Compilateur de C--

Yohann D'ANELLO

`MCC` est un compilateur du langage `C--` codé en OCamL. Il prend en entrée un fichier source codé en `C--`, puis le traduit en code assembleur (Intel x86_64), avant de l'assembler en exécutable machine. La sémantique de `C--` est disponible ici : http://www.lsv.fr/~goubault/CoursProgrammation/prog1_sem1.pdf

Un analyseur syntaxique commence par analyser le code, et le traduire en un *Abstract Syntax Tree* (AST, arbre de syntaxe abstrait). Le compilateur vient ensuite dérouler l'AST et produire le code assembleur nécessaire.

La première chose effectuée par le compilateur est de récupérer la liste des fonctions déclarées par le code. Cela permet de savoir quelles sont les fonctions qui renvoient un entier codé sur 32 ou sur 64 bits, avec les fonctions système `malloc`, `calloc`, `realloc`, `fopen` et `exit`. Ensuite, le code est compilé.

À la lecture du code, le compilateur dispose d'un environnement transmis et mis à jour à chaque appel de fonctions. Un environnement est modélisé par un 9-uplet contenant un compteur de nombre de labels déclarés, la liste des chaîne de caractères déjà rencontré ainsi que leur nombre, la liste des noms des variables globales, un booléen indiquant si le compilateur est actuellement en train de déclarer des paramètres d'une fonction ou non, un dictionnaire des variables locales indiquant à tout nom sa place sur la pile ainsi que la liste des fonctions disposant d'un retour sur 64 bits.

Déclaration d'une variable globale

Les éléments les plus hauts dans l'AST sont les déclarations de variables globales et de fonctions. Pour déclarer une variable, le compilateur ajoute uniquement dans l'environnement le nom de la variable globale déclarée et ne produit aucun code assembleur. Il attendra d'avoir compilé tout le code avant de déclarer les variables globales dans la section `.data` via l'instruction `.comm <NAME>, 8, 8`.

Déclaration d'une fonction

Le compilateur commence par déclarer créer un label vers la fonction, via `<NAME>:`. Ensuite, la fonction va être parcourue une première fois afin d'estimer la place nécessaire à allouer sur la pile pour déclarer les variables. Pour cela, chaque paramètre et chaque variable déclarée (y compris dans les sous-blocs) compte pour 8 octets), afin de garantir d'avoir toujours de la place pour les besoins nécessaires. Ensuite, l'instruction `ENTERQ $N, 0` est ajoutée, où `N` est le plus petit multiple de 16 supérieur ou égal à la place nécessaire pour la fonction. Ensuite, les paramètres sont déclarés, puis c'est au tour du code de la fonction. On convient que la valeur de retour de la fonction doit se trouver dans `%rax`. Enfin, l'instruction `LEAVEQ` permet d'effectuer l'instruction inverse de `ENTERQ`, et donc de remettre `%rbp` et `%rsp` à leurs bonnes valeurs. L'instruction `RETQ` suit ensuite, et va à l'instruction suivante.

Déclaration d'une variable locale, d'un paramètre

Lors de la déclaration d'une variable locale, celle-ci est ajoutée à l'environnement. En mémoire est conservée la position de son adresse relativement à `%rsp`, qui vaut alors la première place libre sur la pile. L'adresse de la `n`-ième variable locale est alors `-n(%rbp)`. S'il s'agit d'un paramètre, alors cela implique que le paramètre est déjà initialisé, et donc on ajoute une instruction qui permet de récupérer la valeur du paramètre dans le bon registre, ou bien à la bonne position sur la pile s'il s'agit au moins du septième paramètre.

Évaluation d'un morceau de code

Il existe 5 types de morceaux de code : les blocs, les expressions, les tests conditionnels `if`, les boucles `while` et les valeurs de retour `return`.

Les blocs de code

Format : `CBLOCK(declaration list, code)`

Un bloc de code commence par la déclaration des variables locales du bloc. Chaque variable est déclarée une à une, mettant à jour successivement l'environnement courant. Le code du bloc est ensuite exécuté. À la fin du bloc, les variables locales de l'environnement sont remplacées celles présentes avant l'entrée du code. Le reste est conservé.

Les expressions

Format : `CEXPR(expression)`

Un bloc d'expression évalue alors une expression. Une fois évaluée, la valeur de retour est toujours envoyée dans `%rax`. Il existe 11 types d'expression : l'utilisation de variables, l'utilisation de constantes entières, l'utilisation de chaîne de caractères, l'affection dans une variable, l'affectation dans un tableau, l'appel d'une fonction, une opération unaire (opposé, négation binaire, {post,pré}-{in,dé}crémentation), une opération binaire (multiplication, division, modulo, addition, soustraction, accès à l'élément d'un tableau), la comparaison de deux éléments (infériorité stricte, infériorité large, égalité), les conditions ternaires et enfin les séquences d'expression.

Utilisation de variables

Format : `VAR(name)`

Une ligne d'assembleur est ajoutée, qui va alors chercher dans l'environnement la position sur la pile de la variable appelée si elle est locale, sinon donner son nom directement, pour placer le contenu dans `%rax`.

```
MOVQ -24(%rbp), %rax
MOVQ stdout(%rip), %rax
```

Utilisation de constantes entières

Format : `CST(value)`

La constante indiquée est directement enregistrée dans `%rax`.

```
MOVQ $42, %rax
```

Utilisation de chaînes de caractères

Format : `STRING(string)`

La chaîne correspondante est ajoutée à l'environnement. Une optimisation du compilateur permet de ne pas enregistrer des chaînes de caractères déjà existantes. Après avoir compilé le code, dans la section `.data`, toutes les chaînes de caractères sont ajoutées au code assembleur, sous le label `.strN` où `N` est le numéro de la chaîne, par ordre d'apparition, via l'instruction :

```
.strN:
.string STR
.text
```

Le label en question est alors affecté à `%rax` : `MOVQ $.strN, %rax`

Affection dans une variable

Format : `SET_VAR(name, expression)`

L'expression à affecter est évaluée, puis le résultat (dans `%rax`) est affecté dans la variable.

```
MOVQ %rax, -24(%rbp)
```

Affectation dans un tableau

Format : `SET_ARRAY(name, expression, expression)`

La première expression est d'abord évaluée, puis mise sur la pile. La seconde expression est ensuite évaluée, et le résultat est alors dans `%rax`. La valeur mise sur la pile est ensuite dépilée dans `%rbx`. `%rbx` contient alors l'indice du tableau et `%rax` la valeur à affecter. On récupère ensuite l'adresse de la case désirée, via des additions, puis on place le contenu `%rax` dans la bonne case mémoire.

```
MOVQ $1, %rax
PUSHQ %rax
MOVQ $4, %rax
POPQ %rbx
MOVQ -24(%rbp), %rdx
LEAQ 0(, %rbx, 8), %rbx
ADDQ %rbx, %rdx
MOVQ %rax, (%rdx)
```

Appel d'une fonction

Format : `CALL(name, parameter list (expression list))`

On commence par évaluer chacun des arguments, de droite à gauche, et les placer sur la pile un à un. Les (au plus) six premiers sont ensuite dépilés et mis dans l'ordre dans `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`. Par respect de la norme `C`, on fixe `%rax` à `0`, puis on appelle la fonction. Une fois l'appel terminé, on dépile les arguments résiduels éventuels. Si jamais la fonction n'est pas dans la liste des fonctions ayant un retour sur 64 bits, on étend le signe de `%eax` dans `%rax`, via l'instruction `CLTQ`.

```
MOVQ $2, %rax
PUSHQ %rax
MOVQ $.str1, %rax
PUSHQ %rax
POPQ %rsi
POPQ %rdi
MOVQ $0, %rax
CALLQ printf
CLTQ
...
.str1:
.string "Valeur de deux = %d\n"
.text
```

Cette suite d'instruction assembleur modélise l'appel `printf("Valeur de deux = %d\n", 2);`.

Opérateur unaire

Format : `OP1(optype, expression)`

L'expression est évaluée (le résultat est alors dans `%rax`), puis traitée.

Opposé

Une seule instruction suffit : `NEGQ %rax`

Négation logique

De même, il suffit d'une instruction assembleur : `NOTQ %rax`

{Post,Pré}-{in,dé}crémentation

Si on est en post-{in,dé}crémentation, on commence par empiler la valeur de `%rax`, qu'on dépilera plus tard, afin de renvoyer la bonne valeur. Sinon, d'abord on {in,dé}crémente, puis on met dans `%rax` la valeur souhaitée.

Une telle opération est, selon la sémantique `C--`, soit de la forme `s++` où `s` est une variable, soit de la forme `t[e]++` où `e` est une expression et `t` une variable. Dans le premier cas, on se contente d'incrémenter ou de décrémenter la variable via `INCQ` ou `DECQ`. Dans le second cas, on procède de la même manière que l'affectation dans un tableau en récupérant la bonne adresse, puis on {in,dé}crémente la valeur associée.

Opérateur binaire

Format : `OP2(optype, expression1, expression2)`

La deuxième expression est d'abord évaluée (en accord avec la sémantique de `C--` qui suggère de toujours évaluer de droite à gauche), puis la valeur est placée sur la pile. La première expression est ensuite évaluée, dont le résultat est dans `%rax`. On récupère ensuite l'évaluation de la seconde expression dans `%rbx`.

Multiplication, addition, soustraction

L'instruction `IMUL %rbx, %rax` permet directement de multiplier `%rbx` par `%rax` et de placer le résultat dans `%rax`, ce qui est ce que nous voulions. Les instructions `ADDQ` et `SUBQ` permettent la même chose pour l'addition et la soustraction.

Division, modulo

On commence par étendre le signe de `%rax` dans `%rdx` via l'instruction `CQO`. On ajoute ensuite l'expression `IDIVQ %rbx`, qui effectue la division euclidienne de `%rdx:%rax` (nombre vu comme la concaténation des deux registres sur 128 bits) par `%rbx`, et stocke le quotient dans `%rax` et le reste dans `%rdx`, selon la sémantique de `C--`. Selon les cas, on met la bonne valeur dans `%rax`, puis pour des raisons de sécurité on remet `%rdx` à `0`.

Accès dans un tableau

Comme précédemment, on récupère l'adresse de la bonne case mémoire, puis on place le contenu dans `%rax` :

```
LEAQ (0, %rbx, 8), %rbx
ADDQ %rbx, %rax{.asm}\newline MOVQ (%rax), %rax
```

Comparaison

Format : `CMP(cmptype, expression1, expression 2)`

On évalue la première expression dans `%rbx`, puis la seconde dans `%rax`. On compare ensuite `%rax` à `rbx`. Puis, selon les cas (`JL` si l'inégalité est stricte, `JLE` si l'inégalité est large, `JE` si on veut l'égalité), on fait un saut vers le prochain label disponible. On se débrouille ensuite pour mettre `1` dans `%rax` si la comparaison est concluante, `0` sinon.

```
CMPQ %rax, %rbx
JE .destjump1
MOVQ $0, %rax
JMP .destjump2
.destjump1:
MOVQ $1, %rax
.destjump2:
```

Condition ternaire

Format : `EIF(expression1, expression2, expression3)`

On évalue d'abord la première expression, qu'on compare à 0. S'il y a égalité, alors on saute vers un futur label où on évaluera la troisième expression (la partie `else`). Sinon, alors on évalue la deuxième expression, où on ajoute un saut vers la fin de la condition.

Cette suite d'instruction simule `1 ? 5 : 7` :

```
MOVQ $1, %rax
CMPQ $0, %rax
JE .destjump1
MOVQ $7, %rax
JMP .destjump2
.destjump1:
MOVQ $5, %rax
.destjump2:
```

Séquence d'expression

Format : `ESEQ(expression list)`

Cette expression se contente d'évaluer les sous-expressions et de mettre à jour l'environnement au besoin.

Instruction conditionnelle

Format : `CIF(expression, code1, code2)`

On procède de la même manière que pour les expressions ternaires, à la différence près qu'on compile des blocs de code au lieu d'évaluer des expressions.

Boucles

Format : `CWHILE(expression, code)`

On commence par créer un label en haut de la boucle. On en rajoutera un aussi en fin de boucle. On évalue ensuite l'expression, qu'on compare ensuite à `0`. Si la comparaison est concluante, alors on saute directement à la fin de la boucle. Juste avant la fin de boucle, on saute immédiatement en haut de la boucle.

```
.whileloop1:
MOVQ -8(%rbp), %rax
CMPQ $0, %rax
JE .endloop1
# code
JMP .whileloop1
endloop1:
```

Valeurs de retour

S'il n'y a pas de valeur de retour, alors on ne fait rien. La précédente valeur de `%rax` servira de valeur de retour, en accord avec la sémantique qui autorise n'importe quelle valeur de retour si non précisée. Si non, alors on évalue la valeur de retour, qui sera directement dans `%rax`. On ajoute ensuite les instructions `LEAVEQ` et `RETQ{.asm}`, qui permettent de rétablir les précédentes valeurs de `%rsp` et de `%rbp` et de sauter à la nouvelle instruction. Il se peut qu'il y ait redondance avec les instructions ajoutées par la déclaration de la fonction, mais cela n'est pas un problème car ces instructions ne seront tout simplement jamais exécutées. Cela évite le problème d'absence d'instruction `return` dans le code `C`.

Optimisations possibles

Certaines optimisations pourraient être réalisables, notamment celles qu'effectue `GCC` : on pourrait par exemple ne pas avoir à systématiquement affecter le contenu d'une variable dans `%rax`, et utiliser directement l'adresse de la variable. Ce type d'optimisation nécessiterait néanmoins une meilleure maitrise de l'AST, quitte à le parcourir plusieurs fois, pour vérifier si certaines précautions sont nécessaires ou non.

En espérant que ce compilateur vous sera d'une grande aide :)

Exemple de compilation

Code simple :

```
int main(int argc, char** argv) {
int i;
i = 0;
i = (i++ - --i) + 1 + i;
printf("Valeur de i = %d\n", i);
return i;
}
```

Code assembleur généré :

```
.section   .text
.global main
main:
ENTERQ      $32,        $0
MOVQ        %rdi,       -8(%rbp)    # argc
MOVQ        %rsi,       -16(%rbp)   # argv
MOVQ        $0,     %rax
MOVQ        %rax,       -24(%rbp)
MOVQ        -24(%rbp),  %rax        # i
PUSHQ       %rax
MOVQ        $1,     %rax
PUSHQ       %rax
DECQ        -24(%rbp)
MOVQ        -24(%rbp),  %rax        # i
PUSHQ       %rax
MOVQ        -24(%rbp),  %rax        # i
INCQ        -24(%rbp)
POPQ        %rbx
SUBQ        %rbx,       %rax
POPQ        %rbx
ADDQ        %rbx,       %rax
POPQ        %rbx
ADDQ        %rbx,       %rax
MOVQ        %rax,       -24(%rbp)
MOVQ        -24(%rbp),  %rax        # i
PUSHQ       %rax
MOVQ        $.str0,     %rax
PUSHQ       %rax
POPQ        %rdi
POPQ        %rsi
MOVQ        $0,     %rax
CALLQ       printf
CLTQ
MOVQ        -24(%rbp),  %rax        # i
LEAVEQ
RETQ
.section   .data
.str0:
.string     "Valeur de i = %d\n"
.text
```

Sortie standard :

`Valeur de i = 1\n`, code de sortie : `1`

On peut bien sûr compiler des codes plus longs, mais l'assembleur généré ne tiendrait pas dans une page :)