Compilation de MiniC++

Léonard Blier, Jonathan Laurent

8 janvier 2014

On décrit dans dans ce document les caractéristiques techniques de notre compilateur et les choix d'implémentation qui ont été réalisés, ainsi que ce qui les a motivés. On analyse ensuite les difficultés que nous avons rencontrées pendant la réalisation de ce projet ainsi que nos résultats.

1 Présentation des fonctionnalités

1.1 Lancement du programme et options

Le programme se lance avec la commande

```
./minic++ INPUT_FILE [-o OUTPUT_FILE] [OPTIONS]
```

où `INPUT_FILE` doit comporter l'extension `.cpp`. Les options disponibles sont les suivantes :

`--parse-only` : effectue seulement l'analyse syntaxique et signale une erreur éventuelle.
`--type-only` : effectue seulement l'analyse sémantique et signale une erreur éventuelle.
`--ertl` : produit un fichier d'extension `.ertl` contenant le code ERTL intermédiaire
`--obj-descrs` : produit un fichier d'extension `.descrs` explicitant la représentation des objets en mémoire.
`--spill-all` : indique que tous les registres temporaires doivent être spillés, c'est à dire placés dans le cadre de pile.

1.2 Caractéristiques générales

Le compilateur a été conçu selon deux objectifs :

- Le code assembleur produit doit être raisonnablement efficace et lisible. Le compilateur dispose ainsi d'un module simple d'allocation de registres.
- La traduction de la couche objet se fait avec le mot d'ordre *pay as you go*, conformément à l'esprit du langage C++. Cela signifie que le support d'une certaine fonctionnalité ne doit pas créer des écarts de performance importants sur un code qui ne l'utilise pas. Ainsi, un programme n'utilisant pas l'héritage multiple n'est pas affecté par les surcharges que cette fonctionnalité induit : multiplication des tables de méthodes virtuelles et arithmétique implicite de pointeurs.

1.3 Caracéristiques techniques

1.3.1 Allocation des objets locaux sur la pile

Contrairement à ce qui est suggéré dans le sujet, les objets locaux sont alloués dans le cadre de pile et si un objet en contient un autre, il en est de même de leurs représentations en mémoire. Ainsi, aucune allocation dynamique n'est effectuée en dehors d'une utilisation de `new`, ce qui

pourrait permettre, en ajoutant un mot clé `delete` au langage, de compiler des exécutables sans fuites de mémoire.

Ce choix d'implémentation, loin d'alourdir le code, semble au contraire le rendre plus simple dans la mesure où il n'est alors plus nécessaire de générer des déréférencements implicites dans le code. En outre, nous ne voyons aucune implémentation élégante d'une représentation des objets comme pointeurs sur le tas dans la mesure où

- Soit nous décidons qu'un objet et un pointeur sur un objet ont la même représentation en mémoire, ce qui demande de gérer de nombreux cas particuliers lors de la génération de code.
- Soit nous décidons qu'un pointeur sur un objet est un pointeur sur un pointeur alloué dynamiquement, qui lui même pointe sur le tas. Mais cette solution, en introduisant artificiellement des doubles pointeurs, rend difficile une implémentation efficace de l'héritage multiple, comme nous le verrons par la suite.

1.3.2 Convention d'appel

La convention d'appel que nous avons retenue est différente de celle proposée dans le sujet. Tout d'abord, les quatre premiers arguments d'une fonction (ce qui comprend le pointeur `this` pour une méthode) sont passés dans les registres `$a0,...,$a3`. Ensuite, dans la mesure où aucune opération n'est effectuée sur `$sp` entre l'allocation et la suppression du cadre de pile, l'écart entre `$sp` et `$fp` reste constant, et il est possible d'économiser ce dernier registre. Enfin, c'est la fonction appelée qui se charge d'allouer la totalité du cadre de pile, et la fonction appelante doit donc écrire légèrement en dessous de `$sp`, ce qui pourrait réserver des surprises sur certains processeurs. Une explication détaillée de la convention pourra être trouvée dans le fichier *calling-conventions.md*, présent dans le répertoire `src`.

1.3.3 Gestion de l'héritage multiple

L'ajout de l'héritage multiple complique considérablement la représentation des objets en mémoire et la gestion des appels virtuels. Premièrement, si `C` hérite de `A` et de `B`, il est impossible de faire passer un objet de type `C` à la fois pour un objet de type `A` et un objet de type `B` à l'exécution.

Une solution à ce problème consiste à effectuer une addition implicite lors d'un cast de pointeurs. Le code `C c = C() ; B* = &c` devient `C c = C() ; B* = &c + <ofs>` où `<ofs>` est le décalage de la représentation de `B` dans une représentation de `C`. On peut vérifier grâce au code suivant, qui affiche `0` que GCC utilise une technique similaire.

```
class A {public : int a;};
class B {public : int b;};
class C : public A, public B {public : int c;};

int main() {
  C* cp = new C();
  B* bp = cp;
  std::cout << (((void*) cp) == ((void*) bp));
}
```

Il est alors nécessaire d'interdire les conversions du type `C**` vers `A**` sans quoi à l'issu du code suivant

```
C* cp = new C() ; A** app = &cp ; A* ap = *app ;
```

le pointeur `ap` serait mal positionné. Précisons d'ailleurs que GCC interdit également cette conversion.

1.3.4 Gestion des méthodes virtuelles

Nous commençons par préciser un point de sémantique qui nous a paru ambigu et qu'il a fallu éclaircir à l'aide du standard officiel de C++ :

> *Une fonction est implicitement déclarée virtuelle si une méthode de même nom et dont les arguments ont le même type a été déclarée virtuelle dans une classe parente.*

Dans une situation où il n'y a pas d'héritage multiple, la représentation de chaque objet en mémoire contient un unique pointeur sur une table de méthodes virtuelles. Les méthodes sont placées dans le même ordre que les attributs d'un objet dans sa représentation : d'abord les méthodes virtuelles introduites dans une classe, mère et ensuite les méthodes virtuelles nouvellement introduites.

De manière générale, le nombre de tables T_A associées à un objet de classe A respecte la relation :

$$T_A = \max\left(1, \sum_{i=1}^{p} T_{B_i}\right)$$

où $B_1, \dots B_p$ sont les parents de A. Avec cette relation, on retrouve que dans une configuration d'héritage simple, toute classe est associée à une unique table.

En fait, on construit le descripteur de la classe A de la manière suivante
- On concatène les descripteurs de $B_1, \dots B_p$.
- On met à jour le contenu de la table virtuelle associée au parent B_i à partir des méthodes virtuelles redéfinies en A mais introduites plus haut.
- On ajoute à la fin de la première de ces tables les entrées correspondant aux méthodes virtuelles déjà introduites.

On présente à gauche de la Figure 2 ce que nous donne l'option `obj-descrs` du compilateur avec le programme de la Figure 1 :

```
class A { public : int a; virtual void fa();};
class B { public : int b; virtual void fb(); };
class C : public A, public B
        { public : int c; virtual void fc();};
class D { public : int d; virtual void fd();};
class E : public C, public D
        { public : int e; virtual void fb(); virtual void fd();};

int main() {
  C* cp = new E();
  cp->fb();
  return 0;
}
```

FIGURE 1 – Programme en entrée

Enfin, dans le code présenté Figure 1, le pointeur `cp` pointe 8 octets plus bas que le début de la représentation de l'objet de classe `E` qui y est stocké. Après accès à la table `_vt__E__D`, on apprend que c'est la méthode redéfinie en `E` qu'il faut appeler. Cependant, il faudra passer à cette méthode un pointeur `this` vers un objet de type `E`, c'est à dire 8 octets plus haut que `cp`. La valeur de ce déplacement est en fait stockée dans la table virtuelle dont chaque entrée s'étend maintenant sur deux mots. On illustre cette affirmation à droite de la Figure 2.

```
OBJECT <E>
0. VTABLE <_vt__E__E>
4. a
8. VTABLE <_vt__E__B>
12. b
16. c
20. VTABLE <_vt__E__D>
24. d
28. e
Details for vtable at offset 0 :
_m0__A__fa
_m2__C__fc
Details for vtable at offset 8 :
_m4__E__fb
Details for vtable at offset 20 :
_m5__E__fd
```

```
_vt__E__E:
.word 0
.word _m0__A__fa
.word 0
.word _m2__C__fc
_vt__E__B:
.word -8
.word _m4__E__fb
_vt__E__D:
.word -20
.word _m5__E__fd
```

FIGURE 2 – Descripteur pour la classe `E` et extrait de l'assembleur généré

2 Architecture du processeur et détails d'implémentation

2.1 Structure générale

Le texte d'entrée est d'abord transformé, à la suite de l'analyse lexicale (`Lexer`) et syntaxique (`Parser`), en un arbre de syntaxe abstraite (`Ast`). Cet arbre est transmis au module `Typer` qui effectue le typage et l'analyse sémantique, afin d'aboutir à la représentation définie dans `Ttree`. En toute rigueur, toutes les erreurs doivent être signalées lors de cette passe, et les suivantes pourront faire l'hypothèse forte que le programme qui leur est passé en entrée est bien typé. L'arbre obtenu à l'issue du typage va ensuite être traité par le module `Is` dont les fonctions principales sont

- La traduction de la couche objet et en particulier la création des descripteurs d'objets, qui contiennent la position des attributs et des tables virtuelles ainsi que le contenu de ces dernières.
- La création d'un arbre de représentation de plus bas niveau, défini dans `Istree` dont les étiquettes sont proches des instructions MIPS.

C'est lors de cette passe que pourrait être entreprise une sélection d'instructions élaborée mais nous ne nous sommes pas focalisé sur cet aspect et la sélection d'instruction est très basique dans notre compilateur.

Dans la suite, toutes les procédures sont compilées séparemment. Certaines optimisations supplémentaires auraient pu être réalisées sans cette contrainte, comme on le verra par la suite. On va alors passer d'une structure d'arbre à une structure de graphe de flot de contrôle. Dans notre compilateur, on ne passe pas rigoureusement par les représentations RTL, ERTL, et LTL. Une unique structure, paramétrée par le type de registres utilisé (virtuel puis physique) est définie dans `Fgraph` et constitue un compromis entre ERTL et LTL. Notre compilateur n'optimisant pas l'appel terminal, il n'y a pas de traduction vers ce qui pourrait ressembler au langage RTL, et l'arbre issue de la sélection d'instruction est transformé en graphe étiqueté par des registres virtuels par le module `Ertl`.

La passe de traduction LTL a lieu après l'analyse de durée de vie et l'allocation de registres, effectuées respectivement par les modules `Liveness` et `Reg_alloc`. Après calcul d'une correspondance entre registres virtuels et registres physiques, le module `Ltl` est chargé de réécrire le graphe de flot de contrôle avec des registres physiques. Enfin, le graphe est linéarisé à l'aide du module `Linearize` et du code MIPS est produit. Le segment de données est généré à partir des descripteurs d'objets produits par le module `Is`.

2.2 Détails sur le fonctionnement de chaque passe

2.2.1 Analyse lexicale et syntaxique

Nous avons eu deux conflits à résoudre dans la grammaire proposée par le sujet :

- `if (...) if (...) else ...` :
  A quel `if` le `else` se rapporte-t-il ? Nous avons fait le choix, conforme à celui effectué par GCC, de l'associer au deuxième. Pour cela, il a suffi de donner à `else` une priorité supérieure à celle de la production `if (...) ...`
- Gestion du mot clé `virtual` :
  Si on lit une chaîne correspondant à un membre d'une classe, ne commençant pas par `virtual`, doit-on alors réduire ϵ en `option(virtual)` ? Cela revient à prendre la décision de reconnaître une méthode et non un argument, décision qui ne peut pas être prise à ce stade de la lecture.
  On décide donc que tous les membres d'une classe, attributs ou méthodes, peuvent commencer par `virtual`. Une erreur sera signalée ulterieurement dans le cas où le membre s'avérerait être un attribut.

Enfin, la grammaire proposée a été étendue car elle acceptait `A::m()` et non `this->A::m()`. Il est très commode pour la suite d'accepter la deuxième forme, dont la première n'est qu'une abbréviation.

2.2.2 Typage

Il aurait été possible de ne pas conserver les informations de typage des expressions après vérification de leur correction, dans la mesure où ces informations ne sont pas utiles aux passes suivantes. Cela est possible grâce à l'ajout au cours du typage d'opérateurs unaires (`Implicit_op`) explicitant une conversion de pointeurs, le déréférencement implicite qui a lieu lors de l'accès à une référence ...

A l'issue de cette étape, il ne reste aucune ambiguité sur l'instance du programme à laquelle un nom fait référence. Ainsi, les variables locales sont renommées de manière à ce qu'il n'y ait plus de problèmes de résolution de portée, et chaque mention d'un attribut, d'une méthode, ou d'une fonction est explicitée après résolution des problèmes de surcharge et de redéfinition.

Le problème de la résolution de noms est résolu de manière abstraite dans le module `Resolver`. Il peut se formaliser ainsi :

Soit Ω un ensemble partiellement ordonné, $E \subset \Omega$, et $x \in \Omega$. L'ensemble des majorants de x admet-il un plus petit élément ?

Naïvement, on pourrait résoudre ce problème, pour un x donné, en calculant l'ensemble $M_x = \{e \in E \mid x \leq e\}$ de ses majorants dans E et en regardant pour chaque élément de M_x s'il est plus petit que tous les autres, ce qui donne un algorithme en $O(|E|^2)$ comparaisons. On peut faire mieux en remarquant que, pour $x \in \Omega$, M_x admet un plus petit élément si et seulement si $\exists e \in M_x, |M_e| = |M_x|$. En précalculant le graphe de la relation d'ordre sur E, on peut ainsi résoudre le problème en $O(|E|)$.

2.2.3 Traduction de la couche objet et sélection d'instruction

Le module `Is` aurait pu être scindé en deux modules dans la mesure où la génération des descripteurs d'objets est une opération indépendante. En outre, étant donné que nous n'effectuons pas de sélection d'instructions subtile, il aurait été possible de traduire directement en ERTL l'arbre de typage. Cependant, cette étape supplémentaire est assez confortable dans la mesure où elle permet d'effectuer des distinctions (un type différent pour les valeurs gauches et les valeurs droites) qui simplifient la passe suivante.

2.2.4 Graphe de flot de contrôle

Le graphe de flot de contrôle est généré en tirant parti de toutes les astuces de programmation présentées en cours (fonctions `generate`, `loop`...) et la passe de linéarisation le reprend presque exactement. Citons tout de même deux écarts parmi d'autres :

Toutes les variables ne peuvent pas être placées dans un registre. Par exemple, dans le code :

```
{int i = 3 ; f(&i);}
```

`i` doit être placé sur la pile sans quoi il serait impossible d'obtenir son adresse. L'ensemble des variables qui doivent être placées impérativement sur la pile est déterminé par le module `Is`.

La fonction `expr` ne prend pas pour premier argument un registre `ret` destiné à recueillir son résultat mais un argument de type `option register` qui, s'il contient `None`, indique que le résultat de l'expression doit être jeté. C'est le couple de fonctions utilitaires `retreg` et `genret` qui permet de gérer systématiquement les deux cas sans ajouter trop de lignes de code. Notons qu'une autre solution – mais qui s'est avérée décevante en pratique – aurait été d'ajouter un registre fictif `dummy_reg` et de supprimer par la suite toutes les instructions qui le font intervenir.

2.2.5 Allocation de registres

L'algorithme d'analyse de durée de vie est un simple calcul de point fixe en $O(n^4)$, réalisé à l'aide d'une fonction indépendante `fixpoint : ('a -> 'a * bool) -> a -> a`. De manière générale, l'implémentation des algorithmes de ce dernier paragraphe ne vise pas l'efficacité, mais plutôt un code le plus simple possible, de manière à limiter les erreurs.

L'algorithme de coloration de graphe est une synthèse personnelle du cours, du TD 10 et du document trouvé à l'adresse *http ://www.pps.univ-paris-diderot.fr/~balat/compilation/compil08-coloriagedegraphe.pdf*, décrite dans la section suivante. Enfin, la passe de traduction vers LTL se fait en utilisant les deux registres réservés `$v1` et `$fp`.

2.3 L'algorithme de coloration de graphes

Le graphe d'interférence a pour noeuds à la fois des registres physiques et des registres temporaires. L'algorithme est le suivant, où `ncols` est le nombre de couleurs disponibles :

1. On cherche un noeud vérifiant trois propriétés :
   - Il représente un registre temporaire
   - Il est relié par des arêtes d'interférence à moins de `ncols` noeuds
   - Il ne partage pas d'arête de préférence avec un registre temporaire

   Si ces conditions sont remplies, on colore le graphe privé de ce noeud et on est certains qu'il restera au moins une couleur disponible. On choisit de préférence une couleur correspondant à un registre physique avec lequel il partage une arête de préférence, et plutôt un registre *caller-saved* que *callee-saved*.

   Si aucun noeud ne respecte ces trois conditions, on passe en 2.
2. On fusionne deux registres temporaires reliés par une arête de préférence sans risquer de perdre la coloriabilité du graphe. Deux conditions suffisantes sont les suivantes :
   - Le noeud issu de la fusion est relié par des arêtes d'interférence à moins de `ncols` noeuds
   - Tout noeud qui interfère avec le premier interfère avec le second

   Si on trouve deux candidats à une telle fusion, on retourne en 1. Sinon, on passe en 3.
3. On supprime une arête de préférence d'un noeud de faible degré qui sera simplifié en 1. Si on ne trouve pas de candidat, en dernier recours, on passe en 4.

4. On retire le noeud de degré maximal, et on tente de colorier le reste du graphe. S'il reste tout de même des couleurs disponibles, on le colorie. Sinon, on le spill définitivement. On retourne ensuite en 1.

L'algorithme se poursuit tant qu'il reste un registre temporaire dans le graphe, les registres physiques étant coloriés de manière évidente.

Remarque : l'implémentation de cet algorithme dans `minic++` comporte moins de 300 lignes.

3 Nos résultats

3.1 Analyse du code produit

Globalement, le résultat de l'allocation de registres est plutôt satisfaisant. En fait, sur la presque totalité des tests, aucun registre n'est spillé à part les registres *callee-saved* utilisés dans une fonction qui en appelle une autre, et il y a relativement peu d'instructions `move` superflues. Les registres temporaires sont rarement utilisés au delà de `t2`, ce qui est assez frustrant. Le problème vient de ce que, sans informations sur les registres *caller-saved* effectivement réécrits par une procédure appelée et en supposant qu'ils le sont tous, ils ne peuvent pas être utilisés dans la plupart des cas, et en particulier pour la sauvegarde des registres *callee-saved*. Tout ceci laisse à penser qu'il serait extrêmement intéressant d'effectuer une analyse interprocédurale pour utiliser au mieux le nombre élevé de registres proposé par MIPS.

Pour exemple, voici le code assembleur géneré par `minic++` sur le test `fact_loop`, sans et avec l'option `--spill-all` (respectivement Figure 3 et Figure 4). On y observe d'ailleurs que la sélection d'instruction a été soigneusement baclée mais que l'allocation de registres a bien fonctionnée : il n'y a pas de sauvegarde des registres *callee-saved* et `$v0` est utilisé simultanément comme registre de travail et registre de retour.

```
_p1__fact_loop:
        addi $sp, $sp, -4
        li   $v0, 1
L56:
        li   $t0, 1
        sgt  $t0, $a0, $t0
        bnez $t0, L52
        addi $sp, $sp, 4
        jr   $ra
L52:
        move $t0, $a0
        addi $a0, $t0, -1
        mul  $v0, $v0, $t0
        b    L56
```

FIGURE 3 – Version optimisée

3.2 Validation des tests

Tous les tests d'analyse syntaxique et de typage sont réussis, et tous les tests d'exécution le sont à l'exception de deux d'entre eux qui ont été volontairement laissés irrésolus sur le simulateur Mars :

```
_p1__fact_loop:
        addi $sp, $sp, -80
        sw   $s0, 72($sp)
        sw   $s1, 68($sp)
        sw   $s2, 64($sp)
        sw   $s3, 60($sp)
        sw   $s4, 56($sp)
        sw   $s5, 52($sp)
        sw   $s6, 48($sp)
        sw   $s7, 44($sp)
        sw   $a0, 40($sp)
        li   $v1, 1
        sw   $v1, 0($sp)
        lw   $v1, 0($sp)
        sw   $v1, 36($sp)
L57:
        lw   $v1, 40($sp)
        sw   $v1, 8($sp)
        li   $v1, 1
        sw   $v1, 4($sp)
        lw   $v1, 8($sp)
        lw   $fp, 4($sp)
        sgt  $v1, $v1, $fp
        sw   $v1, 12($sp)
        lw   $v1, 12($sp)
        bnez $v1, L53
        lw   $v0, 36($sp)
        lw   $s0, 72($sp)
        lw   $s1, 68($sp)
        lw   $s2, 64($sp)
        lw   $s3, 60($sp)
        lw   $s4, 56($sp)
        lw   $s5, 52($sp)
        lw   $s6, 48($sp)
        lw   $s7, 44($sp)
        addi $sp, $sp, 80
        jr   $ra
L53:
        lw   $v1, 36($sp)
        sw   $v1, 28($sp)
        lw   $v1, 40($sp)
        sw   $v1, 20($sp)
        lw   $v1, 20($sp)
        addi $v1, $v1, -1
        sw   $v1, 16($sp)
        lw   $v1, 16($sp)
        sw   $v1, 40($sp)
        lw   $v1, 20($sp)
        sw   $v1, 24($sp)
        lw   $v1, 28($sp)
        lw   $fp, 24($sp)
        mul  $v1, $v1, $fp
        sw   $v1, 32($sp)
        lw   $v1, 32($sp)
        sw   $v1, 36($sp)
        b    L57
```

FIGURE 4 – Version non optimisée, avec l'option `--spill-all`

– `octal.cpp` : les constantes octales sont réécrites dans le même format, avec un préfixe `0`, dans le fichier assembleur produit. D'après les spécifications de l'assembleur MIPS, ce format devrait être reconnu. Le problème vient donc sûrement du simulateur Mars.
– `string.cpp` : les chaines de caractère sont exactement répliquées telles quelles, et le problème vient du non-traitement par Mars de la séquence d'échappement `\xhh` où `hh` est le codage hexadecimal d'un caractère ascii. Nous n'avons pas trouvé de séquence équivalente reconnue par Mars. Il serait maladroit de traduire `\x41` en le caractère `A` dans la mesure où l'utilisateur utilise un code hexadécimal justement pour ne pas avoir à saisir le caractère, qui aurait pu ne pas avoir de représentation graphique.
Il reste la possibilité de ne pas utiliser `.asciiz` et de représenter les chaînes par une séquence explicite d'octets dans le fichier assembleur, mais nous n'avons pas implémenté cette solution qui produirait du code moins clair.

Nous avons également ajouté de nouveaux tests car certains cas de figure importants ne sont pas traités par les tests fournis, dont l'héritage multiple. Voici dans le désordre quelques situations qui ont été l'objet de tests supplémentaires :
– Cas où une fonction attend plus de 5 **arguments**.
– Situations d'héritage multiple faisant intervenir une **arithmétique implicite de pointeurs**.
– Cas où `A` hérite de `B` et de `C`. `B` et `C` définissent indépendamment une méthode virtuelle `f` de même signature, et `C` la surcharge.
– Cas simples de schéma d : il y a copie de certains champs et il faut s'assurer que c'est toujours au même représentant qu'on accède.
– Cas où les arguments d'une méthode n'ont pas le même nom lors de sa déclaration et de son implémentation.

3.3 Faiblesses de notre compilateur

Une faiblesse de notre compilateur réside dans la qualité des messages d'erreur produits. La ligne est la plupart du temps correctement donnée mais la fourchette de caractères associée est souvent trop large.

Pourtant, tout a été mis en place pour pouvoir générer des messages d'erreur clairs et précis :
– L'écriture des messages d'erreurs est géré par un module indépendant
– L'arbre de syntaxe abstraite est correctement décoré par des informations de localisation
– Le module `Resolver`, quand il détecte un conflit, renvoie une exception contenant la liste de tous les candidats entre lesquels il ne parvient pas à trancher

Nous avons cependant préféré investir notre temps et notre motivation pour améliorer d'autres fonctionnalités.

Enfin, il reste au moins un problème potentiel non résolu : il est toujours possible de contourner à l'aide de références l'interdiction par le typage d'effectuer des conversions de doubles pointeurs. Ce problème ne devrait pas être long à régler, mais nous n'avons pas le temps pour le faire avant la correction du projet.