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 Modifications du typeur

La première étape a consisté à modifier les passages du typeur qui ne s'adaptait pas tout à fait à ce qui était vraiment nécessaire à la production de code. On a veillé à faire aussi peu de changements de ce type que possible. En particulier, les calculs des décalages des variables locales sur le tableau d'activation sont effectués dans la dernière phase de compilation.

On en a profité pour uniformiser certains comportements, comme par exemple la transformation des listes de variables associées à un certain type en une liste de couples. On a également corrigé quelques oublis mineurs de types dans l'arbre typé.

2 Environnement de production de code

On a choisi de travailler directement sur l'arbre issu du typage. Les positions des variables sur la pile sont calculées à la volée [1]. En conséquence, il a fallu garder un certain nombre d'informations dans un environnement :

```
(* Environment data structure *)
type env = {
  (*
   * variable identifier -> (level, offset, size, by_reference)
   * in the current context.
   * *)
  vars: (int * int * int * bool) Smap.t;
  (* Current level *)
  lvl: int;
  (* Current offset *)
  ofs: int;
  (* Record offsets *)
```

[1] En particulier, on ne crée pas d'arbre intermédiaire. La raison essentielle est que les variables en Mini-Ada étant déclarées au début d'une fonction, il suffit d'une passe dans les déclarations pour calculer les tailles, puis d'une passe dans les instructions pour générer du vrai code.

```
  rec_ofs: (int * int) Smap.t SImap.t;
  (* Record size *)
  rec_sz: int SImap.t;
  (* Stack size in curent context *)
  stacksz: int;
  (* Offset of return value *)
  ret_ofs: int;
}
```

3 Gestion des labels

La production du code assembleur nécessite de produire des identificateurs uniques. Pour cela, un composant permet de produire des labels uniques pour les fonctions et les procédures. Pour cela, on associe à chaque identificateur un éventuel label courant. Par les règles de portée, on sait qu'une seule fonction ou procédure portant un identificateur donné peut exister à un certain point du code.

```
(* Functions labels handlers *)
let fct_labels = ref SImap.empty
let nb_labels = ref 0

let add_label i =
  fct_labels := SImap.add i !nb_labels !fct_labels;
  incr nb_labels

let get_label i =
  "_f_" ^ (string_of_int (SImap.find i !fct_labels))
```

En particulier, on utilise le fait que les identificateurs de Mini-Ada ne peuvent pas commencer par un tiret bas.

On a également besoin de labels pour des sauts lors de conditions ou de boucles, par exemple. Dans ce cas, étant donné qu'une utilisation immédiate suffit et qu'on n'a pas de nom canonique associé, on crée à la volée des identificateurs uniques.

```
(* Return a fresh identifier for a new label *)
let new_label =
  let cnt = ref (-1) in
  function () -> incr cnt; "." ^ (string_of_int !cnt)
```

4 Taille des types

Pour éviter les contraintes d'alignement, on stocke la totalité des types de base sur 8 octets. Évidemment, un type enregistrement ne fait pas nécessairement 8 octets : sa taille égale la somme des tailles de ses membres (il n'y a pas besoin de bits de bourrage).

5 Types accès et références

Les paramètres `in out` et les types accès sont traités de manière similaires : lorsqu'une indirection est nécessaire, on rajoute une instruction du type `movq %rsi, 0(%rsi)`.

Les accès à une variable en général sont également abstraits. En particulier, la fonction `compile_access` permet de mettre dans `%rsi` l'adresse de la valeur gauche associée à une certaine expression[2].

6 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.

7 Allocation dynamique

La construction `new` de Mini-Ada est compilée en appelant `malloc`, puis en initialisant la mémoire allouée à 0[3], un peu dans la même idée que la fonction `calloc` du C.

```
1 let sz = size_of_type env (Trecord i) in
2 movq (imm sz) (reg rdi) ++
3 call "malloc" ++
```

[2]Comme on suppose que dans la phase de typage s'est correctement déroulée, on sait que lors d'une affectation, le membre de gauche est d'une forme très particulière

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

```
(* 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)
```

8 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)
```