MLExplain
Reprise de code

Kévin Le Bon

Inria Rennes - Bretagne Atlantique
Équipe Celtique

27 juin 2017

Table des matières

Résumé

Introduction

Chapitre 1

Liste des technologies utilisées

MLExplain est un projet complexe réunissant plusieurs langages. Dans cette partie nous présenterons ces langages ainsi que les principaux outils utilisés pour mettre au point MLExplain.

1.1 Langages utilisés

1.1.1 Javascript

Javascript est un langage de script orienté objet à prototypes. Il est principalement employé dans le développement de pages web interactives, mais aussi dans des applications serveurs grâce à des plateformes de développement telles que NodeJS.

MLExplain utilise intensivement javascript comme langage cible, puisque l'interpréteur d'OCaml est entièrement compilé vers javascript. Il est aussi utilisé pour manipuler l'interface web de MLExplain.

1.1.2 OCaml

Caml est un langage fonctionnel de la famille des langages ML développé par l'Inria. Objective Caml (ou OCaml) en est l'implémentation de référence. OCaml est utilisé dans des projets divers comme l'outil de développement web Ocsigen ou l'assistant de preuves interactif Coq. OCaml est un langage adapté à l'écriture de compilateurs, c'est la raison pour laquelle il est utilisé dans ce projet.

La grande majorité du code OCaml écrit dans MLExplain ne représente qu'un sous-ensemble du langage. En effet, l'un des compilateurs utilisés et présenté dans la section suivante ne gère qu'une sous-ensemble très restraint du langage OCaml. Il n'est pas possible d'utiliser les fonctionalités impératives et objets du langage et la bibliothèque standard est réduite à quelques fonctions du module `Pervasives`. Par ailleurs, les motifs imbriqués ne sont pas disponibles non plus. Dans l'exemple suivant, `Some 5` est un motif imbriqué puisqu'il s'agit d'un motif constructeur avec un motif constante en paramètre.

```
match x with
| Some 5 -> true
| None -> false
```

Le seul moyen de travailler avec des motifs complexes est de lier les données à une variable et d'effectuer un filtrage supplémentaire :

```
match x with
| Some v ->
  begin
    match v with
    | 5 -> true
      | _ -> false
  end
| None -> None
```

1.2 Compilation de code OCaml vers Javascript

MLExplain étant une application web, elle nécessite la compilation du projet en javascript. Par ailleurs, le code de l'interpréteur OCaml est tracé grâce à un système d'enregistrement des évènements (appels de fonctions, retour de fonctions, définitions de variables...). Pour cela, nous utilisons deux compilateurs : js_of_ocaml et le compilateur de JSExplain.

1.2.1 Le compilateur de JSExplain

Le projet JSExplain, sur lequel est basé MLExplain, utilise son propre compilateur d'OCaml. Le but de ce compilateur est d'être capable de placer des appels à la fonction `log_event` qui sert à tracer l'état de l'interpréteur au fur et à mesure de son exécution. Il sert aussi à récupérer les emplacements des différents objets syntaxiques dans le fichier source afin de produire le surlignage du code dans l'interface de JSExplain.

Ce compilateur propose plusieurs modes de fonctionnement :

- unlog : génère un fichier javascript sans traçage
- log : génère un fichier javascript dans lequel les fonctions sont tracées
- mlloc, token et ptoken : génèrent des fichiers contenant des informations sur le placement des éléments syntaxiques du fichier de départ.
- pseudo : génère un fichier de pseudo javascript amélioré avec du sucre syntaxique pour les opérations de filtrage et les opérations monadiques.

Seul le fichier *MLInterpreter.ml* dans le dossier *mlexplain* est tracé, puisqu'il contient les algorithmes pour l'interprétation de code OCaml.

Utilisation de monades

Puisque le compilateur de JSExplain supporte un sous-ensemble limité d'OCaml, nous n'avons pas pu utiliser certaines fonctionalités très communes du langage telles que les exceptions. C'est la raison pour laquelle nous avons eu recours à des constructions comme les monades.

Une monade est une structure de donnée qui représente une action dans un contexte. Elles prennent la forme d'un constructeur de type. Les monades mettent à disposition les fonctions suivantes :

```
Soit une monade m :

val return : 'a -> 'a m
val (>>=) : 'a m -> ('a -> 'b m) -> 'b m
```

La fonction `return` (ou `unit`) permet de contruire une valeur monadique `'a M` à partir d'une valeur de type `'a`. Cette fonction permet d'injecter une valeur dans une monade.

La fonction `bind` (ou l'opérateur infixe `>>=`) sert à composer des monades à l'aide d'une fonction de type `'a -> 'b m`.

Les monades obéissent aux lois suivantes :

$m \Rightarrow return \equiv m$	return est l'élément neutre de bind
$(return\ x) \Rightarrow f \equiv f(x)$	composition à gauche
$(m \Rightarrow f) \Rightarrow g \equiv m \Rightarrow (x \mapsto f(x) \Rightarrow g)$	associativité

Le symbole $\Rightarrow$ correspond à l'opérateur `>>=`.

Comme dit précédemment, les monades sont utilisées afin de calculer dans un contexte. Dans le cadre de MLExplain, elles permettent de gérer élégamment les erreurs. Nous utilisons le type `Unsafe.t` défini ainsi :

```
type ('a, 'b) t =
| Error of string (* Erreur de l'interpréteur *)
| Except of 'a (* Exception dans le programme interprété *)
| Result of 'b (* Résultat dans le programme interprété *)
```

Le constructeur `Result` correspond à un calcul réussi tandis que les deux autres constructeurs correspondent à une erreur. Une fonction susceptible d'échouer retournera un `Unsafe.t`. Par exemple, une division pourrait s'écrire ainsi :

```
let div : int -> int -> (string, int) Unsafe.t =
  fun a -> function
  | 0 -> Unsafe.Except "Division by zero"
  | b -> Unsafe.Result (a / b)
```

Dans le code précédent, on voit bien que `div x` est de type `int -> (string, int) Unsafe.t`, ce qui correspond parfaitement au type de la fonction dont `>>=` a besoin. On pourrait alors écrire :

```
let res = Unsafe.Result 5 >>= div 3 in
match res with
| Unsafe.Error err -> prerr_endline err
| Unsafe.Except err -> prerr_endline err
| Unsafe.Result r -> print_int r
```

Grâce à cette construction, il nous est possible de chaîner les `bind` et ainsi de composer nos monades pour protéger tout un algorithme de l'échec. Nous avons utilisé une extension de syntaxe afin de rendre cette chaîne de `bind` plus lisible :

```
let%result a = Unsafe.Result 5 in
div 3 a

(* équivaut à *)
Unsafe.Result 5 >>= div 3
```

1.2.2 Le compilateur js_of_ocaml

Le compilateur js_of_ocaml est un compilateur de bytecode OCaml vers javascript créé par l'Université Paris-Diderot et Be Sport. Nous utilisons ce compilateur afin de générer le javascript pour la partie frontale de notre interpréteur. Le compilateur de JSExplain ne permet pas d'utiliser des paquets externes, ce qui nous pousse à utiliser un autre compilateur. En effet l'analyse lexicale et syntaxique ainsi que le typage dans notre interpréteur sont gérés grâce au paquet *compiler-libs* d'OCaml. De plus, la partie frontale de l'interpréteur ne doit pas être tracée, il n'est donc pas nécessaire d'utiliser le compilateur de JSExplain.

L'utilisation d'un autre compilateur nous pousse à devoir convertir l'AST du programme en javascript explicitement afin que sa représentation en mémoire soit la même que celle attendue par le code compilé par le compilateur de JSExplain.

Chapitre 2

Architecture du code

Dans ce chapitre nous traiterons de l'architecture du code de MLExplain. Nous allons expliquer le fonctionnement de l'application et nous expliquerons les différentes parties qui la composent. Par la suite nous détaillerons le code de l'interpréteur d'OCaml, qui est la partie centrale du projet.

2.1 Fonctionnement de JSExplain

JSExplain est un interpréteur visuel pour javascript. C'est à dire un interpréteur capable d'afficher l'état du programme exécuté ainsi que son propre état au fur et à mesure de l'exécution. Pour que l'interpréteur puisse afficher l'état dans lequel il se trouve, il est compilé grâce à un compilateur qui ajoute des appels à une fonction de traçage. Les traces obtenues au fur et à mesure de l'exécution permettent de savoir, à tout moment, quelles variables existent, leur valeur ainsi que les valeurs de retour des fonctions de l'interpréteur. L'interface peut réutiliser ces traces afin de d'afficher l'état du programme et de l'interpréteur à chaque état de l'exécution.

2.2 L'interpréteur d'OCaml

L'ensemble des sources de l'interpréteur d'OCaml se trouve dans le dossier *mlexplain/* et peut se découper en trois parties : la partie frontale, la gestion de l'état et des contextes des programmes et enfin l'exécution du code.

La partie frontale de l'interpréteur se trouve essentiellement dans le fichier *TranslateSyntax.ml*. Ce fichier se charge de la traduction de l'arbre syntaxique typée construit par la compiler-libs vers l'arbre de syntaxe défini dans *MLSyntax.ml* puis de la traduction de cet arbre vers son équivalent javascript. Puisque la partie frontale de l'interpréteur n'est pas compilée avec le même compilateur que le reste du code, il est nécessaire de traduire explicitement l'AST vers javascript pour obtenir la même représentation interne que celle attendue par la partie arrière de l'interpréteur. C'est aussi dans le fichier *TranslateSyntax.ml* que se trouve la définition de l'objet javascript `MLExplain` contenant les méthodes `parseExpr` et `parseStructure` appelées depuis l'interface de JSExplain.

Le compilateur OCaml vers javascript de JSExplain ne permet pas d'utiliser la bibliothèque standard d'OCaml, c'est la raison pour laquelle plusieurs modules auxiliaires ont été développés au sein de MLExplain. Parmi ces modules, les plus importants sont `MLArray`, `MLList`, `Vector` et `Map`.

`MLArray` et `MLList` implémentent une partie des fonctions de la bibliothèque standard, ainsi que quelques fonctions utiles, permettant d'utiliser des listes et tableaux en OCaml. `Vector` implémente

un type `'a vec` représentant un tableau muable dynamique (un vecteur). Ce module est écrit en javascript, puisqu'il n'est pas possible de le faire avec le sous-ensemble d'OCaml géré par le compilateur de JSExplain. Ce module sert à modéliser l'état du programme exécuté, à chaque indice du vecteur est associé une valeur et donc, chaque indice sert de référence vers cette valeur. Le module `Map` permet l'utilisation de listes associatives. Ces listes sont utilisé dans MLExplain pour gérer les contextes d'exécution. À chaque identifiant est associé un indice dans le vecteur état du programme.

La partie arrière de l'interpréteur comprend les modules `Value` et `MLInterpreter`. Le module `Value` définit le type `value` qui modélise les données OCaml ainsi que des fonctions qui permettent de manipuler des `value`, notamment le test d'égalité. Le module `MLInterpreter` contient les algorithmes d'exécution du code.

Conclusion