Département Sciences du Numérique

Projet de programmation fonctionnelle et de traduction des langages Compilateur du langage RAT étendu

Louis Blenner, Mahé Tardy

12 janvier 2020

Table des matières

1 Introduction

Dans ce rapport nous allons voir comment ont été ajoutées les extensions du langage RAT au compilateur écrit en OCaml durant les séances de TPs. Les extensions sont les chaînes de caractères, les pointeurs, la surcharge de fonctions et les prototypes. Seront abordés les choix de conception, leurs limites et avantages, et enfin les possibilités d'évolution plus ou moins complexe du compilateur.

2 Types

L'évolution de l'AST reflète les changements effectués au sein de la grammaire. Des expressions ont été ajoutées pour la gestion des chaînes de caractères et des pointeurs telles que `Chaine`, `SousChaine`, `Taille`, `New`, `Adresse`, `Null`, `Acces`. Ce dernier concerne un accès sur un affectable, qui est un type ajouté qui comprend `Ident` ou `Valeur`. Une

nouvelle valeur a été ajoutée pour le type binaire : `Concat` pour les chaînes de caractères. Ensuite, dans les instructions, l'affectation a été modifiée en une structure prenant un affectable et une expression [1]. Pour l'ajout des prototypes, le type fonction a été modifiée en y ajoutant le `Prototype`. Enfin, pour refléter le changement fait dans la grammaire d'un programme discuté dans la partie 3.4, un programme est désormais un `Programme of fonction list * bloc * fonction list`.

Les changements suivants dans les AST des passes suivantes sont identiques aux changements de l'AST issu de la passe d'analyse lexicale. Il a cependant été ajouté de nouvelles instructions d'affichage spécifiques aux chaînes de caractères dans l'AST issu de la passe de typage.

On constate que les changements fait à l'AST correspondent pour la plupart aux changements fait sur la grammaire et qu'ils dépendent plus d'une vision logique que d'une solution unique. En effet, le découpage sur les affectables pourrait être discuté car le fait que son traitement soit régulièrement séparé selon l'usage montre qu'il ne rassemble pas forcément une même structure. Il est toujours possible de créer de nouveaux types ou d'en éliminer mais il faut trouver un équilibre logique entre les structures et les types.

3 Extension du langage RAT

3.1 Les chaînes de caractères

L'intégration des chaînes de caractères ne présente pas de difficulté particulière à la passe de TDS. Lors de la passe de typage, il faut bien valider les jugements de typages indiqués ci-dessous et également résoudre une nouvelle opération binaire : la concaténation de deux chaînes de caractères. La passe de placement est inchangée. Cependant la passe de génération de code est complexe, nous avons choisi de garder la proposition du sujet concernant la structure des chaînes de caractères dans le tas : longueur de la chaîne suivie d'un tableau de caractères [2]. Il faut alors réaliser plusieurs opérations lorsqu'on fait face à une expression de type chaîne de caractères : charger sa taille + 1 dans la pile pour allouer l'espace dans le tas puis charger chaque caractère et la taille dans la pile pour aller les stocker à l'adresse indiquée lors de l'allocation. Ainsi à la fin de l'opération, on obtient l'adresse pour retrouver la chaîne de caractères dans le tas en haut de pile.

Pour l'expression sous-chaîne, on ne vérifie pas dans $E\{E_1, E_2\}$ que les entiers E_1 et E_2 ne dépassent pas la longueur de la chaîne - 1. Ainsi des erreurs au moment de l'exécution peuvent être dues à des dépassements d'indices sur une chaîne de caractère.

Jugements de typage

Voici ci-dessous les jugements de typages ; pour une expression de type chaîne voir le jugement 1, pour une expression de concaténation de chaîne de caractères voir le

1. Ce qui rend le traitement plus élégant lors des différentes passes, en effet lors d'une affectation, on analyse naturellement l'affectable puis l'expression à affecter.

2. On notera que cette structure justifie l'ajout du mot clé `length` dans le langage, car on peut récupérer la taille d'une string de manière très efficace.

jugement 2, pour une expression de sous-chaîne voir le jugement 3 et pour une expression de taille d'une chaîne de caractères voir le jugement 4.

$$\sigma \vdash s : string \tag{1}$$

$$\frac{\sigma \vdash E_1 : string \quad \sigma \vdash E_2 : string}{\sigma \vdash (E_1 \hat{} E_2) : string} \tag{2}$$

$$\frac{\sigma \vdash E : string \quad \sigma \vdash E_1 : int \quad \sigma \vdash E_2 : int}{\sigma \vdash (E\{E_1, E_2\}) : string} \tag{3}$$

$$\frac{\sigma \vdash E : string}{\sigma \vdash length\ E : int} \tag{4}$$

3.2 Les pointeurs

L'intégration des pointeurs ne sera pas détaillée ici car elle a été traitée en TD. Il a été décidé de créer les affectables. Leur traitement doit être fait séparément dans le cas d'une lecture ou d'une écriture pour la passe de TDS. Néanmoins leur traitement a pu être factorisé dans la passe de typage de même que pour la passe de génération du code des affectables. Bien que réalisée dans une unique fonction, la génération est différente dans le cas d'une lecture ou d'une écriture, voir les commentaires associés pour plus d'information sur cette fonction.

Le type `Pointeur of typ` a été ajouté et ainsi pour permettre la déclaration d'un pointeur sur `null`, le type `undefined` est compatible avec n'importe quel autre type.

L'affichage des pointeurs, c'est-à-dire l'adresse vers laquelle un pointeur pointe, a été fait très simplement dans notre projet en utilisant `AffichageInt()`. On pourrait l'améliorer en dédiant une fonction qui ferait la même chose en concaténant un caractère '`@`' au début de l'entier afin de bien discerner qu'il s'agit d'une adresse.

Jugements de typage

Voici ci-dessous les jugements de typage ; pour l'expression de l'adresse *null* voir jugement 5, pour l'expression d'accès à l'adresse d'un identifiant $\&id$ voir jugement 6, pour un affectable accédé en valeur $*a$ voir jugement 7, pour l'expression d'allocation de mémoire dans le tas New voir jugement 8.

$$\sigma \vdash null : Pointeur(Undefined) \tag{5}$$

$$\frac{\sigma \vdash id : t}{\sigma \vdash \&id : Pointeur(t)} \tag{6}$$

$$\frac{\sigma \vdash a : Pointeur(t)}{\sigma \vdash *a : t} \tag{7}$$

$$\frac{\sigma \vdash t : \tau}{\sigma \vdash new\ t : Pointeur(\tau)} \tag{8}$$

3.3 La surcharge de fonctions

Afin de traiter la surcharge, nous avons décalé à la passe de typage la détection d'une double déclaration de fonction, en effet deux fonctions peuvent *a priori* avoir le même identifiant. Nous avons créé un nouveau type, le type `multiFun` qui regroupe à la fin de la passe TDS l'ensemble des types d'arguments possible pour un nom de fonction. Ainsi, lors de la passe de typage, nous retournons une erreur de double déclaration s'il existe deux fonctions de même nom qui ont les mêmes types de paramètres.

Nous n'acceptons pas la surcharge sur le type de retour dans ce cas-là, car cela signifierait faire des modifications au niveau des appels de fonctions, en effet un appel aurait donc plusieurs types possibles. Néanmoins, il serait possible de l'implémenter en faisant remonter une liste de types possibles aux expressions et en définissant des comportements par défaut dans les cas indécis. La difficulté réside alors dans le choix raisonnable des ces comportements par défaut, pour ne pas arriver, par exemple, à des situations comme en Javascript où l'expression `'1'+1` est évaluée en `'11'` et l'expression `'1'-1` en `0`.

Exemple de cas indécidable

```
int f();
string f();
int g(int a int b);
int g(string a string b);
int i = g(f(), f()); // Comment choisir le type de retour de f
```

Nous acceptons néanmoins la surcharge de type de retour du moment que les types des paramètres sont différents puisqu'ils permettent alors de caractériser une fonction particulière et donc un type de retour.

Lors de la passe de typage, lorsque nous rencontrons une fonction, nous créons une `infoFun`, nous l'ajoutons ensuite à l'`infoMultiFun` pour pouvoir associer les différents appels à l'`infoFun` correspondante. Finalement, lors d'un appelle de fonction, nous parcourons l'`infoMultiFun` à la recherche d'une liste de types correspondant aux arguments pour lui associer l'`infoFun` couplé à la liste d'arguments. Si la liste d'arguments n'est pas trouvée, nous renvoyons une erreur type de paramètres inattendus.

Pour créer une infoFun, il nous faut un nom de fonction qui permet d'y faire référence dans le code final, nous générons donc pour chaque fonction un nom de la forme `nom#numéro`, où le numéro est incrémenté à chaque fonction rencontrée. Ainsi les noms internes de nos fonctions ne correspondent plus aux noms initiaux et sont difficilement prévisibles. C'est la raison pour laquelle il est compliqué de faire des tests sur le placement mémoire pour les fonctions dans notre situation. Cependant, lors de la levée d'erreur, les chaînes de caractères représentant les noms des fonctions sont découpées au niveau du caractère `#` pour que les erreurs soient compréhensibles pour l'utilisateur. Il faudrait alors interdire l'utilisation du caractère `#` dans les noms de fonctions du langage RAT.

3.4 Les prototypes

Les prototypes permettent de déclarer une fonction *en avant* sans déclarer son corps. Cela est particulièrement utile dans le cadre d'une récursion mutuelle entre deux fonctions, impossible à écrire sinon.

Leur intégration avec la surcharge (qui résout désormais les doubles déclarations à la passe de typage) oblige à décaler certaines étapes. En effet, à la passe de TDS, l'analyse d'un prototype ajoute l'identifiant de la fonction à la TDS. Il n'y a pas encore de détection de double déclaration à cette étape. Ce n'est qu'à la passe de typage que l'on va être capable de déterminer si une fonction a effectivement été doublement déclarée. À cette étape, on va caractériser l'`infoFun` avec un booléen qui va nous indiquer si elle possède un corps ou non. Ceci va nous permettre, quand on détecte que l'identifiant existe déjà, de savoir si c'est à cause d'une déclaration de prototype préalable ou d'une réelle double déclaration. Ce n'est donc qu'à l'issue de cette passe qu'on va pouvoir, en parcourant de nouveau les `infoFun`, s'il manque le corps d'une fonction. C'est pour cela que cette étape doit se dérouler à la passe de placement mémoire.

Au niveau de la grammaire, il a été décidé de séparer la partie déclarée avant le bloc *main* et la partie déclarée après. Il aurait été en effet possible de considérer que le *main* doit systématiquement être analysé en dernier et dès cette étape, concaténer la liste de fonction *lf1* et *lf2* en lieu et place du bloc initial dans la grammaire précédente qui ne permettait qu'une liste de fonction *avant* le bloc *main*. Ainsi les prototypes auraient uniquement servi pour les déclarations de fonctions. Néanmoins, le choix permet un comportement assez naturel : lorsqu'on définit une fonction après le bloc *main*, il est nécessaire d'indiquer son prototype avant ce même bloc. Ainsi le compilateur analyse le code dans l'ordre d'écriture. Ce n'est qu'à l'issue de la passe de typage que les deux listes de fonctions sont concaténées pour la suite : le placement de mémoire et la génération de code.

grammaire prog

```
prog :
| lf1 = dfs ID li = bloc lf2 = dfs   {Programme (lf1,li,lf2)}
```

analyser de passeTdsRat

```
(* analyser : AstSyntax.ast -> AstTds.ast *)
(* Paramètre : le programme à analyser *)
(* Vérifie la bonne utilisation des identifiants et tranforme le
programme en un programme de type AstTds.ast *)
(* Erreur si mauvaise utilisation des identifiants *)
let analyser (AstSyntax.Programme(fonctions1, prog, fonctions2)) =
let tds = creerTDSMere () in
let nf1 = List.map (analyse_tds_fonction tds) fonctions1 in
let nb = analyse_tds_bloc tds prog in
let nf2 = List.map (analyse_tds_fonction tds) fonctions2 in
Programme (nf1, nb, nf2)
```

4 Conclusion

Pour conclure, les ajouts faits au langage ont présenté des difficultés à des niveaux différents. En effet, l'ajout de la surcharge a nécessité de revoir en profondeur ce qui était qu'une `info` associée à une fonction pour pouvoir ajouter les fonctions puis résoudre la surcharge. L'ajout des prototypes a nécessité de bien comprendre à partir de quelle passe nous pourrions savoir qu'il allait manquer le corps d'une fonction surchargée uniquement déclarée en prototype. L'ajout des chaînes de caractères (que nous avons traité en premier) nous a permis de bien comprendre les mécanismes d'interactions entre la pile et le tas. Enfin, l'ajout des pointeurs a été surtout délicat au moment de la gestion des affectables dans les différentes passes et surtout à la génération de code. En effet, le traitement diffère selon le type de l'identifiant à traiter, c'est-à-dire s'il s'agit d'un identifiant d'une variable ou d'un identifiant d'un pointeur.

Dans l'ensemble, il est intéressant de voir qu'avec un compilateur plutôt simple on peut parvenir à traiter un langage certes avec une écriture parfois lourde, mais des structures complexes : les possibilités offertes par la surcharge et les pointeurs paraissent en premier abord complexe à intégrer. Enfin, la réalisation d'un tel compilateur permet d'acquérir une compréhension profonde de ce que sont les éléments d'un langage et d'expérimenter une manière de les traiter simplement.

Pour terminer, les améliorations que nous aurions pu apporter au compilateur sont : la surcharge sur les types de retours des fonctions, une passe d'optimisation pour éliminer le code mort et notamment les fonctions importées inutilement,