Meta-programmation en OCaml avec ppxlib.

Applications à la bibliothèque data-encoding

Auteur :
Artemiy Rozovyk

Responsable de stage :
Ilias Garnier

Co-encadrants de stage :
Mehdi Bouaziz
Julien Tesson
Mathias Bourgoin

Responsable pédagogique :
Antoine Genitrini

19 septembre 2020

Remerciements

Je tiens tout d'abord à remercier Mehdi Bouaziz, qui m'a fait découvrir le monde magique d'OCaml, ainsi que pour son suivi assidu de mon stage.

De même je remercie Julien Tesson, Ilias Garnier et Mathias Bourgoin qui ont permis l'existence de ce stage et qui ont activement contribué au bon déroulement de ce dernier.

Merci également à Raphaël Proust et à François Thiré d'avoir partagé leur savoir-faire qui m'a permis de surmonter les difficultés que j'ai rencontré.

Enfin, merci à toute l'équipe Nomadic Labs pour leur accueil et l'environnement de travail qu'ils ont réussi à créer malgré les circonstances particulières.

Table des matières

Chapitre 1

Introduction

Ce document présente le travail effectué durant le stage de deux mois qui s'inscrit dans le cadre du parcours d'excellence du Master Informatique de Sorbonne Université, spécialité Sciences et Technologie du Logiciel, en collaboration avec l'entreprise *Nomadic Labs*.

Le cœur du sujet porte sur la technologie *Blockchain*, notamment les échanges bas-niveau qui ont lieu dans un réseau pair à pair.

L'outil que j'ai développé durant ce stage permet la génération automatique des schémas d'encodage utilisés pour la sérialisation de structures de données *OCaml* en format binaire ainsi qu'en *JSON*.

1.1 Blockchain

Une *blockchain* est une base de données immuable (*"append-only"*) distribuée entre un ensemble de participants (appelés *nœuds* ou *pair*). À un instant donné, la *blockchain* comporte toutes les transactions ayant été effectuées entre ces participants et ce depuis la création de la chaîne – c'est-à-dire depuis le bloc initial (*genesis block*). Chaque nouvelle transaction est propagée grâce au *gossip protocol* dans lequel, chaque participant transmet les informations correspondantes à ses voisins. L'un des participants valide toutes les nouvelles transactions en les regroupant en *blocs* qui sont ensuite propagés via le même principe que les transactions et sont appliqués localement à la chaîne par chacun des participants. La création de blocs est récompensée par une quantité de crypto-monnaie.

L'intégrité et la sécurité du système est assurée par un *algorithme de consensus* qui consiste à valider les blocs en s'assurant qu'ils respectent un certain nombre de règles du *protocole*.
L'algorithme de consensus [1] de la première *blockchain* est le *Proof of Work* (PoW), celui-ci a été conçu par une personne ou un groupe de personnes connus sous le pseudonyme *Satoshi Nakamoto* et mis en œuvre pour la crypto-monnaie que l'on connaît sous le nom de Bitcoin.
L'idée principale du *PoW* consiste à obliger le créateur des blocs à effectuer un certain travail, notamment calculer une valeur (*nonce*) qui, ajoutée à la valeur du header du bloc et transformée par une fonction de *hachage* [2], comportera un certain nombre de zéros au début du *hash* produit.
Depuis, d'autres algorithmes de consensus ont été adoptés, notamment le *Proof of Stake* qui a un objectif similaire au *PoW* mais présente des particularités fondamentales quant à la création des blocs. Il s'agit d'un mécanisme dans lequel le créateur des blocs est choisi en fonction de la quantité de crypto-monnaie verrouillée et mise en jeu (*stake*) ainsi qu'à l'aide d'un processus d'élection pseudo-aléatoire prenant en compte la richesse du nœud et l'âge de monnaie verrouillé.

1.2 Nomadic labs

La société d'accueil *Nomadic Labs* est une entreprise française d'environ 50 employés située à Paris et spécialisée dans la recherche et le développement des logiciels distribués ainsi que dans la vérification formelle. La mission principale de *Nomadic Labs* est le développement et la maintenance du protocole *Tezos*.

1. En réalité il s'agit d'une famille d'algorithmes
2. Utilisant le plus souvent l'algorithme SHA256

1.3 Tezos

Tezos est une variante de la technologie *blockchain* proposée initialement en 2014 par Arthur et Kathleen Breitman et déployée en 2018. Contrairement à la plupart des nouvelles *blockchain*, elle n'utilise pas le mécanisme de *PoW*, mais se base totalement sur le modèle de consensus *Proof-of-Stake*. La principale spécificité de *Tezos* est la capacité à modifier les règles du protocole par les participants grâce à un mécanisme de vote, ainsi que l'utilisation des programmes appelés *smart-contracts* qui s'exécutent sur la chaîne.

Chapitre 2

Analyse du problème

2.1 Partage de structures de données

Un logiciel distribué, y compris la *blockchain*, se caractérise par un grand nombre d'échanges via le réseau. Dans le cas de *Tezos* les échanges se font via RPC (Remote Procedure Call). De plus, les données sont éventuellement placées dans un bloc ou stockées sur le disque.

Par conséquent, afin de pouvoir manipuler ces structures de données dans un contexte matériel nous devons trouver une représentation correspondante en format binaire. Cette transformation, appelée aussi *sérialisation*, doit permettre non seulement de stocker ou transmettre les données, mais aussi de retrouver la valeur initiale.

2.2 Data-encoding

Afin de sérialiser les données manipulées, *Tezos* utilise la bibliothèque *data-encoding* qui permet la définition des schémas d'encodage/décodage des types primitifs et fournit un ensemble de combinateurs qui permettent d'exprimer toute structure de données *OCaml*. Une fois que l'utilisateur a défini le schéma d'encodage, il peut être utilisé à la fois pour la conversion en binaire et en *JSON*.

La particularité de la bibliothèque *data-encoding* est la liberté des utilisateurs qui ont beaucoup contrôle sur la forme de l'encodage produit. Cela permet notamment d'avoir un format JSON hautement personnalisable (ce qui le rend, entre autres, directement compatible avec d'autres logiciels).
Un autre avantage est la capacité à créer des encodages optimisés, notamment via l'utilisation des représentations compactes comme `Int8` ou `Int16`.

2.2.1 Exemple d'utilisation

Soit un type simple.

```
type t = (string * int) list
```

Nous définissons le schéma d'encodage suivant :

```
let encoding =
 let open Data_encoding in
  list (tup2 string int31)
```

Le type de `"encoding"` est `(string * int) list encoding`, nous pouvons donc l'utiliser pour les valeurs ayant le type correspondant :

```
let v = [("foo", 32); ("bar", 0)]
let j = Data_encoding.Json.construct encoding v
let w = Data_encoding.Json.destruct encoding j
let () = assert (v = w)
```

2.2.2 Inconvénients

Actuellement tous les encodages sont définis manuellement ce qui a l'avantage d'offrir une certaine liberté aux utilisateurs. Toutefois, cette approche comporte également des inconvénients, notamment le fait de générer des erreurs [1], surtout dans le cas des encodages pour les types complexes (*variants*, *records*).

— Encodings longs :

```
type t = C1
       | C2 of ModuleX.t
       | C3 of string
         ...
       | C142 of (int * string) list
```

```
type record_t = {
 a0 : t;
 a1 : user_defined_t;
 a3 : record_t;
 ...
 a371 : ModuleY.t;
}
```

La définition des schéma d'encodage pour les types ayant beaucoup de champs/attributs peut être très chronophage. De plus, il s'agit d'un code qui contient énormément de répétitions (à quelques caractères prés).

— Oubli des cas (suite à une mise à jour) :

```
type t = A | B of string

let encoding =
 union ~tag_size:`Uint8 [
  case ~title:"A" (Tag 0)
   (obj1 (req "A" unit))
   (function A -> Some () | _ -> None)
   (fun () -> A);
  case ~title:"B" (Tag 1)
   (obj1 (req "B" string))
   (function B b -> Some b | _ -> None)
   (fun b -> B b);
 ]
```

```
type t = A | B of string | C of int

let encoding =
 union ~tag_size:`Uint8 [
  case ~title:"A" (Tag 0)
   (obj1 (req "A" unit))
   (function A -> Some () | _ -> None)
   (fun () -> A);
  case ~title:"B" (Tag 1)
   (obj1 (req "B" string))
   (function B b -> Some b | _ -> None)
   (fun b -> B b);
* case ~title:"C" (Tag 2)
   (obj1 (req "C" int31))
   (function C c -> Some c | _ -> None)
   (fun c -> C c)   ???  *)
 ]
```

Soit un type `t` et son encoding correspondant (à gauche). Il arrive parfois que le type soit mis à jour. Par exemple, on rajoute un cas `C of int` à sa définition (à droite). Actuellement, il est supposé que l'utilisateur ait le réflexe de mettre à jour l'encoding correspondant (code commenté), mais en pratique ce n'est pas toujours le cas. Cette discordance entre la définition d'un type et son encodage provoque des erreurs critiques à l'exécution, comme c'était le cas d'une version du protocole *Tezos* qui était en phase de test [2]. Cependant, ces erreurs sont extrêmement difficiles à repérer car certains cas n'arrivent que très rarement et ce genre d'oubli reste inaperçu pendant la phase de test.

— Incompatibilité en *JSON* :

1. "With great power comes great responsibility" ©
2. Qui s'exécutait dans un réseau détaché du *Mainnet* - le réseau principal de *Tezos* et qui n'a pas causé de dégâts économiques.

```
type t = Toto | Saucisse

let encoding =
 union [
  case (Tag 0) ~title:"Toto"
   empty  (*(obj1 (req "Toto" empty))*)
   (function Toto -> Some () | _ -> None)
   (fun () -> Toto);
  case (Tag 2) ~title:"Saucisse"
   empty  (*(obj1 (req "Saucisse" empty))*)
   (function Saucisse -> Some () | _ -> None)
   (fun () -> Saucisse);
 ]
```

```
let () =
  let v = Saucisse in
  let j = Json.construct encoding v in
  let w = Json.destruct encoding j in
  assert (w = Toto)
```

Pas de "Assert_failure".

Une autre difficulté que pose la bibliothèque *data-encoding* est le fait de pouvoir définir des schémas qui ne sont corrects que si on utilise un seul format d'encodage.
Dans l'exemple ci-dessus (à gauche), nous avons un `encoding` du type `t` qui fonctionne correctement lors de la (dé)sérialisation binaire. Toutefois, comme on s'aperçoit dans le même exemple (à droite), l'utilisation de ce schéma en *JSON* provoque un paradoxe où l'on encode une valeur `Saucisse` mais le décodage donne `Toto`. En effet, le `Tag` permettant de distinguer les cas n'est utilisé qu'en binaire.
Une solution à ce problème consiste à encapsuler l'encodage dans un `"obj1 (req "Id" encoding)"` (code commenté) ce qui permet de distinguer chaque `case` ayant le même type de paramètre.

2.2.3 Bibliothèques alternatives

Il existe d'autres bibliothèques OCaml permettant la sérialisation de données.

— **sexplib** est une bibliothèque qui permet de sérialiser des valeurs OCaml sous format de *S-expression* (expression symbolique) qui est une convention pour la représentation des données arborescentes sous une forme textuelle parenthésée. Cette bibliothèque est souvent utilisé avec ppx_sexp_conv qui est un générateur automatique de schémas de conversion en *s-expression*.

— **yojson** est une autre bibliothèque qui propose des fonctions permettant la lecture et l'écriture de données sous format JSON à partir des donnés OCaml. Comme `sexplib`, cette bibliothèque est conjointement utilisée avec ppx_deriving_yojson qui a également le rôle de générateur automatique d'outils de conversion des valeurs d'un type OCaml en format JSON et vice-versa.

— **bin_prot** est une bibliothèque de sérialisation des données OCaml sous un format binaire. De même, il existe un ppx_bin_prot qui facilite l'utilisation de la bibliothèque.

Cependant, suite à des besoins spécifiques de *Tezos*[3] et afin de limiter les dépendances extérieures l'utilisation de ces bibliothèques n'a pas été envisagée.

Chapitre 3

Implementation de ppx_encoding

3.1 Ppxlib

Ppx est un système de méta-programmation pour le language *OCaml*.

3. Certaines bibliothèques ne supportent pas les types récursifs, comme c'est le cas de *sexplib* (géré par l'opérateur `mu` de data-encoding), ou bien ne comportent qu'un type d'encodage (*yojson*,*bin_prot*).

3.1.1 Principe

Ppx permet notamment de générer du code lors de la compilation. La principale spécificité de *ppx* est le fait d'être étroitement lié au *parser d'OCaml*, et au lieu de fonctionner uniquement au niveau du texte, il opère sur les structures de la représentation interne du langage, c'est-à-dire sur *l'Abstract Syntax Tree (AST)* non typé, donc sur un code *OCaml* syntaxiquement correct.

On s'intéresse plus particulièrement aux *Ppx Derivers* dont le rôle est de créer des nouveaux nœuds, dérivés de la définition d'un type, et de les insérer dans l'AST. Ces derniers sont particulièrement utiles quand il s'agit de produire des fonctions qui sont fastidieuses à écrire ce qui est une source fréquente d'erreurs.

3.1.2 Exemples de PPXes

- **ppx_deriving** est une collection de *deriving plugins* qui inclut entre autres `show`, `eq`, `ord` qui dérivent respectivement la fonction d'affichage (*pretty-printer*), d'égalité et de comparaison des valeurs.
- **ppx_mysql** dérive des fonctions permettant la manipulation des données dans une base de données *MySQL* exprimées comme des valeurs *OCaml*.
- **ppx_python** permet la génération de fonction de conversion des structures de donnés *OCaml* en *Python* et vice-versa.

3.2 Structure générale

La structure générale de ppx_encoding est la suivante :

```
/
├── dune
├── dune-projet
├── expander
│   ├── dune
│   ├── ast_helpers.ml
│   └── ppx_encoding_expander.ml
├── src
│   ├── dune
│   └── ppx_encoding.ml
└── test
    ├── tuple_test.ml
    ├── qualified_type_test.ml
    └── variant_test.ml
```

Il s'agit d'un projet `dune` classique qui contient les modules suivants :

- Le point d'entrée du programme est le fichier `src/ppx_encoding.ml`. Il contient simplement un appel à la bibliothèque *ppx* avec

```
Deriving.add "encoding"
  ~str_type_decl:
    (Deriving.Generator.make_noarg Ppx_encoding_expander.str_type_decl)
```

  ce qui permet d'annoter les types avec `[@@deriving encoding]`.

- Le module `expander/ppx_encoding_expander.ml` définit la fonction :

```
val str_type_decl :
  loc:Location.t -> path:string -> rec_flag * type_declaration list -> structure
```

  Cette fonction prend en paramètres les informations relatives au type (notamment une `type_declaration` et construit une `structure` qui correspond au nœud ayant la forme `"let ... = ..."` qui est ensuite inséré dans l'AST. Ce même fichier expose également un certain nombre d'informations générales relatives au type comme son `ptype_kind`, le fait d'être récursif ou pas, ainsi que les traitements relatifs aux paramètres du type.
- Le fichier `ast_helpers.ml` est le plus gros et contient toutes les fonctions de création des nœuds [1] suivant les éléments contenus dans `type_declaration`.

1. Qui sont naturellement très verbeuses.

— Finalement, le dossier `test` contient des fichiers démontrant la génération des encodages pour les types supportés. Ces tests permettent à la fois d'identifier les types qui sont pris en charge ainsi que de voir leur structure.

3.3 Utilisation de ppx_encoding

Une fois la bibliothèque installée, nous pouvons ajouter la clause :

```
(preprocess (pps ppx_encoding))
```

dans la stance `library` ou `executable` du fichier `dune`.
Ensuite, nous pouvons annoter le type comme suit :

```
type t = (int * string) list [@@deriving encoding]
```

Ce qui rendra l'encodage de `t` disponible dans le reste du programme.
Une autre possibilité est d'utiliser l'annotation `"[@@deriving_inline encoding]"` suivi d'un `[@@@end]` :

```
type t = (int * string) list [@@deriving_inline encoding]
[@@@end]
```

L'exécution de la commande :

```
dune build --auto-promote
```

rendra le code généré visible en mettant à jour les fichiers `.ml` en question :

```
type t = (int * string) list [@@deriving_inline encoding]
let encoding =
 let open! Data_encoding in
  list (tup2 int31 string)
[@@@end]
```

Ce mécanisme est particulièrement intéressant en cas d'ajout d'éléments dans un type, car il permet de voir directement tout changement de encoding lors de *code review*. De plus, en cas de changement de la librairie elle même, cela permet de vérifier la rétrocompatible de la nouvelle version par rapport aux encodages existants.

3.4 Types supportés

Par soucis de clarté et de concision, le code d'ouverture de module (`let open! Data_encoding in` ainsi que les fonctions auxiliaires générées automatiquement par *ppxlib* sont omis.

3.4.1 Nommage

Soit un type nommé `t`, le nom de l'encodage généré suit les conventions de *Tezos* et sera nommé `encoding`. Pour les autres noms, les encodages porteront un nom ayant la forme : `encoding_of_some_name`.

3.4.2 Types de base

```
type t = int
[@@deriving_inline encoding]
 let encoding = int31
[@@@end]
```

```
type t2 = string
[@@deriving_inline encoding]
 let encoding_of_t2 = string
[@@@end]
```

Ces deux exemples montrent le cas basique d'une dérivation d'encodage à partir des types simples : `int` est encodé avec une valeur de *data-encoding* - `int31`. Le type `t2` comportant

une `string` est encodé en une valeur de *data-encoding* portant le même nom.

```
type t3 = t
[@@deriving_inline encoding]
 let encoding_of_t3 = encoding
[@@@end]
```

```
type t4 = t2
[@@deriving_inline encoding]
 let encoding_of_t4 = encoding_of_t2
[@@@end]
```

Dans le cas d'utilisation d'un type nommé pour la définition d'un nouveau type, l'outil suppose qu'une valeur nommée `encoding` est disponible (si le type s'appelle `t`, sinon `encoding_of_...` est attendu.).

```
type t5 = int list
[@@deriving_inline encoding]
 let encoding_of_t5 = list int31
[@@@end]
```

```
type t6 = t list
[@@deriving_inline encoding]
 let encoding_of_t6 = list encoding
[@@@end]
```

Les mêmes règles s'appliquent pour une liste d'éléments.

3.4.3 Types qualifiés

```
type t1 = Test_types.t

let encoding_of_t1 =
 Test_types.encoding
```

```
type t2 = Test_types.SubTest.t

let encoding_of_t2 =
 Test_types.SubTest.encoding
```

```
type t3 = Test_types.some_t

let encoding_of_t3 =
 Test_types.encoding_of_some_t
```

L'utilisation des types qualifiés suit les même règles de nommage évoquées dans 3.4.2. Cela reste vrai pour une profondeur arbitraire d'appel de sous-modules.

3.4.4 Records

```
type t = { er : int }

let encoding =
 conv
  (fun { er } -> er)
  (fun er -> { er })
  int31
```

```
type t2 =
 {
   p : int;
   q : string;
   b : bool;
   f : float
 }

 let encoding_of_t2 =
  conv
   (fun { p; q; b; f } -> (p, q, b, f))
   (fun (p, q, b, f) -> { p; q; b; f })
   (obj4
    (req "p" int31)
    (req "q" string)
    (req "b" bool)
    (req "f" float))
```

```
type t3 =
 { a0 : int;
   a1 : int;
   ...
   a22 : int }

 let encoding_of_t3 =
 conv
  (fun { a0; a1; ...; a22} ->
   (((a0, a1, a2, a3, a4),
     (a5, a6, a7, a8, a9, a10)
     ..., a21, a22)))
  (fun (((a0, a1, a2, a3, a4),
     (a5, a6, a7, a8, a9, a10)
     ..., a21, a22))), ...
     -> { a0; a1; ...; a22})
  (merge_objs
       (merge_objs
          (obj5 (req "a0" int31)
          ... (req "a4" int31))
          (obj6 (req "a5" int31)
          ...(req "a10" int31)) )
       (merge_objs
          (obj6 (req "a11" int31)
          ... (req "a16" int31))
          (obj6 (req "a17" int31)
          ... (req "a22" int31)) ) )
```

Afin d'encoder un *record*, nous avons besoin de la fonction `conv` qui prend en paramètre les fonctions de projection et d'injection qui servent respectivement à encoder et decoder les données, ainsi que le schéma de l'ensemble des champs.
Chacun des exemples ci-dessus doit être traité de manière unique. Dans le cas d'un *record* ayant un seul champ les fonctions ont une forme
`(fun {attribut} -> var)` et `(fun var -> {attribut})`
et le schéma correspond tout simplement à l'encodage du type de l'attribut.
Dans le cas d'un *record* ayant plusieurs champs (moins de 10) les fonctions deviennent
`(fun {attr1; ...; attrn} -> (attr1, ..., attrn)` et

`(fun (attr1, ..., attrn) -> {attr1; ...; attrn})` alors que le schéma prend la forme d'un `objN` où `N` est le nombre d'attributs dont les arguments sont leurs encodages (`req` pour requis) identifiés par leur nom.
Quant aux *records* très longs, nous ne pouvons plus utiliser `objN` car ces fonctions ne sont pas définies au-delà de 10 attributs. Pour les encoder, nous avons besoin de la fonction `merge_objs` qui créée une agrégation des `objN` (en divisant la liste au milieu). De plus nous avons besoin de la fonction de conversion `conv` car le type de `merge_obj` est un tuple de tuples.

3.4.5 Tuples

```
type typ_type0 =
  int
  * int
  * int
  * int
  * int
  * int
[@@deriving_inline encoding]
 let encoding_of_typ_type0 =
  tup6 int31 int31 int31 int31 int31 int31
[@@@end]
```

```
type typ_type1 = int * int * int ... int (*23 fois*)
[@@deriving_inline encoding]
 let encoding_of_typ_type1 =
  conv
  (fun ( v0, v1, ..., v22 ) ->
    ( ((v0, ..., v4), (v5, ..., 10)),
      ((v11, ..., v16), (v17, ..., v22)) ))
  (fun ( ((v0, ..., v4), (v5, ..., v10)),
         ((v11, ..., v16), (v17, ..., v22)) ) ->
    ( v0, v1, ..., v22 ))
  (merge_tups
     (merge_tups
        (tup5 int31 int31 int31 int31 int31)
        (tup6 int31 int31 int31 int31 int31 int31))
     (merge_tups
        (tup6 int31 int31 int31 int31 int31 int31)
        (tup6 int31 int31 int31 int31 int31 int31)))
[@@@end]
```

Dans le cas des tuples ayant une arité inférieure à 10 on utilise le combinateur `tupN` où `n` correspond à son arité. De manière similaire que pour les *records*, les tuples d'arité supérieure à 10 sont traités par la fonction `merge_tups` (qui créée également un tuple de tuples et qui est ensuite converti en tuple ayant la forme initiale).

3.4.6 Types algébriques

```
type t = A

let encoding =
  conv
   (fun A -> ())
   (fun () -> A)
   (obj1 (req "A" unit))
```

```
type t1 = B of int

 let encoding_of_t2 =
   conv
    (fun (B a) -> a)
    (fun a -> B a)
    (obj1 (req "B" int31))
```

```
type t2 = C of int * int

let encoding_of_t2 =
 conv
  (fun (C (c0, c1)) -> (c0, c1))
  (fun (c0, c1) -> C (c0, c1))
  (obj1 (req "C" (tup2 int31 int31)))
```

Les types algébriques ayant un seul cas de constructeur sont encodés avec un `conv` [2]. Similairement, le type de l'argument du constructeur (vide, un élément, un tuple ou *inline record*) change la structure des fonctions de projection et d'injection (au niveau du nœud AST qui doit être généré).

2. Il est plus naturel de séparer ce cas car il nécessite des fonctions de projection et d'injection qui ont une forme différente au cas général (pas de `(function... | _ -> None)`).

```
type t =
 | A
 | B of string
 | C of float list

 let encoding =
  union ~tag_size:`Uint8 [
    case ~title:"A" (Tag 0)
      (obj1 (req "A" unit))
      (function A -> Some () | _ -> None)
      (fun () -> A);
    case ~title:"B" (Tag 1)
      (obj1 (req "B" string))
      (function B b -> Some b | _ -> None)
      (fun b -> B b);
    case ~title:"C" (Tag 2)
      (obj1 (req "C" (list float)))
      (function C c -> Some c | _ -> None)
      (fun c -> C c);
  ]
```

```
type t2 =
| X of t
| F of Test_types.t
| Y of { toto : string }
| Z of { titi : int; tata : string }

 let encoding_of_t2 =
  union ~tag_size:`Uint8 [
    case ~title:"X" (Tag 0)
      (obj1 (req "X" encoding))
      (function X x -> Some x | _ -> None)
      (fun x -> X x);
    case ~title:"F" (Tag 1)
      (obj1 (req "F" Test_types.encoding))
      (function F f -> Some f | _ -> None)
      (fun f -> F f);
    case ~title:"Y" (Tag 2)
      (obj1 (req "Y" (obj1
       (req "toto" string))))
      (function Y { toto } ->
       Some toto | _ -> None)
      (fun toto -> Y { toto });
    case ~title:"Z" (Tag 3)
      (obj1 (req "Z"
       (obj2 (req "titi" int31)
        (req "tata" string))))
      (function Z { titi; tata } ->
       Some (titi, tata) | _ -> None)
      (fun (titi, tata) -> Z { titi; tata });
  ]
```

Pour les types algébriques ayant plusieurs constructeurs, nous utilisons le combinateur `union` qui prend en paramètre la taille du *tag* maximale (`Uint8` permet d'encoder jusqu'à 256 constructeurs) ainsi qu'une liste de `case`. Chaque `case` correspond à un constructeur et prend en paramètre un titre servant pour la documentation des encodages, le `Tag` sert à identifier l'emplacement dans l'encodage binaire, les fonctions de projection et d'injection, ainsi que le schéma d'encodage de l'argument.
Afin de rendre le schéma généré compatible avec le format *JSON*, nous avons besoin d'un moyen de distinguer des constructeurs ayant le même type d'argument. Pour cela on encapsule le schéma dans un `"obj1 (req "Nom_constr" encoding_arg)"`.
Quant aux arguments qui sont des tuples ou des *inline records* leur structure change de même manière comme décrit dans les sections relatives (3.4.5 et 3.4.4).

3.4.7 Options

```
type t = int option

let encoding = option int31
```

```
type t2 = { a : int option }

let encoding_of_t2 =
 conv
  (fun { a } -> a)
  (fun a -> { a })
  (option int31)
```

```
type t3 =
{ a : int option ;
  b : int option }

let encoding_of_t2 =
 conv
  (fun { a; b } -> (a, b))
  (fun (a, b) -> { a; b })
  (obj2
   (opt "a" int31)
   (opt "b" int31)
  )
```

```
type t =
| A of { toto : int option }
| B of float option
| C of { titi : string option list option option }

let encoding =
 union ~tag_size:`Uint8 [
  case ~title:"A" (Tag 0)
    (obj1 (req "A" (obj1 (opt "toto" int31))))
    (function A { toto } -> Some toto | _ -> None)
    (fun toto -> A { toto });
  case ~title:"B" (Tag 1)
    (obj1 (req "B" (option float)))
    (function B b -> Some b | _ -> None)
    (fun b -> B b);
  case ~title:"C" (Tag 2)
    (obj1 (req "C"
      (obj1 (opt "titi"
        option (list (option string)))))))
    (function C { titi } -> Some titi | _ -> None)
    (fun titi -> C { titi });
 ]
```

Les types optionnels sont gérés par la fonction `option` de *data-encoding*.
Une particularité des types optionnels, est la génération d'un champ `opt` (pour optionnel) au lieu d'un `req` dans le cas de *records* ayant plus de 2 attributs ainsi que pour les *inline records*.

3.4.8 Types récursifs

```
type t = A of t list

let encoding =
 mu "t" (fun t_encoding ->
  conv
   (fun (A a) -> a)
   (fun a -> A a)
   (obj1 (req "A" (list t_encoding))))
```

```
type t =
 | A
 | B of { toto : t }
 | C of t list

let encoding =
 mu "t" (fun t_encoding ->
  union ~tag_size:`Uint8 [
   case ~title:"A" (Tag 0)
    (obj1 (req "A" unit))
    (function A a -> Some a | _ -> None)
    (fun a -> A a);
   case ~title:"B" (Tag 1)
    (obj1 (req "B" (obj1 (req "toto" t_encoding))))
    (function B { toto } -> Some toto | _ -> None)
    (fun toto -> B { toto });
   case ~title:"C" (Tag 2)
    (obj1 (req "C" (list t_encoding)))
    (function C c -> Some c | _ -> None)
    (fun c -> C c);
  ])
```

Les types récursifs sont encodés avec le combinateur `mu`[3] qui prend en paramètre une fonction dont l'argument est l'encoding du type même.

3. `mu` provient du lambda calcul, μ x.y permet de définir y en fonction de lui-même(x apparaît dans y)

3.4.9 Types paramétrés

```
type 'a t = 'a list

let encoding =
 fun _a_encoding -> list _a_encoding
```

```
type ('a, 'b) par_type2 =
 | D of 'a * 'b

let encoding_of_par_type2 =
 fun _a_encoding ->
  fun _b_encoding ->
   conv
    (fun (D (d0, d1)) -> (d0, d1))
    (fun (d0, d1) -> D (d0, d1))
    (obj1 (req "D"
     (tup2 _a_encoding _b_encoding)))
```

```
type ('a, 'b, 'c) par_type3 = K of 'a * 'b * 'c

let encoding_of_par_type3 =
 fun _a_encoding _b_encoding _c_encoding ->
  conv
   (fun (K (k0, k1, k2)) -> (k0, k1, k2))
   (fun (k0, k1, k2) -> K (k0, k1, k2))
   (obj1 (req "K"
    (tup3 _a_encoding _b_encoding _c_encoding)))
```

Les schémas d'encodage des types paramétriques sont préfixés avec une fonction qui prend en argument les encodages des types des paramètres qui sont ensuite utilisés aux endroits correspondants. Les nom des arguments commencent par un *underscore* afin assurer la compilation dans le cas où le paramètre de type n'est pas utilisé.
Afin d'utiliser cet encoding nous devons donc spécifier les encodage des arguments.
Soit un type :

```
type t = (int, string) par_type2
```

Son encoding aura la forme suivante :

```
t encoding = encoding_of_par_type2 int31 string
```

Chapitre 4

Conclusion

4.1 Résumé

Pour revenir aux problèmes abordés au 2.2.2 :

— L'outil que j'ai développé simplifie largement la définition des schémas d'encodage pour les types longs. Certains schémas ne peuvent pas être générés automatiquement, notamment quand on veut inclure des informations spécifiques dans le format *JSON*, ou produire des encodages binaires optimisés. Toutefois, l'outil permet de générer en quelques clics une première ébauche du schéma, qui peut être ensuite modifié suivant le besoin.

— Quand un type est mis à jour, la génération automatique des encodages assure la correspondance entre le type et l'encoding qui est mis à jour à chaque lancement du préprocesseur.
Cela étant dit, certaines parties sensibles de *Tezos* exigent que les préprocesseurs soient utilisés avec précaution, voire désactivés, car la modification de l'AST est un processus qui comporte des dangers d'un point de vue de sécurité. Ainsi, il est conseillé d'utiliser `[@@deriving_inline ...][@@@end]` pendant la phase de développement afin de

garder à jour la correspondance type-encoding et désactiver le préprocesseur dans la version *release*.
— Les encodages générés sont compatibles avec le format *JSON* ce qui résout le problème des constructeurs confondus dans les types algébriques.

4.2 Perspectives d'évolution.

Actuellement, l'outil prend en charge la plupart des types *OCaml* couramment utilisés. Cependant, certains cas ne sont pas gérés comme les *GADT* ainsi que les types mutuellement récursifs ce qui peut être inclut dans les versions postérieures.
De plus, afin de raffiner la forme des schémas d'encodage générés, il est possible d'ajouter le support des annotations d'un champ, ce qui permettrait de spécifier son encodage ou la valeur par défaut dans la version *JSON* par exemple.

4.3 Bilan pédagogique

Ayant que très peu d'expérience en *OCaml*, le développement d'un outil travaillant directement sur son *AST* m'a permis d'apprendre non seulement à programmer avec ce langage mais aussi de comprendre son fonctionnement interne.
À travers la création de *ppx_ encoding* j'ai pu découvrir les techniques modernes de développement en OCaml :
— Le gestionnaire des paquets *opam*.
— Le gestionnaire de projets *dune*.
— L'éditeur de code *merlin*.
Finalement, le suivi étroit du développement par mes (co-)encadrants à travers les *code-reviews* a élargi mes connaissances du style fonctionnel (la factorisation des fonctions, nommage des variables, utilisation efficace des bibliothèques standards et bien d'autres) ainsi que des bonnes pratiques de programmation.