Projet Calcul Symbolique en OCaml

Zein SAKKOUR, Robin LACAZE

Dans ce projet, on propose de définir en OCaml un langage d'expressions mathématiques simples en notation préfixée. On cherche ensuite à évaluer (calculer la valeur) d'une expression de ce langage. Puis on cherche à calculer la dérivée symbolique d'une expression par rapport à une variable. Enfin, on cherche à simplifier des expressions à partir de règles simples telles que $x + 0 = x$ et $x * 1 = x$

1 - Définition de type

Nous avons choisi d'implanter en plus des types de bases, deux types d'opérations : les opérations binaires et les opérations unaires

```
type exp =
  |VAR of string
  |N of int
  |MONOP of string * exp
  |BINOP of string * exp * exp
```

On peut différencier les différentes opérations par le "label" (str). Par exemple l'addition de deux élements est une opération binaire. L'avantage de cette implémentation est qu'elle est industrielle, il est très facile d'ajouter de nouvelles opérations et fonctions sans modifier les fonctions de notre code puisqu'elles s'appliquent à toutes formes d'opération unaires et binaires. Par exemple $(pow\ e\ n)$ est une opération binaire désignée par

```
BINOP("pow", e, n)
```

2 - Evaluation d'expressions

Pour notre fonction d'evaluation il suffit de reconnaitre le type de notre expression avec un pattern matching, et nous utilisons une fonction auxiliaire nommé trouver_val laquelle cherche la valeur associé à une variable passée en paramètre dans le tableau d'environnement.

3- Dérivation simbolique

Pour la dérivation on utilise des matching pattern pour detecter les différentes opérations binaires et unaires en fonction du label, à partir de ses labels on applique les règles de dérivation usuelles.

4 - Affichage

Grâce à notre choix d'implémentation notre fonction to_string est très courte :

```
let rec to_string (e:exp) : string =
  match e with
  |N(a) -> string_of_int a
  |VAR(a) -> a
  |MONOP(str,e1) -> "("^str^" "^(to_string e1)^")"
  |BINOP(str,e1,e2) -> "("^str^" "^(to_string e1)^" "^(to_string e2)^")";;
```

Ainsi nous detectons les 4 pattern possible et toutes nos fonctions ont le même format d'affichage à savoir : "*label arg1 arg2*" (pour les fonctions à deux arguments) ou encore

"*label arg*" (pour les fonctions à un argument)

5 - Simplification d’expression

Dans notre fonction simpl, nous avons introduit une boucle qui pour une expression donnée la simplifie tant que c’est possible. (On considère qu’une expression est simplifié au maximum si pour la même expression on ne peut pas avoir d’arbre ayant moins de noeud)

Pour simplifié notre arbre d’expression nous faisons un parcours en profondeur.

6 - Fonction trigonométrique

Notre implémentation nous permet d’ajouter très facilement ces fonctions puisque ce sont simplement des opération unaire avec un “label” particulier. Pour modifier les autres fonction en conséquences nous avons simplement besoin de préciser les comportement particulier en fonction du label.

Exemple :

- Pour les dérivées on fait un pattern matching sur le label pour associé la bonne dérivée.
- Pour la simplification de l’exponentielle par exemple on rajoute les cas particuliers de e^0 et la regle $e^x \times e^y = e^{x+y}$ seulement lorsque le label e est detecté par le pattern matching, on procède de manière analogue pour les autres fonctions et leurs propriétés.
- Nos autres fonctions ne dépendent pas du “label” de nos opérations.

7 - Extension libre

Pour nos assertions nous avons fais le choix de vérifier le bon fonctionnement de toutes les cas spéciaux de la simplification (les valeurs remarquables des fonctions que nous avons implémenter) ainsi que leur règle de simplification respective.

Nous notons qu’il est très facile à l’aide de cette implémentation de rajouter des règles de simplifications très élaborée comme les règles de developpement et de réduction trigonométriques.

8 - Variables locales

Nous implémentations nous permet d’ajouter let comme un opération binaire, par exemple

```
(let x = e1 in e2)
```

Est simplement une opération binaire de la forme

```
BINOP("x", e1, e2)
```

Le comportement est similaire de celui de n’importe quel langage de programmation, certains mots clé ne peuvent être utilisé comme nom de variable, ici ce serait les noms de fonctions déjà implémentée. Une variable locale correspondrait ainsi a la non reconnaissance par le matching pattern d’une opération déjà implémentée.

De plus notre choix d’implémentation nous permet de construire des opérations n-aire en faisant une composition de plusieurs expressions.

on a aussi écrit une fonction comp qui compare si 2 arbres d’expressions son égale