Trusted Computer Mathematics within the Focalize Environment

David Delahaye

David.Delahaye@cnam.fr

The Focalize Project
(CNAM, LIP6, and INRIA)

MAP'10

Logroño, Spain
November 12, 2010

Introduction

The Focalize Environment

- Development of certified applications ;
- Specification and proof assistant tool ;
- Functional and object-oriented (inheritance, parameterization) ;
- Algebraic specification flavor (carrier type, implementation) ;
- Automated (Zenon) and verified (Coq) reasoning.

The Focalize Project

Three sites (and teams) :

- CNAM : D. Delahaye, V. Donzeau-Gouge, C. Dubois, R. Rioboo ;
- LIP6 : T. Hardin, M. Jaume ;
- INRIA : D. Doligez, P. Weis.

A Little History

The BiP Working Group :

- T. Hardin, V. Donzeau-Gouge, J.-R. Abrial ;
- Interactions between the Coq and B communities.

The Foc Project :

- T. Hardin, R. Rioboo, S. Boulmé ;
- Certified library of computer algebra ;
- Structures with inheritance, representation and parameterization.

Design of a Compiler :

- D. Doligez, V. Prevosto ;
- OCaml (execution), Coq (certification), FocDoc (documentation).

The Zenon ATP :

- D. Doligez ;
- First order, classical, with equality (tableaux) ; verification by Coq.

A Little History

Operational Semantics :

- T. Hardin, C. Dubois, S. Fechter ;
- Semantics closer to an implementation (compiler) ;
- Modeling of the object features (without properties and proofs).

Development of Applications :

- Computer algebra (R. Rioboo) ;
- Airport security (D. Delahaye, V. Donzeau-Gouge, J.-F. Étienne) ;
- Security policies (M. Jaume, C. Morisset) ;
- Components (M. V. Aponte, C. Dubois, V. Benayoun).

New compiler (Focalize) :

- F. Pessaux, P. Weis, D. Doligez, R. Rioboo, D. Delahaye, T. Hardin ;
- Rewriting of the compiler (version 0.6.0, may 2010).

Specification : Species

General Syntax

```
species <name> =
  [representation = <type>;]     (* representation *)
  signature <name> : <type>;     (* declaration *)
  let <name> = <body>;           (* definition *)
  property <name> : <prop>;      (* property *)
  theorem <name> : <prop>        (* theorem *)
  proof = <proof>;
end;;
```

Inheritance and Parameterization

```
species <name> (<name> is <name>[(<pars>)], <name> in <name>, ...) =
  inherit <name>, <name> (<pars>), ...;
end;;
```

Features

- Basic structure, more or less abstract (refined by inheritance) ;
- “Self” denotes the encapsulation of the representation.

Implementation : Collection

Syntaxe générale

collection *<name>* = **implement** *<name>* (<*pars*>); **end** ;;

Features

- Implements a completely defined species ;
- Does not provide additional code ;
- Terminal object ;
- Freezes an instance of a complete species ;
- The representation remains encapsulated ;
- Becomes a genuine type.

Compiler : Three Outputs

Execution

- OCaml code ;
- Only deals with the computational aspect (functions) ;
- Model based on records (objects, modules).

Certification

- Coq code ;
- Deals with all the attributes (functions and properties) ;
- Generated with the help of Zenon ;
- Model based on records (modules).

Documentation

- FocDoc code ;
- XML format (DTD, XSD) ;
- XSL stylesheets for LaTeX, HTML, and UML (XMI).

An Example : Stacks

Species *Stack*

```
species Stack (Typ is Setoid) =

  inherit Setoid;

  signature empty : Self;
  signature push : Typ → Self → Self;
  signature pop : Self → Self;
  signature last : Self → Typ;

  let is_empty (s) = equal (s, empty);

  property ie_push : all e : Typ, all s : Self, ~(is_empty (push (e, s)));

  property lt_push : all e : Typ, all s : Self,
    Typ!equal (last (push (e, s)), e);

  property id_ppop : all e : Typ, all s : Self, equal (pop (push (e, s)), s);

  theorem ie_empty : is_empty (empty)
  proof = by property equal_reflexive definition of is_empty;

end;;
```

An Example : Stacks

Species *Basic_object* (Root)

```
species Basic_object =
  let print (x : Self) = "<abst>";
  let parse (x : string) : Self = focalize_error ("not␣parsable");
end;;
```

Species *Setoid*

```
species Setoid =
  inherit Basic_object;

  signature equal : Self → Self → bool;
  signature element : Self;
  let different (x, y) = ~~equal (x, y);

  property equal_reflexive : all x : Self, equal (x, x);
  property equal_symmetric : all x y : Self, equal (x, y) → equal (y, x);
  property equal_transitive : all x y z : Self,
    equal (x, y) → equal (y, z) → equal (x, z); ...
end;;
```

Finite Stacks

Species *Is_finite*

```
species Is_finite (max in Int) =
  inherit Basic_object;
  signature length : Self → int;
  property length_max : all s : Self, length (s) <=0x Int!from_rep (max) ;
end;;
```

Finite Stacks

Collection *Int*

```
species Int_def =

  inherit Setoid;

  representation = int;

  let from_rep (a : Self) : int = a;
  let to_rep (a : int) : Self = a;
  let element = 0;
  let equal = ( =0x );
  let print (e) = string_of_int (e);
  let parse (s) = int_of_string (s);

  proof of equal_reflexive = assumed (* To do *);
  proof of equal_symmetric = assumed (* To do *);
  proof of equal_transitive = assumed (* To do *);

end;;

collection Int = implement Int_def; end;;
```

Finite Stacks

Species *Finite_stack*

```
species Finite_stack (Typ is Setoid, max in Int) =

  inherit Stack (Typ), Is_finite (max);

  let is_full (s) = length (s) =0x Int!from_rep (max);

  property lth_empty : length (empty) =0x 0;

  property lth_push : all e : Typ, all s : Self, ~(is_full (s)) →
    length (push (e, s)) =0x (length (s) + 1);

  property lth_pop : all s : Self, ~(is_empty (s)) →
    length (pop (s)) =0x (length (s) − 1);

end;;
```

An Implementation with Lists

Species *Fstack_list* (Complete)

```
species Fstack_list (Typ is Setoid, max in Int) =
  inherit Finite_stack (Typ, max);

  representation = list (Typ);

  let empty = [];
  let push (e, s) = if is_full (s) then focalize_error ("Full␣stack!")
                    else e :: s;
  let pop (s) = if is_empty (s) then focalize_error ("Empty␣stack!")
                else list_tl (s);
  let last (s) = if is_empty (s) then focalize_error ("Empty␣stack!")
                 else list_hd (s);
  let length (s) = list_length (s);
  proof of ie_push = ...;
  proof of lt_push = ...; ...

  let element = empty;
  let equal (s1, s2) = list_eq (Typ!equal, s1, s2);
  proof of equal_reflexive = ...;
  proof of equal_symmetric = ...;
  proof of equal_transitive = ...;

  let print (e : Self) = list_print (Typ!print, e) ^ "\n";
end;;
```

Collection of Stacks of Integers

Collection *Fstack_int*

```
collection Fstack_int = implement Fstack_list (Int, Int!to_rep (5)); end;;
```

Remarks

- The first effective parameter (collection parameter “is”) must be a collection implementing species *Setoid* (*Int*) ;
- The second effective parameter (entity parameter “in”) must be an entity of the collection passed as the first effective parameter (*Int*) ;
- The encapsulation of the representation by a collection requires to use injection functions for entity parameters (*to_rep*) ;
- Effective parameters of species are either collections, or entities, but never species (effective parameters are therefore concrete) ;
- Collections cannot be parameterized and the effective parameters of their implementations are therefore not formal parameters.

Use of the Collection

Some Tests

```
let a = Int!to_rep (1);;
let b = Int!to_rep (2);; ...
let s1 = Fstack_int!push (a, Fstack_int!push (b, Fstack_int!push (c,
  Fstack_int!push (d, Fstack_int!push (e, Fstack_int!empty)))));;

print_string (Fstack_int!print (s1));;
print_string ("Length␣=␣");;
print_endline (string_of_int (Fstack_int!length (s1)));;
```

Execution

```
1 2 3 4 5
Length = 5
```

Encapsulation of the Representation

```
print_int (list_hd (s1));;
```

```
Error: Types stack#Fstack_int and basics#list ('_a) are not compatible.
```

Another Implementation

Species *Efstack_list* (Complete)

```
species Efstack_list (Typ is Setoid, max in Int) =

  inherit Finite_stack (Typ, max);

  representation = int * list (Typ);

  let empty = (0, []);

  let push (e, s) =
    let lth = length (s) in
    if ( =0x ) (lth, Int!from_rep (max)) then focalize_error ("Full␣stack!")
    else ((lth + 1), e :: snd (s));

  let pop (s) =
    let lth = length (s) in
    if lth =0x 0 then focalize_error ("Empty␣stack!")
    else ((lth - 1), list_tl (snd (s)));

  let last (s) = if is_empty (s) then focalize_error ("Empty␣stack!")
                 else list_hd (snd (s));

  let length (s) = fst (s);
```

Another Implementation

Species *Efstack_list* (continued)

```
  let is_empty (s) = length (s) =0x 0;

  proof of ie_push =  ...;
  proof of lt_push = ...; ...
  proof of ie_empty = ...;

  let element = empty;

  let equal (s1, s2) =
    (fst (s1) =0x fst (s2)) && list_eq (Typ!equal, snd (s1), snd (s2));

  proof of equal_reflexive = ...;
  proof of equal_symmetric = ...;
  proof of equal_transitive = ...;

  let print (e in Self) = list_print (Typ!print, snd (e)) ^ "\n";

end;;
```

Another Implementation

Redefinition

- Function *is_empty* is redefined ;
- The proof of property *ie_empty* must be invalidated and redone !

Influences of Redefinition

- Redefinition requires to deal with late binding, both for functions and properties (method generators) :
  - For functions : all the functions occurring in the body of a function are systematically abstracted ;
  - For statements of properties : similar to functions, except that properties are abstracted as well ;
  - For proofs of properties : similar to statements, except that functions whose definition is used are not abstracted.
- The compiler deals with all of that automatically, and this is quite transparent for the user.

Another Collection of Stacks of Integers

Collection *Fstack_int*

```
collection Efstack_int = implement Efstack_list (Int, Int!to_rep (5)); end;;
```

Some Tests

```
let s2 = Efstack_int!push (a, Efstack_int!push (b, Efstack_int!push (c,
  Efstack_int!push (d, Efstack_int!push (e, Efstack_int!empty)))));;

print_string (Efstack_int!print (s2));;
print_string ("Length␣=␣");;
print_endline (string_of_int (Efstack_int!length (s2)));;
```

Execution

```
1 2 3 4 5
Length = 5
```

Another Example : Additive Monoids

Species *Additive_monoid*

```
species Additive_monoid =

  inherit Additive_semi_group, Setoid_with_zero;

  signature plus : Self → Self → Self;

  property zero_is_neutral : all x : Self,
    equal (plus (x, zero), x) ∧ equal (plus (zero, x), x);

  theorem zero_is_unique : all o : Self,
    (all x : Self, equal (x, plus (x, o))) → equal (o, zero)
  proof = ...;

end;;
```

Proof of *zero_is_unique*

- The proof is completed using Zenon, but must be detailed.
- We use a declarative language inspired by a proposition by L. Lamport.

A Detailed Proof

Proof of *zero_is_unique*

```
theorem zero_is_unique : all o : Self ,
  ( all x : Self , equal (x, plus (x, o))) → equal (o, zero)
proof =
  <1>1 assume o : Self ,
       hypothesis H1: all x : Self , equal (x, plus (x, o)) ,
       prove equal (o, zero)
       <2>1 prove equal (zero , plus (zero , o))
            by hypothesis H1
       <2>3 prove equal (o, zero)
            by step <2>1
               property zero_is_neutral , equal_transitive , equal_symmetric
       <2>4 conclude
  <1>2 conclude ;
```

Library of Computer Algebra

Formalized by R. Rioboo (Focalize Team).

Contents of The Library

- Standard CA constant domains : integers, modular arithmetics, etc.
- General polynomial arithmetics :
  - Distributed (sparse) representations ;
  - Recursive representations.
- Algorithms for :
  - Resultant computations ;
  - Univariate polynomial factorization over finite fields.

The Library in Figures

- 12,000 lines of Focal code ;
- Producing 40,000 lines of Coq ;
- And 9,500 lines of OCaml.

Airport Security Regulations

D. Delahaye, J.-F Étienne, and V. Viguié Donzeau-Gouge (Focalize Team).

Conception en Focal

Remarques sur l'exemple

- Exemple très simple (pour bien comprendre) ;
- Autres développements (Calcul Formel, sécurité des aéroports, ...) ;
- Les preuves peuvent être plus complexes (voir exposé de D. Doligez) ;
- On ne détaille pas la compilation (voir exposé F. Pessaux).

«Design patterns»

- Traits orientés objets de Focal ;
- Certains mis en évidence par la traduction de Focal vers UML ;
- «Design patterns» non comportementaux ;
- Collection : «Factory / singleton patterns» ;
- Place des preuves : V. Prevosto et M. Jaume, Calculemus 2003.

Focal dans le monde des méthodes formelles

Contexte

- Preuves formelles : partie infime du spectre ;
- Nombreux outils de preuves formelles (B, Coq, PVS, Mizar, ...) ?

	B	Focal
Langage	Impératif	Fonctionnel
Logique	Théorie des ensembles	Théorie des types
Spécification	Machine abstraite ou non	Espèce / collection
Développement	Raffinement	Héritage
Preuves	Prouveur automatique	Zenon (Coq)

	Coq	Focal
Langage	Fonctionnel	Fonctionnel
Logique	Th. types (ordre sup.)	Th. types (1er ordre)
Spécification	Section / Module	Espèce / collection
Développement	Inclusion	Héritage
Preuves	Manuel	Automatique (Zenon)

Conclusion

Quelques perspectives

- Génération de modèles UML ;
- Modélisation récursive ;
- Prouveur Zenon (induction, arithmétique, ...) ;
- Propriétés temporelles, systèmes réactifs.

Récupérer Focal

- Site Web : `http://focalize.inria.fr/` ;
- Distribution, documentation, tutoriel (bientôt), publications, ...

Exposés à suivre :

- «Preuves en Focal avec Zenon» (D. Doligez) ;
- «Focalize : le nouveau compilateur de Focal» (F. Pessaux) ;