Animation de sémantiques formelles définies inductivement

David Delahaye

David.Delahaye@cnam.fr
CEDRIC/CNAM, Paris, France

Séminaire LIFL, Lille
27 mars 2013

Introduction

Travaux en collaboration avec Catherine Dubois, Jean-Frédéric Étienne, et Pierre-Nicolas Tollitte.

Comment formaliser des sémantiques de langages ?

Utilisation de l'outil d'aide à la preuve Coq :

- Codage des règles sémantiques avec des types inductifs ;
- Codage naturel et direct ;
- Raisonnement possible sur les sémantiques formalisées ;
- Support important de l'induction en Coq.

Mais l'évaluation est bloquée :

- On ne peut pas tester les spécifications inductives ;
- Seule solution : écrire l'évaluateur et montrer sa correction.

Introduction (suite)

Objectif principal : exécuter des spécifications.

Problème : comment exécuter des relations inductives ?

- Elles peuvent avoir plusieurs comportements calculatoires ;
- Leur évaluation peut nécessiter du « backtracking ».

Contrainte : rester purement fonctionnel.

- Les langages cibles de l'extraction sont fonctionnels ;
- Les langages des outils de preuve sont fonctionnels.

Notre méthode repose sur :

- Une analyse de mode pour savoir si un calcul est possible ;
- Une génération de code fonctionnel (heuristiques).

Un exemple

La relation d'addition add sur les nombres naturels : add n m p.

En Coq, la relation add est définie comme suit :

```
Inductive add : nat → nat → nat → Prop :=
  | addO : forall n, add n O n
  | addS : forall n m p, add n m p → add n (S m) (S p).
```

En OCaml, extraction avec le mode $\{1, 2\}$;

```
let rec add p0 p1 = match p0, p1 with
  | n, O → n
  | n, S m → let p = add n m in S p
  | _ → assert false
```

Un exemple (suite)

Extraction avec le mode $\{3, 2\}$:

```
let rec add p0 p1 = match p0, p1 with
  | n, O → n
  | S p, S m → add p m
  | _ → assert false
```

Extraction avec le mode $\{1, 2, 3\}$:

```
let rec add p0 p1 p2 = match p0, p1, p2 with
  | n, O, m when n = m → true
  | n, S m, S p → add n m p
  | _ → false
```

Le mode $\{1, 3\}$ est non déterministe : (S n, S n).

Analyse de cohérence de mode

Analyse de flots de données (entrées/sorties) :

- Un mode $\equiv$ un ensemble de positions d'entrée ;
- Au plus une position de sortie.

```
Inductive add : nat → nat → nat → Prop :=
  | addO : forall n, add n O n
  | addS : forall n m p, add n m p → add n (S m) (S p).
```

Analyse en mode $\{1, 2\}$:

- add0 : $S_0 = \{n\}, \{n\} \subseteq S_0$;
- addS : $S_0 = \{n, m\}, \{n, m\} \subseteq S_0$, $S_1 = \{n, m, p\}, \{p\} \subseteq S_1$.

Analyse de cohérence de mode (suite)

Un exemple d'échec : λ-calcul simplement typé (utilisation d'un style à la Curry).

```
Inductive typing : env → term → type → Prop := ...
  | abs : forall (e : env) (t1 t2 : type) (x : var) (t : term),
    (typing (add_env (x, t1) e) t t2) →
    (typing e (Abs (x, t)) (Arr t1 t2)).
```

Le mode $\{1, 2\}$ n'est pas cohérent : t1 $\notin S_0 = \{e, x, t\}$.

Génération de code

```
Inductive add : nat → nat → nat → Prop :=
  | addO : forall n, add n O n
  | addS : forall n m p, add n m p → add n (S m) (S p).
```

Mode $\{1, 2\}$:

```
let rec add p0 p1 = match p0, p1 with
  | n, O → n
  | n, S m →
    (match add n m with
      | p → S p
      | _ → assert false)
  | _ → assert false
```

Génération de code

```
Inductive add : nat → nat → nat → Prop :=
  | addO : forall n, add n O n
  | addS : forall n m p, add n m p → add n (S m) (S p).
```

Mode $\{1, 2\}$:

```
let rec add p0 p1 = match p0, p1 with
  | n, O→ n
  | n, S m→
    (match add n m with
      | p→ S p
      | _→ assert false)
  | _→ assert false
```

Génération de code

```
Inductive add : nat → nat → nat → Prop :=
  | addO : forall n, add n O n
  | addS : forall n m p, add n m p → add n (S m) (S p).
```

Mode $\{1, 2\}$:

```
let rec add p0 p1 = match p0, p1 with
  | n, O → n
  | n, S m →
    (match add n m with
      | p → S p
      | _ → assert false)
  | _ → assert false
```

Génération de code

```
Inductive add : nat → nat → nat → Prop :=
  | addO : forall n, add n O n
  | addS : forall n m p, add n m p → add n (S m) (S p).
```

Mode $\{1, 2\}$:

```
let rec add p0 p1 = match p0, p1 with
  | n, O → n
  | n, S m →
    (match add n m with
      | p → S p
      | _ → assert false)
  | _ → assert false
```

Génération de code

```
Inductive add : nat → nat → nat → Prop :=
  | addO : forall n, add n O n
  | addS : forall n m p, add n m p → add n (S m) (S p).
```

Mode $\{1, 2\}$:

```
let rec add p0 p1 = match p0, p1 with
  | n, O→ n
  | n, S m→
    (match add n m with
      | p→ S p
      | _→ assert false)
  | _→ assert false
```

Génération de code

```
Inductive add : nat → nat → nat → Prop :=
  | addO : forall n, add n O n
  | addS : forall n m p, add n m p → add n (S m) (S p).
```

Mode $\{1, 2\}$:

```
let rec add p0 p1 = match p0, p1 with
  | n, O → n
  | n, S m →
    (match add n m with
     | p → S p
     | _ → assert false)
  | _ → assert false
```

Génération de code

```
Inductive add : nat → nat → nat → Prop :=
  | addO : forall n, add n O n
  | addS : forall n m p, add n m p → add n (S m) (S p).
```

Mode $\{1, 2\}$:

```
let rec add p0 p1 = match p0, p1 with
  | n, O→ n
  | n, S m→
    (match add n m with
     | p→ S p
     | _→ assert false)
  | _→ assert false
```

Génération de code

```
Inductive add : nat → nat → nat → Prop :=
  | addO : forall n, add n O n
  | addS : forall n m p, add n m p → add n (S m) (S p).
```

Mode $\{1, 2\}$:

```
let rec add p0 p1 = match p0, p1 with
  | n, O→ n
  | n, S m→
    (match add n m with
     | p→ S p
     | _→ assert false)
  | _→ assert false
```

Génération de code (suite)

```
Inductive add : nat → nat → nat → Prop :=
  | addO : forall n, add n O n
  | addS : forall n m p, add n m p → add n (S m) (S p).
```

Mode $\{1, 2, 3\}$:

```
let rec add p0 p1 p2 = match p0, p1, p2 with
  | n, O, m when n = m → true
  | n, S m, S p → add n m p
  | _ → false
```

Génération de code (suite)

```
Inductive add : nat → nat → nat → Prop :=
  | addO : forall n, add n O n
  | addS : forall n m p, add n m p → add n (S m) (S p).
```

Mode $\{1, 2, 3\}$:

```
let rec add p0 p1 p2 = match p0, p1, p2 with
  | n, O, m when n = m → true
  | n, S m, S p → add n m p
  | _ → false
```

Génération de code (suite)

```
Inductive add : nat → nat → nat → Prop :=
  | addO : forall n, add n O n
  | addS : forall n m p, add n m p → add n (S m) (S p).
```

Mode $\{1, 2, 3\}$:

```
let rec add p0 p1 p2 = match p0, p1, p2 with
  | n, O, m when n = m → true
  | n, S m, S p → add n m p
  | _ → false
```

Génération de code (suite)

```
Inductive add : nat → nat → nat → Prop :=
  | addO : forall n, add n O n
  | addS : forall n m p, add n m p → add n (S m) (S p).
```

Mode $\{1, 2, 3\}$:

```
let rec add p0 p1 p2 = match p0, p1, p2 with
  | n, O, m when n = m → true
  | n, S m, S p → add n m p
  | _ → false
```

Génération de code (suite)

```
Inductive add : nat → nat → nat → Prop :=
  | addO : forall n, add n O n
  | addS : forall n m p, add n m p → add n (S m) (S p).
```

Mode $\{1, 2, 3\}$:

```
let rec add p0 p1 p2 = match p0, p1, p2 with
  | n, O, m when n = m → true
  | n, S m, S p → add n m p
  | _ → false
```

Implantation

Implanté pour dernière la version (8.4) de Coq (plugin « RelationExtraction ») :

- Intégré au mécanisme usuel d'extraction (transparent) ;
- Plusieurs sorties d'extraction : OCaml, Scheme, Haskell.

Voir le papier :

- David Delahaye, Catherine Dubois, and Jean-Frédéric Étienne. Extracting Purely Functional Contents from Logical Inductive Types. *Theorem Proving in Higher Order Logics (TPHOLs)*, 2007.

Optimisations

Sorties de prémisses, mode $\{1, 2\}$:

```
Inductive typecheck : env → expr → type → Prop := ...
  | if : forall (g : env) (b, e1, e2 : expr) (t : type),
      (typecheck g b bool) → (typecheck g e1 t) →
      (typecheck g e2 t) → (typecheck g (if b then e1 else e2) t).

let rec typecheck g e = match g, e with ...
  | g, If (b, e1, e2) →
    (match typecheck g b with
       | Bool → let t = typecheck g e1 in
         (match typecheck g e2 with
          | t' when t' = t → t
          | _ → assert false)
       | _ → assert false)
```

Optimisations (suite)

Entrées de conclusions, mode $\{1, 2\}$:

```
Inductive exec : store → command → store → Prop := ...
  | while1 : forall (s s1 s2 : Sigma) (b : expr) (c : command),
      (eval s b true) → (exec s c s1) →
      (exec s1 (while b do c) s2) → (exec s (while b do c) s2)
  | while2 : forall (s : Sigma) (b : expr) (c : command),
      (eval s b false) → (exec s (while b do c) s).
```

```
let rec exec s c = match s, c with ...
  | s, While (b, c) →
    (match (eval s b) with
       | true →
         let s1 = exec s c in
         let s2 = exec s1 (While (b, c)) in s2)
       | false → s
```

Motivations

Idée : extraire directement vers Coq.

Avantages :

- Pas besoin de sortir du formalisme de Coq ;
- Possibilité d'effectuer les preuves de correction.

Difficultés :

- Environnement plus contraint ;
- Terminaison ;
- Fonctions partielles ;
- Exhaustivité du filtrage.

Prototype

Restriction à la récursion structurelle (pour le moment).

Génération automatique des preuves de correction :

- Modes partiels (au moins une sortie) ;
- Spécifications complètes.

Forme des théorèmes de correction générés :

```
Theorem add12_correct :
  forall (n m p : nat), (add12 n m) = p → add n m p.
```

Voir le papier :

- Pierre-Nicolas Tollitte, David Delahaye, and Catherine Dubois. Producing Certified Functional Code from Inductive Specifications. *Certified Programs and Proofs (CPP)*, 2012.

Conclusion

Perspectives :

- Compléter le processus de génération de preuves ;
- Traiter la récursion générale, la coinduction ;
- Extraire des fonctions avec des compteurs ;
- Passer à l'échelle (challenge) : CompCert.

Activités dans DaRT/DreamPal :

- Architectures dynamiquement reconfigurables ;
- Environnement logiciel de développement ;
- Plusieurs langages (de bas et haut niveaux), compilateur ;
- Outils sémantiques permettant d'animer les programmes.