

TP de Compilation n° 2 : De Clight à Cminor

13 Février 2014

Vous pouvez trouver tous les sujets au format électronique sur internet à l'adresse :

<http://www.pps.univ-paris-diderot.fr/~pboutill/CompilM1/>.

Ce sujet propose d'étudier la première phase de compilation du compilateur CerCo : c'est la traduction du programme source du langage Clight vers le langage Cminor.

Ce sujet se base sur la version taggée TP2 du code CerCo. Vous pouvez l'obtenir en exécutant dans le dossier CerCo : `git checkout TP2`

Pensez à tester régulièrement votre code !

I) Présentation

La syntaxe abstraite du langage Clight se trouve dans le fichier `src/clight/clight.mli` ; celle du langage Cminor se trouve dans le fichier `src/cminor/cminor.mli`. Dans un premier temps, nous vous conseillons de lire et d'essayer de comprendre ces deux fichiers d'introduction.

a) Différences entre Clight et Cminor

Il y a deux différences principales entre le langage Clight et le langage Cminor. La première est une simplification des *structures de contrôle*. Par exemple, il n'y a plus de `while` en Cminor : il n'y a qu'une instruction de boucle infinie que l'on peut interrompre. La seconde différence est l'*explicitation des variables*. Lors de la traduction d'une fonction, une variable Clight peut devenir en Cminor :

- soit un *offset* dans la pile de la fonction si la variable était locale ou un paramètre de fonction en Clight, à quoi son adresse est pointée et à quoi son type est complexe (tableau, structure, union, etc) ;
- soit une variable locale si la variable était locale en Clight à quoi elle ne correspond pas au cas précédent ;
- soit un paramètre de fonction si la variable était un paramètre de fonction en Clight à quoi elle ne correspond pas aux cas précédents ;
- soit une variable globale si la variable était déjà globale en Clight à quoi elle ne correspond pas aux cas précédents.

La première phase de compilation de CerCo consiste à traduire un programme Clight de type OCaml

`Clight.program`, en un programme Cminor de type OCaml `Cminor.program`. Cette traduction est faite par la fonction `translate` du module `ClightToCminor` défini dans le répertoire `src/clight`. Le fichier qui nous intéresse donc est `src/clight/clightToCminor.ml`.

Dans ce fichier, des portions de code ont été remplacées par des `assert false`. Vous pouvez les identifier facilement car ils sont suivis du commentaire `TODO M1`. La suite de ce sujet vous expliquera comment rétablir les portions de code désactivées.

b) Tester votre code

Pour tester votre code, vous pouvez par exemple compiler un programme C en Cminor, et évaluer le résultat de l'interprétation. Supposons que vos fichiers C testés soient nommés *fichier1.c*, *fichier2.c*,

tc. Pour les compiler en Cminor, téléchargez CerCo après vos modifications, vous devez tout d'abord compiler le code source de CerCo. Pour cela, placez-vous à la racine de CerCo puis exécutez la commande : `make`. Ensuite, la compilation vers Cminor suivie de l'interprétation se font avec la commande :

```
./cc -l Cminor -i fichier1.c fichier2.c ...
```

II) Structures de contrôle

La traduction des instructions Clight se fait principalement dans la fonction `f_stmt`. Cette traduction peut éventuellement créer de nouvelles variables locales, c'est pourquoi elle retourne une liste de variables Cminor nouvellement créées ainsi que l'instruction qui est le résultat de la traduction.

1. **Séquence** (constructeur `Clight.Ssequence`). La séquence de deux instructions Clight se traduit simplement en la séquence de deux instructions Cminor. On dispose d'ores et déjà dans l'environnement du résultat `stmt1` (respectivement `stmt2`) de la traduction de la première (respectivement) instruction de la séquence Clight d'origine. Le résultat de la traduction de la séquence est donc un couple formé d'une liste vide (si vous n'avez créé aucun variable supplémentaire) et de la séquence Cminor de `stmt1` et `stmt2`.
2. **Conditionnelle** (constructeur `Clight.Sifthenelse`). Une conditionnelle Clight se traduit en une conditionnelle Cminor. Pour effectuer la traduction, vous disposez dans l'environnement du résultat `e` de la traduction de l'expression conditionnelle, ainsi que du résultat `stmt1` (respectivement `stmt2`) de la traduction des instructions du cas vrai (respectivement faux) de l'instruction Clight d'origine.
3. **Boucle while** (constructeur `Clight.Swhile`). Informellement, la traduction de l'instruction Clight `while (e) { stmt }` en Cminor est :

```
block {
  loop {
    if (!e') exit 0;
    block { stmt' }
  }
}
```

où `e'` est la traduction de `e` et `stmt'` est la traduction de `stmt`. Dans votre environnement de traduction, vous disposez de `e` qui est le résultat de la traduction de l'expression de boucle, et de `stmt` qui est le résultat de la traduction du corps de la boucle.

4. **Appel de fonction** (constructeur `Clight.Scall`). On considère le cas où la fonction retourne une valeur. En syntaxe C concret, on a donc une instruction de la forme suivante à traduire :

```
e = f(arg1, arg2, ...);
```

En Clight, `e` peut avoir une forme quelconque. Mais en Cminor, le constructeur `Cminor.Stcall` pour l'appel de fonction ne peut retourner le résultat d'un appel de fonction qu'à l'intérieur d'une variable locale. Il faut donc créer une nouvelle variable locale dont le type sera la traduction en Cminor du type Clight de `e`. Ensuite, on peut créer l'instruction Cminor qui consiste à faire l'appel de fonction et à retourner le résultat dans la variable nouvellement créée. Enfin, on finit par une instruction qui va affecter la valeur de cette variable à la traduction de `e`. Soit `tmp` la variable fraîchement créée. Les instructions Cminor obtenues auront la forme informelle suivante :

```
tmp = f'(arg'_1, arg'_2, ...);
```

```
e' = tmp;
```

où e' , f' , arg'_1 , arg'_2 , etc, sont les traductions de e , f , arg_1 , arg_2 , etc. L'environnement de traduction contiendra e qui est l'expression `Clight` à laquelle on affecte le retour de la fonction, f qui est la traduction de l'expression qui représente la fonction à appeler, $args$ qui est la traduction de la liste d'arguments. De plus, pour vous aider, vous pouvez utiliser les fonctions suivantes qui sont déjà définies dans le fichier `src/clight/clightToCminor.ml` :

```
— clight_type_of : retourne le type d'une expression Clight;
```

```
— sig_type_of_c_type : traduit un type Clight en un type Cminor;
```

```
— fresh : retourne un nouveau nom de variable;
```

```
— c_ll_sig : retourne une signature Cminor étant donné un type de retour et une liste d'expressions Cminor (les arguments);
```

```
— ssign : retourne une instruction Cminor qui affecte une expression Cminor au résultat de la traduction d'une expression Clight.
```

Au final, n'oubliez pas de retourner la variable qui a été créée en plus de la traduction de l'instruction !

III) Expressions

La traduction des expressions `Clight` se fait à partir dans la fonction `f_expr`. Cette traduction a un argument nommé `v_r_locs` : c'est un environnement qui associe à chaque variable son genre (global, en pile, argument de fonction, etc).

1. **Expression ternaire** (constructeur `Clight.Econdition`). Une expression ternaire `Clight` se traduit en une expression ternaire `Cminor`. Pour effectuer la traduction, vous disposez dans l'environnement de expressions `Cminor` e_1 , e_2 et e_3 qui sont la traduction des sous-expressions de l'expression `Clight` d'origine, ainsi que du type `Cminor` `t_cminor` du résultat.
2. **Et booléen** (constructeur `Clight.Endbool`). L'expression `Clight` $e_1 \ \&\& \ e_2$ se traduit en `Cminor` en les expressions ternaires imbriquées $e'_1? \ (e'_2? \ 1 : 0) : 0$, où e'_1 et e'_2 sont la traduction de e_1 et e_2 . Le type `Cminor` du résultat est le même que celui des sous-expressions soit `t_cminor`. Vous pouvez utiliser la fonction déjà définie `cst_int` qui permet de créer une expression `Cminor` à partir d'un constant entier et d'un type `Cminor`.
3. **Variable**. La traduction d'une variable se fait par la fonction `tr_nsl te_ident`. C'est dans cette fonction que le genre d'une variable va être particulièrement important. Le type `v_r_locations` déjà défini associe à chaque variable `Clight` un genre ainsi que sont les types `Clight`. Le genre représente l'un des cas suivants :
 - variable global (constructeur `Glob l`);
 - paramètre de fonction (constructeur `P r m`);
 - paramètre de fonction en pile (constructeur `P r mSt ck`);
 - variable local (constructeur `Loc l`);
 - variable local en pile (constructeur `Loc lSt ck`).
 Soit x une variable `Clight`; alors sa traduction est la suivante en fonction de son genre :

G nr	Traduction
Loc l ou P r m	Variabl local Cminor (construct ur Cminor.Id)
Loc lSt ck ou P r mSt ck (quand l typ d la variabl st <i>simple</i>)	L ctur à l'adr ss n pil (construct ur Cminor.Mem)
Loc lSt ck ou P r mSt ck (quand l typ d la variabl n' st pas <i>simple</i>)	Adr ss n pil
Glob l (quand l typ d la variabl st <i>simple</i>)	L ctur à l'adr ss du symbol global (construct ur Cminor.Mem)
Glob l (quand l typ d la variabl n' st pas <i>simple</i>)	Adr ss du symbol global

Pour effectuer l'ensemble des opérations nécessaires à la connaissance du g nr ou à la construction de l'expression Cminor résultat, vous pouvez vous servir des fonctions suivantes déjà définies :

- `find_v r_locs` : retourne l'ensemble des g nr et l'ensemble des typ associés à un variabl local ;
- `is_simple_ctype` : retourne si un typ est simple (entier, flottant ou pointeur) ou non ;
- `dd_st ck` : ajoute un offset au pointeur de pil ;
- `quantity_of_ctype` : transforme un typ Clight en une *quantité mémoire* ;

Enfin, le constructeur `AST.Cst_ ddrsymbol` vous permettra de construire l'adresse d'un symbol global.