PetitGo
– projet de compilation –

Maxime FLIN

12 janvier 2020

1 Manuel d'utilisation

1.1 Organisation du dépôt

Le compilateur a été réalisé en `Ocaml` avec `Ocamllex` et `Menhir`.
Les principaux fichiers du projet sont dans le dossier `src` :

main Fichier d'entrée du programme

ast Arbre de syntaxe abstraite utilisé construit pendant le parsage

config Fonctions d'utilitées générales et les paramètres passé en ligne de commande (voir section 1.3)

error Ensemble de fonctions permettant de lever des erreurs et de les afficher proprement (voir section 4.2)

graph Implémentation de la recherche de cycle dans un graphe

lexer et parser Fichiers `Ocamllex` et `Menhir`

typer Fonctions de typage

builder Construit un arbre de syntaxe abstraite prêt à être compilé

optimizer Optimise l'abre de syntaxe abstraite de compilation

compiler Compile l'arbre de syntaxe abstraite vers du code assembleur

On trouve aussi dans le dossier `tests` les fichiers de test et le script `tester.sh`.

1.2 Compiler le compilateur

Pour compiler le projet, j'ai fait le choix d'utiliser `dune`. Il suffit donc d'entrer les commandes suivantes

```
git clone https://git.eleves.ens.fr/mflin/petitgo.git
cd PetitGo
dune build src/main.exe
```

1.3 Compiler avec le compilateur

L'exécutable est assez simple à utiliser. Pour compiler un fichier `PetitGo` il faut le passer en argument le nom de ce fichier à l'exécutable.

Le programme peut aussi prendre des options en fonction des besoins.

-v Mode verbeux.

–parse-only L'éxécution s'arrête après le parsage.

-type-only L'éxécution s'arrête après le typage.

-o <**file**> Précise le nom du fichier produit.

-E Produit un executable avec le même nom que le fichier assembleur.

-O Optimise le code produit

-allow-unused-var Autorise les variables inutiles.

-allow-unused-pkg Autorise les imports inutiles.

-allow-unused Active les deux options précédentes.

-wild-mode Désactive les sécurités ajoutées à l'exécution.

-quiet Désactive les avertissements.

Ainsi, pour simplement compiler le fichier de test `tests/exec/abr.go` vers un fichier assembleur, il suffit d'entrer la commande suivante

```
main.exe tests/exec/abr.go
```

Pour avoir un exécutable optimisé qui s'appelle `a.out`

```
main.exe -E -O -o a.out tests/exec/abr.go
```

2 Implémentation du sujet

2.1 L'arbre de syntaxe abstraite

L'arbre de syntaxe abstrait est représenté par les structures décrites dans le fichier `ast.ml`. Les deux types les plus importants sont les types représentants les expressions et les instructions

```
and expr =
  Enil
| Eident  of string
| Eint    of int64
| Estring of string
| Ebool   of bool
| Etuple  of expr loc list
| Eattr   of expr loc * ident loc
| Ecall   of (ident loc option) * ident loc * (expr loc list)
| Eunop   of unop * expr loc
| Ebinop  of binop * expr loc * expr loc

and instruction =
  Inop
| Iexpr   of expr loc
| Iasgn   of expr loc * expr loc
| Iblock  of instruction list
| Idecl   of ident loc list * ty loc option * expr loc option
| Ireturn of expr loc
| Ifor    of expr loc * instruction
| Iif     of expr loc * instruction * instruction
```

On remarquera le constructeur d'expression `Etuple` qui est simplement une liste d'expression. Le langage `PetitGo` n'a pas de tuple comme `Ocaml`, j'ai tout de même fait le choix d'ajouter ce constructeur pour uniformiser le type dans les autres déclarations. Ce choix est questionnable et présenta des désaventages lors du typage, rien d'insurmontable pour autant.

On notera de plus la présence du type α `loc` plusieurs fois dans les déclarations. Ce type est juste un enregistrement qui permet de se souvenir de la position des éléments retenu dans le fichier.

```
type 'a loc = { v : 'a; position : position }
```

2.2 Le lexer et le parser

Il n'y a pas grand chose à dire sur ces parties du projets, elles sont une implémentation plus ou moins directe de la syntaxe décrite dans le sujet.

Le point virgule automatique en fin de ligne est géré dans le lexer. J'ai utilisé une référence `is_semi` indiquant s'il faut insérer un point virgule après le retour à la ligne ; une fonction `tok` pour la mettre un jour ; et une fonction `eol` pour insérer le retour à la ligne si besoin.

```
let is_semi = ref false

let eol f lexbuf =
  Lexing.new_line lexbuf;
  if !is_semi
  then begin is_semi := false; SEMI end
  else f lexbuf

let tok t =
  let _ =
    match t with
    | IDENT _ | INT _ | STRING _ | TRUE | FALSE
      | NIL | RETURN | INCR | DECR | RPAR | END ->
       is_semi := true
    | _ -> is_semi := false
  in
  t
```

J'ai légèrement étendu la syntaxe du `PetitGo` pour permettre d'utiliser des fonctions d'autres packages que `main` et `fmt` (voir section 4.1). Plus précisement les modifications sont les suivantes

```
<type> ::= <ident> | <ident> . <ident> | * <type>
<expr> ::= <ident> . <ident> (<expr,>*) | ...
```

2.3 Le typage

L'implémentation du typeur a été beaucoup plus longue que celle du parser et du lexer. J'ai commencé par définir un type représentant les types de valeurs possibles dans le fichier `ast.ml`

```
type typ =
  Tvoid
| Tnil
| Tint
| Tbool
| Tstring
| Ttuple  of typ list
| Tstruct of ident
| Tref    of typ
```

Je reviens ici rapidement sur la remarque que j'ai faite en section 2.1 à propos du constructeur `Etuple` introduit dans les expressions. Sans ce constructeur, on pourrait se passer du contructeur `Tvoid` dans le

type `typ`. En effet, une expression serait alors du type `typ list` et la liste vide représenterait le type `Tvoid`. Il est possible que cette modification simplifie quelques parties du code du typeur mais je n'ai pas eu la volonté de la faire.

Ensuite, j'ai redéfini un type d'ast *typé* dans le fichier `ast.ml`.

```
type texpr =
  Tenil
| Teint    of int64
| Testring of string
| Tebool   of bool
| Tident   of ident * typ
| Tetuple  of texpr list
| Tattr    of texpr * ident * ident * typ
| Tcall    of ident * texpr list * typ list * typ
| Tunop    of unop * texpr * typ
| Tbinop   of binop * texpr * texpr * typ
| Tprint   of texpr list
| Tnew     of typ

type tinstruction =
  Tnop
| Texpr   of texpr * typ
| Tasgn   of texpr * texpr * typ
| Tblock  of tinstruction list
| Tdecl   of ident list * typ * texpr option
| Treturn of texpr
| Tfor    of texpr * tinstruction
| Tif     of texpr * tinstruction * tinstruction
```

On ne trouve plus de localisations dans ici car, passé le typage, le compilateur n'est plus censé échouer. Les localisations n'étant utiles que pour préciser la position d'une erreur dans un fichier, passé le typage on peut se permettre de les oublier.

Le typage d'un package renvoie un environnement du type

```
type env = {
    structs : tstruct Smap.t;
    types : typ Smap.t;
    funcs : tfunc Smap.t;
    funcs_body : tinstruction Smap.t;
    vars  : typ Smap.t;
    packages : Vset.t;
    (* ordre topologique de dépendance des structures *)
    order : string list }
```

qui retient toutes les informations dont on peut avoir besoin à propos du programme pour pouvoir le compiler plus tard. Cet environnement est ensuite conservé dans une table globale, permettant de le retrouver rapidement, quand il est importé dans un autre package par exemple.

3 La compilation

J'ai séparé le travail en deux étapes :

1. La construction d'un arbre de syntaxe abstrait de compilation
2. La transformation de cet arbre en code assembleur.

La raison pour laquelles j'ai fait ce choix est simples : séparer l'allocation des variables, le calcul l'espace de pile nécessaire à chaque fonction et autre détails dont nous allons discuter tout de suite, de la production effective de code assembleur.

3.1 La construction de l'arbre de compilation

Gestion des données statiques C'est à ce moment que je calcule la taille en mémoire des variables. Tous les types de bases font 8 octets et les structures ont pour taille la somme de la taille de leurs champs. C'est pour cette raison que dans l'environnement je conserve un ordre topologique sur les structures, afin de m'assurer que lorsque je calcule la taille de l'une d'entre elles, je connaisse bien la taille de toutes celles qu'elle contient.

La structure vide a la même taille qu'un entier. On aurait sûrement put règler le problème de manière plus subtile, en utilisant un seul octet par exemple, mais les journées ne font que vingt quatres heures et j'en perds déjà trop à tenter sans succès de combler le vide de mon existence pour ne pas le faire excessivement de celle de la structure vide. J'ai ajouté relativement à ce cas limite le test `exec/empty_struct.go`.

C'est aussi à ce moment que je construis les données qu'il faudrait conserver dans le segment `.data`. J'y ajoute essentiellement les chaines de caractères tapées en dur dans le code. J'en profite pour construire les formats d'affichage que je passerais à `printf`.

Pour réduire le nombre d'appels à la fonction `printf`, je construis un format pour chaque `print` en fonction du type de ses arguments. J'ai pris la précaution d'échaper les `%` qui pourraient se trouver dans une chaine passée en paramètre. De plus, lorsque la donné est connue statiquement, je l'ajoute directement au format.

Pour factoriser un peut de code entre les chaines de caractères et les formats, j'ai implémenté un foncteur `MakeSym` qui inclu le module `Hashtbl` de `Ocaml`. L'addition incrémente un compteur qui référence le nom de la donnée dans le segment data seulement si elle n'y est pas déjà (typiquement la chaine `\n`).

L'arbre de compilation C'est est essentiellement le même que précedement. J'ai ajouté une instruction `Cdefault` qui correspond à une déclation sans spécification de la valeur.

```
(** Stack ident *)
type sident = int

type cexpr =
  Cnil
| Cint     of int64
| Cident   of sident * typ
  (* ... *)
| Cprint   of (cexpr * typ) list * sident
| Cnew     of int * typ

type cinstruction =
  Cnop
| Cexpr    of cexpr * typ
| Cincr    of cexpr
| Cdecr    of cexpr
| Casgn    of cexpr * cexpr * typ
| Cdefault of sident list * typ
```

```
| Cdecl    of sident list * cexpr * typ
| Cblock   of cinstruction list
| Creturn  of cexpr * int * int
| Cfor     of cexpr * cinstruction
| Cif      of cexpr * cinstruction * cinstruction

type sz_cinstruction = cinstruction * int
```

Pour chaque variable locale, je lui associe une place sur la pile et je calcule en même temps que je construis l'abre de compilation l'espace nécessaire en mémoire à chaque fonction.

Échappement de variables Je gère l'échappement possible de variables après avoir construit l'arbre de compilation. Je me rend compte maintenant que c'est idiot car dans l'éventualité où une structure est déplacée de la pile au tas, alors l'espace qui lui est aloué sur la pile reste le même, alors qu'il n'y a maintenant plus qu'une référence vers celle-ci. Ça représente un certain gâchis d'espace mémoire, mais je me suis aperçu du problème trop tard pour le règler. Il marche parfaitement pour les types de 8 octets (sur une machine 64 bits).

La stratégie adoptée pour choisir quelles variables passer de la pile au tas est assez simple : si à un moment on manipule l'adresse de cette variable, alors on la place sur le tas. On remplace la variable par un pointeur vers la réelle variable et on modifie l'arbre de syntaxe abstraire en conséquence.

3.2 Production de code

À partir de l'arbre de compilation, la transcription en code assembleur est assez directe. Le code est essentiellement constitué des fonctions suivantes

```
type compile_info = {
    heap_alloc : Iset.t;
    is_main : bool;
    frame_size : int; }

val compile_expr : ?push_value:bool -> cexpr -> [ `text ] asm

val compile_instruction : compile_info -> cinstruction -> [ `text ] asm

val compile : ident -> env -> [ `text ] asm * [ `text ] asm
```

Le type `compile_info` transporte des informations sur la fonction en cours de compilation et dont je pourrais avoir besoin.

Convention de retour et d'appel Par défaut, la fonction `compile_expr` place la valeur de l'expression qu'elle calcule dans `%rax` si elle de taille 8, sur la pile sinon. On peut forcer l'expression à être écrite sur la pile ; c'est pratique dans certains cas, pour `printf` par exemple.

Les conventions d'appel sont celles du sujet. Les conventions de retour dépendent de la taille des types en question. Si le type est de taille 8 ou moins (si la fonction ne retourne rien), alors, comme pour `compile_expr`, la valeur de retour est placée dans `%rax`. Sinon elle est placée sur la pile à la place anciennement occupée par les arguments. C'est toujours l'appellant qui dépile les arguments, s'il en reste sur la pile toutefois.

Quand plusieurs valeurs sont retournées, alors la dernière est placée en dernière sur la pile. Ainsi, lors d'une composition, il n'y a rien à faire après le retour de la première fonction, les arguments sont déjà en place au sommet de la pile.

J'ai ajouté quelques tests sur l'implémentation des structures, car il n'y en avait aucun dans ceux fournis. En particulier les fichiers `exec/struct4.go` et `exec/struct5.go` qui testent respectivement que je copie correctement les structures lorsqu'elles sont passées en paramètres ou retournées, et que les tests d'égalités sur les structures sont corrects.

Affichage avec `printf` Pour l'affichage je fais directement un appel à la fonction `printf` de la `libc`. Il y a une petite subtilité. En effet, on ne peut pas utiliser directement les formats de `printf`, pour les booléens ou pour les pointeurs par exemple. J'ai donc implémenté une fonction

```
val prepare : int * [ `text ] asm ->
              cexpr * typ ->
              int * [ `text ] asm
```

qui pour une expression donnée avec son type, l'évalue et la change si c'est nécessaire. Par exemple, le code produit pour les types booléens est le suivant :

```
code ++ compile_expr ce ++
xorq (imm 0) !%rax ++ je l ++
pushq (ilab (true_string ())) ++
jmp el ++ label l ++
pushq (ilab (false_string ())) ++
label el
```

Ainsi, on affiche bien `true` ou `false` et non la valeur entière de l'expression. Il y a un comportement similaire pour les références.

J'ai cru comprendre que sur certains système d'exploitation, il fallait lors d'un appel à une fonction de la `libc`, et donc à `printf`, que la position de la pile soit alignée. Je n'ai pas eu ce problème personnellement et n'ai donc pas pris la peine de le faire puisque ce n'était pas demandé.

Les opérateurs booléens sont bien paresseux J'ai bien fait attention à ce détail. Pour m'assurer de ne pas l'oublier, j'ai ajouté le test `exec/lazy.go`.

4 Quelques petite extentions du sujet

4.1 Compiler plusieurs packages

J'ai légèrement étendu la syntaxe du `PetitGo` pour avoir la possibilité d'importer d'autres packages que `fmt`. La raison pour laquelle je l'ai fait est simplement que je trouvais amusant de pouvoir construire de petites bibliothèques en `PetitGo` qui s'utiliseraient les unes les autres.

Pour pouvoir importer un autre fichier, il faut compiler ce dernier en même temps que le fichier qui l'importe. Par exemple, pour utiliser des arbres binaires de recherches dans mon programme, je compile avec la commande

```
main.exe abr.go mon_programme.go
```

On notera que les fichiers doivent être importés dans un bon ordre, sans quoi ils ne seront pas compilé. De plus, un fichier définit un package [1] dont le nom est exactement celui donné au debut du fichier par la ligne

```
package abr
```

1. contrairement au langage `Go`

Le nom de ce package ne dépendra pas du nom du fichier ni de sa position relative dans l'arborescence de fichier [2].

Pour utiliser une fonction ou une structure du package importé, il faut les faire précéder du nom du package et d'un point. Comme, par exemple, dans le code suivant utilisant le fichier de test `abr.go`

```
var dico *abr.BST = nil
abr.add(&dico, 42)
abr.add(&dico, -1)
abr.print(dico); fmt.Print("\n")
```

À la compilation le nom des fonctions est préfixé par le nom du package auquel elles appartiennent (sauf pour la fonction `main` du package `main`). Et les packages sont simplements compilés dans le même ordre que celui dans lequel ils ont étés typés.

4.2 Gestion des erreurs

J'ai travaillé un petit peu plus pour afficher de belles erreurs, en particulier lors d'une erreur de typage. J'ai consigné dans le fichier `error.ml` un ensemble de fonctions qui gèrent le rendu des erreurs. L'ensemble des exceptions que l'éxécution du programe est suceptible de lever à un moment sont les suivantes. Elles sont chacune accompagnées d'un ensemble de fonctions levant ces exceptions avec un message personnalisé.

```
exception Error of string
exception Compile_error of position * string
exception Hint_error of position * string * string
exception Double_pos_error of position * position * string
exception Cycle_struct of string list
```

Lorsqu'une erreur survient, elle est localisé à une position du fichier passé en argument du compilateur. J'affiche donc la position au format demandé, suivit d'un message d'erreur et d'un petit bout du fichier correspondant au passage de l'erreur.

```
File "typing/bad/testfile-leftvalue-2.go", line 3, characters 24-25:
Error: invalid argument for &: has to be a left value.

 2:
 3: func main() { var x = &1 }
    ------------------------^--
```

Parfois, on a plus d'information lors de l'erreur. Par exemple, lorsqu'une variable ou une structure est déclarée plusieurs fois.

```
File "typing/bad/testfile-redeclared-1.go", line 4, characters 6-7:
Error: a structure with name ‘T‘ already exists.

 1: package main
 2: type T struct {}
    ^---------------
 3: func main() {}
 4: type T struct {}
```

2. pas de package `math/rand` par exemple

```
 3: func main() {}
 4: type T struct {}
    -----^----------
```

Pour ces types d'erreurs, il peut y avoir un problème quand la seconde position est originairement d'un autre fichier. Le problème n'est pas très difficile à résoudre, mais compte tenu du peu de pertinence que cela avait pour le projet je ne l'ai pas fait.

Quand l'erreur est sur un nom de variable (ou de structure ou de packages ou...) je regarde parmis les noms existants celui qui est le plus proche [3] et je propose un nom de variable qui pourrait convenir.

```
File "test.go", line 32, characters 7-8:
Error: unknown function ‘a‘.
Hint: did you mean ‘add‘ ?

 31:        x := (55 * i) % 34
 32:        abr.a(&dico, x)
     -----------^----------
 33:        abr.prt(dico)
```

4.3 Miscellanées

Optimisation du code produit Je n'ai rien fait de comparable à l'implémentation d'un compilateur optimisant dans le sens où nous l'avons vu dans le cours (j'ai commencé dans un élan de motivation pendant les fêtes, puis lâchement abandonné).

J'ai néanmoins proposé un petit, ridiculement petit, module d'optimisation du code produit. Nous conviendrons qu'il est plus présent pour montrer qu'on aurait put le faire et montrer vaguement comment que par nécessité ou désir de le faire.

Il optimise vaguement les additions, les soustractions et les opérateurs booléens. C'est à dire que s'il est capable (et encore seulement dans certains cas) de calculer la valeur statique d'un entier ou d'un booléen, alors il le fait. Il se permet aussi de supprimer les boucles dans lequelles on est sûr de ne jamais entrer, ainsi que les conditions inutiles

Production d'un exécutable Je me suis permis d'ajouter une option qui produit directement un exécutable. Simplement parce que je trouve que c'est quand même mieux d'avoir un compilateur qui est capable d'en produire un.

Sécurités à l'exécution Encore une fois, on parle ici plus d'une *preuve de concept* que d'une vraie extention. J'ai ajouté un petit bout de code qui affiche une erreur dans le cas où un malloc retourne le pointeur nul (voilà, c'est tout). On peut toutefois étendre le concept :

1. tester si l'addition d'entier produit un overflow
2. tester si on déréférence un pointeur nul
3. tester si on divise par zéro

C'est quand même plus agréable d'avoir un programme qui produit un message d'erreur compréhensible en cas de problème visible plutôt qu'une vilaine erreur de segmentation. Bien sûr, une fois qu'on est assuré que le programme est correct, alors on peut désactiver ces sécurités en passant en *wild mode*.

3. pour la distance minimum d'édition

5 Conclusion et améliorations possibles

La compilateur passe tous les tests. Je suis un peu déçu par le travail que j'ai rendu (je pense que ça s'est senti sur la fin du rapport). J'ai fait des choix dans mon implémentation du projet qui sont de mon point de vue très questionnables, bien qu'ils ne soient pas tout à fait déraisonnables pour autant. J'aurais aimé proposer quelque chose de plus abouti dans les extentions proposées (compilateur un peu plus optimisant, proposer une système de compilation des packages plus consistant avec Go ou encore implémenter toutes les sécurités). Je peux quand même dire que je suis content de l'avoir fait au moins une fois.

Malgrés tout ça me semble décent : en définitive, il y presque tout ce qu'il faut pour avoir un bon petit langage. Aussi, la suite serait d'implémenter un compilateur de MoyenGo très optimisant en PetitGo (et ne pas oublier de recompiler le compilateur MoyenGo avec lui même pour avoir un compilateur optimisé).