Rapport projet compilation

Dorian Lesbre

9 janvier 2019

Gestion des erreurs : le module `erreurs` gère les exceptions à la fois du compilateur et de l'entrée, respectivement par les fonctions `compiler_failure` (code de sortie 2 spécifie le fichier `.ml` qui soulève l'erreur) et `raise_error` (code de sortie 1). La plupart des autres modules renomment ces deux fonctions par application partielle.

Arbres de syntaxe : ils sont définis dans `syntax` et `type_syntax`, décoré par des positions et des types (pour le second). Ces fichier n'ont pas d'interface car elle se limiterais à recopier leur contenu.

Lexer : fait avec ocamllex. Afin de limiter le nombre d'état, le mots-clés et identificateurs ont été regroupés, ainsi que les opérateurs et le ponctuation. L'identification précise se fait alors par recherche dans un table de hachage du lexème reconnu. Le lexer gère également les commentaires et les chaines. Les chaines sont construites caractère par caractère pour vérifier que chaque caractère est valide et bien insérer les caractères spéciaux (\n et non \\n).

Parser : fait avec menhir. Tous les lexèmes sont définis avec le préfixe "T" (abréviation "TK" pour keyword, "TS" pour séparateur (ponctuation), TO pour opérateur, Int pour ?).

Les principaux conflits rencontrés sont les suivants :

— sur les listes séparés par des point virgules pouvant finir ou non par des points virgules. Ce conflit à été géré à l'aide d'un constructeur pending_list cherchant à réduire en priorité le couple (élément, séparateur).

— les priorité sur If et While ont été traités avec des priorités explicites règle par règle (et non juste des priorités sur les mots clés)

— Conflit sur les types entre les règles `(`$\langle typ \rangle$`)` et `(`$\langle typ \rangle^+_,$`) -> `typ (gérés par un même règle cherchant la flèche en option).

Typeur : il est réparti en trois modules : `environnement`, `type_functions` et `typeur` (dans l'ordre des dépendances). Le premier gère les environnements : les variable, fonctions et classe sont stockés dans deux Map, locales et globales. Les environnements sont mutables (pour ajouter des variable au fur et a mesure qu'on les voit). Mais réalise une copie indépendante avant de typer un sous-bloc (fonction `child`, qui écrase le variables globales masquée par les variables locales). Les environnement contiennent également une option pour spécifier le type de retour.

Le module `type_functions` regroupe diverse fonction utiles au typeur, notamment les comparaisons de type. Afin de palier à la non injectivité du typage, les types t? sont décorés d'un booléens qui assure leur non-nullité. La fonction `compatible a b` renvoie vrai si l'on peut écrire une expression de type a dans une de variable de type a (non-symétrique), `not_equal` teste l'inégalité structurelle (en ignorant les booléens).

Enfin le module `typeur` contient les fonctions transformant un arbre abstrait en un arbre décoré par des types. Les deux principales difficultés du typeur été :

— les fonctions polymorphe, gérés en stockant dans l'environnement non pas le type, mais un constructeur distinguant type réel (`TrueType` et prototype `AbstFunc`)

— les variables annulables, gérés de deux façon :
  — à court terme dans l'évaluation paresseuse de booléens dans une pile (`nullstack` de `environnement`) stockant une paire d'ensemble, le premier indiquant les variables non-nulle et le deuxième les variables nulles.
  — à plus long terme elle le sont dans l'environnement (ensembles `lc_nulls` et `gb_nulls` (sous-bloc if ou après une affectation explicite).

Le typeur renvoi des arbres contenant des informations supplémentaires pour la production de code :
— les types, même si, en rétrospection, les seuls endroits où ces types sont utiles lors de la compilation sont lors des accès (pour déterminer l'offset entre le début de la classe et l'attribut cherché).
— les listes de variables libres des fonctions anonymes (fonction `find_free` de `type_functions`), obtenues en reparcourant leur définitions.
— le nombre de variable locales de chaque fonction et déclaration de variable globale, obtenue en comptant les variables dans les environnements (fonction `nb_vars` de `environnement`).

Allocation de variables : L'allocation des variables se fait dans le module `allocation`. Il procède de manière similaire à `environnement`, à base de Map associant le nom d'une variable à son emplacement de stockage (chaine de caractère, par exemple `-8(%rbp)`). Une différence notable qui empêchait de le faire directement depuis le module `environnement` est que ce dernier ne garde en mémoire que les variables qui sont accessible dans le bloc courant, tandis qu'`allocation` procède à l'aide d'une pile, gardant donc toutes les variables en mémoire. Les principe généraux d'allocation sont les suivant :
— toutes les variables sont allouées sur la pile, tous les arguments également.
— Chaque opération stocke son résultat dans `%rax`.
— Les résultat intermédiaires (par exemple le calcul d'un membre d'un addition) sont poussés sur la pile. `%rdx` et `%rsi` servent de variables temporaires au besoin.
— Les constructeurs de classe stocke la classe en constuction dans `%rbx`, les fonctions anonymes stockent leur fermetures dans `%r12`. Ces deux registres sont donc sauvegardé en cas de conflit (appel d'un constructeur dans un constructeur ou d'une fonction dans une fonction anonyme)

Production de code : j'ai choisi de formatter directement le code avec `Format.fprintf` plutôt que le module `X86-64` fournit. Afin de différencier les étiquettes introduites des identifiant petit kotlin, toute mes étiquettes commencent par un point. Les principaux problème rencontrés lors de la production de code sont les suivants :
— mauvais calcul des décalage de variable, position dans une classe ou dans une fermeture.
— ajouter `this` comme attribut de classe plutôt que de le faire pointer directement vers la classe elle même (la syntaxe `a.this` est invalide en petit kotlin car this est un mot-clé).
— une première version remplaçait les appels terminaux à `printf` dans `.print_int` et `.print_string` par des jumps. Cela donne lui à un bug particulier : le programme s'exécute correctement quand on pipeline la sortie dans un fichier ou à `less`, mais fait un erreur de segmentation lors qu'il tente d'afficher directement en console.
— calcul du nombre total de variable faux (plus faible qu'en réalité), ce qui amène un conflit entre les variables stocké après `%rsp` et les valeurs poussées sur la pile.
— L'opération `negq` ne correspond pas directement à la négation booléen dans la représentation choisie.