Projet de Sémantique et Vérification : Solveur *SMT*

Florentin Guth Lionel Zoubritzky

10 juin 2017

Table des matières

1 Liste des fichiers

1.1 Structures de données

- `union_find.ml` : sructure d'*union-find* classique (mutable),
- `persistent_array.ml` : implémente les tableaux non mutables de [1],
- `persistent_union_find.ml` : implémente la structure d'*union-find* non mutable de [1],
- `explain_union_find.ml` : implémente un *union-find* permettant d'expliquer pourquoi deux éléments donnés sont équivalents (voir [3]),
- `proof_union_find.ml` : *union-find* auxiliaire utilisé par l'algorithme d'explication (voir [3]).

1.2 Fichiers auxiliaires

- `lexer.mll` : lexeur pour les fichiers `.cnfuf` (améliorés avec la prise en charge des fonctions),
- `parser.mly` : parseur pour les fichiers `.cnfuf`, réalisant une curryfication et un « désenbriquement » des termes,
- `printer.ml` : fonctions d'affichages générales,
- `main.ml` : point d'entrée, lit la ligne de commande et effectue la résolution des fichiers donnés,
- `test_generator.ml` : générateur de tests aléatoires, ne gère ni les fonctions ni les formules non satisfiables,
- `Makefile` : `make all` pour compiler le solveur (`solve`) et le générateur de tests (`generate`), et `make clean` pour nettoyer les fichiers créés.

1.3 Fichiers principaux

- `sat.ml` : solveur *SAT* prenant en entrée une formule booléenne, et renvoyant un assignement le cas échéant,
- `mc.ml` : ancien *Model Checker*, non incrémental,
- `incremental_mc.ml` : *Model Checker* incrémental, mais non *backtrackable*,
- `smt.ml` : fichier principal réalisant l'interfaçage entre le solveur *SAT* et le *Model Checker*.

2 Fonctionnement général

2.1 *Model Checker*

Nous avons choisi la théorie de l'égalité, augmentée des symboles de fonction non-interprétés (d'arité quelconque).

Pour cela, on procède à une transformation sur la formule d'entrée afin de simplifier la forme des égalités :

incremental_mc.mli[7-13]

```
(** A literal can only be of two types:
    - either a variable (which can actually be a function!)
    - the application of a unique function (the application, noted 'f' here) to two variables
    Moreover, a clause is either a=b, a<>b or f(a,b)=c
    For instance, g(a,h(b),b)=b is replaced by f(f(f(g,a),f(h,b)),b)=b (currifying)
    and then by f(g,a)=c /\ f(h,b)=d /\ f(c,d)=e /\ f(e,b)=b           (flattening)
*)
```

Listing 1 – Transformation

On fournit une interface incrémentale, qui permet de plus de produire des explications pour les égalités inférées, sous forme de (petite) liste de fusions effectuées préalablement.

incremental_mc.mli[21-34]

```
(** Creates a new incremental Model Checker structure.
    Its argument is the number of variables.
*)
val create : int -> t

(** Performs the equality a=b or f(a1,a2)=b on the structure *)
val merge : t -> eq -> unit

val are_congruent : t -> var -> var -> bool

(** Returns a list of merges that entails the equality between the two variables.
    Raises Not_found if the two variables are not equal
*)
val explain : t -> var -> var -> eq list
```

Listing 2 – Interface du *Model Checker*

Cependant, cette interface incrémentale n'est de peu d'utilité, puisqu'elle n'est pas persistante. Ainsi, le *backtracking* est impossible, et il faut tout recommencer de zéro à chaque modèle trouvé par le solveur *SAT*.

Les explications permettent en revanche d'ajouter des petites clauses à la formule à chaque appel du *Model Checker*, ce qui a pour conséquence, en plus d'augmenter la vitesse du solveur *SAT*, de produire des contre-exemples plus généraux et réduit grandement le nombre d'itérations entre les deux outils.

Concrètement, les explications sont générées en stockant, en parallèle de la structure d'*union-find* classique, une structure appelée « forêt de preuve ». Celle-ci correspond à l'arbre *union-find*, mais sans compression de chemin, et en réalisant des fusions qui correspondent réellement à des égalités.

2.2 Solveur *SAT*

Le solveur *SAT* fonctionne en suivant les règles *DPLL(T)* avec quelques améliorations.

À chaque itération de la boucle principale du solveur, la règle *Unit* est appliquée autant que possible, puis le solveur vérifie si le modèle ainsi obtenu valide ou invalide la formule. Si un des deux cas est avéré, alors il s'arrête. Sinon, il choisit un littéral, selon l'heuristique VSIDS propoée par [2], dont il devine la valeur, et il s'appelle récursivement.

L'interface proposée pour la formule et son modèle est persistante, ce qui permet de backtracker facilement en cas d'échec. Cela permet aussi d'effacer de la formule les clauses qui sont validées au fur

et à mesure par le modèle. Ainsi, une clause déjà validée n'est pas explorée plusieurs fois ; elle réapparaît cependant par persistance après backtracking.

Le solveur *SAT* est entièrement incrémental : dès qu'il trouve un modèle valide, il demande au *Model Checker* une validation. Si celui-ci la lui donne, alors le problème est terminé ; sinon, il backtrack en ajoutant à sa formule l'indication du *Model Checker*, et en mettant à jour le compteur des littéraux de l'heuristique *VSIDS*.

Deux autres heuristiques ont été programmées (mais non raccordées au programme) :

— Celle notée *RAND* dans [2], consistant à prendre une littéral au hasard

— Une autre consistant à prendre le premier littéral non assigné de la clause suivante (avec sa valeur de vérité de façon à valider cette clause)

Par ailleurs, nous avons tenté de supprimer au fur et à mesure les littéraux non validés dans les clauses, de façon à ne se préoccuper que de ceux non assignés et donc de réduire le temps d'analyse de chaque clause. Cependant, le surcoût nécessaire à la réécriture de chaque clause dépassait systématiquement le gain en temps d'analyse. Il aurait pu être plus efficace d'effectuer cette opération de nettoyage des clauses une fois de temps en temps, comme une heuristique supplémentaire, mais cela ne s'est pas vérifié dans nos tests.

3 Améliorations

Le caractère extrêmement brouillon de [3] a conduit à une multiplication de structures de données tout en ayant un code peu modulaire, et dont l'efficacité et la simplicité pourraient être grandement améliorées. Les améliorations proposées dans l'article sur la suppression de la redondance dans les preuves fournies n'ont pas été implémentées.

Une grande amélioration serait d'utiliser des idées similaires à celle de [1] afin d'obtenir un *Model Checker* totalement immutable, ce qui permettrait le *backtracking* et accélererait sans doute grandement la vitesse du solveur.

Il reste également à compléter le générateur de tests afin qu'il gère toutes les configurations et produise des tests plus difficiles (i.e. des formules vraies pour peu d'assignements).

Nous n'avons pas essayé d'implémenter un solveur *SAT CDCL* afin de comparer son efficacité au solveur *DPLL*. La méthode du *two-watched literal* n'a pas non plus été implémentée, car elle se prête assez mal à la structure de liste chaînée utilisée pour représenter les clauses.

Enfin, on pourrait imaginer que dans le cas ou la formule n'est pas satisfiable, le solveur produise une preuve de ceci, vérifiable par un programme externe comme *Coq*.

Références

[1] Sylvain Conchon and Jean-Christophe Filliâtre. A persistent union-find data structure. In *Proceedings of the 2007 Workshop on Workshop on ML*, ML '07, pages 37–46, New York, NY, USA, 2007. ACM.

[2] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff : Engineering an efficient sat solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[3] Robert Nieuwenhuis and Albert Oliveras. *Proof-Producing Congruence Closure*, pages 453–468. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.