Projet 2 L3IF — Troisième rendu

à faire en binôme, à rendre pour le **dimanche 19/4/2015 à 23h59**
envoyez une archive qui compile à
`julien.bensmail@ens-lyon.fr`, `daniel.hirschkoff@ens-lyon.fr` et `vincent.lanore@ens-lyon.fr`

Ci-dessous, les ingrédients du rendu 3. Vous recevrez par mail votre "menu personnalisé".

Si vous êtes avancé et avez eu à faire CL+WL au deuxième rendu, et si ce rendu s'est bien passé, vous devrez faire Tseitin + SMT en ligne.

1 Formules logiques: transformation de Tseitin

Dans cette partie, il vous est demandé d'utiliser votre solveur SAT pour prendre en compte une formule logique quelconque.

Cela signifie qu'il vous faudra implémenter des analyseurs lexical et syntaxique afin de saisir des formules logiques quelconques, pour ensuite transformer celles-ci en formules en forme normale conjonctive.

La grammaire des formules en entrée est la suivante:

$$\mathcal{F} \quad ::= \quad \mathcal{F}_1 \vee \mathcal{F}_2 \;\big|\; \mathcal{F}_1 \wedge \mathcal{F}_2 \;\big|\; \neg\mathcal{F} \;\big|\; \mathcal{F}_1 \Rightarrow \mathcal{F}_2 \;\big|\; \texttt{x}$$

`x` désigne une *variable propositionnelle*. Concrètement, les variables seront représentées par des entiers (vous pouvez, si vous le souhaitez, réfléchir à la variante où on peut utiliser des chaînes caractères pour les variables). La conjonction et la disjonction seront notées respectivement `/\` et `\/`, l'implication `=>` et la négation `~` (bien entendu, les parenthèses peuvent aussi être utilisées pour écrire les formules).

Ainsi, , `(2\/7) => ~4 /\ 6` est un exemple de formule.

À vous de réfléchir aux priorités respectives des opérateurs.

Usage. Le programme devra pouvoir être utilisé de la façon suivante:

```
./resol -tseitin fichier.txt
```

où `fichier.txt` contiendra une formule logique au format écrit ci-dessus. La sortie devra obéir au même format que pour les problèmes SAT (`SATISFIABLE/UNSATISFIABLE`, et une affectation des variables dans le premier cas).

De plus, votre programme devra pouvoir afficher la formule en forme normale conjonctive obtenue à partir d'une formule logique en entrée (affichage au format d'entrée de DPLL).

Jeux de tests. Il est indispensable que vous construisiez vous-mêmes des formules pour tester votre solveur.

Au-delà de l'écriture de formules particulières, pour vérifier que les divers connecteurs logiques et leurs combinaisons sont bien traités, vous pourrez fabriquer des exemples plus larges à partir d'exemples SAT, en replaçant (par exemple)

$$F_1 \vee F_2 \;\text{ par }\; \overline{F_1} \Rightarrow F_2 \qquad \text{et} \qquad (F_1 \vee F_2) \wedge F_3 \;\text{ par }\; (F_1 \wedge F_3) \vee (F_2 \wedge F_3)\ .$$

2 SMT

2.1 Considérations en facteur pour les gens qui traiteront du SMT

Formats d'entrée. Les formules prises en entrée par votre solveur SMT seront construites avec les connecteurs logiques habituels : cf. partie précédente. De plus, s'agissant des théories:

- Pour la congruence et l'égalité, les notations habituelles pour les termes. La *diségalité* sera notée `!=`, l'égalité sera notée `=`.

- Pour la logique de différence, il faudra que vous suiviez la grammaire ci-dessous, en convertissant ensuite en une "forme canonique", quitte à engendrer une erreur si le problème ne peut pas se mettre sous une forme correspondant à la logique de différences.

Des **grammaires**, un peu plus formellement: les *atomes* sont les A pour la congruence, les A sans le $f(t_1, \ldots, t_k)$ pour l'égalité (les symboles de fonction ne peuvent pas commencer par `x`), et les A' pour la logique de différences.

$$F \quad ::= \quad A \mid \texttt{~}F \mid F_1 \texttt{/\textbackslash} F_2 \mid F_1 \texttt{\textbackslash/} F_2 \mid F_1 \texttt{=>} F_2 \mid (F)$$

$$A \quad ::= \quad t_1 \texttt{=} t_2 \mid t_1 \texttt{!=} t_2 \qquad\qquad t \quad ::= \quad \texttt{x}i \mid \texttt{f(}t_1, \ldots, t_k\texttt{)}$$

$$A' \quad ::= \quad \texttt{x}i - \texttt{x}j \ op \ n \mid \texttt{x}i \ op \ n \qquad\qquad op \quad ::= \quad \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{=} \mid \texttt{!=}$$

Jeux de tests. Il ne s'agit pas pour les solveurs SMT que vous écrirez de comparer leur efficacité avec je ne sais quel outil développé dans le monde de la recherche.

En revanche, il vous faudra développer des jeux de tests pas trop naïfs, histoire de pouvoir s'assurer du fait que l'on explore en profondeur le "dialogue" entre solveur SAT et solveur pour une théorie.

Vous expliquerez dans le README comment vous avez conçu ces tests, et dans quelle mesure vous pensez qu'ils sont pertinents au vu de ce qui est suggéré ci-dessus.

Modularité. Que vous soyez paresseux ou en ligne (cf. ci-dessous), vous aurez plus d'une théorie à traiter. Réfléchissez donc à la structuration de votre code, pour que la partie "T" soit un constituant du code bien délimité, qui puisse être instancié de plusieurs manières suivant la théorie traitée.

2.2 SMT light : paresseux hors ligne

On vous demande de coder un solveur SMT hors ligne : le solveur pour la théorie T est une "boîte noire", qui est juste capable de dire s'il est content ou pas.

Dans cette version, le solveur SAT peut être avec ou sans apprentissage de clauses.

Il faudra adapter votre algorithme DPLL pour pouvoir apprendre une nouvelle clause lorsque le solveur est mécontent. Vous apprendrez la clause naïve exprimant que l'état courant est inconsistant... tout du moins dans un premier temps : vous êtes les bienvenus si vous voulez travailler à la construction d'une formule un tant soit peu moins bête.

Il vous est demandé de traiter deux théories:

1. l'égalité (sans congruence);
2. la logique de différences.

2.3 SMT en ligne, DPLL(T)

On vous demande de programmer DPLL(T), avec trois déclinaisons pour la théorie:

1. la théorie des égalités;
2. la théorie de la congruence (qui enrichit la précédente);
3. la logique de différences.

Il s'agit donc de programmer une version paresseuse, en ligne, de SMT.

Il vous faudra vous attacher à ce que les solveurs de théorie soient:

- incrémentaux, pour pouvoir faire de la détection précoce d'inconsistances;

- capables de faire du backtrack, du coup, en suivant le backtrack de DPLL;
- capables de fournir une explication pertinente lors de la détection d'inconsistance, afin d'"alimenter" l'apprentissage de clauses de DPLL;
- *si possible* (mais cela n'est pas obligatoire), capables d'aider dans la phase de propagation de DPLL, en indiquant des valeurs de littéraux pouvant être déduites, et en indiquant quels littéraux contribuent à cette déduction.

Dans le README, vous expliquerez, dans chaque cas:

- comment est implémentée la structure de données sur laquelle s'appuie chaque solveur de théorie;
- comment sont implémentés les divers mécanismes nécessaires au dialogue avec DPLL (incrémentalité, backtrack, explications..). Même si vous n'avez pas trouvé de solution particulièrement astucieuse ou inattendue, expliquez ce qui se passe.

Si possible, vous ferez également une moulinette prenant en entrée un problème d'ordonnancement (décrit à un format qui sera spécifié par vos soins), et produisant un problème pour la version "logique de différences" de SMT (cf. esquisse d'exemple dans les transparents du cours).

3 Bonus

3.1 Utiliser les clauses apprises pour diriger les paris

On présente une heuristique pour choisir le prochain pari, en présence d'apprentissage de clauses : VSIDS *(Variable State Independent Decaying Sum)*.

On associe à chaque littéral un *score*, qui augmente à chaque fois qu'il apparaît dans une clause apprise (variante : à chaque fois qu'il est manipulé dans l'analyse de conflit).

On divise régulièrement par une constante (par exemple 2), pour privilégier l'activité récente.

Lorsque l'on doit faire un pari, on choisit le littéral non affecté dont l'activité est la plus grande.

4 Une "check list" pour votre rendu

N'oubliez pas les éléments du README qui sont précisés plus haut dans cet énoncé (on estime que vous savez désormais ce qui est attendu dans un README).