Logique : implémentation de la méthode de résolution

Julie Nix & Raphael Javaux

Année académique 2013-2014

Introduction

Nous avons implémenté la méthode de résolution à l'aide du langage fonctionnel Haskell. Nous avons utilisé la librairie Parsec pour réaliser notre parseur.

Le principal intérêt d'implémenter notre solution en Haskell a été la possibilité d'utiliser le *pattern matching* dans nos équations de résolution. Par exemple, l'étape de suppression des doubles négations s'écrit très concisement :

```
doubleNot (Not (Not p)) = doubleNot p
doubleNot (And p q)     = And (doubleNot p) (doubleNot q)
doubleNot (Or  p q)     = Or  (doubleNot p) (doubleNot q)
doubleNot p             = p
```

La possibilité d'utiliser la notation « mathématique » des ensembles lorsque nous travaillions sur les listes s'est avérée également très élégante. Ainsi, l'énumération de l'ensemble des dérivées dérivables à partir d'une liste de clauses peut s'écrire de cette manière :

```
[ c' | c1 <- clauses, c1 <- clauses, lit1 <- c1
     , let lit2 = complement lit1
     , lit2 `S.member` c2
     , let c' = S.delete lit1 c1 `S.union` S.delete lit2 c2
     ]
```

Ce qui correspond directement à la notation mathématique :

```
{ c' | c1 ∈ clauses, c2 ∈ clauses, lit1 ∈ c1
     , soit lit2 le littéral complémentaire de lit1,
     , lit2 est dans c2
     , soit c' = (c1 \ lit1) ∪ (c2 \ lit2)
     }
```

L'exécutable peut être compilé sur une machine ayant le compilateur GHC installé à l'aide du `Makefile` à la racine de notre projet[1]. Le projet et sa compilation ont été testés sur les machines `ms8xx`.
L'exécutable peut ensuite être appellé de cette manière :

```
./resolution ex1.txt ex1.sol
```

Nous avons particulièrement travaillé sur l'algorithme de normalisation. Celui-ci essaye de simplifier au maximum la CNF de telle manière que l'algorithme de résolution s'effectue sur le minimum de clauses possible.

[1] Un certain nombre de *warnings* sont émis par le compilateur lors de la compilation. Ceux-ci sont uniquement dûs aux bibliothèques. En exécutant la commande `touch Resolution.hs && make`, seul le fichier de notre projet sera recompilé et aucun *warning* ne sera émis.

Parseur

Le parseur de notre projet utilise la librairie Parsec. Parsec est une librairie de parseurs combinables. Elle fournit un ensemble de parseurs de bases pour parser des chaines de caractères (`string`), des caractères (`char`), une lettre quelconque (`letter`), un caractère alphanumérique (`alpha`) … et un ensemble de combinateurs pour séquencer ces parseurs (`>>` ou un retour à la ligne), expresser l'alternative (`<|>`), la répétition (`many`) … Ces combinateurs créent de nouveaux parseurs pouvant eux-mêmes être combinés à nouveau à l'aide de ces mêmes combinateurs.

Par exemple, le parseur suivant permet de reconnaitre un atome :

```
atom =     between (char '(') (char ')') expr
       <|> (char '~' >> (Not <$> atom))
       <|> (try (string "$true") >> return (Val True))
       <|> (     string "$false" >> return (Val False))
       <|> variable
```

Dans l'exemple précédent, `try` permet de définir un « point de backtracking » : les deux symboles `$true` et `$false` ne peuvent être distingués par leur premier caractère et il est nécessaire pour Parsec d'« entrer » dans le parseur `string "$true"` lors de la lecture du symbole `$false` pour le distinguer du symbole `$true`.

Le parseur convertit immédiatement les implications et les équivalences en disjonctions et conjonctions. Cela simplifie l'arbre syntaxique.

Normalisation

Simplification de la formule

Une simplification des formules est effectuée avant la phase de normalisation. Cette simplification est particulièrement importante car elle fait remonter les formules `true` et `false` à la racine de l'arbre syntaxique. L'arbre syntaxique transmis à l'algorithme de normalisation ne contient donc plus de formules `true` ou `false`, à l'éventuelle exception de sa racine, ce qui facilite le traitement de ces formules particulières.

Normalisation

L'algorithme de normalisation applique les règles de distribution récursivement, en propageant les disjonctions vers le bas de l'arbre syntaxique à chaque application d'une de ces deux règles.

L'algorithme s'exécute avec un « cache » retenant les distributions déjà effectuées. Ceci permet de repérer les sous-formules récurrentes et de ne les traiter qu'une seule fois, au coup d'une utilisation mémoire un peu plus importante.

Ce repérage des formules identiques reste cependant limité, par exemple les sous-formules `a | (b | c)` et `(b | a) | c` ne seront pas reconnues comme identiques. Une amélioration de l'algorithme serait de le faire travailler sur des opérateurs n-aires et non plus binaires et d'utiliser une représentation « ordonnée » pour corriger ce problème[2].

Simplification de la CNF

Une fois la formule en CNF, nous éliminons d'abord les clauses valides (contenant un littéral et son opposé, comme `s | ~r | ~s`) car elles sont des tautologies. Il est possible que toutes les clauses de la formule soient des tautologies et dans ce cas la CNF est également une tautologie.

Ensuite, nous recherchons des clauses composées d'un seul littéral (clauses unitaires). Nous savons que le complément de ce littéral ne saurait être satisfait et de ce fait, nous pouvons le supprimer de toutes les clauses qui le contiennent. Par exemple, la CNF `p & (q | r | ~p)` peut être simplifiée en `p & (q | r)`.
Si de nouvelles clauses unitaires ont été créées ainsi, on ré-itère l'opération sur celles-ci. Il se peut également que, lors de cette étape de simplification, la clause vide soit dérivé (pour la CNF `p & ~p` par exemple). Dans ce cas, la CNF simplifiée sera tout simplement la clause vide, et aucune étape de résolution ne sera nécessaire.

Finalement, les clauses récurentes (dont il existe une seconde clause composée d'un sous-ensemble de leurs littéraux) sont également supprimées. Par exemple `(p | q) & (p | q | r)` peut être simplifiée en `p | q`. Cette phase de simplification doit être effectuée après la phase précédente car cette dernière pourrait faire apparaître de nouvelles récurrences entre les clauses.

Ces trois phases de simplification permettent généralement de supprimer un grand nombre de clauses et/ou littéraux inutiles de la CNF.
Par exemple, la CNF non simplifiée de l'exercice 2 de l'énoncé est la suivante :
`(p | ~r | s) & (q | ~r | s) & (~p | ~q) & (~p | r) & (~p | ~s) & (r | ~q) & r & (r | ~s) & (~s | ~q) & ~s`
Aucune clause n'est une tautologie, la première étape de simplification ne supprime donc aucune clause.
La formule contient deux clauses unitaires, `r` et `~s`. En supprimant les compléments de ces deux littéraux dans les deux premières clauses, nous obtenons :
`p & q & (~p | ~q) & (~p | r) & (~p | ~s) & (r | ~q) & r & (~s | ~q) & (~s | r) & ~s`
On peut à nouveau effectuer l'opération sur la nouvelle clause unitaire `p` :

[2] Les règles de distribution devraient également être généralisées de cette manière :

$$(p_1 \,\&\, p_2 \,\&\, \dots \,\&\, p_N) \mid (q_1 \,\&\, q_2 \,\&\, \dots \,\&\, q_N) \Leftrightarrow \begin{aligned} &(p_1 \mid q_1) \,\&\, (p_1 \mid q_2) \,\&\, \dots \,\&\, (p_1 \mid q_N) \\ &\&\, (p_2 \mid q_1) \,\&\, (p_2 \mid q_2) \,\&\, \dots \,\&\, (p_2 \mid q_N) \\ &\dots \\ &\&\, (p_N \mid q_1) \,\&\, (p_N \mid q_2) \,\&\, \dots \,\&\, (p_N \mid q_N) \end{aligned}$$

`p & q & ~q & (r | ~q) & r & (~s | ~q) & (~s | r) & ~s`
A présent, si nous tentons de propager `q`, nous alons créer une clause vide en supprimant `~q`. La formule est donc tout simplement la clause vide, et aucune étape de résolution ne sera nécessaire pour déterminer sa consistance.

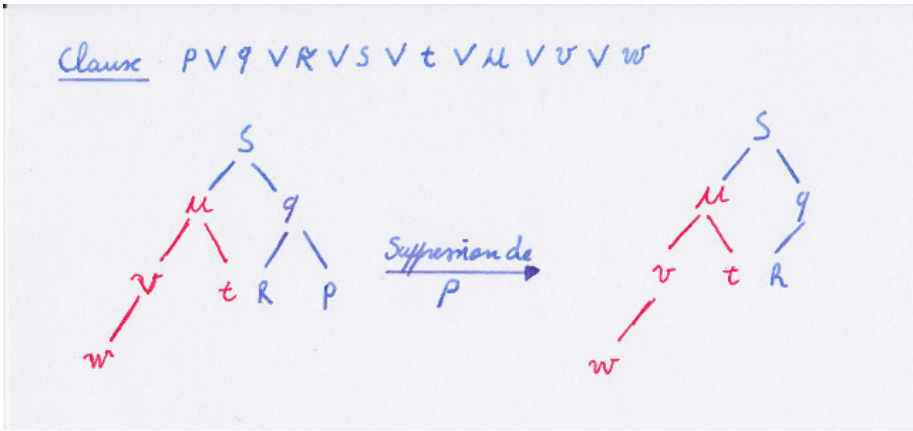
Représentation des clauses

Pour stocker les littéraux des clauses générées par l'algorithme de normalisation, nous avons opté pour `Set` de la librairie standard d'Haskell, une structure en arbre binaire équilibré, où chaque noeud de l'arbre contenant un littéral.

Cette structuration nous permet d'effectuer les opérations nécessaires à la normalisation (recherche, insertion et suppression de littéral, détection des sous-clauses et des clauses valides) et à la résolution (recherche et suppression de littéral et union de deux clauses) de manière efficace :

Opération	**Complexité** **(n est le nombre de littéreaux des clauses)**
Recherche, insertion ou suppression d'un littéral dans une clause	`O(log n)`
Union de deux clauses a est-elle un sous-clause de b ?	$O(n)^3$
a est-elle une clause valide ?	`O(n log n)`

De plus, l'arbre est une structure immuable. Ajouter ou supprimer un littéral à une clause ne modifie pas cette dernière mais retourne une nouvelle clause. Ceci est particulièrement pratique lors de la phase de résolution où de nombreuses clauses sont crées de cette façon.
Cela peut paraitre à première vue inefficace, mais en pratique, la majorité des sous-arbres peuvent être partagés par deux arbres :



[3] L'algorithme utilisé pour résoudre ces problèmes est similaire à celui de la fusion du merge-sort.

Sur le schéma précédent, le sous-arbre rouge est partagé entre l'arbre original et l'arbre auquel le prédicat p a été supprimé. Le nombre de noeuds modifiés entre deux versions d'un même arbre sera toujours proportionnel à la hauteur de cet arbre, c'est à dire au logarithme de deux de son nombre de littéraux.

Résolution

Suppression des clauses pures

La phase de résolution commence par la détection des littéraux purs — dont aucun complément ne peut être trouvé dans aucune clause — afin de supprimer toutes les clauses qui les contiennent. Ces clauses ne seront en aucun cas « combinables » lors de la résolution et ne sauraient donc aboutir à une clause vide.

Fermeture

Chaque étape de la fermeture est effectuée en tentant de combiner les clauses dérivées à l'étape précédente avec l'ensemble des clauses connues[4]. Il est inutile de tenter de combiner les clauses plus anciennes avec elles-mêmes car ceci a déjà été fait aux étapes précédentes.
Les dérivées valides sont ignorées comme celles-ci ne sauraient dériver vers la clause vide.

Si l'itération de la fermeture n'a pas abouti à une clause vide et que de nouvelles dérivées ont été trouvées, alors une nouvelle itération est tentée avec ces dernières.

Chaque dérivée trouvée contient des références vers ses deux clauses parentes, de telle manière à pouvoir reconstruire la réfutation.

[4] A l'exception de la première itération de l'algorithme où toutes les clauses de la CNF sont testées avec elles-mêmes.