

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), puis de prouver qu'un cycle compression/décompression préserve bien l'intégrité des données.

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, et de prouver un certain nombre de propriétés de ces tables, en prévision du travail sur LZW.

Modalités

Ce projet doit être effectué en *binôme* ou en *monôme* exclusivement.

On rendra par e-mail une archive nommée `nom1-nom2.tgz` contenant un répertoire `nom1-nom2` (où `nom1` et `nom2` désignent les noms des éléments du binôme) dans lequel on trouvera :

- le ou les fichiers Coq (fichiers `*.v`) du développement ;
- le fichier `Makefile` permettant de compiler le développement.

Il n'est pas explicitement demandé (mais pas interdit non plus) de fournir un rapport séparé des sources. En revanche, on fera attention à rendre un code **lisible** (dont les lignes ne dépassent pas 78 caractères) et **commenté** de manière pertinente (un commentaire n'est pas une paraphrase).

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 chaîne¹ 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 `assoc2` peuvent échouer, et en l'absence d'exceptions du style `Not_found`, un échec se caractérise par la réponse `None` du type `option`. Il est également possible (et même conseillé, cf. le fichier squelette fourni) 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.

1.2 Prédicats et propriétés

Question Définissez successivement :

- un prédicat `In1:string->dict->Prop` caractérisant la présence d'une chaîne donnée parmi les associations du dictionnaire.
- un prédicat `In2:nat->dict->Prop` caractérisant de même la présence d'un entier donné.
- un prédicat `In:string*nat->dict->Prop` signalant la présence d'une association donnée quelque part dans le dictionnaire
- un prédicat `Bijjective:dict->Prop` caractérisant les dictionnaires bijectifs, c'est-à-dire ceux pour lesquels toute chaîne ou nombre n'apparaît qu'au plus une fois.

Question Parmi les énoncés suivants, certains ne sont vrais que pour des dictionnaires bijectifs. Déterminez lesquels, adaptez les énoncés en question, et prouvez toutes ces propriétés.

```
Lemma In1_assoc1: forall s d, In1 s d <-> assoc1 s d <> None.
Lemma In2_assoc2: forall p d, In2 p d <-> assoc2 p d <> None.
Lemma In_assoc1: forall s d p, In (s,p) d -> assoc1 s d = Some p.
Lemma In_assoc2: forall p d s, In (s,p) d -> assoc2 p d = Some s.
Lemma assoc1_In: forall s d p, assoc1 s d = Some p -> In (s,p) d.
Lemma assoc2_In: forall p d s, assoc2 p d = Some s -> In (s,p) d.
Lemma assoc1_assoc2: forall d s p, assoc1 s d = Some p -> assoc2 p d = Some s.
Lemma In1_add: forall d s0 s p, In1 s0 d -> In1 s0 (add_dict s p d).
Lemma In2_add: forall d p0 s p, In2 p0 d -> In2 p0 (add_dict s p d).
```

1.3 Un invariant utile pour LZW

En fait, les dictionnaires que nous allons utiliser pour LZW ont une forme encore plus précise :

- ils seront bijectifs
- 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.

¹Les chaînes de caractères seront représentées 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.

Question Définir un prédicat `Invariant` tel que `(Invariant d max)` impose que `d` remplissent les conditions ci-dessus. Prouver ensuite que cet invariant est préservé lors d'un ajout "raisonnable" :

```
Lemma Invariant_propagates : forall d max s, assoc1 s d = None ->
  Invariant d max -> Invariant (add_dict s max d) (S max).
```

1.4 Un dictionnaire initial pour LZW

Question Définissez un dictionnaire initial `init_dict` associant chaque caractère avec son code ascii. Prouvez ensuite que ce dictionnaire est bien convenable, au sens où l'on a bien `(Invariant init_dict 256)`.

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 formats 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 basé 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'ancien 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 placés 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 "aaaaaa" :

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 est une autre histoire ...

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 côté 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 dans le fichier fourni les fonctions `encode` et `decode` réalisant respectivement les boucles principales de compression et de décompression. Pour `decode`, on pourra tirer parti de la fonction fournie `nextstr`. Prouver ensuite les trois derniers lemmes décrivant successivement les comportements de `nextstr`, de la composée `decode` ◦ `encode` et enfin de la composée `uncompress` ◦ `compress`

3 Conseils et notions utiles

Afin d'utiliser au mieux les différentes fonctionnalités du système (commandes, tactiques, notations, librairie standard, etc.), il est fortement recommandé de consulter attentivement la documentation que l'on trouve en ligne à l'URL

<http://coq.inria.fr/>

Voici en particulier une liste (pas forcément exhaustive) d'éléments du système qui peuvent s'avérer intéressants dans le cadre de ce projet :

Types élémentaires

La paire s'écrit comme en Caml, avec la notation (a,b) pour l'objet et la notation $A*B$ pour son type. De même, le type `option` est le même qu'en Caml.

Arith et Omega

Vous aurez à comparer des entiers naturels du type `nat`. La bibliothèque standard propose pour cela une fonction `eq_nat_dec` de type

```
forall n m : nat, { n=m } + { n<>m }
```

Cette fonction répond donc plus d'information qu'un simple oui/non à la question de savoir si `n` et `m` sont égaux : selon le cas de figure, elle va renvoyer une preuve de `n=m` ou une preuve de `n<>m`. En tout cas, dans un programme Coq, on peut s'en servir comme d'une simple fonction booléenne, par exemple en tête d'un `if then else`. Par la suite, lors d'une preuve, si l'on se retrouve face à un but contenant un `eq_nat_dec n m`, il convient alors d'utiliser `destruct (eq_nat_dec n m)` pour explorer les deux alternatives possibles.

Par ailleurs, il y aura aussi à parler d'entiers strictement inférieurs à d'autres, via `<` qui est en fait une notation pour le prédicat `lt`. Une utilisation de `SearchAbout (F2 dans coqide)` vous en dira plus sur tous les résultats à votre disposition concernant `lt`. En pratique, `auto` et surtout `omega` (qui résout les formules sans quantificateurs de l'arithmétique de Presburger) vous suffiront probablement.

List

Ce module propose la même définition du type `list` que celle vue en TD, avec en bonus la possibilité d'utiliser la syntaxe `t::l` à la Caml au lieu de `(cons t l)`. Parmi la vaste liste de choses fournies dans ce module, seul le prédicat `In` d'appartenance à une liste devrait vous être utile. Il est défini de manière à avoir les simplifications suivantes :

```
In x nil = False
In x (t::l) = (x=t) \/ (In x l)
```

Ascii, String et MoreString

Attention, `MoreString.v` n'est pas (encore) dans la bibliothèque standard. Vous devrez donc le récupérer sur le site du projet et le compiler dans votre répertoire de travail avec `coqc`.

Coq propose depuis peu un codage des caractères (type `ascii`) et des chaînes (type `string`). 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, voir les lemmes `ascii_nat_embedding` et `nat_ascii_embedding`.

Pour ce qui est des chaînes de caractères, elles sont représentées 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`). Parmi les lemmes à noter : `append_empty`, `append_String` et `head_append`.

Un mécanisme de coercion permet d'utiliser un caractère en lieu et place d'une chaîne. Coq insère alors magiquement au bon endroit un `ascii_to_string` (qui n'est autre qu'un appel au constructeur `String ... ""`). Même si cela est très commode, attention aux rares effets pervers : par exemple, si `a` et `a'` sont deux caractères, une égalité affichée `a=a'` peut être soit l'égalité que l'on pense sur les caractères, soit l'égalité sur les chaînes avec des `ascii_to_string` cachées.

Tactiques pratiques

- Dans le cadre de ce projet, l'usage des tactiques automatiques telles que `auto` est fortement encouragé. Si un lemme `truc` revient fréquemment dans vos `apply`, vous pouvez l'apprendre à `auto` avec la commande : `Hint Resolve truc`.

- Les tactiques `f_equal` et `case_eq` peuvent être une aide précieuse. La première vous aide à prouver une égalité de la forme $f\ x \dots = f\ x' \dots$ en vous proposant de prouver les égalités entre arguments : $x=x'$, etc. La seconde, `case_eq`, est une alternative à `destruct` ou `case` permettant de garder une trace de l'objet que l'on souhaite casser.
- Dans le dernier théorème, il se peut que l'usage de la tactique `simpl` pose un problème, que l'on peut contourner avec la tactique `remember`. Voir le détail dans un commentaire du fichier fourni.
- Une dernière "ruse" : après avoir été pendant longtemps un poison à manier, les lemmes parlant d'équivalence logique sont maintenant utilisable de façon confortable via la tactique `rewrite`. Par exemple, si vous disposez d'un lemme `truc:A<->B` et que votre but contient une instance de `A`, alors `rewrite truc` changera ce `A` en `B`.
- Notez également que le fichier `squelette` qui vous est fourni propose deux tactiques additionnelles pouvant vous aider : `discr` qui est un `discriminate` aux pouvoirs étendus, et `cassepaire` pour casser les égalités sur les paires.
- Sachez enfin qu'il existe une tactique `admit` permettant d'admettre un sous-but sans le prouver. Évidemment, l'usage de cette tactique n'est pas recommandé, mais peut être commode pour se focaliser d'abord sur certaines portions d'une preuve. Il va de soit que la notation finale dépendra du nombre et de l'importance des `Axiom` et des `admit` restant lors du rendu.

Méthodologie

L'objectif est de remplacer dans le fichier fourni les `Parameter` par des `Definition` (ou des `Fixpoint` s'il s'agit de fonctions récursives) et les `Axiom` par des `Lemma`. Ceci peut être fait dans un ordre quasi-arbitraire. Lorsque vous attaquez une preuve, il faut simplement qu'il n'y ait plus de `Parameter` dans ce qui précède. Au delà de ça, vous pouvez vous occuper des `Axiom` dans l'ordre qui vous plaît. Même si cela permet une forte division du travail, il est rappelé que tout membre d'un binôme doit être à même d'expliquer l'intégralité du travail rendu.