FoCaLiZe
Mixing Programs and Proofs

François Pessaux - ENSTA ParisTech (U2IS)
francois.pessaux@ensta-paristech.fr

PARKAS Seminar
26 May 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.

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
« Why should I Burden Myself with FoCaLiZe? »

- (*)& Code and properties not compiled separately.
- Data-types and properties both seen as Coq types.
- Definitions and theorems considered the same way.
  ➡ More confidence in « what is proved is (quite) what is ran ».

- Proofs directly in Coq? Requires specific user skills.
  ➡ Dedicated language: independence from logical target language.
  ➡ Hierarchical proof structure: human readable.
  ➡ Use of Zenon automated theorem prover: reduces the user’s burden.

- *: reveals usually implicit dependencies.
- No errors from target languages: ensure no missing « stuff one depends on ».
- Get compact code for traceability: minimize dependencies.
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**.

```plaintext
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**.

```plaintext
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**.
• Abstraced « instance » of a **complete** species.
• The **only** form of **proved** run-able code.

```plaintext
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.

```plaintext
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 \( m \))
    
    \( \Rightarrow \) 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.

```plaintext
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) \( \backslash \) 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**.

```ocaml
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) \( \lor \) 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 be typecheck-able.

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

Coq ⇒

  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 ∈ **transitive closure** of the def-dependency relation + methods on which these latter decl-depend: the visible universe.
Visible Universe

\[
\begin{align*}
y &\in \{x\}_{S} & y &\overset{\text{def}}{\leq} x \\
y &\in \{x\} & y &\in \{x\}
\end{align*}
\]

\[
\begin{align*}
z &\overset{\text{def}}{<} x & y &\in \{z\}_{S} \\
z &\in \{x\} & y &\in \{\mathcal{T}_S(z)\}_{S}
\end{align*}
\]

\begin{itemize}
  \item \( x \overset{\text{def}}{<} y \) : « y def-depends on x by transitivity »
  \item \( \mathcal{T}_S(x) \) : « the type of x in the species S ».
\end{itemize}
Minimal Typing Environment

\[ \emptyset \cap x = \emptyset \]

\[ y \notin \ x \mid \{y_i : \tau_i = e_i\} \cap x = \Sigma \]
\[ \{y : \tau = e ; y_i : \tau_i = e_i\} \cap x = \Sigma \]

\[ y \in \ x \mid y <^d e f x \]
\[ \{y_i : \tau_i = e_i\} \cap x = \Sigma \]
\[ \{y : \tau = e ; y_i : \tau_i = e_i\} \cap x = \{y : \tau = e ; \Sigma\} \]

\[ y \in \ x \mid y <^d e f x \]
\[ \{y_i : \tau_i = e_i\} \cap x = \Sigma \]
\[ \{y : \tau = e ; y_i : \tau_i = e_i\} \cap x = \{y : \tau ; \Sigma\} \]

- 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 \( t \ (\text{'a}) = \ldots \)
- \( \ldots (S \ast \text{int}) \ldots \)
- all \( x : t \ (\text{int}), \ y : S, \ f \ (x, S) \ldots \)

#### Table

<table>
<thead>
<tr>
<th>Peut dépendre de</th>
<th>Type</th>
<th>Preuve</th>
<th>Définition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Preuve</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Définition</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
</tbody>
</table>

- by type definition of \( \ldots \)
- On the representation:
  \(<2>1\) assume \( x : \text{Self}, \) prove \( x = 0 \)
- by type \( u \)
- all \( x : t \ (\text{int}), \ f \ (x) \ldots \)
- by property \( \ldots \)
- let \( f \ (x : S) = \ldots \)
- let \( g \ (x : \text{Self}) = \ldots \)

#### Diagram

- On the representation:
  let \( h \ (x : \text{Self}) = \) if \( x \ldots \)
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)

\[
\mathcal{D}oP_{\text{BODY}}(S, C)[x] = \mathcal{D}oP_{\text{EXPR}}(S, C)[B_S(x)]
\]

\[
\mathcal{D}oP_{\text{TYPE}}(S, C)[x] = \mathcal{D}oP_{\text{EXPR}}(S, C)[T_S(x)]
\]

- [Body]: harvest dependencies on a method \textbf{explicitly stated} in the \textbf{body} of a definition.
- [Type]: harvest dependencies on a method \textbf{explicitly stated} in the \textbf{type} of a definition.
Rules for Deps. on Parameters Methods (2/4)

\[ \mathcal{DoP}[^{\text{DEF}}](S, C)[x] = \mathcal{DoP}[^{\text{EXPR}}](S, C)[B_S(z)] \quad \text{for all } z \text{ such as } z <_{S}^{\text{def}} x \]

\[ \mathcal{DoP}[^{\text{UNIV}}](S, C)[x] = \mathcal{DoP}[^{\text{EXPR}}](S, C)[T_S(z)] \quad \text{for all } z \text{ such as } z \in | x | \]

- [Def] and [Univ]: collect dependencies of \textbf{a method} on a parameter induced by the dependencies this method has in \textbf{its hosting} species.
- Note: methods \textit{z} introduced by [Def] included in those introduced by [Univ] (\textit{vis. univ. wider than only transitive def-deps and their related decl-deps}).
Rules for Deps. on Parameters Methods (3/4)

\[
\mathcal{E}(S) = (\ldots, C_p \text{ is } \ldots, C_p' \text{ is } S'(\ldots, C_p, \ldots)) \\
\mathcal{E}(S') = (\ldots, C_k' \text{ is } I_k', \ldots)
\]

\[z \in \text{DoP}_{[\text{TYPE}]}(S, C_p')[x] \lor z \in \text{DoP}_{[\text{BODY}]}(S, C_p')[x]\]

\[(y : \tau_y) \in \text{DoP}_{[\text{TYPE}]}(S', C_k')[z]\]

\[
(y : \tau_y[C_k' \leftarrow C_p]) \in \text{DoP}_{[\text{PRM}]}(S, C_p)[x]
\]

- 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.
• 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 $\lambda$-lifted (get rid of only declared symbols),
  • calls are replaced by these $\lambda$-lifted variables,
  • methods $(n) m$ def-depends on are not $\lambda$-lifted: use of $n$’s method generator
  • … applied to methods $n$ itself has $\lambda$-lifted.
 ➞ Method generator shared along inheritance and between collections of a same species.
• Explicit polymorphism ➞ extra λ-lifts to introduce representations of Self and of parameters.

• Methods and representation can depend on representations and methods of collection parameters.

  ➞ λ-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 λ-lifts of all parameters dependencies outside the record.
- The obtained function is a collection generator.

- Go further and replace λ-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.*
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