Projet de Programmation : Compilateur C–

Younesse Kaddar

2016-01-08

Abstract

Projet de Programmation : compilateur C–, avec support des exceptions.

Méthode utilisée pour compiler les exceptions

Modalités de rendu

http://younesse.net/Programmation/Projet-Compilateur

Younesse Kaddar

Section introductive :

1. J'ajoute le nom des exceptions à l'environnement (une Map OCamL) de chaînes de caractères global `env_strings` (qui associe aux chaînes de caractère leur labels en assembleur).
2. J'utilise une variable globale `exception_not_caught` (qui vaudra `1` (pour "vrai") ou `0` (pour "faux")), initialisée à `0`, et dont la fonction sera de signaler un paquet d'exception non rattrapé (notamment dans une fonction appelée : pour que la fonction appelante puisse alors tenter de rattraper le paquet après le `call` de la fonction appelée).

```
let env_strings = fold_left add_str_to_env_from_decl (StringMap.empty) decl_list in
let exception_not_caught = (genlab "exception_not_caught") in
(
  (* ... *)

  (* declaring the global variable used as a flag "exception_not_caught" *)
  Printf.fprintf out ".comm %s,8,8\n" exception_not_caught;

  (* ... *)
```

```
  (* initializing to false the global variable used as a flag "exception_not_caught" *)
  Printf.fprintf out "\tmovq $0, %s\n" exception_not_caught;

  (* Compilation des fonctions avec compile_code *)

)
```

L'idée générale est que j'utilise :

- un environnement local d'exceptions `env_exceptions` pour transmettre les handlers d'exception susceptibles d'être rattrapés (*i.e* correspondant au corps du `CTRY` courant) de l'appelant à l'appelé (récursivement).
- le registre `%rcx` (ni caller-saved, ni callee-saved) et la variable globale `exception_not_caught` pour transmettre les paquets d'exception de l'appelé à l'appelant (puisque les portées des clauses `catch` sont dynamiques).

Dans la fonction `compile_code` :

3. Les codes dont le traitement a été modifié sont :

- `CRETURN(loc_expr_option)` :
  - on vérifie que le `return` courant est dans un `finally` ou non avec la variable `is_in_finally` : le cas échéant, tout paquet d'exception est dès lors "écrasé" par le paquet de retour de ce finally, et la variable globale `exception_not_caught` est mise à `0`.

```
ocaml   if is_in_finally then     Printf.fprintf out "\tmovq
$0, %s\n" exception_not_caught;
```

  - On compile l'expression du `return`, dont la valeur est alors stockée dans le registre `%rax`, puis on sauvegarde cette valeur à l'aide du registre **callee-saved** `%r13`

```
(match loc_expr_option with
 | Some (_, expr) -> compile_expr current_fun endFunctionLabel finallyLabel env_var env
 | None -> ());

(* saving %rax (in case of a "finally" without packet)
   in the calle-saved register %r13 *)
Printf.fprintf out "\tmovq %%rax, %%r13\n";
```

  - On récupère le label `finallyLabel` de l'éventuel "finally" à exécuter, et on génère un label "point de retour" (sauvegardé dans le registre **callee-saved** `%rbx`), pour revenir (après le "finally", éventuellement)

à l'emplacement où l'on est et renvoyer la valeur sauvegardée précédemment dans `%r13` au cas où le "finally" ne lance ni paquet de retour, ni paquet d'exception.

```
(match finallyLabel with
 | Some str_finally -> (
     (* in case exceptionLabel is a "finally", the "return point" label is stored
        in the callee-saved register %rbx *)
     let returnPointLabel = genlab (current_fun^"returnPoint") in
     Printf.fprintf out "\tmovq $%s, %%rbx\n" returnPointLabel;

     Printf.fprintf out "\tjmp %s\n" str_finally;

     Printf.fprintf out "%s: # return from a 'finally' without 'packet'\n" returnPointL

     Printf.fprintf out "\tmovq %%r13, %%rax\n";
   )
 | None -> ());
```

- `CTHROW(str, (_, expr))` :

  Je mets le nom de l'exception dans `%rcx`, sa valeur dans `%rax`, puis :

  - si le handler lancé est connu (*i.e* appartient à `env_exceptions` : la `Map` des handlers d'exception susceptibles d'être rattrapés), la valeur `0` est attribuée à `exception_not_caught`
  - *sinon*, la valeur `1` est attribuée à `exception_not_caught`

  Enfin, je génère un label de "point de retour" (sauvegardé dans le registre **callee-saved** `%rbx`) et j'exécute le "finally" (s'il existe), avec les mêmes précautions de sauvegarde de `%rax` - mais aussi de `%rcx`, maintenant - que précédemment.

- `CTRY((_, code), str_str_locCode_list, loc_code_option)` :

  Le seul point délicat est le "finally" : au cas où le code du "finally" ne revoie pas de paquet de retour ou d'exception, il faut bien penser à se brancher sur le label de "point de retour" contenu dans `%rbx` (lequel n'a pas été altéré par d'éventuels appels de fonction dans le code du "finally", puisque `%rbx` est callee-saved) avec un `jmp *%rbx`.

Dans la fonction `compile_expr` :

4. La seule expression dont le traitement a été modifié est l'appel de fonction `CALL(str, loc_expr_list)`.

   Après avoir appelé la fonction appelée, on tire profit de la variable globale `exception_not_caught`, du registre `%rcx`, et du registre `%rax` pour (respectivement) :

- tester la présence d'un paquet d'exception non rattrapé par la fonction appelée
- et le cas échéant : récupérer le nom et la valeur de l'exception

Si l'exception peut être rattrapée par l'appelant, on se branche alors sur la clause `catch` appropriée

Sinon, on termine la fonction appelante et on propage le paquet d'exception.

```
(* is exception_not_caught equal to 0 ?
   in other words :  has every exception raised by the function called
   already been handled ?
*)
Printf.fprintf out "\tcmpq $0, %s\n" exception_not_caught;

let endExceptionCaught = (genlab (current_fun^"_endExceptionCaught")) in
(
  Printf.fprintf out "\tje %s\n" endExceptionCaught;

  (* if an exception has been caught *)
  StringMap.iter (fun str_excep excep_label ->
      (
        let stri = StringMap.find str_excep env_strings in
        Printf.fprintf out "\tmovq $%s, %%rdx\n" stri;

        Printf.fprintf out "\tcmpq %%rcx, %%rdx\n";
        Printf.fprintf out "\tje %s\n" excep_label;
      )
    ) env_exceptions;

    (match finallyLabel with
      | Some str_finally -> (
        (* in case exceptionLabel is a "finally", the "return point" label is sto
           in the callee-saved register %rbx *)
        let returnPointLabel = genlab (current_fun^"returnPoint") in
          Printf.fprintf out "\tmovq $%s, %%rbx\n" returnPointLabel;
        Printf.fprintf out "\tjmp %s\n" str_finally;
        Printf.fprintf out "%s: # return from a 'finally' without 'packet'\n" retu
        )
      | None ->());

  (* exception still not caught by the caller : end function *)
  Printf.fprintf out "\tjmp %s\t# exception still not caught by the caller\n" endE

  Printf.fprintf out "%s:\n" endExceptionCaught;
```

Avantages de cette méthode :

- L'usage de `exception_not_caught` et de `%rcx` permet de savoir vers quelle clause `catch` se brancher en *runtime*.
- Si les conventions d'utilisation des registres callee-saved/caller-saved sont respectées : le nom de l'exception lancée (stocké dans `%rcx`) et sa valeur (stockée dans `%rax`) ne peuvent pas être perdus, de même que le label de "point de retour" (stocké dans `%rbx`) utilisé à la fin de l'exécution d'un "finally" sans paquets.
- L'astuce de l'adresse de retour du "finally" permet de respecter "naturellement" la sémantique, dans la mesure où si un "finally" renvoie un paquet, il l'emporte sur le paquet courant, sinon : celui-ci est transmis.

Inconvénients :

- En cas de fonctions $f_1, , f_n$ imbriquées (f_1 appelle f_2 qui appelle … qui appelle f_n) : si f_n renvoie un paquet d'exceptions qui ne peut être rattrapé que par f_1, le paquet entraînera un dépilement de toutes les fonctions imbriquées successivement (il sera non rattrapé dans f_{n-1}, puis non rattrapé dans f_{n-2}, etc … jusqu'à finalement être rattrapé par f_1), ce qui est très lourd. Je pense qu'avec une implémentation différemment pensée, il est possible de rattraper directement le paquet dans f_n.
- L'emploi d'une variable globale, de nouveaux "rôles" attribués aux registres `%rcx` et `%rbx`, et d'un nouvel environnement d'exceptions ajoute une touche d'hétérogénéité peu commode au code