

Génération de code fonctionnel certifié à partir de spécifications dans l'environnement Focalize

D. Delahaye & C. Dubois & P.-N. Tollitte

journée SSURF

29 juin 2010



Objectifs

- Extraction de code fonctionnel à partir de spécifications
- Extraction de fonctions : déterminisme
- Extraction dans l'atelier Focalize

Utilité

- Gain de temps ; réduction du risque de bugs
- Implantation temporaire
- Animation de spécifications
- Oracle pour un framework de test

Moyens

- Ecriture des spécifications dans le langage Focalize
- Vérification des spécifications [DDE07]
- Extraction de code fonctionnel [DDE07]
- Optimisations

[DDE07] David Delahaye, Catherine Dubois, and Jean-Frédéric Étienne, Extracting Purely Functional Contents from Logical Inductive Types, Theorem Proving in Higher Order Logics (TPHOLs) (Kaiserslautern (Germany)), LNCS, vol. 4732, Springer, September 2007, pp. 70-85.

Plan

Spécification

- Types inductifs
- Déclaration d'une relation dans *bool* à partir de laquelle sera réalisée l'extraction
- Propriétés qui spécifient cette relation

Exemple

```
type nat = | Zero | Succ (nat);;  
  
species AddSpecif =  
  signature add : nat → nat → nat → bool;  
  property addZ : all n : nat, add (n, Zero, n);  
  property addS : all n m p : nat, add (n, m, p) →  
    add (n, Succ (m), Succ (p));  
end;;
```

Que génère t-on ?

La génération de code se fait :

- À partir des spécifications (propriétés)
- En attribuant un rôle aux arguments de la relation déclarée : entrées ou sorties (mode)

Pour l'addition :

- Mode (1, 2) : addition
- Mode (2, 3) : soustraction
- Mode `all` (complet) : vérification de l'addition

Commande d'extraction

Exemple d'extractions

```
species AddImpl =  
  inherits AddSpecif;  
  (** {extract} add from add all with (addZ, addS) *)  
  signature add : nat → nat → nat → bool;  
  (** {extract} add12 from add (1, 2) with (addZ, addS) *)  
  signature add12 : nat → nat → nat;  
end;;
```

- Les spécifications utilisées sont fermées au moment de l'extraction

Restriction sur la spécification (1)

Forme des propriétés

```
signature  $f : \tau_1 \rightarrow \dots \rightarrow \tau_p \rightarrow bool;$   
property  $prop : \mathbf{all} \ x_i : \tau_i,$   
           $f_1 (a_{11}, \dots, a_{1p_1}) \rightarrow \dots \rightarrow f_n (a_{n1}, \dots, a_{np_n}) \rightarrow$   
           $f (a_1, \dots, a_p);$ 
```

- τ_i : types inductifs
- x_i : variables associées aux τ_i
- f_n : fonctions pour lesquelles on a donné un mode d'extraction
- a_{jk} : imbrication de constructeurs de types inductifs, de variables x_i et d'appels de fonctions (sauf dans les sorties des prémisses)

Restriction sur la spécification (2)

- On doit pouvoir calculer les sorties à partir des entrées
 - Analyse de cohérence de mode
- Non recouvrement des conclusions des propriétés (déterminisme)

Spécification de l'addition

```
signature add : nat → nat → nat → bool;  
property addZ : all n : nat, add (n, Zero, n);  
property addS : all n m p : nat, add (n, m, p) →  
  add (n, Succ (m), Succ (p));
```

Analyse de cohérence de mode

- Permet de vérifier qu'on peut obtenir les sorties de la fonction générée à partir des entrées (analyse de flot de données)
- Est réalisée propriété par propriété

Pour simplifier on se donne deux fonctions : *in* et *out* qui donnent la liste des variables qui sont des éléments connus ou à calculer pour une prémisse ou la conclusion d'une propriété.

Exemple

```
property addS : all n m p in nat , add(n , m, p) →  
add(n, Succ(m), Succ(p));
```

```
in(add(n, Succ(m), Succ(p)), (1, 2)) = {n, m}
```

```
out(add(n, Succ(m), Succ(p)), (1, 2)) = {p}
```

Algorithme

Pour une propriété de la forme $c_1 \rightarrow c_2 \rightarrow \dots \rightarrow c_n \rightarrow c_p$ (en mode m) : E_x représente l'ensemble des variables connues à une étape de l'algorithme.

- $E_0 = in(c_p, m)$
- On vérifie : $\forall x \in 1..n, in(c_x, m) \subseteq E_{x-1}$
- $\forall x \in 1..n, E_x = out(c_x, m) \cup E_{x-1}$
- On vérifie : $out(c_p, m) \subseteq E_n$

Exemple : addition en mode (1, 2)

- $addZ : add(n, Zero, n)$
 - $E_0 = in(add(n, Zero, n), (1, 2)) = \{n\}$
 - $out(add(n, Zero, n), (1, 2)) = \{n\} \subseteq E_0$
- $addS : add(n, m, p) \rightarrow add(n, Succ(m), Succ(p))$

Exemple : addition en mode (1, 2)

- $addZ : add(n, Zero, n)$
 - $E_0 = in(add(n, Zero, n), (1, 2)) = \{n\}$
 - $out(add(n, Zero, n), (1, 2)) = \{n\} \subseteq E_0$
- $addS : add(n, m, p) \rightarrow add(n, Succ(m), Succ(p))$

Exemple : addition en mode (1, 2)

- $addZ : add(n, Zero, n)$
- $addS : add(n, m, p) \rightarrow add(n, Succ(m), Succ(p))$
 - $E_0 = in(add(n, Succ(m), Succ(p)), (1, 2)) = \{n, m\}$
 - $in(add(n, m, p)) = \{n, m\} \subseteq E_0$
 - $E_1 = E_0 \cup \{p\} = \{n, m, p\}$
 - $out(add(n, Succ(m), Succ(p)), (1, 2)) = \{p\} \subseteq E_1$

Exemple : addition en mode (1, 2)

- $addZ : add(n, Zero, n)$
- $addS : add(n, m, p) \rightarrow add(n, Succ(m), Succ(p))$
 - $E_0 = in(add(n, Succ(m), Succ(p)), (1, 2)) = \{n, m\}$
 - $in(add(n, m, p) = \{n, m\} \subseteq E_0$
 - $E_1 = E_0 \cup \{p\} = \{n, m, p\}$
 - $out(add(n, Succ(m), Succ(p)), (1, 2)) = \{p\} \subseteq E_1$

Exemple : addition en mode (1, 2)

- $addZ : add(n, Zero, n)$
- $addS : add(n, m, p) \rightarrow add(n, Succ(m), Succ(p))$
 - $E_0 = in(add(n, Succ(m), Succ(p)), (1, 2)) = \{n, m\}$
 - $in(add(n, m, p)) = \{n, m\} \subseteq E_0$
 - $E_1 = E_0 \cup \{p\} = \{n, m, p\}$
 - $out(add(n, Succ(m), Succ(p)), (1, 2)) = \{p\} \subseteq E_1$

Exemple : addition en mode (1, 2)

- $addZ : add(n, Zero, n)$
- $addS : add(n, m, p) \rightarrow add(n, Succ(m), Succ(p))$
 - $E_0 = in(add(n, Succ(m), Succ(p)), (1, 2)) = \{n, m\}$
 - $in(add(n, m, p)) = \{n, m\} \subseteq E_0$
 - $E_1 = E_0 \cup \{p\} = \{n, m, p\}$
 - $out(add(n, Succ(m), Succ(p)), (1, 2)) = \{p\} \subseteq E_1$

Analyse de non recouvrement

- Elle permet de vérifier qu'il n'y a pas de non déterminisme (le résultat de la fonction générée doit être unique)
- Elle est réalisée par unification des sorties des conclusions des propriétés
- Cette contrainte peut être en partie contournée

Génération du code

```

species AddSpecif =
  signature add : nat → nat → nat → bool;
  property addZ : all n : nat, add (n, Zero, n);
  property addS : all n m p : nat, add (n, m, p) →
    add (n, Succ (m), Succ (p));
end;;

species AddImpl =
  inherits AddSpecif;
  (** {extract} add from add all with (addZ, addS) *)
  signature add : nat → nat → nat → bool;
end;;

```

Dans un langage fonctionnel de type ML (avec gardes) :

```

let rec add [[args]] = match [[args]] with
  | [[addZ]]
  | [[addS]]
  | [[defaut]]

```

Génération du code

```

species AddSpecif =
  signature add : nat → nat → nat → bool;
  property addZ : all n : nat, add (n, Zero, n);
  property addS : all n m p : nat, add (n, m, p) →
    add (n, Succ (m), Succ (p));
end;;

species AddImpl =
  inherits AddSpecif;
  (** {extract} add from add all with (addZ, addS) *)
  signature add : nat → nat → nat → bool;
end;;

```

Dans un langage fonctionnel de type ML (avec gardes) :

```

let rec add (p1, p2, p3) = match (p1, p2, p3) with
  | [[addZ]]
  | [[addS]]
  | [[defaut]]

```

Génération du code

```

species AddSpecif =
  signature add : nat → nat → nat → bool;
  property addZ : all n : nat, add (n, Zero, n);
  property addS : all n m p : nat, add (n, m, p) →
    add (n, Succ (m), Succ (p));
end;;

species AddImpl =
  inherits AddSpecif;
  (** {extract} add from add all with (addZ, addS) *)
  signature add : nat → nat → nat → bool;
end;;

```

Dans un langage fonctionnel de type ML (avec gardes) :

```

let rec add (p1, p2, p3) = match (p1, p2, p3) with
  | (n, Zero, x) when x = n → true
  | [[addS]]
  | [[defaut]]

```

Génération du code

```

species AddSpecif =
  signature add : nat → nat → nat → bool;
  property addZ : all n : nat, add (n, Zero, n);
  property addS : all n m p : nat, add (n, m, p) →
    add (n, Succ (m), Succ (p));
end;;

species AddImpl =
  inherits AddSpecif;
  (** {extract} add from add all with (addZ, addS) *)
  signature add : nat → nat → nat → bool;
end;;

```

Dans un langage fonctionnel de type ML (avec gardes) :

```

let rec add (p1, p2, p3) = match (p1 , p2 , p3) with
  | (n, Zero, x) when x = n → true
  | (n, S m, S p) → [[add(n, m, p)]]
  | [[defaut]]

```

Génération du code

```

species AddSpecif =
  signature add : nat → nat → nat → bool;
  property addZ : all n : nat, add (n, Zero, n);
  property addS : all n m p : nat, add (n, m, p) →
    add (n, Succ (m), Succ (p));
end;;

species AddImpl =
  inherits AddSpecif;
  (** {extract} add from add all with (addZ, addS) *)
  signature add : nat → nat → nat → bool;
end;;

```

Dans un langage fonctionnel de type ML (avec gardes) :

```

let rec add (p1, p2, p3) = match (p1 , p2 , p3) with
  | (n, Zero, x) when x = n → true
  | (n, S m, S p) → if add(n, m, p) then true else false
  | [[default]]

```

Génération du code

```

species AddSpecif =
  signature add : nat → nat → nat → bool;
  property addZ : all n : nat, add (n, Zero, n);
  property addS : all n m p : nat, add (n, m, p) →
    add (n, Succ (m), Succ (p));
end;;

species AddImpl =
  inherits AddSpecif;
  (** {extract} add from add all with (addZ, addS) *)
  signature add : nat → nat → nat → bool;
end;;

```

Dans un langage fonctionnel de type ML (avec gardes) :

```

let rec add (p1, p2, p3) = match (p1, p2, p3) with
  | (n, Zero, x) when x = n → true
  | (n, S m, S p) → if add(n, m, p) then true else false
  | _ → false

```

Génération du code

```

species AddSpecif =
  signature add : nat → nat → nat → bool;
  property addZ : all n : nat, add (n, Zero, n);
  property addS : all n m p : nat, add (n, m, p) →
    add (n, Succ (m), Succ (p));
end;;

species AddImpl =
  inherits AddSpecif;
  (** {extract} add from add all with (addZ, addS) *)
  signature add : nat → nat → nat → bool;
end;;

```

Dans un langage fonctionnel de type ML (avec gardes) :

```

let rec add (p1, p2, p3) = match (p1, p2, p3) with
  | (n, Zero, x) when x = n → true
  | (n, S m, S p) → if add(n, m, p) then true else false
  | _ → false

```

Astuce pour les gardes

On remplace les gardes par des tests.

Code généré avec gardes

```
let rec my_add (p1, p2, p3) = match (p1 , p2 , p3) with
  | (n, Zero, x) when x = n → true
  | (n, S m, S p) → if add (n, m, p) then true
                    else false
  | _ → false
```

Code généré sans garde

```
let rec add (p1, p2, p3) = match (p1 , p2 , p3) with
  | (n, Zero, x) → if x = n then true else false
  | (n, S m, S p) → if add (n, m, p) then true
                    else false
  | _ → false
```

Code Focalize généré

Représentation de l'AST

```
let rec add (p1, p2, p3) (struct p2) =  
  match (p1, p2, p3) with  
  | (n, Zero, n0) → if (n0 = n) then true else false  
  | (n, Succ (m), Succ (p)) → if add (n, m, p) then true  
                               else false  
  | _ → false;
```

Plan

Génération simultanée d'extractions dépendantes

```

signature eval12 : expr → envi → val;
(** {extract} eval12 from eval (1, 2) with (evalZero, evalTrue,
                                           evalFalse, evalSucc, evalVar)
      exec12 from exec (1, 2) with (pSkip, pAffect, pSequ, plfTrue,
                                   plfFalse, pWhileTrue, pWhileFalse) *)
signature exec12 : instr → envi → envi;
  
```

- Une seule annotation, plusieurs extractions
- Toutes les signatures doivent être présentes

Conclusions recouvrantes

```
property plfTrue : all env env1 : envi, all exp : expr,  
  all i1 i2 : instr,  
    eval(exp, env, VTrue) → exec(i1, env, env1) →  
      exec(If(exp, i1, i2), env, env1);
```

```
property plfFalse : all env env1 : envi, all exp : expr,  
  all i1 i2 : instr,  
    eval(exp, env, VFalse) → exec(i2, env, env1) →  
      exec(If(exp, i1, i2), env, env1);
```

- Utilisation des premisses pour choisir la propriété utilisée en fonction des entrées
- Fusion de propriétés
- Renommage de variables

Fusion de propriétés

```

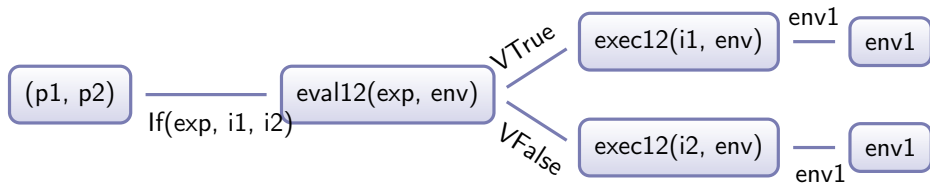
property plfTrue : all env env1 : envi, all exp : expr,
  all i1 i2 : instr,
    eval(exp, env, VTrue) → exec(i1, env, env1) →
      exec(If(exp, i1, i2), env, env1);

```

```

property plfFalse : all env env1 : envi, all exp : expr,
  all i1 i2 : instr,
    eval(exp, env, VFalse) → exec(i2, env, env1) →
      exec(If(exp, i1, i2), env, env1);

```



Bilan

```

signature eval : expr → envi → val → bool;

property evalZero : all env: envi, eval(Zero, env, VZero);
property evalTrue : all env: envi, eval(TTrue, env, VTrue);
property evalFalse : all env: envi, eval(TFalse, env, VFalse);
property evalSucc : all env: envi, all n: expr, all v: val, eval(n, env, v) →
  eval(Succ(n), env, VSucc(v));
property evalVar : all env: envi, all v: var,
  eval(Var(v), env, list_assoc(v, env));

signature exec : instr → envi → envi → bool;

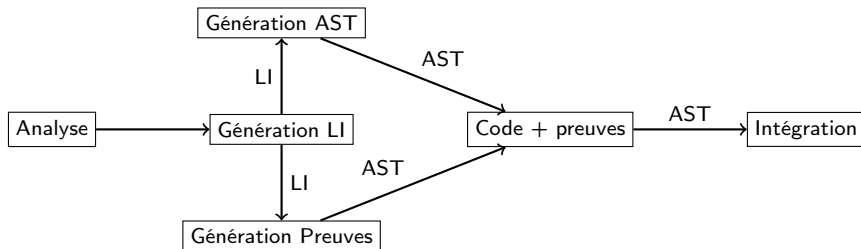
property pSkip : all env : envi, exec(Skip, env, env);
property pAffect : all env : envi, all v : var, all exp : expr, all va: val,
  eval(exp, env, va) → exec(Affect(v, exp), env, modif_envi(env, v, va));
property pSequ : all env env1 env2 : envi, all i1 i2 : instr,
  exec(i1, env, env1) →
  exec(i2, env1, env2) →
  exec(Sequ(i1, i2), env, env2);
property plfTrue : all env env1 : envi, all exp : expr, all i1 i2 : instr,
  eval(exp, env, VTrue) → exec(i1, env, env1) →
  exec(If(exp, i1, i2), env, env1);
property plfFalse : all env env1 : envi, all exp : expr, all i1 i2 : instr,
  eval(exp, env, VFalse) → exec(i2, env, env1) →
  exec(If(exp, i1, i2), env, env1);

```

Plan

Implantation

- Module activable ou non à la compilation
- Arrêt de la compilation sur une erreur



Intégration du code produit

- Remplacement des signatures par des définitions de fonction (**let**)
- Préservation des annotations
- Utilisation de la fonction possible dans le reste du programme

Exemple

```
species AddSpecif =  
  signature add : nat → nat → nat → bool;  
  property addZ : all n : nat, add (n, Zero, n);  
  property addS : all n m p : nat, add (n, m, p) →  
    add (n, Succ (m), Succ (p));  
end;;  
species AddImpl =  
  inherits AddSpecif;  
  (** {extract} add from add all with (addZ, addS) *)  
  signature add : nat → nat → nat → bool;  
  let my_function(a, b, c) =  
    let res = add(a, b, c) in ...;  
end;;
```

Plan

Conclusion

- Conclusion
 - Module de génération de code fonctionnel entièrement automatique
 - Code fonctionnel utilisable dans la spécification
 - Génération simultanée de plusieurs spécifications dépendantes
 - Optimisations
- Limites
 - Déterminisme
 - Récursion structurelle (pour la compilation vers Coq)
 - Fonctions en sortie de prémisses

Travaux futurs

- Backtracking
- Arbres de décision
- Preuves de correction de la génération de code