ENSEIRB-MATMECA

Projet du semestre 6
Informatique

Derivation

Auteurs :
Alexis Chan
Florian Fichant
Sébastien Gruchet
Florian Le Vern

Responsable :
M. David Janin

Mai 2015

Sommaire

Introduction

L'objet de ce projet est d'implémenter la dérivation symbolique d'expressions mathématiques dans un langage fonctionnel, le `Scheme`. Une expression est définie par la grammaire suivante :

```
expression ::= nombre | variable | (opérateur, liste d'expressions)
```

L'objectif de ce projet est d'afficher la dérivée symbolique d'une expression écrite avec la syntaxe `Scheme`, d'implémenter les techniques de dérivation de base, d'écrire un code modulaire qui permette à l'utilisateur d'ajouter facilement des techniques de dérivation, et d'implémenter des techniques de simplification d'une expression.

Par ordre chronologique, nous nous sommes d'abord intéressés à la façon dont nous allions représenter une expression en `Scheme` avant d'envisager une structure pour stocker et ajouter facilement des techniques de dérivation. Puis nous nous sommes penchés directement sur le problème de l'automatisation de la dérivation d'une expression symbolique, notamment concernant les compositions. Enfin, nous nous sommes concentrés sur les techniques de simplification d'une expression.

1 Analyse du projet et représentation des données

1.1 Définition du type `expr`

Une expression mathématique est définie par la grammaire :
`expression ::= nombre | variable | (opérateur, liste d'expressions)`
Il paraît donc naturel de créer un nouveau type qu'on appellera `expr`. Ce type est défini comme un nombre ou un symbole ou une structure `oper`. Cette dernière contient deux champs, un symbole et une liste d'expressions. Ainsi, l'implémentation correspond complètement à la grammaire.
L'utilisation de la structure `oper` permet d'opérer un 'pattern matching' dans les fonctions pour distinguer l'opérateur de ces arguments. De plus, elle permet de ne pas confondre l'opérateur avec une variable qui sont tous deux représentés par un symbole en langage `Scheme`.

Comme le projet nécessite de faire la distinction entre les nombres, les opérateurs, les variables, les expressions etc., tout le code est écrit en **typed/racket**. Cela implique plus de 'pattern matching' mais il en résulte un code plus rigoureux où chaque variable manipulée possède un type bien connu.

1.2 Les fonctions de traduction

Pour que l'utilisateur puisse écrire en syntaxe `Scheme` pour demander une dérivée au programme, des fonctions de traductions ont été implémentées et se trouvent dans le fichier `Fonctions.rkt`. Ainsi la fonction `trad` permet de passer d'une liste représentant une expression en syntaxe `Scheme` vers une expression de type `expr` qui est exploitable par le programme. Réciproquement, la fonction `trad-inv` permet de passer d'une `expr` vers une liste en syntaxe `Scheme`. Par exemple :

$$\text{'(+ 2 x)} \overset{trad-inv}{\underset{trad}{\longleftrightarrow}} \text{(oper '+ (list 2 'x))}$$

2 Le calcul d'une dérivée

2.1 Les opérations n-aires

Afin d'avoir un programme modulaire, une norme est imposée pour ajouter une règle dans le dictionnaire. En particulier, elle prennent en argument une liste d'au maximum deux expressions. En effet, il faut pouvoir gérer le cas des fonctions n-aires. Ainsi, une expression mathématique donnée par l'utilisateur est préconditionnée grâce à la fonction `precond` avant d'être dérivée. Ce préconditionnement consiste à découper les opérations qui prennent plus de deux arguments en une série d'opérations à deux arguments. Par exemple : `precond` appliquée à l'expression de type `expr` illustrant $3 + 4 + x + 2$ donne $3 + (4 + (x + 2))$.

2.2 La composition

A première vue, pour dériver une expression le problème réside dans la composition. En effet, une expression mathématique est une succession d'opérations qu'il faut dériver dans un ordre précis suivant la règle :

$$(fog)' = g' \times (f'og)$$

Cependant, au lieu de gérer cette règle qui est différente des opérations usuelles (comme $+$, $\times$ etc.), il est possible de déplacer le problème au moment de la définition de la technique de dérivation : elles appellent elles-mêmes la fonction `derivee` ce qui implique un effet récursif. La fonction `derivee` définie dans `Fonctions.rkt` consiste donc simplement à parcourir l'expression mathématique à dériver et à appliquer la règle de dérivation correspondant au premier symbole d'opérateur rencontré.

On notera que la fonction `derivee` ne prends pas de variable en paramètre. En effet, le programme dérive uniquement par rapport à la variable x. Il aurait été possible de choisir la variable par rapport à laquelle on souhaite dériver en ajoutant un argument aux règles de dérivation. Ici le choix s'est porté sur l'allègement de l'écriture des règles de dérivation, mais en contrepartie l'utilisateur doit bien distinguer la variable de dérivation au moment de l'écriture de l'expression.

3 La bibliothèque et les techniques de dérivation

Afin de rendre le programme modulaire, il est nécessaire de pouvoir stocker les différents opérateurs mathématiques et leurs techniques de dérivation associées. Ces informations sont donc insérées dans une structure de données : un dictionnaire `dict`, où à chaque fonction mathématique correspond la façon dont on la dérive.
En `Racket`, le type abstrait de données correspondant à ces attentes se trouve sous la forme de tables de hachages (*hash tables*). Il permet de gérer l'initialisation, l'ajout, le retrait, la comparaison, la lecture, le comptage d'entrées et la mise à jour des entrées déjà présentes.
On notera que les tables de hachages ont pour avantage une complexité en temps faible pour la recherche : de l'ordre de O(1) en moyenne. Cela s'avère intéressant dans le cadre de ce projet puisqu'un appel au dictionnaire sera fait pour chaque opérateur qui apparaît dans l'expression à dériver, et ainsi l'utilisation d'une fonction d'initialisation, de lecture ou de modification aura une influence négligeable sur les algorithmes dans lesquels elles interviennent.
C'est dans le ficher `Fonctions` qu'elles sont présentes, pour permettre l'ajout des fonctions mathématiques et leurs dérivées associées.

Les techniques de dérivation disponibles dans le dictionnaire doivent avoir une norme précise pour garantir la reproductibilité de celles-ci par l'utilisateur. Voici un exemple d'ajout de règle au dictionnaire avec la norme choisie :

```
; dérivée d'une somme
(: d_sum (expr * -> expr))
(define (d_sum . l)
   (oper '+ (list (derivee (car l)) (derivee (cadr l)))))

(dict-add dict '+ d_sum)
```

La règle est une fonction qui prend comme argument une liste d'au maximum deux expressions. Ces expressions sont considérées comme des fonctions dérivables. Pour dériver une de ces expressions, il faut faire appel à la fonction `derivee`. Le type de retour est `expr`, il faut donc s'adapter à cette syntaxe qui est définie dans le fichier `Definition.rkt` et ne pas retourner une expression avec la syntaxe `Scheme` qui aurait été `'(+ (derivee (car l)) (dérivée (cadr l)))`. Enfin, pour ajouter la règle au dictionnaire, il faut faire appel à la fonction `dict-add` en spécifiant le symbole correspondant à la technique de dérivation ajoutée.

On obtient ainsi une norme qui correspond aux spécifications de la fonction `derivee` définie dans le fichier `Fonctions.rkt` et qui permet de se décharger du problème de la composition.

4 Les simplifications

Les simplifications que le programme peut effectuer sont grandement orientées vers les polynômes. Les cas particuliers correspondants aux opérateurs ne sont pas traités.

4.1 Le cas de la différence

Certaines opérations comme $-$, / et `^` sont particulières au sens où elles ne sont pas commutatives : par exemple $2-3 \neq 3-2$. Cela pose parfois problème lorsqu'on veut simplifier une expression en plaçant arbitrairement les arguments dans la liste d'arguments.
Pour le cas de la différence, il est possible contourner le problème. En effet, la différence n'est qu'une somme d'éléments multipliés par -1 sauf le premier terme :

$$x - y - 2 = x + (-1 \times y) + (-2)$$

C'est de quoi s'assure la fonction `diff-somme`, définie dans `Simplification.rkt` de manière récursive dans toutes les sous-expressions.

Les cas de / et `^` n'ont pas été implémentés mais les règles suivantes auraient pu être utilisées pour ce faire :

$$\frac{x}{y} = x \times \frac{1}{y} = x \times y^{-1}$$

$$y^z = e^{z \times ln(y)}$$

4.2 Concaténation

La fonction `conc_s*` permet de concaténer les expressions n-aires ($n \geq 2$) ayant le même opérateur en une seule. Cela est principalement utile pour les opérations $+$ et $\times$. Par exemple : $3 + (4 + x))$ devient $3 + 4 + x$.
La concaténation des expressions s'avère nécessaire à la suite de la dérivation à cause principalement du préconditionnement qui a été appliqué.

On notera que cette fonction n'est pas totalement efficace sur toutes les sous-expressions. Ceci s'explique : pour que cela soit le cas, il faudrait que la récursivité commence de la sous-expression la plus "profonde" jusqu'à l'expression "de profondeur 0" or ni une récursivité simple ni la fonction `map` ne permettent en `Sceme` de faire une telle récursivité inversée.

4.3 Développement

Dans l'objectif de simplifier au maximum les expressions polynomiales, la fonction `developpe` qui se trouve dans le fichier `Simplification.rkt` permet de développer une expression en produit de sommes vers une somme de produits.

4.4 Factorisation

Pour les fonctions commutatives comme $+$ et $\times$, la fonction `factorise` de `Simplification.rkt` permet d'appliquer l'opérateur sur tous les arguments de type nombre et de ne laisser que le reste. Ainsi, $2 + 3 + x + 4$ devient $x + 9$.

Lors de la dérivation, des facteurs neutres ou absorbant pour $+$ ou $\times$ peuvent apparaître. Les facteurs neutres sont supprimés (dans la mesure où ils ne sont pas les seuls arguments de l'opération) et une multiplication par 0 retourne 0. Il peut alors ne rester qu'un seul argument pour une fonction n-aire ($n \geq 2$). Dans ce cas, la fonction `parentheses` permet d'ajouter ce seul élément aux arguments de l'opération de profondeur supérieure.

5 Résultats et perspectives

Chaque fonctionnalité importante a fait l'objet d'une partie test, dans laquelle nous avons pu essayer les différentes simplifications présentées, et voir de possibles améliorations à celles-ci.

5.1 Tests

Certaines parties du projet ont pu être analysées au fur et à mesure de leur avancement étant donné l'utilisation d'un langage fonctionnel. Pour les autres, les tests se sont décomposés en plusieurs parties : Tout d'abord, les tests de la fonction d'affichage en écriture infixe, puis les tests de simplification des expressions, et enfin l'organisation de ses simplifications. Les fonctions réalisées montrent ainsi un comportement adéquat lorsque le cas a été envisagé, cependant la plupart ne gère pas un aspect important des expressions : la récursivité de l'application des règles. Pour certaines, cela fonctionne, mais le problème vient aussi du fait de la necessité d'imposer un ordre dans l'application des règles : il est possible en simplifiant avec une règle de créer une situation simplifiable par une règle déjà utilisée. Exemple :
Lors de la simplification de `(* 2 (+ 5 1) (* 2 0))` , la règle de simplification du produit ne simplifiera que la partie intérieure (`(* 2 0)`) et pas la totalité de l'expression qui devrait être nulle. Ce genre de problème est inhérent à l'implémentation utilisée ici pour les règles.

5.2 Améliorations

En plus du problème discuté sur l'ordre des simplifications, il est possible de trouver plusieurs autres points qui peuvent être améliorées :

Par exemple, ajouter la possibilité de dériver des fonctions à plusieurs variables, ou encore créer une base de donnée pour les simplifications comme pour celle des dérivations pour améliorer leur lisibilité et aussi potentiellement ajouter des règles de simplification de quotients, de logarithmes et exponentielles, ou encore l'ajout de formules trigonométriques par exemple. De plus, il est aussi possible de rajouter une fonction qui calcule si la simplification proposée est pertinente ou non avant de l'appliquer en s'appuyant sur la taille de l'expression finale par exemple.
Une simplification assez pertinente est également la mise sous forme polynômiale lorsqu'elle est pertinente.

Il est possible, avec notre implémentation d'ajouter des règles de dérivation au dictionnaire, cependant l'ajout d'une fonction qui permettrait à l'utilisateur d'ajouter d'autres règles de dérivation de manière plus simple, syntaxiquement par exemple, serait envisageable également.

Conclusion

Ce projet s'est révélé très formateur. En effet, il nous a offert la possibilité de renforcer notre maitrise du langage `Scheme` (dans un cadre plus large et plus complexe que de courtes séances de TD), et nous a permis de découvrir pleinement (ou du moins de mieux nous rendre compte) des possibilités en programmation fonctionnelle par rapport à la programmation impérative, qui représentait jusqu'à maintenant la majorité de notre expérience. En outre, ce projet est par nature différent des projets auxquels nous étions habitués (s'agissant de l'implémentation d'un système de dérivation, et non d'un jeu), et bien que moins captivant au premier abord, il s'est très vite révélé tout aussi intéressant que les autres projets, et d'autant plus inovant qu'il était différent sur tous les points.

En somme ce projet s'est révélé fort intéressant, et très innovant, tout en nous permettant d'expérimenter à loisir avec le langage `Scheme` (notamment par le fait qu'il était toujours améliorable, et donc toujours source d'expérimentation en ce but).