École Normale Supérieure

Langages de programmation et compilation

examen 2014–2015

Jean-Christophe Filliâtre

15 janvier 2015

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

---

1 Enregistrements

On considère ici une variante de mini-ML avec des entiers et des enregistrements, dont la syntaxe abstraite est la suivante :

e	::=	n	constante entière
	|	$-$	soustraction
	|	x	variable
	|	`let` $x = e$ `in` e	déclaration locale
	|	`ifzero` e `then` e `else` e	conditionnelle
	|	`fun` $x \to e$	abstraction
	|	$e\ e$	application
	|	$\{x = e; \ldots; x = e\}$	construction d'un enregistrement
	|	$e.x$	accès à un champ

Comme on l'imagine, la construction `ifzero` teste si la première expression est l'entier 0.

Question 1 Équiper ce langage d'une sémantique opérationnelle à petits pas traduisant une sémantique d'appel par valeur avec évaluation de la gauche vers la droite. Plus précisément, on définira la notion de valeur v, les réductions de tête $\xrightarrow{\epsilon}$ et les contextes de réduction E.

Correction :

$$v ::= n \mid - \mid (- \ n) \mid \texttt{fun}\ x \to e \mid \{x = v; \ldots; x = v\}$$

On note que l'application partielle de la soustraction à un seul argument est une valeur.

$$\begin{gathered}
(- \ n_1)\ n_2 \xrightarrow{\epsilon} n_1 - n_2 \\
(\texttt{fun}\ x \to e)\ v \xrightarrow{\epsilon} e[x \leftarrow v] \\
\texttt{let}\ x = v\ \texttt{in}\ e \xrightarrow{\epsilon} e[x \leftarrow v] \\
\texttt{ifzero}\ n\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \xrightarrow{\epsilon} e_1 \quad \text{si } n = 0 \\
\texttt{ifzero}\ n\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \xrightarrow{\epsilon} e_2 \quad \text{si } n \neq 0 \\
\{x_1 = v_1; \ldots; x_n = v_n\}.x_i \xrightarrow{\epsilon} v_i
\end{gathered}$$

$$
\begin{array}{rcl}
E & ::= & \square \\
& | & E\ e \\
& | & v\ E \\
& | & \texttt{let}\ x = E\ \texttt{in}\ e \\
& | & \texttt{ifzero}\ E\ \texttt{then}\ e\ \texttt{else}\ e \\
& | & \{x = v; \ldots; x = v; x = E; x = e; \ldots; x = e\} \\
& | & E.x
\end{array}
$$

Question 2 Donner les étapes de réduction à petits pas de l'expression

```
(ifzero - 4 2 then fun x -> - x 1 else fun x -> - x 2) 44
```

ainsi que de l'expression

```
let id = fun x -> x in let r = { f = id id; g = id 42 } in r.g
```

Correction :

```
(ifzero - 4 2 then fun x -> - x 1 else fun x -> - x 2) 44 -->
(ifzero 2     then fun x -> - x 1 else fun x -> - x 2) 44 -->
(fun x -> - x 2) 44 -->
- 44 2 -->
42
```

```
let id = fun x -> x in let r = { f = id id; g = id 42 } in r.g -->
let r = { f = (fun x -> x) (fun x -> x); g = (fun x -> x) 42 } in r.g -->
let r = { f = fun x -> x; g = (fun x -> x) 42 } in r.g -->
let r = { f = fun x -> x; g = 42 } in r.g -->
{ f = fun x -> x; g = 42 }.g -->
42
```

Question 3 On souhaite typer ces programmes dans le système de Hindley-Milner, avec les types suivants

$$
\begin{array}{rcll}
\tau & ::= & \alpha \mid \tau \to \tau \mid \texttt{int} \mid \{x = \tau; \ldots; x = \tau\} & \text{type} \\
\sigma & ::= & \tau \mid \forall \alpha.\sigma & \text{schéma de type}
\end{array}
$$

où $\{x_1 : \tau_1; \ldots; x_n : \tau_n\}$ désigne le type (anonyme) d'un enregistrement ayant les champs $x_1, \ldots, x_n$, respectivement de types $\tau_1, \ldots, \tau_n$. Donner les règles de typage pour la soustraction (constante $-$), la construction `ifzero`, la construction d'un enregistrement et l'accès à un champ.

Correction :

$$
\frac{}{\Gamma \vdash - : \texttt{int} \to \texttt{int} \to \texttt{int}}
$$

$$
\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{ifzero}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}
$$

$$
\frac{\Gamma \vdash e_1 : \tau_1 \quad \cdots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \{x_1 = e_1; \ldots; x_n = e_n\} : \{x_1 : \tau_1; \ldots; x_n : \tau_n\}}
$$

$$
\frac{\Gamma \vdash e : \{x_1 : \tau_1; \ldots; x_n : \tau_n\}}{\Gamma \vdash e.x_i : \tau_i}
$$

Question 4 Discuter la pertinence de types d'enregistrements anonymes dans un contexte où on chercherait à faire de l'inférence de types (par opposition à un langage comme OCaml où les types d'enregistrements doivent être déclarés préalablement).

Correction : On ne saurait pas quel type donner à l'expression

$$\texttt{fun}\ x \to x.f$$

Plus précisément, cette fonction admet une infinité de types (x prenant tout type d'enregistrement possédant au moins un champ f) et il n'y a pas de type principal. Avec des types d'enregistrements nommés (comme en OCaml par exemple) le nom du champ f permet de retrouver l'unique type d'enregistrement correspondant.

Question 5 On adopte un schéma de compilation où tout enregistrement est un pointeur vers un bloc alloué sur le tas. Si un enregistrement contient n champs, ce bloc contient n mots et les champs y sont rangés par ordre alphabétique. Donner alors le code MIPS correspondant à la fonction suivante :

```
fun (p: { fst: int; snd: int }) ->
  ifzero p.fst then
    p
  else
    { snd = p.fst; fst = p.snd }
```

On supposera que l'argument `p` est passé dans le registre `$a0` et que le résultat de la fonction est renvoyé dans le registre `$v0`. Un aide-mémoire MIPS est donné à la fin du sujet.

Correction :

```
fun1:
   lw   $t0, 0($a0)
   bnez $t0, Lelse
   move $v0, $a0
   jr   $ra
Lelse:
   lw   $t1, 4($a0)
   li   $a0, 8
   li   $v0, 9
   syscall           # sbrk
   sw   $t0, 4($v0)  # p.fst est déjà dans $t0
   sw   $t1, 0($v0)
   jr   $ra
```

Question 6 On souhaite maintenant ajouter le caractère mutable aux champs des enregistrements. Indiquer les modifications qui doivent être apportées à la syntaxe, à la sémantique, au typage et à la production de code. En ce qui concerne le typage, en particulier, on prendra soin de discuter le problème des références polymorphes (s'il est absent, indiquer pourquoi ; s'il est présent, expliquer comment y remédier).

Correction :

syntaxe On ajoute une construction

$$e.x \leftarrow e$$

sémantique Dans la notion de valeur v, la construction $\{x = v; \ldots; x = v\}$ est remplacée par une adresse a et on ajoute une nouvelle valeur `()`. La relation de réduction devient $M, e \xrightarrow{\epsilon} M', e'$ où M est un dictionnaire (la mémoire) associant à des adresses des enregistrements de la forme $\{x_1 = v_n; \ldots; x_n = v_n\}$. Les nouvelles réductions de tête sont

$$M, \{x_1 = v_1; \ldots; x_n = v_n\} \xrightarrow{\epsilon} (M + a \mapsto \{x_1 = v_1; \ldots; x_n = v_n\}), a \quad \text{avec } a \notin M$$

et

$$M, a.x_i \leftarrow v \xrightarrow{\epsilon} M', () \qquad \begin{array}{l} \text{si } M(a) = \{x_1 = v_1; \ldots; x_n = v_n\} \\ \text{et avec alors } M' = M[a \leftarrow \{x_1 = v_1; \ldots; x_i = v; \ldots; x_n = v_n\}] \end{array}$$

Par ailleurs, la réduction de tête $\{x_1 = v_1; \ldots; x_n = v_n\}.x_i \xrightarrow{\epsilon} v_i$ devient

$$M, a.x_i \xrightarrow{\epsilon} M, v_i \quad \text{si } M(a) = \{x_1 = v_1; \ldots; x_n = v_n\}$$

Les autres réductions de tête se voient seulement ajouter un même dictionnaire M à gauche et à droite. Enfin, les contextes de réduction sont augmentés par

$$E ::= \ldots \mid E.x \leftarrow e \mid v.x \leftarrow E$$

typage On ajoute un type `unit` et la règle

$$\frac{\Gamma \vdash e_1 : \{x_1 : \tau_1; \ldots; x_n : \tau_n\} \quad e_2 : \tau_i}{\Gamma \vdash e_1.x_i \leftarrow e_2 : \texttt{unit}}$$

Oui, le problème des références polymorphes est bien présent ici, car on peut coder une référence comme un enregistrement avec un unique champ mutable (c'est ce qui est fait en OCaml, par exemple). Une solution consiste à utiliser la *value restriction*, c'est-à-dire à ne généraliser le type de e_1 dans `let` $x = e_1$ `in` e_2 seulement lorsque e_1 est syntaxiquement une valeur.

production de code La compilation de $e_1.x_i \leftarrow e_2$ consiste à compiler la valeur de e_1, par exemple dans `$t0`, puis la valeur de e_2, par exemple dans `$t1`, puis à effectuer

```
sw $t1, n($t0)
```

où n est la position du champ x_i dans l'enregistrement désigné par e_1. On connaît cette position grâce au type de e_1.

%2 f(%1)			L_8 :	%9 $\leftarrow$ 1	L_9
entrée L_1 sortie L_{17}			L_9 :	%10 $\leftarrow$ %8 $-$ %9	L_{10}
L_1 :	%3 $\leftarrow$ 0	L_2	L_{10} :	%3 $\leftarrow$ %7 $+$ %10	L_{11}
L_2 :	%4 $\leftarrow$ %1	L_3	L_{11} :	%11 $\leftarrow$ %1	L_{12}
L_3 :	%5 $\leftarrow$ 1	L_4	L_{12} :	%12 $\leftarrow$ 1	L_{13}
L_4 :	%6 $\leftarrow$ %4 $-$ %5	L_5	L_{13} :	%13 $\leftarrow$ %11 $-$ %12	L_{14}
L_5 :	*bgtz* %6	L_6, L_{16}	L_{14} :	%1 $\leftarrow$ %13	L_{15}
L_6 :	%7 $\leftarrow$ %3	L_7	L_{15} :	*goto*	L_2
L_7 :	%8 $\leftarrow$ %1	L_8	L_{16} :	%2 $\leftarrow$ %3	L_{17}

FIGURE 1 – Code RTL d'une fonction f.

2 Analyse de flot de données

Le contexte de ce problème est celui du langage RTL (*Register Transfer Language*). On va y réaliser une analyse statique, dans le but de réaliser des optimisations. On rappelle ici les différentes instructions du langage RTL :

i	::=	$r \leftarrow n$	chargement d'une constante
	|	$r \leftarrow r$	copie
	|	$r \leftarrow n(r)$	lecture en mémoire
	|	$n(r) \leftarrow r$	écriture en mémoire
	|	$r \leftarrow$ *unop* r	opération unaire
	|	$r \leftarrow r$ *binop* r	opération binaire
	|	*ubranch* r	branchement unaire
	|	*bbranch* r r	branchement binaire
	|	$r \leftarrow f(r, \ldots, r)$	appel de fonction
	|	$r \leftarrow$ *malloc*(n)	allocation
	|	*goto*	saut inconditionnel

Les conventions sont les suivantes : n désigne une constante entière, r un pseudo-registre et L une étiquette de code (à laquelle on trouve une instruction RTL). La figure 1 contient le code RTL d'une fonction f qui reçoit un argument dans le pseudo-registre %1 et renvoie un résultat dans le pseudo-registre %2. Le point d'entrée est l'étiquette L_1 et le point de sortie l'étiquette L_{17}.

Question 7 Donner un programme C possible pour le code RTL de la figure 1.

Correction :

```
int f(int x) {
  int s = 0;
  while (x-1 > 0) {
    s += x-1;
    x = x-1;
  }
  return s;
}
```

Expressions disponibles. Le but de notre analyse est de calculer, en chaque point du code RTL d'une fonction donnée, un ensemble d'*expressions disponibles*, c'est-à-dire de valeurs qui ont déjà été calculées et sont contenues dans des pseudo-registres. On représente de telles valeurs *symboliquement*, à l'aide de la syntaxe abstraite suivante :

$$\begin{array}{rcll} v & ::= & \alpha_i & \text{valeur arbitraire, inconnue} \\ & | & n & \text{constante entière} \\ & | & op\ v & \text{opération unaire} \\ & | & v\ op\ v & \text{opération binaire} \\ & | & v[n] & \text{accès en mémoire à l'adresse } v+n \end{array}$$

En particulier, la construction α_i nous permet de représenter une valeur arbitraire inconnue de l'analyse statique. C'est le cas notamment des arguments de la fonction. Ainsi, sur l'exemple de la figure 1, on donnera initialement au pseudo-registre %1 une valeur arbitraire α_1.

Plus précisément, on va calculer pour chaque étiquette L du graphe de flot de contrôle un ensemble $in(L)$ d'expressions disponibles avant l'exécution de l'instruction à l'étiquette L et un ensemble $out(L)$ d'expressions disponibles après son exécution. Sur l'exemple de l'instruction L_4 de la figure 1, on a les ensembles suivants

$$\begin{array}{rl} & in(L_4) = \{\alpha_2, \alpha_3, 1\} \\ L_4 : & \%6 \leftarrow \%4 - \%5 \quad L_5 \\ & out(L_4) = \{\alpha_2, \alpha_3, 1, \alpha_2 - 1\} \end{array}$$

où α_2 désigne la valeur contenue dans %1 et %4, et α_3 celle contenue dans %3.

Question 8 Donner des ensembles $in(L)$ et $out(L)$ possibles pour toutes les instructions de la figure 1 sous la forme d'un tableau :

étiquette L	$in(L)$	$out(L)$
L_1	$\{\alpha_1\}$	$\{\alpha_1, 0\}$
etc.		

Comme le code contient une boucle, on s'autorisera des approximations, en introduisant des valeurs arbitraires α_i aux endroits pertinents.

Correction :

étiquette L	$in(L)$	$out(L)$
L_1	$\{\alpha_1\}$	$\{\alpha_1, 0\}$
L_2	$\{\alpha_2, \alpha_3\}$	$\{\alpha_2, \alpha_3\}$
L_3	$\{\alpha_2, \alpha_3\}$	$\{\alpha_2, \alpha_3, 1\}$
L_4	$\{\alpha_2, \alpha_3, 1\}$	$\{\alpha_2, \alpha_3, 1, \alpha_2 - 1\}$
L_5	$\{\alpha_2, \alpha_3, 1, \alpha_2 - 1\}$	$\{\alpha_2, \alpha_3, 1, \alpha_2 - 1\}$
L_6	$\{\alpha_2, \alpha_3, 1, \alpha_2 - 1\}$	$\{\alpha_2, \alpha_3, 1, \alpha_2 - 1\}$
L_7	$\{\alpha_2, \alpha_3, 1, \alpha_2 - 1\}$	$\{\alpha_2, \alpha_3, 1, \alpha_2 - 1\}$
L_8	$\{\alpha_2, \alpha_3, 1, \alpha_2 - 1\}$	$\{\alpha_2, \alpha_3, 1, \alpha_2 - 1\}$
L_9	$\{\alpha_2, \alpha_3, 1, \alpha_2 - 1\}$	$\{\alpha_2, \alpha_3, 1, \alpha_2 - 1\}$
L_{10}	$\{\alpha_2, \alpha_3, 1, \alpha_2 - 1\}$	$\{\alpha_2, \alpha_3, 1, \alpha_2 - 1, \alpha_3 + \alpha_2 - 1\}$
L_{12}	$\{\alpha_2, \alpha_3, 1, \alpha_2 - 1, \alpha_3 + \alpha_2 - 1\}$	$\{\alpha_2, \alpha_3, 1, \alpha_2 - 1, \alpha_3 + \alpha_2 - 1\}$
L_{13}	$\{\alpha_2, \alpha_3, 1, \alpha_2 - 1, \alpha_3 + \alpha_2 - 1\}$	$\{\alpha_2, \alpha_3, 1, \alpha_2 - 1, \alpha_3 + \alpha_2 - 1\}$
L_{14}	$\{\alpha_2, \alpha_3, 1, \alpha_2 - 1, \alpha_3 + \alpha_2 - 1\}$	$\{\alpha_2, \alpha_3, 1, \alpha_2 - 1, \alpha_3 + \alpha_2 - 1\}$
L_{15}	$\{\alpha_2, \alpha_3, 1, \alpha_2 - 1, \alpha_3 + \alpha_2 - 1\}$	$\{\alpha_2, \alpha_3, 1, \alpha_2 - 1, \alpha_3 + \alpha_2 - 1\}$
L_{16}	$\{\alpha_4\}$	$\{\alpha_4\}$
L_{17}	$\{\alpha_4\}$	—

Calcul des expressions disponibles. Pour calculer les ensembles $in(L)$ et $out(L)$ de façon systématique, on va commencer par calculer les valeurs respectivement définies et invalidées par *une* instruction RTL. On notera respectivement $gen(L)$ et $kill(L)$ ces deux ensembles pour l'instruction située à l'étiquette L.

Question 9 Définir les ensembles $gen(L)$ et $kill(L)$ pour chaque instruction du langage RTL, en complétant le tableau suivant :

Instruction RTL	$gen(L)$	$kill(L)$
$r_1 \leftarrow r_2$ *binop* r_3	$\{e(r_2)$ *binop* $e(r_3)\}$	toute expression associée à r_1
etc.		

où $e(r)$ dénote la valeur de $in(L)$ contenue dans r, le cas échéant, et une nouvelle valeur α_i sinon.

Correction :

Instruction RTL	$gen(L)$	$kill(L)$
$r_1 \leftarrow n$	$\{n\}$	toute expression associée à r_1
$r_1 \leftarrow r_2$	$\{e(r_2)\}$	toute expression associée à r_1
$r_1 \leftarrow n(r_2)$	$\{e(r_2)[n]\}$	toute expression associée à r_1
$n(r_1) \leftarrow r_2$	$\emptyset$	toute expression contenant un $n'(r)$
$r_1 \leftarrow$ *unop* r_2	$\{$*unop* $e(r_2)\}$	toute expression associée à r_1
$r_1 \leftarrow r_2$ *binop* r_3	$\{e(r_2)$ *binop* $e(r_3)\}$	toute expression associée à r_1
ubranch r_1	$\emptyset$	$\emptyset$
bbranch r_1 r_2	$\emptyset$	$\emptyset$
$r_0 \leftarrow f(r_1, \ldots, r_n)$	$\{\alpha_i\}$ frais	toute expression associée à r_0
$r_1 \leftarrow malloc(n)$	$\{\alpha_i\}$ frais	toute expression associée à r_1
goto	$\emptyset$	$\emptyset$

Question 10 Donner des équations définissant $in(L)$ et $out(L)$

$$\begin{cases} in(L) &= \ldots \\ out(L) &= \ldots \end{cases}$$

en fonction de $gen(L)$, $kill(L)$, $in(L)$ et $out(L)$.

Correction :

$$\begin{cases} in(L) &= \bigcap_{p \in pred(L)} out(L) \\ out(L) &= gen(L) \cup (in(L) - kill(L)) \end{cases}$$

L'intersection traduit le fait qu'une expression est disponible en L si elle est disponible sur *tous* les chemins menant à L.

Question 11 Donner les types et profils de fonctions OCaml que vous écririez pour réaliser une telle analyse, en supposant donnés des types `Rtl.t` pour les instructions RTL, `Register.t` pour les pseudo-registres et `Label.t` pour les étiquettes de code. (On ne demande pas d'écrire le code OCaml, mais seulement de décrire son architecture.)

Correction : On introduit bien sûr un type `expression` pour représenter les expressions disponibles :

```
type expression = Alpha of int | Const of int | ...
```

La difficulté, ensuite, vient du fait qu'on ne peut pas de contenter de calculer les expressions disponibles. Il faut aussi conserver l'information sur les pseudo-registres qui les contiennent à chaque instant. Là, plusieurs choix sont possibles, selon la finesse de l'analyse. On peut par exemple indiquer pour chaque expression disponible un *ensemble* de pseudo-registres la contenant.

```
type available_expressions = Register.Set.t Expression.Map.t
```

en supposant un module `Register.Set` pour des ensembles de pseudo-registres et un module `Expression.Map` pour des dictionnaires indexés par des expressions.
On peut alors des fonctions `gen` et `kill` avec le type

```
val gen, kill: available_expressions -> Rtl.t -> available_expressions
```

puis des fonctions ensemblistes pour réaliser les opérations utilisées dans la question précédente :

```
val inter, diff, union: available_expressions ->
                        available_expressions ->
                        available_expressions
```

Reste alors la fonction d'analyse proprement dite, qui prend un graphe de flot de contrôle en argument et renvoie le résultat de l'analyse :

```
type control_flow_graph = Rtl.t Label.Map.t

type result = {  in: available_expressions Label.Map.t;
                out: available_expressions Label.Map.t; }

val available_expressions: control_flow_graph -> result
```

C'est dans cette dernière fonction que se cache un calcul de point fixe (qui n'était pas demandé ici).

Sous-expressions communes. On souhaite maintenant exploiter le résultat de l'analyse des expressions disponibles pour effectuer l'optimisation dite des *sous-expressions communes*, qui consiste à éviter de faire plusieurs fois les mêmes calculs.

Question 12 Identifier les calculs redondants sur l'exemple de la figure 1, en montrant notamment comment l'analyse des expressions disponibles a permis de les trouver. Donner un code RTL simplifié en conséquence pour la fonction f.

Correction : On calcule trois fois de suite $\%1 - 1$ pour la même valeur de $\%1$. Les deux derniers calculs peuvent être supprimés. L'analyse des expressions disponibles permet de l'identifier, car les instructions L_9 et L_{13} calculent $\alpha_2 - 1$ avec la valeur $\alpha_2 - 1$ disponible, en l'occurrence dans $\%6$. Il suffit donc de remplacer ces deux instructions par un `move` à chaque fois. Du coup, les instructions L_7, L_8, L_{11} et L_{12} deviennent du code mort (ce que l'analyse de durée de vie, mise à jour, montre facilement) et peuvent être éliminées. Au final, on obtient

$\%2$ $f(\%1)$
entrée L_1 sortie L_{17}

L_1 :	$\%3 \leftarrow 0$	L_2
L_2 :	$\%4 \leftarrow \%1$	L_3
L_3 :	$\%5 \leftarrow 1$	L_4
L_4 :	$\%6 \leftarrow \%4 - \%5$	L_5
L_5 :	*bgtz* $\%6$	L_6, L_{16}
L_6 :	$\%7 \leftarrow \%3$	L_9
L_9 :	$\%10 \leftarrow \%6$	L_{10}
L_{10} :	$\%3 \leftarrow \%7 + \%10$	L_{13}
L_{13} :	$\%13 \leftarrow \%6$	L_{14}
L_{14} :	$\%1 \leftarrow \%13$	L_{15}
L_{15} :	*goto*	L_2
L_{16} :	$\%2 \leftarrow \%3$	L_{17}

Question 13 Donner un exemple de code RTL où une même expression est disponible en un certain point du programme, mais contenue dans deux pseudo-registres différents selon le chemin du graphe de flot de contrôle qui a permis d'atteindre ce point de programme.

Correction : Dans un langage de haut niveau, ce pourrait être

```
if (...) y = x-1; else z = x-1;
```

Après cette conditionnelle, l'expression `x-1` est disponible, mais dans deux pseudo-registres différents selon la branche de la conditionnelle qui a été suivie. Traduit en RTL, cela pourrait être

```
L1: beqz %2          --> L2, L6
L2: %3 <- %1         --> L3
L3: %4 <- 1          --> L4
L4: %5 <- %3 - %4    --> L5
L5: goto             --> L9
L6: %6 <- %1         --> L7
L7: %7 <- 1          --> L8
L8: %8 <- %6 - %7    --> L9
L9:
```

et le point qui nous intéresse ici est L_9.

Question 14 Décrire de manière générale comment l'analyse des expressions disponibles doit être utilisée pour réaliser l'optimisation des sous-expressions communes. On prendra soin d'expliquer comment le code mort résultant de cette optimisation est éliminé.

Correction : On a déjà en partie répondu dans la question 12 : si une instruction calcule une expression qui est disponible et que celle-ci si trouve dans un pseudo-registre, il suffit de remplacer le calcul par un `move`. Mais la question précédente montre qu'une expression peut être disponible sans se trouver pour autant dans un registre donné. Il peut y avoir par exemple dans branches dans le graphe de flot de contrôle qui précède, l'une mettant la valeur de l'expression dans r_1 et l'autre dans r_2. Dans ce cas, on choisit un nouveau pseudo-registre r, dans chaque branche on y copie la valeur de l'expression, puis enfin on réalise l'optimisation avec un `move` prenant la valeur dans r.

Le code mort résultant de l'optimisation peut être éliminé facilement en mettant à jour l'analyse de durée de vie. En effet, on peut éliminer toute instruction qui affecte un pseudo-registre qui n'est plus vivant après cette instruction. Dans notre exemple, l'instruction L_7 affecte le pseudo-registre %8 qui n'est plus vivant après cette instruction suite à l'optimisation. De même pour les instructions L_8, L_{11} et L_{12}.

Annexe : aide-mémoire MIPS

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

`li`	r_1, n	charge la constante n dans le registre r_1
`la`	r_1, L	charge l'adresse de l'étiquette L dans le registre r_1
`addi`	r_1, r_2, n	calcule la somme de r_2 et n dans r_1
`add`	r_1, r_2, r_3	calcule la somme de r_2 et r_3 dans r_1 (on a de même `sub`, `mul` et `div`)
`move`	r_1, r_2	copie le registre r_2 dans le registre r_1
`lw`	r_1, n(`(`r_2`)`)	charge dans r_1 la valeur contenue en mémoire à l'adresse $r_2 + n$
`sw`	r_1, n(`(`r_2`)`)	écrit en mémoire à l'adresse $r_2 + n$ la valeur contenue dans r_1
`beq`	r_1, r_2, L	saute à l'adresse désignée par l'étiquette L si $r_1 = r_2$ (on a de même `bne`, `blt`, `ble`, `bgt` et `bge`)
`beqz`	r_1, L	saute à l'adresse désignée par l'étiquette L si $r_1 = 0$ (on a de même `bnez`, `bltz`, `blez`, `bgtz` et `bgez`)
`j`	L	saute à l'adresse désignée par l'étiquette L
`jr`	r_1	saute à l'adresse contenue dans le registre r_1
`jal`	L	saute à l'adresse désignée par l'étiquette L, après avoir sauvegardé l'adresse de retour dans `$ra`
`jalr`	r_1	saute à l'adresse contenue dans le registre r_1, après avoir sauvegardé l'adresse de retour dans `$ra`

Quelques appels systèmes (`syscall`) :

appel	`$v0` (entrée)	`$a0` (entrée)	`$v0` (sortie)
`print_char`	11	caractère à afficher	
`print_int`	1	entier à afficher	
`print_string`	4	pointeur vers la chaîne à afficher	
`read_int`	5		entier lu
`sbrk` (`malloc`)	9	nombre d'octets à allouer	pointeur vers le bloc alloué