École Normale Supérieure 2009–2010

projet de compilation

Mini Modules

version 1 — 15 octobre 2009

L'objectif de ce projet est de réaliser un compilateur pour un fragment de Caml, appelé Mini Modules par la suite, produisant du code MIPS. Il s'agit d'un fragment contenant des entiers, des booléens, des listes, des références et un système de modules et de foncteurs. En revanche, il n'y a ni inférence de types, ni ordre supérieur (de fonctions ou de modules). Il s'agit d'un fragment 100% compatible avec Objective Caml, au sens où tout programme de Mini Modules est aussi un programme Objective Caml correct. Ceci permettra notamment d'utiliser ce dernier comme référence. Le présent sujet décrit précisément Mini Modules, ainsi que la nature du travail demandé.

1 Analyse lexicale et syntaxique

Dans la suite, nous utilisons les notations suivantes dans les grammaires :

$\langle règle \rangle^\star$	répétition de la règle $\langle règle \rangle$ un nombre quelconque de fois (y compris aucune)
$\langle règle \rangle^\star_t$	répétition de la règle $\langle règle \rangle$ un nombre quelconque de fois (y compris aucune), les occurrences étant séparées par le terminal t
$\langle règle \rangle^+$	répétition de la règle $\langle règle \rangle$ au moins une fois
$\langle règle \rangle^+_t$	répétition de la règle $\langle règle \rangle$ au moins une fois, les occurrences étant séparées par le terminal t
$\langle règle \rangle$?	utilisation optionnelle de la règle $\langle règle \rangle$ (i.e. 0 ou 1 fois)
($\langle règle \rangle$)	parenthésage ; attention à ne pas confondre ces parenthèses avec les terminaux `(` et `)`

1.1 Analyse lexicale

Espaces, tabulations et retour-chariots constituent les blancs. Les commentaires débutent par `(*` et s'étendent jusqu'à `*)`, et ils peuvent être imbriqués. On distingue deux catégories d'identificateurs, définies par les expressions régulières $\langle lident \rangle$ et $\langle uident \rangle$ suivantes :

$$\begin{array}{lcl} \langle chiffre \rangle & ::= & \texttt{0–9} \\ \langle alpha \rangle & ::= & \texttt{a–z} \mid \texttt{A–Z} \\ \langle lident \rangle & ::= & (\texttt{a–z})\ (\langle alpha \rangle \mid \texttt{\_} \mid \texttt{'} \mid \langle chiffre \rangle)^\star \\ \langle uident \rangle & ::= & (\texttt{A–Z})\ (\langle alpha \rangle \mid \texttt{\_} \mid \texttt{'} \mid \langle chiffre \rangle)^\star \end{array}$$

Les identificateurs suivants sont des mots clés :

```
else   end    false   if      in
let    match  mod     module  open
rec    sig    struct  then    true
val    with
```

Les constantes entières obéissent à l'expression régulière ⟨*entier*⟩ suivante :

$$\langle entier \rangle \quad ::= \quad \langle chiffre \rangle^{+}$$

1.2 Analyse syntaxique

La grammaire des fichiers sources considérée est donnée figure 1. Le point d'entrée est le non-terminal ⟨*fichier*⟩.

⟨*fichier*⟩	::=	⟨*elt*⟩* `EOF`
⟨*elt*⟩	::=	`let ()` `=` ⟨*expr*⟩
	|	`let rec`? ⟨*lident*⟩ ⟨*farg*⟩* `:` ⟨*type*⟩ `=` ⟨*expr*⟩
	|	`module` ⟨*uident*⟩ ⟨*marg*⟩* (`:` ⟨*signature*⟩)? `=` ⟨*mexpr*⟩
	|	`open` ⟨*upath*⟩
⟨*type*⟩	::=	`unit` | `bool` | `int` | ⟨*type*⟩ `ref` | ⟨*type*⟩ `list`
⟨*farg*⟩	::=	`(` ⟨*lident*⟩ `:` ⟨*type*⟩ `)`
⟨*signature*⟩	::=	`sig` ⟨*decl*⟩* `end`
⟨*decl*⟩	::=	`val` ⟨*lident*⟩ `:` ⟨*type*⟩$^{+}_{\texttt{->}}$
	|	`module` ⟨*uident*⟩ ⟨*marg*⟩* `:` ⟨*signature*⟩
⟨*marg*⟩	::=	`(` ⟨*uident*⟩ `:` ⟨*signature*⟩ `)`
⟨*mexpr*⟩	::=	`struct` ⟨*elt*⟩* `end`
	|	⟨*upath*⟩ (`(` ⟨*upath*⟩ `)`)*
⟨*simple_expr*⟩	::=	`(` ⟨*expr*⟩ `)` | ⟨*lpath*⟩ | ⟨*const*⟩ | `!` ⟨*simple_expr*⟩
⟨*expr*⟩	::=	⟨*simple_expr*⟩
	|	⟨*lpath*⟩ ⟨*simple_expr*⟩$^{+}$
	|	⟨*unop*⟩ ⟨*expr*⟩ | ⟨*expr*⟩ ⟨*binop*⟩ ⟨*expr*⟩
	|	`if` ⟨*expr*⟩ `then` ⟨*expr*⟩ `else` ⟨*expr*⟩
	|	`if` ⟨*expr*⟩ `then` ⟨*expr*⟩
	|	`let` ⟨*lident*⟩ `=` ⟨*expr*⟩ `in` ⟨*expr*⟩
	|	`match` ⟨*expr*⟩ `with` `|`? `[]` `->` ⟨*expr*⟩ `|` ⟨*lident*⟩ `::` ⟨*lident*⟩ `->` ⟨*expr*⟩
⟨*binop*⟩	::=	`+` | `-` | `*` | `/` | `mod` | `<=` | `>=` | `>` | `<` | `<>` | `=`
	|	`&&` | `||` | `::` | `:=` | `;`
⟨*unop*⟩	::=	`-` | `not` | `ref`
⟨*const*⟩	::=	`true` | `false` | ⟨*entier*⟩ | `()` | `[]`
⟨*upath*⟩	::=	⟨*uident*⟩$^{+}_{\texttt{.}}$
⟨*lpath*⟩	::=	(⟨*upath*⟩ `.`)? ⟨*lident*⟩

FIG. 1 – Grammaire des fichiers de Mini Modules

Les associativités et précédences des divers opérateurs ou construction sont données par la table suivante, de la plus faible à la plus forte précédence :

opérateur ou construction	associativité
`let in`	—
`;`	à gauche
`if then else`	—
`match with`	—
`:=`	à droite
`||`	à gauche
`&&`	à gauche
`<  <=  >  >=  =  <>`	à gauche
`::`	à droite
`+  -`	à gauche
`*  /  mod`	à gauche
`-` (unaire)	—
application (inclus `not` et `ref`)	—

2 Analyse sémantique

2.1 Types et environnements de typage

Dans tout ce qui suit, les expressions de types sont de la forme suivante :

$$\tau ::= \texttt{unit} \mid \texttt{bool} \mid \texttt{int} \mid \tau\ \texttt{ref} \mid \tau\ \texttt{list}$$

Un environnement de typage Γ est une suite ordonnée de déclarations $x : \delta$, avec x un identificateur et δ de la forme suivante :

$$\begin{array}{rcll} \delta & ::= & \texttt{val}\ \tau_1 \to \cdots \to \tau_n \to \tau & \text{avec } n \geq 0 \\ & \mid & \texttt{module}\ \Gamma_1 \to \cdots \to \Gamma_n \to \Gamma & \text{avec } n \geq 0 \end{array}$$

Un nom qualifié p est de la forme $x_1.\cdots.x_n$, où les x_i sont des identificateurs et $n \geq 1$. On note $\Gamma(p)$ l'accès à p dans l'environnement Γ, ainsi défini:

$$\begin{array}{rcll} \Gamma(x) & = & \delta & \text{si } x : \delta \in \Gamma \\ \Gamma(p.x) & = & \Gamma'(x) & \text{si } \Gamma(p) = \texttt{module}\ \Gamma' \end{array}$$

Primitives. On supposera par la suite que les programmes sont toujours typés dans l'environnement initial suivant :

$$\begin{array}{rl} \texttt{print\_int} : & \texttt{int} \to \texttt{unit} \\ \texttt{print\_newline} : & \texttt{unit} \to \texttt{unit} \\ \texttt{read\_int} : & \texttt{unit} \to \texttt{int} \end{array}$$

2.2 Typage des expressions

On introduit le jugement $\Gamma \vdash e : \tau$ signifiant « dans l'environnement Γ, l'expression e est bien typée de type τ » et on le définit par les règles d'inférence suivantes :

$$\frac{c \text{ constante de type } \tau}{\Gamma \vdash c : \tau} \qquad \frac{\Gamma(p) = \texttt{val}\ \tau}{\Gamma \vdash p : \tau}$$

$$\frac{\Gamma \vdash e : \texttt{int}}{\Gamma \vdash \texttt{-}\ e : \texttt{int}} \qquad \frac{\Gamma \vdash e : \texttt{bool}}{\Gamma \vdash \texttt{not}\ e : \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int} \quad op \in \{\texttt{=},\ \texttt{<>}, \texttt{<=}, \texttt{>=}, \texttt{<}, \texttt{>}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int} \quad op \in \{\texttt{+},\ \texttt{-},\ \texttt{*}, \texttt{/}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{int}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : \texttt{bool} \quad op \in \{\texttt{\&\&},\ \texttt{||}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x\texttt{=}e_1\ \texttt{in}\ e_2 : \tau_2}$$

$$\frac{}{\Gamma \vdash \texttt{[]} : \tau\ \texttt{list}} \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau\ \texttt{list}}{\Gamma \vdash e_1 :: e_2 : \tau\ \texttt{list}}$$

$$\frac{\Gamma \vdash e_1 : \tau'\ \texttt{list} \quad \Gamma \vdash e_2 : \tau \quad \Gamma, x_1 : \tau', x_2 : \tau'\ \texttt{list} \vdash e_3 : \tau \quad x_1 \neq x_2}{\Gamma \vdash \texttt{match}\ e_1\ \texttt{with}\ [] \to e_2 \mid x_1 :: x_2 \to e_3 : \tau}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{ref}\ e : \tau\ \texttt{ref}} \qquad \frac{\Gamma \vdash e : \tau\ \texttt{ref}}{\Gamma \vdash !e : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau\ \texttt{ref} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : \texttt{unit}}$$

$$\frac{\Gamma(p) = \texttt{val}\ \tau_1 \to \cdots \to \tau_n \to \tau \quad \Gamma \vdash e_i : \tau_i}{\Gamma \vdash p\ e_1\ \ldots\ e_n : \tau :}$$

On notera qu'à la différence de Caml, la construction `let in` ne généralise pas le type de son premier argument. En particulier, l'expression

```
let l = [] in let x = 1 :: l in let y = true :: l in ()
```

n'est pas bien typée dans Mini Modules.

2.3 Typage des modules

Une définition d'un élément de module est notée d et peut prendre quatre formes :

$$\begin{array}{rcl} d & ::= & \texttt{let}\ ()\texttt{=}e \\ & \mid & \texttt{let}\ x\ (x_1 : \tau_1) \ldots (x_n : \tau_n) : \tau = e \\ & \mid & \texttt{let rec}\ x\ (x_1 : \tau_1) \ldots (x_n : \tau_n) : \tau = e \\ & \mid & \texttt{module}\ x\ (x_1 : S_1) \ldots (x_n : S_n) = m \\ & \mid & \texttt{module}\ x\ (x_1 : S_1) \ldots (x_n : S_n) : S = m \end{array}$$

Une signature S est une liste d'éléments de signature. Un élément de signature est noté s et peut prendre deux formes :

$$\begin{array}{rcl} s & ::= & \texttt{val}\ x : \tau_1 \to \cdots \to \tau_n \to \tau \\ & \mid & \texttt{module}\ x(x_1 : S_1) \ldots (x_n : S_n) : S \end{array}$$

On note $\overline{s}$ la déclaration d'environnement correspondante. Pour une valeur, $\overline{s} = s$. Pour un module, on a $\overline{\texttt{module}\ x(x_1 : S_1) \ldots (x_n : S_n) : S} = \texttt{module}\ x : \overline{S_1} \to \cdots \to \overline{S_n} \to \overline{S}$. Pour une signature S, $\overline{S}$ est l'environnement constitué par les différents éléments de signature S, c'est-à-dire $\overline{\texttt{sig}\ s_1\ \ldots\ s_n\ \texttt{end}} = \overline{s_1}, \ldots, \overline{s_n}$. Enfin, pour une définition d, on note $\overline{d}$ la déclaration d'environnement correspondante, le cas échéant (la définition `let ()` $= e$ ne correspond pas à une déclaration).

Définitions. On introduit le jugement $\Gamma_1 \vdash d \Rightarrow \Gamma_2$ pour le typage des définitions. On le définit par les règles d'inférence suivantes :

$$\frac{\Gamma \vdash e : \texttt{unit}}{\Gamma \vdash \texttt{let ()} = e \Rightarrow \Gamma}$$

$$\frac{\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau}{\Gamma \vdash \texttt{let } x\ (x_1 : \tau_1) \ldots (x_n : \tau_n) : \tau = e \Rightarrow \Gamma, x : \tau_1 \to \cdots \to \tau_n \to \tau}$$

$$\frac{n \geq 1 \qquad \Gamma, x : \tau_1 \to \cdots \to \tau_n \to \tau, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau}{\Gamma \vdash \texttt{let rec } x\ (x_1 : \tau_1) \ldots (x_n : \tau_n) : \tau = e \Rightarrow \Gamma, x : \tau_1 \to \cdots \to \tau_n \to \tau}$$

$$\frac{\Gamma, x_1 : \overline{S_1}, \ldots, x_n : \overline{S_n} \vdash m : \Gamma'}{\Gamma \vdash \texttt{module } x\ (x_1 : S_1) \ldots (x_n : S_n) = m \Rightarrow \Gamma, x : \Gamma_1 \to \cdots \to \Gamma_n \to \Gamma'}$$

$$\frac{\Gamma, x_1 : \overline{S_1}, \ldots, x_n : \overline{S_n} \vdash m : \Gamma' \quad \Gamma' \sqsubseteq \overline{S}}{\Gamma \vdash \texttt{module } x\ (x_1 : S_1) \ldots (x_n : S_n) : S = m \Rightarrow \overline{S}}$$

Expressions de modules. Une expression de module m est définie par

$$\begin{array}{rcl} m & ::= & \texttt{struct } d_1 \ldots d_n \texttt{ end} \\ & | & x(x_1) \ldots (x_n) \end{array}$$

On introduit le jugement $\Gamma \vdash m : \Gamma'$ signifiant que m est bien formé dans Γ et de signature Γ'.

$$\frac{}{\Gamma \vdash \texttt{struct end} : \emptyset}$$

$$\frac{\Gamma \vdash d_1 \Rightarrow \Gamma_1 \quad \Gamma_1 \vdash \texttt{struct } d_2 \ldots d_n \texttt{ end} : \Gamma'}{\Gamma \vdash \texttt{struct } d_1 \ldots d_n \texttt{ end} : \overline{d_1}, \Gamma'}$$

$$\frac{\Gamma(p) = \texttt{module } \Gamma_1 \to \cdots \to \Gamma_n \to \Gamma' \quad \Gamma(p_i) = \texttt{module } \Gamma'_i \quad \Gamma'_i \sqsubseteq \Gamma_i}{\Gamma \vdash p(p_1) \ldots (p_n) : \Gamma'}$$

Sous-typage des signatures. Pour deux signatures Γ_1 et Γ_2, on note $\Gamma_1 \sqsubseteq \Gamma_2$ la relation « la signature Γ_1 est incluse dans la signature Γ_2 » et qui signifie que tout module réalisant la signature Γ_1 est acceptable là où un module de signature Γ_2 est attendu. Cette relation est ainsi définie :

$$\frac{\Gamma(x_i) = \delta_i \quad \delta_i \sqsubseteq \delta'_i}{\Gamma \sqsubseteq x_1 : \delta'_1, \ldots, x_n : \delta'_n}$$

où le sous-typage entre déclarations est défini par

$$\frac{}{\texttt{val } \tau_1 \to \ldots \to \tau_n \to \tau \sqsubseteq \texttt{val } \tau_1 \to \ldots \to \tau_n \to \tau}$$

$$\frac{\Gamma'_i \sqsubseteq \Gamma_i \quad \Gamma \sqsubseteq \Gamma'}{\texttt{module } \Gamma_1 \to \ldots \to \Gamma_n \to \Gamma \sqsubseteq \texttt{module } \Gamma'_1 \to \ldots \to \Gamma'_n \to \Gamma'}$$

Règles d'unicité. On ajoute aux règles précédentes la condition d'unicité suivante : une signature ou une structure ne peut contenir deux modules ou deux valeurs de même nom.

Typage des fichiers. Un fichier est un ensemble de définitions, qui sont typées exactement comme le contenu d'un module encadré par `struct end`.

3 Compilation des modules

On se propose ici de compiler les modules par *défonctorisation*. Cette opération consiste à remplacer toute application de foncteur $F(M_1)\ldots(M_n)$ par le module obtenu en substituant dans le corps du foncteur F les arguments formels de F par les arguments effectifs $M_1, \ldots, M_n$. Au final, il ne reste que des modules, mais plus de foncteurs. Les modules peuvent alors être ignorés dans le processus de compilation, car ils ne participent qu'à la structuration de l'espace de noms. Au prix du renommage de certains identificateurs, on peut même supprimer les modules eux-mêmes, pour ne conserver qu'une suite linéaire de déclarations de variables et de fonctions.

Ainsi, on peut supprimer les foncteurs dans le programme suivant

```
module Double(X : sig val f : int -> int end) = struct
  let f (x:int) : int = X.f (X.f x)
end
module X2 = struct let f (x:int) : int = x*x end
module X4 = Double(X2)
module X16 = Double(X4)
```

pour obtenir le programme suivant :

```
module X2 = struct let f (x:int) : int = x*x end
module X4 = struct let f (x:int) : int = X2.f (X2.f x) end
module X16 = struct let f (x:int) : int = X4.f (X4.f x) end
```

En renommant les trois fonctions `f` on obtient enfin le programme suivant :

```
let f1 (x:int) : int = x*x
let f2 (x:int) : int = f1 (f1 x)
let f3 (x:int) : int = f2 (f2 x)
```

La phase de défonctorisation pourra être insérée entre l'analyse sémantique et la production de code.

4 Production de code

L'objectif est de réaliser un compilateur simple mais correct. En particulier, on ne cherchera pas à faire d'allocation de registres mais on se contentera d'utiliser la pile pour stocker les éventuels calculs intermédiaires. Bien entendu, il est possible, et même souhaitable, d'utiliser localement les registres de MIPS.

La sémantique est celle d'Objective Caml. En particulier, l'ordre d'évaluation des arguments d'une fonction ou des éléments d'une liste n'est pas spécifié. On notera que les règles de typage limitent l'utilisation des opérateurs de comparaison aux entiers. Il n'est

donc pas demandé de réaliser une égalité structurelle comme en Caml. On ne cherchera pas à libérer la mémoire allouée pour les listes.

Le schéma de compilation pourra être le suivant :
- toutes les valeurs contenues dans des variables sont stockées dans un mot de 32 bits ; plus précisément
  - les entiers sont directement représentés par des entiers 32 bits (toute petite différence avec Caml),
  - les constantes `()`, `false` et `[]` sont représentées par l'entier 0 et la constante `true` par l'entier 1,
  - une liste autre que `[]` est représenté par un pointeur vers une donnée allouée sur le tas, contenant deux mots de 32 bits ;
- le filtrage sur les listes est réalisé en comparant la valeur filtrée avec 0, qui représente nécessairement `[]` (toute valeur allouée sur le tas sera un pointeur différent de 0) ;

Remarque importante. La correction du projet sera réalisée en partie automatiquement, à l'aide d'un jeu de petits programmes réalisant des affichages (avec `print_int` et `print_newline`), qui seront compilés avec votre compilateur et dont la sortie sera comparée à la sortie attendue. Il est donc très important :
- de correctement compiler les appels à `print_int` et `print_newline` ;
- de respecter la sémantique de ces fonctions (pas de retour-chariot après `print_int` notamment).

5 Travail demandé

Écrire un compilateur, appelons-le mimo, acceptant sur sa ligne de commande exactement un fichier Mini Modules (portant l'extension `.ml`) et éventuellement l'option `-parse-only` ou l'option `-type-only`. Ces deux options indiquent respectivement de stopper la compilation après l'analyse syntaxique et l'analyse sémantique.

Si le fichier est conforme à la syntaxe et au typage décrits dans ce document, votre compilateur doit produire du code MIPS dans un fichier (portant le même nom que le fichier source mais avec le suffixe `.s`) et terminer avec le code de sortie 0 (`exit 0` explicite ou terminaison normale du programme), sans rien afficher.

En cas d'erreur lexicale, syntaxique ou de typage, celle-ci doit être signalée (de la manière indiquée ci-dessous) et le programme doit terminer avec le code de sortie 1 (`exit 1`). En cas d'autre erreur (une erreur du compilateur lui-même), le programme doit terminer avec le code de sortie 2 (`exit 2`).

Localisation des erreurs. Lorsqu'une erreur est détectée par votre compilateur, elle doit être signalée le plus précisément possible, par sa nature et sa localisation dans le fichier source. On adoptera le format suivant pour cette signalisation :

```
File "test.ml", line 4, characters 5-6:
syntax error
```

L'anglicisme de la première ligne est nécessaire pour que la fonction `next-error` d'Emacs puisse interpréter la localisation et placer ainsi automatiquement le curseur sur l'emplacement de l'erreur. (En revanche, le message d'erreur proprement dit pourra être écrit en français si vous le souhaitez.)

Les localisations peuvent être obtenues pendant l'analyse syntaxique grâce aux mots-clés `$startpos` et `$endpos` de Menhir, puis conservées dans l'arbre de syntaxe abstraite.

Modalités de remise de votre projet. Votre projet doit se présenter sous forme d'une archive tar compressée (option "`z`" de tar), appelée *vos_noms*`.tgz` qui doit contenir un répertoire appelé *vos_noms* (exemple : `dupont-durand.tgz`). Dans ce répertoire doivent se trouver les *sources* de votre programme (inutile d'inclure les fichiers compilés). Quand on se place dans ce répertoire, la commande `make` doit créer votre compilateur, qui sera appelé `mimo`. La commande `make clean` doit effacer tous les fichiers que `make` a engendrés et ne laisser dans le répertoire que les fichiers sources.

L'archive doit également contenir un court rapport expliquant les différents choix techniques qui ont été faits et, le cas échéant, les difficultés rencontrées ou les éléments non réalisés. Ce rapport pourra être fourni dans un format ASCII, PostScript ou PDF.

Le projet est à faire seul ou en binôme. Il doit être remis par email à `filliatr@lri.fr`. La date limite de remise sera annoncée sur le site du cours.