Compilation de Mini-Scheme/Mini-ML en instructions pour une machine à environnement

Projet de Langages Fonctionnels et Programmation Fonctionnelle

avril 1996

Le but de ce projet est d'écrire, en Caml-Light, un compilateur pour, au choix, le noyau du langage Scheme, appelé pour l'occasion Mini-Scheme, ou, un ML restreint, appelé Mini-ML. Pour Mini-Scheme comme pour Mini-ML les liaisons seront lexicales et l'ordre d'évaluation par valeur de gauche à droite (à l'exception, bien sûr, des conditionnelles). Ainsi, on renonce totalement à reproduire les liaisons dynamiques qui apparaissent dans certaines situations en Scheme.

Le langage cible ne sera pas le langage machine de tel ou telle machine mais un langage de bas niveau portable (dit aussi bytecode) constitué d'instructions pour une machine d'exécution abstraite.

Le compilateur devra accepter des programmes Mini-Scheme (fichiers d'extension `.miniscm`) ou Mini-ML (fichiers d'extension `.miniml`) et produire des fichiers exécutables.

Il sera nécessaire d'écrire un interpréteur pour exécuter les instructions de la machine abstraite. Cet interpréteur sera écrit aussi en Caml-Light et sera invoqué en première ligne des fichiers exécutables produits par le compilateur.

La compilation procèdera en plusieurs étapes. La première étape traduira un flux de caractères en un arbre de syntaxe abstraite pour le langage considéré. Pour cette étape, on utilisera soit les outils camllex et camllyacc (documentés par exemple dans le manuel de référence de Caml-Light), soit le filtrage sur les flux (dont des exemples d'application courent les rues). Dans le cas de Mini-Scheme, cette première étape sera suivie d'une étape intermédiaire traduisant la syntaxe des listes en une syntaxe mettant plus en avant la spécificité de chaque instructions.

La deuxième étape traduira la forme abstraite de programme en une suite d'instructions assembleur, qu'il conviendra ensuite de recopier dans un fichier exécutable.

Aucun algorithme de typage n'est demandé dans le cas de Mini-ML.

1 Les syntaxes de Mini-Scheme et Mini-ML

On décrit ici la syntaxe des programmes Mini-Scheme ou Mini-ML que le compilateur devra être à même de comprendre. On utilise la notation BNF. Les { } autour d'une catégorie syntaxique signifient un nombre arbitraire d'occurrences d'un élément de cette catégorie.

1.1 Définitions communes aux syntaxes de Mini-Scheme et Mini-ML

entier	::=	entier_positif
	|	`-` entier_positif
entier_positif	::=	chiffre {chiffre}
flottant	::=	entier`.`{chiffre} lettre_exposant entier_positif
	|	entier`.`{chiffre}
lettre_exposant	::=	`e` | `E` | `e+` | `E+` | `e-` | `E-`
chaîne	::=	`"` {caractère_littéral} `"`

En résumé, un entier est une suite de un ou plusieurs chiffres de 0 à 9, éventuellement précédée par un signe moins. Un flottant est un entier suivi d'un point, puis, éventuellement, d'une partie fractionnaire consistant en une suite de chiffres, puis, éventuellement encore, d'un exposant constitué de la lettre `e` ou de la lettre `E`, suivie, éventuellement, d'un signe `+` ou `-`, puis d'un entier.

1.2 La syntaxe de Mini-Scheme

La syntaxe de Mini-Scheme est donnée en figure 1.

Un symbole est une suite de caractères littéraux ne commençant ni par un espace, ni par un caractère de contrôle, ni par les caractères “\”, “(”, “)”, “.”, “,”, “’”, “‘”, “@” et “#”.

Les symboles suivants sont des symboles prédéfinis et non redéfinissables : `if`, `define`, `let`, `letrec`, `lambda`, `set!` et `begin`. Pour chacun d'entre eux, le compilateur devra spécifier le comportement que l'on attend d'eux. De plus, dans le cas de `define` et `set!`, les versions avec un seul argument devront aussi être implémentées.

Plus précisément, les syntaxes possibles pour ces symboles seront :

(`if` *expr expr*)
(`if` *expr expr expr*)
(`define` *symbole*)
(`define` *symbole* {*expr*} *expr*)
(`define` (*symbole* {*symbole*}) {*expr*} *expr*)
(`let` ({(*symbole expr*)}) {*expr*} *expr*)
(`letrec` ({(*symbole expr*)}) {*expr*} *expr*)
(`lambda` ({*symbole*}) {*expr*} *expr*)
(`set!` *symbole expr*)
(`set!` *symbole*)
(`begin` {*expr*} *expr*)

Les opérateurs `+`, `-`, `*`, `/`, `=` devront être accessibles dans l'environnement initial de compilation. On se contentera, pour ces opérateurs, de leurs versions à deux arguments. La commande `display` devra elle aussi être disponible dans l'environnement initial.

1.3 Syntaxe de Mini-ML

La syntaxe de Mini-ML est donnée en figure 2.

Un identificateur (abrégé ident) est une suite de lettres, chiffres, contenant éventuellement le caractère de soulignement, mais ne commençant pas par un chiffre.

Les opérations arithmétiques `+`, `-`, `*`, `/` associeront à gauche. Les opérations `+` et `-` lieront plus fort que les opérations `*` et `/`.

La juxtaposition d'atomes correspondra à l'application d'une fonction à ses arguments. Aussi la construction `do`...`return`... correspond à une séquence d'expressions à évaluer dans l'ordre et dont seul le résultat de l'évaluation de la dernière expression est retourné.

La fonction `print` devra aussi être disponible dans l'environnement initial.

2 La syntaxe abstraite des programmes

2.1 La syntaxe abstraite des programmes Mini-Scheme

Le résultat de l'analyse syntaxique d'un programme Mini-Scheme fournira un objet du type `programme_lisp` :

```
type atome_lisp =
   True
 | Nil
 | SInt of int
```

```
programme   ::=  {expr}                    caractère  ::=  #\caractère_littéral

     expr   ::=  symbole                   booléen    ::=  #!true
              |  caractère                              |  #!false
              |  chaîne                                 |  ()
              |  entier                                 |  #!T
              |  flottant                               |  #!t
              |  booléen                                |  #!F
              |  ' expr                                 |  #!f
              |  ( liste_expr )

liste_expr  ::=  expr liste_expr
              |  expr . expr
              |  expr
```

Figure 1 : Syntaxe de Mini-Scheme

```
programme ::= {phrase;;}                           expr1      ::=  expr2
                                                                |  expr2 = expr2
phrase    ::= expr
           |  let liaisons                         expr2      ::=  expr2 + expr2
           |  let rec liaisons                                  |  expr2 - expr2
                                                                |  expr2 * expr2
expr      ::= expr1                                             |  expr2 / expr2
           |  atom {atom}
           |  fun ident {ident} -> expr            atom       ::=  ident
           |  if expr then expr else expr                       |  caractère
           |  let liaisons in expr                              |  chaîne
           |  let rec liaisons in expr                          |  entier
           |  do {expr ;} expr return expr                      |  flottant
           |  ident := expr                                     |  booléen
                                                                |  ( expr )
liaisons  ::= ident {ident} = expr
           |  ident {ident} = expr and liaisons    caractère  ::=  `caractère_littéral`

                                                   booléen    ::=  true | false
```

Figure 2 : Syntaxe de Mini-ML

```
 | SFloat of float
 | SString of string
 | SChar of char
 | Symbole of string
;;

type arbre_lisp =
   Cons of arbre_lisp * arbre_lisp
 | Atome of atome_lisp
;;

type programme_lisp == arbre_lisp list
;;
```

Le premier travail consistera à transformer un `programme_lisp` en un simple `arbre_lisp` à l'aide de l'instruction `begin`.

2.2 La syntaxe abstraite des programmes Mini-ML

Le résultat de l'analyse syntaxique d'un programme Mini-ML fournira un objet du type `programme_miniml` suivant :

```
type constante =
   Bool of bool
 | Int of int
 | Float of float
 | String of string
 | Char of char
 | Indefini       (* Utilise plus tard pour mini-scheme *)
 | Nom of string  (* Utilise plus tard pour mini-scheme *)
 | Unit
;;


type primitive =
   Plus
 | Fois
 | Moins
 | Divise
 | Egal
 | Affiche
;;

type expr_miniml =
   MLApp of expr_miniml * expr_miniml
 | MLFun of string * expr_miniml
 | MLVar of string
 | MLCst of constante
 | MLIf of expr_miniml * expr_miniml * expr_miniml
 | MLSet of string * expr_miniml
 | MLLetIn of (string * expr_miniml) list * expr_miniml
 | MLLetRecIn of (string * expr_miniml) list * expr_miniml
 | MLSeq of expr_miniml * expr_miniml
```

```
 | MLPrim of primitive
;;

type phrase_miniml =
   MLExpr of expr_miniml
 | MLLet of (string * expr_miniml) list
 | MLLetrec of (string * expr_miniml) list
;;

type programme_miniml == phrase_miniml list
;;
```

Les fonctions devront être mises sous une forme commençant par un `fun`.

Comme pour Mini-Scheme, un premier travail sera de traduire un programme de type `programme_miniml` en un terme de type `expr_miniml`, en remplaçant les `MLLet` par des `MLLetIn` et les `MLExpr`, par exemple, par des séquences.

2.3 La traduction des programmes Mini-Scheme et Mini-ML en une syntaxe commune

On va définir une syntaxe interne commune pour Mini-Scheme et Mini-ML. Les objets de cette syntaxe commune auront le type `expr`. Ce type dépend des types `constante` et `primitive` utilisés pour représenter abstraitement les programmes Mini-ML.

```
type expr =
   App of expr * expr
 | Abs of string * expr
 | Var of string
 | Cst of constante
 | LetIn of (string list * expr list) * expr
 | LetRecIn of (string list * expr list) * expr
 | If of expr * expr *expr
 | Set of string * expr
 | Prim of primitive
;;
```

Une fois obtenu un objet de type `arbre_scheme` ou `expr_miniml`, on effectuera une traduction en un objet de type `expr`.

Dans cette traduction, les séquences (`begin` ou bien la construction `do`...`return` ...) seront traduites avec des `let`...`in`... bidon (i.e. des `let`...`in`... liant une variable arbitraire inutilisée par la suite).

Toutes les fonctions devront être décomposées en fonction unaires. En particulier, dans le cas de Mini-Scheme, une fonction définie par `(define (f x y) ...)` devra être vue comme une fonction définie par `(define (f x) (lambda (y) ...))`.

En Scheme, pour les fonctions définies sans argument : `(define (f) ...)`, un argument arbitraire devra être ajouté pour obtenir explicitement une fonction. Pareillement pour les applications de fonctions sans arguments : `(f)`.

Par ailleurs, pour les `let` et `letrec` à plusieurs liaisons, les identificateurs et leurs expressions associées devront être séparés en deux listes distinctes.

C'est l'objet obtenu de type `expr` qui sera, à proprement parler, compilé.

2.4 Le remplacement des noms par des indices

Une traduction supplémentaire consiste à traduire un objet de type `expr` en un objet de type `debruijn_expr` où les noms de variables ont été remplacés par des indices. Un indice n pour les constructeurs `DVar` et `Dset` indique qu'il faut remonter n abstractions (les `fun`) dans l'environnement local.

```
type debruijn_expr =
   DApp of debruijn_expr * debruijn_expr
 | DAbs of debruijn_expr
 | DVar of int
 | DCst of constante
 | DLetIn of debruijn_expr list * debruijn_expr
 | DLetRecIn of debruijn_expr list * debruijn_expr
 | DIf of debruijn_expr * debruijn_expr * debruijn_expr
 | DSet of int * debruijn_expr
 | DPrim of primitive
;;
```

Dans les constructeurs `DAbs`, `DLetRecIn` et `DLetIn`, on n'a donc plus besoin de mémoriser le nom de la variable.

3 Les instructions pour la machine abstraite

3.1 Forme structurée des instructions

Les instructions pour la machine abstraite sont décrites par le type ML suivant :

```
type asm =
   Applique
 | Lambda
 | Fonction of asm list
 | Retour
 | Accede of int
 | Constante of constante
 | Depile of int
 | Reserve of int
 | Let of int
 | Letrec of int
 | Select of asm list * asm list
 | Modifie of int
 | Primitive of primitive
;;
```

Ce sont des instructions pour une machine à environnement avec 4 registres. Ces quatres registres sont : la pile de données (s), l'environnement (e), le code (c), la pile de retour de sous-programme (d).

La pile de donnée contiendra des valeurs. Celles-ci consisteront, soit en des constantes, soit en des fermetures (une fermeture est la paire une fonction (i.e. du code) et de l'environnement de définition de cette fonction). La pile de retour de sous-programme contiendra des fermetures. L'environnement contiendra des listes de valeurs. Quant au code, il consistera en une suite d'instructions pour la machine abstraite.

Les opérations correspondant à ces instructions sont décrites en figure 3.

Remarque : Pour Scheme, `false` doit être compris comme `()` et `true` comme différent de `()`.

Lorsque la pile de retour est vide et l'instruction est *Retour*, alors l'exécution s'arrête.

Pile données	Environnement	Pointeur d'instruction	Pile retour
$(c', e'), s$ s	e e'	$Applique; c$ c'	d $(c, e), d$
s $(c, e), s$	e e	$Fonction\ c; c'$ c'	d d
v, s s	e v, e	$Lambda; c$ c	d d
v, s v, s	e e'	$Retour; c$ c'	$(c', e'), d$ d
s v_m, s	$v_1, v_2, ...$ $v_1, v_2, ...$	$Accede\ m; c$ c	d d
s v, s	e e	$Constante\ v; c$ c	d d
s s	$v_1, ..., v_n, v, ...$ $v, ...$	$Depile\ n; c$ c	d d
s s	e $\underbrace{?, ..., ?}_{\text{n fois}}, e$	$Reserve\ n; c$ c	d d
$v_n, ..., v_1, s$ s	e $v_1, ..., v_n, e$	$Let\ n; c$ c	d d
$v_n, ..., v_1, s$ s	$?, ..., ?, e$ $v_1, ..., v_n, e$	$Letrec\ n; c$ c	d d
$true, s$ s	e e	$Select\ (c_1, c_2); c$ c_1	d $(c, e), d$
$false, s$ s	e e	$Select\ (c_1, c_2); c$ c_2	d $(c, e), d$
v, s v, s	$v_1, ..., v_{m-1}, v_m, v_{m+1}, ...$ $v_1, ..., v_{m-1}, v, v_{m+1}, ...$	$Modifie\ m; c$ c	d d
s $op(v_1, ..., v_p), s$	$v_p, ..., v_1, e$ e'	$Prim\ op; c$ c'	$(c', e'), d$ d

Figure 3 : Assembleur portable

3.2 La compilation en instructions de la machine abstraite

Le mécanisme de la compilation est simple. Si on appelle C l'opération de compilation, alors :

- Une application à un argument ($e\ e_1$) sera compilée en $C(e_1); C(e); Applique$
- Une fonction à un argument (équivalent de (`lambda` $(x)\ e$) en Scheme, ou `fun` $x -> \ e$ en Mini-ML) sera compilée en $Fonction(Lambda; C(e); Retour)$
- Les déclarations locales de variables (correspondant aux (`let` $((x_1\ e_1)...(x_n\ e_n))\ e$) de Scheme et aux `let` $x_1 = e_1$ `and` ... `and` $x_n = e_n$ `in` e de ML) devront être compilés en

  $C(e_1); ...; C(e_n); Let\ n; C(e); Depile\ n$
- Les (`letrec` $((x_1\ e_1)...(x_n\ e_n)\ e)$ de Scheme et `letrec` $x_1 = e_1$ `and` ... `and` $x_n = e_n$ `in` e de ML devront être compilés en

  $Reserve\ n; C(e_1); ...; C(e_n); Letrec\ n; C(e); Depile\ n$
- L'accès à une variable locale sera compilé en $Accede\ m$ où m est le nombre d'abstraction à remonter pour retrouver l'argument associé à la variable en question.
- Un test de la forme `if` e `then` e_1 `else` e_2 sera compilé en

  $C(e); Select((C(e_1); Retour), (C(e_2); Retour))$
- Une opération primitive "op" sera compilée en $Primitive\ op$.
- Une opération d'affectation, (`set!` $f\ e$) en Scheme, $f := e$ en Mini-ML, sera compilée en

  $C(e); Modifie\ m$,

  où m est le numéro associé à f dans l'environnement.

3.3 Forme séquentialisée des instructions

```
type seqasm =
   SApplique
 | SLambda
 | SFonction of int
 | SRetour
 | SAccede of int
 | SConstante of constante
 | SDepile of int
 | SReserve of int
 | SLet of int
 | SLetrec of int
 | SSelect of int * int
 | SModifie of int
 | SPrimitive of primitive
;;

type machine == seqasm vect
;;
```

Le passage de la forme structurée (un programme de type `asm ist`) à la forme séquentialisée `machine` du code pour la machine abstraite consiste à remplacer les suites d'instructions, arguments de `Fonction` et de `Select` par leur adresse. Cette adresse sera une adresse dans le tableau là où on aura trouvé une zone libre pour y déplacer le code argument des instructions `Fonction` et de `Select`.

4 La production de l'exécutable

Pour obtenir un fichier exécutable, on se contentera d'effectuer un `output_value` sur le type `seqasm vect` et de mettre une ligne de la forme `#!interprete` en tête du fichier.

L'interprète devrait donc ainsi être invoqué par la première ligne de commande du fichier exécutable. Cependant, les shells s'attendent, pour l'invocation `#!interprete`, à ce que `interprete` soit un vrai exécutable binaire (ce qui n'est pas le cas des exécutables Caml-Light qui eux-mêmes sont des fichiers de données dont la première ligne est justement une invocation à l'interpréteur Caml-Light `camlrun`). On remplacera donc l'invocation `#!interprete` par une invocation `#!lanceur` où `lanceur` sera le nom de l'exécutable C obtenu à partir du code suivant. L'exécutable Caml-Light pourra alors sans problème se nommer `interprete`.

```
/* Fichier lanceur.c */
main(int argc,char **argv)
{execl("interprete","interprete",argv[1],0);}
```

5 L'interprète

L'interprète récupèrera le programme à l'aide d'un `input_value` sur le type `seqasm vect`.

Voici quelques idées à retenir pour concevoir l'interprète. Les piles seront représentées par des vecteurs d'une taille raisonnablement grande. De même pour le code qui sera chargé à partir du fichier exécutable.

L'environnement pourra être représenté par une liste de valeurs.

Le type des valeurs pourra être celui-là :

```
type valeur = Const of constante | Fermeture of adresse * env

and adresse == int

and env == valeur list
;;
```

6 Et si le temps ou le cœur vous en dit...

On pourra envisager diverses extensions.

Une priorité sera d'ajouter un typeur à Mini-ML (avec η-expansion automatique du code).

On pourra sinon étendre la syntaxe de manière à autoriser la manipulation de données structurées. Ceci se fera en permettant l'instruction `quote` et en autorisant les macro-caractères "‘", "‘", "," et ",@".

On pourra parallèlement songer à étendre la syntaxe de Mini-ML avec des objets de base structurés comme des listes à la Lisp.