# Un régime au concentré d'automate

Pierre-Marie Pédrot

# Un régime au concentré d'automate

Pierre-Marie Pédrot

*PPS, équipe $\pi r^2$, Univ. Paris Diderot, Sorbonne Paris Cité,*
*UMR 7126 CNRS, INRIA Paris-Rocquencourt, Paris, France*
pierre-marie.pedrot@inria.fr

**Résumé**

On présente dans cet article une optimisation *a posteriori* du format de fichier utilisé par l'assistant à la preuve Coq pour sauvegarder ses bibliothèques. L'implémentation purement fonctionnelle des structures de données contenues dans ces fichiers permet d'utiliser des algorithmes standards sur les automates qui garantissent de fait une optimalité du partage de la mémoire. Notre outil peut se généraliser directement au calcul du partage maximal lors de la sérialisation de toute structure de données OCaml utilisée de manière purement fonctionnelle.

## 1. Introduction

L'assistant à la preuve Coq [5] est un logiciel destiné à la preuve interactive de théorèmes. Ses fondements théoriques reposent sur le Calcul des Constructions Inductives (CCI), lui permettant de marier calculs et preuves au sein de la correspondance de Curry-Howard. De nombreux développements sont connus à ce jour, parmi lesquels, sur le versant mathématique, le théorème des quatre couleurs [4] et, sur le versant informatique, le compilateur Compcert [7].

Coq a aujourd'hui atteint une base d'utilisateurs d'une taille importante, et sa maturité est telle que les problèmes de performances se posent chaque jour de manière plus vive. L'optimisation est devenue une préoccupation importante de l'équipe de développement ; tâche éminemment difficile au vu de la complexité d'un tel programme. La technique du *hashconsing* est, par exemple, utilisée pour minimiser la mémoire allouée. Outre le gain lié au volume géré par le ramasse-miettes, cela permet d'effectuer certaines comparaisons en temps constant, en testant simplement l'égalité de pointeurs, au lieu de faire appel à des algorithmes plus coûteux.

En tant que langage de programmation, Coq propose la compilation séparée, en divisant le code en différents sous-fichiers, qui sont compilés sous la forme de fichiers `.vo` (pour *vernacular object*). Ces fichiers compilés embarquent les différentes données contenues dans une bibliothèque, comme évidemment les définitions et les termes de preuve correspondant, mais aussi toutes sortes d'informations relatives à l'état du moteur de Coq, comme les notations définies par l'utilisateur, les différentes options comportementales (*flags* d'affichage, de tactiques, etc.), ainsi que divers caches évitant d'avoir à recalculer certaines opérations coûteuses.

Pour des raisons expliquées en section 2, le mécanisme de compilation de fichiers `.vo` est basé sur l'algorithme de sérialisation fourni nativement par OCaml, le langage dans lequel Coq est écrit, via le module `Marshal`. En d'autres termes, les fichiers Coq compilés contiennent directement la représentation mémoire d'une partie du tas OCaml. Cette manière de faire a (seulement) l'avantage de la simplicité : Coq se contente d'utiliser une primitive fournie par la bibliothèque standard d'OCaml.

Cependant, la sérialisation native ne fait que représenter une partie de la mémoire d'un programme, et ce sans aucune considération pour la sémantique de son contenu. En particulier, malgré une utilisation purement fonctionnelle des fichiers objets, le sérialiseur ne cherche pas à fournir une représentation mémoire minimale, laissant de fait une redondance inutile qui pourrait être éliminée au moment de l'écriture d'un fichier objet.

En utilisant un algorithme bien connu de minimisation d'automate, l'algorithme de Hopcroft, nous avons écrit un programme permettant, à partir d'une structure OCaml, de générer la structure de donnée minimale correspondante pour le partage de mémoire. À notre connaissance, personne ne semblait avoir appliqué ce type d'algorithme au problème du partage maximal.

Une application directe de ce programme réside dans la compression des fichiers objets de Coq, qui permettent des gains de l'ordre de 40% en espace et de quelques pourcents en temps de compilation de la bibliothèque standard de Coq.

# 2. Dans les entrailles de Coq

## 2.1. Structure d'un fichier objet

L'écriture d'un fichier `.vo` repose sur une sérialisation quasiment directe de la représentation mémoire de l'état interne de Coq. Cette sérialisation ne permet pas d'exporter de manière fiable tout objet contenant des structures d'ordre supérieur (typiquement des fermetures) ou qui ne vivent pas sur le tas OCaml (typiquement des données allouées en C). Cela force les fichiers objets à avoir une structure très simple et facilement manipulable.

Dans l'implémentation actuelle, un fichier objet contient, dans l'ordre :

1. un en-tête dépendant de la version du compilateur, qui évite de mélanger des fichiers incompatibles ;
2. la sérialisation de l'état du moteur à proprement parler, décrit au paragraphe suivant ;
3. un *hash* cryptographique de la réprésentation ci-dessus, qui garantit la non-corruption des données ;
4. une table des termes opaques utilisés au point 2.

La séparation entre les structures 2 et 4 est purement technique et relève de l'optimisation. La partie importante du contenu d'un fichier objet réside dans la deuxième structure.

## 2.2. État de Coq

Le moteur de Coq maintient son état sous forme d'une pile d'actions ayant permis d'arriver à l'état courant, et c'est précisément cette structure qui est exportée lors de la sauvegarde d'un fichier objet. Chaque commande modifiant l'état du moteur modifie la pile conséquemment. Le contenu d'un fichier objet est ainsi similaire à la figure 1, donnée à titre purement indicatif.

```
((Top._20, Top._20), Leaf DOT);
((Top._19, Top._19), FrozenState <abstr>);
((Top._18, Top._18), Leaf ARGUMENTS-SCOPE);
((Top._17, Top._17), Leaf HEAD);
((Top._16, Top._16), Leaf IMPLICITS);
((Top.foobar, Top.foobar), Leaf CONSTANT);
```

FIGURE 1 – Ajouts de la commande `Definition foobar := nat.` sur la pile.

Chaque élément de cette pile correspond à une certaine action. Dans l'exemple, la ligne contenant `Top.foobar` correspond à la déclaration d'une constante nommée `foobar` dans l'environnement global ; les autres lignes sont des déclarations anonymes. Les trois lignes au-dessus de cette déclaration gèrent divers détails de comportement de cette constante, comme la gestion de ses arguments implicites, et

son espace de nommage, au niveau des notations. Les deux dernières lignes sont des données relatives au mécanisme de *backtrack*.

Le problème apparaît alors à ce moment, car les données présentes dans cette pile sont purement dynamiques, et appartiennent en pratique à des types ininformatifs (type `Obj.t`).

Pour peu qu'il ait défini une demi-douzaine de comportements de manipulation contextuelles que l'objet doit fournir à l'interface de la gestion de l'état, le programmeur peut mettre à peu près n'importe quoi dans la pile. Ce laxisme est justifié par l'extensibilité de Coq : un développeur extérieur pourrait souhaiter définir, par exemple, de nouvelles commandes dans un *plugin*.

La sérialisation de ces objets ne connaît pas *a priori* leur type, c'est donc pour cela qu'il faut recourir au module `Marshal` de la bibliothèque standard d'OCaml.

## 2.3. Invariant

Pour rester cohérent avec le mécanisme de *backtrack* de la boucle interactive de Coq, la bibliothèque de gestion de la pile présuppose qu'on n'y introduira jamais de structure mutable, sous peine de comportements non-définis en cas de modification dynamique. Par ailleurs, à cause des phases de sérialisation-désérialisation, le programmeur ne peut pas supposer l'invariance de l'emplacement mémoire des objets de la pile. En d'autres termes, le contenu de la pile doit rester purement fonctionnel.

La compression des fichiers objets repose précisément sur cet invariant, condition *sine qua non* de la correction de l'algorithme vis-à-vis de la sémantique opérationnelle.

## 2.4. Hashconsing

Pour minimiser l'espace mémoire et optimiser certaines opérations, Coq utilise aussi le *hashconsing* [2]. Cette technique consiste à mémoïser dynamiquement les structures de données manipulées.

Étant donné un type `t` et une relation d'équivalence `eq : t → t → bool`, on construit une table de *hash* sur `t` et `eq`, et toute opération d'allocation d'un élément de `t` vérifie que l'objet à créer n'est pas déjà présent dans cette table (au sens de `eq`). Si c'est le cas, on renvoie la valeur de la table, sinon on déclare la structure créée comme étant la structure canonique attachée à cette valeur.

Le but visé est ainsi d'avoir au plus un objet en mémoire pour chaque classe d'équivalence de `eq`, pour un surcoût acceptable. Outre la diminution de la consommation mémoire, on permet aussi à certaines comparaisons de s'effectuer en $O(1)$ puisqu'elles se réduisent à l'égalité de pointeurs.

En pratique, ce n'est pas tout à fait le cas. En effet, le mécanisme de chargement de fichier objets se contentant de désérialiser directement les structures en mémoire, celles-ci ne repassent pas par les fonctions de *hashconsing*, brisant de fait le partage maximal des classes d'équivalence.

Ce défaut est particulièrement visible dans les fichiers ayant le plus de dépendances de la bibliothèque standard de Coq, où l'on peut par exemple trouver la chaîne de caractères `"Coq"` plusieurs centaines de fois en mémoire. Il s'agit en fait du préfixe commun à tous les chemins des fichiers objets la bibliothèque standard, qui se retrouve dupliquée chaque fois qu'une commande `Require` est émise. On peut facilement imaginer la redondance mémoire introduite par le mécanisme de compilation séparée si chaque structure de donnée échappe au *hashconsing* lors du chargement d'un fichier objet.

Coq pourrait assurément être plus subtil lors des `Require` en réeffectuant le *hashconsing* à la volée, mais cela risquerait de se révéler potentiellement coûteux, puisqu'il faudrait traverser des structures d'une taille certaine. En outre, l'architecture du code de gestion des bibliothèques ne permet pas de faire cela dans l'état actuel. Il faudrait en fait se passer complètement de la sérialisation fournie par OCaml et réécrire notre propre système de représentation des fichiers objets.

Notre solution, elle, est complètement externe et insensible à ce genre de considérations.

# 3. Modèle mémoire d'OCaml

OCaml est un langage de programmation multi-paradigmes basé sur un noyau fonctionnel ML. On discute dans cette section de la représentation en mémoire des structures de données OCaml. On fait abstraction dans cette section des problématiques liées aux différents tas et au ramasse-miettes ; par ailleurs les assertions vis-à-vis de la sémantique opérationnelle ne seront pas formalisées plus avant. Les pseudo-théorèmes de cette section sont donc plus à prendre comme des justifications de la validité de notre méthode plutôt que comme une preuve *stricto sensu*.

## 3.1. Structures de données

En ce qui concerne la représentation mémoire, un objet OCaml peut être :
- un entier immédiat ;
- un bloc résident en mémoire, muni d'un tag $t$ et d'une longueur $l$.

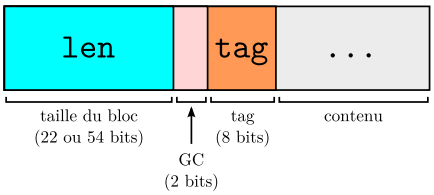


FIGURE 2 – Représentation d'un bloc

Le tag d'un bloc permet de spécifier son utilisation. Pour les structures de premier ordre trouvées dans Coq, on peut se limiter aux deux cas suivants [1] :
- chaîne de caractères (tag 252) ; il s'agit techniquement d'un tableau d'octets, mais nous le traiterons de manière opaque.
- tableau structuré (tags 0 – 244), dont chaque champ compris entre 0 et $l - 1$ contient
  1. soit un entier immédiat ;
  2. soit un pointeur vers un autre bloc en mémoire.

On notera $\{t :: x_1 \dots x_n\}$ le tableau structuré de tag $t$ et de contenu $x_1, \dots, x_n$, et $[s]$ la chaîne de caractère $s$. La définition suivante formalise le fait qu'une mémoire correcte ne contient pas de pointeur pendant.

**Définition 1.** Une mémoire est une paire $\mathcal{M} = (P, \varphi)$ où $P$ est un ensemble fini de pointeurs et $\varphi$ une fonction de $P$ dans les blocs telle que si $p \in P$, $\varphi(p) = \{t :: x_1 \dots x_n\}$ et $x_i$ est un pointeur, alors $x_i \in P$. On notera $p \mapsto x$ si $\varphi(p) = x$.

On définit figure 3 une réification de la mémoire d'un programme dans les types algébriques. Les pointeurs y sont représentés par les entiers. Par la suite, par souci de lisibilité, on confondra la mémoire d'un programme et sa réification ; notons cependant que tous les habitants du type des mémoires réifiées ne sont pas nécessairement l'image d'une mémoire.

On dénotera $(t, \mathcal{M})$ un terme $t$ vu dans une mémoire $\mathcal{M}$ et on notera $(t, \mathcal{M}) \Downarrow (v, \mathcal{N})$ si $t$ dans $\mathcal{M}$ s'évalue sur la valeur $v$ et retourne la mémoire $\mathcal{N}$.

1. Les structures du premier ordre non prises en compte comprennent notamment les flottants et les tableaux de flottants.

```
type ptr = int
type tag = int
module Memory : Map.S with type elt = ptr
type content = Int of int | Ptr of ptr
type block = Struct of tag * content array | String of string
type memory = block Memory.t
```

FIGURE 3 – Réification de la mémoire

## 3.2. Égalité structurelle

OCaml est fourni avec une fonction d'égalité générique `(=) : 'a → 'a → bool`, appelée *égalité structurelle*. Comme son nom l'indique, elle compare la structure de la représentation mémoire d'objets OCaml, sans tenir compte de leur type. L'égalité structurelle n'est pas une fonction totale, car elle peut boucler sur les données récursives.

Sur les structures du premier ordre, elle découle d'une relation d'équivalence [2] aisément définie par coinduction.

**Définition 2** (Équivalence structurelle)**.** Soient $t$ et $u$ deux blocs, et $x$ et $y$ deux champs. On définit l'équivalence structurelle $t \simeq u$ et $x \simeq_f y$ par coinduction :

$$\frac{\text{pour tout } i \leq n \qquad x_i \simeq_f y_i}{\{t :: x_1 \dots x_n\} \simeq \{t :: y_1 \dots y_n\}} \qquad \frac{}{[s] \simeq [s]} \qquad \frac{}{\texttt{Int}\ i \simeq_f \texttt{Int}\ i} \qquad \frac{p \mapsto x \qquad q \mapsto y \qquad x \simeq y}{\texttt{Ptr}\ p \simeq_f \texttt{Ptr}\ q}$$

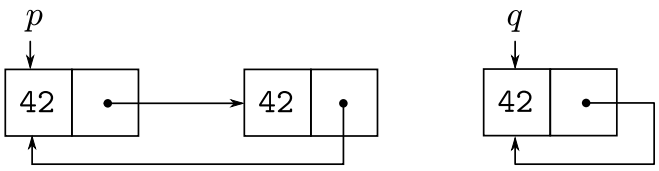


FIGURE 4 – Deux listes structurellement équivalentes

On peut trouver en figure 4 la représentation mémoire de deux listes structurellement équivalentes, l'une correspondant au code `let rec p = 42 :: 42 :: p` et l'autre au code `let rec q = 42 :: q`. Notons que dans cet exemple, nous sommes dans le cas où le test d'égalité structurelle boucle bien que les deux objets soient structurellement équivalents, les structures considérées étant récursives.

**Définition 3.** Une fonction `f : t → u` est dite structurellement insensible si pour tous `x : t` et `y : t` tels que `x` $\simeq$ `y`, alors $(\texttt{f x}, \mathcal{M})$ est défini si et seulement si $(\texttt{f y}, \mathcal{M})$ l'est aussi, et leur valeur coïncide par égalité structurelle.

Remarquons que l'égalité structurelle entre les deux évaluations n'est pas nécessairement établie dans la même mémoire. En effet, dans la définition 2, on peut très bien relativiser la relation $p \mapsto x$ à une mémoire différente de chaque côté de l'équivalence. Nous avons omis ce détail par souci de lisibilité.

**Proposition 1.** *L'égalité structurelle est structurellement insensible, de même que la fonction de* hash *de la bibliothèque standard. Inversement,* $(\texttt{(=)}\ x\ y, \mathcal{M}_1) \Downarrow (\texttt{true}, \mathcal{M}_2)$ *implique* $x \simeq y$*.*

2. À l'exception bien connue des flottants, où pour des raisons de respect de la norme IEEE, `nan` n'est pas comparable avec lui-même.

*Démonstration.* D'après la documentation d'OCaml. □

L'existence de fonctions ne respectant pas l'insensibilité structurelle provient essentiellement de l'emploi du test d'égalité physique `(==)`. La plupart des autres fonctions [3], quand bien même non-paramétriques [4], préservent l'égalité structurelle.

Cependant, l'insensibilité structurelle n'est pas une condition suffisante pour garantir la préservation de la sémantique opérationnelle lors du remplacement sans précautions d'une mémoire par une autre équivalente point à point. La composition $g \circ f$ de deux fonctions structurellement insensibles $f$ et $g$ n'est pas nécessairement insensible, car $f$ peut modifier la mémoire en place.

**Définition 4.** Une fonction `f : t → u` est référentiellement transparente si pour tout objet `x : t` et toute mémoire $\mathcal{M}$ tels que $(\texttt{f x}, \mathcal{M}) \Downarrow (v, \mathcal{N})$ avec $\mathcal{M} = (P, \varphi)$ et $\mathcal{N} = (Q, \psi)$, on a :

$$\forall p \in P, \quad p \in Q \Rightarrow \varphi(p) \simeq \psi(p).$$

La preuve du théorème suivant nécessiterait une formalisation complète de la sémantique opérationnelle d'OCaml, en particulier de la spécification du ramasse-miettes, qui nous mènerait bien au-delà de la portée de cet article.

**Pseudo-théorème.** *Si $f$ et $g$ sont deux fonctions insensibles structurellement et transparentes référentiellement, alors leur composition $g \circ f$ l'est aussi.*

Si l'on admet ce qui précède, la conjonction des deux conditions de stabilité, à la fois de la mémoire et de l'égalité structurelle paraît être la propriété requise pour qu'une fonction respecte l'équivalence de mémoire point à point : tandis que l'insensibilité garantit que localement les résultats ne seront pas modifiés, la transparence référentielle permet elle le passage à la composition de programmes.

Remarquons que le fragment fonctionnel pur d'OCaml ne permet justement d'écrire que des fonctions à la fois insensibles et transparentes. Ainsi, tout programme purement fonctionnel désirant compacter sa mémoire peut utiliser notre méthode.

**Conjecture.** *Toutes les fonctions de Coq ayant pour argument un objet généré par la lecture d'un fichier `vo` sont structurellement insensibles et référentiellement transparentes.*

Une justification simple de cet état de fait repose sur la nécessité de l'invariant de la section 2.3. En effet, les phases de sérialisation-désérialisation imposent l'insensibilité, car la localisation précise en mémoire ne peut être garantie par ces opérations. D'autre part, si une telle fonction violait la transparence référentielle, elle briserait les hypothèses faites sur la pile-état de Coq. Dans ce cas, il s'agirait d'un bug explicite du programme.

# 4. Minimisation

Afin d'appliquer la compaction de mémoire sur les fichiers objets, on rappelle ici quelques définitions de base sur les systèmes de transition qui nous permettront d'exprimer les résultats de correction de l'algorithme.

**Définition 5.** Un LTS (*labelled transition system*) sur un alphabet $\mathcal{L}$ est une paire $A = (|A|, R_A \subseteq |A| \times \mathcal{L} \times |A|)$, où $|A|$ est l'ensemble d'états du système, et $R_A$ ses transitions. On notera $x \xrightarrow{\alpha} y$ si $(x, \alpha, y) \in R_A$.

Un LTS $A$ est dit déterministe si $x \xrightarrow{\alpha} y_1$ et $x \xrightarrow{\alpha} y_2$ impliquent $y_1 = y_2$ pour tous $x, y_1, y_2 \in |A|$.

3. À l'exception évidente de la sérialisation.
4. Y compris les fonctions du module `Obj`.

**Définition 6** (Bisimulation)**.** Étant donnés deux LTS $A$ et $B$, une bisimulation entre $A$ et $B$ est une relation $\mathcal{R} \subseteq |A| \times |B|$ telle que pour tous $(x, y) \in \mathcal{R}$ :

– si $x \xrightarrow{\alpha} x'$, alors il existe $y' \in B$ tel que $y \xrightarrow{\alpha} y'$ et $(x', y') \in \mathcal{R}$ ;

– si $y \xrightarrow{\alpha} y'$, alors il existe $x' \in A$ tel que $x \xrightarrow{\alpha} x'$ et $(x', y') \in \mathcal{R}$.

On dira que $A$ et $B$ sont équivalents s'il existe une bisimulation $\mathcal{R}$ entre $A$ et $B$, telle que pour tout $x \in A$, il existe $y \in B$ tel que $(x, y) \in \mathcal{R}$ et inversement.

Les bisimulations sont les relations naturelles décrivant l'équivalence de deux LTS. En pratique, on ne considérera dans la suite que des LTS déterministes finis, où la relation de bisimulation devient quelque peu triviale [5].

On peut maintenant voir la représentation mémoire d'un objet OCaml comme un LTS, en choisissant l'ensemble des transitions de manière astucieuse :

$$\mathcal{L} ::= \mathtt{T}\ t \in \mathbb{N} \mid \mathtt{S}\ s \in \Sigma^* \mid \mathtt{F}\,(k, i) \in \mathbb{N}^2 \mid \mathtt{P}\ k \in \mathbb{N}$$

où `T`, `F` et `P` représentent respectivement le *tag*, le contenu entier et le contenu pointeur d'un bloc structuré, et `S` représente le contenu d'une chaîne, $\Sigma$ représentant les caractères ASCII. Le contenu entier se trouve sous la forme d'une paire $(k, i)$ où $k$ encode l'indice du tableau où se trouve cet entier, et $i$ sa valeur. Le contenu pointeur contient uniquement l'indice $k$ du tableau, sa valeur se trouvant encodé dans les transitions.

**Définition 7.** Soit $\mathcal{M}$ une mémoire. On lui associe le système déterministe $L(\mathcal{M})$ comme suit :

1. Ses états $|L(\mathcal{M})|$ sont exactement les pointeurs de la mémoire ;
2. Ses transitions sont définies selon la nature de l'objet pointé, par les règles ci-dessous.

$$\frac{p \mapsto \{t :: \vec{x}\}}{p \xrightarrow{\mathtt{T}\ t} p} \qquad \frac{p \mapsto [s]}{p \xrightarrow{\mathtt{S}\ s} p} \qquad \frac{p \mapsto \{t :: x_1 \dots x_n\} \qquad x_k = \mathtt{Ptr}\ q}{p \xrightarrow{\mathtt{P}\ k} q}$$

$$\frac{p \mapsto \{t :: x_1 \dots x_n\} \qquad x_k = \mathtt{Int}\ i}{p \xrightarrow{\mathtt{F}\,(k,i)} p}$$

On donne un exemple de cette traduction à la figure 5.

On montre alors dans la proposition qui suit que cette étape de traduction préserve l'équivalence structurelle au travers des bisimulations. La correction de la traduction s'appuie sur le codage de valeurs internes en transitions supplémentaires de la forme `T`, `F` et `S`.

**Proposition 2.** *Soient $\mathcal{M}$ et $\mathcal{N}$ deux mémoires, soit $\mathcal{R}$ une bisimulation entre $L(\mathcal{M})$ et $L(\mathcal{N})$. Si $(p, q) \in \mathcal{R}$, $p \mapsto x$ et $q \mapsto y$, alors $x \simeq y$.*

*Démonstration.* Preuve par coinduction.

Pour tout bloc $z$, il y a exactement une transition $\alpha \in \{\mathtt{S}\ s, \mathtt{T}\ n\}$ telle que $z \xrightarrow{\alpha} z$ selon que $z$ est une chaîne ou un tableau structuré. D'où $x$ et $y$ sont du même type de bloc, sinon cette transition serait différente. Pour les même raisons, si $x$ et $y$ sont des chaînes, alors elles sont égales. De même, si ce sont des tableaux, ils ont le même *tag*.

Supposons que $x$ et $y$ sont des tableaux. À cause des conditions de bonne définition de la mémoire, ils ont alors la même longueur et le même type de contenu champ à champ (sinon ils auraient une transition `P` ou `F` qui les distingueraient). De plus, les transitions `F` assurent qu'ils coïncident sur leur valeurs entières. Quant aux transitions `P`, elles garantissent que leur valeurs pointeurs sont deux à deux en relation par $\mathcal{R}$. On conclut par hypothèse de coinduction. □

5. Elle est alors équivalente à l'égalité des langages reconnus par les LTS vus comme des automates.

```
let x = Some 42 in let y = Some 42 in
let s = "foobar" in
let rec p = (s, x) :: (s, y) :: p in
(p, 18, 57)
```

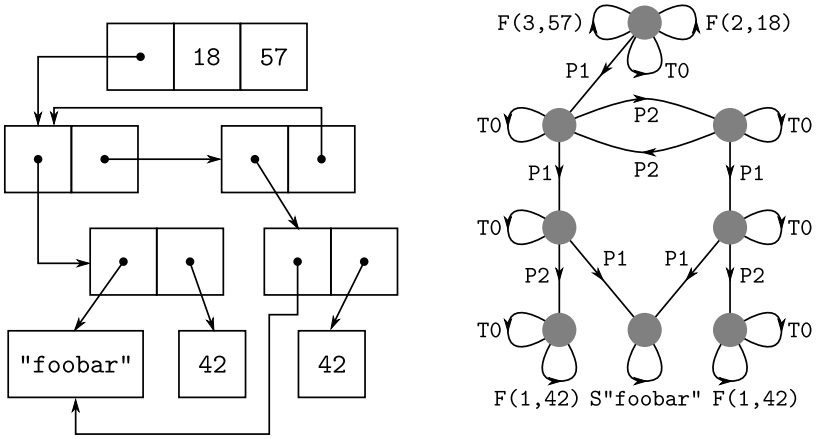


FIGURE 5 – Représentation mémoire et traduction en LTS du programme présenté

**Proposition 3.** *La fonction $L$ est une injection des mémoires dans les LTS sur $\mathcal{L}$. De plus, si $\mathcal{M}$ est une mémoire et $A$ un LTS déterministe fini tel que $A$ et $L(\mathcal{M})$ soient équivalents, alors il existe une mémoire $\mathcal{N}$ telle que $A = L(\mathcal{N})$.*

*Démonstration.* L'injectivité de $L$ est évidente. En effet, les blocs de la mémoire sont en bijection avec les états de l'automate résultant, et les transitions encodent exactement l'information qui n'a pas été traduite dans les états.

Soit $A$ un LTS déterministe et $\mathcal{M} = (P, \varphi)$ une mémoire. On définit $\mathcal{N} = (Q, \psi)$ par $Q = |A|$, et pour tout $p \in Q$ :

1. Si $p \xrightarrow{\mathtt{S}\ s} p$, alors $\psi(p) = [s]$ ;
2. Si $p \xrightarrow{\mathtt{T}\ t} p$, alors $\psi(p) = \{t :: x_1 \dots x_n\}$ où

$$x_k = \begin{cases} \mathtt{Int}\ i & \text{si } p \xrightarrow{\mathtt{F}\,(k,i)} p \\ \mathtt{Ptr}\ q & \text{si } p \xrightarrow{\mathtt{P}\ k} q \end{cases}$$

La mémoire $\mathcal{N}$ est bien définie, car il y a d'une part, par déterminisme, au plus un état vers lequel pointe une transition d'un certain type (aucun choix pour $q$ dans le deuxième cas du $x_k$), et d'autre part il n'y a pas de champ $x_k$ mal défini à cause de l'existence d'une bisimulation vers une mémoire bien formée (sinon $\mathcal{M}$ serait mal définie). En outre, $\mathcal{N}$ est bien une mémoire, car tout pointeur appartenant à un tableau structuré pointe bien dans $Q$ par construction.

On vérifie immédiatement que $A$ est bien l'image de $\mathcal{N}$ par $L$. □

**Proposition 4.** *Si $A$ est un LTS déterministe à $n$ états, on peut calculer* $\min A$*, un LTS déterministe équivalent à $A$ de nombre d'états minimal, en temps $O(n \log n)$.*

*Démonstration.* On applique l'algorithme bien connu de Hopcroft [6]. □

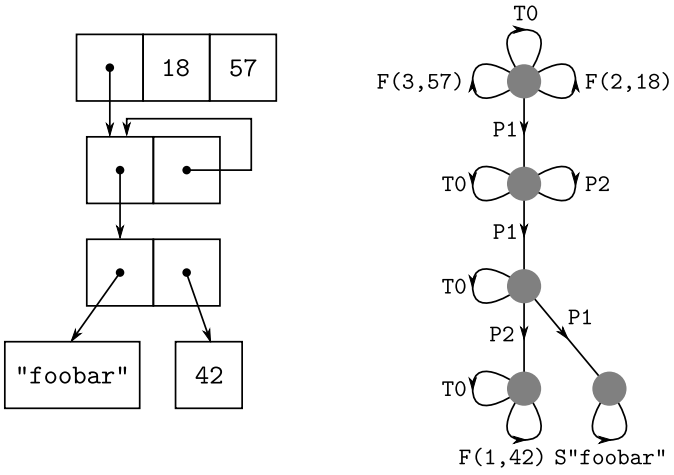


FIGURE 6 – Mémoire et LTS minimaux équivalents à ceux de la figure 5

Enfin, en composant toutes ces propositions, on en déduit le théorème principal.

**Théorème.** *Soit $\mathcal{M}$ une mémoire contenant $n$ blocs. Alors on peut calculer en temps $O(n \log n)$ la mémoire* $\min \mathcal{M}$ *correspondant au partage maximal de $\mathcal{M}$.*

# 5. Implémentation

On discute dans cette section des aspects pratiques de notre méthode.

## 5.1. Choix de conception

Notre implémentation est intégralement écrite en OCaml. Elle est essentiellement décomposable en deux parties :

1. Une réimplémentation du désérialiseur (fonctions `Marshal.from_*`) qui permet d'obtenir directement la réification de la mémoire correspondant à un objet, au lieu de la reconstruire *a posteriori* ;
2. Une implémentation de l'algorithme de Hopcroft ajustée à nos besoins.

Pour des raisons d'efficacité, nous aurions pu choisir d'écrire le programme en C, mais OCaml permettait d'écrire un *proof of concept* rapidement.

Ce choix amène plusieurs désavantages vis-à-vis d'une problématique aussi bas-niveau. Par exemple, notre désérialisation est déjà entre 3 et 4 fois plus lente que celle fournie de la bibliothèque standard d'OCaml. De plus, certaines limitations quant à la taille maximale des tableaux en 32 bits se sont révélées gênantes. Afin de ne pas faire exploser davantage le temps de calcul, notre outil s'appuie sur les tableaux comme représentation interne de la mémoire. Malheureusement, sur des fichiers assez gros, la taille maximale des tableaux [6] est atteinte et empêche la minimisation. Utiliser d'autres structures, comme les arbres binaires, semble introduire un surcoût non-négligeable.

Le choix de passer par le résultat d'une sérialisation a été motivée par deux aspects. D'abord, notre outil se veut être complètement externe, voire générique et indépendant de Coq. Ensuite, la réification

6. Très exactement 4194303 en 32 bits.

de la mémoire en OCaml est assez délicate à implémenter sans passer par le C. La seule structure de donnée qui nous permettrait de vérifier si un objet a déjà été croisé lors du parcours du graphe de la mémoire est la table de *hash*, qui ne nécessite qu'une égalité (ici l'égalité de pointeur) et un *hash* (ici le hash générique). Si sur de petits exemples ce choix est tout à fait acceptable, les collisions apparaissent très rapidement lorsqu'on essaye de passer à l'échelle, provoquant un comportement quadratique.

## 5.2. Utilisation pratique

La généricité de notre outil est telle qu'on pourrait l'imaginer comme le dual de l'option désactivant le partage de la désérialisation d'OCaml (flag `No_sharing` des fonctions `Marshal.to_*`). En effet, en présence de fonctions impures, désactiver le partage est tout aussi incorrect vis-à-vis de la sémantique opérationnelle que de le maximiser comme nous le faisons. Par ailleurs, l'algorithme employé est quasi-linéaire, ce qui peut représenter un surcoût en calcul minime si l'espace mémoire utilisé prime. Enfin, désactiver le partage sur des structures cycliques fait boucler la fonction de désérialisation, alors que notre compacteur est totalement insensible à ces considérations.

Une autre manière d'utiliser ce programme, plus dynamique, se présente sous la forme d'une fonction `share : 'a → 'a` qui construit la représentation minimale d'un objet en mémoire, une sorte de *hashconsing* effectué *a posteriori*. Le code actuel permet déjà d'écrire simplement une telle primitive.

Dans tous les cas, les seules contraintes à respecter afin de garantir la correction de l'algorithme sont d'interdire la modification en place des structures de données après compaction. Comme OCaml ne fait aucune différence à l'exécution entre les structures mutables et les structures pures, on peut tout à fait minimiser des structures mutables ; et tant qu'elles ne seront utilisées qu'en lecture, le pseudo-théorème affirme que la version brute et la version compressée seront observationnellement équivalentes.

Ce n'est plus vrai en présence d'écritures, et du fait de l'insensibilité au typage de la représentation mémoire, cela peut même mener à des erreurs de segmentation, si certains blocs de types incompatibles se retrouvent unifiés parce qu'ils partagent la même représentation mémoire.

En ce qui concerne le domaine d'application, on peut très bien étendre l'algorithme à des structures de données qui ne sont pas du premier ordre ou qui résident hors du tas d'OCaml. Pour ce faire, il suffit d'ajouter un autre type de transition dans la traduction, de la forme `A` $p$, qui représente un pointeur *abstrait*. En se contentant d'associer à $p$ l'adresse mémoire de sa cible, et en appliquant l'égalité de pointeurs sur ces transitions, on obtiendra des états similaires aux états « chaînes de caractères », mais de contenu abstrait, et qui ne seront ainsi jamais unifiées par l'algorithme. Ceci permet d'ailleurs d'écrire une fonction `share` totale, qui se contente de laisser telles quelles les sous-structures qu'elle ne peut analyser.

## 5.3. Benchmarks

Le programme de compression a été utilisé sur la bibliothèque standard de Coq du *trunk* et sur le récent développement du théorème de Feit-Thompson, afin de constater son efficacité sur deux ensembles de fichiers objets plutôt différents. On peut trouver quelques données à ce propos aux figures 7 et 8. Les tests ont été effectués sur un `i686` bicœur cadencé à 3.16 GHz possédant 4 Go de RAM.

La compression est déterministe et les taux de compression sont indépendants de l'architecture, la sérialisation d'OCaml étant multi-plateforme. Ces résultats sont donc parfaitement reproductibles en l'état. Le temps et la mémoire pris par la compilation sont, eux, variables, bien que cohérents d'une exécution à l'autre. Nous avons choisi de ne mettre ici que le temps d'une exécution à titre indicatif.

| | Compilation (s) | RSS (ko) | Espace (ko) | Compression (s) |
|---|---|---|---|---|
| Brut | 870,1 | 29199648 | 121191 | – |
| Compressé | 813,2 | 18528652 | 79515 | 214,8 |
| Variation | 5,9% | 36,5% | 34,4% | – |

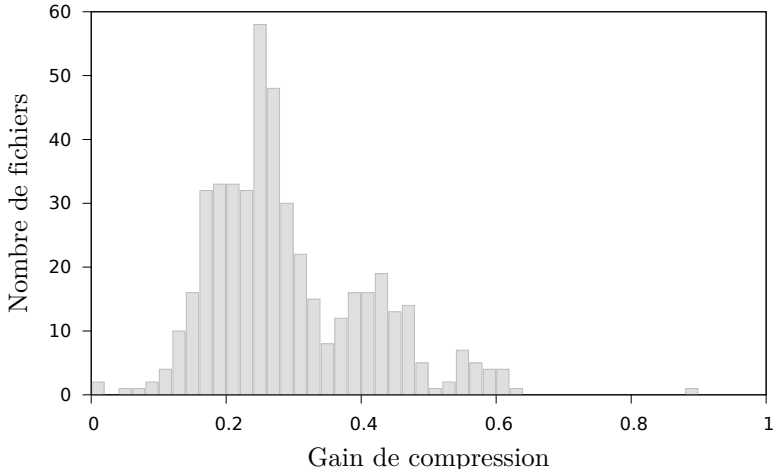


FIGURE 7 – Performances de l'algorithme sur la bibliothèque standard de Coq

| | Compilation (s) | RSS (ko) | Espace (ko) | Compression (s) |
|---|---|---|---|---|
| Brut | 11719,4 | 306242832 | 84544 | – |
| Compressé | 11711,3 | 269859808 | 113709 | 375.7 |
| Variation | 0% | 11,9% | 25,6% | – |

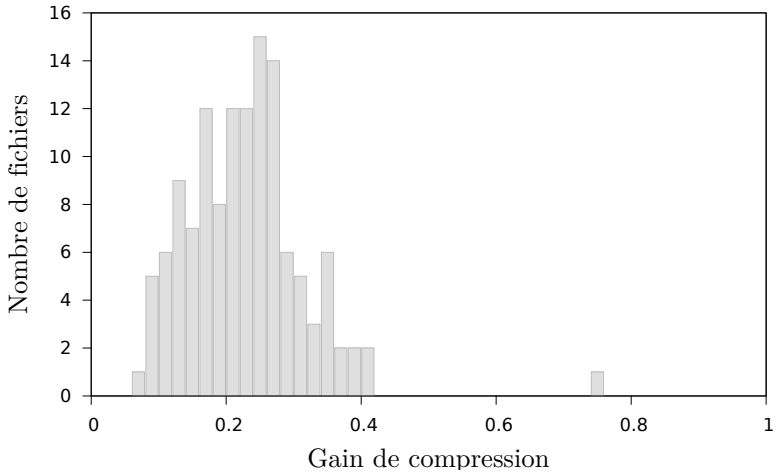


FIGURE 8 – Performances de l'algorithme sur SSReflect + Feit-Thompson

#### 5.3.1. Bibliothèque standard

On constate au premier abord que la compression a un coût de l'ordre du quart de celui de la compilation, ce qui n'est pas très encourageant quand au gain en temps espéré. Le gain en temps de compilation pur n'est que d'environ 6%. Ce léger gain est probablement dû à une empreinte mémoire nettement inférieure, ainsi qu'à l'optimisation de certaines comparaisons structurelles qui se retrouvent effectuées par des comparaisons de pointeurs.

En ce qui concerne le gain en espace, il est effectivement assez important, entre 30 et 40%. Deux fichiers de la bibliothèque standard ne sont pas pris en compte dans ces calculs. Ces deux fichiers mettent en exergue le mauvais partage induit par l'emploi de foncteurs en Coq, qui dupliquent inutilement la mémoire : alors que ce sont les deux plus gros fichiers (de l'ordre de 10 Mo chacun), on gagne effectivement un facteur 5 lors de la compression.

#### 5.3.2. SSReflect + Feit-Thompson

Dans ce cas, la compression a un coût complètement négligeable vis-à-vis du temps de compilation. Elle n'apporte malheureusement aucun gain en efficacité. La compilation de la bibliothèque SSReflect semble pour ainsi dire indépendante des problèmes de partage maximal ; il est évident que l'essentiel du temps de compilation se passe dans les tests de conversion, qui sont effectués de manière externe.

Le gain en espace est inférieur à celui de la bibliothèque standard, environ 25%, sans doute parce que SSReflect n'utilise pas la machinerie des foncteurs de Coq, qui est, comme le souligne le paragraphe précédent, une source importante de redondance mémoire inutile. Remarquons cependant que ce taux est loin d'être portion congrue.

#### 5.3.3. Comparaison avec d'autres algorithmes de compression

Notre compresseur est moins efficace que d'autres outils comme `gzip`, qui atteint, lui, des taux de l'ordre de 40–50% sur les fichiers objets. Cependant, il s'en distingue de plusieurs manières. Notre compresseur n'introduit pas de surcoût lié à la décompression à la volée, puisqu'il ne fait que réduire la représentation elle-même des objets, et est transparent pour le désérialiseur. Mieux, comme évoqué précédemment, il permet un léger gain.

De plus, ces deux techniques sont parfaitement orthogonales : les taux de compression par `gzip` de fichiers objets, qu'ils soient bruts ou compactés par notre outil, sont similaires. Ceci permet d'appliquer les deux techniques avec succès : la bibliothèque standard se retrouve compressée avec un taux de l'ordre de 70%.

#### 5.3.4. Remarques générales

L'implémentation du compresseur n'est pas des plus efficaces, et un passage au C pourrait contrebalancer les pertes dues à la compression d'office lors de la compilation. Par ailleurs, notre outil a plus vocation à être utilisé une unique fois sur les fichiers objets devant être distribués lors de la *release* d'une bibliothèque, permettant de fait des gains en performance lors de l'appel à cette bibliothèque.

L'analyse en détail du contenu des fichiers objets et de leur taux de compression est aussi très informatif. En l'état, les fichiers objets sont très redondants car ils n'offrent aucun partage au delà d'eux-mêmes. Il pourrait être intéressant de développer des fichiers « bibliothèques » correspondant à un ensemble de fichiers objets destinés à être utilisés ensemble. En d'autres termes, un format qui serait aux `vo` ce que les fichiers `cma` sont aux `cmo` en OCaml.

## 6. Conclusion

Poussés par les problématiques d'efficacité de Coq, un programme complexe et nécessitant des garanties statiques de correction, nous avons développé un outil permettant de réduire effectivement aussi bien la mémoire consommée par la compilation d'une preuve que celle prise par les fichiers compilés eux-mêmes. La correction de notre technique repose sur celle d'un algorithme bien connu de la théorie des automates et sur le comportement purement fonctionnel des parties de Coq mises en jeu.

Plus largement, cette technique pourrait facilement être appliquée à un grand nombre de besoins divers de maximisation du partage en OCaml. Il est de fait assez curieux que personne ne semble avoir utilisé à ce jour de tels algorithmes sur la représentation mémoire d'un programme purement fonctionnel. C'est chose faite.

Nous projetons d'écrire une implémentation en C qui puisse s'interfacer aisément avec la sérialisation native d'OCaml, et qui pourrait être distribuée dans une bibliothèque générique à destination des développeurs souhaitant avoir un contrôle fin de la représentation mémoire des objets de leurs programmes.

## Remerciements

Nous remercions chaleureusement Pierre Boutillier pour son aide précieuse lors de la réflexion ayant mené à la création de cet outil et pour son diligent concours aux benchmarks, ainsi qu'Alexis Saurin et Yann Régis-Gianas pour les nombreux retours et suggestions.

## Références

[1] Andrew W. Appel et Marcelo J. R. Goncalves. *Hash-Consing Garbage Collection*. Rap. tech. TR-412-93. Department of Computer Science, Princeton University, fév. 1993, p. 18.

[2] Pascal Cuoq et Damien Doligez. « Hashconsing in an incrementally garbage-collected system : a story of weak pointers and hashconsing in ocaml 3.10.2 ». Dans : *Proceedings of the 2008 ACM SIGPLAN workshop on ML*. ML '08. New York, NY, USA : ACM, 2008, p. 13–22. isbn : 978-1-60558-062-3. doi : 10.1145/1411304.1411308.

[3] Jean-Christophe Filliâtre et Sylvain Conchon. « Type-safe modular hash-consing ». Dans : *Proceedings of the 2006 workshop on ML*. ML '06. New York, NY, USA : ACM, 2006, p. 12–19. isbn : 1-59593-483-9. doi : 10.1145/1159876.1159880.

[4] Georges Gonthier. « The Four Colour Theorem : Engineering of a Formal Proof ». Dans : *Proc. ASCM*. T. 5081. Lecture Notes in Computer Science. Springer, 2007, p. 333.

[5] Hugo Herbelin et al. *The Coq Proof Assistant, Reference Manual*. 2012. url : http://coq.inria.fr/distrib/current/refman/.

[6] J. E. Hopcroft. « An $n \log n$ algorithm for minimizing the states in a finite automaton ». Dans : *Theory of Machines and Computations*. Sous la dir. de Z. Kohavi. Academic Press, NY, USA, 1971, p. 189–196.

[7] Xavier Leroy. « Formal verification of a realistic compiler ». Dans : *Communications of the ACM* 52.7 (2009), p. 107–115.

[8] Xavier Leroy et al. *The OCaml system, Documentation and user's manual*. 2012. url : http://caml.inria.fr/pub/docs/manual-ocaml/.

[9] J. Rutten. « Automata and coinduction (an exercise in coalgebra) ». Dans : *CONCUR'98 Concurrency Theory*. Sous la dir. de Davide Sangiorgi et Robert de Simone. T. 1466. Lecture Notes in Computer Science. 10.1007/BFb0055624. Springer Berlin / Heidelberg, 1998, p. 194–218. isbn : 978-3-540-64896-3.

[10] Didier Rémy. « Using, Understanding, and Unraveling the OCaml Language. From Practice to Theory and Vice Versa. » Dans : *APPSEM*. Sous la dir. de Gilles Barthe et al. T. 2395. Lecture Notes in Computer Science. Springer, 2000, p. 413–536. isbn : 3-540-44044-5.

[11] Jeremy Yallop. « Practical generic programming in OCaml ». Dans : *Proceedings of the 2007 workshop on Workshop on ML*. ML '07. New York, NY, USA : ACM, 2007, p. 83–94. isbn : 978-1-59593-676-9. doi : 10.1145/1292535.1292548.