

Rapport du projet

Auteurs:

Kyrylo Hrynevych

Raphaël Dubach

Remarques générales:

La partie 1 a été réalisée intégralement, tandis qu'il manque certaines petites choses à la partie 2, ce qui sera détaillé plus tard dans le rapport. Les tests sont retrouvables dans le makefile, et il y a une commande pour lancer chaque type de test (tests positifs, ceux qui lèvent une erreur de syntaxe et ceux qui lèvent une erreur de type).

Analyse lexicale : *mgolexer.mli*

Ce fichier fait les actions suivantes:

- Importe Lexing et les tokens générés par Mgoarser, et définit une exception Error pour les erreurs lexicales.
- Construit une table keyword_or_ident qui distingue mots-clés (package, import, type, struct, if, for, var, true, false, nil, return, ...) et identifiants génériques (IDENT s).
- Gère les constantes entières (intlit) avec conversion en int64 et contrôle de bornes.
- Gère les chaînes via une règle dédiée string_lit qui reconstruit la chaîne dans un Buffer et traite les séquences \\, \", \n, \t .
- Reconnaît les commentaires sur une ligne (// ...) et multi-lignes (/ * ... */) via la règle auxiliaire comment.
- Produit tous les tokens de ponctuation et opérateurs (parenthèses, accolades, +, -, ==, !=, &&, ||, :=, etc.) ainsi qu'un token spécifique pour fmt.Print.

Nous avons implémenté tout ce qui était demandé dans l'énoncé, y compris le mécanisme d'insertion automatique de point-virgule : un booléen insert_semi est mis à jour par la fonction "sop" selon le dernier token émis, et à la lecture d'un retour chariot on insère éventuellement un token SEMI au lieu de simplement reprendre l'analyse.

Analyse syntaxique: *mgoparser.mly*

Ce fichier fait les actions suivantes:

- Définit tous les tokens et les règles de priorités des opérateurs.
- Définit toutes les règles de la grammaire et de la syntaxe des fichiers .go, et lève une erreur si les règles ne sont pas respectées.
- Toutes les règles y sont:
 - Les variables, types, et la structure générale avec la liste de déclarations.
 - Toutes les instructions, les boucles for avec toutes les options possibles (boucle infinie, boucle for normale; etc).
 - Toutes les expressions et opérations.
 - etc.
- Donne à tout le programme les types de la syntaxe abstraite, permettant l'analyse des types.
- Le fichier ne lève pas de conflit, et la syntaxe abstraite n'a pas été modifiée.

Vérification des types : *typechecker.ml*

Le fonctionnement :

- Une exception `Error(location * string)` et des helpers `error` / `type_error` pour signaler les erreurs de typage.
- Trois environnements implémentés par `Map.Make(String)` :
 - `tenv` : `typ Env.t` pour les variables.
 - `senv` : `((ident * typ) list) Env.t` pour les structures (nom -> liste de champs).
 - `fenv` : `(typ list * typ list * func_def) Env.t` pour les fonctions (paramètres, retours, définition).
- Les fonctions de base : `equal_typ` (comparaison de types), `check_typ` (vérifier qu'un type est bien formé), `check_fields` (types et unicité des champs), `lookup_field` (recherche d'un champ dans une struct).

Le typage des expressions est géré par :

- `type_expr`, qui calcule le type d'une expression en tenant compte d'un booléen `fmt_imported` pour autoriser ou non `Print`.
- `call_signature`, pour vérifier les appels de fonctions et renvoie la liste des types de retour
- `check_expr`, qui vérifie qu'une expression a un type donné.

Le typage des instructions est fait par :

- `is_lvalue` (détection des valeurs gauches).
- `check_instr` :
 - les instructions expression, blocs, if/else, for.
 - les incréments/décréments sur entiers lvalues.
 - etc.
- `check_seq`, qui applique `check_instr` séquentiellement et propage l'environnement.

Enfin, par la fonction `prog`, on construit les environnements de structures et de fonctions à partir des déclarations du programme, on vérifie les champs de toutes les structures, puis on lance `check_function` sur chaque fonction pour valider l'ensemble des règles de typage décrites dans le sujet.

Pour ce qui est du bonus, nous avons implémenté le typage avancé des fonctions à résultats multiples : pour les appels, les affectations multiples et les déclarations `var / :=` à partir d'un appel. Mais nous n'avons pas fait la vérification des variables locales inutilisées.

Partie 2: transformation en code MIPS

Nous avons fait quelques ajouts dans [mips.ml](#), pour rajouter certaines opérations.

Tout le reste a été réalisé dans le fichier [compile.ml](#).

- Nous avons opté dès le départ pour la réalisation de fonctions qui récupèrent les informations sur les structures, les fonctions et les variables dans les fonctions et que l'on lance dans `tr_prog`.
- Tout ce qui est dans `.data` sert uniquement à définir les strings, et tout le reste du code mips est dans la partie `.text`.
- Lorsqu'une fonction est appelée, on sauvegarde à chaque fois la return adress ainsi que le frame pointer avant de faire le saut; on récupère l'adresse à la fin et on libère la place utilisée dans la pile. Ce système permet d'appeler des fonctions dans des fonctions, et ce autant de fois qu'on veut.
- Nous utilisons presque exclusivement le registre `t0` (et parfois `t1`) avec des sauvegardes dans le tas pour toutes les actions: enregistrer des variables, faire des calculs, etc.
- Pour les structs, on stocke la taille totale et l'offset de chaque champ dans `senv` et l'expression `a.x` est compilé comme un chargement à l'adresse `a+offset(x)`. `new(T)` est traité comme une primitive `sbrk` avec la taille de la struct.

Difficultés:

Certaines implémentations compliquées dans la partie 2: par exemple le retour de plusieurs variables.

Puisque la première option proposée dans le sujet avait l'air très compliquée à implémenter, nous avons plutôt opté pour la deuxième, à savoir considérer les n-uplets comme des valeurs allouées sur le tas.

Elle était en vérité également assez difficile, puisqu'il fallait éviter de se perdre dans les sauvegardes d'adresses et les allocations dans le tas. Il fallait également traiter le cas où l'on déclarait plusieurs variables à partir d'une fonction rendant plusieurs variables, et dans ce cas précis aller chercher les valeurs présentes dans le tas.

Pour ce qui est des tests:

Nous manquons probablement de tests, mais ceux que nous avons ajoutés vérifient:

- Que l'analyseur syntaxique et le typechecker renvoient bien une erreur s'il y en a une, dans les cas les plus courants.
- Que toutes les instructions soient bien reconnues par l'analyseur syntaxique.
- Que les programmes en MIPS renvoient la bonne valeur, en utilisant un compilateur de MIPS à côté.

Malgré cela, il est possible que certaines erreurs soient passées entre les mailles du filet, bien que nous restions confiants dans notre implémentation.

Compilation de la fonction print:

Nous n'avons pas réussi à faire imprimer tous les types:

- Les booléens renvoient 1 si vrai, 0 sinon.
- Les éléments des structures ne peuvent pas s'imprimer; par exemple on ne peut pas faire: `fmt.Print(a.x)`, par contre on peut faire:
`var b = a.x;`
`fmt.Print(b);`
- Les fonctions qui rendent plusieurs variables ne peuvent pas être imprimés directement, mais on peut mettre les résultats dans des variables puis imprimer ces variables.
- Les chaînes de caractères s'impriment seulement si on cherche directement à en imprimer une, et non pas une variable ayant comme valeur un string, ou une fonction renvoyant un string.

Cette erreur vient du fait que nous n'avons pas trouvé d'autre moyen que changer la syntaxe abstraite, et le print étant presque la toute dernière fonction que nous avons implémenté dans [compile.ml](https://github.com/compileml/compileml), il aurait fallu changer le projet tout entier juste pour pouvoir imprimer des chaînes de caractères dans certaines situations. Nous avons donc décidé de faire ce compromis.

La gestion de la mémoire:

Nous savons que notre approche a quelques limites : par exemple la mémoire allouée pour les structs et les n-uplets de retour n'est jamais libérée et la recherche des champs se fait

uniquement par nom, ce qui pourrait être ambigu si deux structs avaient un champ avec le même identifiant.