Master Parisien de Recherche en Informatique
Cours 2-7-2 Assistants de preuve
2007–2008

Projet

Preuves automatiques en arithmétique

Le projet doit être réalisé individuellement. Il sera envoyé par courrier électronique à l'adresse `paulin@lri.fr`, avec le sujet « `projet cours 2-7-2` », sous la forme d'une archive contenant les fichiers sources et un rapport synthétique. Le projet est à rendre vendredi 1er février, les soutenances auront lieu mardi 5 ou 12 février.

Le projet est facultatif mais fortement recommandé : la note finale est le maximum de la note d'examen écrit et de la moyenne examen-projet.

L'accent sera mis sur la qualité et la clarté de la modélisation, il vaut mieux ne pas faire l'ensemble du projet mais le faire bien (en particulier certaines preuves peuvent être admises).

1 Objectif

Le projet consiste en la réalisation en Coq d'une tactique automatique pour des formules arithmétiques simples qui s'appuie sur la recherche de cycles dans les graphes.

Le projet comportera les éléments suivants :

- Le développement d'un foncteur réalisant la recherche de chemins minimaux dans un graphe qui s'appuie sur une description abstraite du graphe analogue aux représentations utilisées dans Ocamlgraph (`http://ocamlgraph.lri.fr`)
- Une représentation concrète d'une structure de graphes.
- Une définition du type des formules arithmétiques simples et de la sémantique associée.
- La définition d'une fonction qui teste l'insatisfiabilité d'une formule arithmétique simple.
- La construction d'une tactique réflexive pour prouver des formules arithmétiques simples dans Coq.

2 Formules arithmétiques

2.1 Sémantique

Définitions

Les *formules arithmétiques simples* sont définies de la manière suivante :

$$\begin{array}{ll} F ::= & t\ \mathbf{binop}\ t | F \wedge F | F \vee F | F \Rightarrow F | \neg F | \bot \\ t ::= & \mathbf{var} | \mathbf{cte} | \mathbf{var} + \mathbf{cte} | \mathbf{var} - \mathbf{cte} \\ \mathbf{binop} & \leq | \geq | \neq | = | < | > \end{array}$$

Les *formules arithmétiques normalisées* sont définies de la manière suivante :

$$\begin{array}{ll} N ::= & C | C \vee N \\ C ::= & A | A \wedge C \\ A ::= & \mathbf{var}\ \leq\ \mathbf{var} + \mathbf{cte} \end{array}$$

Les constantes sont dans l'ensemble $\mathbb{Z}$ des entiers relatifs. Etant donné une *valuation* qui associe un entier relatif à chaque variable, il est possible de donner une sémantique à chaque formule (simple ou normalisée) sous la forme d'une proposition.

Questions

1. Définir deux fonctions qui à toute formule simple F (resp. normalisée N) et toute valuation ρ associe sa sémantique. On notera $[F]_\rho$ et $[N]_\rho$ ces sémantiques.
2. Montrer dans `Coq` que pour toute formule F et toute valuation ρ, on peut montrer $[F]_\rho \vee \neg[F]_\rho$.
3. Montrer dans `Coq` que $\neg(\exists\rho, [\neg F]_\rho) \Rightarrow \forall\rho, [F]_\rho$.
4. Montrer que pour toute formule arithmétique simple F, il existe une formule normalisée N equivalente, c'est-à-dire que pour toute valuation ρ les sémantiques de F et N coîncident. Afin d'interpréter les formules de la forme x **binop** c, il est nécessaire d'utiliser dans N une variable particulière x_0 qui est interprétée comme 0.
5. Tester les fonctions construites sur quelques exemples.

2.2 Méthode de résolution

Une formule F est *insatisfiable* si pour tout ρ, $\neg[F]_\rho$.

Une formule conjonctive C peut se voir comme un graphe dont les sommets sont les variables et les arêtes sont étiquetées par des entiers relatifs. Chaque formule atomique de C de la forme $x \leq y + c$ est interprétée par une arête de x à y de poids c. Lorsqu'il y a plusieurs arêtes, il suffit de garder celle de poids minimal. De même une arête de x à x de poids c correspond à une formule insatisfiable si $c < 0$ et peut être supprimée si $0 \leq c$.

L'existence d'un cycle de poids négatif dans le graphe correspondant à une formule C implique son insatisfiabilité (ie $\forall\rho\neg[C]_\rho$).

Une formule normalisée N peut se voir comme un ensemble fini de graphes ainsi construits. La formule N est insatisfiable si chacune des formules qui la compose est insatisfiable.

Questions

1. Introduire un type pour représenter un cycle dans une formule conjonctive C.
2. Construire une fonction qui étant donnée une formule conjonctive C et un cycle de poids négatif, construit une preuve d'insatisfiabilité de C.

3 Graphes

L'objectif de cette partie est de programmer en `Coq` un algorithme de recherche de chemins de poids minimaux dans un graphe orienté permettant de décider de la satisfiabilité des formules étudiées dans la première partie. Cette partie sera développée sur une organisation modulaire.

Questions

1. Spécifier en `Coq` un type de module pour la structure de graphes. On s'intéresse à des graphes orientés dont les sommets et les arêtes sont étiquetés. On suivra les grandes lignes de la description modulaire Ocaml tirée de la bibliothèque `ocamlgraph` donnée dans la figure 1 en ajoutant une partie logique correspondant à une description des propriétés des opérateurs.

   Les point principaux de cette interface sont les suivants :
   - La fonction `compare` appliquée à deux objets x et y renvoie 0 si x et y sont égaux, un entier négatif si $x < y$ et un entier positif si $y < x$. En `Coq` on pourra utiliser le type `comparaison` au lieu du type entier.
   - Le parcours de graphe se fait à l'aide d'itérateurs :
     - `fold_vertex` prend en argument une fonction f, un graphe g de sommets $v_1, \ldots, v_n$ et un objet x dans un type α. Le résultat de l'évaluation de `fold_vertex` $f\, g\, x$ est $f\, v_1\, (f\, v_2 \ldots (f\, v_n\, x) \ldots))$, l'ordre de traitement des sommets n'est pas spécifié.
     - `fold_succ_e` prend en argument une fonction f, un graphe g, un sommet v et un objet x dans un type α. Le résultat de l'évaluation de `fold_succ_e` $f\, g\, v\, x$ est $f\, e_1\, (f\, e_2 \ldots (f\, e_k\, x) \ldots))$, où $e_1, \ldots, e_k$ sont les arêtes issues de v dans le graphe g.

2. Construire dans Coq un foncteur qui étant donné un graphe, cherche les chemins de poids minimaux dans le graphe. La figure 2 contient une description Ocaml de l'algorithme pour la recherche de chemins minimaux dont le principe est expliqué ci-dessous.

   On s'appuie sur une description abstraite de la notion de poids d'une arête (un ensemble muni d'une fonction de comparaison, un zéro et une fonction d'addition). On remarque que si le graphe contient un cycle de poids négatif alors il n'existe pas de chemin de poids minimal (passer par le cycle permet de construire des chemins de poids arbitrairement petit). On choisit donc d'échouer si un tel cycle est trouvé. Les graphes considérés n'auront que des arêtes dont les sommets de source et de destination sont différents.

   L'idée de l'algorithme est la suivante : si G est un graphe et V est un sous-ensemble des sommets de G, on appelle G_V le graphe de mêmes sommets que G tel que il existe une arête de x à y dans G_V si et seulement si il existe un chemin de x à y dans G dont tous les sommets internes sont dans V. On choisit l'étiquette correspondant au chemin de poids minimal.

   Il est assez aisé de construire G_V de manière itérative sur les sommets.
   - Si $V = \emptyset$ alors $G_V = G$.
   - Si on a construit G_V et qu'on considère un sommet c qui n'est pas dans V alors soit $V' = V \cup \{c\}$. Pour chaque sommet x de G_V :
     - s'il n'y a pas d'arêtes de x à c dans G_V alors les arêtes issues de x dans $G_{V'}$ sont les arêtes issues de x dans G_V,
     - si on a une arête e_1 de x à c dans G_V on considère toutes les arêtes issues de c dans G_V. Soit e_2 une telle arête de c à y. On pose n' la somme des poids de e_1 et de e_2.
       - s'il y a une arête de x à y dans G_V de poids n alors on compare n et n'. Si $n' < n$ alors on modifie l'arête de x à y de G_V pour lui associer le poids n'
       - s'il n'y a pas d'arête de x à y dans G_V alors on crée une telle arête dans $G_{V'}$ avec le poids n', sauf si $x = y$ et $n' < 0$ auquel cas on échoue.
3. Définir en Coq une module implémentant la structure de graphes et dont la signature correspond à celle attendue par le foncteur précédent.
4. Utiliser les fonctions précédentes pour construire une tactique automatique de résolution de problèmes arithmétiques simples.

```
module type VERTEX = sig
  type t
  val compare : t → t → int
  type label
  val create : label → t
  val label : t → label
end

module type EDGE = sig
  type t
  val compare : t → t → int
  type vertex
  val src : t → vertex
  val dst : t → vertex
  type label
  val create : vertex → label → vertex → t
  val label : t → label
end

module type G = sig
  type t (* Abstract type of graphs *)

  module V: VERTEX
  type vertex = V.t
(* Vertices have type V.t and are labeled with type V.label-note that an *)
(* implementation may identify the vertex with its label *)
  module E: EDGE  with type vertex = vertex
  type edge = E.t
(* Edges have type E.t and are labeled with type E.label. *)

  val is_empty : t → bool
  val mem_vertex : t → vertex → bool
  val find_edge : t → vertex → vertex → edge
(* find_edge g v1 v2 returns the edge from v1 to v2 if it exists. The *)
(* behaviour is unspecified if g has several edges from v1 to *)
(* v2. Raises Not_found if no such edge exists. *)
  val succ_e : t → vertex → edge list
(* succ_e g v returns the edges going from v in g. Raises *)
(* Invalid_argument if v is not in g.*)

  val fold_vertex : (vertex → 'a → 'a) → t → 'a → 'a
  val fold_succ_e : (edge → 'a → 'a) → t → vertex → 'a → 'a

  val empty : t
(* The empty graph. *)
  val add_vertex : t → vertex → t
(* add_vertex g v adds the vertex v from the graph g. Just return g if *)
(* v is already in g. *)
  val add_edge_e : t → edge → t
(* add_edge_e g e adds the edge e in the graph g. Add also E.src e *)
(* (resp. E.dst e) in g if E.src e (resp. E.dst e) is not in g. Just *)
(* return g if e is already in g. *)
  val remove_edge : t → vertex → vertex → t
(* remove_edge g v1 v2 removes the edge going from v1 to v2 from the *)
(* graph g. Just return g if this edge is not in g. Raises *)
(* Invalid_argument if v1 or v2 are not in g.*)
end
```

FIG. 1 – Module Ocaml pour les structures de graphes (selon ocamlgraph)

```
(* Spécification abstraite des poids associés aux labels sur les arêtes *)
module type WEIGHT = sig
   type label
   type t
   val weight : label → t
   val zero : t
   val add : t → t → t
   val compare : t → t → int
end

module Floyd =
functor (G:G) →
functor (W : WEIGHT  with type label = G.E.label and type t=G.E.label) →
  struct
    exception Negative
    let min_trans g =
    G.fold_vertex
      (fun c gc → (* gc: fermeture transitive par les sommets traites avant c *)
    let lce = G.succ_e gc c in (* liste des aretes issues de c *)
    G.fold_vertex
      (fun x gx → (* gx: sommets avant x traités *)
         try let e1 = G.find_edge gc x c (* e1:arc de x à c *)
         in List.fold_left
         (fun ge e2 (* e2:arc de c à y *) →
            let y = G.E.dst e2 in
            let n' = W.add (W.weight (G.E.label e1)) (W.weight (G.E.label e2)) in
            if compare x y = 0 (* x = y *) then
              if compare n' W.zero < 0 then raise Negative (* boucle de poids négatif *)
              else ge (* pas de boucle créée *)
            else try let e = G.find_edge gc x y in (*e: arête de x à y dans gc *)
                 let n = W.weight (G.E.label e)
                 in if compare n' n < 0 then (* chemin de poids plus faible *)
                    G.add_edge_e (G.remove_edge ge x y) (G.E.create x n' y)
                    else ge
                 with Not_found → (* créer une arête *) G.add_edge_e ge (G.E.create x n' y)
         ) gx lce
         with Not_found → gx)
      gc gc)
      g g
end
```

FIG. 2 – Description Ocaml de l'algorithme de Floyd