Gestion de versions de formats avec Camlp4

Fabrice Le Fessant

Gestion de versions de formats avec Camlp4

Fabrice Le Fessant

Projet ASAP, INRIA-Futurs/LIX,
Ecole Polytechnique,
F-91128 Palaiseau Cedex, France
fabrice.le_fessant@inria.fr

Résumé

La gestion des changements de formats est un problème crucial de l'informatique : encore aujourd'hui, peu d'applications sont capables de lire des données sauvées dans un format d'une version antérieure ; en distribué, les versions de programmes devant communiquer entre eux sont de plus en plus hétérogènes, en particuliers dans les systèmes pair-à-pair.

Dans ce papier, nous présentons une extension de syntaxe d'Objective-Caml en Camlp4 permettant de gérer plus efficacement les évolutions de formats, aussi bien dans les fichiers sauvegardés que dans les messages échangés dans un système distribué.

1. Introduction

L'import et l'export de structures de données complexes sont devenus des fonctionnalités très utilisées des langages de programmation modernes. Nous appellerons dans la suite *emballeur* la fonction qui convertit une valeur en chaîne de caractères, et *déballeur* la fonction inverse qui convertit cette chaîne de caractères en une valeur. On parle souvent de *serialization* (Java, .net) ou *marshalling* (Objective-Caml). Nous ne traitons pas ici le cas des références distantes vers des objets, mais uniquement celui de données statiques.

Deux approches existent : certains langages proposent des emballeurs génériques pouvant traiter tous leurs types de données, tandis que d'autres fournissent des outils permettant de générer automatiquement des emballeurs spécifiques à chaque type de données. Dans les deux cas, nous ne connaissons pas d'implantations permettant de gérer correctement les changements de versions, tels que les modifications des types des données exportées, ou l'importation de données exportées par une version précédente du logiciel.

La gestion des versions de formats de données est pourtant un problème crucial dès que deux applications doivent communiquer des données, dans l'espace comme dans le temps. Ainsi, les systèmes massivement distribués actuels doivent de plus en plus souvent être capables de faire communiquer différentes versions d'un même logiciel entre elles : dans les systèmes pair-à-pair, il est souvent difficile de forcer l'ensemble des utilisateurs à utiliser la toute dernière version d'un client, et il n'est pas rare de trouver des versions anciennes de plusieurs années. La simplicité des protocoles a, jusque ici, permis d'éviter les problèmes d'incompatibilité, mais les applications devenant de plus en plus complexes, il est important de penser dès maintenant ces applications avec ce problème en tête.

De même, il n'est plus rare aujourd'hui qu'un utilisateur possède des fichiers importants qu'il a créés quelques années auparavant. Hors, pour assurer la pérennité des données, il est important que ces fichiers restent utilisables par les versions récentes de ces logiciels. L'application doit alors être capable d'identifier la version du format utilisé pour sauver le fichier, puis utiliser un lecteur adapté.

Il incombe donc aux développeurs de telles applications de maintenir en permanence des convertisseurs permettant de lire tous les précédents formats de données, puis de les convertir dans le format courant. Cette tache devient rapidement pénible à continuer à long terme.

Dans ce papier, nous présentons un environnement que nous utilisons dans nos applications en Objective-Caml[5] pour gérer quasi-automatiquement le problème de la gestion des versions de formats, aussi bien pour les fichiers sur disque que pour les protocoles distribués. Cet environnement s'appuie sur l'utilisation d'extensions de syntaxe Camlp4[1], pour définir le format des données à sauver (fichiers) ou transmettre (messages). Ces extensions définissent automatiquement des emballeurs/déballeurs, qui sont sauvegardés pour chaque version du programme. La modification du format des données impose alors simplement la modification de tous les anciens emballeurs/déballeurs.

Ce papier s'articule ainsi : nous commençons par donner un exemple d'évolution de format. Nous expliquons ensuite comment notre approche fournit des emballeurs/déballeurs évolutifs pour cet exemple. Nous discutons ensuite comment négocier la version de protocole à utiliser entre deux clients. Enfin, nous proposons une nouvelle méthode qui évite un certain nombre d'inconvénients de notre méthode.

2. Un exemple

Dans la suite, nous allons prendre un premier exemple simple : l'utilisateur désire développer un logiciel permettant de sauver un index contenant la liste des fichiers présents sur son disque. Cet index devra pouvoir être lu quelques mois, quelques années plus tard, par exemple pour la restauration du disque après une panne.

Nous définissons donc trois types `file_entry`, `dir_entry` and `index` :

```
type file_entry = {
    file_name : string;
    file_dir : string;
    file_size : int;
  }

type dir_entry = {
    dir_name : string;
    dir_dir : string;
  }

type index =
  File of file_entry
| Dir of dir_entry
```

Notre but est d'écrire un fichier contenant plusieurs de ces enregistrements :

```
let index = [
    File { file_name = "jfla.tex";
           file_dir = "articles/jfla2007";
           file_size = 1484 };
    Dir { dir_name = "jfla2007";
          dir_dir = "articles"; }
  ]
```

Avec les emballeurs/déballeurs classiques d'Objective-Caml, nous pouvons définir :

```
let save_index filename values =
  let oc = open_out "index.bin" in
  List.iter (fun v ->  output_value oc v) values;
  close_out oc
```

```
let load_index filename =
  let ic = open_in filename in
  let items = ref [] in
  try
    while true do
      items := input_value ic :: !items
    done
  with End_of_file ->
    close_in ic;
    List.rev !items
```

Les fonctions `save_index` et `load_index` nous permettent de sauver facilement une liste d'enregistrements, puis de la relire. Néanmoins, cette approche ne résoud pas le problème de la pérennité des données : un changement de version d'Objective-Caml peut rendre inutilisable les fichiers générés. L'utilisation d'emballeurs/déballeurs dédiés peut résoudre ce problème.

Mais une modification simple des types peut aussi à nouveau rendre ces fichiers obsolètes. Ainsi, la version 2 du logiciel redéfinit les types utilisés dans l'index : la taille des fichiers est maintenant sur 64 bits, la dernière date de modification est aussi archivée, et les liens symboliques sont supportés :

```
type file_entry = {
    file_name : string;
    file_dir : string;
        (* Gestion des grands fichiers: int devient int64 *)
    file_size : int64;
        (* Sauvegarde de la dernière date de modification: *)
    file_mtime : int;
  }

type dir_entry = {
    dir_name : string;
    dir_dir : string;
  }

type index =
| Link of string * string  (* Gestion des liens symboliques *)
| File of file_entry
| Dir of dir_entry
```

Ces modifications posent le problème de la gestion des versions : il devient nécessaire de conserver dans le fichier la version de l'emballeur utilisé pour sauver les données. L'application doit aussi être capable d'ajouter facilement les champs manquant, et de supporter des modifications de syntaxe dans les types.

Si cette tache parait relativement simple à effectuer sur des exemples simples, elle devient bien plus fastidieuse pour des protocoles complexes ou devant évoluer vite : ainsi, le protocole de communication entre le client pair-à-pair MLdonkey et ses interfaces graphiques a subit 63 évolutions officielles, résultant en 30 changements de numéro de version, les ajouts de nouveaux messages ne nécessitant pas toujours un tel changement.

Ainsi, le code suivant est celui du déballeur d'un enregistrement de fichier en téléchargement, et contient de nombreuses références à la version du protocole. Le développeur peut facilement y introduire des bugs, en particulier des incohérences de version avec l'emballeur associé.

```
let get_file version s pos =
  [.................................]
  let availability, pos =
    if version > 17 then
      get_list (fun s pos ->
          let net = get_int s pos in
          let avail, pos = get_string s (pos+4) in
          (net, avail), pos
      ) s pos
    else
    let avail, pos = get_string s pos in
    [net, avail], pos in
  [.................................]
  let name, pos = if version >= 8 then
      get_string s pos else List.hd names, pos in
  let last_seen, pos = if version >= 9 then
      get_int s pos, pos+4 else BasicSocket.last_time (), pos in
  let priority, pos = if version >= 12 then
      get_int s pos, pos+4 else 0, pos in
  let comment, pos = if version > 21 then get_string s pos
    else "", pos in
  [.................................]
  { ... }
```

Nous nous proposons donc d'apporter une première solution, partielle, à ce problème en générant automatiquement des emballeurs/déballeurs supportant facilement les changements de versions.

3. Notre approche

Notre approche consiste à utiliser Camlp4, le préprocesseur évolué incorporé à Objective-Caml, pour étendre la syntaxe d'Objective-Caml avec des macros générant automatiquement le code d'emballage/déballage des types devant être exportés. Le code ainsi généré est ajouté au code source de l'application, et pourra, par la suite, être modifié pour tenir compte des évolutions futures.

Notre système définit trois premières macros :

PROTOCOL *filename* :
: indique que les formats des données doivent être exportées dans les fichiers *filename*`.ml` et *filename*`Defs.ml`.

type DECLARE *typename* = *typedef* :
: enregistre la définition de type *typedef* associée avec le nom *typename*.

FORMAT_OF typename END :
: génère des fonctions élémentaires `buf_`*typename* (emballage du type dans un buffer), `get_`*typename* (déballage du type depuis une chaîne) et `string_of_`*typename* (affichage du type).

Si nous reprenons les définitions de types de notre exemple, le code devient :

```
PROTOCOL "jflaProto"

type DECLARE file_entry = {
    file_name : string;
    file_dir : string;
    file_size : int;
  }
type DECLARE dir_entry = {
    dir_name : string;
    dir_dir : string;
  }
type DECLARE index =
  File of file_entry
| Dir of dir_entry

FORMAT_OF file_entry END
FORMAT_OF dir_entry END
FORMAT_OF index END
```

Nous allons maintenant recoder les fonctions `save_index` et `load_index`, permettant de sauver et de lire un index dans un fichier :

```
module INDEX_FILE = JOURNAL_OF index END

let save_index filename index =
  let oc = open_out filename in
  output_string oc INDEX_FILE.message_of_protocol;
  List.iter (fun v ->
      output_string oc (INDEX_FILE.message_of_item v)
  ) index;
  close_out oc

let load_index filename =
  let list = ref [] in
  let _final_pos = INDEX_FILE.load
         (fun t -> list := t :: !list) filename in
  List.rev !list
```

Ce code définit, grace à la macro `JOURNAL_OF typename END`, un module `INDEX_FILE` dont la signature est :

```
val load : (index -> unit) -> string -> int64
val open_file_iterator : string -> unit -> index
val message_of_protocol : string
val message_of_item : index -> string
```

On peut remarquer, à ce stade, que ces macros permettent d'écrire un code très proche de celui utilisant les emballeurs/déballeurs automatiques d'Objective-Caml, et probablement identique à celui d'un emballeur/ déballeur sans gestion des versions de formats.

3.1. Fonctionnement

La principale difficulté de la gestion des versions est de conserver de façon pérenne dans le code de l'application les codes des emballeurs/déballeurs définis dans les versions précédentes.

Dans notre approche, cette tache est effectuée par le préprocesseur Camlp4, qui sauvegarde dans deux fichiers, d'une part les paramètres de configuration du format, et d'autre part, le code des emballeurs/déballeurs qu'il génère automatiquement. Dans notre exemple, il s'agit des fichiers `jflaProto.ml` pour la configuration et `jflaProtoDefs.ml` pour l'archivage des emballeurs/déballeurs, introduits par la macro `PROTOCOL`.

Les emballeurs/déballeurs sont archivés sous forme de code Objective-Caml : il est important de noter que tous nos emballeurs et déballeurs travaillent sur le type courant, i.e. tel qu'il est définit dans la version courante du logiciel. Lors d'un changement de version, il incombe donc au programmeur de modifier tous les emballeurs/déballeurs précédents pour qu'ils puissent fonctionner avec la nouvelle définition du type. Nous verrons dans la partie 5 comment dépasser cette limitation.

3.2. Génération des emballeurs et déballeurs

L'emballeur est une fonction prenant en arguments le numéro de version à utiliser, un tampon de type `Buffer.t`, et une donnée à emballer. Le code généré pour l'emballeur du type `file_entry` est le suivant :

```
let buf_File_entry_MB0006B706FF0D9E24A72492E74005E64
  version b
    { file_name = file_name; file_dir = file_dir;
      file_size = file_size} =
  buf_string version b file_name;
  buf_string version b file_dir;
  buf_int version b file_size
let buf_file_entry version b x =
  if version >= 0 then
    buf_File_entry_MB0006B706FF0D9E24A72492E74005E64 version b x
  else raise ProtocolFailure
```

On peut remarquer que le numéro de version est transmis à tous les emballeurs appelés récursivement, évitant ainsi de devoir regénérer un emballeur systèmatiquement pour un type dès que le format d'un type dont il dépend change.

Le déballeur est un fonction prenant en arguments une chaîne de caractères et une position dans cette chaîne, et retournant un couple contenant la donnée extraite et la position suivante à lire dans la chaîne. Le code généré pour `file_entry` est le suivant :

```
let get_File_entry_MB0006B706FF0D9E24A72492E74005E64 version s pos =
  let (file_name, pos) = get_string version s pos in
  let (file_dir, pos) = get_string version s pos in
  let (file_size, pos) = get_int version s pos in
  { file_name = file_name; file_dir = file_dir;
    file_size = file_size}, pos
let get_file_entry version s pos =
  if version >= 0 then
    get_File_entry_MB0006B706FF0D9E24A72492E74005E64 version s pos
  else raise ProtocolFailure
```

Ce code s'obtient de manière relativement simple : les déclarations de type sont sauvegardées

lors de l'analyse syntaxique des macros `type DECLARE typename = ...` dans une table. Lorsqu'un `FORMAT_OF type` apparaît, la déclaration est extraite de la table, et utilisée pour générer les emballeurs/déballeurs correspondant. L'utilisation de la macro `DECLARE` est nécessaire, car Camlp4 intervient sur l'arbre de syntaxe non typé.

Notons que ce code s'appuie sur l'existence d'emballeurs/déballeurs prédéfinis pour les types de base (`int, string, float, array, list`) et utilise les dénominations `buf_`*`type`* et `get_`*`type`* pour les types abstraits.

3.3. Le fichier de configuration

Dans l'exemple précédent, pour effectuer un changement de version et modifier les définitions des types, le développeur doit d'abord éditer le fichier `jflaProto.ml`, généré et tenu à jour par Camlp4, dont voici le contenu :

```
let min_version = 0
let current_version = 0
let open_version = 0
let codes_IndexFile = [| "CA07D78626E767E59EC5AE847320E45B" |]
```

Ce fichier *de configuration* permet de spécifier les versions minimales, courantes et de développement du protocoles, ainsi que d'obtenir les identifiants des différentes versions de protocoles.

`min_version` est la version minimale du protocole supportée par le logiciel, `current_version` est la version courante du protocole utilisée par le logiciel pour sauver ses données, tandis que `open_version` est la version de développement, dont les emballeurs/déballeurs ne sont pas encore fixés, et sont donc modifiés à chaque modification des types.

Ainsi, lorsqu'une nouvelle distribution du logiciel est fournie, il est important de fixer `open_version > current_version`, afin d'empécher par erreur la modification d'emballeurs/déballeurs déjà utilisés par les utilisateurs. Pour garantir cela, une option de développement `-testproto` doit être fournie à l'exécution quand l'assertion précédente n'est pas vérifiée. De plus, il est possible de fournir au macro-processeur une option permettant de "fermer" tous les protocoles, afin d'obtenir les sources d'une nouvelle distribution du logiciel.

La dernière ligne de `jflaProto.ml` définit un tableau contenant des identifiants de protocoles : chaque numéro de version est identifié par le SHA1 du code de tous les emballeurs/déballeurs dont il dépend.

Pour effectuer sa modification des types tout en permettant de lire l'ancienne version des fichiers, le développeur fixe dont `open_version` et `current_version` à 1.

3.4. Le fichier d'archivage des convertisseurs

Les emballeurs/déballeurs sont stockés tous ensemble dans un fichier source Objective-Caml. Un module est défini pour chaque type, puis dans ce module, un sous-module correspond à chaque version.

```
module File_entry = struct
    module MB0006B706FF0D9E24A72492E74005E64 = struct
        let buf_File_entry_MB0006B706FF0D9E24A72492E74005E64
          version b
            {file_name = file_name; file_dir = file_dir; file_size = file_size} =
          buf_string version b file_name;
          buf_string version b file_dir;
          buf_int version b file_size
        let get_File_entry_MB0006B706FF0D9E24A72492E74005E64 version s pos =
          let (file_name, pos) = get_string version s pos in
          let (file_dir, pos) = get_string version s pos in
          let (file_size, pos) = get_int version s pos in
          {file_name = file_name; file_dir = file_dir; file_size = file_size },
          pos
      end
    let MB0006B706FF0D9E24A72492E74005E64 = 0
    let depend = Int
    let depend = String
    let depend = Float
  end
```

Dans chaque sous-module sont définis l'emballeur et le déballeur correspondant à cette version. Leur code est utilisé pour calculer une somme de contrôle SHA1, qui donne son nom au module.

```
module Index = struct
    module M983723F8B93B506748C2D03984A8A64A = struct
        let buf_Index_M983723F8B93B506748C2D03984A8A64A version b content =
          match content with
          | File camlp4_arg_1 ->
              buf_int8 version b 0; buf_file_entry version b camlp4_arg_1
          | Dir camlp4_arg_2 ->
              buf_int8 version b 1; buf_dir_entry version b camlp4_arg_2
        let get_Index_M983723F8B93B506748C2D03984A8A64A version s pos =
          let (content, pos) =
            let (camlp4_opcode_5, pos) = get_uint8 s pos in
            match camlp4_opcode_5 with
              0 ->
                let (camlp4_arg_4, pos) = get_file_entry version s pos in
                File camlp4_arg_4, pos
            | 1 ->
                let (camlp4_arg_3, pos) = get_dir_entry version s pos in
                Dir camlp4_arg_3, pos
            | n -> failwith (Printf.sprintf "Bad␣tag␣%d␣in␣enum␣index" n)
          in
          content, pos
      end
    let M983723F8B93B506748C2D03984A8A64A = 0
    let depend = Dir_entry
    let depend = File_entry
    let depend = String
  end
```

Le module correspondant à un type contient aussi d'autres informations : la version minimale correspondant à un identifiant de version, et des dépendances.

Les dépendances sont utilisées pour calculer l'identifiant du protocole : celui-ci ne dépend pas uniquement des emballeurs/déballeurs des messages, mais aussi de ceux des types dont ils dépendent.

Les dépendances permettent donc d'inclure dans le calcul tous les types devant influer sur l'identifiant final du protocole.

```
module File_entry = struct
    module MB0006B706FF0D9E24A72492E74005E64 = struct
        let buf_File_entry_MB0006B706FF0D9E24A72492E74005E64
          version b
            {file_name = file_name;
             file_dir = file_dir;
             file_size = file_size} =
          buf_string version b file_name;
          buf_string version b file_dir;
          buf_int version b (Int64.to_int file_size)
        let get_File_entry_MB0006B706FF0D9E24A72492E74005E64 version s pos =
          let (file_name, pos) = get_string version s pos in
          let (file_dir, pos) = get_string version s pos in
          let (file_size, pos) = get_int64 version s pos in
          {file_name = file_name; file_dir = file_dir; file_size = file_size;
           file_mtime = 0},
          pos
      end
    module MF4C3866CCDC84748736E923469E271F3 = struct
        let buf_File_entry_MF4C3866CCDC84748736E923469E271F3
          version b
            {file_name = file_name; file_dir = file_dir;
             file_size = file_size; file_mtime = file_mtime} =
          buf_string version b file_name;
          buf_string version b file_dir;
          buf_int64 version b file_size;
          buf_int version b file_mtime
        let get_File_entry_MF4C3866CCDC84748736E923469E271F3 version s pos =
          let (file_name, pos) = get_string version s pos in
          let (file_dir, pos) = get_string version s pos in
          let (file_size, pos) = get_int64 version s pos in
          let (file_mtime, pos) = get_int version s pos in
          {file_name = file_name; file_dir = file_dir; file_size = file_size;
           file_mtime = file_mtime}, pos
      end
    let MB0006B706FF0D9E24A72492E74005E64 = 0
    let MF4C3866CCDC84748736E923469E271F3 = 1
    let depend = Int
    let depend = String
    let depend = Float
    let depend = Int64
  end
```

Le code ci-dessus correspond au code présent dans le fichier d'archivage après les modifications des types de l'exemple de la section 2. Un deuxième module correspondant à la seconde version du protocole est présent, tandis que nous avons mis en italique les modifications que le programmeur a dû apporter au code de la version précédente.

```
module Index = struct
    module M983723F8B93B506748C2D03984A8A64A = struct
        let buf_Index_M983723F8B93B506748C2D03984A8A64A version b content =
          match content with
            Link _ -> failwith "Link didn't exist in that version"
          | File camlp4_arg_1 ->
              buf_int8 version b 0; buf_file_entry version b camlp4_arg_1
          | Dir camlp4_arg_2 ->
              buf_int8 version b 1; buf_dir_entry version b camlp4_arg_2
        let get_Index_M983723F8B93B506748C2D03984A8A64A version s pos =
          let (content, pos) =
            let (camlp4_opcode_5, pos) = get_uint8 s pos in
            match camlp4_opcode_5 with
              0 ->
                let (camlp4_arg_4, pos) = get_file_entry version s pos in
                File camlp4_arg_4, pos
            | 1 ->
                let (camlp4_arg_3, pos) = get_dir_entry version s pos in
                Dir camlp4_arg_3, pos
            | n -> failwith (Printf.sprintf "Bad tag %d in enum index" n)
          in
          content, pos
      end
    module M0458EE1C7AE3A9934EC4A549101B0907 = struct
        let buf_Index_M0458EE1C7AE3A9934EC4A549101B0907 version b content =
          match content with
            Link (camlp4_arg_1, camlp4_arg_2) ->
              buf_int8 version b 0; buf_string version b camlp4_arg_1;
              buf_string version b camlp4_arg_2
          | File camlp4_arg_3 ->
              buf_int8 version b 1; buf_file_entry version b camlp4_arg_3
          | Dir camlp4_arg_4 ->
              buf_int8 version b 2; buf_dir_entry version b camlp4_arg_4
        let get_Index_M0458EE1C7AE3A9934EC4A549101B0907 version s pos =
          let (content, pos) =
            let (camlp4_opcode_9, pos) = get_uint8 s pos in
            match camlp4_opcode_9 with
              0 ->
                let (camlp4_arg_7, pos) = get_string version s pos in
                let (camlp4_arg_8, pos) = get_string version s pos in
                Link (camlp4_arg_7, camlp4_arg_8), pos
            | 1 ->
                let (camlp4_arg_6, pos) = get_file_entry version s pos in
                File camlp4_arg_6, pos
            | 2 ->
                let (camlp4_arg_5, pos) = get_dir_entry version s pos in
                Dir camlp4_arg_5, pos
            | n -> failwith (Printf.sprintf "Bad tag %d in enum index" n)
          in
          content, pos
      end
    let M983723F8B93B506748C2D03984A8A64A = 0
    let M0458EE1C7AE3A9934EC4A549101B0907 = 1
    let depend = Dir_entry
    let depend = File_entry
    let depend = String
  end
```

Dans le cas du type `index` ci-dessus, les modifications sont légèrement plus simples. Remarquons néanmoins qu'en l'absence de ces modifications, le compilateur produirait des erreurs lors de l'insertion de ce code, et que ces erreurs permettent au programmeur de rapidement corriger les anciennes fonctions pour tenir compte des modifications apportées au nouveau type de données.

La structuration en modules n'est présente que pour organiser le fichier d'archivage, et aider le programmeur à s'y retrouver quand il doit modifier les emballeurs/déballeurs. En effet, les fonctions d'emballage/déballage sont incluses dans le code source à l'appel de la macro `FORMAT_OF` sans cette structure de modules, ce qui explique pourquoi elles sont suffixées par le nom du module. D'autre part, Camlp4 ne permet pas d'étendre l'arbre de syntaxe utilisé par Ocaml, et il n'est donc pas possible d'ajouter une syntaxe particulière à notre problème. La structuration en module permet donc de relire le fichier plus facilement et d'en extraire les informations qui devront être insérées dans le source final.

Finalement, `FORMAT_OF` insère après le code des emballeurs/déballeurs le code permettant de choisir en fonction des versions les bonnes fonctions à appeler, grâce aux numéros de versions indiqués dans le fichier d'archivage. Pour le type `index`, on obtient :

```
let buf_index version b x =
  if version >= 1 then buf_Index_M0458EE1C7AE3A9934EC4A549101B0907 version b x
  else if version >= 0 then
    buf_Index_M983723F8B93B506748C2D03984A8A64A version b x
  else raise ProtocolFailure
let get_index version s pos =
  if version >= 1 then
    get_Index_M0458EE1C7AE3A9934EC4A549101B0907 version s pos
  else if version >= 0 then
    get_Index_M983723F8B93B506748C2D03984A8A64A version s pos
  else raise ProtocolFailure
```

Cette approche résoud le problème qui nous est posé, mais souffre d'un problème majeure : à chaque modification d'un type, il est nécessaire de modifier tous les emballeurs/déballeurs définis pour les versions précédentes du type. Nous proposons dans la section 5 une extension de notre approche qui résoud ce problème.

4. Négociation des protocoles

Actuellement, la plupart des systèmes que nous connaissons supportant plusieurs versions de formats utilisent des numéros de version pour reconnaître les formats des fichiers. Ce modèle devient inadapté dans certains cas :

- De nombreux projets travaillent sur des versions concurrentes de certains Logiciels Libres. Ainsi, les logiciels Emule et Amule sont des projets distincts, issus des sources du premier. Ces deux logiciels doivent pourtant souvent communiquer entre eux sur le réseau Edonkey. Hors, il est possible que les numéros de version divergent, suite à des évolutions concurrentes, alors que des synchronisations entre les projets vont permettre de voir apparaître de temps en temps des protocoles identiques.
- Pour éviter de maintenir trop de versions de formats, il peut être intéressant de ne conserver que certaines versions des formats, plus stables par exemple, et de ne plus gérer les versions intermédiaires.

Notre approche qui consiste à utiliser des identifiants (ici, des sommes de contrôles) s'adapte plus aisément à ces situations : la somme de contrôle marque chaque version du protocole disponible dans le logiciel, puisqu'elle est obtenue à partir des sources des emballeurs/déballeurs de cette version. Bien-sûr, une approche plus générique devrait être utilisée pour supporter plusieurs langages, par exemple à base d'un langage de description des formats emballés (ASN.1[3]).

L'identifiant est sauvé dans l'entête des fichiers, ce qui permet au logiciel de tester s'il est capable de lire le fichier, et de connaître immédiatement le numéro de version interne lui correspondant. De même, si des ajouts peuvent être faits dans un fichier déjà créé, il est possible d'insérer l'identifiant du nouveau protocole avant un lot d'enregistrements pour changer dynamiquement le déballeur qui sera utiliser pour lire ces enregistrements.

Pour des logiciels communicants, la négociation peut être un peu plus longue : le client se connectant envoie une petite liste d'identifiants correspondant aux versions les plus récentes, le serveur retourne alors l'identifiant de la version la plus haute qu'il peut utiliser. Si un tel identifiant n'existe pas, le processus est itéré sur tous les identifiants précédents par petits paquets. Le client peut conservé le numéro de protocole choisi par le serveur d'une session à l'autre, afin de l'insérer dans le premier paquet lors des négociations suivantes, donnant ainsi au serveur le choix entre les versions les plus récentes du protocole et la dernière version du protocole déjà utilisée entre eux, dès la première itération de la négociation.

5. Optimisation des modifications

Lors du changement de la définition d'un type, il peut s'avérer fastidieux de mettre à jour l'ensemble des emballeurs/déballeurs générés précédemment pour ce type. Pour simplifier cette tache, nous proposons d'utiliser une technique permettant de réutiliser, quand cela est possible, les emballeurs/déballeurs générés précédemment systématiquement, plutôt que de générer de nouveaux emballeurs/déballeurs complets.

Dans cette section, nous allons examiner plus spécifiquement le cas des enregistrements :

```
type t = {
  field_1 : type_1;
  field_2 : type_2;
  ...
  field_n : type_n;
}
```

Les modifications de type sur un enregistrement sont de trois types :
- Ajout d'un nouveau champ : `ADD ( field`$_{n+1}$ `: type`$_{n+1}$ `)`
- Suppression d'un ancien champ : `DEL ( field`$_i$ `: type`$_i$ `)`, pour $i \in [1..n]$
- Modification d'un ancien champ : `MOD ( field`$_i$ `: type`$_i$`' )`, pour $i \in [1..n]$

Le renommage d'un champ peut-être vu comme sa suppression, puis l'ajout d'un champ portant le nouveau nom.

L'idée principale est de séparer les emballeurs/déballeurs des différents champs : au lieu de contenir les deux fonctions d'emballage/déballage de la section 3, chaque module de version contient les emballeurs/déballeurs de chaque champs qui a été modifié à ce changement de version.

La fonction de déballage générée pour le type a alors la forme finale suivante :

```
let get_t version s pos =
  let _field_j, pos = ..... in
  ...
  {
    field_i = _field_i;
    ...
  }
```

où chaque `field`$_j$, $j \in [1..n]$ est un champ qui a été défini dans le type depuis la définition du type. Seuls les champs présents dans le type courant sont utilisés par la suite dans la définition de

l'enregistrement (variable i). Toutes les variables sont préfixées d'un _ pour éviter les avertissements du compilateur s'ils ne sont pas utilisés.

Pour chaque champ, des tests du paramètre `version` permettent de choisir quel déballeur utiliser pour ce champ, indépendemment des autres. Les déballeurs/emballeurs ne sont définis pour un champ que si celui-ci a été modifié à cette version, ce qui diminue considérablement le travail de modification pour le développeur : quand il modifie un champ, il ne doit modifier que les emballeurs/déballeurs associés à ce champ.

Pour connaître quel champs ont été modifiés à chaque version, il est nécessaire de conserver la liste des champs ajoutés, modifiés et supprimés : ceci se fait simplement avec Camlp4 en ajoutant au sous-module de la version trois variables `add`, `mod` et `del`, comme dans l'exemple suivant :

```
module T = struct
  module M574320940A998BCD7879 = struct
    ...
    let del = ( field1, field4, field_7 )
    let mod = ( field5 )
    let add = ( field8, field9 )
  end
end
```

Les trois opérations de modification décrites sur le type enregistrement demandent alors le travail suivant :

- Ajout d'un nouveau champ : le préprocesseur génère automatiquement les emballeurs/déballeurs de ce champ pour la version courante, et un emballeur vide pour la version initiale. Le développeur doit cependant définir le déballeur de la version initiale pour qu'il retourne une constante de type compatible.
- Suppression d'un ancien champ : un emballeur vide est généré automatiquement. Le développeur doit définir un déballeur comme dans le cas précédent, mais pour la version courante cette fois.
- Modification du type d'un champ : le préprocesseur génère automatiquement les emballeurs/-déballeurs de ce champ pour la version courante. Les développeur doit cependant modifier tous les emballeurs/déballeurs précédents de ce champ pour qu'ils travaillent sur le nouveau type du champ.

Cette technique permet de réduire considérablement le travail du développeur pour rendre les versions compatibles, car il n'a besoin de modifier que le code des versions où un champ qu'il modifie avait été modifié. Enfin, cette technique s'étend directement au cas des types sommes, les enregistrements et les types sommes représentant généralement les seuls constructeurs de structures complexes en Objective-Caml.

6. Travaux proches

Peu de travaux existent dans le domaine du support à l'évolution des systèmes. [2] signale l'absence de solution à ce problème mais ne propose pas de solution réelle.

La documentation de Java[4] contient une section traitant du changement de version, mais cette section explique uniquement sous quelles conditions les classes emballées pourront être relues. Aucun mécanisme n'est fourni pour gérer la lecture d'anciennes structures de données. Par ailleurs, un identifiant est aussi utilisé pour les versions des classes, mais sa gestion n'est pas automatisée.

7. Conclusion

Dans cet article, nous avons présenté une ébauche de solution pour la gestion des versions de formats, utilisant le préprocesseur évolué Camlp4. Le problème de la gestion des versions de formats est un problème de plus en plus présent dans les systèmes informatiques modernes, à la fois par l'évolution extrêmement rapide des logiciels et par leur processus de développement de plus en plus parallèlisé. Ce problème devient crucial dans les systèmes pair-à-pair, où des clients de versions et d'origines très diverses doivent pouvoir communiquer ensemble.

L'originalité de notre approche réside dans :
- l'utilisation d'identifiants généré automatiquement à la place des numéros de version.
- la sauvegarde sous forme de sources Objective-Caml des emballeurs/déballeurs générés automatiquement.

Ces deux points permettent de supporter plus facilement l'environement de notre projet, i.e. un Logiciel Libre pour les réseaux pair-à-pair. L'utilisation d'identifiants permet de pouvoir développer plusieurs branches du projet avec des compteurs de numéros de version séparés. Ces mêmes identifiants augmentent aussi la pérénité des données sauvegardées dans des fichiers, puisque notre logiciel peut relire tous les formats précédents. Enfin, la sauvegarde des emballeurs sous forme de sources permet de pouvoir les modifier aisément pour tenir compte de l'évolution du format interne des structures de données manipulées par l'application.

Nous travaillons maintenant à améliorer notre approche sur divers points tels que :
- la conservation des descriptions de type, afin de générer automatiquement une partie du code à modifier entre deux versions. Ce travail vient finaliser la proposition d'amélioration décrite en section 5.
- la séparation entre protocole principale et extensions, afin de permettre à deux applications de s'accorder sur un protocole principal, tous en acceptant des divergences sur les extensions.
- l'utilisation d'identifiants auto-générés indépendant du langage utilisé.

Références

[1] Daniel de Rauglaudre. Camlp4. http ://caml.inria.fr/camlp4, 2002.

[2] Huw Evans. Why Is Distributed System Evolution Not Better Supported ? In *International Workshop on the Principals of Software Evolution (IWPSE)*, colocated with ESEC/FSE, Vienna, Austria, September 2001.

[3] International Telecommunication Union. Abstract syntax notation one (ASN.1) : specification of basic notation. Recommendation X.680, Telecommunication Standardization Sector of ITU, Geneva, Switzerland, December 1997.

[4] Javasoft. Java object serialization specification. Technical report, 1999.

[5] Xavier Leroy, Jérôme Vouillon, Damien Doligez, et al. The Objective Caml system. Available on the web, 1996.