2ème année du MPRI
Université Paris VII

Projet
Chantal Keller

Critère de terminaison sous-terme

Récriture

Professeurs: Évelyne Contejean
Xavier Urbain

9 janvier 2009

1 Introduction

1.1 Présentation

Ce rapport explique ma démarche concernant le projet de démonstration automatique consistant à implanter une extension du critère de terminaison "sous-terme" pour les paires de dépendances. Il est accompagné d'une archive contenant les répertoires et les fichiers suivants :

- `src` implante le projet en lui-même ; il contient les fichiers suivants :
  - `match_unif.ml` tel que fourni par le sujet ;
  - `utils.ml` implante quelques fonctions génériques utiles ;
  - `projet.ml` et `projet.mli` : le fichier implantant le critère de terminaison, et son interface ;
  - `printing.ml` implante du "pretty printing" pour les paires de dépendances, les graphes...
- `test` implante des tests très détaillés (la plupart des fonctions de `src/projet.ml` y sont testées) ;
- `base-test` implante des tests basés sur les systèmes de récriture TRS du Termination Problem Data Base, version 4, de Claude Marché [1].

La compilation nécessite la présence de :

- ocaml, version 3.10 ou plus, avec en particulier ocamlopt, ocamlyacc et ocamllex ;
- omake, version 0.9.8 ou plus ;
- ocamlgraph, version 1.0 ou plus ;
- ocamlfind, version 1.2.1 ou plus ;
- dot.

1.2 Généralités

1.2.1 Structures de données et efficacité

J'ai cherché à écrire du code le plus efficace possible. Pour cela, j'ai :

- utilisé les structures de données qui me paraissaient le plus efficace ;
- utilisé au maximum des fonctions récursives terminales (d'où l'utilisation fréquente des primitives `List.rev_append` et `List.rev_map`, lorsque l'ordre des listes n'importait pas).

J'ai choisi pour le type des graphes un enregistrement contenant trois champs :

- `graph : Graph.Pack.Digraph.t` : le graphe en lui même, ayant pour type le type des graphes orientés du paquet `Pack` de la bibliothèque ocamlgraph ;
- `vertex_of_dp : (dp, Graph.Pack.Digraph.V.t) Hashtbl.t` : une table de hachage associant un sommet à chaque paire de dépendance ;
- `dp_of_vertex : (Graph.Pack.Digraph.V.t, dp) Hashtbl.t` : une table de hachage associant une paire de dépendance à chaque sommet.

Les deux derniers champs sont nécessaires, car je n'ai vu aucun moyen de pouvoir associer un label du type que l'on souhaite aux sommets des graphes de la bibliothèque ocamlgraph. Elles servent donc à pouvoir faire correspondre les paires de dépendance et les sommets du graphe.

L'utilisation de deux tables de hachages peut paraître coûteuse comparée, par exemple, à l'utilisation d'une liste d'association. J'ai choisi cette option, car :

- si elle est plus coûteuse en mémoire, elle l'est moins en temps de recherche ;

– toutes les primitives de recherche dont j'avais besoin existent, ce qui n'est pas le cas pour une liste d'association.

1.2.2 Interface `projet.mli`

Cette interface est légèrement modifiée par rapport à celle qui nous a été fournie :
– j'ai ajouté les types des fonctions calculant les listes de symboles (définis, constructeurs...) afin que celles-ci soient accessibles par le fichier `test.ml`, pour pouvoir les tester ;
– le type des graphes n'est pas un type abstrait, afin d'être une fois encore connu du fichier `test.ml` pour pouvoir tester les fonctions traitant des graphes ;
– les types qui sont déjà définis dans `match_unif.ml` ne sont pas redéfinis, car `projet.ml` dépend du module `Match_unif` ;
– le type de `removable` devient `system -> (string*int) list -> dp -> can_be_removed`, ceci afin d'être en correspondance avec le type de retour de la fonction `find_projection` ;
– le type de `find_projection` devient
`system -> graph -> (symb * int) list -> (symb * int) list * dp`, car :
  – l'entier qui sert à définir la profondeur de recherche maximale est codé par une variable globale (voir paragraphe ci-dessous), et n'est pas utilisé par la fonction `find_projection`, mais par la fonction `removable` ;
  – j'ai besoin de la liste des symboles définis **avec leur arité**.

1.2.3 Profondeur de recherche

Lors du calcul pour savoir si l'on peut supprimer une paire de dépendance d'une partie fortement connexe du graphe de dépendances, il faut chercher, pour certaines paires de dépendance $\langle u, v\rangle$, si :

$$P(u)(\triangleright \cup \to_R)^+ P(v) \text{ ou } P(u)(\triangleright \cup \to_R)^* P(v)$$

On ne sait *a priori* pas en combien d'étape de réduction cela se produit ! Je l'ai donc imposé au travers de la variable globale `max_test`.

1.3 Plan

Dans une première partie, je détaillerai les fichiers du répertoire `src`, et donc mes idées pour implanter le critère "sous-terme". Ensuite, j'expliquerai les tests détaillés du répertoire `test`. Enfin, je présenterai comment le code peut "passer à l'échelle", au travers de l'application aux systèmes de récriture TRS du Termination Problem Data Base du répertoire `base-test`.

2 Répertoire `src`

La commande `omake` compile les fichiers se trouvant dans ce répertoire sous forme d'une bibliothèque `projet.cmxa` et `projet.a`.

2.1 Fichier `utils.ml`

Il implante les fonctions suivantes :

- `superfluous_list` retire les éléments redondants d'une liste passée en argument, en gardant pour chaque élément sa première apparition, et en inversant la liste (pour être récursive terminale) ;
- `max` calcule le maximum de deux entiers ;
- `list_replace` remplace le $i^{\text{ème}}$ élément d'une liste par un élément passé en argument ;
- `list_of_fun` calcule la liste de tous les $(f\ i)$ pour i compris entre deux valeurs passées en argument (elle est récursive terminale) ;
- `list_prod` calcule le produit d'une liste d'éléments de type `'a` avec une liste d'éléments de type `'a list`.

2.2 Fichier `projet.ml`

2.2.1 Calcul d'ensembles de symboles

Nous allons détailler quelques-unes des fonctions suivantes :
- `compute_def_symbs` calcule l'ensemble des symboles définis d'un système de récriture ;
- `compute_def_symbs_arit` calcule l'ensemble des symboles définis d'un système de récriture avec leurs arités ;
- `compute_symbs` calcule l'ensemble des symboles d'un système de récriture ;
- `compute_symbs_arit` calcule l'ensemble des symboles d'un système de récriture avec leurs arités ;
- `compute_const_symbs` calcule l'ensemble des symboles constructeurs d'un système de récriture.

Pour calculer l'ensemble des symboles définis d'un système de récriture s, il suffit, pour chaque règle $l \to r$ de s, d'ajouter à l'ensemble le symbole de tête de l (en évitant les redondances). Pour avoir également l'arité, il suffit de le coupler avec la longueur de la liste des arguments de ce symbole.

Pour calculer l'ensemble des symboles d'un système s, le principe est le même que précédemment, mais en regardant tous les symboles de toutes les règles de récriture (et non seulement le symbole de tête du membre gauche).

L'ensemble des symboles constructeurs est calculé en faisant la différence ensembliste des deux ensembles précédents.

2.2.2 Calcul de l'ensemble des paires de dépendance d'un système de récriture

Pour chaque règle de récriture $l \to r$, on ajoute à l'ensemble tous les couples (l, t), lorsque t parcourt l'ensemble des sous-termes de r commençant par un symbole défini.

Telle quelle est implantée, cette fonction peut *a priori* renvoyer un ensemble redondants de paires de dépendance ; mais en pratique, cela arrive très rarement.

2.2.3 Calcul d'une sur-approximation du graphe de dépendances

On commence par définir la fonction `max_term`, qui calcule la valeur maximale des variables du terme passé en argument. Cela va permettre de générer des variables fraîches.

On implante ensuite les fonctions `cap` et `ren`, correspondant respectivement aux fonctions CAP et REN du cours. `cap` remplace tous les sous-termes stricts d'un terme commençant pas un symbole définis par une variable fraîche ; `ren` renomme toutes les variables pour qu'elles

soient deux à deux distinctes. On a donc besoin d'un générateur de variables fraîche, défini comme suit :

```
let cur_var = ref (max_term t);;

let new_var () =
  incr cur_var;
  !cur_var;;
```

La création de la sur-approximation du graphe de dépendances consiste alors en :
- la création d'un sommet par paire de dépendance ;
- pour chaque couple de paire de dépendances (s, t), l'ajout d'une arête du sommet correspondant à s vers le sommet correspondant à t si et seulement si $\mathrm{REN}(\mathrm{CAP}(t))$ et s sont unifiables (on fait appel à la fonction `unification` de `match_unif.ml`).

Remarque : La fonction `ren` doit prendre comme argument t **et** s : en effet, on ne doit pas, en introduisant de nouvelles variables pour t, capturer des variables de s. `cur_var` est donc initialisée à `ref (max (max_term t) (max_term s))`.

2.2.4 Calcul des composantes fortement connexes du graphe de dépendances

Comme nous avons utilisé les graphes orientés de la bibliothèque ocamlgraph, nous pouvons faire appel à la fonction `Components.scc_list` donnant la liste des composantes fortement connexes d'un graphe, sous forme de listes de sommets. Il faut alors :
- reconstituer chaque sous-graphe à partir de la liste des sommets, ce que fait la fonction `subgraph_of_vertex_list` ;
- éliminer les graphes à un sommet mais sans arête, qu'ocamlgraph considère comme des composantes fortement connexes, mais qui n'en sont pas (du moins, ils ne nous intéressent pas pour notre problème : sans arête, il n'y a aucune chaîne de dépendance) ; c'est ce que fait la fonction `remove_no_edge`.

2.2.5 Réduits d'un terme en n pas

Pour pouvoir calculer la liste des réduits d'un terme t en n pas, il faut déjà calculer celle de ses réduits en un pas ! C'est ce que fait la fonction `compute_one_step_reds` : pour chaque règle de récriture $l \to r$, on regarde toutes les unifications possibles d'un sous-terme de t avec l, et on applique alors la substitution correspondante sur ce sous-terme (placé dans son contexte).

La liste des réduits d'un terme en n pas est n applications de `compute_one_step_reds` successivement à chaque liste de termes.

2.2.6 Projection simple

La projection simple de p à t est tout simplement :
- si t est une variable, t lui même ;
- si t est le terme $f(t_1, \ldots, t_n)$, le terme $t_{p(f)}$.

2.2.7 Application de l'ordre sous-terme à une paire de dépendance et une projection simple

Le théorème ne cherche pas à appliquer le système $(\rightarrow_R)$, mais le système $(\rhd \cup \rightarrow_R)$. Mon idée a donc été de commencer par calculer le système $(\rhd)$.

En pratique, on ne peut pas calculer ce système : il comporte un nombre infini de règles ! En revanche, on peut calculer le système $(\rhd')$, où $t \rhd' s$ si et seulement s est un sous-terme *direct* de t.

Je rappelle qu'au final, on veut savoir, étant donnés deux termes t et s, s'il existe n entre 0 (ou 1) et `max_test` tel que $t\ (\rhd \cup \rightarrow_R)^n\ s$. Pour `max_test` "suffisamment grand", cela revient à savoir s'il existe n entre 0 (ou 1) et `max_test` tel que $t\ (\rhd' \cup \rightarrow_R)^n\ s$. On répond donc bien au problème en calculant $(\rhd')$ et non $(\rhd)$.

Pour calculer le système $(\rhd')$, il suffit, pour chaque symbole f d'arité m, de générer l'ensemble de règles :

$$\{f(x_0, \ldots, x_{m-1}) \rightarrow x_0; f(x_0, \ldots, x_{m-1}) \rightarrow x_1; \ldots; f(x_0, \ldots, x_{m-1}) \rightarrow x_{m-1}\}$$

Finalement, pour savoir le résultat de l'application de l'ordre sous-terme à une paire de dépendance $\langle u, v\rangle$ et une projection simple p données, il suffit de chercher n entre 0 et `max_test` tel que $p(u)\ (\rhd' \cup \rightarrow_R)^n\ p(v)$. La fonction `removable` renvoie alors :

- si $n = 0$, `Large` ;
- si $n > 0$, `Strict` ;
- s'il n'existe pas de tel n, `No`.

2.2.8 Trouver une projection simple pour l'application du théorème à une paire de dépendance donnée

On se donne une paire de dépendance $\langle u, v\rangle$, qui se trouve dans une composante fortement connexe c du graphe de dépendance.

Définissons tout d'abord ce qu'est une *bonne* projection simple : c'est une projection qui respecte les conditions d'application du théorème, c'est-à-dire telle que `removable` renvoie `Strict` pour $\langle u, v\rangle$, et `Strict` ou `Large` pour les autres paires de dépendances de c. C'est ce que calcule la fonction `good_projection`.

Pour trouver une bonne projection, le plus simple et d'essayer toutes les projections, ce qui prend un temps :

$$O\left(\prod_{i=1}^{k} n_i\right)$$

où k est le nombre de symboles de fonction et n_i et l'arité de chaque symbole de fonction.

Une méthode plus élaborée consisterait à ne chercher à définir d'abord la projection que pour les deux (ou le) symbole(s) de têtes de u et v, de manière à ce que `removable` renvoie `Strict` pour $\langle u, v\rangle$, puis à définir progressivement la projection pour les autres symboles pour être dans les conditions d'application du théorème.

Cependant, cette méthode peut avoir la même complexité que la méthode consistant à tout essayer dans le pire des cas, et est assez compliquée à mettre en place si on veut qu'elle soit efficace. C'est pourquoi j'ai choisi la méthode qui consiste à tout essayer.

La fonction `all_projections` calcule l'ensemble de toutes les projections possibles. Enfin, `find_projection_dp` cherche une bonne projection pour $\langle u, v \rangle$, en levant l'exeption `Not_found` si elle n'en trouve pas.

2.2.9 Trouver une projection simple pour l'application du théorème

Je pense qu'il doit être possible de pouvoir "deviner" les paires de dépendances les plus susceptibles d'être retirées d'une composante fortement connexe ; cependant, comme nous n'en n'avons pas parlé en cours, je me contente d'essayer de trouver une projection simple pour chaque paire de dépendance.

La fonction `find_projection` applique donc `find_projection_dp` pour chaque paire de dépendance, jusqu'à soit trouver une projection, soit avoir testé toutes les paires de dépendance en n'ayant rien trouvé.

2.2.10 Répondre au problème

Enfin, la fonction `main` répond au problème posé. Prenant en argument un système s, elle calcule la liste des composantes fortement connexes de la sur-approximation du graphe de dépendances associé à s, et pour chacune de ces composantes fortement connexe, applique `find_projection` autant de fois qu'il le faut jusqu'à soit n'avoir plus d'arête dans les composantes fortement connexes générées, soit ne plus pouvoir retirer de sommet.

Il faut bien penser, lorsqu'on retire un sommet d'une composante fortement connexe, à recalculer les nouvelles composantes fortement connexes : retirer un sommet peut casser la forte connexité !

2.3 Fichier `printing.ml`

Ce fichier fait du pretty printing, en implantant des fonctions qui transforment en chaîne de caractère un terme, une paire de dépendance, ou encore une projection.

3 Répertoire `test`

3.1 Présentation

Dans ce répertoire, on trouve le fichier `test.ml`, qui implante neuf exemples, et a une sortie très détaillée.

Les cinq premiers exemples sont les exemples 7.3, 7.13, 7.13, 7.19 et 6.12 de la version de décembre 2008 du polycopié du cours. Les trois exemples suivants sont également tirés du cours ; il s'agit des lois de De Morgan pour la logique classique, de l'exemple classique :

$$\left\{ \begin{array}{rcl} a(a(x)) & \to & b(c(x)) \\ b(b(x)) & \to & a(c(x)) \\ c(c(x)) & \to & a(b(x)) \end{array} \right.$$

et du codage de l'addition pour les entiers binaires. Enfin, le dernier exemple est le contre-exemple 7.25 du polycopié.

La fonction `test` prend en argument un exemple, et fait un appel à toutes les fonctions de `src/projet.mli`. Elle affiche alors sur la sortie standard :

- l'ensemble des symboles, l'ensemble des symboles définis et celui des symboles constructeurs (avec ou sans arité) ;
- l'ensemble des paires de dépendances ;
- pour chacune de ces paires de dépendance, leur "ôtabilité" vis-à-vis de deux projections simples arbitraires ;
- pour chacune de ces paires de dépendance, si on trouve ou non une projection simple qui les met dans les conditions d'application du théorème.

Elle crée également les fichiers `.dot` suivants :

- `dep_graph.name.dot` : la sur-approximation du graphe de dépendances ;
- `dep_scc.7.name.i.dot` : les composantes fortement connexes de la sur-approximation du graphe de dépendances (i variant) ;
- `dep_scc_remaining.name.i.dot` : les composantes fortement connexes minimales au sens de l'énoncé (i variant).

3.2 Utilisation

La commande `omake` crée le fichier `run_test`, dont l'application `./run_test` fait tout ce qui est décrit dans le paragraphe ci-dessus. `omake` suivi de `./run_test` est équivalent à la commande `omake test`.

Enfin, la commande `omake dot` permet de compiler tous les fichiers `.dot` du répertoire vers des fichiers `.png`.

4 Répertoire `base-test`

4.1 Présentation

Pour savoir si mon code pouvait "passer à l'échelle", j'ai voulu le confronter à une base de problèmes de terminaison. Sur les conseils de Xavier Urbain, j'ai choisi la Termination Problems Data Base de Claude Marché [1], et plus particulièrement la structure de donnée TRS. J'ai repris les fichiers `trs_lexer.mll`, `trs_parser.mly` et `trs_ast.ml` fournis avec et permettant de générer l'arbre de syntaxe abstraite d'un fichier au format TRS. J'ai également repris tous les exemples de fichiers `.trs`, disponibles dans le répertoire `base-test/trs`.

J'ai implanté, dans le fichier `trs2system.ml`, une traduction de l'arbre de syntaxe abstraite TRS vers notre format des systèmes de récriture. Comme ce dernier est moins précis (par exemple, il ne gère pas les stratégies de réduction), j'ai "oublié" toutes ces précisions.

Enfin, le fichier `test.ml` lit un ou plusieurs fichiers `.trs`, calcule les graphes minimaux résultants pour prouver la terminaison (au sens de l'énoncé), et :

- s'il n'y en a pas, affiche sur la sortie standard que le système termine ;
- s'il y en a, génère des fichiers `.dot` correspondant.

Le fait d'afficher que le système termine, lorsque notre "termination checker" est capable de le dire, est intéressant ; par contre, le fait de générer des fichiers `.dot` pour lesquels on ne sait pas à quoi correspondent les sommets n'est pas très utile. En pratique, on aurait d'autres "termination checkers" qui exploiteraient les graphes fournit par le mien. Mon but était uniquement de voir si mon code pouvait fonctionner, et ce de manière efficace, sur des exemples concrets de taille moyenne.

Sur environ 60% des exemples, mon programme répond instantanément. Sur environ 25% des exemples, il répond en quelques secondes. Sur les 15% restant, il met beaucoup plus de

temps. Au niveau de la rapidité d'exécution, il est donc performant sur de petits et de moyens exemples, mais pas lorsque le graphe de dépendances est trop grand.

Au niveau de l'efficacité de l'algorithme, le critère sous-terme est assez performant, puisqu'il arrive à prouver qu'entre un quart et un tiers des systèmes terminent.

4.1.1 Utilisation

La commande `omake` permet de compiler le tout pour générer le fichier `test`. Ensuite, un appel à `./test foo.trs bar.trs ...` applique notre "termination checker" à chacun des fichiers passés en argument, selon la méthode décrite dans le paragraphe ci-dessus. `./test --help` explique comment utiliser le fichier `test`.

Une utilisation typique est donc, par exemple, `./test trs/Beerendonk/*.trs`.

Enfin, `omake dot` permet de créer des fichiers `.png` à partir de tous les fichiers `.dot` existants, mais ce n'est une fois encore pas très utile.

5 Conclusion

J'ai écrit un programme qui implante le critère de terminaison sous-terme pour les paires de dépendances des systèmes de récriture. Étant donné un système passé en argument, il peut dire s'il termine selon ce critère ; et s'il ne termine pas, il renvoie les composantes fortement connexes minimales du graphe de dépendance sur lesquelles on ne peut plus appliquer le critère.

On peut voir sur la base de tests simples que le programme semble correct (bien sûr, cela ne constitue pas une preuve...). La base de tests TRS montre que le programme est efficace, aussi bien au niveau du temps de calcul qu'au niveau de l'algorithme en lui-même, sur des exemples petits et moyens.

Références

[1] The termination problems data base. `http://www.lri.fr/~marche/tpdb`.