Preuves d'optimisation et de compilation

Pierre Wilke
pierre.wilke@centralesupelec.fr

10 octobre 2018

Dans ce TP, on propose de travailler sur un simple langage d'expressions, sur lequel nous allons programmer des optimisations et un compilateur vers un langage à pile. Nous prouverons que les optimisations et le compilateur sont corrects.

1 Un langage d'expressions `Expr`

On considère le langage d'expressions suivant :

$$\begin{array}{rcl} \texttt{expr} & := & \texttt{Const}\ n \mid \texttt{Var}\ v \\ & \mid & \texttt{Add}\ e_1\ e_2 \mid \texttt{Mul}\ e_1\ e_2 \mid \texttt{Sub}\ e_1\ e_2 \end{array}$$

où n est un entier relatif (type `Z` en Coq), v est un identifiant de variable (type `var`, défini dans le squelette fourni), e_1 et e_2 sont des expressions.

Pour évaluer ces expressions, on a besoin d'un *environnement* qui associe à chaque identifiant de variable une valeur. Les environnements, dont le type est donné ci-dessous et défini dans le squelette, sont des fonctions qui prennent des variables et retournent des valeurs.

```
Definition envt := var -> val.
```

Question 1.1 *Écrire une fonction* `eval_expr` *qui prend un environnement* `env` *et une expression* `e`*, et qui retourne la valeur de l'expression dans cet environnement.*

Question 1.2 *Écrire une fonction* `pow` *qui prend une expression* `e` *et un entier naturel* `n`*, et qui retourne l'expression* $\texttt{e}^{\texttt{n}}$*.*

On souhaite créer des expressions qui correspondent à des polynômes. On représente les polynômes comme des listes de paires (c, n) où $c \in \mathbb{Z}$ est un coefficient et $n \in \mathbb{N}$ est un exposant. Chaque paire (c, n) correspond au monôme $c \cdot x^n$ et un polynôme est la somme de tous les monômes de la liste. Ainsi, la liste $[(2, 3); (4, 1)]$ correspond au polynôme $2 \cdot x^3 + 4 \cdot x$.

Question 1.3 *Écrire la fonction* `make_poly` *qui prend une telle liste et renvoie l'expression du polynôme correspondant. Cette fonction pourra utiliser la fonction* `pow` *définie précédemment.*

2 Optimisations

On cherche maintenant à optimiser ces expressions, c'est-à-dire les transformer en des expressions équivalentes, plus simples à évaluer. Pour cette optimisation, on part de l'observation qu'une expression sans variables s'évalue de la même manière quel que soit l'environnement.

Question 2.1 *Écrire une fonction* `no_vars` *qui prend une expression* `e` *et qui rend un booléen vrai si et seulement si l'expression ne contient pas de variables. La conjonction booléenne (le « et ») s'écrit* `&&` *(ou* `andb`*).*

Question 2.2 *Énoncer et prouver un lemme* `no_vars_constant` *qui exprime le fait qu'une expression sans variables s'évalue de la même manière dans tous les environnements. Vous aurez peut-être besoin du lemme suivant :*

```
andb_true_iff : forall b1 b2 : bool,
  b1 && b2 = true <-> b1 = true /\ b2 = true
```

Question 2.3 *Écrire une fonction* `constant_propagation` *qui prend une expression* `e` *et qui retourne une expression équivalente en remplaçant les sous-expressions sans variables par leur évaluation.*

Question 2.4 *Prouver que l'optimisation* `constant_propagation` *est correcte, c'est-à-dire :*

$$\forall\ env\ \forall\ e, \texttt{eval\_expr}\ env\ e = \texttt{eval\_expr}\ env\ (\texttt{constant\_propagation}\ e)$$

3 Une machine à pile

On s'intéresse maintenant à un langage de plus bas-niveau, qui opère sur une machine à pile. L'état d'une machine à pile est composé d'un environnement (comme pour le langage d'expressions) et d'une pile sur laquelle on stockera des valeurs.

On a trois instructions (type `instr`) dans le squelette qui vous est fourni :

- `SConst n` : écrit la valeur `n` au sommet de la pile ;
- `SLoad x` : charge la valeur de la variable `x` (depuis l'environnement) sur le sommet de la pile ;
- `SBinop b` : applique l'opérateur binaire `b` (`Badd`, `Bmul` ou `Bsub`) aux deux premières valeurs sur la pile. Plus précisément, si la valeur au sommet de la pile est v_1 et la valeur juste en dessous est v_2, on enlève v_1 et v_2 de la pile et on ajoute sur la pile $v_1 \oplus v_2$, où $\oplus$ représente l'opération effectuée par l'opérateur `b`.

L'évaluation d'une instruction dans ce langage dépend d'un environnement (pour l'instruction `SLoad`) et fait évoluer une pile. La fonction d'évaluation des instructions a le type suivant :

```
Definition eval_instr (i: instr) (e: envt) (s: stack) : option stack :=
  (* à compléter *).
```

Le type de retour de cette fonction est `option stack`. Le type `option A`[1] pour n'importe quel type `A` permet d'exprimer des valeurs de type `A` avec le constructeur `Some` et une absence de valeur avec le constructeur `None`.

Dans notre cas, l'évaluation des instructions peut échouer si la pile n'a pas assez de valeurs pour effectuer une opération. Dans ce cas, on retournera `None` pour dénoter l'absence de valeur de retour adéquate.

Question 3.1 *Compléter le corps de la fonction* ***`eval_instr`****.*

Un programme dans ce langage est une liste d'instructions. L'évaluation d'un programme est simplement l'évaluation des instructions les unes après les autres.

Question 3.2 *Écrire la fonction* ***`eval_prog`*** *qui évalue un programme, et dont la signature est donnée ci-dessous.*

```
Fixpoint eval_prog (p: list instr) (e: envt) (s: stack) : option stack :=
  (* à compléter *).
```

Question 3.3 *Soit l'environnement qui associe à la variable* ***`vx`*** *la valeur 3. Quel devrait être, selon vous (sans demander à Coq), la valeur de retour du programme suivant :*

```
[SLoad vx; SConst 1; SBinop Badd; SLoad vx; SBinop Bmul]
```

Vérifier que votre fonction ***`eval_prog`*** *donne bien votre réponse.*

4 Compilation d'expressions vers la machine à pile

On va à présent compiler les expressions vers des programmes de la machine à pile. Chaque expression va donc correspondre à une liste d'instructions, qui doivent calculer la même chose, et placer finalement le résultat sur le sommet de la pile.

Question 4.1 *Écrire une fonction* ***`compile_expr`*** *qui effectue la compilation des expressions. Vous aurez besoin de la fonction* ***`append`*** *qui concatène deux listes en les mettant bout-à-bout. Grâce aux notations, on pourra écrire* ***`l1 ++ l2`*** *au lieu de* ***`append l1 l2`****. (Utilisez la commande* ***`Print append`****. pour plus de détails si nécessaire).*

```
Fixpoint compile_expr (e: expr) : list instr := (* ... *).
```

On veut maintenant prouver que la compilation est *correcte*. C'est-à-dire que si p est le résultat de la compilation d'une expression e, alors en partant d'une pile `s`, l'évaluation du programme p doit retourner la pile `v :: s`, où v est le résultat de l'évaluation de e.

1. Vous pouvez vérfier cela en tapant `Print option.` dans `coqide`.

Question 4.2 *Avant de prouver ce théorème, on a besoin d'un lemme intermédiaire qui spécifie comment évaluer un programme* `p1 ++ p2`*. Compléter l'énoncé du lemme suivant et effectuer sa preuve :*

```
Lemma eval_prog_app:
  forall p1 e s p2,
    eval_prog (p1 ++ p2) e s =
    match eval_prog p1 e s with
    | Some s' => (* ??? *)
    | None => (* ??? *)
    end.
```

Question 4.3 *Énoncer le théorème de correction de la compilation (qui vous est donné plus haut informellement).*

Question 4.4 *Effectuer la preuve de ce théorème. Vous aurez peut-être besoin des lemmes suivants :*

```
Z.add_comm : forall n m : Z, n + m = m + n
Z.mul_comm : forall n m : Z, n * m = m * n
```

A Tactiques utiles

A.1 Apply

La tactique `apply` permet d'utiliser une hypothèse ou un lemme de la forme $A_1 \Rightarrow \cdots \Rightarrow A_n \Rightarrow B$ à un but de la forme B. Cela résoud le but courant et génère n sous-buts : un pour chaque A_i. Dans le cas particulier où $n = 0$, cette tactique ne génère aucun sous-but.

On peut aussi appliquer une telle hypothèse/un tel lemme `L` dans une hypothèse avec la variante `apply L in H.`, où `H` doit avoir la forme A_i. Cette tactique transforme l'hypothèse `H` en `B` (la conclusion de `L`) et génère $n - 1$ sous-buts : un pour chaque A_j, $j \neq i$.

A.2 Simpl

La tactique `simpl` déroule le corps des fonctions et tente d'évaluer au maximum. `simpl` s'applique sur la conclusion du but courant, `simpl in H` s'applique sur l'hypothèse `H` et `simpl in *` s'applique dans toutes les hypothèses et le but courant.

A.3 Rewrite

La tactique `rewrite L`, où `L` est un lemme / une hypothèse de la forme $a = b$, remplace les occurences de a par b dans le but courant. La variante `rewrite <- L` remplace les occurences de b par a. Chacune de ces variantes peut être appliquée à une hypothèse donnée en ajoutant `in H` après `L` ou dans toutes les hypothèses (`in *`).

A.4 Destruct

La tactique `destruct t` permet de déconstruire un terme de type inductif `t`. Selon le type de `t`, cela peut avoir plusieurs effets :

- si `t` est de la forme $A \wedge B$, alors `destruct t as (Ha & Hb)` générera deux hypothèses `Ha` et `Hb`, chacune contenant respectivement A et B.
- si `t` est de la forme $A \vee B$, alors `destruct t as [Ha | Hb]` générera deux sous-buts : un avec l'hypothèse `Ha`, l'autre avec l'hypothèse `Hb`.
- si `t` est un autre type de données inductif, `destruct t eqn:Ht` générera un sous-but par constructeur possible de `t`, avec dans chaque sous-but une hypothèse `Ht` qui sert à se souvenir de notre chemin dans la preuve. Par exemple, `destruct (f a) eqn:Hfa`, où `f` est un prédicat booléen, générera deux sous-buts : l'un avec `Hfa: f a = true` et l'autre avec `Hfa: f a = false`.

A.5 Induction

La tactique `induction t`, comme son nom l'indique, permet de raisonner par induction structurelle, selon le type de `t`. On peut utiliser l'induction sur

les entiers naturels ou sur les listes, comme on l'a vu en cours, mais aussi sur n'importe quel type inductif que l'on aura déclaré nous-mêmes (les expressions, peut-être ?)

A.6 Reflexivity

La tactique `reflexivity` résoud les buts de type `a = a`.

A.7 Inv

La tactique `inv H` (définie dans le squelette de ce TP) s'applique notamment aux buts de la forme `f a = g b` où `f` et `g` sont des constructeurs (`Some`, `None`, `nil`, `cons`...). Si `f` et `g` sont les mêmes constructeurs, alors la tactique `inv H` génère l'égalité `a = b` et essaie de remplacer `a` par `b` ou `b` par `a`.

A.8 Auto

La tactique `auto` permet de résoudre des buts simples :
— $a = a$
— $A \Rightarrow B, A \vdash B$
— $A \vdash A$

A.9 Séquence

Étant données deux tactiques t_1 et t_2, on peut chaîner ces deux tactiques avec un point-virgule : la tactique `t1; t2.` applique la tactique `t1`, puis applique `t2` à **tous les sous-buts** générés par la tactique `t1`.

Des usages classiques de la séquence :
— `induction n; simpl; auto.` preuve par induction sur `n`, simplifie le but et résoud les buts simples avec `auto`. Permet de se concentrer seulement sur les cas non-triviaux.
— `apply H; auto.` `apply` génère un certain nombre de sous-buts ; `auto` essaie d'en résoudre le maximum.