Preuve de programmes probabilistes en Coq

Pierre Clairambault

18 avril 2005

Résumé

Le thème de ce stage était le développement d'outils nécessaires à la preuve de propriétés de programmes probabilistes dans l'assistant de preuves Coq [5]. Son objectif concret a été finalement d'élaborer un exemple d'utilisation de la bibliothèque écrite par Christine Paulin-Mohring. Cette bibliothèque s'appuie sur une interprétation des programmes probabilistes comme transformateurs de mesure, plutot que comme transformateurs (non déterministes) d'état.
L'algorithme concerné par le stage est un algorithme de recherche de coupe minimale dans un graphe, qui s'appuie à chaque étape sur le choix non déterministe d'une arete dans un graphe et fait subir à ce graphe des transformations en conséquence. En s'appuyant sur une réprésentation des graphes adaptée, on s'intéresse à prouver en Coq qu'on obtient bien une coupe, puis à minorer la probabilité que cette coupe soit effectivement minimale.

1 Introduction

Coq [5] est un assistant de preuves, c'est à dire qu'il permet la spécification formelle de programmes, leur implémentation et leur certification par des preuves formelles. Cependant il ne permet a priori de parler que de programmes déterministes, ce qui est assez restrictif puisqu'on a des exemples où les algorithmes probabilistes se révèlent plus performants que leurs équivalents déterministes, ou du moins donnent rapidement des approximations efficaces de problèmes complexes.
Le premier problème est de trouver une représentation pour les programmes probabilistes. D. Kozen [2] et C.Morgan [3] introduisent l'idée d'interpréter les programmes probabilistes comme des transformateurs de mesure plutot que comme des transformateurs d'état. Plus récemment, Joe Hurd [1] explique comment modéliser et montrer des propriétés de programmes probabilistes dans l'assistant de preuves HOL, en utilisant une transformation monadique des programmes. C.Paulin a construit une bibliothèque s'inspirant de ces travaux, s'appuyant sur une autre transformation monadique des programmes, en les interprétant comme des transformateurs de mesure.
L'objet du stage était alors de construire un premier exemple conséquent de preuve s'appuyant sur cette bibliothèque. Motwani [4] propose un algorithme probabiliste de recherche de coupe minimale dans un graphe, intéressant pour jouer ce rôle par la non-trivialité de sa correction.
Après avoir décrit la librairie de C.Paulin, nous verrons la modélisation des graphes adoptée, pour enfin passer à l'étude de la partie probabiliste de l'algorithme.

2 La librairie de preuves de programmes Probabilistes

2.1 Représentation des mesures

Soit p la sortie d'un programme aléatoire. On veut exprimer $Pr(p \in Q)$ la probabilité que p soit dans Q, c'est à dire vérifie un certain prédicat. Parler de mesures permet d'exprimer cela sans avoir à formaliser la notion de probabilité, puisque la notion de mesure est axiomatisée comme présenté ci-dessous, sans nécessité de recours à l'intuition sur ce qu'elle représente. C'est donc un cadre tout à fait adapté aux preuves formelles.

Mesures

D'une distribution de probabilités Pr sur un ensemble A avec une σ-algèbre E, on peut construire une mesure μ sur les fonctions positives de A vers $\mathbb{R}^+$.
Si f est définie comme une combinaison linéaire positive $\Sigma_i a_i \mathbb{I}_{X_i}$ de fonctions caractéristiques d'ensembles disjoints $X_i \in E$ alors $\mu(f) = \Sigma_i a_i Pr(X_i)$, ce qui s'étend facilement aux fonctions générales.
Dans le cas où l'ensemble A est fini, la mesure d'une fonction f correspond à la notion intuitive d'espérance de cette fonction (selon la distribution de probabilités donnée). Dans ce cas :

$$\mu(f) = \Sigma_{x \in A} f(x) \ \times \ Pr(\{x\})$$

Implémentation des mesures

Les notions de mesure et de distribution de probabilités sont équivalentes au sens ou définir une mesure définit implicitement une distribution de probabilités (Soit X un ensemble, μ une mesure, alors $Pr(X) = \mu(\mathbb{I}_X)$), et vice-versa (par intégration, comme vu au dessus).
On se servira donc dans la modélisation, non pas de probabilités mais uniquement de mesures. On se restreindra aux fonctions bornées (à valeurs dans $[0, 1]$), une mesure sera donc pour nous une fonction de type $(A \to [0, 1]) \to [0, 1]$ satisfiant ces propriétés :

$$\mu(f_1 + f_2) = \mu(f_1) + \mu(f_2)$$
$$\mu(k \times f) = k \times \mu(f)$$
$$\mu(1 - f) \leq 1 - \mu(f)$$

Si $f \leq g$ (c'est à dire $\forall x \ f(x) \leq g(x)$) alors $\mu(f) \leq \mu(g)$

On pourra ainsi traiter dans Coq les mesures comme des objets fonctionnels usuels et axiomatisés, sans avoir à formaliser les notions de probabilité et de σ-algèbre.

2.2 Interprétation d'expressions et transformation monadique

Placons-nous dans un langage fonctionnel à la ML minimaliste généré par la grammaire suivante :
- Constantes : c
- Conditionnelle : **if** b **then** e_1 **else** e_2
- Liaison locale : **let** $x = a$ **in** e
- Abstraction : **fun** $(x : \tau) \Rightarrow$e
- Application : $(e_1 \ e_2)$

On affecte ce langage d'un système de types simple (sans polymorphisme), et on y ajoute des primitives aléatoires comme **random**, qui prend un entier n et qui en renvoie un autre choisi aléatoirement entre 0 et n. On pourrait aussi avoir **flip**, qui renvoie **true** ou **false** avec probabilité $\frac{1}{2}$.
Soit e une expression de type τ. e représente en fait un ensemble de valeurs de type τ, puisque l'inerprétation de e peut conduire à ces valeurs de façon non déterministe.
On voudrait connaitre la distribution de ces valeurs : il faut pour cela interpréter $e : \tau$ comme une mesure sur τ, c'est à dire une fonction de type $(\tau \to [0, 1]) \to [0, 1]$ (noté $[\tau]$)

L'idée est donc de faire correspondre à tout terme $e : \tau$ de notre langage minimaliste un terme $\tilde{e} : (\tau \to [0, 1]) \to [0, 1]$ de COQ, représentant intuitivement la distribution des valeurs possibles de l'interprétation de e.

On peut aisément obtenir la mesure $[p]$ en faisant subir à p une transformation monadique, où chaque

expression de type τ est interprétée comme une expression purement fonctionnelle de type $[\tau]$. On a besoin pour cela de deux opérateurs :

$$\textbf{unit} : \tau \to [\tau] \quad \textbf{bind} : [\tau] \to (\tau \to [\sigma]) \to [\sigma]$$

On a également besoin, pour chaque constructeur de base, d'une interprétation. (Pour **random** de type $int \to int$, on a besoin d'une interprétation de type $int \to [int]$)
On a alors l'interprétation des programmes suivantes :

Expression $p : \tau$	Valeur fonctionnelle $[p] : [\tau]$
let $x = a$ **in** b	(bind $[a]$ **fun**$(x : \tau) \Rightarrow [b]$)
fun$(x : \tau) \to t$	**fun**$(x : \tau) \Rightarrow [t]$
$(t\ u)$	(bind $[u]$ $[t]$)
if b **then** e_1 **else** e_{e2}	(bind $[b]$ **fun** $(x : bool) \to$ **if** x **then** $[e_1]$ **else** $[e_2]$)

Définition de unit et bind

La définition des opérateurs monadiques est la même que dans le cas des continuations :

$$\begin{array}{lcl} \textbf{unit} & : & \tau \to [\tau] \\ & = & \textbf{fun}\ (x : \tau) \Rightarrow \textbf{fun}\ (f : \tau \to [0,1]) \Rightarrow (f\ x) \\ \textbf{bind} & : & [\tau] \to (\tau \to [\sigma]) \to [\sigma] \\ & = & \textbf{fun}\ (\mu_a : [\tau]) \Rightarrow \textbf{fun}\ (\mu_b : \tau \to [\sigma]) \Rightarrow \textbf{fun}\ (f : \sigma \to [0,1]) \\ & & \Rightarrow (\mu_a\ \textbf{fun}\ (x : \tau) \to (\mu_b\ x\ f)) \end{array}$$

On peut se pencher rapidement sur le sens de ces définitions. $(unit\ x)$ représente la mesure de Dirac au point x. Pour une fonction $f : \tau \to [0,1]$ donnée, elle renvoie donc la valeur de f en x.
$(bind\ \mu_a\ \mu_b\ f)$, quant à elle, procède à une sorte d'*intégration*. Intuitivement, elle calcule l'espérance de f sur une distribution égale à l'espérance des $\mu_b\ x$ où x est "choisi" par la distribution représentée par μ_a ...

Définition de random

random n doit rendre une distribution uniforme sur {0,...,n}, donc :

$$random(n) = \textbf{fun}\ (f : int \to [0,1]) \Rightarrow \sum_{i=0}^{n} \frac{f\ i}{n+1}$$

$random\ n\ f$ calcule simplement la moyenne de f sur {0,...,n}.

2.3 Utilisation dans Coq

On peut, à l'aide des combinateurs précédemment définis, transformer toute expression $e : \tau$ en $\mu_e : [\tau]$. Avec ceci, il est possible de formaliser la probabilité pour e de vérifier une propriété P.
Notons $\mathbb{I}_P$ la fonction caractéristique de la propiété P, alors :

$$Pr(P\ e) = \mu_e\ \mathbb{I}_P$$

L'implémentation en Coq repose sur une axiomatisation adaptée de l'intervalle $[0,1]$. On formalise également la notion de mesure, on écrit les opérateurs monadiques. Dès lors on peut démontrer des propriétés sur la sortie de programmes aléatoires, grâce à la remarque ci-dessus.
L'implémentation effective ne sera pas décrite ici puisqu'elle n'est pas indispensable à la compréhension de ce qui suit et qu'elle sort de l'objectif de ce rapport. Les lemmes utilisés seront cependant préalablement présentés.

3 Graphes, objectifs et représentation

Nous allons maintenant décrire le développement effectué en se servant de cette bibliothèque. L'algorithme de recherche de coupe minimale dans un graphe présenté par Motwani [4] a été choisi car il s'appuie sur une structure de haut niveau nécessitant une modélisation, et car la propriété qu'on veut montrer sur sa sortie n'est pas triviale. Ce choix permet donc non seulement de construire l'exemple, mais aussi de mettre à l'épreuve la bibliothèque.

3.1 Présentation de l'algorithme

Définition. *Soit $G = (S, A)$ un graphe, supposé connexe. On appelle* coupe *de ce graphe tout ensemble d'arêtes qui déconnecte le graphe lorsqu'il est supprimé. C'est à dire :*
C est une coupe de G si et seulement si il existe V un ensemble de sommets de G non vide, non égal à S tel que $\forall x \in V, \forall y \in (S - V)$, tout chemin de x à y passe par C.
Une coupe minimale *est une coupe de taille (nombre d'arêtes) minimale sur G.*

Bien entendu la définition n'est pas restrictive aux graphes connexes, sur un graphe non connexe tout ensemble d'arêtes est une coupe. Mais pour simplifier la représentation en Coq, on restreint l'étude aux graphes connexes sans boucle (c'est aussi les cas où il est intéressant de mener une telle étude...)

Algorithme. *Soit $G = (S, A)$ un graphe connexe.*
- *Si $|S| = 2$ alors renvoyer A, c'est une coupe.*
- *Sinon, prendre $a \in A$ au hasard,* contracter *a, c'est à dire fusionner les extremités de a et supprimer les arêtes superflues qui de par cette fusion deviendraient des boucles, et poursuivre récursivement.*

En CAML, on aurait quelque chose comme :

```
let rec min_cut g = match (nb_sommets g) with
|2 -> aretes g
|n -> let a = random_arete (aretes g) in
min_cut (contraction g a)
```

Dans ce programme, on considère que `random_arete` renvoie un élément aléatoire de l'ensemble d'arêtes donné en argument, et que `contraction g a` rend le graphe où l'arête a a été contractée. On remarque également que le programme COQ ne pourra pas être pris exactement sur ce modèle, puisqu'en COQ la récurrence doit être structurelle (La différence ne sera malgré tout pas fondamentale).

On a la propriété qu'une coupe du graphe contracté était forcément une coupe du graphe initial. Dès lors, il est clair par récurrence immédiate que cet algorithme renvoie une coupe de G. Cependant cette coupe a toutes les chances de ne pas être minimale. On peut cependant minorer la probabilité qu'il a de rendre une bonne réponse (c'est un algorithme à la Monte Carlo).
Soit $G = (S, A)$ un graphe connexe. Soit C une coupe minimale de ce graphe (de taille k), qui pourrait d'ailleurs bien être la seule. Si $|S| = n$, alors

$$|A| \geq \frac{nk}{2}$$

En effet, tous les sommets sont au moins de degré k, sinon on aurait une coupe de plus petite taille que C.
Donc, la probabilité de prendre et de contracter une arête hors de C est

$$\frac{|A| - k}{|A|} \geq \frac{n-2}{n}$$

On en déduit que la probabilité p d'obtenir C comme coupe finale vérifie :

$$p \geq \prod_{k=3}^{n} \frac{k-2}{k} = \frac{2}{n(n-1)} \geq \frac{2}{n^2}$$

Il suffit dès lors d'itérer l'algorithme pour avoir une probabilité de succès arbitrairement proche de 1.

Remarque : Il faut cependant itérer de nombreuses fois pour obtenir un résultat convenable. Il existe des algorithmes déterministes plus efficaces qui s'appuient sur des considérations de flots dans les graphes. Celui-ci reste cependant intéressant du fait qu'il parvient à de bons résultats alors qu'il est techniquement assez simple. Il existe également des variantes aléatoires plus efficaces, procédant aux itérations plus finement, en tenant compte du fait que les premières arêtes ont plus de chances d'être hors de C.

3.2 Représentation des graphes

L'idée originale est simplement de représenter un graphe de taille n par l'intervalle discret $\{1, \ldots, n\}$ et par l'ensemble de ses arêtes, une liste de couples d'entiers. (Les arêtes ici seront considérées comme non orientées, même si l'implémentation n'a pas cette symétrie).
Cependant, il est crucial de pouvoir facilement fusionner des sommets puisque c'est l'opération principale lors de l'exécution de l'algorithme. On veut également, à la fin de l'exécution, être capable de renvoyer l'ensemble d'arêtes qui est une coupe du graphe original. La représentation des arêtes ne doit donc pas etre modifiée par la contraction.
Les sommets forment donc en permanence une partition de {1,...,n}, et la représentation des arêtes n'est pas modifiée.

Les sous-ensembles de {1,...,n}, c'est-à-dire les sommets, seront représentés par des listes.
On a donc finalement la représentation suivante :

```
Definition sommet := list Z.
Definition arete : Set := Z * Z.
Definition aretes := list arete.
Definition graph := aretes*(list sommet).
```

L'opération de contraction devient alors simple : il suffit de fusionner les listes correspondant aux deux sommets, et de supprimer les boucles, ce qui correspond à un parcours de liste.
Et pour accéder plus intuitivement au contenu d'un élément de type `graph` :

```
Definition ar := fst.
Definition som := snd.
```

Définition de quelques prédicats :

- `Definition sommet_sur_arete (s :sommet) (a :arete) : Prop`
  Exprime si le sommets s est atteint par l'arête a.
- `Definition arete_dans_ensemble (x :arete) (la :aretes) : Prop`
  Les arêtes ne sont pas orientées, mais sont représentées par des objets orientés (les couples). On a donc besoin d'un autre prédicat d'appartenance que celui des listes.
- `Definition relies_som (g :graph)(x y :sommet)(x'y' :Z) : Prop`
  Exprime si deux sommets x et y sont reliés dans le graphe g par l'arete (x',y').
- `Inductive chemin (g :graph)(x :sommet) : sommet → aretes → Prop`
  Définition inductive : `chemin g x y l` si l est un ensemble d'arêtes menant dans g de x à y.

- `Definition cut (g :graph)(la :aretes) : Prop`
  Cette définition se base sur la caractérisation des coupes donnée plus haut lors de l'étude informelle.
- `Definition is_mincut (g :graph) (la :aretes) : Prop`
  Une coupe est minimale si toute les autres coupes sont plus grandes...

3.3 Limites de la représentation

Au cours du développement, la représentation choisie pour les graphes s'est révélée insuffisante ou mal adaptée. Cependant le travail étant bien avancé, la poursuite des preuves a été préférée à la conception d'une représentation plus appropriée (qui par manque de temps aurait empêché de finir les preuves). Des axiomes ont donc été indispensables à la preuve de propriétés sur les graphes. Il y en a de plusieurs catégories : certains doivent être vérifiés par les graphes initialement, et sont conservés par les opérations sur ces graphes (la contraction). Pour bien faire, ceux-ci devraient être inclus dans la définition du type `graph` à l'aide d'un `Record`. D'autres sont la conséquence d'avoir raisonné avec le prédicat `List.In`, trop proche de la représentation effective, qui rend notament difficile de raisonner sur des graphes non orientés, ou avec l'égalite de base là où d'autres relations d'équivalence auraient mieux convenu.
En voici la liste exhaustive.

Invariants des graphes : ces axiomes sont les propriétés invariantes par la contraction.

- `Hypothesis no_duplication : ∀(g :graph)(x y :sommet)(z :Z),`
  $x \in$`(som g)` $\rightarrow y \in$`(som g)` $\rightarrow z \in x \rightarrow z \in y \rightarrow x = y$`.`
  Les sommets forment une partition de $\{1, \ldots, n\}$, donc si on a $x \in y$ et $x \in z$ c'est que y et z sont identiques.
- `Hypothesis sommets_non_vides : ∀(g :graph)(x :sommet),`
  $x \in$`(som g)` $\rightarrow x \neq$`nil.`
  Les sommets formant une partition de $\{1, \ldots, n\}$, parler de sommet vide n'a aucun sens.
- `Hypothesis connexite : ∀(g :graph)(x y :sommet),`
  $\exists(l : aretes)$, `chemin g x y l.`
  Comme on considère toujours des graphes d'au moins 2 sommets, on déduit de cet axiome le fait que l'ensemble des arêtes du graphes n'est jamais vide (qui sert effectivement dans les preuves)
- `Hypothesis no_repetition :∀(g :graph)(m n :nat),`
  $m <$`nb_sommets g` $\rightarrow n <$`nb_sommets g` $\rightarrow$
  $m \neq n \rightarrow$`nth m (som g) nil ≠ nth n (som g) nil.`
  Cette hypothèse est nécessaire puisque la structure de liste permet les répétitions. Elle est due au fait qu'ici les listes sont employées pour représenter des ensembles.

Incomplétude de la représentation : ces axiomes sont dus aux inadaptations de la représentation

- `Hypothesis eq_sommet_intro :∀ (x y :sommet),` $(\forall z, z \in x \leftrightarrow z \in y) \rightarrow x = y$.
  L'égalité sur les listes ne convient pas ; ce qui nous intéresse est une égalité sur les ensembles.
- `Hypothesis eq_graph :∀(`g_1 g_2 `: graph),`
  `(`$\forall x, x \in$`(ar` g_1`)` $\Leftrightarrow x \in$`(ar` g_2`))`
  $\rightarrow$ $(\forall x, x \in$`(som` g_1`)` $\Leftrightarrow x \in$`(som` g_2`))` $\rightarrow g_1 = g_2$`.`
  De même, l'égalité syntaxique sur les graphes ne convient pas.

3.4 Opérations sur les graphes

L'opération cruciale sur les graphes est bien sûr, étant donnée une arête, la contraction de celle-ci. La méthode adoptée pour construire cette opération est d'écrire sa spécification, puis de faire la preuve que son type est habité.
La spécification est la suivante :
`Definition contraction : ∀(g : graph)(a : arete), arete_dans_ensemble a (ar g) →`

{ g' | Post(g,a,g')}
Ou Post(g,a,g') est une (grosse) expression exprimant que :

- les sommets de g' sont ceux de g qui ne touchaient pas a, ou le résultat de la fusion des sommets qui touchaient a.
- les arêtes sont celles de g qui ne sont pas sur les mêmes sommets que a
- le graphe contracté a toujours *un sommet de moins* que le graphe original (puisqu'on fusionne deux sommets)

Le schéma de la preuve, et donc celui du programme extrait, est le suivant : On filtre les sommets selon qu'ils sont ou ne sont pas sur l'arête a. On fusionne ceux qui touchent a, et on concatène le sommet obtenu avec la liste de ceux qui ne touchaient pas a, ce qui forme la liste des sommets de g' le graphe contracté. Pour obtenir les arêtes, on filtre les arêtes de g, on ne garde que celles qui ne sont pas sur les mêmes sommets que a.
On remarque que les hypothèses sont bien conservées ; il n'y pas de boucle possible. On obtient un multigraphe, tout en gardant la nature initiale des arêtes : les arêtes multiples entre deux sommets ne sont pas syntaxiquement identiques.

3.5 Propriétés des graphes

Lors de la preuve de l'algorithme aléatoire, on minore en fait la probabilité d'obtenir non pas une coupe minimale quelconque, mais une en particulier initialement fixée. Le succès de l'algorithme repose donc sur des propriétés de conservation des coupes par la contraction.
La première propriété est la suivante : Soit $G = (S, A)$ un graphe, $a \in A$, et G' la contraction de G par a. Soit C une coupe de G', alors C est nécessairement une coupe de G.

```
Theorem cut_cons : ∀ (g :graph) (la :aretes) (a :arete), arete_dans_ensemble a (ar g) →
cut (contraction g a) la → cut g la.
```

(Cet énoncé n'est pas l'énoncé exact. En effet, formellement une autre formulation est nécessaire puisque (contraction g a) renvoie, en plus du graphe contracté, la preuve que celui-ci respecte les spécifications. Par souci de présentation, on considère ici qu'elle ne renvoie que le graphe et que la preuve de correction est accessible autrement. C'est d'autant plus justifiable qu'on pourrait tout de même avoir cet énoncé exact à l'aide d'une coercion.)
Propriété suivante : Soit $G = (S, A)$ un graphe, la une coupe de ce graphe, soit $a \in S$ n'appartenant pas à la. Alors la est une coupe de la contraction de G par a.

```
Theorem cut_cons_back : ∀ (g :graph)(la :aretes)(a :arete), arete_dans_ensemble a (ar g)
→ not arete_dans_ensemble a la → cut g a → cut (contraction g a) la.
```

Intuitivement, cette propriété se justifie facilement : la coupe la sépare le graphe en deux composantes connexes. La seule façon de les réunir et donc d'invalider la est de contracter une arête de la.
Et bien sûr, puisque la démonstration sera une récurrence, on a besoin du cas de base : si le graphe n'a que deux sommets, l'ensemble de ses arêtes est bien sûr une coupe.

```
Lemma cut_base : ∀(g :graph), nb_sommets g = 2 → cut g (fst g).
```

On a maintenant suffisament d'éléments pour montrer que l'algorithme étudié donne à coup sûr une coupe. Quelques autres propriétés sont nécessaires pour estimer la probabilité que la coupe soit minimale. On doit montrer l'inégalité présentée plus haut : $|A| \geq \frac{nk}{2}$, où n est le nombre de sommets du graphe et k est la taille d'une coupe minimale.

```
Lemma inegalite : ∀ (g :graph)(k :nat), nb_sommets g ≥ 2 →
(∀la, cut g la → k ≤length la) → k(nb_sommets g)≤ 2(nb_aretes g).
```

Pour prouver cette inégalité on va utiliser une méthode de double décompte.

On pose :

$$\texttt{compte(x,a)} = \begin{cases} 1 & \text{si x est sur a} \\ 0 & \text{sinon} \end{cases}$$

Alors, pour $G = (S, A)$ un graphe, $n = |S|$ on a :

$$\sum_{x \in S} \sum_{a \in A} \texttt{(compte x a)} \geq nk$$

(puisque nécessairement, tous les sommets sont de degré au moins k)
De même, on a :

$$\sum_{a \in A} \sum_{x \in S} \texttt{(compte x a)} = 2|A|$$

(Une arête touche toujours exactement 2 sommets, et on fait le compte pour toutes les arêtes)
De plus, on montre facilement que ces sommes commutent, ce qui nous donne une preuve de notre inégalité.

$$2|A| \geq nk$$

4 Graphes : aspects probabilistes

4.1 Quelques préliminaires

L'objet de cette section est de décrire certains points d'implémentation indispensables pour ce qui va suivre.

L'ensemble U $\leftrightarrow$ [0,1] : On introduit tout d'abord l'ensemble U, correspondant à l'axiomatisation de $[0, 1]$, dont les constantes (0 et 1) se notent U0 et U1.

Les distributions : L'ensemble des distributions sur A est noté `distr A`. Il s'agit en fait d'un type `Record`, dont les champs contiennent les axiomes définissant les mesures, à l'exception du champ `mu` qui contient la mesure effective.
Donc, si `m : distr A`, alors `mu m : (A` $\rightarrow$ `[0,1]) ` $\rightarrow$ `[0,1]`.

Les combinateurs monadiques : Les combinateurs **unit** et **bind** sont appellés respectivement `Munit` et `Mlet` sur les distributions, noms qui se rapprochent de leur sémantique sur ces ensembles.

Les fonctions caractéristiques : On a besoin, pour exprimer la probabilité pour une distribution de vérifier un prédicat, de la fonction caractéristique de ce prédicat.
Soit `P` : $\alpha \rightarrow$ `Prop` un prédicat décidable.
Alors on a `(carac P)` : $\alpha \rightarrow$ `U` qui vérifie :

$$\texttt{(carac P) x} = \begin{cases} \texttt{U1} & \text{si (P x)} \\ \texttt{U0} & \text{sinon} \end{cases}$$

4.2 Transformation monadique de `min_cut`

Commençons par écrire le programme `min_cut` en Ocaml :

```
#let rec min_cut (g :graph) (n :int) = match n with
|0 -> ar g
```

```
|n -> let k = random (nb_aretes g - 1) in
(fun k -> min_cut (contraction g (nth (ar g) k)) (n-1)) ; ;
val min_cut : graph -> int -> aretes
```
(le n donné en argument doit être $k-2$, où k est le nombre de sommets du graphe. C'est nécessaire pour appliquer la transformation monadique vers COQ puisque la récurrence en COQ doit être structurelle)

La transformation en COQ est, suivant les règles énoncées en première partie :
```
Fixpoint min_cut (g :graph)(n :nat) {struct n} : distr aretes :=
match n with
|0 => Munit (ar g)
|S n => Mlet (Random ((nb_aretes g) - 1))
(fun k => min_cut (contraction g (nth_in g k)) n)
end.
```

Montrons que le programme renvoie bien une coupe du graphe avec probabilité 1. Dans notre formalisation, cela s'exprime par :
```
Lemma min_cut_is_cut : ∀ (n :nat)(g :graph), (nb_sommets g) ≥ 2 →
n = ((nb_sommets g) - 2) →
U1 <= mu (min_cut g n) (carac (cut g)).
```

La preuve se fait par induction. Pour le cas de base, il suffit d'appliquer `cut_base` ainsi que le lemme :
```
Lemma deterministe : forall (A :Set)(P :A -> Prop)(a :A),
(P a) -> U1 <= mu (Munit a) (carac P).
```
Qui permet de passer du vocabulaire usuel à celui des mesures.

Maintenant, on sait que $\forall k$, `min_cut (contraction g (nth_in g k)) n` est une coupe de g. Pour montrer le cas général, il faut "intégrer" par rapport à la distribution `(Random ((nb_aretes g) - 1))`, d'où l'utilité du lemme suivant :
Soient a une mesure sur A, p une fonction de A vers les mesures sur B, et P une propriété. Alors :
Si $\forall x, Pr(p\ x \in P) \geq 1$, alors $Pr((\texttt{let x=a in}\ p\ x) \in P) \geq 1$ si a termine presque sûrement.
Dans notre formalisation, cela s'exprime :
```
Lemma integration_cons : ∀ (A B :Set)(m1 : distr A)(m2 : A → distr B)(P :B → Prop)
, mu m1 (f_one A) ≥ U1 ->
(forall x :A, U1 ≤ mu (m2 x) (carac P)) -> U1 ≤ mu (Mlet m1 m2) (carac P).
```
(`f_one A` est la fonction égale à 1 sur A. La condition `mu m1 (f_one A) == U1` est vraie dès que le calcul termine, cette condition est donc une assurance que `m1` est définie partout, condition intuitivement nécessaire, et qui est assurément vraie pour `random` puisqu'il s'agit d'une somme finie).
Minorons la probabilité d'obtenir une coupe minimale
Soit `k : int -> U` défini par la définition récursive suivante :

$$\begin{cases} k(0) = 1 \\ k(n+1) = k(n)\frac{n+1}{n+3} \end{cases}$$

On veut maintenant prouver l'énoncé suivant :
```
Theorem final : forall (C :aretes)(n :nat)(g :graph)(hyp1 : nb_sommets g = n + 2),
(k n)*((carac (is_mincut g)) C) ≤ mu (min_cut g n) (carac (eq_aretes C)).
```

Comme vu auparavant, le raisonnement est le suivant : On prend une coupe minimale fixée, et on estime la probabilité qu'elle soit conservée au cours des contractions, c'est à dire la probabilité que les arêtes choisies pour la contraction soient toujours hors de cette coupe.
Il faut encore une fois raisonner par induction sur la définition de min_cut.

Pour le cas de base, on est amené à prouver `U1` $\leq$ `mu (Munit (ar g)) (carac (eq_aretes C))`, c'est à dire que la probabilité que l'ensemble des arêtes de g (dans le cas où g a seulement deux sommets) soit égal à C est de 1.
Grâce au lemme `deterministe`, on se ramène à `eq_aretes C (ar g)` qui est conséquence d'un lemme précédent.

Intuitivement, on voudrait ensuite dire que si on contracte une arête hors de C, le fait que C est une coupe minimale se conserve, or la probabilité de prendre (au hasard) une arête hors de C est $\frac{n-2}{n}$, ce qui permet, combiné à l'hypothèse de récurrence, de montrer le résultat. En termes de mesures, on a besoin du lemme d'intégration suivant : Soient $a : [\tau], f : \tau \to [\sigma], r : U, p : \tau \to U, q : \sigma \to U$. Alors :

$$\texttt{apply\_rule} : \frac{r \leq a\ p \qquad \forall x (p\ x) \leq f\ x\ q}{r \leq (\texttt{Mlet}\ a\ f)\ q}$$

En d'autres termes, Si $\forall x Pr(f\ x) \in Q \geq p(x)$ alors $Pr($`let x=a in` $f\ x \in Q) \geq a\ p$ (cela revient à intégrer par rapport à a), donc si $a\ p \geq r$ alors $Pr($`let x=a in` $f\ x \in Q) \geq r$, ce qui en termes de mesures donne le lemme annoncé.

A quelques adaptations près pour que tout se combine syntaxiquement, la première prémisse est simplement le fait que la probabilité de prendre dans un graphe de taille n une arête qui n'appartient pas à la coupe minimale est supérieure à $\frac{n-2}{n}$. La deuxième prémisse est quant à elle une conséquence proche de l'hypothèse de récurrence (elle affirme que pour toute arête e, la probabilité pour `min_cut (contraction g e)` d'être égal à C est supérieure à la fonction caractéristique des arêtes hors de C, appliquée à e).
Les détails de la syntaxe sont loin d'être anecdotiques puisque c'est en préparant ces prémisses qu'on se sert de résultats fondamentaux comme le théorème `cut_cons_back`. Cependant les présenter ici serait fastidieux, leurs énoncés étant conséquents, et ne simplifieraient pas nécessairement la compréhension de cette démonstration.

5 conclusion

Ce stage s'est donc finalement constitué d'une part de l'approche et de la compréhension de la bibliothèque de C.Paulin sur les preuves de programmes probabilistes, et d'autre part de la modélisation et de l'étude formelle de l'algorithme de recherche de coupe minimale sur les graphes présenté par Motwani[4]. Ce projet s'est révélé viable et est arrivé à terme, même s'il est certain qu'on peut y apporter de nombreuses améliorations, notament dans la représentation des graphes. Cependant, une grande partie du code produit serait réutilisable dans le cas de développements ultérieurs avec une modélisation plus adaptée. Ce travail, en plus d'avoir constitué un premier exemple conséquent de l'emploi de la bibliothèque de Christine Paulin, a servi à la mettre à l'épreuve et à la compléter, par exemple en ce qui concerne les fonctions caractéristiques et l'expression de résultats déterministes en vocabulaire de mesures.

Références

[1] Joe Hurd. *Formal Verification of Probabilistic Algorithms.* PhD thesis, University of Cambridge, 2002.

[2] Dexter Kozen. A probabilistic PDL. In *15th ACM Symposium on Theory of Computing*, 1983.

[3] Carroll Morgan and Annabelle McIver. pGCL : formal reasoning for random algorithms. *South African Computer Journal*, 1999.

[4] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge, 1995.

[5] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.0*, April 2004. `http://coq.inria.fr`.