UNIVERSITÉ PIERRE ET MARIE-CURIE

MASTER 1
PROJET STL

PHP_ of_ Ocaml

réalisé par :

Hacene KASDI - 3524196
Sofiane GHERSA - 3525755

travail encadré par :

M. Vincent Botbol
M. Abdelraouf Ouadjaout

Année universitaire : 2016-2017

Table des matières

Introduction

Le monde de compilation a grandement changé, les langages de programmation sont en perpétuel évolution, présentant de nouveaux problèmes de compilation, avant qu'un programme puisse être lancé il faut qu'il soit traduit dans une forme qu'un ordinateur puisse l'exécuter. les logiciels qui réalisent cette traduction sont des compilateurs.

Nous mettrons donc l'accent sur une technique de compilation et nous allons exploiter les avantages de la compilation, durant ce projet nous définissons les étapes et le chemin que nous avons emprunté pour concevoir un outil qui permettrait aux développeurs Ocaml de traduire leurs code écrit en OCaml, une suite de déclarations, qui sont évaluées de haut en bas en un langage interprété plus précisément en des scripts PHP, qui seront exécutés sur les serveurs web.

PHP est un langage multi-paradigme et son système de type dynamique permet une grande flexibilité de programmation. Un langage interprété,ceci implique que les erreurs de programmation ne seront détectées qu'à l'exécution, les langages typés dynamiques permettent un développement rapide mais sacrifient la capacité à chercher des erreurs tôt et à inspecter son code rapidement, en particulier sur de nombreuses lignes de code. Cependant, le typage statique fournit une plus grande sécurité, pour illustrer ça, Les ingénieurs Facebook présentent Hack, un langage inspiré de PHP l'ajout principal est le typage statique. de telle sorte que dans le même fichier, une partie du code peut être convertie en Hack tandis que le reste demeure en PHP. Cependant, Hack ajoute des fonctionnalités supplémentaires au-delà de la vérification du type statique parmi lesquelles on trouve les Collections ou les expressions lambda.

OCaml est un langage fonctionnel augmenté de fonctionnalités permettant la programmation impérative.Il se prête à la programmation dans un style fonctionnel, impératif ou orienté objets. Pour toutes ces raisons, OCaml entre dans la catégorie des langages multi-paradigme et très expressif. Sa grande expressivité et son système de type permettent le développement d'application d'une grande fiabilité. beaucoup de grande entreprise, on trouve également facebook et JaneStreet, l'intégrent dans leurs projets. Il est également dans le monde du web grâce au "multi-tiers" permettant le développement d'une application web via un seul langage de programmation assurant le côté client et serveur de manière transparente.

Dans un premier temps nous allons définir la problématique et les motivation, dans un second temps on passera pour la présentation de l'état de l'art, nous présenterons par la suite la conception de php_of_Ocaml,nous détaillerons également le choix de l'approche AST typé $\rightarrow$ php et décrivons les

squelettes de traduction de certaines expressions OCaml et l'équivalent en PHP, nous faisons également quelques observations sur la conception de js_of_OCaml, Nous conclurons par l'implémentation de php_of_OCaml.

Chapitre 1

État de l'art

Durant ce chapitre, Nous allons présenter les phases de compilation d'un programme ensuite nous détaillerons chaque étape de compilation et au final nous mettrons l'accent sur quelques projets existants de génération d'un langage cible à partir du langage Ocaml, On parlera également de js_of_ocaml.

1.1 Étapes de la compilation

Un fichier exécutable est obtenu en traduisant et en reliant comme décrit dans la figure 2.1 .

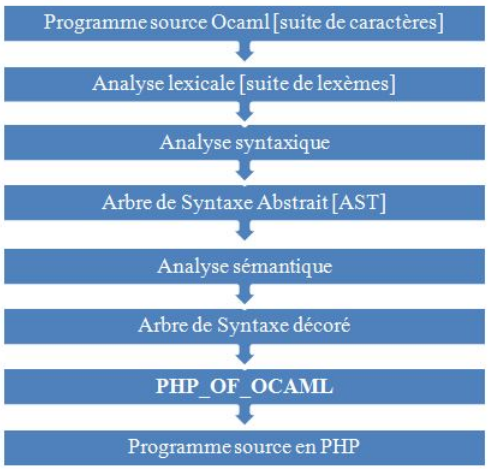

FIGURE 1.1 – Étapes de la production d'un exécutable.

1.2 Les compilateurs Objective CAML

1.2.1 Les phases de génération de code

Les phases de génération de code du compilateur Objective CAML sont détaillées à la figure 2.2 . La représentation interne du code généré par le compilateur est appelée langage intermédiaire (IL).

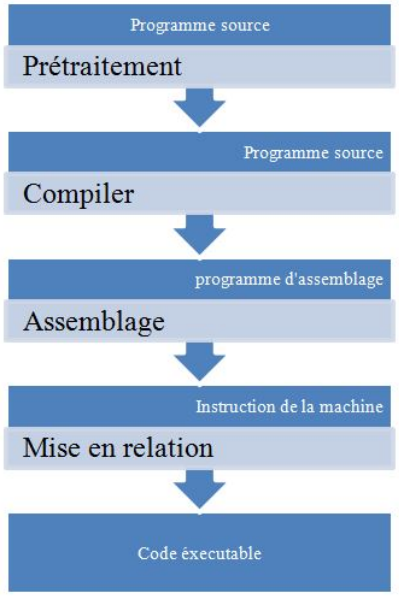

FIGURE 1.2 – Étapes de compilation.

1.2.2 Analyse lexicale

L'étape d'analyse lexicale transforme une séquence de caractères en une suite d'éléments lexicaux. Ces entités lexicales correspondent principalement à des entiers, des nombres en virgule flottante, des caractères, des chaînes de caractères et des identificateurs. Au sein d'une unité lexicale, le premier composant (nom d'unité lexicale) est un symbole abstrait qui sera utilisé lors de l'analyse syntaxique. le second composant (valeur de l'attribut) référence une entrée de la table des symboles [1], ensuite cette entrée sera utile et associée à cette unité lexicale, servira à rassembler des informations à destination de la phase d'analyse sémantique et de la production du code.

1.2.3 Analyse syntaxique

L'étape d'analyse syntaxique construit un AST [2] et vérifie que la séquence des éléments lexicaux est correcte par rapport à la grammaire de la langue. l'analyse syntaxique se décompose en deux étapes : d'abord la construction d'un flux de lexèmes à partir d'un flux de caractères, puis l'obtention de la

1. Conserve des informations sur l'ensemble du programme source, est utilisée pour la suite des phases de compilation
2. Abstract syntax tree ou arbre syntaxique abstrait.

syntaxe abstraite à partir du flux de lexèmes. Durant cette phase il est utile d'imaginer que l'on construit un arbre d'analyse, sachant qu'en pratique un compilateur ne peut pas en construire. Cependant, en théorie l'analyseur syntaxique doit être capable de construire un AST, sans garantir que la traduction soit correcte.

1.2.4 analyse sémantique

L'étape d'analyse sémantique utilise l'arbre abstrait (AST) et les informations de la table de symboles pour vérifier que le programme source est sémantiquement correcte vis-à-vis de la grammaire du langage définie auparavant. L'analyse consiste principalement en une inférence de type qui, en cas de succès, produit le type le plus général d'expression ou de déclaration. en gros durant cette phase on effectue les vérifications nécessaires à la sémantique du langage avant de passer à la phase de génération du code.

1.2.5 AST Typé

Un arbre de syntaxe abstraite est un arbre dont la structure ne garde plus trace des détails de l'analyse, mais seulement de la structure du programme.
Dans un arbre de syntaxe abstraite, construit à partir d'un arbre syntaxique, on n'a pas besoin de garder trace des terminaux qui explicitent le parenthésage, vu qu'on le connaît déjà grâce à l'arbre syntaxique. De même, tous les terminaux de la syntaxe concrète (comme les virgules, les points virgules, les guillemets, les mots clefs etc.) qui ont pour but de signaler des constructions particulières, n'ont plus besoin d'apparaître.

1.2.6 Le principe du code intermediaire

Le but du code intermédiaire est de passer d'une structure arborescente à une (bonne) structure linéaire et de préparer la sélection d'instructions.Les détails dépendant de l'architecture sont relégués à une phase ultérieure de sélection d'instructions.

Quelques caractéristiques du code intermédiaire

- Les branchements sont explicites.
- Code arborescent (expressions) ou linéaire (instructions)
- Utilise une infinité de registres (temporaires), dont l'utilisation est privilégiée (réversible) et le coût négligé.
- L'adressage en mémoire est une forme séparée qui n'est retenue que lorsque c'est indispensable (irréversible).
- L'appel de fonction est implicite, et sera résolu dans une phase ultérieure. (Différentes formes dans différents langages d'assemblage) [3].

3. Code Intermédiaire Génération de Code Linéarisation Canonisation. Didier Rémy Octobre 2000 http ://cristal.inria.fr/remy/poly/compil/3

1.2.7 Génération du code

Il existe deux compilateurs OCaml différents, l'un compilant vers du code-octet (byte-code), l'autre vers du code natif [4].

1. Compilation vers du code-octet

   Le code-octet peut être exécuté par une machine virtuelle OCaml sur d'autres architectures que celle pour laquelle il a été compilé. C'est le même principe que le code-octet pour machine virtuelle Java. Des performances moindres sont la contrepartie de cette portabilité. Par ailleurs, dans le cas d'utilisation de bibliothèques C, ces bibliothèques doivent être présentes sur la machine d'exécution.

2. Compilation vers du code natif

   La compilation vers du code natif permet d'obtenir de bien meilleures performances pour l'exécutable produit. Cependant, l'exécutable obtenu ne peut pas être exécuté sur d'autres architectures que celle sur laquelle il a été compilé.

1.3 Js_of_Ocaml :

Nous présentons les grandes lignes du projet de conception et de mise en œuvre d'un compilateur qui prend du bytecode OCaml et génère un langage cible JavaScript. sachant que ce compilateur traduit le bytecode en une représentation statique à l'affectation unique sur laquelle les optimisations sont effectuées, avant de générer JavaScript.

1.3.1 Fonctionnement de js_of_ocaml

En partant du bytecode plutôt que du code source c'est un défis à relever, sachant que la présentation des données est de bas niveau,Par exemple, les fonctions ont été compilées jusqu'à des fermetures plates ; et le passage d'un code de bas niveau à un langage interprété risque une perte d'information.

Dans le projet de Jérôme Vouillon et Vincent Balat, The js_of_ocaml compiler, ils ont représenté le code non structuré en utilisant le Javascript limité Contrôle.

Dans ce cas, ils ont conçu un moyen pour utiliser facilement les API de Javascript disponible, bien qu'elles soient orientées objet et la convention d'appel de Javascript diffère de celui d'OCaml. [5]

1.3.2 Pourquoi du Bytecode → Javascript

La prise de bytecode comme étant une entrée au lieu d'un langage de haut niveau est un choix judicieux. Un tel compilateur est donc facile à entretenir. Il est également pratique à utiliser, et il peut simplement

4. http ://form-ocaml.forge.ocamlcore.org/modules/presentation.html
5. Jérôme Vouillon and Vincent Balat. From bytecode to Javascript : the Js_of_ocaml compiler. Presented at the OCaml Meeting 2011.

être ajouté à une installation existante des outils de développement. Les bibliothèques déjà compilées peuvent être utilisées directement, sans avoir à réinstaller quoi que ce soit. Enfin, certaines machines virtuelles sont la cible de plusieurs langages. js_of_Ocaml [6] un outil permettant à partir du bytecode OCaml de générer du code javaScript exécutable sur un serveur web.

6. https ://ocsigen.org/js_of_ocaml/

Chapitre 2

Conception

Ce chapitre portera sur la phase de conception, en mettant en place tous les mécanismes qui nous permettront la génération du code php à partir de l'arbre de syntaxe abstrait d'un programme Ocaml. En premier lieu nous allons présenter quelques traits du langage que nous avons opté pour la traduction, et la squelette de traduction de certaines expressions du langage.

2.1 L'objectif de php_of_OCaml

Sachant que php est un langage de programmation interprété, les erreurs de programmation ne seront détectées qu'à l'exécution, les langages typés dynamiques permettent un développement rapide mais sacrifient la capacité à chercher des erreurs tôt,

Le langage OCaml conçu pour garantir la sûreté et la fiabilité des programmes.En partant d'un programme OCaml pour générer un code sémantiquement équivalent en langage PHP typé dynamique, résout le problème des erreurs d'exécution le passage par un typage statique fournit une plus grande sécurité bien avant de produire un script écrit en langage cible typé dynamique.

2.2 Choix de l'approche AST Typé → php

Jérôme Vouillon et Vincent Balat dans leur projet js_of_OCaml Compiler [1],En partant du bytecode pour générer du code javascript est un choix judicieux car la machine abstraite de Zinc (ZAM) offre une API stable, il n'est pas nécessaire de modifier le compilateur à chaque nouvelle version du langage de programmation.

Php_of_OCaml est une approche basée sur la vérification de type du compilateur d'OCaml, en basant sur cette chaîne de compilation d'un programme, à partir de l'AST Typé d'un programme Ocaml produit après la compilation de ce dernier, notre outil va parcourir cet AST bien typé et génère un programme

1. http ://ocsigen.org/js_of_ocaml/

PHP équivalent sémantiquement et qu'il soit lisible pour des programmeurs PHP. On va exploiter les expressions typées que contient chaque noeud de l'AST afin de produire un noeud, une expression équivalente sémantiquement en langage cible, en convertissant les noeuds de l'AST typé en des instructions PHP. à l'aide de cette notion de typage nous obtiendrons un code php sûre et sans perte d'information.

2.3 Grammaire du langage source

Nous avons opté pour la traduction de certains traits du langage Ocaml on citera ci-dessous la grammaire du langage que nous avons traduit :

```
EXPR ::= IDENT
    | 'if' EXPR 'then' EXPR 'else' EXPR (optional)
    | 'for' IDENT '=' EXPR 'to' EXPR 'do' ( EXPR )* 'done'
    | 'while' EXPR 'do' (EXPR)+ 'done'
    | '{' (IDENT '=' EXPR) ( ';' IDENT '=' EXPR)* '}'
    | 'match' EXPR 'with' ( ('|' Pattern '->' EXPR)+)
    |'try' EXPR 'with' '|' IDENT '->' EXPR
    | NUMS
    | 'let' 'rec'? IDENT (' ' IDENT)* '=' EXPR
    | LIST

LIST ::=
      '[' (EXPR)? (';' EXPR)* ']'
    | '(' (EXPR) (','EXPR)* ')'
    | '[|' (EXPR) (';'EXPR)* '|]'

Pattern ::=
     '_'
    |IDENT
    |LIST
```

2.4 Squelette de traduction

2.4.1 Traduction d'une affectation

Nous illustrons ci-dessous trois cas de traduction d'affectation (DÉCLARATION DE VARIABLES) :

1. Global : let ident = expression ; ;
   let IDENT = <EXPR> [2] ;

2. une expression de type CNST.

Ocaml	php
let x =10 ; ;	**$x** = (10);

2. Multiple : let ... and ident2 = expression2 and ident3 = ...
   let IDENT = let IDENT = <EXPR> in let IDENT = <EXPR> in ...
   ou
   let IDENT = let IDENT = <EXPR> and IDENT = <EXPR> in ...

   Dans ce troisième cas, On aura une séquence de **déclarations locales** qui seront traduites au premier lieu, on va suspendre **la déclaration globale**.Au final, on va vérifier si le nom de la variable globale existe dans la liste des déclarations locales, dans ce cas on renomme la variable globale et on lui affecte la dernière instruction, l'exemple ci-dessous illustre ce cas particulier.

Ocaml	php
let x = let x=5 in let a=(7*2) in let y=1 in y+a+x ; ;	$x = 5; $a = (7 * 2); $y = 1; **$x1**=(($x+$a)+$y) ;

2.4.2 Traduction de la boucle FOR

$$\frac{for \quad IDENT \quad = \quad EXPR \quad to \quad EXPR \quad do \quad (EXPR) \quad done}{for(\$[<IDENT>] = [EXPR] \; ; \; \$[<IDENT>] <= EXPR \; ; \; \$[<IDENT>] + +)(EXPR)}$$

$$\frac{for \quad IDENT \quad = \quad EXPR \quad downto \quad EXPR \quad do \quad (EXPR) * \quad done}{for(\$[<IDENT>] = [EXPR] \; ; \; \$[<IDENT>] >= EXPR \; ; \; \$[<IDENT>] - -)(EXPR)*}$$

2.4.3 Traduction de la boucle WHILE

$$\frac{while \quad COND^3 \quad do \quad (EXPR) * \quad done}{do \quad [<COND>] \quad WHILE \quad ([<EXPR>])}$$

2.4.4 Traduction des types ARRAY

$$\frac{let \quad IDENT \quad = \quad [|(EXPR)? \quad (;(EXPR)) * |]}{\$[<IDENT>] \quad = array \quad (([<(EXPR)>])? \quad (,([<(EXPR)>]))*)}$$

2.4.5 Traduction des types TUPLE

$$\frac{let \quad IDENT \quad = \quad ((EXPR)? \quad (,(EXPR))*)}{\$[<IDENT>] \quad = array \quad (([<(EXPR)>])? \quad (,([<(EXPR)>]))*)}$$

2.4.6 Traduction des fonctions

Pour la traduction des fonctions on aura deux cas, les fonctions récursives, le nom de la fonction est précédé par un mot clé **rec** et les fonctions non récursives san le mot clé **rec**.

$$\frac{let \quad rec? \quad IDENT \quad ('\,'IDENT)* \quad = \quad EXPR}{function \quad [<IDENT>] \quad (\quad [<IDENT>]? \quad (',' \, [<IDENT>])*) \quad \{[<EXPR>]\}}$$

2.4.7 Traduction des Exceptions

$$\frac{try \quad EXPR \quad |IDENT \quad -> \quad EXPR}{try\{ \quad [<EXPR>] \quad \}catch(Exception[<IDENT>])\{ \quad [<EXPR>]\}}$$

2.4.8 Traduction pattern matching

```
match EXPR with
  |p1 -> EXPR1
  :
  |pn -> EXPRn
```

Nous avons traduit deux cas concernant les pattern match, selon le type de l'expression EXPR dans **match EXPR with** :

1. Le type de l'expression est un type primitif (int , bool , char ,string ..)
   dans ce cas nous allons utiliser le **switch** de php pour pouvoir comparer la même variable (ou expression de type primitif) avec un grand nombre de valeurs différentes, et d'exécuter différentes parties de code suivant la valeur à laquelle elle est égale.

Ocaml	php
let match b with	switch ($b) {
| 'A' → print_string "vowel"	case 'A' : echo ("vowel") ; break ;
| 'E' → print_string "vowel"	case 'E' : echo ("vowel") ; break ;
| 'B' → print_string "consonant"	case 'B' : echo ("consonant") ; break ;
| _ → print_string "other"	default : echo ("other") ; break ;
	}

2. L'expression est de type List (Construct), dans ce cas nous allons calculer la taille de chaque proposition **p** de | p → EXPR, pour pouvoir utiliser le même mécanisme des **if..then..else..** imbriqués.

Ocaml		php
let match x with		
| []	→ print_int 7	if (sizeof($x) == 0) { echo (7) ; }else{
| a : :b : :[]	→ print_int 19	if (sizeof($x) == 2) { echo (19) ; }else{
| a : :b : :c	→ print_int 39	if (sizeof($x) >= 2) { echo (39) ; }else{
| _	→ print_int 98	echo (98) ; } } }

2.4.9 Traduction des types RECORDS

$$\frac{\{(IDENT \quad = EXPR(;(IDENT \quad = EXPR))*\}}{[\quad([<IDENT>] => [<(EXPR)>] \quad (,([<IDENT>] => [<(EXPR)>]))*]}$$

Voici un exemple de traduction des records ocaml en des tableaux associatifs en php :

Ocaml		php	
let user = {		$user =	[
id	= 2 ;	'id'	=> 2,
pseudo	= "breakLIMITS" ;	'pseudo'	=> "breakLIMITS",
age	= 23	'age'	=> 23
	}		] ;

2.4.10 Les primitives

Nous allons définir quelques primitives les plus utilisées dans le projet et l'équivalent en php.

2.4.10.1 Module List

Ocaml	php
List.length lst	sizeof($lst)
List.nth lst i	$lst[i]
List.append l1 l2	array$_merge($l1, $l2)$

2.4.10.2 Module String

Ocaml	php
String.length str	strlen($str)
String.get str i	$str[i]

2.4.10.3 Module Pervasives (conversion)

Ocaml	php
$\text{int}_o f\_string\_str$	intval($str)
string_of_int	strval($str)