Alt-Ergo

vers une intégration réflexive en Coq d'un démonstrateur automatique modulaire

Sylvain Conchon, Evelyne Contejean et Stéphane Lescuyer

ADT Coq – 24 Mars 2009

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE
INRIA
centre de recherche SACLAY - ÎLE-DE-FRANCE
LABORATOIRE DE RECHERCHE EN INFORMATIQUE
UNIVERSITÉ PARIS-SUD 11

Plan de la présentation

1. Présentation du projet
   - Alt-Ergo
   - Certification/Intégration en Coq
2. Etat du projet
   - Une tactique réflexive pour la logique propositionnelle
   - Clôture par congruence
3. Problèmes récurrents
   - Modules
   - Réification
   - Hashconsing

Présentation d'Alt-Ergo

Alt-Ergo c'est :

- un démonstrateur automatique SMT dédié à la preuve de programmes
- 2 syntaxes d'entrée Why et SMT-lib

Présentation d'Alt-Ergo

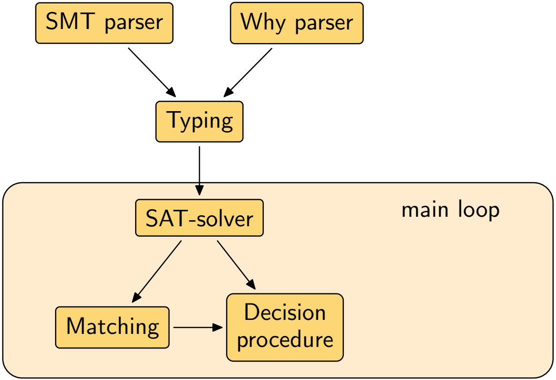
Alt-Ergo c'est :

- un démonstrateur automatique SMT dédié à la preuve de programmes
- 2 syntaxes d'entrée Why et SMT-lib
- logique du premier ordre
  - multi-sortée et polymorphe
  - opérateurs arithmétiques prédéfinies
  - théories spécifiques supplémentaires

Architecture générale d'Alt-Ergo

Alt-Ergo est basé sur :

- un solveur SAT
- un algorithme de combinaison de procédures de décision
- un mécanisme d'instanciation de quantificateurs



Prêt pour une certification ?

Le noyau d'Alt-Ergo :

- architecture totalement modulaire, à base de foncteurs
- théorie de l'égalité + ...
- structures de données purement fonctionnelles (sauf pour le *hash-consing*)
- 4000 lignes d'Ocaml (7500 au total)
- le code suit précisément la formalisation

Motivations

La certification d'Alt-Ergo en Coq poursuit un double but :

1. valider les algorithmes qui sont au cœur d'Alt-Ergo
   SAT solveur, algorithme de clôture par congruence, combinaison, ...
2. en dériver un (mini) Alt-Ergo certifié sous forme d'une tactique réflexive qui puisse être utilisée en Coq pour automatiser certaines preuves

Pourquoi “mini” ?

On ne veut pas tout faire dans Coq !

- parties plutôt calculatoires ⇒ les implémenter en Coq
- parties plutôt heuristiques ⇒ faire à l’extérieur, donner trace du résultat à Coq

Pourquoi “mini” ?

On ne veut pas tout faire dans Coq !

- parties plutôt calculatoires ⇒ les implémenter en Coq
- parties plutôt heuristiques ⇒ faire à l’extérieur, donner trace du résultat à Coq

Si l’interprétation des traces n’est pas sensiblement plus facile que la recherche de preuve elle-même ⇒ réflection

Pourquoi "mini" ?

On ne veut pas tout faire dans Coq !

- parties plutôt calculatoires $\Rightarrow$ les implémenter en Coq
- parties plutôt heuristiques $\Rightarrow$ faire à l'extérieur, donner trace du résultat à Coq

Si l'interprétation des traces n'est pas sensiblement plus facile que la recherche de preuve elle-même $\Rightarrow$ réflection

1. mécanisme de matching : à faire dans Alt-Ergo
   $\Rightarrow$ ne traiter que le fragment sans quantificateur dans Coq

Pourquoi “mini” ?

On ne veut pas tout faire dans Coq !

- parties plutôt calculatoires $\Rightarrow$ les implémenter en Coq
- parties plutôt heuristiques $\Rightarrow$ faire à l'extérieur, donner trace du résultat à Coq

Si l'interprétation des traces n'est pas sensiblement plus facile que la recherche de preuve elle-même $\Rightarrow$ réflection

1. mécanisme de matching : à faire dans Alt-Ergo
   $\Rightarrow$ ne traiter que le fragment sans quantificateur dans Coq
2. combinaison de théories : à faire dans Coq

Pourquoi “mini” ?

On ne veut pas tout faire dans Coq !

- parties plutôt calculatoires $\Rightarrow$ les implémenter en Coq
- parties plutôt heuristiques $\Rightarrow$ faire à l'extérieur, donner trace du résultat à Coq

Si l'interprétation des traces n'est pas sensiblement plus facile que la recherche de preuve elle-même $\Rightarrow$ réflection

1. mécanisme de matching : à faire dans Alt-Ergo
   $\Rightarrow$ ne traiter que le fragment sans quantificateur dans Coq
2. combinaison de théories : à faire dans Coq
3. SAT : intermédiaire, que faire ?
   $\Rightarrow$ possibilité d'utiliser des traces mais problème de taille
   $\Rightarrow$ on fait le choix de faire le SAT en Coq

Etat de la certification

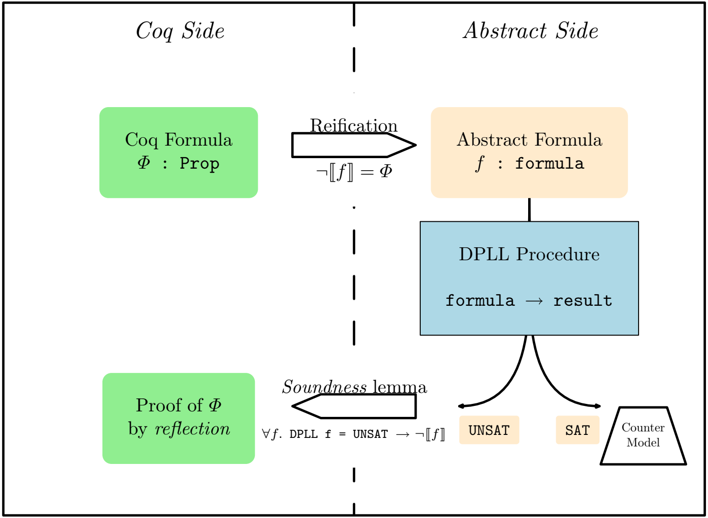
L'intégration en Coq peut se faire de manière modulaire, à l'image de l'architecture d'Alt-Ergo.

Etat de la certification

L'intégration en Coq peut se faire de manière modulaire, à l'image de l'architecture d'Alt-Ergo.

$\Rightarrow$ le SAT-solveur tout seul donne une tactique qui traite la logique propositionnelle
$\Rightarrow$ la combinaison de la théorie de l'égalité (et de la théorie vide) peut donner une tactique qui traite la clôture par congruence

La tactique unsat : schéma général



Formalisation modulaire

```
Module Type LITERAL.
  Parameter t : Set.
  ...
End LITERAL.

Module Type CNF.
  Declare Module L : LITERAL.
  Declare Module LSet : FSetInterface.S with Module E := L.
  ...
End CNF.

Module SAT (Import Cnf : CNF).
  ...
End SAT.
```

Séparer preuves et stratégies

Séparer dérivation et recherche de preuve

- raisonner sur une formalisation syntaxique abstraite
- implémenter séparément la stratégie de recherche de preuve
- partage de preuves entre différentes stratégies

Séparer preuves et stratégies

Séparer dérivation et recherche de preuve

- raisonner sur une formalisation syntaxique abstraite
- implémenter séparément la stratégie de recherche de preuve
- partage de preuves entre différentes stratégies

```
Module Type DPLL (Import Cnf : CNF).
  Inductive Res : Set :=
    Sat : LSet.t → Res
  | Unsat.

  Parameter dpll : formula → Res.
  Axiom dpll_correct :
    ∀Pb, dpll Pb = Unsat → incompatible (∅ ⊢ make Pb)
End DPLL.
```

Instancier le SAT sur Prop

```
Module LPROP < : LITERAL.
  Definition t := (index × bool)%type.
  Definition mk_not (p,b) : t := (p, negb b).
  ...
End LPROP.

Module CNFPROP < : CNF with Module L := LPROP := ...

Module PROPSAT := SAT(CNFPROP).
```

Instancier le SAT sur `Prop`

```
Module LPROP <: LITERAL.
  Definition t := (index × bool)%type.
  Definition mk_not (p,b) : t := (p, negb b).
  ...
End LPROP.

Module CNFPROP <: CNF with Module L := LPROP := ...

Module PROPSAT := SAT(CNFPROP).
```

⇒ toute la partie réification dans un foncteur `LoadTactic` paramétré par un module de type `DPLL(CNFPROP)`

Instancier le SAT sur Prop

```
Module LPROP < : LITERAL.
  Definition t := (index × bool)%type.
  Definition mk_not (p,b) : t := (p, negb b).
  ...
End LPROP.

Module CNFPROP < : CNF with Module L := LPROP := ...

Module PROPSAT := SAT(CNFPROP).
```

⇒ toute la partie réification dans un foncteur `LoadTactic` paramétré par un module de type `DPLL(CNFPROP)`

⇒ également possible de l'instancier sur `bool` !

Les tactiques unsat et valid

1. unsat : preuve d'un but par réfutation du contexte

```
   A ∧ (C ∨ ~B ∧ (~D → ~A)) → B ∧ ~A → D → D → 2 = 3.

> unsat.
Proof completed.
```

2. valid : preuve d'un but par l'absurde

```
   A ∧ (C ∨ ~B ∧ (~D → ~A)) → ~C → ~D → ~A

> valid.
Proof completed.
```

Optimisations

Optimisations du SAT

- backjumping
- apprentissage de clauses
- (heuristiques)

$\Rightarrow$ Réalisées, mais peu probantes sur des buts Coq de taille typique

Optimisations

Optimisations du SAT

- backjumping
- apprentissage de clauses
- (heuristiques)

$\Rightarrow$ Réalisées, mais peu probantes sur des buts Coq de taille typique

La mise en CNF

- une mise en forme normale paresseuse
- n'ajoute pas de nouvelles variables
- n'augmente pas la taille de la formule
- permet un partage maximal des sous-formules

$\Rightarrow$ "converted a prover that didn't work into one that did"

Quelques chiffres

	tauto	CNF_C	CNF_A	Tseitin	Tseitin2	Lazy	LazyN
hole3	–	0.72	0.06	0.24	0.21	0.06	0.05
hole4	–	3.1	0.23	3.5	6.8	0.32	0.21
hole5	–	10	2.7	80	–	1.9	1.8
deb5	83	–	0.04	0.15	0.10	0.09	0.03
deb10	–	–	0.10	0.68	0.43	0.66	0.09
deb20	–	–	0.35	4.5	2.5	7.5	0.35
equiv2	0.03	–	0.06	1.5	1.0	0.02	0.02
equiv5	61	–	–	–	–	0.44	0.42
franzen10	0.25	16	0.05	0.05	0.03	0.02	0.02
franzen50	–	–	0.40	1.4	0.80	0.34	0.35
schwicht20	0.48	–	0.12	0.43	0.23	0.10	0.10
schwicht50	8.8	–	0.60	4.3	2.2	0.57	0.7
partage	–	–	–	13	19	0.04	0.06
partage2	–	–	–	–	–	0.12	0.11

Combinaison de procédure de décision

Le module de procédure de décision est basé sur CC(X), un algorithme de clôture par congruence modulo une théorie X.

Combinaison de procédure de décision

Le module de procédure de décision est basé sur CC(X), un algorithme de clôture par congruence modulo une théorie X.

Il permet de déterminer si un ensemble d'équations closes en entraîne une autre dans la théorie de l'égalité modulo une théorie X :

$$\bigwedge_{i \in I} u_i = t_i \models_X u = v$$

Combinaison de procédure de décision

Le module de procédure de décision est basé sur CC(X), un algorithme de clôture par congruence modulo une théorie X.

Il permet de déterminer si un ensemble d'équations closes en entraîne une autre dans la théorie de l'égalité modulo une théorie X :

$$\bigwedge_{i \in I} u_i = t_i \models_X u = v$$

- avec X = théorie vide, clôture par congruence traditionnelle
- avec X = arithmétique linéaire, `omega+congruence`

Formalisation

Mêmes principes que pour la formalisation du SAT-solveur

- modularité (théories, structures de données)

```
Module Type Theory.
  ...
End Theory.

Module Type UF (X : Theory).
  ...
End UF.

Module CCX (X : Theory) (Uf : UF X).
  ...
End CCX.
```

Formalisation

Mêmes principes que pour la formalisation du SAT-solveur

- modularité (théories, structures de données)

```
Module Type Theory.
  ...
End Theory.

Module Type UF (X : Theory).
  ...
End UF.

Module CCX (X : Theory) (Uf : UF X).
  ...
End CCX.
```

- implémentation calculatoire séparée de la formalisation
- raisonne sur des termes abstraits pour des raisons d'efficacité

Contrairement au SAT-solveur,

- pas de tactique qui permet d'utiliser l'algorithme de manière réflexive
- pas d'instantiation sur une théorie non triviale

Contrairement au SAT-solveur,

- pas de tactique qui permet d'utiliser l'algorithme de manière réflexive
- pas d'instantiation sur une théorie non triviale

La preuve en Coq de CC(X) aura permis de :

1. Corriger quelques inexactitudes du formalisme
2. Réduire le fossé entre le code et la formalisation
3. Trouver des critères plus fins pour la sélection de nouvelles égalités
4. Savoir quels axiomes il est suffisant qu'une théorie vérifie pour pouvoir être utilisée dans CC(X)

Problèmes rencontrés

Modules 1/2

Le temps d'instantiation des modules peut devenir rédhibitoire !

- temps d'instantiation pour avoir une tactique `unsat` : 15s
- avec des `FSetAVL` à la place des `FSetList` : plus d'1min

Modules 1/2

Le temps d'instantiation des modules peut devenir rédhibitoire !

- temps d'instantiation pour avoir une tactique `unsat` : 15s
- avec des `FSetAVL` à la place des `FSetList` : plus d'1min

```
Module L := ...
Module LSet := FSetAVL.Make(L).     (* 0.1s *)
Module CSet := FSetAVL.Make(LSet). (* 1.6s *)
```

```
Module X := ......          (* 0.2s *)
Module X' < : T := ..... (* 0.8s *)
Module M (X : T). End M.
Module MX := M X'.          (* 2s *)
```

Modules 2/2

En pratique, pour améliorer la situation :

- restreindre les signatures des modules avec : autant que possible
  - ⇒ mais empêche le calcul
  - ⇒ séparer types/fonctions et preuves de manière pas forcément souhaitable
  - ⇒ expliciter de nombreuses signatures
  - ⇒ impossible pour les modules de la lib. standard

Modules 2/2

En pratique, pour améliorer la situation :

- restreindre les signatures des modules avec : autant que possible
  - $\Rightarrow$ mais empêche le calcul
  - $\Rightarrow$ séparer types/fonctions et preuves de manière pas forcément souhaitable
  - $\Rightarrow$ expliciter de nombreuses signatures
  - $\Rightarrow$ impossible pour les modules de la lib. standard
- utiliser des records à la place des modules ? :)

Modules 2/2

En pratique, pour améliorer la situation :

- restreindre les signatures des modules avec : autant que possible
  - ⇒ mais empêche le calcul
  - ⇒ séparer types/fonctions et preuves de manière pas forcément souhaitable
  - ⇒ expliciter de nombreuses signatures
  - ⇒ impossible pour les modules de la lib. standard
- utiliser des records à la place des modules ? :)

- un < : amélioré ?
- un : qui ne bloque pas les calculs ?

Réification 1/2

La réification est une étape indispensable à toute tactique réflexive.

- Ltac s'avère trop lent pour des exemples non triviaux
- `quote` résoud bien des problèmes mais a des limitations :
  - réifier dans une hypothèse
  - utiliser une varmap existante
  - problème pour inverser $\longrightarrow$

Réification 1/2

La réification est une étape indispensable à toute tactique réflexive.

- Ltac s'avère trop lent pour des exemples non triviaux
- `quote` résoud bien des problèmes mais a des limitations :
  - réifier dans une hypothèse
  - utiliser une varmap existante
  - problème pour inverser $\rightarrow$
- Faire sa tactique en ML :
  - demande de se plonger dans l'API
  - compiler avec les sources
  - moins pratique à distribuer

Réification 2/2

Réifier des expressions dont les types des termes ne sont pas connus à l'avance

$\Rightarrow$ nécessaire pour faire une tactique réflexive à partir de CC(X)

Réification 2/2

Réifier des expressions dont les types des termes ne sont pas connus à l'avance

$\Rightarrow$ nécessaire pour faire une tactique réflexive à partir de CC(X)

- on réifie les symboles de fonction ET leurs types, en utilisant une varmap de varmaps

```
Definition vars := varmap {x : Set & (x × varmap x)%type}.
Definition symb := (index × index)%type.
```

- la fonction d'interprétation des termes devient très dépendante
- le type des termes réifiés est très complexe, et il devient difficile de faire le lien avec la notion de sémantique utilisée dans l'algorithme de combinaison

Hashconsing 1/3

Dans Alt-Ergo, on utilise la technique de *hashconsing* pour les structures de formules et de termes :

- partage maximal
- comparaison en temps constant
- un accès instantané à toute la structure de la donnée

$\Rightarrow$ absolument critique pour les performances

On aimerait utiliser une technique similaire dans l'implémentation Coq pour les formules dans le SAT solver, et pour les termes dans CC(X).

Hashconsing 2/3

On peut construire des tables de hash et un module de hashconsing en Coq...

$\Rightarrow$ à l'utilisation, on a besoin du fait que deux données hashconsées sont structurellement égales quand leurs indexs le sont

Hashconsing 2/3

On peut construire des tables de hash et un module de hashconsing en Coq...

$\Rightarrow$ à l'utilisation, on a besoin du fait que deux données hashconsées sont structurellement égales quand leurs indexs le sont

On aimerait "cacher" la table de hash et les invariants des données hashconsées

- $\Rightarrow$ "méta-invariant" qui lie n'importe quelle paire de données qui ont été créées via le module de hashconsing
- $\Rightarrow$ devrait être possible quand la table est fixe

Hashconsing 3/3

On ne connaît pas la table à l'avance, elle change à chaque appel à la tactique réflexive.

$\Rightarrow$ possible de faire un foncteur qui renvoit un module de termes (ou de formules) en prenant comme argument une table de hashconsing déjà remplie

$\Rightarrow$ nécessite d'instancier tout le système à chaque utilisation de la tactique, ce qui est beaucoup trop lent

Conclusion

En bref

+ une tactique réflexive pour la logique propositionnelle
+ une formalisation et une implémentation de CC(X)
+ + de 30000 lignes de Coq
+ modularité (stratégies, abstraction)
- modularité (temps d'instantiation)
- pas de comparaison efficace des données structurées
- on ne peut pas se passer de ML :)

Merci !

Questions ?