SMT Solver

Anatole Dahan, Paul-Nicolas Madelaine

12 juin 2017

Introduction

Nous présentons ici notre projet de sémantique et vérification : un SMT Solver pour la théorie de l'égalité, basé sur un SAT Solver DPLL, et la structure d'Union Find persistante donnée dans le sujet, modifiée pour prendre en compte les inégalités. Nous avons aussi rendu le SAT Solver utilisable en front-end, et un parser de fichiers dimacs a été implémenté.

1 Utilisation, composition du projet

1.1 Compilation et execution

Pour compiler l'exécutable, il suffit d'exécuter make. Nous utilisons `ocamlbuild`, il est donc nécessaire de supprimer les fichiers `cmi` et `cmo`.

Concernant l'exécution, elle sera de la forme

```
./main.native type mode path
```

où :

- `type` est `sat` ou `smt`, selon le type de formule qu'on cherche à résoudre.
- `mode` est `one` ou `all`, selon si l'on veut une ou toutes les valuations vérifiant la formule.
- `path` est le chemin vers le fichier contenant la formule à résoudre.

1.2 Le code source

Le code source est composé de 6 fichiers, dans l'ordre de dépendance :

- `mytypes.ml` : Contient les définitions des types utilisés dans le reste du code : `cnf` pour une formule de la théorie de l'égalité. `sat` pour une instance SAT.
- `unionfind.ml` : Contient une implémentation de la structure Union-Find persistante, comme décrite dans le document donné dans le sujet, modifiée pour gérer les inégalités. Lève l'exception `Impossible_action` lors d'une assignation illicite (inégalité entre deux éléments d'une même classe ou égalité entre deux éléments de classes qui ne peuvent être égales).

— `sat.ml` : Le SAT Solver. Le type `DPLL.t` est mutable, de façon à ce que la fonction `DPLL.solgen`, appelée sur le même objet de type `DPLL.t` donne toutes les valuations satisfaisantes. Lève `No_more_models` lorsque toutes les valuations satisfaisantes ont été données.

— `smt.ml` : Contient le démonstrateur SMT. La fonction principale est `gen_sol`, qui génère une solution, et renvoie la continuation, pour permettre de relancer le calcul jusqu'à épuisement des solutions, et donc les trouver toutes. Lève `No_more_models` lorsque toutes les valuations satisfaisantes ont été données.

— `parser.ml` : Tout le parsing des fichiers `cnfuf`. Le parsing des fichiers `dimacs` est contenu dans `sat.ml`.

— `main.ml` : Parse les arguments donnés lors de l'exécution, lance les tâches nécessaires en conséquences.

2 Fonctionnement

2.1 Union-Find

La gestion des inégalités est gérés par un tableau persistant de `Set`, tels qu'ils sont implémentés en OCaml, c'est à dire par des ABR. On obtient ainsi une complexité $O(\log n)$ pour la recherche et l'insertion, et $O(n \log n)$ pour l'union. On peut améliorer l'union de deux ABR jusqu'à O(n), en les transformant tous deux en listes triées, puis en les fusionnant pour enfin refaire un ABR. Nous avons gardé les fonctions fournies par OCaml.

Nous n'avons pas trouvé de meilleure structure pour implémenter les opérations nécessaires à cet "Union-Find-Disjoin", et c'est un des points qu'il aurait pu être intéressant d'approfondir. Nous avons aussi réfléchis à l'utilisation d'un tableau (persistant) de tableaux (persistants), mais asymptotiquement, nous conservions les mêmes complexités sur l'union, qui est bien le problème de ces inégalités.

2.2 SAT et SMT

On utilise la stratégie DPLL pour le SAT Solver. Une spécificité du module DPLL est qu'il ne renvoie pas une valuation à proprement parler, c'est à dire Vrai ou Faux pour chaque variable, mais Vrai (Aff), Faux (Ref) ou Décision (Dec). Ceci permet au solver SMT de savoir quels Union-Finds garder en mémoire. En effet, c'est toujours au niveau d'une décision que deux valuations "consécutives" vont commencer à se différencier. On a donc à garder en mémoire que les Union-Find précédant une décision.

De plus, lorsqu'une valuation ne convient pas au solver SMT, il ne fait pas recommencer le SAT-solver avec une clause supplémentaire, il lui demande simplement de backtrack jusqu'à la dernière décision qu'il a traité (puisque c'est elle qui a causé l'échec de l'Union-Find).

Le code du solver SMT peut sembler complexe. Ceci est principalement causé par le style de passage de continuation. Il semblait assez important de rendre cette fonction tail-recursive. Sa version "naïve" est beaucoup plus naturelle.