École Polytechnique

INF564 : Compilation

examen 2023 (X2020)

Jean-Christophe Filliâtre

13 mars 2023

Les notes de cours manuscrites ou reprographiées sont les seuls documents autorisés.
L'épreuve dure 3 heures.
Les trois parties **ne sont pas** indépendantes.
Suggestion : détacher la feuille contenant les figures 1–3.

1 Un mini langage RTL

On considère un langage RTL simplifié à l'extrême pour les besoins de ce sujet. La figure 1 décrit les cinq instructions de ce langage RTL, où $n \in \mathbb{Z}$ désigne une constante entière, v désigne un pseudo-registre (qu'on appellera une *variable* par la suite), L désigne une étiquette et f désigne un nom de fonction. L'instruction `mov` n v charge la constante n dans la variable v ; l'instruction `mov` u v copie la variable u dans la variable v ; l'instruction `sub` u v soustrait la valeur de la variable u à la variable v (attention au sens) ; l'instruction `jnz` $\to L_1, L_2$ saute à l'étiquette L_1 si le résultat de la dernière soustraction est non nul et à l'étiquette L_2 sinon ; enfin, l'instruction $u \leftarrow$ `call` $f(v, w)$ affecte à la variable u le résultat de l'appel $f(v, w)$.

Une fonction est définie par son graphe de flot de contrôle. Les N instructions du graphe sont étiquetées avec des entiers, de 0 à $N - 1$. Le point d'entrée est l'étiquette 0 et le point de sortie est l'étiquette N. On suppose que chaque fonction a exactement deux paramètres, reçus dans les variables x et y, et qu'elle renvoie un résultat dans la variable z. La figure 2 contient un exemple de graphe RTL avec $N = 8$ instructions.

Question 1 Que calcule la fonction de la figure 2 ? On pourra faire l'hypothèse que $x \geq 0$ à l'entrée de cette fonction.

Correction : Cette fonction calcule le produit $x \times y$ dans la variable z. Par ailleurs, au final
— la variable x contient 0 ;
— la variable y est inchangée ;
— la variable a contient $-y$ si initialement $x > 0$, et est inchangée sinon ;
— la variable t contient 1 si initialement $x > 0$ et 0 sinon.

Question 2 Donner un code RTL pour la fonction C suivante.

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

Correction :

```
z myst(x, y)
   0: mov y z -> 1
   1: mov 0 a -> 2
   2: sub a x -> 3
   3: jnz -> 4, 12
   4: mov 1 b -> 5
   5: sub b x -> 6
   6: jnz -> 9, 7
   7: mov -1 c -> 8
   8: sub c z -> 12
   9: y <- call myst(x, y) -> 10
  10: sub b x -> 11
  11: z <- call myst(x, y) -> 12
```

Question 3 Quelle forme prend un appel terminal dans ce langage RTL ? Donner un exemple.

Correction : Un appel terminal est une instruction de la forme

```
L: z <- call f(u, v) -> N
```

où `u` et `v` sont des variables quelconques.
On en a un exemple avec la dernière ligne de la réponse à la question précédente.

Variables vivantes. Afin de réaliser une allocation de registres (dans la partie suivante), on a besoin de déterminer les variables vivantes en entrée de chaque instruction. On les calcule comme expliqué en cours, à partir des définitions et des utilisations de chaque instruction puis d'un calcul de point fixe sur le graphe de flot de contrôle.

Question 4 Pour chaque instruction RTL de la figure 1, indiquer quelles sont les variables que cette instruction définit (*def*) et quelles sont les variables que cette instruction utilise (*use*).

Correction :

	def	use
movi n v	v	
mov u v	v	u
sub u v	v	u, v
jnz		
u<-call f(v,w)	u	v, w

Question 5 En supposant que la variable z est vivante à la sortie du graphe de la figure 2, donner les variables vivantes en *entrée* de chacune des huit instructions.

Correction :

0	x, y
1	x, y, z
2	a, x, y, z
3	a, x, z
4	a, x, z
5	a, x, z
6	a, t, x, z
7	a, x, z

---

2 Une autre allocation

Dans cette partie, on étudie un algorithme d'allocation de registres alternatif, différent de celui étudié en cours et en projet (par coloration du graphe d'interférence). Ce nouvel algorithme a notamment le mérite d'être moins coûteux, ce qui permet de l'utiliser par exemple pendant de la compilation à la volée. On réalise cette allocation de registres dans le contexte du langage RTL introduit dans la partie précédente (sans passer par un langage ERTL intermédiaire). L'allocation de registres est réalisée indépendamment pour chaque fonction du programme.

Intervalles. Pour chaque variable v du graphe de flot de contrôle, on définit son *intervalle* comme le plus petit intervalle d'entiers $[i, j]$, avec $0 \leq i \leq j \leq N$, tel que la variable v n'est pas vivante à l'entrée de l'instruction k pour tout k en dehors de cet intervalle. En particulier, la variable v est donc vivante en entrée de l'instruction i et de l'instruction j. Mais elle n'est pas nécessairement vivante en entrée de toute instruction k pour $i < k < j$. On fait l'hypothèse que toute variable apparaissant dans le graphe de flot de contrôle est vivante à l'entrée d'au moins une instruction. On note qu'il est tout à fait possible que $i = j$, c'est-à-dire qu'un intervalle soit réduit à une seule instruction.

Ainsi, pour le graphe donné à gauche, on obtient les intervalles donnés à droite :

graphe	variables vivantes en entrée	intervalles
0 : `mov` 0 $t \to 1$	0 : $\{x, y\}$	
1 : `sub` t $x \to 2$	1 : $\{t, x, y\}$	x : $[0, 3]$
2 : `jnz` $\to 3, 4$	2 : $\{x, y\}$	y : $[0, 4]$
3 : `mov` x $z \to 5$	3 : $\{x\}$	z : $[5, 5]$
4 : `mov` y $z \to 5$	4 : $\{y\}$	t : $[1, 1]$
	5 : $\{z\}$	

Les variables vivantes en entrée de chaque instruction sont données au centre. Pour exprimer que la variable z est vivante à la sortie du graphe, on la considère vivante à l'entrée de l'instruction N (ici $N = 5$ sur cet exemple), même s'il n'y a pas vraiment d'instruction N.

Algorithme d'allocation des registres. On introduit maintenant un algorithme d'allocation des registres utilisant ces intervalles. On suppose qu'il y a K registres physiques, notés $R_1, \ldots, R_K$.

L'algorithme va affecter à chaque variable v une localisation, notée $loc[v]$, qui est soit un registre physique R_i, soit la valeur `Spill` pour indiquer que cette variable est vidée en mémoire.

L'algorithme est donné figure 3. Il est composé de trois fonctions : **allocation**, **expiration** et **vidage**. Le point d'entrée est la fonction **allocation**. On prendra le temps de bien lire et de bien comprendre cet algorithme. On note que cet algorithme ne dépend *que* de la valeur de K et des intervalles associés à chaque variable.

Question 6 Donner le résultat de cet algorithme pour $K = 3$ et les intervalles suivants :

$$x : [0, 7] \quad y : [0, 2] \quad z : [1, 8] \quad a : [2, 7] \quad t : [6, 6]$$

En cas d'égalité sur une valeur de i ou de j, quand l'algorithme indique de procéder selon un certain ordre ou de sélectionner une valeur maximale, on pourra choisir arbitrairement.

Correction : On obtient successivement :

```
x: R1
y: R2
z: R3
a: R3 et z: Spill
t: R2
```

Question 7 Dans la fonction **vidage**, on pourrait se contenter de la partie **sinon**, c'est-à-dire de toujours effectuer $loc[v] \leftarrow$ `Spill`. Expliquer l'intérêt de la partie **alors** de la fonction **vidage**.

Correction : L'idée est de libérer ainsi un registre actuellement occupé par un intervalle susceptible d'interférer avec un maximum d'intervalles à venir, d'où le choix de j' maximal. Dans l'exemple ci-dessus, on a libéré ainsi R_3 pour la variable a. Sans cette partie-là du code, on aurait vidé à la fois a et t.

Question 8 Cet algorithme d'allocation des registres permet-il d'allouer un même registre physique à deux variables u et v pour lesquelles il existe une instruction `mov` u v, à l'instar de ce que permettent (parfois) les arêtes de préférences dans l'allocation par coloration de graphe ?

Correction : Si on a une instruction i : `mov` u v, il est possible d'avoir un intervalle $[?, i]$ pour u et un intervalle $[i + 1, ?]$ pour v, c'est-à-dire deux intervalles disjoints. En effet, la variable u est vivante en entrée de l'instruction `mov` u v mais pas la variable v. Sous cette condition, l'algorithme peut alors affecter un même registre aux variables u et v.

Question 9 Montrer que cet algorithme est *correct*, au sens où il n'affecte jamais le même registre physique à deux variables distinctes vivantes en entrée d'une même instruction. On s'attachera notamment à énoncer clairement les *invariants* de cet algorithme.

Correction : On remarque que si deux variables sont vivantes en entrée d'une même instruction, leurs intervalles s'intersectent. (La réciproque est fausse.) On montre la propriété suivante, notée I : deux variables dont les intervalles s'intersectent ne se voient pas allouer le même registre, et le résultat s'en déduit donc. Plus généralement, les invariants de la boucle de **allocation** sont :

1. tous les intervalles déjà examinés respectent I, ce qui inclut tous les intervalles actuellement dans *actifs* ;
2. tous les intervalles $[i', j']$ avec $j' < i$ ne sont pas dans actifs et leurs variables ont été allouées ;
3. les variables des intervalles dans *actifs* sont allouées dans des registres physiques, distincts ;
4. l'ensemble *libres* et l'ensemble des registres alloués aux intervalles de *actifs* sont disjoints et leur union est égale à $\{R_1, R_2, \ldots, R_K\}$.

Ces invariants sont trivialement vérifiés initialement.

La fonction **expiration** les maintient, car d'une part un intervalle qui sort de *actifs* voit son registre (nécessairement physique) remis dans *libres* et d'autre part **expiration** procède par j' croissant ce qui permet de maintenir 2.

Il y a ensuite deux cas de figure :

— si on appelle **vidage** : peu importe le choix de v', puis
  — si $j' > j$, les registres utilisés par *actifs* ne changent pas et la propriété I reste vérifiée ;
  — sinon, rien à vérifier car $loc[v]$ reçoit `Spill` ;

— si on n'appelle pas **vidage** : le registre retiré de *libres* est ajouté à *actifs*, ce qui maintient 4, et l'invariant 1 est maintenu

---

Question 10 Le résultat de la question précédente ne suffit pas en pratique, car si le code appelle une fonction avec `call` alors les registres physiques utilisés par cette fonction vont être écrasés. Proposer une solution pour y remédier.

Correction : Il suffit de vider en mémoire systématiquement toute variable v vivante en sortie d'une instruction $u \leftarrow$ `call`(...) et différente de u. Puis on fait tourner l'algorithme inchangé sur les autres variables.

Question 11 Quelle est la complexité de cet algorithme, en fonction du nombre V de variables et de la valeur K ?

Correction : Chaque intervalle est ajouté une fois à *actifs* et supprimé (au plus) une fois de *actifs*, soit $O(V)$ ajouts et suppressions au total. Les ensembles *actifs* et *libres* contiennent au plus K éléments. La complexité des opérations sur ces ensembles dépend de la structure de données choisie. Avec un arbre équilibré, on aura des opérations en $O(\log K)$ et donc un coût total $O(V \log K)$. Avec des listes triées, on aura des opérations en $O(K)$ et donc un coût total $O(V \times K)$.

Numérotation des instructions. Comme on l'a compris, le résultat de l'algorithme dépend fortement de la numérotation des instructions. Or il existe $(N-1)!$ façons différentes de numéroter les instructions, car seul le numéro de l'instruction 0 est imposé.

Question 12 Donner un exemple de graphe de flot de contrôle et deux numérotations possibles pour les instructions de ce graphe, l'une permettant une allocation avec seulement deux registres physiques et l'autre nécessitant au moins trois registres physiques (sans rien vider en mémoire à chaque fois).

Correction : Considérons le graphe suivant, avec ces deux numérotations :

```
0: mov 2 x -> 3        0: mov 2 x -> 1
1: mov 1 t -> 2        1: sub x z -> 2
2: sub t z -> 4        2: mov 1 t -> 3
3: sub x z -> 1        3: sub t z -> 4
4: sub t z -> 5        4: sub t z -> 5
```

À gauche, on a les intervalles $z : [0, 5]$, $t : [2, 4]$ et $x : [3, 3]$, ce qui nécessite trois registres minimum. À droite, en revanche, on a les intervalles $z : [0, 5]$, $t : [3, 4]$ et $x : [1, 1]$, ce qui ne demande que deux registres.

Numérotation en profondeur. On se propose de re-numéroter les instructions à l'aide d'un parcours en profondeur du graphe de flot de contrôle. À chaque instruction i, on va associer un nouveau numéro $num[i]$, toujours entre 0 et $N-1$. On utilise une variable globale *next* et une marque sur chaque instruction. On initialise *next* à $N-1$ et on lance **dfs**(0), où **dfs** est la fonction suivante :

dfs(i)
  marquer i comme visitée
  pour chaque successeur $j < N$, non visité, de l'instruction i
    appeler **dfs**(j)
  $num[i] \leftarrow next$
  $next \leftarrow next - 1$

En supposant que toutes les instructions sont atteignables à partir de l'instruction 0, l'appel à **dfs**(0) va visiter toutes les instructions du graphe et terminer en donnant la valeur 0 à $num[0]$.

Question 13 Donner le résultat de cette numérotation sur le graphe de la figure 2.

Correction : La fonction **dfs** est appelée successivement sur $0, 1, 2, 3, 6, 7, 4, 5$, ce qui donne la numérotation suivante :

```
0 -> 0
1 -> 1
2 -> 2
3 -> 3
6 -> 4
7 -> 5
4 -> 6
5 -> 7
```

Question 14 Donner un exemple de graphe de flot de contrôle montrant que la numérotation en profondeur ne donne pas forcément un résultat optimal, au sens où, après numérotation en profondeur, l'algorithme d'allocation a besoin de plus de registres physiques qu'avec la numérotation initiale.

Correction : Avec le programme suivant, l'algorithme n'a besoin que de deux registres physiques

```
0: mov 1 t -> 1
1: sub t z -> 4
2: mov 3 t -> 3
3: sub t z -> 6
4: mov 2 u -> 5
5: sub u z -> 2
```

car les intervalles sont $u : [5, 5]$, $z : [0, 6]$ et $t : [1, 3]$. Mais avec la numérotation en profondeur, on obtient le programme

```
0: mov 1 t -> 1
1: sub t z -> 2
2: mov 2 u -> 3
3: sub u z -> 4
4: mov 3 t -> 5
5: sub t z -> 6
```

qui a besoin maintenant de trois registres car les deux utilisations de `t` sont maintenant de part et d'autre de l'utilisation de `u`.

---

3 Exécution

Dans cette dernière partie, on se préoccupe d'exécuter notre petit langage RTL. Un programme RTL est une liste de fonctions RTL, chacune étant donnée par son nom et son graphe. On suppose que cette liste contient au moins une fonction appelée `main`, et que c'est là le point d'entrée du programme.

Question 15 Décrire la réalisation en Java ou en OCaml d'un interprète du langage RTL. On ne demande pas d'écrire tout le code mais de décrire précisément les classes/types, méthodes/fonctions et structures de données utilisées. Détailler avec soin le cas d'un appel (`call`) et l'optimisation d'un appel terminal.

Correction : On commence par se donner de la syntaxe abstraite pour le langage RTL, en OCaml

```
type label = int
type var = string
type instr =
  | Movi  of int * var * label
```

```
    | Mov   of var * var * label
    | Sub   of var * var * label
    | Jnz   of label * label
    | Call  of var * string * var * var * label
  type program = (string * instr array) list
```

ou en Java

```
abstract class RTL {}
class Movi extends RTL { int n; String v;   int l;      }
class Mov  extends RTL { String u, v;       int l;      }
class Sub  extends RTL { String u, v;       int l;      }
class Jnz  extends RTL {                    int l1, l2; }
class Call extends RTL { String w, f, u, v; int l;      }
class RTLfun { String f; RTL[] code; }
class RTLprogram { List<RTLfun> funs; }
```

Pour l'interprète proprement dit, on se donne une table de hachage contenant les fonctions, où chaque nom de fonction est associé à son code (un tableau d'instructions).

L'interprète est essentiellement une fonction/méthode `doCall` avec comme paramètres un nom de fonction à appeler et deux valeurs pour x et y. À l'intérieur de `doCall`, on déclare

— une table de hachage `vars` pour les valeurs des variables ;
— un booléen `zero` pour la nullité de la dernière soustraction.

On fixe dans `vars` les valeurs de x et y, puis on exécute les instructions avec une boucle

```
while (pc < N)
```

où `pc` est l'étiquette de la prochaine instruction à exécuter (initialisée avec 0) et `N` est le nombre d'instructions et, on le rappelle, l'étiquette de sortie.

L'exécution des instructions autres que `call` est immédiate. Pour une instruction `call`, on rappelle récursivement `doCall`, ce qui a pour effet de créer une nouvelle table `vars` (ce que l'on veut, car les variables sont locales à chaque fonction). Le booléen `zero`, en revanche, pourrait être global, ce qui serait plus conforme aux drapeaux d'un processeur.

Pour optimiser un appel terminal, on l'identifie comme expliqué à la question Q3, puis on se contente de remettre `pc` à 0 après avoir donné aux variables x et y les valeurs des deux paramètres.

Une fois sorti de la boucle, on renvoie la valeur contenue dans `vars` pour la variable z.

Compilation vers l'assembleur x86-64. On souhaite maintenant compiler notre langage RTL vers l'assembleur x86-64. Les paramètres x et y d'une fonction sont passés respectivement dans `%rdi` et `%rsi` et son résultat z est passé dans `%rax`.

Question 16 Donner un code assembleur x86-64 pour le code RTL de la figure 2, avec une allocation au choix pour les différentes variables.

Correction : On alloue naturellement x et y dans `%rdi` et `%rsi` et z dans `%rax`. Pour le reste, on alloue a dans `%rdx` et t dans `%rcx` (des registres *caller-save*).

```
mult:   mov $0, %rax
        mov $0, %rdx
        sub %rsi, %rdx
        mov $0, %rcx
        jmp 2f
1:      sub %rdx, %rax
        mov $1, %rcx
2:      sub %rcx, %rdi
        jnz 1b
        ret
```

Question 17 Décrire la traduction du langage RTL vers l'assembleur x86-64. On suppose avoir réalisé une allocation de registres, donnée sous la forme d'une affectation *loc* (comme dans la partie précédente) envoyant toute variable vers un registre x86-64 ou vers la valeur `Spill`.

Correction : On peut suivre les étapes vues en cours :

1. Étape ERTL : On ajoute au début de chaque fonction

   ```
   push %rbp
   mov  %rsp, %rbp
   sub  $m, %rsp
   mov  %rdi, loc(x)  # sauf si loc(x)=%rdi
   mov  %rsi, loc(y)  # sauf si loc(y)=%rsi
   ```

   où m est la place nécessaires aux registres vidés en mémoire. De même, on ajoute à la fin de chaque fonction

   ```
   mov  loc(z), %rax  # sauf si loc(z)=%rax
   add  $m, %rsp
   pop  %rbp
   ret
   ```

   Par ailleurs, chaque appel `w <- call f(u,v)` devient

   ```
   mov  u, %rdi
   mov  v, %rsi
   call f
   mov  %rax, w
   ```

2. Étape LTL : Les instructions `mov u v` et `sub u v` doivent être décomposées lorsque `u` et `v` sont toutes les deux vidées en mémoire. On peut utiliser pour cela un temporaire, par exemple `%r11`, réservé à cet effet. Ainsi, `mov u v` devient

   ```
   mov ofsu(%rbp), %r11
   mov %r11, ofsv(%rbp)
   ```

   L'instruction `mov n v`, en revanche, ne pose pas de problème.

3. Linéarisation. Comme en cours, par un parcours de graphe.

Annexe : aide-mémoire x86-64

On donne ici un fragment du jeu d'instructions x86-64. Vous êtes libre d'utiliser tout autre élément de l'assembleur x86-64. Dans ce qui suit, r_i désigne un registre, n une constante entière et L une étiquette.

`mov` r_2`,` r_1	copie le registre r_2 dans le registre r_1
`mov` \$$n$`,` r_1	charge la constante n dans le registre r_1
`mov` L`,` r_1	charge la valeur à l'adresse L dans le registre r_1
`mov` \$$L$`,` r_1	charge l'adresse de l'étiquette L dans le registre r_1
`add` r_2`,` r_1	calcule la somme de r_1 et r_2 dans r_1 (on a de même `sub` et `imul`)
`mov` n`(`r_2`),` r_1	charge dans r_1 la valeur contenue en mémoire à l'adresse $r_2 + n$
`mov` r_1`,` n`(`r_2`)`	écrit en mémoire à l'adresse $r_2 + n$ la valeur contenue dans r_1
`push` r_1	empile la valeur contenue dans r_1
`pop` r_1	dépile une valeur dans le registre r_1
`cmp` r_2`,` r_1	positionne les drapeaux en fonction de la valeur de $r_1 - r_2$
`test` r_2`,` r_1	positionne les drapeaux en fonction de la valeur de $r_1 \& r_2$
`jnz` L	saute à l'adresse désignée par l'étiquette L en cas de résultat non 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

$$
\begin{array}{rcl}
\text{instruction RTL} & ::= & \texttt{mov}\ n\ v \to L \\
& | & \texttt{mov}\ v\ v \to L \\
& | & \texttt{sub}\ v\ v \to L \\
& | & \texttt{jnz} \to L, L \\
& | & v \leftarrow \texttt{call}\ f(v, v) \to L
\end{array}
$$

FIGURE 1 – Un langage RTL minimal.

$$
\begin{array}{l}
z\ \mathit{myfun}(x, y) \\
\quad 0: \ \texttt{mov}\ 0\ z \to 1 \\
\quad 1: \ \texttt{mov}\ 0\ a \to 2 \\
\quad 2: \ \texttt{sub}\ y\ a \to 3 \\
\quad 3: \ \texttt{mov}\ 0\ t \to 6 \\
\quad 4: \ \texttt{sub}\ a\ z \to 5 \\
\quad 5: \ \texttt{mov}\ 1\ t \to 6 \\
\quad 6: \ \texttt{sub}\ t\ x \to 7 \\
\quad 7: \ \texttt{jnz} \to 4, 8
\end{array}
$$

FIGURE 2 – Un exemple de graphe RTL.

allocation()
    actifs $\leftarrow \emptyset$
    libres $\leftarrow \{R_1, R_2, \ldots, R_K\}$
    pour chaque intervalle $v : [i, j]$, par ordre de i croissant
        expiration$(v : [i, j])$
        si *actifs* contient K éléments **alors**
            vidage$(v : [i, j])$
        sinon
            $loc[v] \leftarrow$ un registre retiré de *libres*
            ajouter $v : [i, j]$ à l'ensemble *actifs*

expiration$(v : [i, j])$
    pour chaque intervalle $v' : [i', j']$ de l'ensemble *actifs*, par ordre de j' croissant
        si $j' \geq i$ **alors sortir** de **expiration**
        enlever $v' : [i', j']$ de l'ensemble *actifs*
        ajouter $loc[v']$ à l'ensemble *libres*

vidage$(v : [i, j])$
    soit $v' : [i', j']$ un élément de *actifs* avec j' maximal
    si $j' > j$ **alors**
        $loc[v] \leftarrow loc[v']$
        $loc[v'] \leftarrow$ `Spill`
        enlever $v' : [i', j']$ de l'ensemble *actifs*
        ajouter $v : [i, j]$ à l'ensemble *actifs*
    sinon
        $loc[v] \leftarrow$ `Spill`

FIGURE 3 – Algorithme d'allocation des registres.

Énoncé en français.