École Polytechnique

INF564 : Compilation

examen 2019 (X2016)

Jean-Christophe Filliâtre

18 mars 2019

Les notes de cours manuscrites ou reprographiées sont les seuls documents autorisés.
L'épreuve dure 3 heures.

Dans tout ce sujet, on considère un tout petit fragment du langage C dont la syntaxe abstraite est donnée figure 1. Chaque fonction a exactement deux arguments, notés `x` et `y`. On note que ce fragment de C est un sous-ensemble strict de celui du projet. En particulier, les fonctions peuvent être récursives, mais pas mutuellement récursives : le corps d'une fonction f ne peut appeler que des fonctions définies préalablement ou la fonction f elle-même. On rappelle que la construction `if` du langage C teste si son argument est non nul. Voici un programme dans ce fragment :

```
int mul(int x, int y) {
  if (x) return y - (0 - mul(x - 1, y)); else return 0; }
int fact(int x, int y) {
  if (x) return fact(x - 1, mul(x, y)); else return y; }
```

Avec ce programme, un appel à `fact(`n`, `m`)`, pour deux entiers n et m, va calculer $n! \times m$ (en supposant une absence de débordement arithmétique). On peut supposer qu'une fonction `main` est ajoutée systématiquement à un tel programme, mais cet aspect-là ne nous intéresse pas ici.

1 Sémantique

Question 1 Indiquer ce que calcule un appel à `myst(`n`, `m`)`, pour deux entiers n et m, avec $n \geq 0$ et la définition suivante :

```
int myst(int x, int y) {
  if (x)
    if (x - 1) return myst(x - 2, myst(x - 1, y)); else return 1 - (0 - y);
  else
    return y;
}
```

Question 2 Donner le code d'une fonction `lt` qui reçoit en arguments deux entiers n et m, en supposant $n \geq 0$ et $m \geq 0$, et renvoie un entier non nul si et seulement si $n < m$.

e	::=	**expression**
	| n	constante entière
	| `x`	la variable `x`
	| `y`	la variable `y`
	| e `-` e	soustraction
	| f`(`e`,` e`)`	appel de fonction
s	::=	**instruction**
	| `return` e`;`	retour de fonction
	| `if (`e`)` s `else` s	conditionnelle
d	::=	**définition de fonction**
	| `int` f`(int x, int y) {` s `}`	
p	::=	**programme**
	| $d \ldots d$	

FIGURE 1 – Syntaxe abstraite.

Sémantique opérationnelle à grands pas. On souhaite munir notre langage d'une sémantique opérationnelle à grands pas sous la forme de deux jugements

$$E, e \twoheadrightarrow n \qquad \text{et} \qquad E, s \twoheadrightarrow n$$

où E est une fonction donnant la valeur associée aux variables `x` et `y`. Le jugement $E, e \twoheadrightarrow n$ signifie « dans l'environnement E, l'évaluation de l'expression e termine, avec la valeur n ». Le jugement $E, s \twoheadrightarrow n$ signifie « dans l'environnement E, l'évaluation de l'instruction s termine, en aboutissant à une instruction `return` qui renvoie la valeur n ».

On suppose que les fonctions du programme sont données sous la forme d'un ensemble F de paires (f, s) où f est le nom d'une fonction et s l'instruction qui constitue le corps de cette fonction.

Question 3 Donner les règles d'inférence définissant les relations $E, e \twoheadrightarrow n$ et $E, s \twoheadrightarrow n$.

Question 4 Donner un ensemble de fonctions F, un environnement E et une expression e pour lesquels il n'existe pas de valeur n telle que $E, e \twoheadrightarrow n$.

2 Compilation vers x86-64

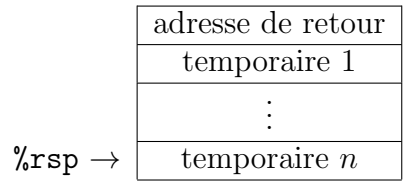
On se propose maintenant de compiler notre langage vers l'assembleur x86-64, *en utilisant exclusivement* les registres `%rdi`, `%rsi` et `%rax` pour les calculs et le registre `%rsp` pour la pile, et *exclusivement* les instructions qui sont données en annexe de ce sujet. On adopte le schéma de compilation suivant :

— Toutes les valeurs sont des entiers signés 64 bits.
— La valeur de `x` est passée dans `%rdi`, celle de `y` dans `%rsi` et la valeur renvoyée par la fonction dans `%rax`.
— Tous les registres sont *caller-saved*.

`mov` n $r \to L$
`mov` r $r \to L$
`sub` n $r \to L$
`sub` r $r \to L$
`jz` $r \to L, L$
`call` $f \to L$
`alloc_frame` $\to L$
`delete_frame` $\to L$
`return`

FIGURE 2 – Instructions ERTL.

— Le tableau d'activation prend la forme suivante :



L'adresse de retour est celle déposée par l'instruction `call`. En dessous, on trouve la place allouée à n entiers 64 bits, pour toutes les valeurs temporaires qui ne peuvent pas être stockées dans les trois registres.

Question 5 Justifier qu'il n'est pas utile d'utiliser aussi le registre `%rbp` pour accéder au tableau d'activation (contrairement à ce qui a été vu en cours et fait en projet).

Question 6 Donner un code x86-64 possible pour la fonction `myst` de la question 1, en optimisant l'appel terminal.

Compilateur optimisant. Pour écrire un compilateur optimisant pour notre langage, on suit l'architecture présentée en cours et réalisée en projet. On suppose que la sélection d'instructions et la traduction vers RTL ont déjà été réalisées (elles sont ici très simples) et on s'intéresse à l'étape ERTL. La figure 2 contient les instructions du langage ERTL que l'on considère ici, où n désigne une constante entière, r un pseudo-registre ou un registre physique et L une étiquette dans le graphe de flot de contrôle ERTL.

Question 7 Donner un code ERTL pour la fonction suivante.

```
int mul(int x, int y) {
  if (x) return y - (0 - mul(x - 1, y)); else return 0; }
```

On pourra commencer par construire un code RTL, pour le transformer ensuite en un code ERTL. On pourra omettre les étiquettes quand les instructions sont purement séquentielles.

Question 8 Pour le code ERTL suivant, donner le résultat de l'analyse de durée de vie (variables vivantes en chaque point de programme), le graphe d'interférence, un coloriage possible de ce graphe

et le code LTL donné par ce coloriage.

$$
\begin{array}{rl}
L_1: & \texttt{alloc\_frame} \to L_2 \\
L_2: & \texttt{mov \%rdi \#1} \to L_3 \\
L_3: & \texttt{mov \%rsi \#2} \to L_4 \\
L_4: & \texttt{mov \#1 \#4} \to L_5 \\
L_5: & \texttt{mov \#2 \#5} \to L_6 \\
L_6: & \texttt{sub \#5 \#4} \to L_7 \\
L_7: & \texttt{mov \#1 \#3} \to L_8 \\
L_8: & \texttt{sub \#4 \#3} \to L_9 \\
L_9: & \texttt{mov \#3 \%rax} \to L_{10} \\
L_{10}: & \texttt{delete\_frame} \to L_{11} \\
L_{11}: & \texttt{return}
\end{array}
$$

Le point d'entrée du graphe est L_1.

Analyse de durée de vie plus précise. Pour calculer les variables vivantes en tout point du graphe de flot de contrôle ERTL, on se sert notamment des *définitions* (*def*) et des *utilisations* (*use*) de chaque instruction. Dans le cas où l'instruction à l'étiquette L est un appel (`call` f), on pose a priori

$$
\begin{array}{rcl}
\mathit{def}(L) & = & \{\,\texttt{\%rdi},\ \texttt{\%rsi},\ \texttt{\%rax}\,\} \\
\mathit{use}(L) & = & \{\,\texttt{\%rdi},\ \texttt{\%rsi}\,\}
\end{array}
$$

(On rappelle que les trois registres sont *caller-saved*.) C'est toujours correct mais c'est parfois une sur-approximation des registres que la fonction utilise ou définit effectivement.

Question 9 Donner un exemple de fonction f pour laquelle les ensembles de registres effectivement utilisés et définis par un appel à f sont tous les deux strictement plus petits que les ensembles ci-dessus.

Question 10 Expliquer comment on peut améliorer le calcul de *use* pour un appel à une fonction dans le cas général.

Question 11 Expliquer comment on peut améliorer le calcul de *def* pour un appel à une fonction f dans le cas général, pour une fonction f *non récursive*. (Indication : il faut aller jusqu'au bout de l'allocation de registres.)

Question 12 Même question avec maintenant une fonction f récursive.

Question 13 Donner un exemple de fonction récursive, ainsi que son code ERTL, pour lequel l'ensemble *def* est strictement plus petit que $\{\,$`%rdi`, `%rsi`, `%rax`$\,\}$ pour une allocation de registres que l'on précisera.

Question 14 Expliquer pourquoi il convient de calculer les *def*/*use* de chaque fonction dans l'ordre où elles apparaissent dans le programme.

3 Sous-expressions communes

Pour compiler encore plus efficacement nos programmes, on propose l'idée suivante : si le corps d'une fonction fait apparaître au moins deux fois la même expression e, on cherchera à ne l'évaluer qu'une seule fois, *quand cela est possible*. On appelle cela le partage des sous-expressions communes.

Question 15 Identifier dans le programme suivant les sous-expressions dont le calcul pourrait être partagé :

```
int f(int x, int y) { return x - (0 - y); }
int g(int x, int y) { return g(f(x - 1, x - 1), f(x - 1, x - 1)); }
int h(int x, int y) {
  if (x - 1) return g(x - 1, y);
             else if (y - 1) return 2; else return g(x - 1, y - 1); }
```

Question 16 Donner un critère permettant d'assurer que l'évaluation d'une expression ou d'une instruction termine. Le problème étant indécidable, on proposera une approximation correcte (mais nécessairement incomplète).

Partage des sous-expressions communes. Pour mettre en œuvre le partage des sous-expressions communes, on étend la syntaxe abstraite de notre langage avec de nouvelles variables (notées $\texttt{\#}_i$ comme les pseudo-registres du code RTL).

$$
\begin{array}{rcll}
e & ::= & \ldots & \\
 & | & \texttt{\#}_i & \text{utilisation d'une variable de partage} \\
 & & & \\
s & ::= & \ldots & \\
 & | & \texttt{\#}_i \leftarrow e\texttt{;}\ s & \text{introduction d'une variable de partage}
\end{array}
$$

Ainsi, on peut écrire maintenant des programmes comme

```
int p(int x, int y) { #1 <- x - 1; if (#1) return p(#1, y); else return #1 - 1; }
```

pour factoriser ici le calcul de l'expression `x - 1`.

Question 17 Réécrire le programme de la question 15 en partageant le calcul des sous-expressions communes.

Question 18 Proposer un algorithme pour effectuer le partage des sous-expressions communes, c'est-à-dire un algorithme pour transformer un programme dans la syntaxe abstraite originale (de la figure 1) vers un programme de la syntaxe abstraite étendue avec les variables de partage. La sémantique du programme doit être conservée. La complexité du programme ne doit pas être dégradée.

Indications : Écrire cette transformation à l'aide de dictionnaires associant à des expressions des variables de partage. La transformation prend en arguments un tel dictionnaire (les expressions à partager déjà connues) et un élément de programme à transformer (expression ou instruction). La transformation renvoie à la fois un nouveau dictionnaire et l'élément transformé.

On pourra supposer qu'on a répondu à la question 16 et qu'on dispose donc d'un critère pour assurer que l'évaluation d'une expression termine.

Annexe : sous-ensemble de x86-64 considéré

Dans ce sujet, on se limite au fragment du jeu d'instructions x86-64 décrit ci-dessous et aux quatre registres `%rdi`, `%rsi`, `%rax` et `%rsp`. Dans ce qui suit, L désigne une étiquette, r désigne un registre, n une constante entière et o une opérande qui est soit un registre r, soit une opérande indirecte n`(`r`)`.

`mov` r`,` o	copie le registre r dans l'opérande o
`mov` o`,` r	copie l'opérande o dans le registre r
`mov` `$`n`,` r	charge la constante n dans le registre r
`sub` o`,` r	calcule $r - o$ et l'affecte à r
`sub` r`,` o	calcule $o - r$ et l'affecte à o
`sub` `$`n`,` o	calcule $o - n$ et l'affecte à o
`push` r	empile la valeur de r
`pop` r	dépile une valeur dans le registre r
`test` r_2`,` r_1	positionne les drapeaux en fonction de la valeur de r_1 ET r_2
`jz` L	saute à l'adresse désignée par l'étiquette L si les drapeaux signalent un résultat nul
`jmp` L	saute à l'adresse désignée par l'étiquette L
`call` L	saute à l'adresse désignée par l'étiquette L, après avoir empilé l'adresse de retour
`ret`	dépile une adresse et y effectue un saut