Université Paris 7 - M2 LMFI Année 2018-2019

Preuves assistées par ordinateur – Projet :

Tables d'associations symétriques et compression LZW

Présentation

Le but du projet est de définir en Coq des fonctions de compression et décompression utilisant la méthode LZW (pour Lempel, Ziv et Welch).

La compression LZW utilise une notion de dictionnaires permettant d'associer des codes numériques à des chaînes de caractères. Lors de la décompression, on recherche réciproquement à retrouver des chaînes à partir de codes. La première partie de ce projet propose donc de réaliser une structure de tables d'associations symétriques. La seconde partie est le travail sur LZW proprement dit.

1 Première partie : tables d'associations symétriques

Comme mentionné en introduction, la compression LZW utilise une notion de dictionnaire pour associer des codes numériques à des chaînes de caractères, tandis que l'association se fait dans l'autre sens lors de la décompression : à partir de codes, on souhaite retrouver des chaînes correspondantes.

Ces deux phases de compression et de décompression étant indépendantes, elles peuvent parfaitement utiliser deux structures distinctes pour leurs besoins respectifs. D'ailleurs les implantations cherchant l'efficacité maximale procèdent ainsi.

Ici, nous nous intéresserons plutôt à une approche "tout-en-un", afin d'éviter de multiplier les structures de données et les preuves associées. Nos tables d'associations seront donc *symétriques* : pour toute chaîne, on pourra y chercher un code numérique associé, et réciproquement. Un dictionnaire pour LZW codé ainsi pourra donc servir à la fois pour la compression et la décompression. Dans ce qui suit, nous utiliserons le mot dictionnaire pour parler de table d'associations symétrique.

1.1 Implantation

Question Proposer une implantation pour le type des dictionnaires, et pour les opérations de création d'un dictionnaire vide, d'ajout d'une association dans le dictionnaire, et de recherche dans le dictionnaire (nombre à partir d'une chaîne [1] et chaîne à partir d'un nombre). Les types de ces éléments devront correspondre à l'interface abstraite suivante :

```
Parameter dict : Set.
Parameter empty_dict : dict.
Parameter add_dict : string -> nat -> dict -> dict.
Parameter assoc1 : string -> dict -> option nat.
Parameter assoc2 : nat -> dict -> option string.
```

Comme le montre l'interface ci-dessus, on ne cherche pas à réaliser une structure générique (autrement dit, polymorphe), mais plutôt une structure dédiée aux chaînes du type `string` et aux entiers du type `nat`. Le choix du codage est laissé libre, mais nous vous incitons à privilégier la simplicité : ni efficacité ni propriétés logiques particulières (p.ex. unicité) ne sont cruciales ici.

Autre remarque, les fonctions de recherche `assoc1` et `assoc1` peuvent échouer, et en l'absence d'exceptions du style `Not_found`, un échec se caractérise par la réponse `None` du type `option`. Il

1. Les chaînes de caractères seront représentés par le type `string` dont dispose Coq à partir du moment où un `Require Import String` a été fait. Voir plus de détails sur ce type `string` dans la section 3.

est également possible de réaliser en plus des variantes `assoc1_noerr` et `assoc2_noerr` à n'utiliser que lorsque l'on sait d'avance que la recherche va être fructueuse. On pourra alors renvoyer une valeur arbitraire en cas d'erreur.

1.2 Un invariant utile pour LZW

En fait, les dictionnaires que nous allons utiliser pour LZW ont une forme très précise :
— ils seront bijectifs, c'est-à-dire qu'une chaîne ou un nombre n'y apparaît qu'au plus une fois.
— ils contiendront des associations pour au moins toutes les chaînes de taille 1 (c.à.d les caractères)
— les nombres présents rempliront exactement tout un intervalle entre 0 inclus et un certain nombre `max` exclus.

Question Définir une fonction de test permettant de vérifier si les conditions précédentes sont bien satisfaites.

1.3 Un dictionnaire initial pour LZW

Question Définissez un dictionnaire initial `init_dict` associant chaque caractère avec son code ascii. Vérifiez que ce dictionnaire initial satisfait bien les conditions précédentes.

2 Seconde partie : la compression LZW

2.1 Un peu d'histoire

L'algorithme LZW a été proposé par Welch en 1984, et constitue un raffinement d'algorithmes proposés initialement par Lempel et Ziv. L'idée générale des algorithmes de cette famille est de tirer parti des motifs répétés dans le document à compresser. Un tel motif se voit attribué un code particulier. Par la suite, toute nouvelle apparition de ce motif est remplacé par le code correspondant, normalement choisi pour être plus court que le motif, ce qui peut mener à de substantielles économies de place.

Dans le cas des algorithmes originaux de Lempel et Ziv (LZ77 et LZ78), on se retrouve alors avec un dictionnaire entre motifs et codes, qu'il faut également placer dans le fichier compressé. Les format de compression utilisant cette méthode (zip, gzip) compressent alors ces dictionnaires avec une autre méthode, celle de Huffman, afin d'obtenir des taux de compression encore meilleurs. Pour mémoire, l'algorithme de Huffman est basée sur une autre approche : au lieu de s'intéresser à l'organisation des caractères en motifs, on regarde la fréquence de chaque caractère, afin de représenter en peu de bits les caractères les plus fréquents.

Pour revenir à LZW, il possède la particularité de fabriquer son dictionnaire d'une façon telle qu'il n'y a pas besoin de le stocker : lire le flot de codes correspondant au fichier compressé permet de reconstituer au fur et à mesure le dictionnaire employé lors de la compression, comme nous allons le voir juste après. Au niveau pratique, on rencontre de la compression LZW dans les images gif, dans l'antique format compress/uncompress de unix (les .Z), ainsi que comme compression possible de certains vieux pdf. Aux États-unis, un brevet logiciel de la société Unisys sur LZW a longtemps posé problème aux développeurs voulant utiliser cet algorithme. Ce brevet a expiré il y a quelques années.

Pour finir le tour d'horizon des méthodes de compression, il existe des algorithmes plus récents fournissant de meilleurs taux de compressions, au prix souvent d'un coût de calcul plus élevé. On peut citer en particulier `bzip2`, basé sur des mathématiques plus complexes, à savoir la transformée de Burrows-Wheeler.

Évidemment, si vous souhaitez en savoir plus, wikipédia est votre ami...

2.2 La compression

La compression se fait à l'aide des éléments suivants :

— la chaîne de caractère à compresser, que l'on va parcourir linéairement
— un tampon dans lequel peuvent être placé des caractères en attente de codage
— un dictionnaire associant des chaînes de caractères à des codes numériques. Initialement, ce dictionnaire associe à chaque caractère son code ascii.

Le déroulement de la compression se résume alors en deux formules : "tant qu'on gagne, on rejoue" et "ne jamais faire deux fois la même erreur". Plus précisément :

— Tant que le tampon contient un mot présent dans le dictionnaire, on essaie de rallonger ce mot à l'aide de caractères pris dans la chaîne à compresser.
— Le jour où le caractère lu dans la chaîne forme avec le tampon un mot inconnu du dictionnaire, on émet le code correspondant au mot du tampon actuel, on enrichit le dictionnaire d'une entrée pour le mot inconnu, au cas où il se représenterait, et enfin on relance la manoeuvre avec un tampon vidé contenant juste le dernier caractère lu.

Voyons le déroulement de l'algorithme sur le mot `"repetition"` :

étape	tampon	caractère lu	code émis	ajout au dictionnaire
0		r		
1	r	e	114 (r)	re : 256
2	e	p	101 (e)	ep : 257
3	p	e	112 (p)	pe : 258
4	e	t	101 (e)	et : 259
5	t	i	116 (t)	ti : 260
6	i	t	105 (i)	it : 261
7	t	i		
8	ti	o	260 (ti)	tio : 262
9	o	n	111 (o)	on : 263
10	n	EOF	110 (n)	

et sur le mot `"aaaaaaa"` :

étape	tampon	caractère lu	code émis	ajout au dictionnaire
0		a		
1	a	a	97 (a)	aa : 256
2	a	a		
3	aa	a	256 (aa)	aaa : 257
4	a	a		
5	aa	a		
6	aaa	EOF	257 (aaa)	

Dans les descriptions de LZW que vous pourrez trouver par ailleurs, les codes sont décalés de 1. Le véritable LZW réserve en effet le code zéro pour un usage particulier : signaler que le nombre de bits utilisé actuellement pour émettre les codes ne suffit plus. Tous les codes émis par la suite contiendront alors un bit de plus, et ainsi de suite. Dans le cadre de ce projet, nous nous contenterons de produire une liste de `nat` correspondant à l'objet compressé. La représentation de cette liste de `nat` en terme de flots de bits n'est pas demandé, même si cela peut être une extension très intéressante de ce projet.

2.3 La décompression

L'objectif est maintenant renversé : on reçoit une liste de codes, et il s'agit de retrouver la chaîne d'origine. De nouveau, on dispose d'un dictionnaire, initialement rempli avec les caractères et leurs codes ascii.

Essayons sur le codage `[114;101;112;101;116;105;260;111;110]` du mot `"repetition"` :

— il est facile de se convaincre que le premier code est forcément celui d'un caractère, qui est bien présent dans notre dictionnaire. ici `114=r`. Par ailleurs, lorsque ce 114 a été émis, le dictionnaire a été enrichi avec une entrée `(r?:256)`. Malheureusement, derrière le `?` se cache un caractère que l'on ne peut pas déterminer pour l'instant.
— `101=e` a le bon goût d'être dans notre dictionnaire, et de ne pas faire intervenir l'entrée inconnue numéro 256. On est d'ailleurs maintenant en mesure de savoir ce qui se cache en 256 : c'est `e` qui formait avec `r` le mot inconnu `re` au moment de la création de l'entrée 256. On est donc maintenant en mesure de compléter cette entrée avec un temps de retard : `(re:256)`. Pendant ce temps, une entrée `(e?:257)` a eu lieu du coté de la compression, de nouveau avec un caractère qui nous est inconnu pour le moment.
— Et ainsi de suite : `112=p`, et on sait maintenant que le dictionnaire contient l'entrée `(ep:257)`, mais aussi l'entrée partiellement inconnue `(p?:258)`.
— etc etc

Résumons (on utilise ici une variable `prev` pour stocker le mot décodé à l'étape précédente) :

étape	code lu	prev	mot décodé	ajout certain au dictionnaire
0	114		r	
1	101	r	e	re : 256
2	112	e	p	ep : 257
3	101	p	e	pe : 258
4	116	e	t	et : 259
5	105	t	i	ti : 260
6	260	i	ti	it : 261
7	111	ti	o	tio :262
8	110	o	n	on : 263

On remarquera qu'on n'a jamais eu besoin d'utiliser une entrée de dictionnaire au moment où elle nous est encore partiellement inconnue. Mais essayons maintenant avec le code `[97;256;257]` provenant de la compression de `"aaaaaa"` :

— On commence avec `97=a`. A ce moment là, existe une entrée `(a?:256)`, où le `?` correspond au premier caractère du mot qui va venir à l'étape d'après.
— Mais voici justement que le mot suivant est celui se cachant derrière l'entrée 256. C'est donc notre `a?`. Le premier caractère de ce mot est donc `a`, et c'est ce premier caractère qui doit remplacer le `?` à la fin du mot. On en déduit donc l'entrée `(aa:256)`, et donc en partie l'entrée suivante `(aa?:257)`.
— Rebelotte avec 257 : le dernier caractère qui nous manque doit être égal au premier, c'est donc un `a`, et l'entrée complétée est donc `(257:aaa)`.

On peut donc en déduire la règle suivante : si le décodage fait intervenir une entrée du dictionnaire non encore totalement connue, le mot recherché est alors `prev++(head prev)`, avec `++` la concaténation de mots et `head` la fonction d'accès au premier caractère d'un mot. En résumé :

étape	code lu	prev	mot décodé	ajout certain au dictionnaire
0	97		a	
1	256	a	aa	aa : 256
2	257	aa	aaa	aaa : 257

2.4 Travail demandé

Question Implanter les fonctions `encode` et `decode` réalisant respectivement les boucles principales de compression et de décompression. Tester vos fonctions sur des exemples, et vérifier sur ces exemples que `decode` ∘ `encode` est bien l'identité.

2.5 Extensions possibles

— Réalisation d'un programme exécutable autonome (un "binaire"), en utilisant l'extraction Coq vers OCaml ou Haskell
— Traitement des vrais fichiers `.Z`, tels que le font les programmes historiques `compress` et `uncompress` (cf. le paquet debian/ubuntu `ncompress`).

3 Conseils et notions utiles

Ascii et String

Coq propose un codage des caractères (type `ascii`) et des chaînes (type `string`), à charger via `Require Import Ascii String`. On peut ensuite activer les notations concernant les caractères et/ou les chaînes via `Open Scope char_scope` ou `Open Scope string_scope`.

Concernant les caractères, il vous suffit de savoir qu'il y en a 256, qu'on passe du code ascii au caractère et vice-versa via les fonctions `nat_of_ascii` et `ascii_of_nat`. La composée de ces fonctions fait ce que l'on pense.

Pour ce qui est des chaînes de caractères, elles sont représentés par une structure inspirée des listes : une chaîne est soit vide (`EmptyString`, noté également `""`) soit un collage d'un caractère et d'une chaîne (constructeur `String`). La syntaxe `"abc"` permet de saisir directement une chaîne. Les fonctions utiles sont le test d'égalité `string_dec`, la concaténation `append` (notée `++`) et l'accès à la tête (fonction `head`).