UTC - CNRS

Méthode B - Expressions régulières et automates

Spécification formelle d'un nouvel opérateur

Laurent WOUTERS

21/03/2008

L'objectif du projet est d'étendre le langage des expressions régulières en implémentant un nouvel opérateur. La méthode B étant utilisée pour spécifier et définir la sémantique de cet opérateur ainsi que son implémentation.

On sait que les langages réguliers peuvent être exprimés sous forme d'expressions régulières ainsi que sous forme d'automates à état finis. Dans un outil informatique tel qu'un générateur de lexer les opérations de bases sont la transformation d'une expression régulière en automate fini non déterministe puis la transformation de ce dernier en automate fini déterministe. (1) Le générateur de lexer part de l'ensemble des expressions régulières pour en former une générale avec plusieurs états finaux capable de reconnaitre tous les terminaux d'une grammaire d'un langage par exemple.

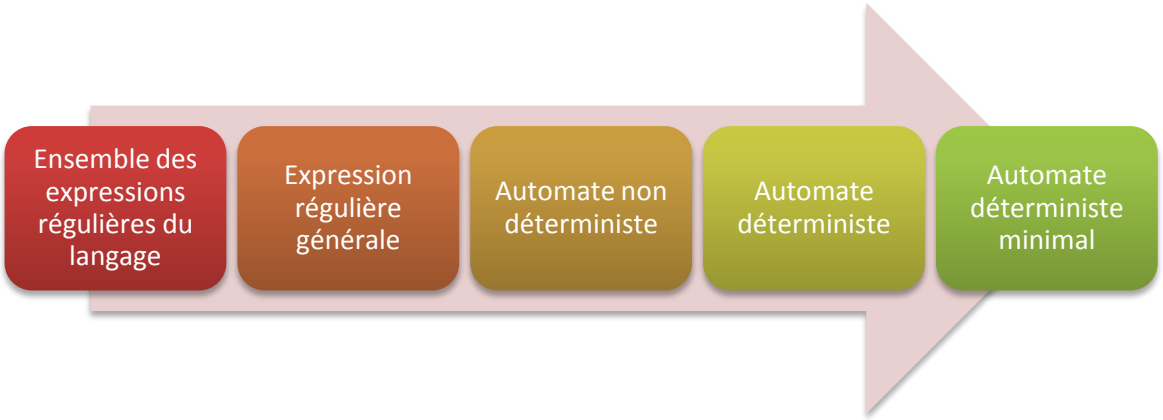

Figure 1 - Générateur de lexer : transformation des données

L'automate déterministe minimal peut alors avec un peu de liant faire office de lexer.

On propose ici d'étendre le langage des expressions régulières pour introduire un nouvel opérateur, l'opérateur de restriction. Un opérateur équivalent existe déjà dans la théorie des ensembles : la différence d'ensembles. On peut donc en particulier écrire $L_1 \setminus L_2$ si L_1 et L_2 sont respectivement des langages. $L_1 \setminus L_2$ désigne l'ensemble des éléments de L_1, excepté ceux appartenant aussi à L_2.

Approche intuitive

La sémantique de l'opérateur de restriction est définie comme suit :

- ➔ Soit exp_1 une expression régulière et $L(exp_1)$ le langage décrit par exp_1.
- ➔ Soit exp_2 une expression régulière et $L(exp_2)$ le langage décrit par exp_2.

Alors $L(exp_1 – exp_2) = L(exp_1) \setminus L(exp_2)$ est l'ensemble des chaînes de $L(exp_1)$ privé de l'ensemble des chaînes de $L(exp_2)$ (zone bleue sur la figure ci-dessous). C'est aussi le complémentaire de $L(exp_1) \cap L(exp_2)$ dans $L(exp_1)$.

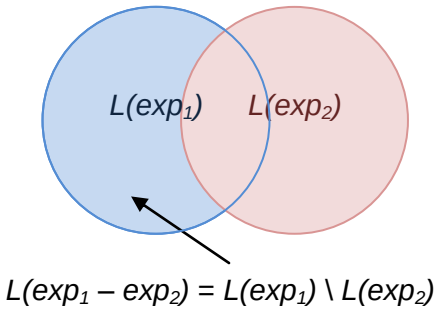

Figure 2 - Opérateur de restriction sur les ensembles

On sait construire un automate non déterministe à partir de l'arbre syntaxique de l'expression régulière associée. En particulier, l'opérateur d'union | peut se représenter ainsi : pour l'expression exp_1 | exp_2 on obtient l'automate suivant :

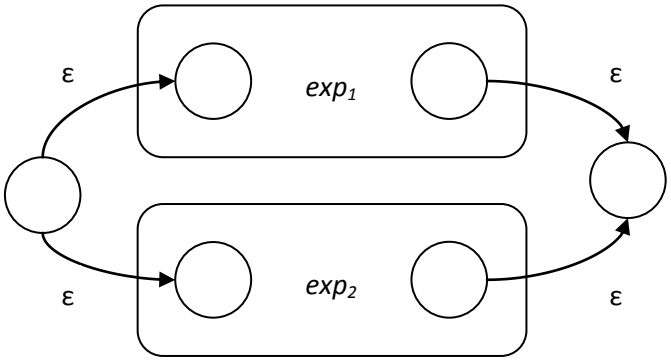

Figure 3 - Représentation de l'opérateur d'union

Pour représenter l'opérateur de restriction, on utilise quasiment la même méthode que pour l'union, mais on utilise des marques comme suit. Une marque positive est ajoutée sur l'état final du sous-automate représentant exp_1 et une marque négative est ajoutée sur l'état final du sous automate représentant exp_2. Pour l'expression exp_1 - exp_2 on définie que l'automate non déterministe associé est le suivant :

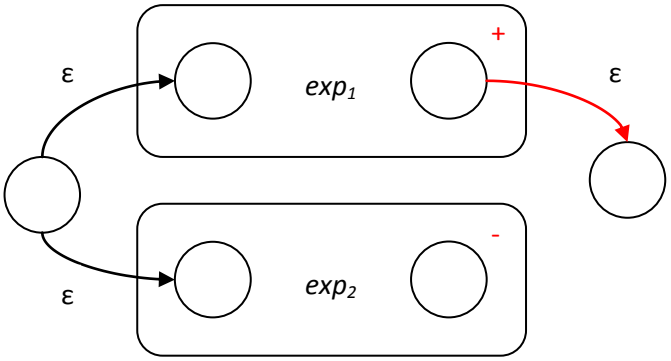

Figure 4 - Représentation de l'opérateur de restriction

En ce qui concerne la transformation en automate déterministe, intuitivement, lors de construction de l'automate équivalent on déterminera les chaînes reconnues à la fois par exp_1 et exp_2. Lorsqu'une telle chaîne est découverte il suffira d'interdire à l'algorithme l'epsilon-transition (en rouge) vers l'état final. On autorisera cette transition que si on ne peut accéder qu'à l'état marqué positivement, c'est-à-dire que seule exp_1 est reconnue. A l'inverse si seule exp_2 *est* reconnue, l'algorithme ne fera rien de toute façon puisque l'état final de exp_2 n'a pas de transitions sortantes.

Pour éviter les conflits entre les marques représentant différents opérateurs de restriction dans une même expression régulière, on peut associer à chacun un couple de valeurs opposées différent.

Spécification formelle

Afin de réaliser plus tard l'implémentation de ce nouvel opérateur dans un générateur de lexer on commence par spécifier de manière formelle les objets et méthodes. Ainsi il convient en particulier de spécifier les automates non déterministe et déterministe. Enfin comme le point crucial du processus ici est la conversion du l'automate non-déterministe en automate déterministe, on écrit une spécification pour cet algorithme.

Les automates spécifiés ici sont un peu particuliers comme les valeurs associées aux transitions sont un peu plus complexes. Au lieu de n'utiliser que de simples symboles comme valeurs pour les transitions des automates on utilise ici des ensembles de symboles. Ces ensembles seront en fait des intervalles. En effet, en considérant que chaque symbole est associé à un nombre entier différent (qui est en fait la représentation binaire du symbole dans un encodage particulier) il est possible d'ordonner l'ensemble des symboles.

L'utilisation d'intervalles amène un certain nombre de problème, dont la définition de ce qu'est un automate déterministe avec ce type de transition. On considère que dans un automate déterministe, pour chaque état les valeurs des transitions sortantes ont des intersections nulles deux à deux. Cela implique qu'au cours de la transformation, d'un automate non déterministe vers un automate déterministe une « normalisation » des transitions des états interviennent afin de correspondre à cette description.

Les spécifications des éléments communs aux deux types d'automates ont été regroupées dans un autre composant. Les spécifications sont en annexes.

- ➔ FA.mch contient les spécifications des ensembles communs aux 2 automates.
- ➔ NFA.mch contient la spécification de l'automate non déterministe
- ➔ DFA.mch contient la spécification de l'automate déterministe.

Implémentation

On suppose ici qu'un générateur de lexer converti d'abord un ensemble d'expressions régulières vers un automate fini non-déterministe à epsilon-transition puis ce dernier vers une automate fini déterministe. On suppose également que le nouvel opérateur de restriction est représenté dans l'automate non déterministe par deux marques opposées sur deux états différents, comme décrit précédemment.

Le point important de l'implémentation concerne la conversion d'un automate non-déterministe en automate déterministe. On peut ici utiliser les algorithmes standards à l'exception de celui de l'epsilon-fermeture d'un ensemble d'état. Pour implémenter le nouvel opérateur il faut alors utiliser l'algorithme 2 utilisant les marques au lieu de l'algorithme 1. De plus, l'algorithme 2 utilise l'algorithme standard d'epsilon-fermeture rappelé dans l'algorithme 1. Dans une implémentation dans un langage particulier il faut donc retranscrire les 2 algorithmes.

Algorithme 1 : **ε-fermeture(Q)** (2 p. 35)

```
Pour tout e ∈ Q
     Empiler e dans la pile P
Initialiser ε-fermeture(Q) avec Q
Tant que P est non vide
     Dépiler P dans s
     Pour chaque état q tel qu'il existe un arc ε entre s et q
          Si q n'est pas dans l'ε-fermeture(Q)
               Ajouter q dans l'ε-fermeture(Q)
               Empiler q
```

Algorithme 2 : **ε-fermeture avec marques(Q)**

```
Initialiser ε-fermeture avec marques(Q) avec ε-fermeture(Q)
Tant qu'il existe un couple d'états (p, n) de l'ε-fermeture avec
marques(Q) tel que p est marqué par une marque positive et n est
marqué par la marque opposée.
     Calculer C l'ε-fermeture({p, n})
     Enlever les états de C à l'ε-fermeture avec marques(Q)
```

Exemple d'application

Soit l'expression régulière suivante :

```
RES         = ('a' | 'b')* - ('aaaa' | 'bbbb') ;
```

Cette expression régulière reconnait toutes les chaînes composées de a et de b sauf les chaînes de 4 'a' ou de 4 'b' consécutifs. L'expression régulière reconnaît bien les chaînes de 5 'a' ou de 5 'b' ou plus consécutifs.

On voit ici qu'il serait complexe d'écrire une expression régulière équivalente sans utiliser l'opérateur de restriction (bien que cela soit faisable). Si les opérandes deviennent un peu plus complexe il devient rapidement impossible en pratique d'écrire de réécrire l'expression sans l'aide de l'opérateur.

```
DATA        = [a-zA-Z]* - ([a-zA-Z]* 'this' [a-zA-Z]*) ;
```

L'expression régulière DATA reconnait toutes les expressions constituées de lettres latines majuscules ou minuscules, d'une longueur quelconque éventuellement nulle, à l'exception celles contenant l'expression 'this'. Ecrire une expression équivalente serait difficile en pratique.

Retour d'expérience sur la méthode B

La technologie B

La méthode B permet de construire des systèmes sûrs par la formalisation du cahier des charges techniques en spécification formelle puis par le raffinement progressif et prouvé de ces dernières vers le système exécutable. Incidemment, la méthode permet d'obtenir un système conforme aux spécifications et donc aux attentes du client dès lors que le passage des spécifications informelles aux spécifications formelles a été fait avec soin. La méthode B est donc un outil puissant dont on peut n'utiliser qu'une partie, par exemple la formalisation des spécifications.

Même si le bagage mathématique requis n'est pas très important la méthode demande un réel investissement pour être vraiment bien utilisée (ce qui peut passer par une formation). En effet, l'utilisation d'un langage très proche des mathématiques pour décrire des systèmes n'est que peu naturel, même si le langage B met à disposition une certaine couche de sucre syntaxique le rendant ainsi plus accessible pour les profanes.

De plus, le langage gagnerait à être modernisé. Il est ainsi peu logique de séparer la déclaration des variables de leurs contraintes de typage ; ce qui oblige à une certaine gymnastique, au moins musculaire, si ce n'est intellectuelle. De plus, les règles de raffinement ne sont pas toujours claires (et de loin).

D'une manière générale il est dommage que l'on ne trouve quasiment pas de documentation en ligne (quelques mauvais tutoriaux ou exemples). La documentation de Clearsy sur le langage est à l'image de la technologie : formelle.

L'outil Atelier B

L'Atelier B est le logiciel produit par Clearsy permettant d'utiliser industriellement la méthode B. Il est indéniable que l'utilisation du logiciel ne laisse pas indifférent par son côté un peu « old school ».

Concernant la partie interface, on peut apprécier son efficacité/austérité dans la gestion des projets (les contrôles se comptent sur les doigts des deux mains). Mais on pourrait regretter le manque d'information que fournie le logiciel sur les composants des projets. De plus la manière de les présenter (une liste) peut rendre l'exploration assez difficile. A contrario lors d'une preuve interactive le fait d'avoir 3 fenêtres différentes à afficher et lire en même temps peut être difficile à gérer. On peut également regretter que le logiciel se limite à un composant par fichier physique. Dans le cas d'une petite machine il aurait été judicieux d'autoriser de mettre dans le même fichier la spécification, les raffinements et l'implémentation. Cela marche peut être chez les autres mais chez moi ca ne marche pas, bien que j'ai suivi ce qui était marqué dans la documentation.

Un gros point négatif est quand même à mon avis le générateur d'obligation de preuve qui génère des expressions illisibles avec des noms de variable incompréhensibles. De plus il n'est pas possible de visualiser les obligations de preuve dans les fichiers générés.

Quant au prouveur, il se révèle efficace dans l'ensemble. On regrette par contre les noms de commandes qui paraissent assez ésotériques au début (une fois que l'on a compris la logique c'est bon). Par contre, l'utilisation se fait parfois difficile, notamment comme il est impossible de copier-coller du texte dans l'invite de commande de l'interface de preuve interactive. La base de règle du prouveur pourrait également être étendue. Elle ne contient par exemple aucune règle concernant les lambda-expressions.

En fait on aperçoit ici les limites de ce que peut faire un industriel seul sur ce genre de produit. Il pourrait être bénéfique qu'un comité scientifique maintienne de manière régulière le prouveur et sa base de règle afin de les compléter. C'est ce qui est fait sur d'autres technologies/langages comme C++, Java, C#.

De même il serait je pense assez profitable de disposer d'un véritable éditeur de texte intégré à l'Atelier, avec une coloration syntaxique et pourquoi pas une auto-complétion, et des snippets pour des éléments de langage comme les opérateurs 'pour tous' et 'il existe'. Dans le même ordre d'idée, le langage pourrait être avantageusement modifié pour le clarifier.

Conclusion

L'opérateur de restriction dans l'algèbre des expressions régulières a bien pu être défini avec la méthode B. Il a également été possible de l'implémenter dans un générateur de parser suivant les spécifications. Ce nouvel opérateur se révèle utile pour écrire des expressions complexes.

Concernant l'utilisation de la méthode B, l'expérience est intéressante mais je crois que la méthode est trop lourde à mettre en œuvre pour qu'elle soit rentable dans une utilisation complète dans un projet dont le produit est non critique. Toutefois, l'utilisation de la partie formalisation des spécifications pourrait être plus largement répandue.

Bibliographie

1. **AHO, Alfred V., et al.** *Compilers: Principles, Techniques, and Tools (2nd Edition).* s.l. : Addison Wesley, 2006. 978-0321486813.

2. **MOULIN, Claude.** *Théorie des Langages.* Compiègne : BUTC, 2007.