and `[Univ]`: collect dependencies of **a method** on a parameter induced by the dependencies **this** method has in **its hosting** species.
- Note: methods z introduced by `[Def]` included in those introduced by `[Univ]` (*vis. univ. wider than only transitive def-deps and their related decl-deps*).

Rules for Deps. on Parameters Methods (3/4)

$$\dfrac{\begin{array}{c}\mathcal{E}(S) = (\ldots, C_p \ \textbf{is}\ ^{\cdots}, \ldots, C_{p'} \ \textbf{is}\ S'(\ldots, C_p, \ldots)) \\ \mathcal{E}(S') = (\ldots, C'_k \ \ \textbf{is} \ \ I'_k, \ldots) \\ z \in \mathcal{D}o\mathcal{P}_{[\textsc{Type}]}(S, C_{p'})[x] \ \vee \ z \in \mathcal{D}o\mathcal{P}_{[\textsc{Body}]}(S, C_{p'})[x] \\ (y : \tau_y) \in \mathcal{D}o\mathcal{P}_{[\textsc{Type}]}(S', C'_k)[z]\end{array}}{(y : \tau_y[C'_k \leftarrow C_p]) \in \mathcal{D}o\mathcal{P}_{[\textsc{Prm}]}(S, C_p)[x]}$$

- Harvest dependencies of a method on a **previous parameter C_p used as argument** to build the **current** parameter $C_{p'}$.
- Difference with previous rules: result is **not only** a set of names: **types** are **explicit**.

  Because type of the methods of this set differs from the one computed during typechecking of the species used as parameter.

Rules for Deps. on Parameters Methods (4/4)

$$\frac{\mathcal{E}(S) = (\ldots, C_p \textbf{ is } I_p, \ldots) \qquad z \in \mathcal{D}(S, C_p)[x] \qquad (y : \tau_y) \in \wr \mathcal{T}_{I_p}(z) \int_{I_p}}{(y : \tau_y[\texttt{Self} \leftarrow C_p]) \in \mathcal{D}^+(\mathcal{D}, S, C_p)[x]} \text{ Close}$$

- Take into account **decl-**dependencies that methods of parameters have inside **their own species** and that are visible through **types**.

```
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 ;;
```

Code Generation: Method Generators

- Starts after resolution of inheritance and late-binding, typing and dependency analysis.
- For **traceability** and **assessment**: **common** code generation model `OCaml` / `Coq`.
- Generate code for **only** collection? ➡ **no** code **sharing**.
- Want to share methods bodies: reduces code **size** and assessment **duration**.
- Method `m`: when defined ➡ emit its **method generator**:
  - compiled version of `m`'s body,
  - methods `m` **decl-**depends on are **λ-lifted** (get rid of only declared symbols),
  - calls are replaced by these λ-lifted variables,
  - methods (n) `m` **def-**depends on are **not** λ-lifted: use of n's method generator
  - … **applied** to methods n **itself** has λ-lifted.

➡ Method generator **shared** along **inheritance** and between **collections** of a same species.

Code Generation: Method Generators (ended)

- Explicit polymorphism ➡ extra ƛ-lifts to introduce `representation`s of *Self* and of parameters.
- Methods and `representation` can depend on `representations` and methods of **collection parameters**.

➡ ƛ-lifts of dependencies upon parameters : **outermost** abstractions to fit `Coq`'s dependencies.

- Generated code grouped in a **module**.

➡ Enforce **modularity**.

➡ Benefit from a convenient **namespace** mechanism.

Code Generation: Collection Generators

- Code generation for collections: create **computational runnable** code and **checkable logical** term.
- Right version of the method generator: **last** definition in the inheritance tree.
- **Effective arguments** for method generator: retrieved from the species **hosting it** and **instantiations** of formal parameters done during **inheritance**.
- Apply separately each method generator to its effective arguments?

➡ **No** code sharing between **collections** issued from the **same** parameterized species.

- Share the **applications** of method generators to their arguments between **collections**: ↗ sharing.

Code Generation: Collection Generators (ended)

- Applications grouped into a **record** … move ƛ-lifts of all parameters dependencies **outside** the record.
- The obtained function is a **collection generator**.

- Go further and replace ƛ-lifts by **one unique** abstracting the whole collection parameter?

➡ **No**: would require **first-class** modules and **subtyping** in target languages!
Would reduce target languages candidates.

- Collection: obtained by **application** of its **generator** to get a **record** value.
- Methods of the collection: **picked** inside the **record** and surrounded by a **module**.

Conclusion

- **Design** and **feature choices** leading to an original compilation problem.

  Computational and logical aspects handled together, flexible development constructs, readable proofs, traceable code, etc.

- Difficulty 1: **dependency calculus** for consistency and code generation.
- Difficulty 2: **common code generation** model for all target languages.
- Difficulty 3: create the **context** where to insert Zenon proof.
- Difficulty 4: ensure **no errors** are raised by target languages.
- And number of other ones not presented here!

  Normal form, parameters instanciation, recursion & termination proofs, etc.

Thank you for your Attention

Some questions ?

I would like to thank:

- **Thérèse Hardin**, **Renaud Rioboo** (`FoC`'s parents),
- **Damien Doligez** from INRIA / Microsoft Research (`Zenon`'s dad),
- and other **folks** who gave advice and contribute to FoCaLiZe.

http://focalize.inria.fr