Rapport sur la deuxième partie du projet de compilation

Lucas Pesenti

On ajoute à la partie précédente le fichier `compiler.ml`, réalisant la production de code. Il utilise le module fourni `x86_64.ml`, auquel on a rajouté le support de l'instruction `idivl` permettant de faire des divisions sur 32 bits.

1 Retour des enregistrements

Les types enregistrements sont traités de manière analogue aux types classiques. Pour réaliser l'abstraction, on dispose d'une fonction `size_of_type` associant à chaque type sa taille en octets. Alors, toutes les opérations de copie sont réalisées, quel que soit le type sous-jacent en itérant une opération de copie élémentaires de 8 octets autant de fois que nécessaire. Un problème se pose cependant lorsque l'on veut retourner un enregistrement, le registre `%rax` ne suffit alors plus. On adopte en conséquence la convention suivante : toute fonction retournant une valeur d'un certain type la place dans la pile, juste avant ses arguments. Ainsi, le modèle d'un tableau d'activation pour une fonction devient :

Valeur de retour
Argument 1
⋮
Argument n
`%rbp` de la procédure père
Adresse de retour
`%rbp` appelant
Variables locales

⋮

Pour les procédures, on garde le même tableau d'activation, en enlevant l'espace alloué pour la valeur de retour.

2 Allocation dynamique

La construction `new` de Mini-Ada est compilée en appelant `malloc`, puis en initialisant la mémoire allouée à 0[1], un peu dans la même idée que la fonction `calloc` du C.

[1]Par exemple, le fichier test `bst.adb` nécessite une telle initialisation.

```
let sz = size_of_type env (Trecord i) in
movq (imm sz) (reg rdi) ++
call "malloc" ++
(* Initialize to 0 *)
movq (reg rax) (reg rdi) ++
iter (fun () -> movq (imm 0) (ind rdi) ++
                subq (imm wordsz) (reg rdi))
     (sz / wordsz) () ++
pushq (reg rax)
```

3 Opérations sur 32 bits

Une attention particulière a été portée à la gestion des opérations arithmétiques. En particulier, il fallait travailler sur 32 bits, alors que les données sont stockées sur 64 bits. Dans cette optique, on utilise les instructions suffixées par `l`, puis, pour récupérer une valeur sur 64 bits quand on veut l'empiler, on utilise l'instruction `movslq` permettant de faire la conversion en étendant correctement les signes. Par exemple, l'opération `+` est compilée de la manière suivante :

```
compile_expr env e1 ++ compile_expr env e2 ++
popq rcx ++ popq rax ++
addl (reg ecx) (reg eax)
movslq (reg eax) rax ++
pushq (reg rax)
```