Trouver automatiquement des fichiers polyglots

David DAILY

I. Présentation du sujet

I.1. Les fichiers polyglots

I.1.a. Principe général des polyglots

Tout comme les polyglottes sont des personnes capables de communiquer dans plusieurs langues, les fichier polyglots sont des fichiers qui peuvent « communiquer » avec la machine sous différentes approches, c'est-à-dire fonctionner aussi bien comme un fichier d'un certain type T que comme un fichier de type T'. Ces fichiers peuvent être construits grâce à l'exploitation judicieuse de la façon dont est défini leur format, notamment pour trouver des façons d'insérer une suite binaire là où un lecteur/interpréteur de fichiers de type T ne regardera pas ceux-ci alors qu'un lecteur/interpréteur de fichiers T' pourra le lire sans être perturbé par la présence des chaines binaires du fichier T.

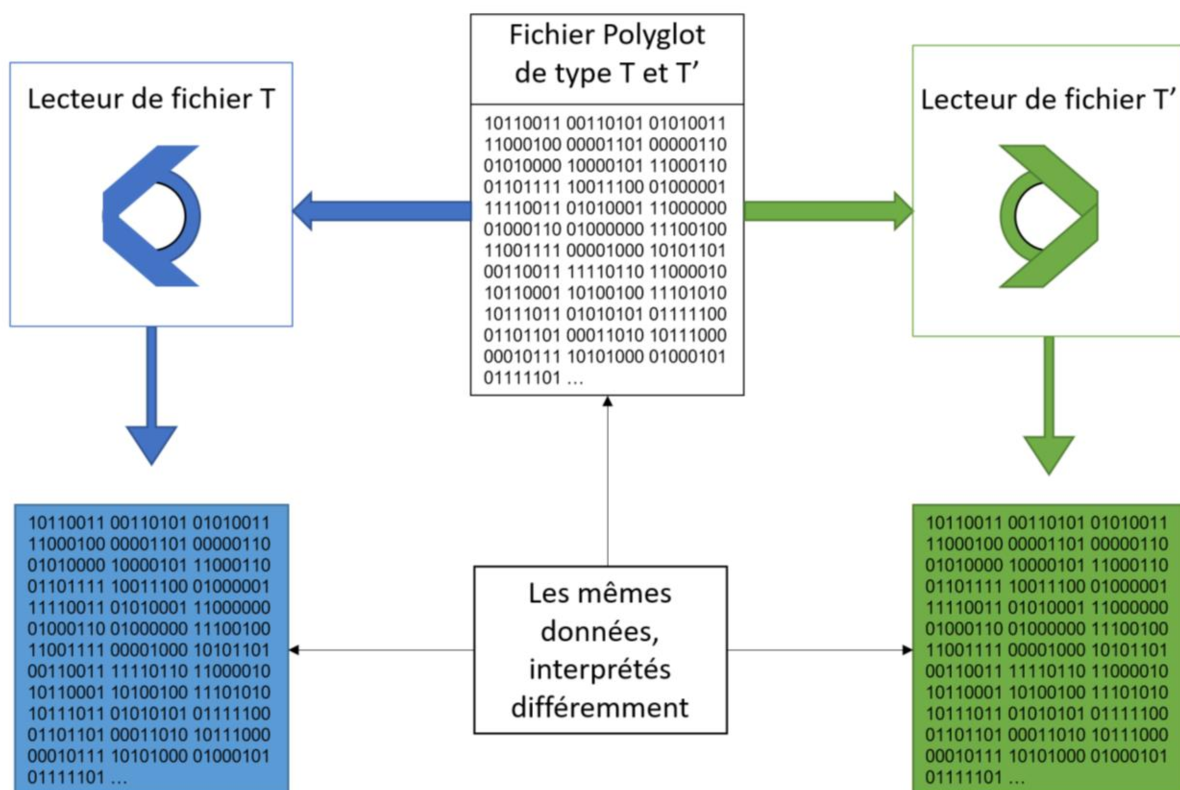

Figure 1 : Schéma explicatif d'un fichier polyglot

Au niveau du binaire, les fichiers ont généralement un en-tête qui permet à l'OS de reconnaître le type du fichier et indiquer les emplacements mémoire à lire pour trouver les informations inhérentes au fichier. Cet en-tête peut être placé au début de la chaine binaire (par exemple, les fichiers .ico, .png, .wav ont des en-têtes en début de fichier), ou à la fin (les fichiers .zip ont l'index des fichiers de l'archive à la fin du fichier .zip). Cela peut nous permettre d'insérer du code après la fin des données utiles du premier type de fichier, qui pourra être référencé par la deuxième « en-tête » en fin de fichier. Ainsi, le fichier pourra être reconnu et parsé comme 2 types de fichiers différents.

Toutefois, le fonctionnement des fichiers polyglots dépend aussi des programmes qui les interprètent. Par exemple, ces fichiers peuvent être détectés comme corrompus. À ce moment-là, le programme peut rejeter le fichier ou tenter de récupérer les données utilisables.

II. Objectif du projet

En ce moment, les fichiers polyglots sont un domaine relativement peu étudié, où les fichiers produits sont souvent créés de manière « artisanale, » par une construction à la main du développeur qui l'a écrit. Nous nous sommes fixé l'objectif d'avancer dans le domaine et de tenter de proposer une méthode automatique pour détecter des formats de fichiers « compatibles » à être entremêlés et exploités pour créer des polyglots. Dans cette optique, nous avons cherché à concevoir une méthode de représentation logique des contraintes indiqués dans un format de fichiers dans une logique satisfiable modulo théories (SMT), afin de pouvoir utiliser un solveur SMT pour tenter créer un modèle qui satisfait plusieurs formats en même temps.

II.1. La Satisfiabilité Modulo Théories

La Satisfiabilité Modulo Théories est une extension de la logique du premier ordre qui considère ses formules par rapport à des théories, qui définissent certaines actions admissibles (ou fonctions), ou encore des symboles. Par exemple, la théorie des nombres entiers définira entre autres la négation, la soustraction, l'addition, la multiplication, etc. Une formule SMT est donc une formule logique où les « variables booléennes » sont en fait des formules atomiques de la théorie choisie.

Par exemple, dans la théorie des chaines de charactères Unicode où la fonction « at(String s, Int i) » est défini comme la fonction renvoyant le caractère à la position i, la formule ***at(« Hello World », 0) = « H »*** peut être évaluée, et trouvée satisfiable dans ce cas. On peut également utiliser une variable ***at(« Hello World », 0) = x*** pour savoir quelles sont les valeurs de x qui permettrait de rendre vraie cette formule.

Il serait possible d'écrire les contraintes d'un format de fichier sous la forme d'un ensemble de formules SMT qui pourrait ensuite être appliqués sur un cas concret, c'est-à-dire le binaire d'un fichier. Avec les contraintes de 2 formats de fichiers encodés, ce serait possible de résoudre si les formats sont satisfiables en même temps, et éventuellement générer un modèle valide.

```
(declare-const x String)
(assert (= (str.at "Hello World" 0) x))

(declare-const y Int)
(assert (= (* y y) 4))

(check-sat)
(get-model)
```

```
sat
[y = -2, x = "H"]
```

Figure 2 : Exemple simple de formules satisfiables, ainsi qu'un modèle trouvé par z3

III. Les outils

z3 Theorem Prover est un solveur SMT multi-plateforme développé et distribué par Microsoft pour la recherche. Il permet notamment d'exploiter des fonctions sur les chaines de caractères, ce qui nous permet d'encoder le binaire brut d'un fichier donné, qu'on peut ensuite analyser avec les contraintes qu'on définit en fonction du format de fichier décrit dans Kaitai. Nous avons choisi d'écrire nos programmes SMT avec des fichiers smtlib, un format de fichier standardisé pour les solveurs SMT.

Pour générer ces représentations logiques, nous avions besoin d'une façon automatique de générer une représentation logique à partir d'un fichier décrivant un format de fichier, ce qui implique bien sûr d'avoir une représentation cohérente des formats de fichier.

Heureusement, l'outil Kaitai Struct permet de représenter les formats de fichiers binaires et en plus de générer le code d'un parseur d'un fichier de ce type. Par exemple, avec Kaitai, on peut définir la structure nécessaire à une chaine binaire pour être interprété comme un .zip, puis utiliser le compilateur Kaitai pour générer du code Java, C++, Python qui saura lire ce format de fichier et rendre exploitable les données dans un autre programme. Kaitai Struct est open-source, ce qui nous permet d'envisager d'écrire une extension pour Kaitai qui nous permettrait de générer un « programme » SMT que l'on pourrait exploiter dans notre solveur pour analyser des fichiers potentiellement polyglots.

formats/archive/zip.ksy

```
meta:
  id: zip
  title: ZIP archive file
  file-extension: zip
  xref:
    iso: 21320-1
    justsolve: ZIP
    loc:
      - fdd000354
      - fdd000355
      - fdd000362
      - fdd000361
    pronom: x-fmt/263
    wikidata: Q136218
  endian: le
  license: CC0-1.0
doc: |
  ZIP is a popular archive file format, introduced in 1989 by Phil Katz
  and originally implemented in PKZIP utility by PKWARE.

  Thanks to solid support of it in most desktop environments and
  operating systems, and algorithms / specs availability in public
  domain, it quickly became tool of choice for implementing file
  containers.

  For example, Java .jar files, OpenDocument, Office Open XML, EPUB files
  are actually ZIP archives.
doc-ref: https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT
seq:
  - id: sections
    type: pk_section
    repeat: eos
types:
  pk_section:
    seq:
      - id: magic
        contents: 'PK'
      - id: section_type
        type: u2
      - id: body
        type:
          switch-on: section_type
          cases:
            0x0201: central_dir_entry
            0x0403: local_file
            0x0605: end_of_central_dir
            0x0807: data_descriptor
  data_descriptor:
    seq:
      - id: crc32
```

Figure 3 : Un extrait de fichier Kaitai (.ksy) décrivant la structure d'une archive .zip

Le compilateur Kaitai est écrit en Scala, et permet de générer des programmes capables de parser un fichier binaire défini dans un fichier Kaitai .ksy dans du C++/STL, C#, Go, Java, JavaScript, Nim, Perl, PHP, Python et Ruby. La première partie du projet a été consacrée à la prise en main de Scala, puisque je n'avais pas encore utilisé cette langue auparavant, afin de me préparer pour coder une extension au compilateur.

IV. Les difficultés

Ce projet a été proie à plusieurs difficultés. Dans un premier temps, par suite d'un malentendu, le lancement du projet a été retardé de plusieurs semaines, ce qui a grandement réduit les possibilités de réalisation envisageables de manière réaliste. Toutefois, nous avons cherché à aller le plus loin possible dans le temps disponible.

Le domaine des fichiers polyglots est assez éloigné du programme du master, et la complexité du sujet a mené à un peu de confusion en début de projet sur les étapes qu'on cherchait à accomplir : notamment, j'ai passé une semaine de travail à réfléchir longuement comment compiler directement un fichier binaire lambda en code SMT, au lieu de penser à comment créer le parseur de fichiers parce que j'avais confondu les capacités de Kaitai Struct.

Après la prise en main du langage Scala, nécessaire pour développer une extension au compilateur Kaitai Struct, je suis passé à l'écriture de fichiers SMT pour prendre en main les différentes fonctionnalités du solveur et jouer avec les formats de fichiers et les fichiers polyglots.

Initialement, on avait prévu d'utiliser soit les bitvectors, des tableaux de bits qui sembleraient à première vue idéals pour stocker et exploiter des fichiers binaires. Cependant, les opérations de concaténation et surtout de subdivision de ces tableaux ne sont pas permises sur les bitvectors dans le format smtlib. Puisqu'il est essentiel de pouvoir découper la chaine binaire brute dans les sous-parties correspondant aux différents champs du fichier définit dans leur format, on a dû abandonner la voie des bitvectors et se tournes vers les chaines de caractères.

Les chaines de caractères dans smtlib sont des chaines Unicode, et proposent quelques fonctions utiles pour nos objectifs. Substr permet de générer des sous-chaines à partir d'une chaine source, et char_at permet de relever le caractère à une position arbitraire. Il existe aussi une fonction pour lire une chaine et proposer un entier. En revanche, il n'existe pas de fonction permettant de parser une chaine, de considérer que c'est un nombre en représentation binaire et de calculer sa valeur. Cela signifie que chaque mot binaire doit être interprété bit par bit, et octet par octet, en considérant correctement le boutisme selon le fichier (cf. figure 4).

```
71  (assert
72      (= img_count_int
73          (+
74              (+ (* 256 img_count_bit15)
75                  (+ (* 512 img_count_bit14)
76                      (+ (* 1024 img_count_bit13)
77                          (+ (* 2048 img_count_bit12)
78                              (+ (* 4096 img_count_bit11)
79                                  (+ (* 8192 img_count_bit10)
80                                      (+ (* 16384 img_count_bit9) (* 32768 img_count_bit8))
81                                  )
82                              )
83                          )
84                      )
85                  )
86              )
87              (+ (* 128 img_count_bit0)
88                  (+ (* 64 img_count_bit1)
89                      (+ (* 32 img_count_bit2)
90                          (+ (* 16 img_count_bit3)
91                              (+ (* 8 img_count_bit4)
92                                  (+ (* 4 img_count_bit5)
93                                      (+ (* 2 img_count_bit6) img_count_bit7)
94                                  )
95                              )
96                          )
97                      )
98                  )
99              )
100         )
101     )
102 )
```

Figure 4 : Expression SMT-LIB qui calcule la valeur entière d'un mot binaire petit-boutiste de longueur 2 octets

Ce problème serait moindre lorsqu'on travaillerait avec du code généré par l'extension du compilateur Kaitai, mais pour la prise en main manuelle qui précédait cette étape du projet, cela a représenté un obstacle à la progression rapide.

L'autre difficulté majeure du travail sur la représentation SMT est la gestion des champs répétés. Dans un grand nombre de fichiers, il y a des éléments internes aux fichiers qui sont répétés plusieurs fois. Les exemples les plus simples à comprendre sont les fichiers d'archive comme les .zip, qui contiennent la représentation d'un nombre variable d'autres fichiers. La description de ces fichiers dans l'en-tête binaire du .zip ne variera pas, mais le nombre d'occurrence de celle-ci va varier. Il faut donc pouvoir vérifier que chaque élément répété correspond effectivement au format défini.

Pour ce faire, nous avons cherché à exploiter les tableaux dans SMT. Les tableaux dans SMT sont plutôt des fonctions permettant d'enregistrer certaines valeurs et de les lire plus tard, mais doivent être soigneusement utilisés parce que le langage permet notamment de demander de lire un élément à un emplacement en dehors des bornes du tableau.

L'autre difficulté majeure que nous avons rencontré dans l'écriture de ces tableaux a été l'enregistrement de sous-chaines. Cette étape est nécessaire pour insérer les différentes parties de notre fichier original qui devrait correspondre à un champ répété, pour pouvoir ensuite vérifier que chaque élément du tableau respecte le format défini. Par contre, l'opération d'enregistrement d'élément ne fonctionne que sur des variables préalablement déclarées : l'insertion d'une valeur retournée par la fonction substr n'a aucun effet sur le tableau, ce qui nous a finalement empêché de faire la vérification des champs répétés.

V. Conclusion

Suite aux difficultés rencontrées, la production finale reste assez loin de l'objectif initial. Nous avons réussi à proposer une approche pour modéliser automatiquement des fichiers, mais nous n'avons pas réussi à la faire fonctionner dans le temps qui nous restait. Nous n'avons pas non plus pu entreprendre le travail d'extension du compilateur Kaitai Struct sans un prototype fonctionnel de représentation SMT. Toutefois, si on peut trouver une manière d'exploiter les chaines de caractères dans des tableaux SMT, ce modèle aboutirait à quelque chose d'exploitable et la production de l'extension de Kaitai Struct serait possible.

L'étape d'analyse des fichiers produits et parsées serait très intéressante et riche en possibilités. La première approche du problème serait d'exploiter les modèles SMT générés à partir de Kaitai pour déduire le fichier le plus court qui satisfait 2 formats arbitrairement choisis.

VI. Références

Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB), `www.SMT-LIB.org`*. 2016. [BibTeX entry]*

Kaitai Struct: http://kaitai.io/

Z3 Theorem Prover : https://github.com/Z3Prover/z3