Preuves Interactives et Applications

Christine Paulin & Burkhart Wolff
http://www.lri.fr/~paulin/PreuvesInteractives

Université Paris-Saclay

Master Informatique FIIL, 2016–17

Polymorphisme

- Un objet est dit polymorphe lorsqu'il peut s'utiliser dans des contextes de typage différents.
- Exemple sur les listes, on peut construire avec les mêmes opérations des listes d'entiers, de booléens, de chaînes de caractères, des listes de listes.
- Le principe est que le même code s'applique à des objets de nature différente.
- Les langages de programmation offrent des mécanismes plus ou moins avancés pour mettre en œuvre cette facilité (comparer C, Java, Scala, ML par exemple)

Polymorphisme implicite

Dans les langages de programmation (et Isabelle), le polymorphisme est géré de manière plutôt implicite

```
# [];;
- : 'a list = []
# ([]:int list);;
- : int list = []
# ([]: 'a list list) ;;
- : 'a list list = []
```

- une infinité de types possibles
- se décrit par un type unique avec variables
- les variables sont (implicitement) universellement quantifiées

$$[] : \forall \alpha, \alpha \ \texttt{list}$$

- règle générale d'instanciation $\dfrac{t : \forall \alpha, \tau(\alpha)}{t : \tau(\tau')}$

Polymorphisme dans Coq

- Le système de type de Coq est (beaucoup) plus puissant que le λ-calcul simplement typé.
- Le polymorphisme est par défaut explicite
  - les variables de types doivent être *déclarées*
  - les instanciations sont données par l'utilisateur
  - chaque terme a un type *unique*

```
Inductive list (A : Type) : Type :=
    nil : list A | cons : A → list A → list A
nil : ∀ A : Type, list A
nil nat : list nat
Variable A : Type.
nil (list A): list (list A)
```

Arguments implicites dans Coq

- laisser le système deviner un terme : _

```
Check (cons _ 0 (nil _))
```

- forcer certains paramètres à être toujours omis
  - a posteriori

    ```
    Arguments nil {A}.  Arguments cons [A] a l.
    Check (cons 0 nil).
    ```

  - dans la déclaration

    ```
    Inductive listi {A : Type} : Type :=
        nili : listi | consi : A → listi → listi.
    Check (consi 0 nili).
    ```

  - dans un contexte : **Set Implicit Arguments.**
- Forcer certains arguments

```
Check (nil (A:=nat)).   Check @nil.
```

- Se renseigner sur les implicites d'une constante : **About** *name*.

Tactiques automatiques

- appliquer des lemmes enregistrés dans une base 1 fois (`trivial`) ou bien récursivement à profondeur max 5 (`auto`)
- utiliser une base spécifique (exemple `arith`, `bool`)

  `auto` **with** *name*
- ajouter des lemmes dans une base :
  - théorème : **Hint Resolve** *name*.
  - constructeurs d'un type : **Hint Constructors** *name*.
  - résoudre immédiatement les sous-buts : **Hint Immediate** *name*.
  - déplier la constante en tête : **Hint Unfold** *name*.
- résoudre des buts d'arithmétique linéaire : `omega`
  - **Require Omega**.
  - `auto` **with** *zarith*

Tacticals

Opérations pour combiner les tactiques

- appliquer `tac2` à tous les sous-buts issus de `tac1`

  `tac1` ;`tac2`

- essayer d'appliquer une tactique

  try `tac`

- appliquer *n* fois une tactique

  do *n* `tac`

- répéter une tactique jusqu'à ce qu'elle échoue

  repeat `tac`

Organisation des développements

- Les développements peuvent être structurés en modules.
- Les modules sont compilés de manière séparée (rechargement rapide).
- Un fichier *file.v* est compilé en un fichier *file.vo* (module *file*).
- Le module est chargé grace à la commande **Require** *file*
- Les noms de théorèmes dans les modules sont *qualifiés* *file.thm* (cf **Locate** *thm*)
  - **Require Import** *file* : donne accès aux noms courts dans le fichier
  - **Require Export** *file* : donne accès aux noms courts dans le fichier et récursivement dans les fichiers qui chargeront le fichier concerné.
- Coq est lancé par défaut avec peu de bibliothèques chargées.

```
Require Export Bool List Arith.
```

- Compiler ses propres développements
  - mettre les noms des fichiers Coq à compiler dans un fichier `Make`
  - exécuter la commande (linux)

    ```
    coq_makefile -f Make -o Makefile
    ```

  - utiliser `make`

Cours 4

1. Compléments sur Coq
   - Polymorphisme
   - Tactiques automatiques
   - Compilation

2. Modélisation de la logique propositionnelle
   - Formules propositionnelles
   - Arbres de décision binaires

3. Exercices

Exemple de modélisation : logique propositionnelle

- outil essentiel de preuve automatique
  - SAT-solver
  - BDD pour des preuves d'équivalence
- présentation théorique
- représentation plus efficace à l'aide d'arbres de décision
- étude de quelques algorithmes simples

Logique propositionnelle

$$A \doteq \top | \bot | x | \neg A | A \wedge B | A \vee B | A \Rightarrow B$$

- syntaxe des formules
- sémantique des formules : $[\![A]\!]_I \in \mathbb{B}$ avec I une interprétation des variables propositionnelles
- preuve d'équivalence entre formules
- cf fichiers `Formula.v` et `Interpretation.v`

Choix de modélisation

- représentation des variables : nat
- représentation des constantes propositionnelles $\top, \bot$ par un booléen
- un type pour les connecteurs binaires

```
Definition var := nat.
Definition atom := bool.
Inductive binop := And | Or | Impl.
Inductive form :=
     Var : var → form
   | Atom : atom → form
   | Neg : form → form
   | Bin : binop → form → form → form.
Notation top := (Atom true).
Notation bot := (Atom false).
Notation negp := Neg.
Notation andp := (Bin And).
Notation orp := (Bin Or).
Notation implp := (Bin Impl).
```

Sémantique

- une interprétation est une fonction des variables propositionnelles dans les booléens
- chaque connecteur propositionnel correspond à une fonction booléenne
- deux formules sont équivalentes si elles ont mêmes valeur de vérité pour toutes les interprétations

```
Definition interpbin (p:binop) : bool → bool → bool :=
     match p with And ⇒ andb | Or ⇒ orb |  Impl ⇒ implb end.

Fixpoint interp (I:interpretation) (P:form) : bool :=
  match P with (Var x) ⇒ I x
             | (Atom a) ⇒ a
             | (Neg Q)  ⇒ negb (interp I Q)
             | (Bin o Q R) ⇒ interpbin o (interp I Q) (interp I R)
     end.
Definition equiv P Q : Prop := ∀ I, interp I P = interp I Q.
Definition valid P : Prop := ∀ I, interp I P = true.
```

Arbres de décision binaires (BDT)

- Représentation arborescente des formules proche de la sémantique
- Définition $t \coloneqq \top | \bot | \mathtt{if}(x, t, t)$
- Version ordonnée : les variables sont ordonnées, croissantes sur les branches
- Version réduite : deux sous-arbres différents (non traité)
- Sémantique
- Opérations de construction
- Preuves de correction
- Fichier `BDT.v`

Définitions

```
Inductive bdt :=
  | Atomt : atom → bdt
  | IFt : var → bdt → bdt → bdt.
Notation topt := (Atomt true).
Notation bott := (Atomt false).
Definition vart n := IFt n topt bott.
Fixpoint interpt (I:interpretation) (P:bdt) : bool :=
     match P with
       | (Atomt a) ⇒ a
       | (IFt x P Q) ⇒ if I x then interpt I P else interpt I Q
     end.
```

Opérations sur les BDT

- Propriété de base
  - soit $f \in \texttt{BDT} \to \texttt{BDT}$ stable par équivalence ($A \equiv B \Rightarrow f(A) \equiv f(B)$)
  - on a $f(\texttt{if}(x,t,u)) = \texttt{if}(x, f(t), f(u))$
  - il suffit donc de se donner $f(\top)$ et $f(\bot)$
- Exemple de la négation

```
Fixpoint negt (t:bdt) : bdt := match t with
   Atomt a ⇒ Atomt (negb a)
 | IFt x l r ⇒ IFt x (negt l) (negt r)
 end.
```

Opérations binaires sur les BDT

- On pourrait faire de même pour la conjonction

```
Fixpoint andt (t u:bdt) : bdt := match u with
   Atomt a ⇒ if a then t else u
 | IFt x l r ⇒ IFt x (andt t l) (andt t r)
 end.
```

- correct mais ne préserve pas l'ordre des variables sur les branches
- définition correcte de $f(t,u)$

$$
\begin{array}{lll}
f(a_1,a_2) & = & a_1,a_2\ \texttt{atomiques} \\
f(a,\texttt{if}(x,l,r)) & = \texttt{if}(x,f(a,l),f(a,r)) & a\ \texttt{atomique} \\
f(\texttt{if}(x,l,r),a) & = \texttt{if}(x,f(l,a),f(r,a)) & a\ \texttt{atomique} \\
f(\texttt{if}(x,l_1,r_1),\texttt{if}(x,l_2,r_2)) & = \texttt{if}(x,f(l_1,l_2),f(r_1,r_2)) & \\
f(\texttt{if}(x_1,l_1,r_1),\texttt{if}(x_2,\_,\_)) & = \texttt{if}(x_1,f(l_1,u),f(r_1,u)) & x_1 < x_2 \\
f(\texttt{if}(x_1,\_,\_),\texttt{if}(x_2,l_2,r_2)) & = \texttt{if}(x_2,f(t,l_2),f(t,r_2)) & x_1 > x_2
\end{array}
$$

Définition récursive double

- décroissance se fait soit sur le 1er soit sur le 2nd argument
  $f(\mathtt{if}(x_1,l_1,r_1),\mathtt{if}(x_2,l_2,r_2)) = F(f(l_1,\mathtt{if}(x,l_2,r_2)), f(\mathtt{if}(x_1,l_1,r_1),l_2))$
- pas directement accepté par Coq : un argument qui décroit tout le temps
- Solution : utiliser deux points fixes imbriqués
  $f\ t\ u = F(f\ t'\ u)\,(f\ t\ u')$ avec $t' < t$ et $u' < u$
  - On définit $f\ t$ comme une fonctionnelle récursivement sur t,
    $f\ t$ peut donc appeler $f\ t'$
  - Dans le corps de $f\ t$, on introduit une nouvelle fonction récursive $f_t\ u$ qui calcule $(f\ t\ u)$ récursivement sur u.
    $f_t\ u$ peut donc appeler $f_t\ u'$
  - $\mathbf{fix}\ f\ t := (\mathbf{fix}\ f_t\ u := F(f\ t'\ u)\,(f_t\ u'))$
  - l'équation $f\ t\ u = F(f\ t'\ u)\,(f\ t\ u')$ est prouvable
- permet en fait un ordre lexicographique $f\ t\ u = F(f\ t'\ v)\,(f\ t\ u')$

Définition Coq

```
Variable op : atom → atom → atom.

Fixpoint bint (t u: bdt) : bdt :=
  match t with
     Atomt a ⇒ let fix brect (u:bdt) : bdt :=
                 match u with Atomt b ⇒ Atomt (op a b)
                            |  IFt x l r ⇒ IFt x (brect l) (brect r)
                 end
                in brect u
   | IFt x l r ⇒ let fix brect (u:bdt) : bdt := ...
```

Croissance des variables dans les BDT

- Plusieurs possibilités
  - Fonction récursive qui teste qu'un bdt est ordonné
  - Définition inductive
- Généralisation au fait que les variables sont comprises entre i et n

```
Inductive ordered (n:nat) : nat → bdt → Prop :=
    Oatomt : ∀ i a, ordered n i (Atomt a)
  | Oift : ∀ i x l r, (i <= x < n)
           → ordered n (S x) l → ordered n (S x) r
           → ordered n i (IFt x l r).
```

- Les opérations logiques sur les bdt préservent la croissance
- valeur d'une formule : valeur des variables dans les bornes

```
Lemma ordered_update : ∀ I x b i n t,
      ordered n i t →
      (x < i \/ n <= x) →
      interpt (update I x b) t = interpt I t.
```

- Définition de l'opération $\exists x, t$ correspondant à $t[x \leftarrow \top] \vee t[x \leftarrow \bot]$

Cours 4

Exercices

Utiliser le squelette `tp4.v`. Les deux parties sont indépendantes.

1. Transformation de formules propositionnelles en BDT
   - Ecrire et tester une fonction `form2bdt` qui transforme une formule en un BDT équivalent
   - Montrer que `form2bdt` préserve la sémantique des formules
   - A quelle condition sur i, j, n, m a-t-on `ordered` $n\,i\,t \Rightarrow$ `ordered` $m\,j\,t$? Enoncer et prouver le lemme.
   - Montrer que `form2bdt` renvoie un BDT ordonné entre 0 et le successeur de la plus grande variable de la formule.
2. Tester qu'un BDT (ordonné) est une tautologie
   - Ecrire une fonction booléenne qui teste si un BDT correspond à une tautologie
   - Enoncer et prouver la correction de ce test, on pourra distinguer le cas où le test est positif et le cas où le test est négatif