

Examen du 5 septembre 2005

- *Tous les documents sont autorisés.*
- **Les calembres et les téléphones portables sont interdits.**
- **Mettez votre nom sur la copie, cachez-la, puis écrivez votre numéro d'anonymat au dessus, et reportez ce numéro d'anonymat sur toutes les copies intercalaires.**
- *Cet énoncé comporte 1 exercice et 1 problème.*

Exercice : Cast en O'Caml

On cherche à définir des tests dynamiques de vérification des relations d'héritage et de sous-typage dans le but de construire des contraintes de type (descendantes au sens de l'héritage) sur les objets O'Caml. Chaque classe possèdera un identificateur unique (une valeur de type `cle`). Cette information sera connue de chaque instance de classe. Ainsi il sera possible de créer les informations sur l'arbre d'héritage et l'arbre de sous-typage.

On ajoute alors à chaque déclaration de classe deux méthodes :

- `method cc : cle` : retourne la clé de la classe de construction d'un objet ;
- `method instance_of : cle -> boolean` : teste si la classe de construction d'un objet hérite de la classe passée dont la clé est passée en paramètre ou d'un ancêtre de cette classe ;

ainsi qu'une déclaration globale `cle_NOMDELACLASSE` contenant une clé unique identifiant la classe. On utilise ensuite une macro `M1(expr, c)` qui produit le texte `let o = expr in o#instance_of cle_c` où `cle_c` est la clé de la classe `c`. Cette macro permet de tester si une expression `expr` s'évaluant en un objet `o` est une instance de la classe `c` ou d'un de ses ancêtres.

1. Ecrivez le corps de la méthode `cc` d'une classe `d`.
2. Ecrivez le corps de la méthode `instance_of` pour une classe `d` qui hérite directement des classes `a`, `b` et `c` en les nommant respectivement `super1`, `super2` et `super3`. Ces trois classes possèdent chacune une méthode `instance_of`.
3. On définit maintenant pour chaque classe une méthode `subtype_of : cle -> bool` qui indique si la classe de construction d'une instance est sous-type d'une classe passée en paramètre ainsi qu'une macro `M2(expr, c)` qui produit le texte `let o = expr in o#subtype_of cle_c`. Soient trois classes `a`, `m` et `z`, munies de constructeur sans paramètre, expliquez le résultat de l'appel de `M2(((new z) :> a), m)` avec `z` sous-classe de `m` et `m` sous-classe de `a` dans les cas suivants :
 - (a) `z` est sous-type de `a`, `z` est sous-type de `m`. Vous donnerez un exemple de telles classes.
 - (b) `z` est sous-type de `a`, `z` n'est pas sous-type de `m`. Vous donnerez un exemple de telles classes.
4. On suppose qu'il existe une fonction `magic : 'a -> 'b` qui retourne la valeur de l'argument (fonction identité) et qui autorise la pose d'une contrainte de type sur n'importe quelle expression du langage de la manière suivante : `((magic expr) : int)`. L'expression `expr` est alors considérée de type `int`. Ecrivez une macro `CAST(expr, c)` où `c` est un nom de classe et qui force `expr` à être considérée comme du type `c` et qui introduit un test qui vérifie à l'exécution si la classe de construction de l'objet `o` résultat de l'évaluation de `expr` est une instance de `c` (au sens de `instance_of`) et est un sous-type de `c` (au sens de `subtype_of`). Le test introduit déclenchera une exception si ce n'est pas le cas.

Problème : calcul propositionnel

On cherche à représenter les formules du calcul des propositions par un type O'CamL.

Une formule est soit une constante (vrai ou faux), soit une variable, soit un connecteur avec ses arguments.

On utilise les connecteurs \neg (négation) \vee (disjonction) \wedge (conjonction) \Rightarrow (implication) et \Leftrightarrow (équivalence).

calcul Pour calculer la valeur d'une formule, on utilisera une méthode due à Boyer et Moore qui interprète les connecteurs propositionnels comme des expressions conditionnelles sur les booléens.

1. Ecrire les tables de vérité des connecteurs \neg , \vee , \wedge , \Rightarrow et \Leftrightarrow .
2. Définir un type de données (récuratif) `prop` pour les formules du calcul des propositions. Les noms de variables seront des entiers.
3. Une expression conditionnelle est soit une variable (les mêmes que celles du calcul des propositions) soit une expression de la forme : *if p₁ then p₂ else p₃* où *p₁*, *p₂* et *p₃* sont des expressions conditionnelles. Définir un type de données `ifexpr` pour les expressions conditionnelles.
4. On cherche à déterminer l'expression conditionnelle (`ifexpr`) équivalente (logiquement) à chaque connecteur. Par exemple la formule $A \vee B$ s'écrit sous forme d'expression conditionnelle de la manière suivante : *if A then vrai else B*. Donner la traduction des autres connecteurs : \neg , \wedge , \Rightarrow et \Leftrightarrow .
5. Ecrire la fonction `ifexpr_of` de type `prop -> ifexpr` qui traduit une proposition en une expression conditionnelle équivalente.
6. L'algorithme de Boyer et Moore est défini pour un certain sous-ensemble des expressions conditionnelles : *les expressions canoniques*. On définit la "canonisation" d'une expression conditionnelle quelconque par :

$$\begin{array}{lll} \text{can}(x) & = & x \quad \text{,si } x \text{ est une variable} \\ \text{can}(\text{if}(x, p_1, p_2)) & = & \text{if}(x, \text{can}(p_1), \text{can}(p_2)) \quad \text{,si } x \text{ est une variable} \\ \text{can}(\text{if}(\text{if}(p_1, p_2, p_3), p_4, p_5)) & = & \text{can}(\text{if}(p_1, \text{if}(p_2, p_4, p_5), \text{if}(p_3, p_4, p_5))) \quad \text{,sinon} \end{array}$$

Ecrire la fonction `can` de type `ifexpr -> ifexpr` réalisant la mise en forme canonique.

vérificateur de tautologies On veut définir une fonction testant si une formule du calcul des propositions est une tautologie (formule toujours vraie) ou non. Le calcul d'une formule s'effectue dans un environnement où les variables de la formule sont liées à leur valeur. Pour vérifier qu'une formule est une tautologie, il faudra donc engendrer tous les environnements possibles de la formule et calculer la valeur de la formule pour chaque environnement.

Soit `vs` une liste d'association de type `(int * bool) list`, cette liste associe une valeur booléenne aux variables représentées par des entiers (voir fonction prédéfinie `List.assoc`). Soit `e` une expression conditionnelle, on définit l'algorithme de Boyer et Moore par la fonction `bm(vs, e)` par cas sur `e` :

Si `e` est la variable `x` alors `bm(vs, p)` est égal à la valeur associée à `x` dans `vs`

Sinon, `e` est de la forme `if(x, e1, e2)`.

Si `x` est définie dans `vs` alors

si la valeur de `x` dans `vs` est `true` alors `bm(vs, if(x, e1, e2)) = bm(vs, e1)`

sinon, la valeur de `x` dans `vs` est `false` alors `bm(vs, if(x, e1, e2)) = bm(vs, e2)`

sinon

soit `vs1` la liste obtenue en rajoutant l'association `(x, true)` à `vs`,

soit `vs2` la liste obtenue en rajoutant l'association `(x, false)` à `vs`,

`bm(vs, if(x, e1, e2)) = bm(vs1, e1) \wedge bm(vs2, e2)`

7. Ecrire la fonction `bm` de type `int * bool list -> ifexpr -> bool` qui implante l'algorithme de Boyer et Moore.