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

Liste des figures

Introduction

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, qui dit Ocaml dit une suite de déclarations, qui sont évaluées de haut en bas, un langage compilé, en un script php, un langage interprété, qui sera exécuté sur les serveurs web.

La principale utilisation que l'on peut avoir de PHP est l'utilisation d'un langage de script traité côté serveur pour la création de pages web [1].

En premier temps on va définir la problématique et les motivation, dans un second temps on passera pour la présentation des bases de compilation d'un programme Ocaml et nous présenterons par la suite l'implémentation de php_of_ocaml.

1. PHP avance Eric Daspet Cyril Pierre de geyer 3eme édition 2006, 2eme tirage 2007

Chapitre 1

Problématique

PHP est un langage multi-paradigme qui reste aujourd'hui, de loin, le langage le plus répandu pour la conception de site webs (côté serveur). Cela s'explique par sa popularité parmi les développeurs web et, de manière intrinsèque, par la richesse de son environnement. Cependant, PHP a des défauts : son système de type dynamique qui permet une grande flexibilité de programmation mais se paye par des erreurs de cohérence souvent difficilement détectables. Ensuite, bien qu'il existe aujourd'hui des outils permettant la compilation (e.g. le projet Hack de Facebook), PHP reste bien souvent interprété. Ceci implique que les erreurs de programmation ne seront détectées qu'à l'exécution : ce qui peut avoir des conséquences malheureuses pour un code industriel [1].

OCaml est un langage fonctionnel augmenté de fonctionnalités permettant la programmation impérative. OCaml étend les possibilités du langage en permettant la programmation orientée objet et la programmation modulaire. Pour toutes ces raisons, OCaml entre dans la catégorie des langages multi-paradigme.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.

L'objectif de ce PSTL est le développement d'un outil permettant à partir d'un programme OCaml de générer du PHP exécutable sur un serveur. En se basant sur la vérification de type du compilateur d'OCaml, on s'assure de produire du code exempt de la plupart des erreurs d'exécution.

1. http ://www-master.ufr-info-p6.jussieu.fr :8080/site-annuel-courant/PHP_OF_OCAML

Chapitre 2

État de l'art

2.1 Étapes de la compilation

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

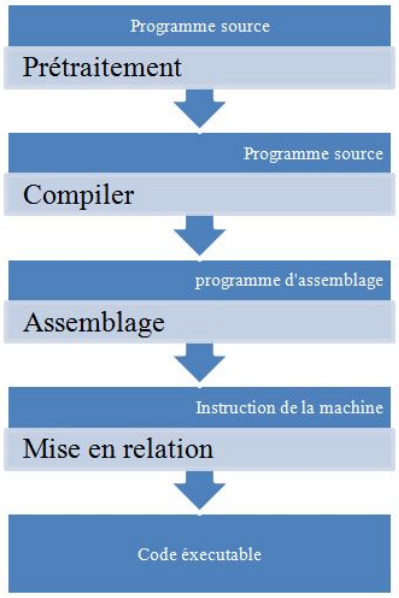

FIGURE 2.1 – Étapes de la production d'un exécutable.

2.2 Les compilateurs Objective CAML

2.2.1 Compilation :

Il existe deux compilateurs OCaml différents, l'un compilant vers du code-octet (byte-code), l'autre vers du code natif [1].

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.

2.2.2 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).

2.2.3 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.

2.2.4 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

1. http ://form-ocaml.forge.ocamlcore.org/modules/presentation.html
2. Abstract syntax tree ou arbre syntaxique abstrait.

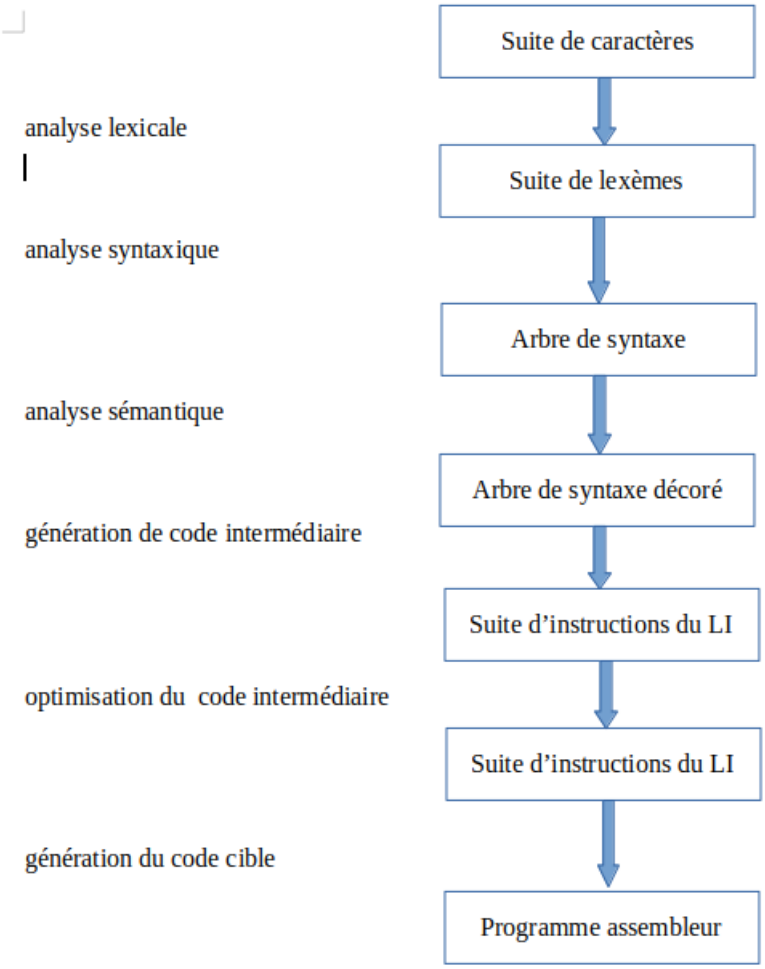

FIGURE 2.2 – Étapes de compilation.

deux étapes : d'abord la construction d'un flux de lexèmes à partir d'un flux de caractères, puis l'obtention de la syntaxe abstraite à partir du flux de lexèmes.

2.2.5 analyse sémantique :

L'étape d'analyse sémantique vérifie un autre aspect de la correction du programme. 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.

2.2.6 AST :

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.

2.2.7 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].

2.2.8 Traduction Bytecode :

M. Mauny B.Vaugon dans leur projet intitulé Traduire OCaml en C en passant par le bytecode ils ont cité que le bytecode OCaml représente un bon point de départ pour de telles opérations : sa conception est stable et les exécutables bénéficient des opérations réalisées par le compilateur OCaml telles l'édition de liens et la collection des constantes. La conception d'outils prenant ces exécutables comme point de départ ne nécessite donc généralement pas de modifier la chaîne de compilation d'OCaml, simplifiant leur évolution et leur maintenance [4].

« Il y a des défis à relever lors du démarrage à partir du bytecode plutôt que le code source. Tout d'abord, la représentation des données est de faible niveau.Par exemple, les fonctions ont été compilées jusqu'à des fermetures plates ; L'interpréteur de bytecode est une machine à pile.

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

4. Michel Mauny, Benoit Vaugon. OCamlCC - Traduire OCaml en C en passant par le bytecode.

En outre, peu d'informations reste à nous aider dans la traduction. Il n'était pas clair au premier abord, si ces représentations de données pourraient être mises en correspondance avec les structures de données Javascript disponibles de manière efficace. Deuxièmement, on peut craindre le passage d'un langage de bas niveau à un langage de niveau faible densité de code.

Troisièmement, il faut trouver des manières de représenter le code non structuré en utilisant le Javascript limité Contrôle. Enfin, il faut concevoir un moyen d'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. Nous croyons que nous avons Défis avec succès et que, à partir de OCaml Bytecode fournit un bon compromis.»
Comme l'ont déjà cité Jérôme Vouillon et Vincent Balat [5]

5. Jérôme Vouillon and Vincent Balat. From bytecode to Javascript : the Js_of_ocaml compiler. Presented at the OCaml Meeting 2011.

Chapitre 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.

La prise de bytecode comme étant une entrée au lieu d'un langage de haut niveau est un choix judicieux. Les machines virtuelles fournissent une API très stable. Un tel compilateur est donc facile à entretenir. Il est également pratique à utiliser, et il peut simplement 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 [1] un outil permettant à partir du bytecode OCaml de générer du code javaScript exécutable sur un serveur web.

1. https ://ocsigen.org/js_of_ocaml/

Chapitre 4

PHP_of_Ocaml

Nous présentons dans cette section le schéma de compilation selon notre outil php_of_ocaml, l'arborescence du projet ainsi que les fichiers de compilation .

Dans notre projet nous allons opter pour une solution qui consiste à partir de l'AST d'un programme OCaml, générer un code PHP équivalent sémantiquement et qu'il soit lisible pour des programmeurs PHP.

4.1 Les fichiers de compilation :

4.1.1 Schéma de compilation :

Processus de génération du code PHP en partant du langage source ocaml.

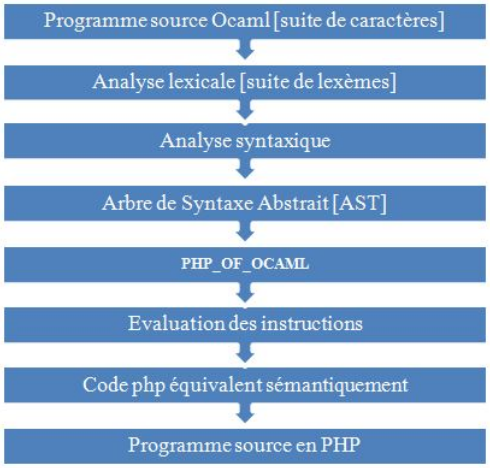

FIGURE 4.1 – processus de génération du code php.

la figure 3.1 ci-dessous vous présente l'arborescence du projet :

FIGURE 4.2 – arborescence du projet php_of_ocaml.

Le Makefile du projet lance la compilation et la traduction des fichiers du dossier /tests.

Le dossier /tests contient les programmes ocaml (les programmes du langage source à traduire), le Makefile parcourt tous les fichiers .ml ensuite il lance l'outil php_of_ocaml pour générer le code en php équivalent sémantiquement qu'on trouvera dans le dossier /Target_language à la racine du projet.

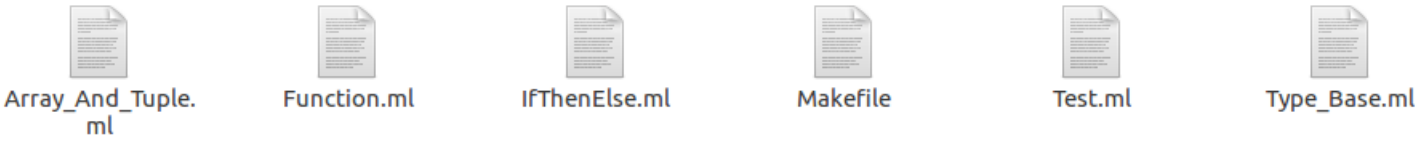

FIGURE 4.3 – les fichiers de tests.

Le dossier /src contient :

1. fichier php_of_ocaml.ml pour parser les programmes ocaml.
2. le fichier generate.ml sert pour la traduction des différentes instructions ocaml vers le langage php.

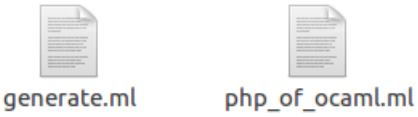

FIGURE 4.4 – les fichiers de traduction.

Vous trouverez notre projet en ligne sur le logiciel de gestion de versions github (php_of_ocaml) [1].

1. https ://github.com/GHERSASofiane/php_of_ocaml

4.2 Implémentation de php_of_ocaml :

4.2.1 Traduction des types de base :

La figure ci-dessous illustre la traduction des constantes et génère l'équivalent sémantiquement en php :

```
 8  (* Generate Constant   ***************************************************************************************
 9
10  let generate_constant fmt cst =
11    let open Asttypes in
12    match cst with
13    | Const_int i -> Format.fprintf fmt "%d" i                                    (* generate an Integer *)
14    | Const_char c -> Format.fprintf fmt "'%c'" c                                 (* generate a char *)
15    | Const_string (s_01 , s_02) -> Format.fprintf fmt "%c%s%c" '"' s_01 '"'      (* generate a String *)
16    | Const_float f -> Format.fprintf fmt "%s" f                                  (* generate a Float *)
17    | _ -> raise (Not_implemented_yet "generate_constant_error")
18
```

FIGURE 4.5 – traduction des types de base.

4.2.2 Traduction des Tuples et tableaux :

On procédera de la même manière pour la traduction des tableaux et tuples :

```
63  (* Generate Array and tuple  ************************************************************************************
64  and  generate_array_and_tuple fmt tpl =
65    let arr_of_tpl = Array.of_list tpl in
66          let taille = Array.length arr_of_tpl - 1 in
67                for i = 0 to taille  do
68                      if i == taille then begin
69                          generate_expression fmt (Array.get arr_of_tpl taille).exp_desc
70                      end
71                      else begin
72                         generate_expression fmt (Array.get arr_of_tpl i).exp_desc ; Format.fprintf fmt " , "
73                      end
74                done
```

FIGURE 4.6 – traduction des Tuples et Arrays.

4.2.3 Traduction des Fonctions :

Traduction des fonctions Ocaml, on distingue deux types ; les fonctions récursives et non récursives :

```
let generate_structure_item fmt item =
  let { str_desc; _ } = item in
  match str_desc with
  | Tstr_value (rec_flag, val_binds) ->begin
                                        match rec_flag with
                                        | Nonrecursive -> List.iter (generate_value_binding fmt) val_binds
                                        | Recursive ->  begin
                                                         (* in order to update the list  *)
                                                         l:= [];
                                                        List.iter (generate_value_binding fmt) val_binds ;
                                                        end
                                    end
```

FIGURE 4.7 – les fonction récursives et non récursives.

```
    match vb_pat.pat_desc with
  | Tpat_var (ident, loc) ->  begin
                               match vb_expr.exp_desc with
                               | Texp_function (label,case,partial) -> begin
                               (* add the Function name to the list *)
                                     l:=loc.txt::!l;
                                     Format.fprintf fmt "function %s%a\n"  loc.txt  generate_expression vb_expr.exp_desc ;

                              end
                                | _ ->  Format.fprintf fmt "$%s = %a;\n"  loc.txt  generate_expression vb_expr.exp_desc
                             end
```

FIGURE 4.8 – traduction de Texp$_{function}$.

```
                        match path with
                        | Pident ident_t ->
                            (* if the function name exits in the list *)
                              if(List.mem ident_t.name !l)
                                then
                                  begin
                                    Format.fprintf fmt " %s " ident_t.name; (*a recursif call*)
                                  end
                                    else
                                      begin
                                        Format.fprintf fmt " $%s " ident_t.name ;
                                      end
```

FIGURE 4.9 – récupération des noms de fonctions.

Chapitre 5

Remarques sur le rapport :

Les points à signaler :

- Traduction assembleur.
- Inclure le fonctionnement de js_of_ocaml (on a juste introduit l'approche)