Compilation Avancée
Université Paris Diderot – Paris 7 – Master 2
Examen

Durée : 3 heures

Tout document autorisé. Matériel électronique interdit.

17 décembre 2009

Exercice : Forme SSA

Soit le programme $\mathrm{R}ex$ suivant :

ℓ_1 : **loadi** $\%r0, 0 \rightarrow \ell_2$
ℓ_2 : **loadi** $\%r1, 2 \rightarrow \ell_3$
ℓ_3 : **loadi** $\%r2, 30 \rightarrow \ell_4$
ℓ_4 : **loadi** $\%r3, 1 \rightarrow \ell_5$
ℓ_5 : **ifz** $\%r2 \rightarrow exit, \ell_6$
ℓ_6 : **ifeq** $\%r1, 1 \rightarrow \ell_7, \ell_{10}$.
ℓ_7 : **add** $\%r1, \%r1, \mathbf{1} \rightarrow \ell_8$
ℓ_8 : **add** $\%r2, \%r2, \%r0 \rightarrow \ell_9$
ℓ_9 : **jump** $\rightarrow \ell_6$
ℓ_{10} : **mul** $\%r3, \%r3, \mathbf{2} \rightarrow \ell_{11}$
ℓ_{11} : **sub** $\%r2, \%r2, \%r0 \rightarrow \ell_{12}$
ℓ_{12} : **sub** $\%r2, \%r2, \%r1 \rightarrow \ell_{13}$
ℓ_{13} : **jump** $\rightarrow \ell_5$

▷ Questions :

1. Dessinez le graphe de flot de contrôle de ce programme.
2. Mettez ce programme en forme SSA.
3. Appliquez enfin les optimisations vues en cours pour simplifier la forme SSA de ce programme.

$$
\begin{array}{rcll}
x\{y \mapsto e\} & = & x & \text{si } x \neq y \\
x\{y \mapsto e\} & = & e & \text{si } x = y \\
\mathbf{n}\{y \mapsto e\} & = & \mathbf{n} & \mathbf{n} \in \mathbb{N} \\
(\mathbf{fun}\ x \Rightarrow e')\{y \mapsto e\} & = & \mathbf{fun}\ x \Rightarrow e'\{y \mapsto e\} & \text{si } x \neq y \text{ et } x \notin FV(e) \\
(e_1\ e_2)\{y \mapsto e\} & = & e_1\{y \mapsto e\} e_2\{y \mapsto e\} & \\
(\mathbf{fix}\ x.\, e')\{y \mapsto e\} & = & \mathbf{fix}\ x.\, (e'\{y \mapsto e\}) & \text{si } x \neq y \text{ et } x \notin FV(e)
\end{array}
$$

FIG. 1 – Substitution sans capture.

$$
\begin{array}{rcll}
(\mathbf{fun}\ x \Rightarrow e)\ v & \rightarrow & e\{x \mapsto v\} & \textit{(}\beta\textit{-reduction)} \\
(\mathbf{fix}\ f.\, e)\ v & \rightarrow & (e\{f \mapsto \mathbf{fix}\ f.\, e\})\ v & \textit{(auto-application)} \\
\mathcal{C}\,[e] & \rightarrow & \mathcal{C}\,[e'] & \textit{(réduction-sous-contexte)} \\
 & & & \textit{si } e \rightarrow e'
\end{array}
$$

où

$$
\begin{array}{rcll}
\mathcal{C} & ::= & & \textit{Contexte d'évaluation} \\
 & | & [] & \textit{Trous} \\
 & | & \mathcal{C}\ e & \textit{Réduction à gauche} \\
 & | & v\,\mathcal{C} & \textit{Réduction à droite}
\end{array}
$$

FIG. 2 – Sémantique opérationnelle de Ha*ml*et

Problème : Analyse de motifs en présence de vues

On rappelle la syntaxe d'Ha*ml*et :

$$
\begin{array}{rcll}
e & ::= & & \textit{Expression} \\
 & | & x, f, \ldots & \textit{Variable} \\
 & | & \mathbf{0}, \mathbf{1}, \ldots, & \textit{Entier} \\
 & | & e\ e & \textit{Application} \\
 & | & \mathbf{fun}\ x \Rightarrow e & \textit{Fonction} \\
 & | & \mathbf{fix}\ x.\, e & \textit{Point fixe}
\end{array}
$$

Les figures 1 et 2 rappellent la sémantique opérationnelle de Ha*ml*et. Dans la suite, on pourra utiliser sans peur les sucres syntaxiques "**let** $x = \ldots$ **in** $\ldots$" et "**let rec** $f\ x = \ldots$ **in** $\ldots$"

Les types algébriques

La syntaxe d'Ha*ml*et est étendue par deux constructions traitant les types algébriques :

$$
\begin{array}{rcll}
e & ::= & \ldots & \\
 & | & K\ \overline{e} & \textit{Application d'un constructeur de données} \\
 & | & \mathbf{match}\ e\ \mathbf{with}\ \overline{c}\ \mathbf{end} & \textit{Analyse de motifs} \\
 & & & \\
c & ::= & p \Rightarrow e & \textit{Cas d'analyse} \\
 & & & \\
p & ::= & & \textit{Motif} \\
 & | & x & \textit{Variable} \\
 & | & K\ \overline{p} & \textit{Constructeur}
\end{array}
$$

Pour finir, un programme est composé d'un ensemble de déclarations de constructeurs de données et d'une expression.

▷ Questions :

1. Étendez les contextes d'évaluation pour prendre en compte ces deux constructions dans l'évaluation des expressions.
2. Étendez les règles de réduction en conséquence.

3. Écrivez une fonction `len` qui calcule la longueur d'une liste et détaillez les étapes de réduction de l'expression :

$$\text{len}(Cons(1, Cons(2, Nil)))$$

où $Cons$ est un constructeur d'arité 2 et Nil un constructeur d'arité 0.

4. Rappelez, en la spécifiant, la compilation d'un programme écrit dans Ha*mlet* étendu par des types algébriques en un programme Ha*mlet* standard de sémantique équivalente. Explicitez les primitives nécessaires à ce codage.

Les vues

On se propose maintenant d'étendre le langage des motifs à l'aide de **vues**. Une vue "$e \rightarrow p$" est un motif qui transforme la valeur à analyser à l'aide d'une expression e en une valeur d'un type algébrique juste avant de l'analyser par le motif p.

Par exemple, si on se donne les constructeurs suivants pour représenter des listes de façon non standard :

- `JNil` : la liste non standard vide ;
- `Join (l1, l2)` : la concaténation de deux listes non standards `l1` et `l2` ;
- `JSingle x` : la liste formée uniquement d'un entier x.

On peut se donner des constructeurs de données permettant de voir le premier niveau des listes non standards comme des listes standards :

- `VNil` : la liste standard vide ;
- `VCons (x, l)` : la liste formée d'un entier x et suivie de la **liste non standard** l.

La fonction suivante traduit une liste non standard en une liste standard :

```
let rec view l =
  match l with
  | JNil ⇒ VNil
  | JSingle (x) ⇒ VCons (x, Nil)
  | Join (l1, l2) ⇒
    match view l1 with
    | VNil ⇒ view l2
    | VCons (x, xs) ⇒ VCons (x, Join(xs, l2))
    end
  end
```

Enfin, le programme suivant calcule la somme des éléments d'une liste non standard à l'aide d'une vue :

```
let rec sum l =
  match l with
  | (view → Nil) ⇒ 0
  | (view → Cons (x, xs)) ⇒ x + sum xs
  end
```

▷ Questions :

1. Étendez la sémantique opérationnelle pour qu'elle traite les vues.
2. Rappelez les règles de typage de l'analyse de motifs (sans vue).
3. Proposez une règles de typage de l'analyse de motifs avec vue.
4. Étendez votre schéma de compilation pour prendre en compte les motifs construits à partir de vues.
5. Quelle optimisation, vue en cours, permet de ne calculer qu'une seule fois la vue correspondant à une valeur donnée ?

Problème : Vérification de code-octet pour une machine virtuelle

Machine virtuelle Une machine virtuelle implémente une machine abstraite (à pile) dont le code-octet s'appuie sur les instructions suivantes :

$$
\begin{array}{lll}
\textit{instruction} & ::= & \\
& | & \textit{Add} \\
& | & \textit{Sub} \\
& | & \textit{Mul} \\
& | & \textit{Div} \\
& | & \textit{Switch}\ (\mathbf{n}) \\
& | & \textit{Dup}\ (\mathbf{n}) \\
& | & \textit{PushInt}\ (\mathbf{n}) \\
& | & \textit{IfEq}\ (\mathbf{n}) \rightarrow l \\
& | & \textit{Jmp} \\
& | & \textit{RJmp} \\
& | & \textit{PushLabel}\ (l) \\
& | & \textit{Exit}
\end{array}
$$

La sémantique de ces instructions est donnée en langage naturel :
- Add/Sub/Mul/Div : Dépile 2 entiers et empile le résultat de l'opération ;
- Switch (**n**) : Échange la n-ième valeur de la pile et la $(n+1)$-ième.
- Dup (**n**) : Duplique la n-ième valeur de la pile.
- PushInt (**n**) : Empile l'entier n au sommet de la pile.
- IfEq (**n**) $\rightarrow$ l : Dépile un entier de la pile, le compare à **n**, continue en cas de succès ou saute en l en cas d'échec.
- Jmp : Dépile une étiquette et saute à cette étiquette.
- RJmp : Dépile une étiquette l, empile la position courante du pointeur de code, et saute à cette étiquette l.
- PushLabel (l) : Pousse l'étiquette l sur la pile.
- Exit : Arrête la machine.

Chaque programme fixe la forme de la pile d'entrée nécessaire à son exécution et spécifie comment lire sa sortie sur cette même pile.

▷ Question :

1. Formalisez la sémantique opérationnelle de cette machine.

Vérification de code-octet On désire intégrer à la machine virtuelle, un **vérificateur de code-octet** dans le but d'autoriser le chargement dynamique de code, éventuellement en provenance d'un tiers auquel on ne fait pas confiance, sans pour autant compromettre la sûreté d'exécution de la machine. Les invariants de sûreté de la machine sont de deux types :
- **Pénurie de la pile** : on doit s'assurer qu'on n'essaie jamais de dépiler un élément d'une pile vide ;
- **Bon typage de la pile** : les valeurs sur la pile sont du type attendu par les instructions de la machine.

La figure 3 regroupe quatre exemples de programmes écrits en code-octet. Les programmes **1** **2** et **3** attendent une pile contenant un entier et le programme **4** attend une pile vide.

▷ Questions :

1. Pour chacun d'eux, déterminez si on doit le rejeter et expliquez pourquoi.
   On se donne l'algèbre de types suivantes :

$$
\begin{array}{lll}
\sigma & ::= & \\
& | & \epsilon \qquad\qquad\qquad\qquad\qquad\qquad \textit{La pile vide} \\
& | & \mathbf{int}.\sigma \qquad\qquad\qquad\qquad\quad \textit{Une pile dont le sommet est un entier} \\
& | & (l_1, \ldots, l_n).\sigma \qquad \textit{Une pile dont le sommet contient l'une des étiquettes } l_1 \ldots l_n \\
& | & \top.\sigma \qquad\qquad\qquad\qquad\quad \textit{Une pile dont le sommet est d'une nature inconnue} \\
& & \bot \qquad\qquad\qquad\qquad\qquad\qquad \textit{Absence totale d'information}
\end{array}
$$

   À chaque instruction peut être associée une fonction de transition. Par exemple, la fonction de transition de mul est $\mathbf{int}.\mathbf{int}.\sigma \rightarrow \mathbf{int}.\sigma$.
2. Donnez les fonctions de transition de chacune des instructions.

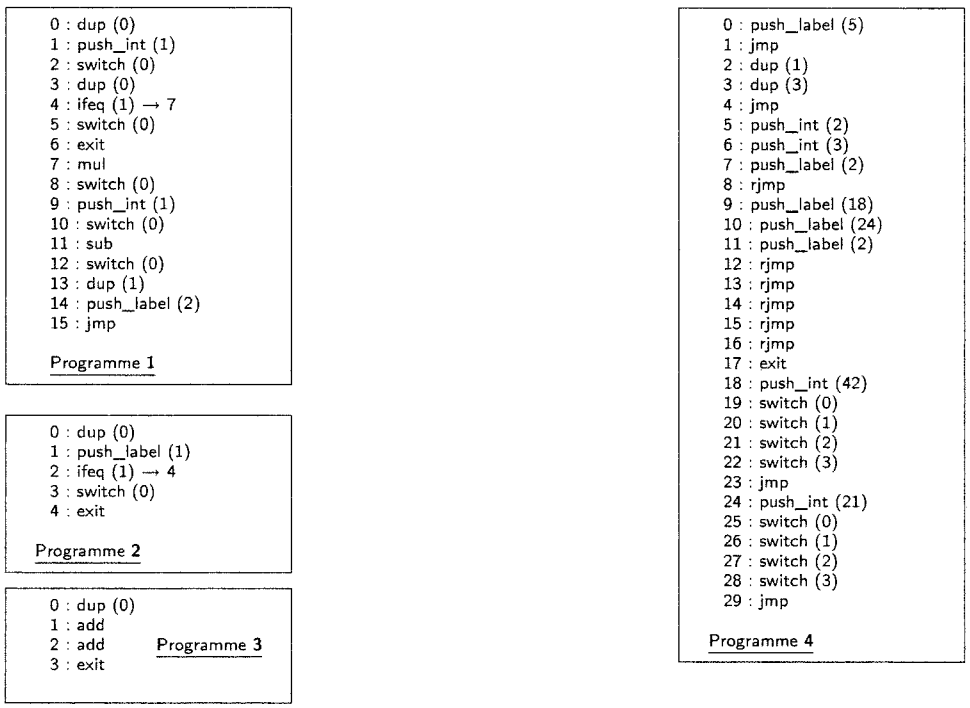

FIG. 3 – Quatre programmes écrits en code-octet.

3. Proposez une opération $\sqcup$, telle que $\sigma_1 \sqcup \sigma_2$ est une approximation de l'ensemble des piles approchées par σ_1 et par σ_2.
   À chaque position p du programme, on associe deux types $in(p)$, représentant l'état de la pile avant l'exécution de p, et $out(p)$ représentant l'état de la pile après l'exécution de p.
4. Donnez la définition de $in(p)$ et de $out(p)$ en fonction des états $out(q)$ de toutes les positions q prédécesseurs de p.
5. Est-ce que le calcul des prédécesseurs d'une position p est aisé ? Pourquoi ?
6. Proposez un algorithme d'analyse de flot de données permettant de vérifier les invariants de sûreté d'un code-octet.
7. Vérifiez les exemples donnés plus haut à l'aide de votre algorithme. Est-ce que tous les programmes sûrs sont acceptés par votre algorithme ? Si un programme sûr est rejeté, pouvez-vous expliquer pourquoi ?