INPT - ENSEEIHT
Département IMA

Anthony FOULFOIN
1ère année
Groupe B

mardi 16 décembre 2008

# Projet CamL
# Calcul de type

# SOMMAIRE

1 Présentation générale ..... 3
1.1 Présentation de MiniML ..... 3
1.2 But du projet ..... 3
1.3 Distribution fournie ..... 3
1.4 Spécifications ..... 4
2 Conception et programmation ..... 5
2.1 Typage ..... 5
2.1.1 La fonction *typer* ..... 5
2.1.2 Les règles d'inférence ..... 6
2.2 Résolution du système d'équation ..... 6
2.2.1 La fonction *résoudre* ..... 6
2.2.2 La fonction *remplacer* ..... 8
2.2.3 La fonction *substituer* ..... 8
3 Tests ..... 9
3.1 Tests unitaires ..... 9
3.2 Tests globaux ..... 10
3.2.1 Jeu de test global ..... 10
4 Conclusion ..... 13

# 1 Présentation générale

## 1.1 Présentation de MiniML

CamL est un langage de programmation généraliste fortement typé se prêtant à des paradigmes de programmation fonctionelle, impérative et orientée objet. Le typage des expressions en CamL est effectué par inférence, autrement dit, le développeur n'a pas à typer ses variables, le compilateur est effectivement capable de détecter leur type de façon non ambigüe grâce au contexte d'utilisation.

MiniML est un langage de programmation fonctionnelle, se présentant comme un sous-ensemble du langage CamL. Celui-ci bénéficie d'une grammaire simplifiée, mais conserve le même principe de fonctionnement.

Tout comme en CamL, MiniMl utilise l'inférence de type afin de déterminer le type d'une expression.

## 1.2 But du projet

Le but du projet est de développer un programme réalisé en CamL capable de déterminer le type d'une expression programmée en MiniMl.

Afin de pouvoir mener à bien ce projet, une partie du programme nous a été fournie, afin que nous puissions nous concentrer uniquement sur le calcul de type.

Ainsi, le projet consiste à programmer 2 fonctions :

- La fonction *typer*, qui produit par inférence le type de l'expression ainsi qu'un ensemble d'équations sur les types introduits.

- La fonction *résoudre*, qui, à partir de l'ensemble d'équations, permet de vérifier si l'expression peut être typée, et de construire son type polymorphe le plus général.

## 1.3 Distribution fournie

Les fichiers nécessaires à la réalisation du projet nous ont été fournis dans une archive se composant de plusieurs modules :

– Le module Asyn qui contient le type expr qui est la description de la syntaxe abstraite (représentation interne) des programmes en MiniML.

– Le module Miniml qui contient la fonction read_expr() : unit -> expr qui lit sur le flot d'entréee (clavier), une expression MiniML se terminant par « ;; » et renvoie sa représentation interne.

– Le module Type_ml qui contient le type ct définissant la syntaxe abstraite des types. Dans cette représentation les variables de type sont simplement numérotées sous la forme Var n.

La fonction string of type : ct -> string, renvoie une chaîne de caractères correspondant à l'écriture habituelle d'un type en Caml.

– L'interface typeur.mli du module Typeur, et un canevas typeur.ml de sa réalisation contenant les deux fonctions *typer* et *résoudre* qui sont à réaliser.

– Un makefile permettant de compiler le résultat.

– Un fichier d'exemples de fonctions MiniML à typer : exemples.miniml.

## 1.4 Spécifications

Plusieurs contraintes ont été imposées dans la réalisation du projet :

- Seul le fichier typeur.ml doit être modifié

- Le nom et le type des fonctions *typer* et *resoudre* doivent être tels qu'ils sont décrits dans le fichier typeur.mli, afin que les fonctions puissent être testés de manière automatique.

- L'algorithme doit être réalisé dans un paradigme de programmation fonctionnelle uniquement.

- Les solutions apportées devront être motivées par une exigence de clarté et de concision en priorité par rapport à un souci d'optimisation.

- Le code source devra être correctement commenté et indenté.

- Les fichiers rendus doivent pouvoir être compilés, même si le travail n'est pas terminé.

# 2 Conception et programmation

La conception du programme c'est organisée autour des deux principales fonctions à programmer, c'est-à-dire la fonction *typer* et la fonction *resoudre*.

Ainsi nous allons voir le travail de conception qui à été réalisé autour de chacune de ces fonctions.

## 2.1 Typage

### 2.1.1 La fonction *typer*

**Paramètres** : Cette fonction prend en paramètres trois éléments :

- Une expression provenant de la fonction *read_expr*(), représentant l'expression à typer sous la forme d'un arbre.

- Un environnement contenant l'ensemble des variables pouvant être utilisées au sein de l'expression. Cet environnement se présente sous la forme d'une liste de couples où chaque couple est constitué d'un nom de variable et du type qui lui est associé.

- Un système d'équations.

**Principe** : La fonction *typer* forme la base du système de typage. C'est elle qui est invoquée lorsque l'on souhaite typer une expression.

Celle-ci retourne le type de l'expression, ainsi qu'un système d'équations permettant de calculer le type le plus général de l'expression. Ce résultat se présente donc sous la forme d'un couple.

**Exemple** : typer *(Let ("x", Cons (Entier 1, Nil), Ident "x")) [] []* aura pour résultat *(List Int, [(List (Var 37), List Int)]).*
Le premier élément présente le type global de l'expression, le second membre le système d'équations. Chaque couple du système d'équations représente une égalité. Parfois, le type global ne peut être déterminé directement par la fonction *typer*, mais peut être déterminé à partir du système d'équations en effectuant des substitutions, ce qui sera le rôle de la fonction *résoudre*.

**Fonctionnement général** : L'expression reçue en paramètre est en fait un arbre dont les nœuds décrivent les différentes structures du programme.
*Typer* va donc filtrer l'expression pour observer quel est le type de son nœud (i.e. sa structure). Une fois le type du nœud détecté, *typer* applique la règle

d'inférence qui lui est associée. Ces règles étant représentées par des fonctions, *typer* invoque donc la fonction correspondante.

### 2.1.2 Les règles d'inférence

Afin de pouvoir typer l'expression, l'algorithme utilise des règles d'inférence. Il y a une règle d'inférence pour chaque construction du langage MiniML (par exemple une operation binaire entre deux entiers, la concaténation d'un élément à une liste…).

Il a été choisi de représenter ces règles d'inférence sous la forme de fonctions afin d'éclaircir le fonctionnement de la fonction typer, et afin de factoriser plusieurs règles d'inférence différentes en une seule et même fonction.

Chaque fonction retourne ainsi un couple formé du type de l'expression et d'un système d'équations. Ces fonctions peuvent être amenées à invoquer récursivement la fonction *typer*. Nous assistons ainsi à une récursivité mutuelle entre typer et les règles d'inférence.

Ainsi, la fonction typer détecte quelle est la règle d'inférence à appliquer pour une expression, et invoque la méthode correspondante qui traite alors le sous-problème.

## 2.2 Résolution du système d'équation

### 2.2.1 La fonction *résoudre*

**Paramètres** : Un type t représentant le type global de l'expression et un système d'équations eq permettant de calculer le type global le plus général, ou de lancer une exception si le système conduit à un échec. Ces 2 paramètres sont fournis par la fonction *typer*.

**Principe** : Le résultat d'un jugement de typage d'une expression MiniML complète par la fonction *typer* se présente sous la forme d'un couple (τ, eq).
Le type τ contient des variables qui sont également présentes dans le système d'équations eq.

Ce système est une liste d'équations qui sont
1. soit de la forme (1) : α = β, ou α = τ, ou τ = α,
2. soit de la forme (2) : τ = τ',
avec α et β variables de types et $\tau, \tau' \in \{\text{Int}, \text{Bool}, \tau_1 \rightarrow \tau_2, \tau_1 \text{ list}\}$.

La résolution du système consiste à éliminer une à une les équations de la manière suivante :

– Une équation de la forme (1), est éliminée en remplaçant (substitution) la variable α par sa valeur dans les autres équations du système.

– Pour les équations du type (2) on élimine Int = Int et Bool = Bool, on remplace $\tau_1 \rightarrow \tau_2 = \tau_3 \rightarrow \tau_4$ par $\tau_1 = \tau_3$ et $\tau_2 = \tau_4$, et on remplace $\tau_1$ list = $\tau_2$ list par $\tau_1 = \tau_2$.

– Dans tous les autres cas (Int = Bool, Int = τ list, τ list = $\tau_1 \rightarrow \tau_2$, . . .) le système n'a pas de solution.

– Le remplacement des variables dans le système d'équations (substitution) doit être fait simultanément sur le type global de l'expression à typer.

Le remplacement des variables dans le système d'équations est effectué par une fonction tierce : la fonction *remplacer*, et le remplacement des variables sur le type global de l'expression est assuré par la fonction *substituer*.

**Fonctionnement général** : La fonction *résoudre* va parcourir la liste eq représentant le système d'équations. Celle-ci va séparer la tête et la queue de la liste. La tête représente alors un couple de deux éléments. La fonction *remplacer* détecte alors quelle est la « forme » du couple (α = β, ou α = τ , ou τ = α, ou τ = τ'), et effectue si besoin une substitution.

- Si une substitution doit être réalisée, alors celle-ci est faite simultanément sur le type global de l'expression ainsi que sur le reste (la queue) du système d'équations. La substitution sur le type global est assurée par la fonction *substituer*. La substitution sur le reste du système d'équations est assurée par la fonction *remplacer*. On appelle alors récursivement *résoudre* sur le type global et le système d'équations, une fois la substitution effectuée sur ceux-ci. N'effectuant la substitution que sur la queue de la liste, l'équation en tête a donc disparue, le système d'équation va ainsi diminuer petit à petit.

- Si aucune substitution n'est à pratiquer, on appelle récursivement *résoudre* sur la queue de la liste (en ayant éventuellement ajouté de nouvelles équations à la queue, comme précisé dans le principe).

- Dans tous les cas où l'équation est de la forme 2, avec des types incompatibles, la fonction lance une exception.

**Cas terminal** :

- Lorsque le système d'équations est vide, alors la fonction renvoie le type global t.

### 2.2.2 La fonction *remplacer*

**Paramètres** : Cette fonction reçoit en paramètres deux types x et y ainsi qu'un système d'équations eq.

**Principe** : Cette fonction a pour but de remplacer toutes les occurrences de x par y dans l'ensemble des équations eq.

**Exemple** : *remplacer (Var 33) Int [(F(Var 33, Int), Var 34) ;(Int,Int)]* aura pour résultat : *[(F(Int, Int), Var 34) ;(Int,Int)].*

**Fonctionnement général** : La fonction *remplacer* va parcourir la liste eq représentant le système d'équations. Celle-ci va séparer la tête et la queue de la liste. La tête représente alors un couple de deux éléments. Pour chacun de ces deux éléments, *remplacer* va invoquer la fonction *substituer* (avec pour paramètres x,y et l'éléments en question). Le nouveau couple obtenu, dont les occurrences de x ont été remplacées par y, est concaténé au résultat de la fonction *remplacer* sur la queue de la liste.

**Cas terminal** :

- lorsque toute la liste est vide, le résultat retourné est une liste vide.

### 2.2.3 La fonction *substituer*

**Paramètres** : Cette fonction reçoit en paramètre trois types x, y et t.

**Principe** : Cette fonction a pour but de remplacer toutes les occurrences de x par y dans le type t.

**Exemple** : *substituer (Var 33) Int (F(Int,List Var 33))* aura pour résultat : *F(Int,List Int).*

**Fonctionnement général** : Cette fonction parcourt récursivement le type t afin d'y remplacer toutes les occurrences de x par y.
Ainsi, si nous avons t = F(a,b), la fonction substituera x par y dans a et dans b. La fonction effectue donc l'opération *F(substituer x y a, substituer x y b).*
Le principe est le même pour les listes.

**Cas terminaux** :

- lorsque t est égal a x alors la fonction renvoie y.
- lorsque t est de la forme Var a, avec Var a ≠ x, alors la fonction renvoie t (on ne peut plus parcourir le type de manière récursive).

# 3 Tests

Afin de s'assurer du bon fonctionnement du programme dans son ensemble, il est nécessaire de le tester avec des arguments significatifs. Des tests unitaires ont été effectués pour chaque fonction, ainsi que des tests de l'application dans son ensemble.

## 3.1 Tests unitaires

L'algorithme étant composé d'un ensemble de fonctions, il est tout d'abord impératif de s'assurer que chacune des fonctions déclarée fonctionne correctement.

Ainsi, des tests unitaires ont été réalisés pour chacune des fonctions décrites dans la partie 2. Ces tests peuvent être consultés dans le code source du programme.

Les jeux de tests se doivent d'être le plus significatif possible. Chaque jeu répond ainsi à plusieurs impératifs :

- Toutes les instructions de la fonction testée doivent être exécutées. Ainsi, si une fonction réalise un match, le jeu de test associé doit s'assurer que toutes les valeurs filtrées soient représentées. Si une fonction contient des structures conditionnelles (if then else), le jeu de test doit s'assurer que chaque bloc d'instruction de la condition est exécuté au moins une fois (le then et le else) par le jeu de tests.

- Prise en compte de la récursivité de la fonction. Si une fonction est récursive, alors il est nécessaire de s'assurer que cette récursivité fonctionne correctement. Par exemple, pour la fonction *substituer*, le test *substituer (Var 4) (Var 5) (List (List (List (List (Var 4)))))* s'assure que la substitution est correctement effectuée (de manière récursive) sur le type List. Le même principe est utilisé pour les fonctions (représentés par la structure F(a,b)).

En respectant ces impératifs, les jeux de tests s'assurent ainsi de couvrir un grand nombre de possibilités.

Il est à noter que les fonctions représentant les règles d'inférence ne possèdent pas de tests unitaires, de même pour la fonction *getFirstLeft* (fonction considérée comme élémentaire qui renvoie le type de la première occurrence d'une variable dans un environnement). Leur test étant effectué via les tests unitaires de la fonction *typer*.

## 3.2 Tests globaux

Une fois les tests unitaires passés avec succès, il est important de tester le programme dans sa globalité.

Le jeu de test global s'assure ainsi de représenter toutes les structures (et quelques cas générant des erreurs) possibles (ou du moins toutes les règles d'inférence) afin de respecter le premier impératif de la partie 3.1. On s'assure ainsi de couvrir un nombre maximum de lignes d'instructions.

De plus certains cas « intéressants » ont été ajoutés au jeu de test, comme par exemple la fonction *let rec test = (function n -> if (hd n = 2) then 0 else 1) in test* qui vérifie si, connaissant le type associé à la tête d'une liste, le programme est capable de déterminer le type de la liste.

### 3.2.1 Jeu de test global

**Test utilisant les structures Let, Entier, et Ident :**
*?let x=1 in x;;*
*int*

**Tests utilisant les structures Let, Entier, Ident, Entier, Plus et Mult :**
*?let x=1 in 2*x+1;;*
*Int*

*?let x=1 in (let y = 2 in x+y);;*
*int*

**Test utilisant les structures Letrec, Fonction, If, Egal, Ident, Entier, Mult, Call, Moins :**
*?let rec fact = (function n -> if n=0 then 1 else n*(fact(n-1))) in fact;;*
*(int -> int)*

**Test portant sur les listes utilisant les structures Letrec, Fonction, If, Egal, Ident, Nil, Cons, Hd, Tl, Call :**
*?let rec append=(function x -> (function y -> if x=[] then y else (hd x)::(appe nd (tl x) y))) in append ;;*
*(('a list) -> (('a list) -> ('a list)))*

**Test utilisant en particulier la structure Sup :**
*?let rec test = (function n -> if (n>1) then 0 else 1) in test ;;*
*(int -> int)*

**Test utilisant en particulier la structure Et :**
*?let rec test = (function n -> if (n&&true) then 0 else 1) in test ;;*
*(bool -> int)*

**Test mettant en évidence le lancement d'exception si une variable est utilisée alors qu'elle est indéfinie :**
*?let x=1 in x+y;;*
*Variable indéfinie: y*

**Test mettant en évidence le typage d'un booléen sachant qu'il est utilisé en tant que condition :**
*?let rec test = (function n -> if (n) then 0 else 1) in test ;;*
*(bool -> int)*

**Test mettant en évidence le typage d'une liste connaissant le type de sa tête :**
*?let rec test = (function n -> if (hd n = 2) then 0 else 1) in test ;;*
*((int list) -> int)*

**Test mettant en évidence le typage d'une fonction sachant qu'elle est appliquée à un entier :**
*?let x = (function x -> x 1) in x;;*
*((int -> 'a) -> 'a)*

**Test mettant en évidence le lancement d'exception si la résolution se solde par un échec (n est utilisé comme booléen et comme entier au sein de la fonction) :**
*?let rec fact = (function n -> if n=false then 1 else n*(fact(n-1))) in fact;;*
*Erreur de type*

**Tests supplémentaires mélangeant de nombreux types de structures :**
*?let rec padovan = (function n -> if (n=0) then 0 else if (n=1) then 0 else if ( n=2) then 1 else (padovan (n-2) + padovan (n-3))) in padovan ;;*
*(int -> int)*

*?let rev = (function l ->let rec aux = (function l -> (function res -> if l =[] then res else aux (tl l) ((hd l)::res) )) in aux l [])in rev;;*
*(('a list) -> ('a list))*

*?let rec aux = (function l -> (function res -> if l =[] then res else aux (tl l) ((hd l)::res) )) in aux;;*
*(('a list) -> (('a list) -> ('a list)))*

*?let rec test = (function l -> (function res -> l + res )) in test;;*
*(int -> (int -> int))*

Toutes les règles d’inférence sont ainsi couvertes par l’ensemble de ces tests, et les cas d’erreurs ont été correctement mis en évidence.

# 4 Conclusion

Il est toujours intéressant de mettre en pratique ses connaissances afin d'effectuer quelque chose de concret. Ce projet en est la parfaite illustration, bien que j'aurais souhaité mettre en œuvre un projet dans son intégralité, autrement dit en ayant comme unique base un cahier des charges décrivant les contraintes et le résultat souhaité, ce qui me semble se rapprocher du travail que nous aurons à accomplir lorsque nous travaillerons en entreprise. Bien sûr, le sujet actuel ne pouvait s'y prêter, certaines opérations, telles que l'analyse lexicale et syntaxique des expressions, pouvant être difficilement mises en œuvre avec nos compétences actuelles.

Le sujet tel que défini nous offre donc une bonne introduction aux problématiques des interpréteurs et compilateurs. Il serait ainsi intéressant de réaliser un futur projet ayant pour but d'effectuer une analyse lexicale et syntaxique (éventuellement sémantique) d'une expression à partir d'une grammaire définissant un langage donné en vue d'une éventuelle interprétation.