T.I.P.E. : Calcul formel en Caml

Quentin Cormier

Table des matières

1 Introduction

Les logiciels de calcul formel comme Maple et Mathematica sont très efficaces pour effectuer des calculs symboliques complexes, comme le calcul de primitives ou la factorisation d'une expression.

Maple et Mathematica offrent un langage de programmation haut niveau pour pouvoir utiliser leurs nombreuses fonctions mathématiques.

Mais ces langages ne sont pas aussi expressifs et sûrs que les langages fonctionnels comme Haskell ou OCaml : le typage statique et l'inférence de type, le filtrage, les fonctions d'ordre supérieur, les types construits rendent ainsi l'écriture de programmes plus agréable en Caml qu'en Mathematica.

Cependant, Caml et Haskell ne permettent pas de faire du calcul symbolique. Le but de ce T.I.P.E. est de concevoir un petit langage de programmation, appelé CamlM, basé sur Caml, et enrichi de quelques fonctionnalités permettant de faire du calcul formel.

Après avoir détaillé les différentes fonctionnalités du langage, on montrera des exemples effectifs de programmes réalisés dans ce langage, comme la dérivation d'une expression mathématique.

On présentera ensuite l'interpréteur de CamlM, écrit en Ocaml.

2 Présentation générale du langage

CamlM supporte les fonctionnalités de base de Caml, à savoir les fonctions récursives, le filtrage, l'inférence de type, l'application partielle, etc. La syntaxe de CamlM est, à quelques détails près, identique à celle de Caml.

Les fonctions anonymes sont notées `\x → f x` en CamlM (notation comme en Haskell) et non `fun x → f x` comme en Caml.

On représente une expression mathématique à l'aide d'une fonction. Par exemple, l'expression $5x^2 + 8x + 1$ est représentée en CamlM par la syntaxe : `\x → 5*x^2+8*x+1`.

Pour pouvoir manipuler des expressions mathématiques, CamlM accorde une importance particulière à certaines fonctions, appelées *fonctions formelles*.

2.1 Fonctions formelles

Dans Ocaml et dans CamlM, les fonctions sont définies par une liste `l = [(p1, e1); ...; (pn, en)]`. Chaque élément de l est un couple dont la première composante est un motif et la deuxième composante est une expression associée à ce motif.

Par exemple, la fonction suivante de Caml est représentée par la liste `[(0, 1); (Variable "n", 0)]` :

```
let f = function
  | 0 -> 1
  | n -> 0
```

Dans CamlM, on définit un sous ensemble des ces fonctions, appelé "fonctions formelles" (qui correspond aux fonctions "simples") de la manière suivante :

Définition : Si `f = [(p1, e1); ...; (pn, en)]` est une fonction, f est une fonction formelle si elle vérifie le prédicat `P` défini par induction structurel de la manière suivante :

Pour toute expression du langage expr, si expr est :
- Une fonction non récursive de la forme `[(Variable "x", expr')]` alors `P(expr) <=> P(expr')`
- L'application de la fonction `g` et de l'argument `expr'`, alors `P(expr) <=> [ P(g) et P(expr') ]`
- Un élément terminal comme un nombre, un booléen, une chaine de caractères, etc. alors `P(expr) = vrai`
- Un couple de la forme `(e1, e2)` ou une liste de la forme `e1::e2`, alors `P(expr) <=> P(e1) && P(e2)`
- Un élément de la forme `Some e`, alors `P(expr) <=> P(e)` (et pour l'élément terminal `None`, `P(expr) = vrai`)
- Tout autre forme d'expression alors `P(expr) = false`

Aussi, les fonctions suivantes sont des fonctions formelles :
- `\x → x+1`
- `\x → (\y → y+1) (x+1)`
- `\x → (\y → x)`

Et celles-ci n'en sont pas :
- `\x → let y = 5 in x+y`
- `\(x, y) → x+y`
- `let rec f x = x in f`

Ces fonctions formelles sont "ouvertes" en CamlM : il est possible d'explorer leur définition à l'aide du filtrage de motif.

2.2 Filtrage sur les fonctions formelles

La manipulation de ces fonctions est rendue possible par l'ajout de fonctionnalités dans le filtrage de motifs. L'ajout des opérateurs `+`, `*`, `/`, `^` et `.` permet de déterminer si une fonction formelle est la somme, le produit, le rapport, l'exponentiation ou la composée de deux fonctions formelles.

La syntaxe est la suivante, elle ne diffère pas du filtrage habituel :

```
function
    f + g -> [...]
  | f * g -> [...]
  | [...]
```

Exemple :

```
# let dec_sum = function
    f+g -> Some (f, g) (* si une fonction est la somme de deux
       fonctions f et g, on renvoit le couple (f, g) *)
  | _ -> None
;;
:- (('a -> int) -> (('a -> int) * ('a -> int)) option) = ()

# dec_sum (\x -> 5*x+1);;
:- ((int -> int) * (int -> int)) option = Some (\x -> (5 * x), \x ->
    1)
(* \x -> 5*x+1 est bien la somme formelle des fonctions \x -> 5*x et
    \x -> 1 *)
# dec_sum (\x -> 5*x);;
:- ((int -> int) * (int -> int)) option = None
```

Il est à noter la différence de signification au niveau du typage de ces opérateurs si l'on se situe dans un filtrage ou dans une expression.
Dans un filtrage `f + g` signifie "une fonction formelle se décomposant en somme de deux fonctions formelles f et g" de type `num → num`. Dans une expression `x+y` signifie "le nombre `x+y`" de type `num`.

On rajoute également dans le filtrage de manière analogue les mots clés `Id`, `Const f` et `Num x` qui permettent respectivement de savoir si une expression est la fonction identité, une fonction constante (indépendante de sa variable), ou un nombre.

2.3 Réduction des fonctions formelles

Les fonctions formelles subissent deux traitements pendant l'évaluation : le remplacement de ses variables libres et une réduction normale.

2.3.1 Remplacement des variables libres

Les variables intervenant dans une fonction formelle sont remplacées par leur expression. Pour chaque fonction formelle `\x → expr` on recherche dans `expr` les variables libres et on les substitue par leur expression.

Par exemple :

```
let g x = x+1 in
\x -> g (5*x)
```

On obtient après ce traitement : `\x → (\x → x+1) (5 * x)`.

On a substitué dans la dernière fonction la variable g par son expression.

2.3.2 Réduction normale de type "béta reduction"

Il s'agit d'une réduction visant à réduire quand cela est possible l'application d'une fonction avec un argument. Aussi la fonction `\x → (\x → x+1) (5 * x)` est réduite après ce traitement en `\x → ((5 * x) + 1)`.

Les règles de substitution et de réduction sont analogues aux règles issues du λ-calcul (voir 4.5).

2.4 Axiomes

Il peut être utile de définir un objet sans lui attribuer d'expression explicite. Par exemple, pour définir le nombre π, on ne souhaite pas donner une valeur particulière à π comme 3.14.

CamlM dispose d'une syntaxe pour définir de tels objets, la syntaxe `declare`.

Ainsi, pour définir le nombre π, ou la fonction `exp`, on peut écrire :

```
declare pi;;
declare exp;;
```

On peut alors utiliser pi comme un nombre comme les autres.

```
# 5*pi+1;;
:- num = ((5 * pi) + 1)
```

Il est possible de savoir si une expression est explicitement un axiome à l'aide de la syntaxe `@` ajoutée dans le système de filtrage de motifs : `@var → ...` est du sucre syntaxique équivalent à `v when v == var → -> ...`.

Exemple :

```
# declare exp;;
# declare ln;;
# let evalue_en_1 = function @ln -> 0 | @exp -> exp 1 | g -> g 1;;

# evalue_en_1 ln;; (* l'axiome ln est bien reconnu *)
# :- int = 0
# evalue_en_1 exp;; (* l'axiome exp est bien reconnu *)
:- int = exp 1
# evalue_en_1 (\x -> 5*x);; (* les deux axiomes ln et exp ne sont
   pas reconnus *)
:- int = 5
```

Lorsque l'on n'utilise pas l'arobase, le comportement du filtrage est analogue à celui d'Ocaml.

```
# let evalue_en_1 = function ln -> 0 | g -> g 1;;
:- ((int -> int) -> int) = ()
# evalue_en_1 (\x -> x);;
:- int = 0
```

3 Exemples de programmes écrits en CamlM

Toutes ces fonctionnalités permettent d'écrire des fonctions pour dériver, développer, simplifier,... une expression mathématique en CamlM.

3.1 Dérivation d'une fonction

On commence par déclarer quelques fonctions usuelles :

```
declare exp;;
declare ln;;
declare sqrt;;
```

On définit alors récursivement la dérivée d'une fonction en implémentant les règles de base des dérivées :

```
let rec deriv = function
    @exp -> exp (* exp' = exp *)
  | @ln -> (\x -> 1/x) (* ln' = 1/x *)
  | @sqrt -> (\x -> 1/(2*sqrt x))
  | Id -> \x -> 1
  | Const _ -> \x -> 0
  | f+g -> (* (f+g)' = f'+g' *)
    let df = deriv f in (* on calcule f' *)
    let dg = deriv g in (* on calcule g' *)
     \x -> df x + dg x (* on renvoie la fonction f'+g' *)
  | -f ->
    let df = deriv f in
    \x -> - df x
  | f*g -> (* (fg)' = f'g + g'f *)
    let df = deriv f in let dg = deriv g in \x -> df x * g x + dg x
       * f x
  | f / g -> (* (f/g)' = (f'g - g'f) / g^2 *)
    let df = deriv f in let dg = deriv g in \x -> (df x * g x - dg x
       * f x)/(g x * g x)
  | f . g -> (* (f o g)' = f' o g * g'  *)
    let df = deriv f in
    let dg = deriv g in
    \x -> df (g x) * (dg x)
;;
```

Exemple :

```
# deriv (\x -> x*ln x);;
:- (num -> num) = \x -> ((1 * ln x) + (((1 / x) * 1) * x))
```

3.2 Développer une expression

Le filtrage sur les fonctions formelles permet de manipuler finement une expression. On peut ainsi facilement développer une expression :

```
let rec expand = function
    a*(b+c) ->
    let f = expand (\x -> (a x)*(b x)) in
    let g = expand (\x -> (a x)*(c x)) in
    \x -> f x + g x
  | f+g ->
    let f' = expand f in
    let g' = expand g in
    \x -> f' x + g' x
  | f.g ->
    let f' = expand f in
    let g' = expand g in
    \x -> f' (g' x)
  | f -> f
;;
```

Exemple :

```
# expand (\x -> 5*x*(ln x + x));;
:- (num -> num) = \x -> (((5 * x) * ln x) + ((5 * x) * x))
```

3.3 Simplification d'une expression

La simplification d'une expression est plus délicate. j'ai écrit une fonction qui regroupe d'abord les termes d'une somme en paquet du même type. Aussi, $5x + 7 + ln(x) + 7x + 13$ se regroupe en $(7 + 13) + (5x + 7x) + (ln(x))$. On peut alors simplifier facilement les termes d'un même paquet. Le code est un peu plus long (150 lignes de code CamlM) et fut un bon moyen de tester l'interpréteur. On pourra le trouver ici : `https://github.com/robocop/CamlM/blob/master/lib/maths.mml`

Exemples :

```
# simplify (\x -> 5*x+7+ln x+7*x+13);;
:- (num -> num) = \x -> ((12 * x) + (20 + ln x))
# deriv (\x -> x*ln x - x);;
:- (num -> num) = \x -> (((1 * ln x) + (((1 / x) * 1) * x)) + - 1)
# simplify (deriv (\x -> x*ln x - x));;
:- (num -> num) = \x -> ln x
```

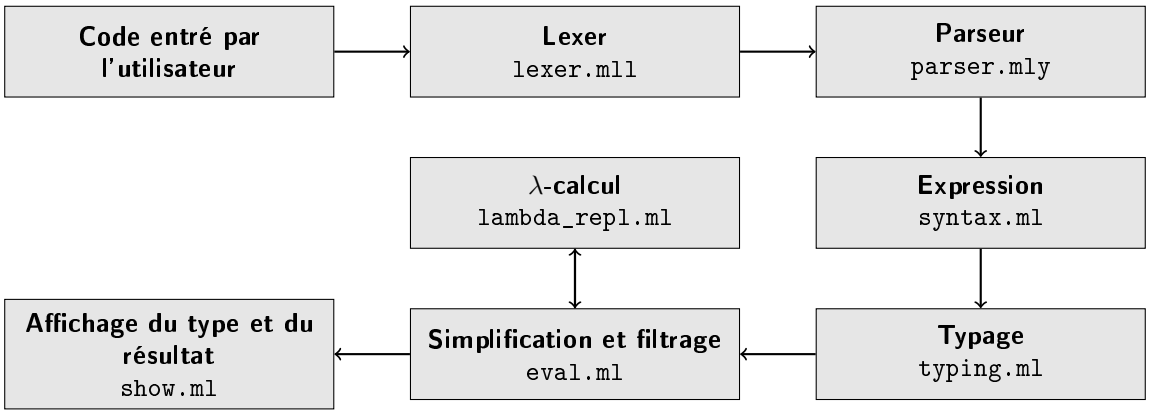
4 Structure de l'interpréteur

L'interpréteur de CamlM est écrit en Ocaml. Il se présente comme une invite de commande (de la même façon que l'interpréteur Ocaml).

Pour l'installer, il suffit de récupérer les sources sur github, et de compiler le projet :

- Faire une copie du dépôt : `git clone git://github.com/robocop/CamlM.git`
- Compiler en exécutant la commande : `./configure && make`
- Lancer l'interpréteur avec `./camlm`

Le code de l'interpréteur est organisé en plusieurs fichiers qui correspondent chacun à une tache précise, organisation que l'on peut résumer avec le schéma suivant :



Une documentation précise de chacun des fichiers pourra être trouvée ici `http://robocop.github.io/CamlM/doc/`.

La structure de l'interpréteur est très proche de celle décrite dans le livre "Le Langage Caml" ([LLC]) de Pierre Weis et Xavier Leroy, chapitre 17 (Exécution d'un langage fonctionnel) et chapitre 18 (Un synthétiseur de type) : ces deux chapitres m'ont servis de base que j'ai peu à peu modifié pour écrire l'interpréteur de CamlM.

4.1 Analyse lexicale et syntaxique

L'analyse lexicale (découpe du code en mots) et syntaxique (construction de l'A.S.T.) est réalisée à l'aide de menhir, dans les fichiers `lexer.mll` et `parser.mly`. La difficulté fut d'éviter les erreurs de type "shift/reduce conflict" qui traduisent une syntaxe ambiguë.

4.2 A.S.T. : représentation interne du code CamlM

Le code CamlM est représenté par un arbre dont le type explicité dans le fichier `syntax.ml`. Il s'agit du type `expression` définit de la sorte :

```
type expression =
|    EVariable of string
|    EFunction of closure
|    EApplication of expression * expression
|    ELet of definition * expression option
|    EDeclare of string * expression option
|    EBoolean of bool
|    ENum of int32
|    EPair of expression * expression
|    EUnit
|    ENil
|    ECons of expression * expression
|    ENone
|    EString of string
|    ESome of expression
```

Une fonction (`EFunction`) est donc définie par sa fermeture :

```
type closure = {
  def :(pattern * expression) list;
  mutable env :fun_env_content env option;
  }
```

Une fermeture, c'est la donnée d'un environnement, c'est à dire la liste des variables que connaît la fonction au moment où on la définit, et d'une liste de couples `motif * expression` qui définissent la fonction (comme expliqué en 2.1).

Le type `pattern` décrit les différents motifs possibles dans un filtrage :

```
type pattern =
|   PAll (* correspond a  (n'importe quelle expression) *)
|   PVariable of string
|   PAxiom of string (* syntaxe '@' *)
|   PBoolean of bool
|   PNum of int32
|   PPair of pattern * pattern
|   PNil (* motif de la liste vide *)
|   PCons of pattern * pattern
|   PNone
|   PSome of pattern
|   PString of string
|   POp of string * pattern * pattern (* motif de la somme de deux
   fonctions formelles f + g *)
|   PMinus of pattern
|   PCompose of pattern * pattern
|   PIdentity (* motif qui teste si une fonction est la fonction
   formelle identite *)
|   PConst of pattern
|   PIsnum of pattern
|   PWhen of expression * pattern
```

Enfin, le type `definition` permet de gérer la définition d'une variable (avec la syntaxe `let`), qui peut être récursive ou non.

```
type definition = {
    recursive :bool;
    name :string;
    expr :expression;
}
```

4.3 Typage

Une fois le code de l'utilisateur parsé et transformé en arbre de type `expression`, celui-ci est typé à l'aide du fichier `typing.ml`. Le typeur est exactement celui décrit par Pierre Weis et Xavier Leroy dans le chapitre 18 de [LLC]. Je me suis contenté d'implémenter sans aucune modification ou presque l'algorithme présenté.

4.4 Simplification d'une expression CamlM

Il s'agit de la partie centrale de l'interpréteur (fichier `eval.ml`). Dans l'interpréteur présenté dans [LLC], l'évaluation d'une expression consiste à prendre un code Caml (de type `expression`), et à l'évaluer entièrement, en renvoyant un résultat final simple, de type `valeur` ($\neq$ `expression`), c'est à dire soit la valeur d'un entier, soit la valeur d'un booléen, d'une paire, etc.

Ici, la logique est un peu différente, il ne s'agit pas vraiment d'une évaluation, mais d'une simplification : on prend en entré une expression CamlM, de type `expression`, et on renvoie en sortie une expression CamlM équivalente, simplifiée, de type `expression` également.

Les règles de simplifications sont les suivantes, si l'expression est

- Une variable : on va chercher la valeur de la variable dans l'environnement en cours, si elle n'existe pas on renvoie une erreur
- Une fonction : on teste si la fonction est une fonction formelle, si oui on la simplifie (cf 4.5), sinon on la laisse telle quelle.
- L'application d'une fonction à un argument : si la fonction est une primitive du langage (comme l'addition de deux entiers naturels par exemple), on évalue directement le résultat, sinon on regarde si la fonction est dans environnement. Si c'est le cas, on cherche sa fermeture, et on l'évalue à l'aide de la fonction `eval_application` en l'argument. `eval_application` teste chacun des motifs qui définit la fonction avec l'argument (à l'aide de la fonction `matching`). Dès qu'un motif correspond, on évalue l'expression associée.
- Une déclaration de variable (`let v = expr in r`) : on évalue la variable v avec l'environnement en cours, et on ajoute v et sa valeur à l'environnement, en distinguant si v est défini récursivement ou non.

Les règles sont donc identiques à celles de Caml, sauf pour les fonctions formelles que l'on simplifie en utilisant des règles de λ-calcul.

4.5 Simplification d'une fonction formelle (λ-calcul)

La manipulation des fonctions formelles à lieu dans le fichier `lambda_repl.ml`. Une fonction permet de tester si une fonction est une fonction formelle. Si c'est le cas, on applique d'abord une règle de remplacement, qui consiste à substituer les variables libres intervenant dans l'expression de la fonction formelle par leurs expressions. Puis on applique l'algorithme "Normal Order Reduct" pour simplifier la fonction finale.

On interprète une fonction formelle comme une expression de λ-calcul, de la forme suivante :

```
x est une variable
M ::= x | λx.M | M M
```

Exemple : λx.(λy.x) x correspond à la fonction CamlM `\x -> (\y -> x) x`.

Calcul des variables libres

```
FV(x) = {x}
FV(λx.M) = FV(M) − {x}
FV(M N) = FV(M) ∪ FV(N)
```

Substitution

On note `N[M/x]` : substitution de la variable x par le terme M dans N. On a alors les règles suivantes :

```
x[M/x] = M
y[M/x] = y si x ≠ y
(λx.N')[M/x] = λx.N'
(λy.N')[M/x] = λz. ((N'[z/y]) [M/x]) si x ≠ y et z ∉ FV(M) ∪ FV(N)
    ∪ {x}
(N1 N2)[M/x] = N1[M/x] N2 [M/x]
```

***β*-réduction et Normal Order Reduction**

Permet de simplifier un terme de λ-calcul.

```
(λx.M) N → M[N/x]
Si M → M' alors λx.M → λx.M'
Si M → M' alors M N → M' N
Si N → N' alors M N → M N'
```

On applique cette simplification autant que possible. Ces règles sont implémentées dans le fichier `lambda_repl.ml` et rendent donc effectives la simplification des fonctions formelles : on peut dire en ce sens que CamlM est un interpréteur de λ-calcul.

Ces simplifications amènent parfois à introduire de nouvelles variables. Exemple :

```
# declare a;;
# let g x = a+x;;
# \a -> g(a+1);;
:- (num -> num) = \b -> (a + (b + 1))
```

CamlM introduit la variable `b` pour simplifier correctement la fonction formelle.

4.6 Affichage du résultat

Enfin, le fichier `show.ml` s'occupe de l'affichage de l'expression ainsi simplifiée.

5 Conclusion

On a ainsi vu qu'en rajoutant quelques fonctionnalités à Caml, on peut effectuer du calcul formel dans un langage fonctionnel.

On a accordé une importance particulière à certaines fonctions, les fonctions formelles, et on peut se demander pourquoi s'imposer une telle limitation. Après tout, on peut sans difficulté étendre les règles λ-calcul à l'ensemble de Caml, et voir le problème de la simplification d'une expression mathématique comme un problème de réduction de λ-calcul étendu. Plutôt que d'ajouter des fonctionnalités de filtrage sur les fonctions comme on le fait actuellement (`f+g -> ...` permet de tester si une fonction est la somme de deux fonctions), on peut ajouter des fonctionnalités de filtrage sur les expressions : `a+Int 5 -> ...` permettrait alors de tester si une expression est de la forme `a+5`, avec `a` une expression quelconque.

C'est le travail que j'ai commencé à réaliser (voir la branche *new_eval* du projet sur github). On s'approche alors des fonctionnalités de méta-programmation de lisp. En lisp on peut programmer la dérivation d'une expression mathématique naturellement à l'aide des quotations [1].

1. Voir [SICP], chapitre Symbolic Differentiation

Cependant, il semble alors que le langage devienne trop expressif pour le typeur (typer du lisp, ce n'est pas facile), et j'ai du renoncer à cette version. On comprend alors que les fonctions formelles sont un bon compromis pour pouvoir faire du calcul symbolique tout en continuant d'avoir un système d'inférence de type statique comme en Caml.

Remerciements : Charlie Paucard (`cgizmo`, étudiant à Imperial College) pour la réalisation du système de Module dans CamlM analogue à celui dans OCaml, et Gabriel Scherer (`gashe`, doctorant à l'INRIA) pour ses précieux conseils.

Références

[LLC] Pierre Weis, Xavier Leroy, *Le langage Caml*, 1999.

[1] Philippa Gardner Cours de λ-calcul de Imperial College, disponible sur internet.

[SICP] Harold Abelson, Gerald Jay Sussman, Julie Sussman *Structure and Interpretation of Computer Programs*, MIT Press 1996.