Permutations et liens dansants vérifiés en CFML

Rapport de stage de recherche L3

Sébastien Michelland, sous la direction de François Pottier

1 Introduction

J'ai passé 6 semaines aux côtés de l'équipe Gallium à l'Inria Paris pour travailler sur un exercice de vérification formelle. Là où des programmes interviennent sur des systèmes industriels un minimum sensibles (cryptographie, électronique, aéronautique par exemple), les preuves formelles assurent un niveau de confiance élevé envers le code. L'équipe Gallium, à l'origine du langage OCaml, s'intéresse entre autres à la formalisation et à la preuve de programmes réels.

L'objectif de ce stage était de prouver dans Coq l'algorithme des liens dansants de Knuth, qui résoud le problème Exact Cover. La difficulté principale est de modéliser de façon simple sa structure de données, une matrice creuse chaînée. L'idée de François Pottier était d'utiliser des permutations pour représenter les listes ; ce choix simplifie grandement la structure et sa formalisation.

J'ai implémenté en OCaml une nouvelle version des liens dansants en suivant cette idée. J'ai ensuite modélisé Exact Cover dans Coq et prouvé que la résolution usuelle par backtracking est correcte. Enfin, j'ai commencé une preuve Coq de la correction de mon implémentation en m'appuyant sur la version abstraite.

Plan du rapport

La section 2 présente le problème de couverture exacte et l'algorithme des liens dansants de Knuth. La section 3 introduit les notions de logique de séparation nécessaires à la preuve d'un programme OCaml, et décrit quelques techniques de vérification formelle de programmes.

J'expose dans la section 4 les choix qu'on a faits pour implémenter l'algorithme de Knuth avec des permutations, et leur intérêt pour la preuve. J'explique dans la section 5 les enjeux majeurs de la preuve, ainsi que les principales difficultés qui se présentent ; pour certaines, des solutions.

2 Couverture exacte et liens dansants

Cette section introduit le problème Exact Cover et sa résolution par backtracking. L'algorithme des liens dansants est un raffinement de ce backtracking ; c'est l'objet de ma preuve.

2.1 Algorithme de backtracking pour Exact Cover

Exact Cover : C un ensemble, $R \subseteq \mathcal{P}(C)$.
Existe-t-il un sous-ensemble de R qui forme une partition de C ?

Ce problème classique fait partie des 21 problèmes NP-complets de Karp [1]. On modélise souvent une instance comme une matrice de booléens A : les colonnes représentent les éléments de C (pour *columns*) et les lignes ceux de R (pour *rows*). Si $c \in C$ et $r \in R$, alors $A[r, c] = 1$ ssi $c \in r$. J'utiliserai toujours ce modèle, même si j'emploie parfois du vocabulaire ensembliste quand c'est plus clair.

$$A = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \qquad \begin{aligned} C &= \{x, y, z\} \\ R &= \{\{z\}, \{x, y\}, \{x, z\}\} \end{aligned}$$

Figure 1 – Représentation matricielle pour Exact Cover

Ce problème se prête bien à l'utilisation de backtracking ; un algorithme typique consiste à construire progressivement une solution $S \subseteq R$. On démarre avec $S = \emptyset$ et on ajoute à chaque étape à S une nouvelle ligne r ; puis on s'exécute récursivement sur une *matrice réduite* en ayant retiré de A :

— Toutes les lignes *en collision* avec r, qui briseraient la propriété de partition de S si elles étaient choisies dans une étape future ;
— Toutes les colonnes *couvertes* par r, qui ne doivent (et ne peuvent pas) être couvertes à nouveau.

Pour construire et afficher les solutions il faut passer la solution partielle en paramètre de l'algorithme. Par simplicité, je ne m'intéresse qu'au nombre de solutions et je ne construis pas S, pour me concentrer sur les opérations matricielles.

algorithme	Backtracking(A)
entrée	Une matrice de booléens A ; on nomme ses lignes R et ses colonnes C
sortie	Le nombre n de parties de R qui forment une partition de C

Si A n'a pas de colonnes :
  (Si l'on veut construire S) Afficher la solution S.
  Renvoyer 1 et s'arrêter.
Choisir de façon déterministe une colonne $c \in C$.
$total \coloneqq 0$
Pour chaque $r \in R$ tel que $A[r, c] = 1$:
  (Si l'on veut construire S) Ajouter r à la solution partielle S.
  $C' \coloneqq C - \{c' \in C \mid c' \in r\} = C - r$
  $R' \coloneqq R - \{r' \in R \mid r \cap r' \neq \emptyset\}$
  Construire la *matrice réduite* A' de colonnes C' et de lignes R'.
  $total \coloneqq total +$ Backtracking(A')
Renvoyer $total$.

Algorithme 1 – Décompte des solutions d'une instance matricielle d'Exact Cover

Notez que c fait toujours partie des colonnes couvertes ($c \in C - C'$) et r des lignes en collision ($r \in R - R'$) ; on n'a donc pas besoin de les traiter spécialement au moment de calculer la matrice réduite (qui ne dépend de toute façon que de r).

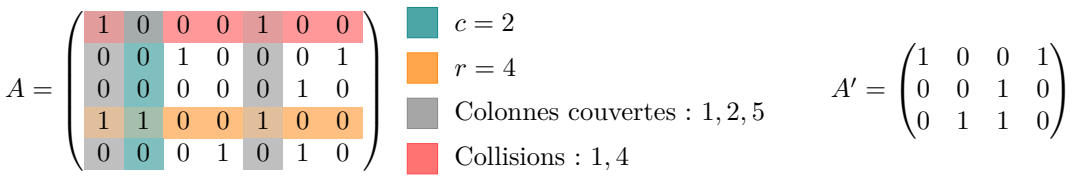


FIGURE 2 – Une étape de backtracking et sa matrice réduite

J'ai formalisé le problème dans Coq en considérant $C = \{1 \dots m\}$ et $R = \{1 \dots n\}$; dans cette situation les colonnes et les lignes sont des indices entiers dans la matrice. Pour éviter de re-numéroter les lignes et colonnes, il est préférable de ne pas modifier directement la matrice. Voyez la section 5.1.

2.2 Les liens dansants de Knuth

Il reste à choisir, dans BACKTRACKING, une structure de données pour la matrice. L'algorithme des *liens dansants* (dit DLX) de Knuth [2] utilise une matrice creuse et plusieurs optimisations. La matrice creuse s'obtient en chaînant verticalement et horizontalement les 1 tout en omettant les 0 :

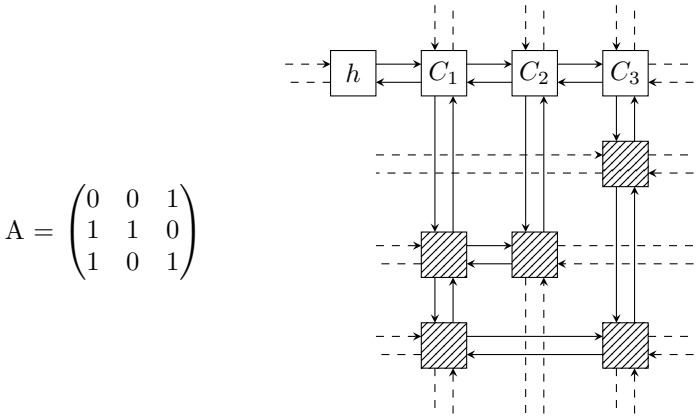


FIGURE 3 – Représentation creuse pour une matrice de booléens

Les chaînages sont doubles et cycliques. Pour accéder aux contenus des colonnes sans connaître la matrice, on place des *en-têtes de colonnes* C_i au sommet et la *racine du problème* h dans la liste des en-têtes. Ces objets servent de points d'entrée dans les listes chaînées ; ils évitent d'avoir à garder la trace d'un « premier élément ».

Cette structure souple permet de supprimer rapidement des lignes ou des colonnes. Pour supprimer une colonne par exemple, on retire tous ses éléments de leurs listes horizontales respectives. Je ne détaille pas les opérations sur les pointeurs, préférant les abstraire derrière l'idée d'une matrice rectangulaire.

L'avantage crucial au moment de changer de r durant l'exploration non-déterministe est que *la suppression d'une ligne ou d'une colonne peut être annulée*. Quand un élément y est retiré d'une liste doublement chaînée, si les pointeurs vers ses voisins x et z sont conservés, alors on peut le réinsérer en temps constant. Je parlerai de *masquage* au lieu de *suppression*. La première utilisation de cette technique servait à accélérer la résolution du problèmes des n dames. [3]

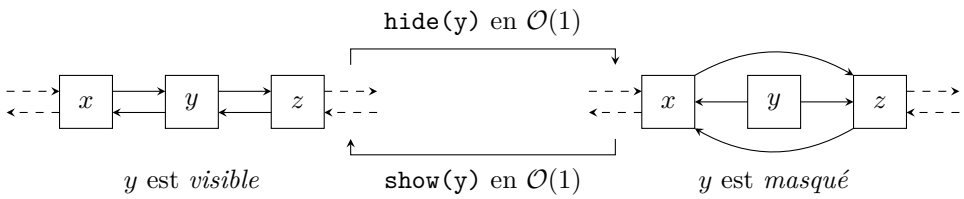


FIGURE 4 – Couverture et affichage d'un élément de liste doublement chaînée

Il est possible de masquer plusieurs éléments en séquence, mais il faut les démasquer dans l'ordre inverse pour retrouver la liste d'origine car les appels à `show` ne commutent pas.

Grâce à cette technique, il est possible d'éviter complètement l'allocation d'une nouvelle matrice réduite. Avant de passer à la récurrence, on cache les lignes et colonnes appropriées et *on réutilise la même matrice*. Après l'appel récursif, on démasque ce qui avait été masqué et on continue l'exploration.

J'ai implémenté et formalisé la matrice creuse en utilisant des permutations comme type abstrait pour les listes chaînées, afin de réduire la quantité de pointeurs. Les fonctions `hide` et `show` sont centrales et ont été étudiées puis prouvées en détail ; ce travail se trouve dans les sections 4.1 et 4.2.

3 Logique et formalisme de preuve

Cette section expose les grands principes de logique de séparation utiles à la preuve d'un programme OCaml. J'y présente également des travaux liés à la vérification formelle.

3.1 Logique de séparation

La logique de séparation est une extension de la logique de Hoare. La logique de Hoare permet de raisonner sur des programmes en leur attribuant des triplets $\{P\}\, f\, \{Q\}$ dits *spécifications*, où :

— P est une formule logique, la *pré-condition* (qui parle typiquement des arguments de f) ;
— f est une fonction dont le comportement est étudié ;
— Q est une fonction ; à la valeur de retour de f, elle associe une formule logique, la *post-condition*.

Le triplet $\{P\}\, f\, \{Q\}$ affirme que quand l'état du programme vérifie P, tout appel à f s'exécute sans erreur et renvoie une valeur x ; de plus, l'état du programme après l'appel vérifie $Q\, x$. Décrire formellement un programme dans la logique de Hoare consiste à lui attribuer une spécification, et le vérifier revient à prouver cette spécification. À titre d'exemple, une spécification possible pour Backtracking est :

> $\forall A$,
> $\{A$ est une matrice de booléens$\}$
> Backtracking(A)
> $\{\lambda n.\ n$ est le nombre de partitions des colonnes de A dans les lignes de $A\}$

La logique de séparation étend ce système avec un *tas* qui est intuitivement le tas d'allocation du programme, et des *prédicats de représentation* de la forme $\underline{p} \rightsquigarrow X$, où $\underline{p}$ est un pointeur (je souligne les pointeurs pour les distinguer). $\underline{p} \rightsquigarrow X$ indique que l'adresse p du tas est occupée par l'objet X. Ce tas permet de raisonner sur des structures de données en présence de partage, comme une liste doublement chaînée, pour laquelle il y a deux pointeurs vers chaque nœud, mais une seule instance des nœuds.

Prédicats de représentation dans Coq

Dans Coq, les prédicats de représentation sur un tas sont de type `Prop`. Différentes implémentations sont possibles ; j'utiliserai celle de CFML, la bibliothèque Coq que j'ai employée dans mes preuves, pour laquelle :

— Les propriétés sur les tas sont de type `hprop` $\equiv$ `heap` $\to$ `Prop`.
— Le prédicat `p ~> X` est défini comme `X p`.

Le terme *prédicat de représentation* désignera alors X, auquel on ajoute souvent des paramètres supplémentaires. Un exemple typique est le prédicat de représentation polymorphe `List` :

```
Definition List : forall (A: Type), A -> list A -> loc -> hprop := ...
```

Ainsi $p \rightsquigarrow$ `List` $[2;3]$ signifie que p est le point d'entrée d'une liste chaînée qui contient les entiers 2 et 3. (Le paramètre `A` $\equiv$ `int` est implicite, et déduit du fait que $[2;3]$ est de type `int list`.) Chaque type de record a un prédicat dédié, que je note avec la syntaxe habituelle $p \rightsquigarrow \{x = v_x, \ldots\}$.

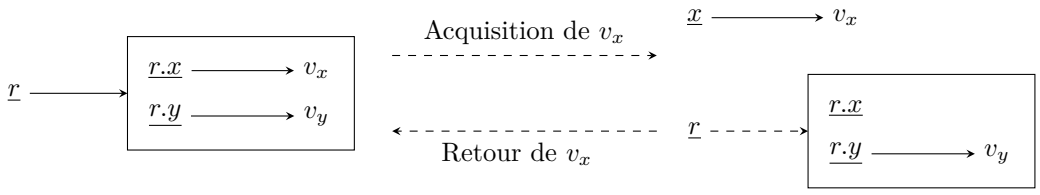
3.2 Possession des objets du tas

Un prédicat de représentation $\underline{p} \rightsquigarrow X$ ne décrit pas que l'état de la mémoire ; il signifie aussi que l'on *possède* X. C'est crucial en ceci qu'on ne peut lire et écrire que les objets que l'on possède. Dans la logique de séparation traditionnelle utilisée ici, chaque objet n'a *qu'un seul propriétaire*, ce qui introduit de nombreuses subtilités.

Supposons que l'on possède à un moment de l'exécution un pointeur $\underline{r}$ vers un record à deux champs $\{x = v_x, y = v_y\}$, de sorte que le pointeur $\underline{r.x}$ pointe vers v_x, et le pointeur $\underline{r.y}$ vers v_y. Pour bien passer à l'échelle quand les structures de données sont imbriquées, il est d'usage de considérer que « le record possède v_x et v_y ». Avoir un prédicat de représentation $\underline{r} \rightsquigarrow \{x = v_x, y = v_y\}$ implique l'existence de prédicats $\underline{r.x} \rightsquigarrow v_x$ et $\underline{r.y} \rightsquigarrow v_y$, mais qui sont cachés du propriétaire de $\underline{r}$.

Si toutefois on veut utiliser la valeur v_x, on doit acquérir *le* prédicat de représentation sur v_x (comme il ne peut y avoir qu'une seul propriétaire, il ne peut y avoir qu'un seul prédicat). Cela « casse » le record, qui ne possède plus que v_y. La logique de séparation permet de formaliser cette situation et de travailler avec x jusqu'à ce qu'on rende le prédicat de représentation au record.

Dans le schéma ci-dessous, les flèches pleines matérialisent les prédicats de représentation.



Situation 1 : On possède le record, mais pas directement v_x et v_y

Situation 2 : On a extrait v_x, mais le record est maintenant incomplet

FIGURE 5 – Acquisition d'un seul des champs d'un record

Ce problème est traité en détail par Arthur Charguéraud [4]. Il propose différentes solutions pour gérer les allées et venues des objets entre propriétaires en présence de conteneurs. Pour mon implémentation des liens dansants, deux sont utiles : une pour les records et une pour les tableaux.

Possession des records

```
type segment = { src: point; dst: point }
```

Considérons à titre d'exemple un record dont les deux attributs `src` et `dst` résident dans le tas. Comme précédemment, on décide que le record « détient ses champs », et le prédicat $\underline{\texttt{s}} \rightsquigarrow \{\texttt{src} = p_1, \texttt{dst} = p_2\}$ implique l'existence de deux points p_1 et p_2 dans le tas. Dans l'état normal du record, dit *fermé* $\square$, les prédicats $\underline{\texttt{s.src}} \rightsquigarrow p_1$ et $\underline{\texttt{s.dst}} \rightsquigarrow p_2$ sont cachés dans `s` et on ne dispose que du prédicat complet sur le record.

Pour utiliser les champs, on ouvre `s`, ce qui nous transfère les prédicats cachés. `s` est alors dans l'état *ouvert* $\square$ et ne possède plus p_1 ni p_2 ; on ne peut donc plus affirmer que $\underline{\texttt{s}} \rightsquigarrow \{\texttt{src} = p_1, \texttt{dst} = p_2\}$. Si on veut réutiliser le record entier ou simplement en rendre la possession en fin de fonction, il faut le refermer et rendre la possession de p_1 et p_2.

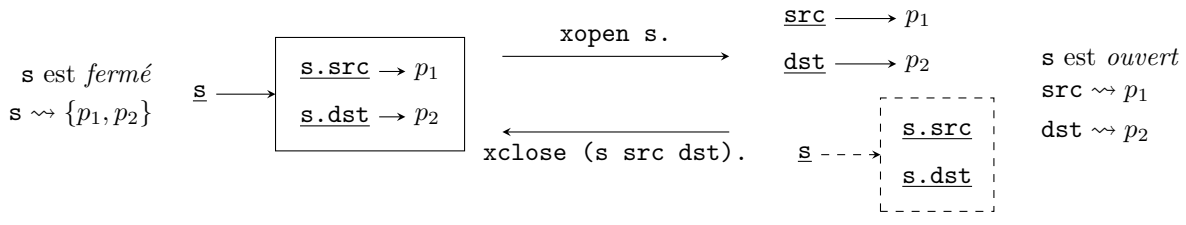


FIGURE 6 – Les deux modes de possession d'un record : fermé et ouvert

Possession des tableaux

Comme pour les records, on aimerait que les tableaux « possèdent leurs éléments » pour améliorer l'abstraction de la structure de données. Bien sûr on doit y déroger temporairement lorsqu'on veut les utiliser. Arthur Charguéraud propose de laisser au tableau la possession d'un *sous-ensemble* des éléments. Le prédicat de représentation `Array` possède alors plusieurs paramètres :

— T, la liste des éléments du tableau
— R, le prédicat de représentation des éléments (utilisé quand les éléments sont des pointeurs)
— *params*, une liste donnant les paramètres de R pour chaque élément
— S, le sous-ensemble des indices possédés par le tableau.

$$\texttt{p} \rightsquigarrow \texttt{Array T R params S} \equiv \texttt{p} \rightsquigarrow \texttt{AllocatedArray T} * \circledast_{i \in S} \texttt{T[i]} \rightsquigarrow \texttt{R params[i]}$$

Ici `AllocatedArray` est une primitive signifiant que les éléments de T sont chargés en mémoire à l'adresse p. On l'étend avec la réunion des prédicats de représentation des éléments possédés par le tableau. L'opérateur $*$ indique que les adresses `p` et `T[i]` sont toutes différentes, pour éviter d'avoir plusieurs prédicats sur un même objet.

Par exemple, avec un tableau de 5 éléments $x_0 \dots x_4$ et $S = \{0, 2, 3\}$, on possède les prédicats suivants :

$\texttt{T} \rightsquigarrow \texttt{Array } (x_i)_i \texttt{ R params } \{0, 2, 3\}$
$x_1 \rightsquigarrow \texttt{R params[1]}$
$x_4 \rightsquigarrow \texttt{R params[4]}$

En pratique c'est une technique assez lourde à utiliser. Pour les liens dansants, il y a rarement plus d'un élément hors du tableau [1], mais dans le cas général surveiller qui est dedans ou dehors est fastidieux.

3.3 Vérification déductive de programmes

Une fois le cadre de la logique de séparation posé, on est en mesure d'exprimer et de prouver formellement des spécifications dans des assistants de preuve. Différents outils implémentent différentes approches ; j'en décris ici trois.

Construction par tactiques dans Coq (CFML)

CFML est un outil de vérification d'OCaml développé par Arthur Charguéraud [5] sur le principe des *formules caractéristiques* qu'il a introduit dans sa thèse. La formule caractéristique d'une fonction est une formule logique qui décrit son comportement en reformulant son code. Il s'agit de sa spécification la plus générale mais aussi la plus « primitive », de laquelle on dérive des spéficiations plus abstraites durant les preuves.

L'outil en lui-même consiste en un générateur qui compile le code OCaml en formules caractéristiques exprimées en Coq, et une bibliothèque Coq qui fournit un jeu de tactiques et de théorèmes pour manipuler les triplets de Hoare, les assertions sur les tas, et des primitives pour les objets d'OCaml.

Arthur Charguéraud étant un ancien membre de l'équipe Gallium où j'ai travaillé, le choix de CFML était naturel. Cela suppose l'utilisation de la bibliothèque standard TLC [6] qui fournit des tactiques et structures de données d'usage général. L'annexe B décrit les grandes lignes de TLC et CFML, utiles à la lecture de mes scripts.

Utilisation de prouveurs SMT externes (Why3)

Une autre approche est celle de Why3. [7] Il s'agit d'un framework qui prouve de façon semi-automatique des programmes écrits en WhyML, une variante d'OCaml annotée avec (entre autres) les spécifications des fonctions et des définitions utiles aux preuves.

1. La logique de séparation possède un outil idiomatique, appelé *magic wand* et noté $P \mathrel{-\!\!*} Q$, qui représente sensiblement Q privé de P (si on rend P, on retrouve Q). Il serait utile dans ce cas, mais CFML ne le propose pas.

Why3 analyse le programme fourni en entrée et dérive des conditions minimales de preuve dont il délègue la preuve à des solveurs SMT comme Z3 ou Alt-Ergo, en temps limité. Une grande partie du travail de Why3 est de reformuler les conditions de preuve dans les logiques de tous les solveurs utilisés. Une fois la preuve effectuée, Why3 génère un programme OCaml classique certifié.

Ce système permet de tirer le meilleur parti des différents solveurs, quitte à en utiliser plusieurs pour prouver une seule fonction. Dans le cas où la vérification s'avère trop difficile, l'utilisateur de Why3 peut formuler une preuve interactive.

Recherche de preuves automatiques dans Isabelle (auto2)

auto2 est un prouveur automatique pour Isabelle, introduit par Bohua Zhan. Il offre une approche encore différente en recherchant exhaustivement des nouveaux faits à partir de ses hypothèses. À chaque *étape de preuve*, il applique automatiquement des théorèmes pour obtenir de nouveaux termes, jusqu'à obtenir une preuve de son objectif.

Son implémentation des prédicats de représentation suppose un tas qui ne contient que des entiers ou des tableaux d'entiers. Un tableau ou un record n'y occupe qu'une seule adresse. Cela contraste avec CFML où l'on peut introduire de nouveaux prédicats primitifs, quitte à les implémenter différement selon les langages (arithmétique d'adresses en C par exemple).

auto2 est capable de prouver efficacement des structures de données type arbre rouge-noir ou union-find [8]. Le degré d'automatisation remarquable permet de conclure certaines étapes de preuve en indiquant seulement « raisonnement par induction sur x », ce qui élimine de nombreux détails de logique de séparation que l'utilisateur de CFML doit gérer.

Le projet VOCaL

Le projet VOCaL (*The Verified OCaml Library*) [9], auquel participent entre autres l'équipe Gallium et l'équipe qui développe Why3, a pour objectif de prouver une bibliothèque de structures de données et d'algorithmes pour OCaml. On y trouve différentes preuves en CFML ou en Why3, par exemple des tables de hachage ou des graphes de coercions.

En plus de la correction de l'implémentation, le projet s'intéresse aussi à prouver la complexité des algorithmes, notamment via un système de « crédits-temps » intégré à CFML.

Mon travail sur les liens dansants ne s'inscrit pas explicitement dans ce projet, mais constitue un exercice de vérification formelle intéressant.

4 Implémenter DLX sur des permutations

Cette section décrit l'implémentation de DLX que j'ai écrite, à base de permutations. J'y détaille les choix faits pendant la phase de code et leur influence sur la formalisation de l'algorithme. Le code est disponible sur la forge de l'INRIA [2] et sur ma page personnelle [3].

4.1 Représentation des listes chaînées par des permutations

L'idée de départ de François Pottier était d'utiliser des permutations pour représenter des listes : l'image d'un élément par la permutation est son successeur dans la liste. On obtient naturellement un cycle ; si des éléments sont cachés, on restreint le domaine D du cycle aux éléments visibles et on évite de spécifier ce qu'il y a en-dehors.

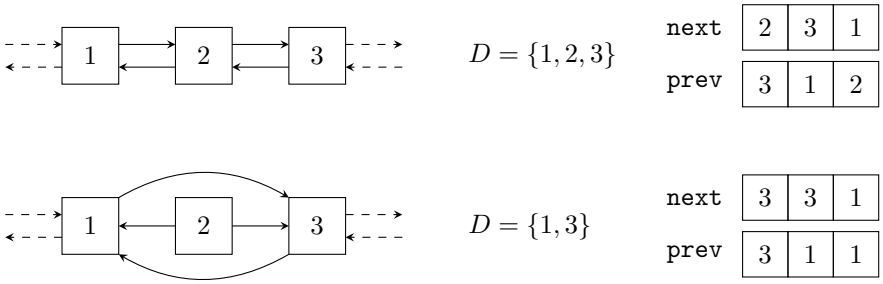


FIGURE 7 – Représentation de listes doublement chaînées par des cycles

On peut alors modéliser une instance d'EXACT COVER avec un record à trois élements :

```
type cover_problem = {
    (* Liste des en-têtes "C_i" (1 <= i <= |C|), plus "h" (élément 0) *)
    head: permutation;
    (* cols[i] est la liste des éléments présents dans la colonne i
       L'élément 0 de chaque cols[i] est l'en-tête C_i *)
    cols: permutation array;
    (* rows[j] est la liste des éléments présents dans la ligne j
       L'élément 0 de chaque rows[j] est inutilisé *)
    rows: permutation array;
}
```

L'espoir était de simplifier la preuve en séparant les permutations des liens dansants. Même si la réalité s'est montrée plus compliquée, les permutations ont d'autres avantages majeurs :

— Elles réduisent considérablement la quantité de pointeurs et rigidifient la structure : on passe d'une matrice creuse à 4 pointeurs par élément et un chaînage complexe à deux tableaux de tableaux.
— Elles *découplent les lignes des colonnes* car les données représentant les chaînages verticaux et horizontaux sont stockées dans des permutations séparées. Cela permet de modifer des lignes pendant un parcours sur une colonne sans avoir à justifier que le parcours n'est pas gêné.

Du point de vue Coq, les permutations sont définies sur un domaine `D` $\subseteq \{1..\texttt{n}\}$ par leur fonction ϕ. L'utilisateur de la permutation doit exhiber son inverse ψ sur `D` pour obtenir les invariants et les théorèmes. Cela peut paraître fastidieux, mais seules `hide` et `show` ont besoin de le faire, donc ça se passe bien. Le prédicat de représentation tient aussi compte de la taille du tableau :

```
Definition Permutation (n: int) (D: set int) (phi psi: int -> int) := ...
```

C'est un formalisme que j'aurais aimé alléger car les permutations sont dans toutes les spécifications, et il faut utiliser des propriétés indépendantes pour exploiter le fait que c'est un cycle sur `D` (indispensable pour les itérations).

2. https://gitlab.inria.fr/sebmiche/dancing-links
3. https://perso.ens-lyon.fr/sebastien.michelland/dlx/

4.2 La fonction `without`

La section 2.2 explique comment masquer et démasquer des éléments de liste chaînée. Cette technique permet de réaliser des suppressions temporaires dans la matrice, que l'on annule pendant le backtracking.

Lorsqu'on masque plusieurs objets en séquence (ceux d'une ligne par exemple), il est naturel de les démasquer dans l'ordre inverse, de sorte que les appels à `hide` et `show` forment un système bien parenthésé. On peut alors effectuer des masquages complexes en imbriquant les appels à une unique fonction `without` qui retire temporairement un élément et exécute une fonction passée en paramètre :

```
(* Executes [f] in a context where [y] has been removed from [p] *)
let without p y f =
  hide p y;
  let x = f () in
  show p y;
  x
```

La gestion de la possession de `p` est délicate. Souvent la sous-fonction a besoin d'accéder à `p`, donc `without` lui en prête la possession. Mais le démasquage ne peut fonctionner que si `p` est rendue intacte : une des préconditions de `without` est donc une *spécification pour `f`*, qui garantit l'invariance de `p` durant l'appel. Cela fait de la spécification de `without` une *spécification d'ordre supérieur*.

L'allure de la spécification de `without` est donnée ci-dessous :

— Pour simplifier, on suppose que `p` est entièrement caractérisée par son domaine `D` ;
— La notation pour les spécifications est
`app <fonction> [<arguments...>] PRE <pré-condition> POST <post-condition>`
— Le prédicat `[= <valeur>]` indique la valeur de retour.

```
Lemma without_spec:
  forall (p: permutation) (D: set int) (y: int),
  forall f (RetType: Type) (x: RetType),

  (* Si [y] fait partie du domaine [D], ie. n'est pas déjà masqué... *)
  y ∈ D ->

  (* Si [f] reçoit [p] sans [y] et le rend inchangé, renvoyant x... *)
  (app f [p]
    PRE  p ~> Permutation (D - {y})
    POST p ~> Permutation (D - {y}) * [= x]) ->

  (* ... alors [without p y f] ne change pas [p] et renvoie x. *)
  app without [p y f]
    PRE  p ~> Permutation D
    POST p ~> Permutation D * [= x].
```

On remarque que la spécification de `f` et celle de `without` sont proches : ce n'est pas un hasard car très souvent, la sous-fonction va elle-même cacher de nouveaux élements.

Prouver cette spécification n'est pas trop difficile. Mais s'en servir est fastidieux pour des raisons liées à la possession de `p` et abordées dans la section 5.2.

4.3 Structure de l'implémentation OCaml

Le code est divisé comme suit :

`Permutation.ml`	Fonctions indépendantes sur les permutations
`DancingLinks.ml`	Implémentation de DLX et outils de génération des instances
`queens.ml`	Résolution du problèmes des n dames

Le fichier `queens.ml` permet de tester l'algorithme en dénombrant les solutions au problème de n dames suivant la modélisation proposée par Knuth [2] : les éléments à couvrir sont les lignes, les colonnes et les diagonales de l'échiquier. Couvrir les diagonales est optionnel, ce qui fait de ce problème un cas de *couverture exacte généralisée*. Cette extension se résoud identiquement avec l'algorithme DLX en modifiant légèrement la liste des en-têtes.

Seuls `Permutation.ml` et la première partie de `DancingLinks.ml` ont été étudiés dans Coq. Les fonctions importantes se partagent l'algorithme de la façon suivante :

`Permutation.ml`
— `hide` et `show` implémentent le va-et-vient de la figure 4 (page 3) ;
— `without` assure le bon parenthésage des appels à `hide` et à `show` ;
`DancingLinks.ml`
— `choose_column` choisit une colonne $c \in C$ pour progresser ;
— `select_row` parcourt la ligne $r \in R$ choisie pour satisfaire c ;
— `cover_column` cache chaque colonne satisfaite par r ;
— `cover_element` cache au passage les lignes en collision avec les précédentes ;
— `dlx` coordonne tout ça.

4.4 Élimination des optimisations de Knuth

L'algorithme DLX original [2] contient deux optimisations qui compliquent la preuve.

À chaque étape de l'algorithme, on choisit une nouvelle colonne $c \in C$ à couvrir, puis on explore toutes les lignes $r \in R$ capables de couvrir c ; avant de passer à la récurrence, on masque toutes les colonnes couvertes par r. Bien sûr cela inclut toujours c. Plutôt que de masquer et démasquer c à chaque branche explorée, Knuth la masque *avant* de commencer à lister les r.

Couvrir une colonne c' nécessite de masquer toutes les lignes qui l'intersectent, sinon on n'a plus $R \subseteq \mathcal{P}(C)$ et donc plus de matrice. Rappelons que masquer une ligne consiste à retirer ses éléments de leurs colonnes respectives. Knuth ne retire pas les éléments dont la colonne est c', car il sait que c' va aussi disparaître de la matrice.

Mon implémentation défait ces optimisations pour éviter de les prouver. Cela simplifie la sémantique de plusieurs fonctions de `DancingLinks.ml` en contrepartie d'une exécution un peu plus lente.

4.5 Comparaison avec une implémentation impérative

Avant de se lancer dans la preuve, on voulait s'assurer que les performances étaient raisonnables ; l'occasion de faire un peu de compilation, un autre sujet de Gallium. Comme comparaison, on a utilisé la bibliothèque `combine` de Rémy El Sibaïe et Jean-Christophe Filliâtre [10] qui contient une implémentation impérative de DLX et du problèmes des n dames.

Niveau mémoire, on constate que les allocations de clôtures ou références sont conséquentes (exponentielles en n, comme le nombre de solutions). Il faut descendre très bas dans l'impératif pour les éliminer. Heureusement, l'impact sur le temps d'exécution est négligeable grâce à l'allocateur rapide d'OCaml.

Implémentation	**Temps**
`combine` de Jean-Christophe Filliâtre	`7.64 s`
Mon implémentation (version fonctionnelle)	`12.53 s`
Mon implémentation (version impérative)	`8.33 s`
Transcription C de ma version impérative	`3.90 s`

TABLE 1 – Résolution du problèmes des 14 dames avec différentes implémentations de DLX

En termes de vitesse, le constat est que les implémentations impératives s'exécutent toujours plus vite car elle s'optimisent mieux (inlining). Remplacer l'itération cruciale, qui était récursive, par une boucle `while` améliore les performances de 15 % !

Ces résultats sont raisonnables (asymptotiquement) : l'enjeu était juste de ne pas prouver une version complètement inefficace. On a donc conservé la version fonctionnelle, plus simple à formaliser dans Coq.

5 Prouver formellement les permutations dansantes

Cette section présente la dernière partie de mon travail : une preuve partielle de mon implémentation, accompagnée d'une vérification formelle de l'algorithme Backtracking. Les détails de Coq seraient fastidieux et inutiles ; je me concentre ici sur les difficultés théoriques et quelques solutions. La preuve est disponible avec le reste du code, et est structurée comme ceci :

`LibPermutation.v`	Propriétés abstraites sur les permutations
`Permutation.v`	Preuve du module `Permutation` (CFML)
`Extensions.v`	Lemmes d'usage général susceptibles d'être intégrés à TLC
`ExactCover.v`	Preuve de l'algorithme Backtracking
`DancingLinks.v`	Preuve partielle de mon module `DancingLinks` (CFML)

5.1 Preuve de l'algorithme pour Exact Cover

Pour simplifier la preuve de mon programme OCaml, j'ai d'abord formalisé et prouvé la validité de l'algorithme Backtracking dans un fichier indépendant `ExactCover.v`. La preuve des liens dansants fait régulièrement référence à ce code pour mettre en relation la progression dans le programme avec les différentes étapes de calcul de l'algorithme.

Dans ma première tentative, je créais une nouvelle matrice avant de passer à la récurrence. C'est une mauvaise idée pour plusieurs raisons : d'abord je perdais l'identité des lignes et des colonnes (impossible de dire où les lignes et les colonnes de la sous-matrice étaient dans la matrice d'origine), d'autre part modifier des listes de listes est très fastidieux. J'ai donc choisi de ne jamais modifier la matrice, et de matérialiser à la place les ensembles `C` et `R` des colonnes et des lignes non cachées.

Les cinq axiomes qui définissent les instances sont résumés ci-dessous. J'impose aux lignes d'être non vides car les lignes vides ne peuvent pas être représentées par les liens dansants. Le prédicat `index x i` se traduit par $0 \leq i < x$ si x est un entier, $0 \leq i < |x|$ si x est une liste.

```
Record Matrix (A: list (list bool)) (n m: int) (R C: set int) := {
  Mheight : length A = n;
  Mwidth  : Forall (fun row => length row = m) A;
  Minv    : forall r, r ∈ R -> exists c, c ∈ C /\ A[r][c] = true;
  Mrows   : forall r, r ∈ R -> index n r;
  Mcols   : forall c, c ∈ C -> index m c;
}.
```

Le lemme intermédiaire le plus important est `reduced_matrix` : il sert à construire la matrice réduite et prouve qu'elle satisfait bien les invariants d'une instance. Le théorème qui conclut la preuve de l'algorithme est `backtracking_step`. Il démontre que l'étape de backtracking est valide ; après avoir choisi une colonne `c` et une ligne appropriée `r` pour la couvrir, les solutions de l'instance originale sont les solutions de l'instance réduite, étendues avec `r`.

```
forall A n m R C (M: Matrix A n m R C) (S: set int) (c: int), c ∈ C ->
solution M S <-> exists r, r ∈ R /\ r ∈ S /\ A[r][c] = true /\
                 solution (reduced_matrix M r) (S - { r }).
```

La preuve des liens dansants est simplifiée grâce à ce fichier : il suffit essentiellement de prouver que les opérations sur les pointeurs implémentent bien la suppression des lignes et des colonnes pour obtenir la validité du programme.

5.2 Preuves de `hide`, `show`, et `without`

Les deux premières fonctions sont prouvées dans `Permutation.v`. Dès qu'on est sur les permutations, on veut éviter de décrire ce qui se passe en-dehors de `D` (les éléments masqués dont les pointeurs ont été conservés) car c'est compliqué et dépendant de l'ordre de masquage. La spécification de `hide` fournit en post-condition les hypothèses nécessaires pour que le `show` se passe bien, mais pas plus. L'essentiel de la preuve consiste en fait à montrer que les sorties de ces deux fonctions sont bien des permutations.

Ce mécanisme de masquage et démasquage a été donné en exercice lors de la compétition VerifyThis 2015 ; Jean-Christophe Filliâtre et Guillaume Melquiond ont soumis une preuve concise en Why3 [11]. Ils utilisent deux prédicats `valid_in` et `valid_out`, le premier décrivant les éléments non masqués et le second les éléments prêts à être démasqués. Eux aussi éludent ce qui se passe ailleurs, et la suite se passe bien.

La preuve de `without`, en revanche, pose un problème de logique de séparation plus sérieux. Considérons une utilisation classique de `without` pour cacher un C_i dans la liste des en-têtes :

```
let f cp_head = ... cp ... in
without cp.head i f
```

La sous-fonction veut utiliser `cp` (capturé dans la clôture), mais suivant le mécanisme présenté en 4.2, `without` ne lui transmettra que le membre `cp.head`. Éventuellement la fonction courante peut lui transférer les autres champs du record, mais une spécification propre pour `f` demandera le record *fermé*, pas en morceaux.

Ce problème m'a amené à *spécialiser* la fonction `without` pour ses deux usages : avec `cp.head` et avec `cp.cols[i]`. Elle prend alors `cp` fermé en paramètre et transmet `cp` fermé à la sous-fonction. La preuve de la première forme se trouve dans `DancingLinks.v` et consiste essentiellement à enrober le lemme `revert_update` de `Permutation.v` dans le formalisme de la logique de séparation.

5.3 Difficultés lors des preuves de couverture

Les preuves que je n'ai pas réussi à finir sont celles des fonctions `cover_column` et `cover_element`. La seconde, en particulier, est subtile parce qu'elle itère à la fois sur les lignes en collision ■ et les colonnes à supprimer ■ ; la récurrence se fait sur deux paramètres.

Les preuves de `select_row` et `cover_element` nécessitent d'itérer le long de lignes ou de colonnes de la matrice. Typiquement la preuve se fait par induction, avec deux possibilités :

— Une récurrence sur le nombre d'éléments restant à masquer (dans ce cas il faut définir et maintenir le compte en fonction de la matrice) ;
— Une récurrence bien fondée sur la position dans la matrice.

TLC possède un formalisme pour la récurrence bien fondée, mais je n'ai pas eu le temps de me pencher dessus. La preuve de `select_row` est essentiellement complète, à part ce passage à la récurrence.

Pour formaliser l'itération, je n'ai pas pu éviter de faire apparaître l'état détaillé du programme dans les spécifications. Par exemple, les différents appels récursifs de `select_row` élaguent progressivement les ensembles C et R ; la spécification décrit précisément tous les états intermédiaires des deux ensembles jusqu'à atteindre les C' et R' de la matrice réduite.

5.4 Formalisation des cycles pour les invariants de boucles

Une dernière partie importante pour les preuves des itérations est de formaliser la notion de permutation cyclique, ce qui est fait dans `DancingLinks.v`. Rétrospectivement, il apparaît que les *cycles* sont bien plus utiles aux liens dansants que les *permutations* ; j'estime que reprendre la preuve pour ne jamais mentionner les secondes serait l'une des meilleurs améliorations à ma portée.

Après avoir tenté différentes notions très « mathématiques » et difficiles à manipuler, j'ai finalement défini un cycle par la liste de ses éléments. Essentiellement, cela revient à :

$$f \text{ est un cycle sur la liste } x_0 \dots x_{n-1} \equiv \forall i \in \{0..n-1\}, f(x_i) = x_{(i+1) \bmod n}$$

J'en déduis tous les lemmes utiles sur les cycles ; l'objectif est de prouver que durant une itération de f, on parcourt des éléments tous différents jusqu'à retrouver l'élément de départ.

Tous ces lemmes font intevernir des entiers de différents types (des `nat` pour les récurrences, et des `int` $\equiv$ `Z` pour les indices de liste, entre autres), ce qui rend les propriétés arithmétiques très fastidieuses à prouver. C'est un exemple typique de problème casse-tête mais absolument pas fondamental : Coq a beaucoup de mal à jongler avec ses types d'entiers.

Une fois la théorie des cycles bien posée, la preuve des itérations peut se faire sans trop d'encombres ; le souci principal devient alors de décrire les états intermédiaires de façon légère. Les problèmes qui restent sont certainement de la technique Coq (récurrences bien fondées...) plus que des soucis cruciaux de logique de séparation ou de modélisation.

6 Conclusion

Cette preuve de l'algorithme DLX était avant tout un exercice de vérification formelle. C'était mon premier projet conséquent dans Coq, et j'ai affiné progressivement ma compréhension de Curry-Howard, du calcul des constructions et du logiciel Coq à proprement parler.

La matrice creuse de Knuth est un objet complexe, mais l'utilisation de permutations permet de simplifier grandement sa représentation. Grâce à ce modèle, j'ai réussi à produire :

— Une nouvelle implémentation OCaml de l'algorithme DLX ;
— Une preuve formelle que l'algorithme Bactracking résoud bien Exact Cover ;
— Une preuve partielle de mon implémentation en logique de séparation.

La logique de séparation révèle de nombreuses subtilités de possession d'objets, qui sont invisibles dans le code OCaml parce que le langage s'en affranchit. Ça incite le programmeur à séparer plus nettement les parties du code qui manipulent des données différentes, et ainsi mieux définir le rôle de chaque fonction.

La preuve aurait certainement été plus facile si j'avais réalisé plus tôt que la notion de cycle surclasse celle de permutation ; ce formalisme trop général ajoute à la preuve des détails abstraits qui alourdissent le dialogue avec Coq, pourtant déjà chargé par la logique de séparation.

7 Remerciements

Pendant cette première expérience de recherche, j'ai côtoyé l'équipe Gallium, mais pas seulement. J'aimerais remercier Arthur Charguéraud, qui m'a longuement expliqué les arcanes de TLC et CFML lors d'un déplacement depuis Strasbourg ; ainsi que Jean-Christophe Filliâtre, qui a démontré un grand intérêt pour ce petit stage, et pris du temps pour en discuter avec moi.

Merci aussi à Armaël Guénaud, souvent disponible pour apporter son expertise à mes questions tordues de Coq, et au reste de l'équipe Gallium, qui m'a accueilli et entretenu de ses histoires pendant 6 semaines.

Un grand merci enfin à François Pottier pour m'avoir proposé sans hésitation ce stage, et fait découvrir le monde la vérification formelle.

Table des figures

Références

[1] R. M. Karp, "Reducibility among combinatorial problems," in *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, pp. 85–103, 1972.

[2] D. E. Knuth, "Dancing links," *Millenial Perspectives in Computer Science*, pp. 187–214, 2000.

[3] H. Hitotumatu and K. Noshita, "A technique for implementing backtrack algorithms and its application," *Inf. Process. Lett.*, vol. 8, no. 4, pp. 174–175, 1979.

[4] A. Charguéraud, "Higher-order representation predicates in separation logic," in *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016*, pp. 3–14, 2016.

[5] A. Charguéraud, "CFML : Characteristic Formulae for ML." http://www.charggueraud.org/softs/cfml/. [Accessed 2018-08-16].

[6] A. Charguéraud, "TLC : a non-constructive library for Coq." http://www.charggueraud.org/softs/tlc/. [Accessed 2018-08-16].

[7] J. Filliâtre and A. Paskevich, "Why3 - where programs meet provers," in *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013*, pp. 125–128, 2013.

[8] B. Zhan, "Efficient verification of imperative programs using auto2," in *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018*, pp. 23–40, 2018.

[9] "ANR Project VOCaL – The Verified OCaml Library." https://vocal.lri.fr/. [Accessed 2018-08-16].

[10] R. E. Sibaïe and J.-C. Filliâtre, "Combine – an ocaml library for combinatorics." https://www.lri.fr/~filliatr/combine/. [Accessed 2018-08-16].

[11] J.-C. Filliâtre and G. Melquiond, "Verifythis 2015 competition, challenge 3 : Dancing links." http://toccata.lri.fr/gallery/verifythis_2015_dancing_links.en.html. [Accessed 2018-08-16].

A Contexte du stage

L'équipe Gallium à l'INRIA Paris

Gallium est une des équipes-projets de l'INRIA Paris, créée en 2006 et bientôt au terme de ses douze ans de durée de vie. On y trouve sept chercheurs permanents, un post-doctorant et quatre thésards, en plus des stagiaires qui vont et vient ; six pendant le mois de Juin.

Son travail porte sur la conception et l'implémentation de langages de programmation, et sur la vérification formelle ; les deux sont souvent liés. Le paradigme fonctionnel et les systèmes de types, entre autres exemples, améliorent l'expressivité et la sécurité des langages, et sont directement à l'interface avec les méthodes formelles qui permettent de vérifier automatiquement le code.

Les réalisations les plus célèbres de Gallium sont certainement le langage OCaml et le compilateur C verifié en Coq, CompCert. Dans ses publications se trouvent aussi des preuves de circuits électroniques ou des formalisations de $\mathcal{O}$ pour la complexité asymptotique. De plus amples détails peuvent être trouvés sur leur site web [4] et sur leur blog [5].

La photo ci-dessous (aimablement fournie par le projet Google Maps) est celle des nouveaux locaux à proximité de la Gare de Lyon, qui sont quand même plus accessibles que ceux de Rocquencourt.

Les interactions dans le métier de la recherche

Comme l'algorithme des liens dansants ne fait pas spécifiquement partie des projets de l'équipe, le seul à connaître les détails de mon sujet était mon encadrant François Pottier. La plupart des point cruciaux de ce rapport ont été découverts avec lui, en face de mon tableau blanc ; évidemment cela limite mes échanges avec le reste des membres.

Heureusement, quand on débute en Coq et qu'on n'a jamais utilisé la logique de séparation en pratique, on a assez de questions pour occuper un régiment de chercheurs ; ça permet de communiquer un peu. En plus des pauses cafés et leur part d'explications sur la bisimulation ou les sujets des autres stagiaires.

Difficile de conclure en à peine 6 semaines, mais j'ai l'impression qu'en définitive, les travaux individuels des chercheurs sont souvent issus des idées de toute l'équipe. (Ce qui est certainement une modèle idéal de la réalité, j'en conviens...)

4. http://gallium.inria.fr/
5. http://gallium.inria.fr/blog/

B TLC et CFML

Cette annexe regroupe différents idiomes de TLC et CFML qui peuplent mes sources Coq.

Listes et tactiques d'instantation

Les listes d'instantiation permettent d'instantier un prédicat en fournissant uniquement certains de ses arguments. L'utilisateur ne spécifie pas lesquels ; seul leur ordre compte. S'il en existe un arrangement typable, TLC l'utilisera. La syntaxe est `(>> Predicate arg1 arg2 ...)`.

Essentiellement, TLC scanne de gauche à droite les positions possibles (typables) pour `arg1` et choisit la première (ce choix est définitif). Elle scanne ensuite les positions restantes à droite pour placer `arg2`, et ainsi de suite. Les arguments non spécifiés sont inférés. Par exemple :

```
Parameter P : nat -> nat -> list nat -> unit -> nat.
applys (>> P 1 nil 3).
(* Interprété comme (P 1 _ nil _ 3). *)
```

Les tactiques d'instantiation qui peuvent utiliser les listes sont les suivantes :

`lets <name>: <instantiation>`
Ajoute une hypothèse du type du terme (proche de `pose proof`) ; si si tous les arguments ne peuvent pas être inférés, la tactique échoue.

`applys <instantiation>`
Comme `apply`, mais supporte les listes d'instantiation. Tous les arguments non spécifiés doivent pouvoir être inférés.

`forwards <name>: <instantiation>`
Annonce l'intention d'instantier un lemme ; tous les arguments qui ne peuvent pas être inférés sont placés dans des sous-buts. Une fois ces sous-buts prouvés, le lemme instantié est ajouté aux hypothèses.

Tactiques de réécriture

Diverses tactiques de la forme `rew_*` : `rew_list`, `rew_set`, `rew_index`... qui contiennent un grand nombre de lemmes de réécriture pour un domaine spécifique.

Suffixes `~` et `*`

Toutes les tactiques de TLC supportent les suffixes `~` et `*`. Le premier invoque automatiquement `auto` après la tactique ; le second invoque `jauto`, qui détruit les conjonctions, instancie les existentielles et termine sur `eauto`.

Tactiques diverses

`introv <H1> <H2> ...`	Comme `intros`, mais nomme automatiquement les variables.
`sets <name>: <term>`	Pose une définition.
`false_invert`	Tente d'extraire une contradiction des hypothèses.
`case_if`	Produit deux sous-cas à partir d'un but en `if-then-else`.
`tests: <prop>`	Sépare le but en deux sous-buts où `prop` est respectivement vraie ou fausse.

Tactiques CFML

La plupart des tactiques CFML agissent sur le triplet de Hoare que l'on cherche à prouver. Ce triplet est décrit par deux conditions et l'arbre de syntaxe du code ; la racine de l'arbre détermine souvent la tactique à utiliser. `xapp` est quasiment universelle.

— Les tactiques en `x*` agissent sur les triplets de Hoare.
— Les tactiques en `h*` agissent sur les conditions (tas), notamment `hsimpl` pour les simplifications.

Le suffixe `s` signifie *substitute* et remplace automatiquement la variable résultant de la tactique quand c'est pertinent. Par exemple, `xlets` introduit un identifieur et le substitue dans le reste du code.