École Polytechnique

INF564 : Compilation

examen 2022 (X2019)

Jean-Christophe Filliâtre

14 mars 2022

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

1 Opérateur &

Dans ce problème, on considère un tout petit fragment du langage C dont la syntaxe abstraite est donnée figure 1. Dans ce fragment, on a des entiers et pointeurs. Si p est un pointeur de type τ`*`, vers un emplacement mémoire contenant une valeur de type τ, alors `*`p désigne la valeur de type τ à cet emplacement. Si e est une valeur gauche de type τ alors `&`e désigne un pointeur vers cet emplacement, et ce pointeur a le type τ`*`. Dans ce fragment, une valeur gauche est soit une variable, soit une expression de la forme `*`e.

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. Les fonctions ne renvoient pas de valeur. Toutes les variables locales sont déclarées en début de fonction. On rappelle que la construction `if` du langage C teste si son argument est non nul. Ici, on ne teste que des valeurs entières (il n'y a pas de pointeur nul et donc aucune raison de tester un pointeur). Voici un exemple de programme dans ce fragment :

```
void loop(int *a, int *b, int n) {
  if (n) { *b = *b - (0 - *a); *a = *b - *a; loop(a, b, n - 1); }
}
void fib(int *x, int n) {
  int b; *x = 0; b = 1; loop(x, &b, n);
}
```

La fonction `fib` calcule le `n`-ième nombre de Fibonacci et l'écrit dans `*x`. Pour cela, elle alloue une variable locale `b`, dont elle passe l'adresse à la fonction `loop`. On prendra le temps de bien lire et de bien comprendre cet exemple.

1.1 Sémantique

Question 1 Donner le code d'une fonction `void mul(int *x, int y)` qui a pour effet de multiplier `*x` par `y`, c'est-à-dire le même effet que l'affectation `*x = *x * y` si on avait une multiplication à notre disposition. On suppose que la valeur de `y` est positive ou nulle.

e	::=	**expression**
	n	constante entière
	x	variable
	e `-` e	soustraction
	`*`e	déréférencement de pointeur
	`&`e	prise d'adresse
s	::=	**instruction**
	e `=` e	affectation
	f`(`$e, \ldots, e$`)`	appel de fonction
	`if (`e`)` s `else` s	conditionnelle
	`{` $s \ldots s$ `}`	bloc
d	::=	**définition de fonction**
	`void` f`(`$\tau\ x, \ldots, \tau\ x$`)` `{` $\tau\ x$`;` $\ldots$ $\tau\ x$`;` $s \ldots s$ `}`	
τ	::=	**type**
	`int`	entier
	τ`*`	pointeur
p	::=	**programme**
	$d \ldots d$	

FIGURE 1 – Syntaxe abstraite.

Sémantique opérationnelle des expressions. On souhaite munir notre langage d'une sémantique opérationnelle à grands pas sous la forme d'un premier jugement

$$E, M, e \twoheadrightarrow v$$

qui s'interprète comme « dans l'environnement E et la mémoire M, l'évaluation de l'expression e termine, avec la valeur v ». On suppose que toutes les variables sont allouées en mémoire et E est une fonction donnant l'emplacement de chaque variable dans la mémoire. Une valeur v peut être de deux sortes :

v	::=		**valeur**
		n	constante entière
		a	adresse mémoire

Une valeur de la forme a désigne une adresse mémoire et la valeur située à cette adresse est donc $M(a)$.

Question 2 Donner les règles d'inférence définissant la relation $E, M, e \twoheadrightarrow v$.

Question 3 Donner un environnement E, une mémoire M et une expression e pour lesquels il n'existe pas de valeur v telle que $E, M, e \twoheadrightarrow v$.

Sémantique opérationnelle des instructions. Pour les instructions, on se donne un second jugement

$$E, M, s \twoheadrightarrow M'$$

qui se lit comme « dans l'environnement E et la mémoire M, l'évaluation de l'instruction s termine, en aboutissant à une mémoire finale M' ». On donne la règle pour un appel de fonction :

$$\frac{\begin{array}{c}\texttt{void}\ f\texttt{(}\tau_1\ x_1, \ldots, \tau_n\ x_n\texttt{)\{}\ \tau'_1\ y_1\texttt{;}\ \ldots \tau'_m\ y_m\texttt{;}\ s_1\texttt{;}\ldots\texttt{;}s_k\ \texttt{\}} \\ E, M, e_i \twoheadrightarrow v_i \\ E' \stackrel{\text{def}}{=} \{x_1 \mapsto a_1, \ldots, x_n \mapsto a_n,\quad y_1 \mapsto a'_1, \ldots, y_m \mapsto a'_m\} \quad a_j, a'_j \text{ fraîches} \\ M' \stackrel{\text{def}}{=} M + \{a_j \mapsto v_j \mid 1 \leq j \leq n\} + \{a'_j \mapsto v'_j \mid 1 \leq j \leq m\} \\ E', M', \{\ s_1; \ldots; s_k\ \} \twoheadrightarrow M''\end{array}}{E, M, f\texttt{(}e_1, \ldots, e_n\texttt{)} \twoheadrightarrow M''_{|M}}$$

Ici, les valeurs v'_j désignent des valeurs quelconques utilisées pour initialiser les variables locales $y_1, \ldots, y_m$ de la fonction. La notation $M''_{|M}$ désigne la fonction M'' restreinte au domaine de la fonction M. On exprime ainsi que les variables locales de la fonction f ne sont plus visibles après l'appel à f.

Question 4 Donner les autres règles d'inférence définissant la relation $E, M, s \twoheadrightarrow M'$. (Qu'on se rassure, c'est *beaucoup* plus simple que pour la règle précédente.)

1.2 Analyse syntaxique

On souhaite réaliser l'analyse syntaxique de notre petit langage avec un outil de type YACC (par exemple CUP pour Java ou Menhir pour OCaml). Pour les expressions, on écrit notamment un morceau de grammaire de la forme suivante (la syntaxe précise peut légèrement varier selon l'outil), où `expr` est l'unique non terminal :

```
expr:
  | IDENT           {...}
  | CONST           {...}
  | expr MINUS expr {...}
  | STAR expr       {...}
  | AMP expr        {...}
  | LPAR expr RPAR  {...}
```

Les actions sémantiques ne nous intéressent pas ici et sont omises (`{...}`).

Question 5 Lorsque l'outil YACC est lancé sur le fichier ci-dessus, il déclare trois conflits de type lecture/réduction (`shift/reduce`). Les identifier, les expliquer, faire le choix de favoriser lecture ou réduction et expliquer comment indiquer ce choix à l'outil YACC. Les règles de grammaire ne doivent pas être modifiées.

1.3 Typage

On se propose maintenant de réaliser le typage statique de notre langage, notamment sous la forme d'un jugement de typage

$$\Gamma \vdash e : \tau$$

qui signifie « dans l'environnement Γ, l'expression e est bien typée de type τ ». L'environnement Γ est l'ensemble des variables connues, avec pour chacune son type.

Question 6 Donner les règles d'inférence définissant le jugement $\Gamma \vdash e : \tau$. Attention à bien limiter l'utilisation de l'opérateur `&` à des valeurs gauches.

Question 7 Donner un exemple d'expression mal typée ne contenant aucune variable.

Question 8 A-t-on la propriété de sûreté de typage, c'est-à-dire qu'une expression bien typée s'évalue toujours en une valeur ? Si oui, le démontrer. Sinon, donner un contre-exemple.

Typage des instructions. Pour typer les instructions, on se donne un second jugement de typage, de la forme

$$\Delta, \Gamma \vdash s$$

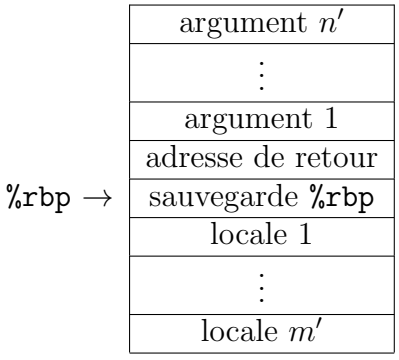
qui signifie « dans les environnements Δ et Γ, l'instruction s est bien typée ». L'environnement Δ est la liste des fonctions connues, avec pour chacune son profil.

Question 9 Donner les règles d'inférence définissant le jugement $\Delta, \Gamma \vdash s$.

Question 10 Donner enfin la règle permettant de vérifier qu'une définition de fonction est bien typée.

1.4 Compilation vers x86-64

On se propose maintenant de compiler notre langage vers l'assembleur x86-64. (Un aide-mémoire est donné en annexe.) On adopte le schéma de compilation suivant. Toutes les valeurs sont sur 64 bits (qu'il s'agisse d'entiers ou de pointeurs). Les variables sont allouées et passées dans des registres, lorsque c'est possible, et sur la pile sinon. On note qu'une variable x dont l'adresse est prise avec `&`x doit nécessairement être allouée sur la pile. Le tableau d'activation prend donc la forme ci-contre. L'adresse de retour est celle déposée par l'instruction `call`. Les valeurs de n' et m' dépendent du nombre de variables qu'on a dû allouer sur la pile. On suppose qu'on est ici dans le contexte d'un compilateur optimisant suivant l'architecture vue en cours (sélection d'instruction, RTL, ERTL, LTL, linéarisation).



Question 11 Donner un code x86-64 possible pour les fonctions `loop` et `fib` de la page 1, en optimisant l'appel terminal à `loop`.

Question 12 Indiquer à quelle(s) étape(s) du compilateur optimisant on va assurer qu'une variable x dont l'adresse est prise avec `&`x est bien allouée sur la pile.

Question 13 Expliquer comment compiler la construction `&`, en détaillant ce qui est fait aux différentes étapes du compilateur optimisant.

Question 14 Montrer que, bien que les fonctions ne renvoient pas de valeur, on a tout de même le problème des références fantômes (*dangling reference*) dans ce fragment de C.

#2 f(#1)
entrée L_1 sortie L_{16}

L_1 :	`mov` 0 #3	$\to L_2$
L_2 :	`mov` #1 #4	$\to L_3$
L_3 :	`mov` 1 #5	$\to L_4$
L_4 :	`binop sub` #5 #4	$\to L_5$
L_5 :	`ubranch >0` #4	$\to L_6, L_{15}$
L_6 :	`mov` #1 #6	$\to L_7$
L_7 :	`mov` 1 #7	$\to L_8$
L_8 :	`binop sub` #7 #6	$\to L_9$
L_9 :	`binop add` #6 #3	$\to L_{10}$
L_{10} :	`mov` #1 #8	$\to L_{11}$
L_{11} :	`mov` 1 #9	$\to L_{12}$
L_{12} :	`binop sub` #9 #8	$\to L_{13}$
L_{13} :	`mov` #8 #1	$\to L_{14}$
L_{14} :	`goto`	$\to L_2$
L_{15} :	`mov` #3 #2	$\to L_{16}$

FIGURE 2 – 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	::=	`mov` n $r \to L$	chargement d'une constante
	|	`mov` r $r \to L$	copie
	|	`load` $n(r)$ $r \to L$	lecture en mémoire
	|	`store` r $n(r) \to L$	écriture en mémoire
	|	`unop` op $r \to L$	opération unaire
	|	`binop` op r $r \to L$	opération binaire
	|	`ubranch` br $r \to L, L$	branchement unaire
	|	`bbranch` br r $r \to L, L$	branchement binaire
	|	`call` $r \leftarrow f(r, \ldots, r) \to L$	appel de fonction
	|	`goto` $\to L$	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 2 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_{16}.

Question 15 Donner un programme C possible pour le code RTL de la figure 2. (Note : On n'est plus dans le fragment C du problème 1.)

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 :

v	::=	α_i	valeur arbitraire, inconnue
	|	n	constante entière
	|	$op\ v$	opération unaire
	|	$v\ op\ v$	opération binaire
	|	$v[n]$	accès en mémoire à l'adresse $v + n$

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 2, 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 2, on a les ensembles suivants

$$\begin{array}{rl} & in(L_4) = \{\alpha_2, \alpha_3, 1\} \\ L_4 : & \texttt{binop sub \#5 \#4} \ \rightarrow L_5 \\ & out(L_4) = \{\alpha_2, \alpha_3, 1, \alpha_2 - 1\} \end{array}$$

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

Question 16 Donner des ensembles $in(L)$ et $out(L)$ possibles pour toutes les instructions de la figure 2 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.

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

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

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

Question 18 Donner des équations définissant $in(L)$ et $out(L)$

$$\left\{ \begin{array}{rcl} in(L) & = & \ldots \\ out(L) & = & \ldots \end{array} \right.$$

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

Question 19 Donner les types et fonctions OCaml ou classes et méthodes Java que vous écririez pour réaliser une telle analyse, en supposant donnés des types/classes pour les instructions RTL, pour les pseudo-registres et pour les étiquettes de code (comme dans le projet). On ne demande pas d'écrire le code, mais seulement de décrire son architecture.

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 20 Identifier les calculs redondants sur l'exemple de la figure 2, 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.

Question 21 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.

Question 22 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.

Annexe : aide-mémoire x86-64

Dans ce qui suit, L désigne une étiquette, r 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
`add` o`,` r	calcule $r + o$ et l'affecte à r
`add` r`,` o	calcule $o + r$ et l'affecte à o
`add` `$`n`,` o	calcule $o + n$ et l'affecte à o
`shl` `$`n`,` r	décale les bits de r vers la gauche, n fois
`lea` n`(`r_1`),` r_2	affecte à r_2 l'adresse représentée par n`(`r_1`)`, c'est-à-dire $r_1 + n$
`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

On pourra utiliser librement toute autre instruction x86-64. Mais les instructions ci-dessus suffisent pour répondre aux questions de ce sujet.