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

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

JFLA 2010

1^{er} février 2010

Objectifs

- Extraction de code fonctionnel à partir de spécifications
- Extraction dans l'atelier Focalize
- Certification du code produit

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 [*DDE*07]
- Extraction de code fonctionnel [*DDE*07]
- Production des preuves

[*DDE*07] 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

Présentation de Focalize

Un atelier complet pour le développement de programmes certifiés :

- Inclut un langage de spéficication, un langage fonctionnel et un langage de preuve
- Comporte des traits orientés objet
- Génère du code OCaml exécutable et du code Coq pour la certification
- Utilise Zenon, prouveur automatique du premier ordre

Le langage

- Espèce :
  - Représentation (type support)
  - Déclarations (**signature**) et définitions (**let**) de fonctions
  - Propriétés et preuves (écrites pour Zenon ou directement en Coq)
- Collection : espèce complète, où tous les attributs sont concrets

Un programme Focalize est une arborescence d'espèces dont les collections sont des feuilles, et des expressions à toplevel.

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 all from (addZ addS) (struct 2);
  extract add12 = add (1, 2) from (addZ addS) (struct 2);
end;;
```

- Les spécifications utilisées ne sont pas fermées contrairement aux prédicats définis inductivement utilisés dans [*DDE*07]

Restriction sur la spécification (1)

Forme des propriétés

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

- 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 $c_1 \longrightarrow c_2 \longrightarrow \ldots \longrightarrow c_n \longrightarrow c_p$ (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 $(1, 2)$

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

Exemple : addition en mode $(1, 2)$

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

Exemple : addition en mode $(1, 2)$

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

Exemple : addition en mode $(1, 2)$

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

Exemple : addition en mode $(1, 2)$

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

Exemple : addition en mode $(1, 2)$

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

Analyse de non recouvrement

- Elle permet de vérifier qu'il n'y a pas de non déterminisme (le résultat de la fonction généré 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 all from (addZ addS) (struct 2);
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 all from (addZ addS) (struct 2);
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 all from (addZ addS) (struct 2);
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 all from (addZ addS) (struct 2);
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 all from (addZ addS) (struct 2);
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
  | [[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 all from (addZ addS) (struct 2);
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 all from (addZ addS) (struct 2);
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;
```

Génération des preuves

- Uniquement pour la récursion structurelle (la fonction doit pouvoir s'écrire sous la forme d'un Fixpoint).
- Induction non gérée par Zenon : preuves générées directement en Coq
- Génération par rapport au code Coq produit par le compilateur
- Deux cas :
  - Mode complet : preuves des propriétés données par l'utilisateur
  - Mode partiel : preuves que la sortie de la fonction générée respecte bien les propriétés

Exemple : pour l'addition en mode (1, 2)

```
theorem add12_addZ :
  all n : nat, add(n, Zero, add12(n, Zero))

theorem add12_addS :
  all n m p : nat, add(n, m, add12(n, m)) →
    add(n, Succ (m), add12(n, Succ (m)))
```

Preuves obtenues

- Cas du mode total :

```
theorem addZ : all n in nat, add (n, Zero, n);
proof = coq proof
  {* intro n; simpl; generalize (basics.beq_refl nat__t n);
     case (basics._equal_ nat__t n n); auto. *};

theorem addS : all n m p in nat, add (n, m, p) →
  add (n, Succ (m), Succ (p));
proof = coq proof {* intros n m p; simpl; case (add n m p); auto.
```

- Cas du mode partiel :

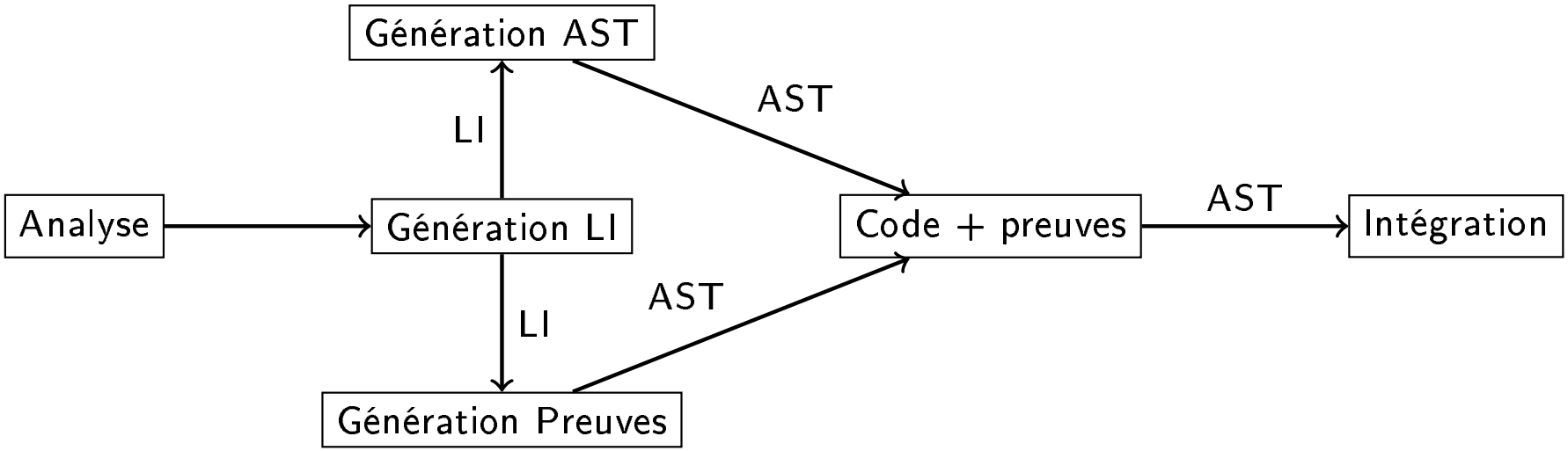
```
theorem add12_addZ : all n : nat, add (n, Zero, add12 (n, Zero))
proof = coq proof {* intro n; simpl; apply addZ. *};

theorem add12_addS : all n m p : nat, add (n, m, add12 (n, m)) →
  add (n, Succ (m), add12 (n, Succ (m)))
proof = coq proof {* intros n m p H; simpl; apply addS; auto. *};
```

Plan

Implantation

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



Mode total

```
Fixpoint add (a1 a2 a3 : (nat__t)%type) {struct a2} : basics.bool__t :
  match (a1, a2, a3) with
  | (n, Zero, n0) ⇒ if (basics._equal_ _ n0 n) then true else false
  | (n, Succ m, Succ p) ⇒ if (add n m p) then true else false
  | _ ⇒ false
  end.

Theorem addZ : forall n : nat__t, Is_true (add n Zero n).
Proof.
  intro n; simpl; generalize (basics.beq_refl nat__t n);
  case (basics._equal_ nat__t n n); auto.
Save.

Theorem addS : forall n m p : nat__t, Is_true (add n m p) →
  Is_true (add n (Succ m) (Succ p)).
Proof.
  intros n m p; simpl; case (add n m p); auto.
Save.
```

Mode partiel

```
Fixpoint add12 (a1 a2 : (nat__t)%type) {struct a2} : nat__t :=
  match (a1, a2) with
  | (n, Zero) ⇒ n
  | (n, Succ m) ⇒
    match (add12 n m) with
    | p ⇒ (Succ p)
    end
  end.

Theorem add12_addZ : forall n : nat__t,
  Is_true (add n (Zero) (add12 n Zero)).
Proof.
  intro n; simpl; apply addZ.
Save.

Theorem add12_addS : forall n m p : nat__t,
  Is_true (add n m (add12 n m)) →
  Is_true (add n (Succ m) (add12 n (Succ m))).
Proof.
  intros n m p H; simpl; apply addS; auto.
Save.
```

Intégration du code produit

- Code et théorèmes produits invisibles pour l'utilisateur
- Remplacement de la commande d'extraction par une définition de fonction (**let**)
- 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 all from (addZ addS) (struct 2);
  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
  - Preuves de correction générées automatiquement
  - Code fonctionnel utilisable dans la spécification
- Limites
  - Déterminisme
  - Récursion structurelle
  - Fonctions en sortie de prémisse
  - Non-linéarité en sortie de prémisse

Travaux futurs

- Génération simultanée de plusieurs spécifications dépendantes
- Optimisations
  - Heuristique pour le non-determinisme
  - Récursion bien fondée
  - Contraintes sur les sorties de prémisse (fonctions, non-linéarité)
- Généralisation du framework de génération de code