Rapport Projet
Programmation Fonctionnelle - Traduction des langages
Compilateur du langage RAT étendu

ROUX Thibault
SADURNI Thomas
FAN Yanghai

Département Sciences du Numérique - Filière Image et Multimédia
2020-2021

Contents

1 Introduction

Durant les séances de TPs, nous avons implémenté un compilateur du langage RAT basique et écrit en OCaml. Nous allons ici ajouter des extensions à ce langage : les pointeurs, la surcharge de fonctions, les types énumérés et enfin la structure de contrôle switch/case. Nous aborderons les choix de conception et les ajouts que nous avons faits au langage et à l'AST pour ces extensions.

Contenu des fichiers :

1. ast.ml contient deux interfaces. Une donnant la structure générale des arbres créés par les 4 passes différentes : TDS, Type, Placement, et Code, l'autre donnant l'affichage de ces arbres. Il contient aussi des modules comportant les définitions du type des AST.
2. lexer.mll réalise l'analyse lexicale.
3. type.ml contient les types utilisés durant les TPs et ceux ajoutés, notamment pour la partie pointeur et énumération.
4. parser.mly réalise l'analyse syntaxique et la construction de l'arbre abstrait
5. Les fichiers "passe*.ml" contiennent les codes des différentes passes.
6. Les fichiers "test*.ml" contiennent les tests des passes ainsi que ceux des extensions que nous avons ajoutées.

2 Extension du langage RAT

2.1 Les pointeurs

L'intégration des pointeurs a été detaillée pendant les séances de TD. Une variable est identifiée par son adresse qui contient sa valeur. Un pointeur est une variable qui contient l'adresse. Nous avons donc, comme le suggérait le TD, créé des affectables dans l'AST. Pour cela nous avons ajouté un nouveau type $Pointeur\ of\ typ$, de nouvelles expressions : New, Adresse, Null et Affectable. Ce dernier concerne un accès sur un affectable, qui est un type ajouté et qui comprend Ident et Valeur.

Le traitement des affectables est différent dans le cas d'une lecture ou d'une écriture dans la passe de TDS, d'où le paramètre $modif$ dans la fonction $analyse\_tds\_affectable$. Nous ne détaillerons pas plus ici car la majeure partie des pointeurs a été vue en TD et en TP (notamment la modification d'$Affectation$).

En ce qui concerne l'affichage des pointeurs, nous avons simplement décidé d'utiliser AffichageInt(), on aurait pu ajouter une fonction qui ferait la même chose en concaténant un caractère @ au début de l'entier pour bien discerner le fait qu'il s'agit d'une adresse.

Pour la passe de placement, nous n'avons rien modifié, nous avons seulement défini la taille du type pointeur à 1.

Enfin pour la génération de code en TAM, nous avons créé plusieurs fonctions pour la lecture ou l'écriture. On retrouve le code de l'affectation à gauche ou à droite, ainsi que $analyse\_code\_ident\_val\_affectable$ qui charge les valeurs.

Jugements de typage

1. Expression de l'adresse $null$: $\sigma \vdash null : Pointeur(Undefined)$
2. Expression de l'accès à l'adresse d'un $Ident$: $\frac{\sigma \vdash id : t}{\sigma \vdash \&id : Pointeur(t)}$
3. Expression d'allocation de mémoire dans le tas New : $\frac{\sigma \vdash t : \tau}{\sigma \vdash new\ t : Pointeur(t)}$
4. Affectable accédé en valeur $*a$: $\frac{\sigma \vdash a : Pointeur(t)}{\sigma \vdash *a : t}$

Les tests réalisés se trouvent dans le dossier $rat - pointeur - test$.

2.2 La surcharge de fonctions

Afin de traiter la surcharge de fonctions, nous avons décidé de modifier l'InfoFun en remplacant la liste des paramètres en liste de liste des parametres de toutes les fonctions ayant le même nom. Notre InfoFun se decrit maintenant de la manière suivante : $InfoFun\ of\ string\ *\ typ\ *\ (typ\ list)\ list$.

Dans la passe de TDS, nous modifions $analyse\_tds\_fonction$ pour pouvoir ajouter tous les paramètres de la fonction surchargée dans la liste de liste ainsi que dans l'arbre. La double déclaration d'une fonction n'est donc plus possible, en revanche nous n'acception spas la surcharge sur le type de retour. Si on avait du faire ceci, nous aurions du faire des modifications au niveau des appels de fonctions, avec un type différent en fonction de l'appel.

Lors de la passe de typage, lorsque nous appelons une fonction dans $AppelFonction$, nous regardons si les types appélés sont compatibles avec une des listes des paramètres stockés. Pour cela, nous implantons la fonction $est\_compatible\_infofun$. Dans l'analyse de la fonction, nous regardons si la fonction en entrée ne contient pas les mêmes parametres que celles du même nom déja stockées. Dans le cas de deux fonctions déclarées avec le même nom et les mêmes paramètres, nous levons l'exception $DoubleDeclaration$, sinon, on ajoute la liste des paramètres à la liste de liste d'InfoFun.

Pour la passe de génération de code, nous avons choisi de changer le nom des fonctions en fonction de leur type. Par exemple, une fonction f avec comme paramètres un entier et un booléen sera nommée f_int_bool. Le plus gros changement de code se trouve dans l'appel de la fonction. En effet, comme nous avons choisi de renommer le nom des fonctions, il faut pouvoir analyser chaque type de paramètre et renvoyer le type général. Nous avons conscience que le code de $AppelFonction$ est un peu "lourd", il aurait été judicieux de faire une fonction pour cela.

Les tests réalisés se trouvent dans le dossier $rat - surcharge - test$.

2.3 Les types énumérés

Nous ajoutons ici les types énumérés au langage RAT étendu. Les identifiants de ces types commencent par une lettre majuscule. Pour cela nous allons dans *lexer.mll* pour ajouter un token TID basé sur le même principe que l'$Ident$. Nous en profitons pour ajouter les tokens de la grammaire donnée dans le sujet : $ENUM$ et $VIRG$ pour la virgule.
Avec les types énumérés, le programme principale est modifié, en effet il faut prendre en compte qu'avant la déclaration des fonctions, il y a celle des types énumérés. Pour cela nous modifions le *main* dans le *parser.mly* pour y ajouter les *enums*. Nous nommons *enums* la liste des différentes énumérations et *enum* une seule énumération, composée du nouveau type (TID dans le lexer et $Tid\ of\ string$ dans l'AST) et de la liste des déclarations séparés par une virgule (*ids*).

Comme dit précédemment, nous ajoutons le nouveau type dans l'AST ainsi qu'une expression $Tide$ dans l'$ASTSyntax$, qui correspond à l'énumération déclarée. Dans l'AST, nous modifions aussi le type $Programme$ qui comporte désormais une liste d'énumérations, une liste de fonction et un bloc comme ceci :
$Programme\ of\ enumerations\ *\ fonction\ list\ *\ bloc$.
Pour cela, il faut nécéssairement créer un nouveau type $Enumeration$ de *string* (le nom de l'énumération) et de *string list* (la liste des énumerations). Enfin, nous ajoutons deux infos : $InfoEnum\ of\ string\ *$ (*string list*) et $InfoEltEnum\ of\ string\ *\ string\ *\ int\ *\ string$ respectivement pour une énumération "générale" (nom + liste) et pour chaque valeur de la liste des énumérations, contenant le nom de son type père (par exemple Mois pour Janvier) son nom (Janvier), son adresse et son registre.

Maintenant que nous avons mis au point tous les éléments rajoutés à l'AST, passons aux passes.
Dans la passe TDS, nous ajoutons l'énumération à la TDS, et nous initialisons chaque élément de la liste à une adresse différente dans le registre SB. Vous pouvez retrouver le code dans la fonction $analyse\_tds\_enumeration$ qui elle même utilise $ajouter\_enum\_info$, cherchant l'existence de l'élément dans l'arbre, lève une exception $DoubleDeclaration$ si elle n'est pas déjà déclarée et l'ajoute sinon. Ensuite, nous rajoutons l'expression $Tide$ dans les expressions et enfin, nous modifions les fonctions déjà implantées (notamment l'$Adresse$ définie dans la partie des pointeurs) et la fonction *analyser* pour prendre en compte notre nouvelle définition de $Programme$ avec les énumérations.

Pour la passe de typage, les strings définis précédemment sont transformées en $info\_ast$, ici nous n'ajoutons pas beaucoup de code, simplement une fonction $analyse\_type\_enumeration$ qui renvoit une énumération dans le module $Placement$ (que nous ne modifierons pas) et nous ajoutons la taille du type Tide à 1 dans $type.ml$, tout simplement.

Dans la passe de génération de code, nous considérons que les valeurs des énumérations sont des entiers, c'est d'ailleurs pour cela que nous avons défini notre nouvelle info $InfoEltEnum$ de la même façon que $InfoVar$. Ici nous ne faisons que "charger" ($LOAD$) et "ecrire" ($STORE$) des nouveaux entiers et nous modifions les fonctions déjà implantées pour ajouter le type énuméré à chaque fois que celà est nécéssaire. Par exemple pour une affectation, on concatène l'expression à gauche du "=" avec l'affectable à droite.

Les tests que nous avons réalisés se trouve dans le dossier $rat - enum - test$, vous y retrouvez les exemples du sujet et des légères modifications de ce dernier pour vérifier les bonnes utilisations (par exemple l'impossibilité d'affecter un mois à un jour)

2.4 La structure de contrôle switch/case

En ce qui concerne le cas des $Switchcase$, nous avons fait le choix de les caractériser comme ceci :
$SwitchCase\ of\ expression\ *\ ((expression\ *\ bloc\ *\ expression)\ list)$ avec :

- expression : l'expression dans le switch, qui peut etre un booléén, un entier ou un type énuméré (nous avons défini cela dans le parser et le lexer).
- expression : l'expression d'un cas
- bloc : la liste d'instructions du cas
- expression : un booléen qui correspond à la présence ou non d'un break à la fin de la liste d'instructions.

On met ces trois derniers dans une liste pour avoir l'information de tous les cas.

Nous rajoutons un type "Def" et une expression "$Default$" correspondant au cas du $default$ dans le $Switch$.

Dans les quatre passes, nous rajoutons l'instruction du $SwitchCase$ et nous l'implantons.
Pour la passe TDS, on sépare la liste avec $split3$ que nous avons placée dans tds.ml, on analyse les expressions et les blocs de tous les cas puis nous recombinons dans une même liste avec $merge3$, également placée dans tds.ml. Nous pouvons maitenant passer à la passe de typage.

Nous procédons de la même manière en séparant la liste en trois avec notre fonction $split3$. On verifie ensuite que tous les booléens de la liste le sont bien et que toutes les expressions des cas sont biens du même type que l'expression du $switch$. On recombine le tout dans une liste et on renvoie le $SwitchCase$.

Pour la passe de placement, on fait la même chose qu'avec la $Conditionnelle$: on analyse les blocs des instructions des $case$.

Enfin, pour la passe de génération de code, nous avons décidé d'afficher à la suite la liste des $JUMPIF$. Si la condition du $JUMPIF$ est verifiée alors le programme passe directement au label correspondant au code de la liste d'instruction du cas. Si ce cas possède un $break$ alors il saute à la sortie de l'instruction switch, sinon il continue sur la liste d'instruction correspondant au cas suivant. Si la condition n'est pas verifiée, alors on re-charge du switch et ainsi de suite. Pour cela il nous faut implanter une fonction récursive $analyse\_code\_case$. Nous avons essayé de mettre en commentaire les étapes majeures de la fonction.

Les tests réalisés se trouvent dans le dossier $rat - case - test$.

3 Tests

Les tests du sujet sont placés dans le dossier $rat - testultime - test$. Ils passent sans erreur sur nos machines.

4 Conclusion

En conclusion ce projet nous a permis de comprendre le fonctionnement d'un compilateur et d'une traduction de langage. Nous avons pu nous améliorer au langage fonctionnel Ocaml en utilisant notamment beaucoup de $List.fold\_right$. Ce projet a présenté de nombreuses diffucultés et nous a surtout permis d'approfondir ce qu'était une $info$, car nous ne pouvions pas copier bêtement la correction du TD. L'ajout du $SwitchCase$ a été pour nous le plus simple et le plus rapide, car nous avions bien compris à quoi servaient toutes les $Infos$, fonctions etc. Cet ajout permet d'éviter d'enchaîner une succession de $if\ else$ très peu agréable pour la lecture.

Pour les types énumérés, nous pensions avoir réussi à ajouter les nouvelles infos du premier coup, mais à force de coder dans la passe de typage par exemple, nous nous sommes rendu compte que la façon dont nous codions étaient ridiculement laide et inefficace, nous sommes donc revenus plusieurs fois au debut pour redéfinir correctement nos $InfoEltEnum$, et nous avons perdu beaucoup de temps à ce niveau-là.

La surcharge nous a pas forcément poser de problème, si ce n'est la complication une fois arrivés à la passe de génération de code où nous avons décidé de renommer les fonctions en fonction de leurs paramètres.

Enfin, l'ajout des pointeurs n'a pas été la plus longue dans un premier temps, mais on s'est rendu compte à la fin du projet, lors du test final du sujet, que les tests que nous avons fait fonctionnaient "miraclement", nous sommes donc revenus à la fin sur eux pour les corriger.

Pour finir, nous pensons que le rapport temps passé / pourcentage de la note dans l'UE n'est pas cohérent. En effet, nous avons largement depassé les 50 heures de travail sur ce projet (les questions posées en début de vacances sont restées sans réponse jusqu'à la rentrée...) et tout ça pour seulement 40% de l'UE. Nous trouvons cela dommage.