Projet de compilation

Grégoire Szymanski

10 janvier 2018

1 Introduction

Avant de commencer un projet informatique, il faut choisir le langage. Dans le cas présent, le choix de ocaml s'est imposé très naturellement. D'une part, c'est celui que nous étudions en cours. D'autre part, les outils permettant de créer un compilateur s'intègrent très bien avec ce langage.

J'ai ensuite étudié la structure générale du programme. J'ai décidé de le découper en plusieurs parties grâce aux modules d'ocaml. Ces derniers ont en grande partie suivis le modèle du cours.

Division suivant le schéma du cours :

La première partie est composée du lexer et du parser. Elle analyse le fichier et construit un premier arbre de syntaxe abstrait très peu décoré. Ensuite, celui-ci est transmis au typer. Cette partie constitue le coeur de l'analyse des fichiers. Un arbre de syntaxe abstraite plus conséquent est produit. Celui-ci contient aussi les informations de typages et rajoute un peu de sel syntaxique. Puis, un deuxième typage pour les ressources a lieu. L'arbre n'est que très peu modifié et nous reprenons donc la structure du typage simple. On passe enfin à la compilation qui se déroule en deux étapes : une pré-compilation qui effectue tous les calculs préliminaires, suivie de la production de code proprement dite.

2 Le programme

2.1 Lexer et parser

Pour cette partie, nous avons utilisé les outils présentés en cours, c'est-à-dire ocamllex et menhir.

Le lexer est simple : il s'agit d'une copie des conventions lexicales données dans l'énoncé. La réalisation du parser s'est avérée plus délicate car les règles sont parfois ambiguës. Par exemple, les expressions à la fin d'un bloc étaient mal reconnues. Pour régler ce genre de problèmes, j'ai donc donné des priorités précises aux opérateurs.

2.2 Typage

Tout comme le lexer, la partie liée au typage simple découlait pour beaucoup des règles données par l'énoncé. Cependant, quelques petits problèmes sont apparus. Tout d'abord, mes premières versions ne reconnaissaient pas les vecteurs vides. J'ai donc crée un type spécial qui est accepté partout à la place d'un vecteur. Puis, j'ai remarqué que la définition de la relation $\tau_1 \leqslant \tau_2$ n'était pas récursive, alors que dans certains tests, cette définition devait l'être. Je l'ai donc modifiée afin de faire fonctionner ces tests. Enfin, j'ai également modifié la règle de typage

des vecteurs afin de prendre le type le plus large de la liste d'expressions définissant le vecteur. En effet, certains exemples nécessitaient également cet arrangement.

Enfin, je n'ai pas eu le temps de terminer les cas particuliers d'inférence des types, notamment dans le cas des vecteurs vides. Il m'aurait fallu de plus modifier le type d'une variable une fois celui-ci connu, ce que je n'ai pas pu réaliser.

2.3 Production de code

La production de code fut découpée en deux étapes :

La première phase est une pré-compilation de l'arbre de syntaxe abstrait fournit par le typage. On commence par supprimer toute référence à des variables en calculant les positions sur la pile ainsi que les chaines de caractères pour les positionner dans le segment de données "data". Puis, les fonctions ainsi que tous les blocs et opérations sont numérotés afin de pouvoir nommer proprement les sauts (sans risque de confusion).

La seconde phase est la production de code. La partie la plus importante du travail étant faite lors de la pré-compilation, on se contente ici de retranscrire l'arbre en instruction assembleur. Cependant, j'ai pris une petite liberté avec le sujet afin de simplifier les calculs. J'ai ainsi décidé de coder les entiers sur 64 bits au lieu des 32 demandés par l'énoncé afin d'éviter les conversions.

Selon moi, la plus grosse difficulté de la production de code a été de bien faire la disjonction de cas entre les "valeurs" codées sur 8 octets que l'on pouvait stocker dans les registres et les structures et vecteurs qui pouvaient prendre plus de place. Pour éviter trop d'accès à la pile, j'ai donc utilisé les registres pour stocker les valeurs lorsque cela était possible. Lorsque cela ne l'était pas, j'ai mis la valeur de l'adresse sur la pile des ressources considérées. A chaque endroit de la compilation, il a donc fallu surveiller le type des variables considérées pour savoir si l'on devait utiliser les valeurs des registres comme des valeurs ou des adresses.

Enfin, cette partie est incomplète car il manque une partie de la libération mémoire après l'utilisation des vecteurs. Mon programme traite ainsi les cas classiques de libération des vecteurs stockés dans une variable ou une structure. Cependant, faute de temps, je n'ai pas pu m'attaquer au problème des vecteurs créés "à la volée" [1], ni à celui des vecteurs de vecteurs. Par ailleurs, la libération mémoire est aussi incomplète lorsqu'une partie de la structure seulement est affectée à un autre endroit. (Voir la partie suivante)

2.4 Typage des ressources

Cette section fut certainement la plus difficile pour moi.

Cette partie mettait en oeuvre trois différents types de vérification. Tout d'abord, elle procédait à une vérification du typage via les durées de vie. Ces dernières ont été modélisées par un entier qui croît à chaque nouveau bloc imbriqué. Il s'agissait ensuite de mettre en place une vérification du propriétaire à chaque instant. Pour cela, on retient à chaque instant de la lecture de l'arbre de syntaxe abstrait, le statut de chaque variable en prenant soin de le modifier à chaque fois que cela est nécessaire. Par ailleurs, contrairement à ce qui est fait en rust, mon programme considère que si un attribut est affecté à un autre endroit, toute la structure est vide. C'est pour cela que la libération mémoire marche mal dans ce cas. Pour régler ce problème, une méthode serait de considérer chaque attribut séparément dans l'analyse des propriétaires. Enfin, le typage des ressources devait aussi vérifier la cohérence des emprunts, mais cette partie n'a pas été traitée faute de temps.

1. typiquement, une expression consistant uniquement en la création d'un vecteur sans le stocker nulle part

3 Le débogage

Afin de debugger efficacement le code, j'ai positionné dans mon code à des endroits stratégiques des affichages des principales variables. Pour le typage, j'affichais également l'environnement complet après chaque instruction pour suivre son déroulement.

La plupart des affichages ont été retirés mais les fonctions d'affichage ont quant à elles été conservées.

J'ai aussi créé un dossier de fichiers tests à l'image de celui proposé avec le sujet pour tester le compilateur. Ceux-ci portent essentiellement sur des tests d'exécution et le plus souvent sur des points techniques comme l'ordre de passage des arguments ou la position des variables dans une structure.

4 Conclusion

Réaliser ce projet m'a beaucoup appris sur le fonctionnement d'un compilateur.

En effet, il m'a permis de mieux cerner le rôle de chaque partie étudiée en cours et de comprendre comment celles-ci pouvaient interagir pour former au final un ensemble cohérent. De plus, les grammaires et les règles de typages, encore abstraits pour moi au début du projet, me semblent plus claires.