Journée Focal 2009

Organisation

10h 10h45 11h30	«Une introduction à l'environnement Focal» (D. Delahaye) «Preuves en Focal avec Zenon» (D. Doligez) «Focalize : le nouveau compilateur de Focal» (F. Pessaux)
12h15	Déjeuner
13h30 17h	Séance de travaux pratiques (équipe Focal) Fin de la journée

Une introduction à l'environnement Focal

David Delahaye

David.Delahaye@cnam.fr

Équipe Focal (CNAM, LIP6, et INRIA)

Journée Focal 2009

LIP6, Paris
11 juin 2009

Introduction

Environnement Focal

- Développement de composants certifiés ;
- Outil de spécification et d'aide à la preuve ;
- Fonctionnel et orienté objets (héritage, paramétrisation) ;
- Orienté spécifications algébriques (type support, implantation) ;
- Preuve automatisée (Zenon) et vérifiée (Coq).

Équipe Focal

Trois sites (et équipes) :

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

Un peu d'histoire

Groupe BiP :

- T. Hardin, V. Donzeau-Gouge, J.-R. Abrial ;
- Interactions entre les communautés Coq et B.

Projet Foc :

- T. Hardin, R. Rioboo, S. Boulmé ;
- Bibliothèque certifiée de calcul formel ;
- Structures avec héritage, représentation et paramétrisation.

Conception d'un compilateur :

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

Prouveur automatique Zenon :

- D. Doligez ;
- Premier ordre classique avec égalité (tableaux) ; vérification par Coq.

Un peu d'histoire (suite)

Sémantique opérationnelle :

- T. Hardin, C. Dubois, S. Fechter ;
- Sémantique plus proche d'une implantation (compilateur) ;
- Modélisation des traits objets (sans les propriétés et les preuves).

Développement d'applications :

- Calcul Formel (R. Rioboo) ;
- Sécurité des aéroports (D. Delahaye, V. Donzeau-Gouge, J.-F. Étienne) ;
- Politiques de contrôle d'accès (M. Jaume, C. Morisset) ;
- Composants (M. V. Aponte, C. Dubois, V. Benayoun).

Nouveau compilateur (Focalize) :

- F. Pessaux, P. Weis, D. Doligez, R. Rioboo, D. Delahaye, T. Hardin ;
- Réécriture du compilateur (version 0.1, RC 1, avril 2009).

Spécification : espèce

Syntaxe générale

```
species <name> =
  [representation = <type>;]      (* représentation *)
  signature <name> : <type>;      (* déclaration *)
  let <name> = <body>;            (* définition *)
  property <name> : <prop>;       (* propriété *)
  theorem <name> : <prop>         (* théorème *)
  proof = <proof>;
end;;
```

Héritage et paramétrisation

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

Caractéristiques

- Brique de base plus ou moins abstraite (raffinée par héritage) ;
- La représentation est référencée par «Self».

Implantation : collection

Syntaxe générale

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

Caractéristiques

- Implante une espèce complètement définie ;
- Ne fournit pas de code ;
- Objet terminal ;
- La représentation est encapsulée.

Compilateur : trois sorties

Exécution

- Code OCaml ;
- Ne considère que l'aspect calculatoire (les fonctions) ;
- Modèle à enregistrements (objets, modules).

Certification

- Code Coq ;
- Considère tous les attributs (fonctions et propriétés) ;
- Produit avec l'aide de Zenon ;
- Modèle à enregistrements (modules).

Documentation

- Code FocDoc ;
- Format XML (DTD, XSD) ;
- Feuilles de style XSL pour LaTeX, HTML, et UML (XMI).

Un exemple : les piles

Espèce *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 in Typ, all s in Self, ~(is_empty (push (e, s)));

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

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

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

end;;
```

Un exemple : les piles

Espèce *Basic_object* (racine)

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

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

Les piles finies

Espèce *Is_finite*

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

Les piles finies

Collection *Int*

```
species Int_def =

  inherit Setoid;

  representation = int;

  let from (a in Self) in int = a;
  let inj (a in int) in 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;;
```

Les piles finies (suite)

Espèce *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 (max);

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

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

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

end;;
```

Une implantation avec des listes

Espèce *Fstack_list* (complètement définie)

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

Collection des piles d'entiers

Collection *Fstack_int*

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

Remarques

- Le premier paramètre effectif (paramètre de collection «is») doit être une collection implantant l'espèce *Setoid* (*Int*) ;
- Le deuxième paramètre effectif (paramètre d'entité «in»)) doit être une entité de la collection passée en premier paramètre effectif (*Int*) ;
- L'encapsulation de la représentation par une collection nécessite de prévoir des fonctions d'injection pour les paramètres d'entité (*inj*) ;
- Les paramètres effectifs sont soit des collections, soit des entités, mais jamais des espèces ;
- Les collections ne peuvent pas être paramétrées et les paramètres effectifs de leurs implantations ne sont donc pas des paramètres formels.

Utilisation de la collection

Un petit jeu de test

```
let a = Int!parse ("1");;
let b = Int!parse ("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)));;
```

Exécution

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

Encapsulation de la représentation

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

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

Une autre implantation

Espèce *Efstack_list* (complètement définie)

```
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 (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);
```

Une autre implantation

Espèce *Efstack_list* (suite)

```
  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;;
```

Une autre implantation

Redéfinition

- La fonction *is_empty* est redéfinie ;
- La preuve de la propriété *ie_empty* est invalidée et doit être refaite !

Influences de la redéfinition

- La redéfinition nécessite de gérer la liaison retardée, à la fois pour les fonctions et pour les propriétés (générateurs de méthodes) ;
- Pour les fonctions : toutes les fonctions apparaissant dans le corps d'une fonction sont systématiquement abstraites ;
- Pour les énoncés de propriétés : comme pour les fonctions, excepté que les noms des propriétés sont également abstraites ;
- Pour les preuves de propriétés : comme pour les énoncés, excepté que les fonctions dont on utilise la définition ne sont pas abstraites.
- Le compilateur gère tout cela automatiquement, et c'est complètement transparent pour l'utilisateur.

Une autre collection des piles d'entiers

Collection *Fstack_int*

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

Un petit jeu de test

```
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)));;
```

Exécution

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

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) ;