Compilateur pour Petit Julia

Hector BUFFIÈRE et Thomas LAURE

1er semestre 2020-2021

1 Introduction

Projet réalisé dans le cadre du cours Langages de Programmation et Compilation du 1er semestre de L3 à l'Ecole Normale Supérieure.

Sujet : `https://www.lri.fr/~filliatr/ens/compil/projet/sujet-v2.pdf`

Ce projet a été réalisé dans le langage Ocaml.

2 Analyse lexicale et syntaxique

2.1 Analyse lexicale

L'analyse lexicale a été réalisée avec Ocamllex.

Le point-virgule automatique en fin de ligne est traité à l'aide d'une référence booléenne globale. Quand on rencontre un lexème, on met celle-ci à jour selon si ce lexème serait suivi d'un point-virgule automatique si l'on rencontrait un retour chariot ensuite. Quand on rencontre un retour chariot, on renvoie le lexème `";"` si la valeur de la référence était Vrai.

Petit Julia accepte que des entiers soient suivis directement d'une parenthèse ou d'une variable, on renvoie donc des lexèmes dédiés dans ce cas, et de même pour les variables suivies ou suivant directement une parenthèse.

Enfin, Ocaml n'acceptant pas les entiers au delà de 2^{62}, on utilise le module $Int64$ d'Ocaml pour représenter les entiers 64 bits de Petit Julia.

Nous avons également ajouté les lexèmes / et \ qui n'étaient pas demandés par le sujet.

2.2 Arbre de syntaxe abstraite

Les expressions sont représentées par un type construit contenant leur localisation dans le fichier, et la nature du nœud de l'arbre qu'elles représentent.

Les déclarations de fonctions et de structures sont des types construits constitués de leurs noms, de leurs arguments, de leurs localisations dans le fichier, de leurs instructions pour les premières et de leur caractère mutable ou non pour les secondes.

Les paramètres sont aussi un type construits donnant comme information leur nom, leur type et leur localisation dans le fichier.

2.3 Analyse syntaxique

L'analyse syntaxique a été réalisée avec Menhir.

L'analyseur tient compte du sucre syntaxique proposé. La fonction div est remplacée par l'opérateur binaire /, la fonction *println* par la fonction print à laquelle est ajoutée l'expression $\backslash n$ après les arguments. Les types omis dans les déclarations de paramètres sont bien évalués à *Any*.

Les éléments des blocs (listes d'expressions) et des listes de paramètres peuvent être séparés par n'importe quel nombre de points-virgules, et la grammaire en tient compte.

Les précédences et associativités sont celles indiquées dans le sujet. Un conflit lié au return a été résolu en déclarant les lexèmes centraux des expressions comme de précédence plus grande que celle du return. Les conflits liés à la succession d'une expression et d'un bloc sans séparateur dans les if, while et for ont été résolus en séparant la production (mot-clé expression) de la production ((mot-clé expression) instructions fin) et en associant une précédence plus grande à la première.

3 Typage

Le typage se fait en deux étapes, comme suggéré dans le sujet.

La première produit un environnement primaire dans lequel on vérifie que les déclarations de types sont valides (types bien formés, pas de doublon dans les arguments d'une fonction, dans les déclarations de fonctions ou de structures). On initialise également les variables globales à ce moment. On garde en mémoire plusieurs dictionnaires contenant les noms et types déclarés des variables globales, tous les types de fonctions associées à un même identifiant, les structures et leurs champs, et réciproquement la structure associée à un champ pour pouvoir typer les accès à ce champ.

Dans un deuxième temps, on construit les environnements locaux associés à chaque fonction puis seulement après on type le corps desdites fonctions (Julia étant fait de telle sorte qu'une déclaration en fin de fonction peut influencer sur un comportement en début de fonction, ce que nous avons mis un certain temps à bien comprendre). Enfin, on peut vérifier que toutes les expressions du programme sont bien typées dans l'environnement global et l'environnement local dans lequel elles sont plongées le cas échéant.

Le typage reconstruit un arbre de syntaxe abstraite sur le même modèle que le précédent en remplaçant la localisation d'une expression dans le fichier par son type. Les fonctions et les structures sont retenues dans un dictionnaire, on détruit donc les instructions des fonctions dans la nouvelle liste de déclarations. On renvoie les dictionnaire des variables globales, des fonctions, des structures et des champs.

Les blocs possèdent également l'information de l'environnement local spécifique dans lequel ils sont éventuellement plongés (s'il s'agit d'instructions d'un for, d'un while ou du corps d'une fonction). A un appel de fonction, on associe la liste des identifiants des fonctions de même nom compatibles avec le nombre d'arguments et leur type. Si aucune fonction n'est compatible, on échoue, et si plusieurs fonctions sont compatibles et que l'on est sûrs de ne pas avoir d'infor-

mations plus précises à l'exécution, c'est à dire que tous les types des arguments sont connus précisément et que parmi les fonctions compatibles aucune n'est plus spécifique que les autres, on échoue aussi.

4 Production de code

4.1 Valeurs

Toutes les valeurs occupent deux mots de 64 bits. Un premier mot indique si le type de la valeur : entier, booléen, chaîne, ou structure (toutes les structures ont ce même identifiant). Ces identifiants sont pairs, sauf s'il s'agit d'une variable non encore déclarée, auquel cas ils sont impairs. Le deuxième mot est la valeur si on a un entier ou un booléen, ou un pointeur dans le cas d'une chaîne ou d'une structure.

Les chaînes de caractères sont allouées dans le segment de données. En effet, on peut déterminer statiquement toutes les chaînes qui apparaissent dans le programme, et on attribue à chacune un identifiant. Les structures sont allouées sur le tas, par un mot de taille $2n + 1$ où n est le nombre de champs de la structure. Le premier mot est un identifiant unique à la structure, et chacun de ses champs est ensuite représenté par un type et une valeur de la même façon qu'indiqué ci-dessus.

La valeur *nothing* est représentée par un double 0.

4.2 Compilation des expressions

La fonction *compile_expr* empile la valeur de l'expression au sommet de la pile (type au sommet et valeur en-dessous). Les expressions n'ayant pas de valeur valent *nothing*. On parcourt récursivement l'arbre de syntaxe abstraite renvoyé par le typeur en compilant les expressions les unes après les autres. On fait des vérifications de type pour les opérations binaires, les conditions des boucles et tests.

Des labels $instr_i$ sont utilisés pour les sauts conditionnels des boucles et des tests. Une référence *instr_id* est incrémentée pour donner des identifiants aux instructions.

4.3 Variables

Les variables globales sont allouées dans le segment de données, avec une valeur de type et une valeur. Leur type est initialisé à 1 (unbound) et est modifié au moment où elles sont affectées.

Les variables locales sont allouées sur la pile. Le typeur renvoie pour tout bloc l'environnement local spécifique du bloc. On garde un environnement qui consiste en une paire (l, e) pour chaque variable locale. l représente le niveau auquel cette variable est conservée. Ainsi, si elle est spécifique au bloc, l vaut 0, si elle est déclarée à un niveau plus haut, l est positif. o représente l'ofset par rapport au rbp du niveau de déclaration auquel la variable est conservée. La fonction *get_vars* récupère le rbp du niveau l en remontant récursivement la pile de rbp en rbp, puis on récupère la variable locale avec l'ofset.

Chaque fois que l'on utilise une variable, on tente d'abord de la chercher parmi les variables locales puis en cas d'échec parmi les variables globales. On

ne peut pas échouer une deuxième fois, car alors le typeur aurait déjà échoué à cet endroit.

4.4 Fonctions

Comme l'environnement en petit-julia est dynamique et ne peut pas être déterminé statiquement, mais ne dépend que de la ligne dans le programme initial, on compile toutes les variantes possibles de chaque fonction à chaque appel de celle ci à une ligne différente (stockées dans $f\_implems$). Ainsi une fonction récursive sera compilée au niveau de son appel initial et au niveau de son appel récursif, puisqu'elle pourrait avoir redéclaré une variable globale en variable locale par exemple, mais elle ne sera pas réimplémentée plus de deux fois (sauf si elle est réutilisée autre part).

Le schéma de compilation est essentiellement celui proposé par le sujet. Un argument ret_depth est passé pour donner la profondeur du dernier appel de fonction afin de trouver son rbp lors d'une instruction return.

Le dispatch n'a pas été réellement implémenté, seulement préparé (toutes les possibilités de fonctions déterminées par le typeurs sont implémentées, mais la première est systématiquement appelée). Un algorithme a cependant été pensé. On remarque qu'il existe un ordre strict non total "plus spécifique que" entre les méthodes d'une même fonction, qui correspond à "chaque argument est plus spécifique que", où plus un type déclaré est plus spécifique qu'un Any. Ceci définit un graphe orienté acyclique sur les méthodes potentielles à un appel retournées par le typeur, qu'on peut calculer en OCaml étant donné qu'elles ont toutes le même nombre d'arguments. En remarquant que si une méthode est valide pour des arguments donnés (elle pourait s'exécuter avec ces arguments), toutes les méthodes accessibles depuis cette fonction le sont, on voit que le problème du dispatch est de savoir s'il existe une unique fonction valide plus spécifique que toutes les autres, donc s'il existe dans le graphe une et une seule méthode valide sans parent valide. On peut alors associer à chaque méthode trois variables locales correspondant à son identifiant et les booléens est_valide et $a\_un\_parent\_valide$. Pour chaque méthode, on peut alors empiler les types de ses arguments, vérifier que tous les arguments sont compatibles avec ceux des arguments réels calculés, puis si elle est valide affecter sa variable est_valide et les variables $a\_un\_parent\_valide$ de tous ses fils en conséquence. Enfin on vérifie qu'il existe une seule méthode valide sans parent valide, auquel cas on renvoit son identifiant et on dépile toutes les variables ayant servi au dispatch. S'il y a plusieurs de ces méthodes on renvoit l'erreur ambiguïté, et s'il y en a aucune l'erreur aucune méthode ne correspond.

4.5 Erreur

Le rsp initial est stocké durant toute l'exécution dans r15. En cas d'erreur, on affiche le message d'erreur, on met 1 dans rax, r15 dans rsp pour remonter tout en haut de la pile, et on sort.