Vérification formelle d'algorithmes probabilistes

Tahina Ramananandro
Ecole Normale Supérieure – Paris

27 juin – 25 août 2005

Table des matières

Préface

J'ai effectué ce stage à l'INRIA Sophia-Antipolis[1], au sein de l'équipe *Marelle*, sous la direction de MM. Philippe Audebaud et Laurent Théry, que je remercie ainsi que tous les membres des équipes *Marelle* ainsi que *Everest*.
L'équipe *Marelle* («Mathématiques, Raisonnement, Logiciel») s'occupe de la vérification formelle de logiciels, et travaille autour du système d'aide à la preuve Coq [Coq], tandis que l'équipe *Everest* («Environments for Verification and Security of Software»), dans le même bâtiment, est spécialisée dans la vérification des aspects de sécurité des logiciels (notamment les protocoles cryptographiques).

Je dois dire que l'ambiance a été chaleureuse étant donné qu'à ma grande surprise, tout le monde se tutoie dans le laboratoire. J'ai pu avoir un aperçu significatif du travail réalisé dans ces deux équipes en participant aux séminaires, aussi bien de cryptographie que de sémantique, organisés chaque jeudi après-midi.
MM. les responsables de mon stage m'ont soutenu tout au long de mon expérience, tant sur le fond que du point de vue de la méthode de travail, tout en me laissant la liberté d'explorer le sujet par moi-même, liberté dont j'ai avantageusement profité pour aboutir à une présentation «atypique» selon leurs propres termes. Pour ce qui est de l'environnement de travail, j'ai commencé par une prise de contact avec le système d'aide à la preuve Coq, et j'avoue que j'y ai pris goût – et c'est un doux euphémisme !
Vers la fin du stage – fin août, où il n'y a plus grand monde au laboratoire, pour cause de vacances –, MM. Philippe Audebaud et Laurent Théry m'ont efficacement conseillé pour la réalisation du diaporama de l'exposé final, ainsi que pour la rédaction du présent rapport.

[1]Institut National de Recherche en Informatique et en Automatique – 2004, route des Lucioles – 06902 Sophia-Antipolis Cedex, France

Introduction

Les algorithmes probabilistes interviennent dans de nombreux domaines où une approche déterministe précise est inabordable, souvent en raison de la complexité des algorithmes déterministes. C'est le cas en algorithmique, par exemple en cryptographie (voir la preuve formelle du test de primalité de Miller-Rabin dans [Hur01]).
Souvent, surtout dans des programmes à vocation statistique comme l'échantillonnage de données environnementales (problèmes de localisation d'un robot, mesures de grandeurs physiques), on recourt à un algorithme probabiliste afin de simuler par ordinateur une distribution de probabilité. Bien entendu, de telles simulations sont souvent abordées par une approche mathématique (voir par exemple [Yca02]).
Il s'agit ici de considérer la possibilité d'élaborer des preuves formelles de tels algorithmes de simulation dans un environnement d'aide à la preuve.

Pour pouvoir raisonner formellement sur un algorithme probabiliste, il faut tout d'abord savoir représenter de façon formelle non seulement l'algorithme lui-même, mais aussi les données aléatoires qu'il va utiliser, et la façon dont il va les utiliser (chapitre 1). Pour ce faire, le premier point concerne l'approche de programmation à adopter pour considérer l'algorithme : modèle impératif (le plus utilisé) ou modèle fonctionnel ? J'ai préféré cette dernière approche ; ainsi, les données aléatoires seront modélisées par un flot, et les tirages par des consommations de ce flot. Je considère alors la représentation des algorithmes probabilistes dans un langage fonctionnel, le *Lambda-O*, ou $\lambda_{\bigcirc}$ [PPT05], qui distingue constamment termes déterministes et expressions aléatoires, tant du point de vue de la réduction que du typage.

L'algorithme étant représenté, comment exprimer des propriétés probabilistes s'y rapportant, et surtout comment raisonner avec ces propriétés (chapitre 2) ? Ici encore, plusieurs approches existent. La plus courante est de procéder par obligations de preuve, à l'instar de Why [Why]. Mais pour ma part, j'ai préféré procéder par «transformation» du programme écrit en $\lambda_{\bigcirc}$ vers une formule logique décrivant son comportement et destinée à être utilisée dans le raisonnement probabiliste. Une telle formule ne comporte pas d'éléments probabilistes à proprement parler, ceux-ci n'intervenant que plus tard, dans le raisonnement sur la formule obtenue.
Une fois la formule obtenue, il faut pouvoir raisonner dessus. Ceci implique la formalisation, dans un système de preuve, de la théorie des probabilités, et le lien entre cette théorie et la représentation des données aléatoires que l'on s'est proposée au départ. Pour cette formalisation, j'ai utilisé le système de preuve Coq [Coq]. Une fois formalisée la théorie des probabilités, je l'applique aux flots de données aléatoires et j'intègre également, dans le Calcul des Constructions Inductives [BC04], le langage des formules générées à partir de programmes en $\lambda_{\bigcirc}$.

Cette formalisation tant de l'algorithme probabiliste que de la logique probabiliste mise en place pour raisonner, m'a permis de traiter quelques exemples simples (chapitre 3). Dans le cadre des simulations sur l'espace des flots dont les éléments sont des réels uniformément distribués sur $[0, 1]$, je montre comment prouver formellement en Coq la terminaison et la correction d'algorithmes de simulation de lois simples telles que la loi de Bernoulli (algorithme non récursif), la loi binomiale (récursivité déterministe), ou encore la loi géométrique (récursivité probabiliste).

1 Représentation d'un algorithme probabiliste

Pour définir formellement ce qu'est un algorithme probabiliste, j'ai choisi de suivre une voie commune à [Hur01] et [PPT05] : de façon informelle, on considère qu'un algorithme probabiliste est simplement une fonction qui prend en entrée une donnée aléatoire, mais qui renvoie toujours la même sortie si la même donnée lui est passée en entrée (en somme, c'est une «vraie» fonction).

1.1 Les données aléatoires

Qu'entend-on donc par *donnée aléatoire* ? Il s'agit en fait d'une donnée *tirée au hasard* dans un certain espace de probabilités. Mais il faut pouvoir définir la nature de cette donnée (type, distribution, etc.), et la façon dont elle est utilisée par l'algorithme.
Un algorithme probabiliste effectue des *tirages aléatoires*. Ces tirages satisfont deux propriétés :
– les tirages sont indépendants deux à deux
– tous les tirages suivent la même loi de probabilité

L'approche impérative consiste à des appels d'une procédure du style `rand()`, avec des effets de bord.

L'approche fonctionnelle, que j'ai choisie pour représenter les algorithmes probabilistes, amène à l'utilisation d'un *flot* de données aléatoires, toutes de même type E et suivant la même loi de probabilité. Un tirage sera donc assimilé à la *consommation* d'un élément de ce flot.

Définition 1.1.1 (Flot de données aléatoires ; extraction) *Soit E un ensemble quelconque non vide. On appelle flot sur E toute famille $\varphi \in E^{\mathbb{N}}$.*
Si $\alpha \in E$ et si φ est un flot sur E, alors on note $\alpha :: \varphi$ le flot tel que :

$$(\alpha :: \varphi)_0 = \alpha$$

$$\forall n \in \mathbb{N} : (\alpha :: \varphi)_{n+1} = \varphi_n$$

On appelle fonction extractrice la fonction

$$\begin{array}{rccc} Extr & : & E^{\mathbb{N}} & \to & E \times E^{\mathbb{N}} \\ & & \alpha :: \varphi & \mapsto & (\alpha, \varphi) \end{array}$$

L'application de cette fonction est l'extraction, ou la consommation, d'un élément de flot. Alors, un flot φ' est une queue (ou un sous-flot) du flot φ si, et seulement si, il est la partie droite (π_2) d'un nombre fini d'extractions de φ :

$$\exists n \in \mathbb{N} : \varphi' = (\pi_2 \circ Extr)^n (\varphi)$$

L'algorithme probabiliste sera modélisé par une fonction déterministe qui prendra en entrée un flot de données aléatoires, et qui renverra la donnée «utile» (celle qu'on veut effectivement obtenir et manipuler) et le reste du flot non consommé.

Définition 1.1.2 (Algorithme probabiliste) *Soit F un ensemble non vide. On appelle algorithme probabiliste calculant une donnée de l'ensemble F toute fonction déterministe :*

$$f : E^{\mathbb{N}} \to F \times E^{\mathbb{N}}$$

telle que pour tous flots φ, φ' et pour tout $y \in F$, si $f(\varphi) = (y, \varphi')$, alors φ' est une queue de φ.

Cette définition, toutefois, ne permet pas de garantir que la fonction f extrait tous les éléments du flot qu'elle utilise. Considérons, par exemple, $E = \{\top, \bot\}$, et soit l'algorithme probabiliste f suivant, calculant une donnée de l'ensemble des entiers naturels :

$$\begin{array}{rcl} f(\varphi) & = & (f_1(\varphi), f_2(\varphi)) \\ f_1(\top :: \varphi') & = & 0 \\ f_2(\top :: \varphi') & = & \top :: \varphi' \\ f_1(\bot :: \varphi') & = & f_1(\varphi') + 1 \\ f_2(\bot :: \varphi') & = & \bot :: f_2(\varphi') \end{array}$$

Cette fonction calcule le nombre de $\bot$ en tête de flot, et les extrait, mais utilise le premier $\top$ rencontré dans le flot, sans l'extraire. Comme nous allons le voir, une telle opération est impossible dans le langage que j'ai choisi pour la représentation des algorithmes probabilistes. Mais il est difficile d'exprimer mathématiquement le fait qu'un algorithme consomme tous les éléments de flot qu'il utilise, sans avoir à parler d'une telle représentation dans un langage.
Une approche possible est de dire qu'une telle fonction est «préfixe», c'est-à-dire que pour tous $\alpha_1, \ldots, \alpha_n, \varphi'$, si sous une telle fonction le flot d'entrée $\alpha_1 :: \ldots :: \alpha_n :: \varphi'$ donne le flot de sortie φ', alors pour tout φ'', le flot d'entrée $\alpha_1 :: \ldots :: \alpha_n :: \varphi''$ donne le flot de sortie φ'', ce qui exprime le fait que les données des queues φ' et φ'' des flots d'entrée ne soient pas utilisées.

1.2 Le langage $\lambda_\bigcirc$

Pour la représentation des algorithmes probabilistes, j'ai préféré utiliser un langage fonctionnel : le langage *Lambda-O*, ou $\lambda_\bigcirc$, mis au point par Park, Pfenning et Thrun en 2005 [PPT05][2], et qui a été présenté au *POPL'05*. Le $\lambda_\bigcirc$ était initialement destiné à étendre le langage OCaml [OCa] pour des applications comme la localisation d'un robot.

Certes, les algorithmes probabilistes sont le plus souvent représentés dans des langages impératifs, mais justement, j'ai préféré me placer dans le modèle fonctionnel pour sortir des sentiers battus et aborder ainsi une approche assez rare des algorithmes probabilistes.

Le langage $\lambda_\bigcirc$ est fondé sur une nouvelle approche *monadique* des langages fonctionnels, introduite par Park et Pfenning. Cette approche monadique consiste à distinguer deux constructions du langage :
– les *termes*, dont la réduction est appelée *évaluation* et n'utilise en rien la monade.
– les *expressions* dont la réduction est appelée *calcul* et peut utiliser la monade.

Le langage $\lambda_\bigcirc$ est un des produits de cette approche. Du point de vue probabiliste, les constructions de termes et d'expressions jouent les rôles suivants :
– les *termes*, dont le résultat est déterministe quel que soit le flot, qui n'est pas consommé par l'évaluation.
– les *expressions*, dont le résultat est rendu aléatoire par le fait que le calcul consomme au besoin des éléments du flot : il s'agit en fait de la représentation de *variables aléatoires*.

Le langage $\lambda_\bigcirc$ a fait l'objet d'une implémentation par les auteurs. De mon côté, j'ai également écrit, en OCaml, un petit interpréteur (avec parseur et *type-checker*) qui m'a servi de plate-

[2]Je ne considère ici que la première définition du langage, c'est-à-dire sans prendre en compte les extensions pour le calcul approché.

forme d'expérimentation, mais m'a également servi de base pour la suite de mes travaux. Je l'ai baptisé *Proba Caml*[3].

Parmi les termes, outre les constructions classiques d'un λ-calcul typé simple, le $\lambda_{\bigcirc}$ permet de représenter des distributions de probabilités (structure prob) comme *mesure-image* de la mesure sur l'espace de probabilité des flots par la variable . Ce sont les expressions qui permettent d'extraire des échantillons des distributions de probabilités (structure sample).

Définition 1.2.1 (Ensemble de base, valeur de base) *Fixons une fois pour toutes un ensemble T_0 d'ensembles dits «de base». On appelle alors valeur de base tout élément $v \in \tau$ où $\tau \in T_0$ est un ensemble de base.*

Dans *Proba Caml*, j'ai implémenté les valeurs de base des entiers naturels et des réels.

Définition 1.2.2 (Termes et expressions) *Soit $\mathcal{V}$ un ensemble non vide de noms de variables. Alors, l'ensemble $Term$ des termes du $\lambda_{\bigcirc}$ et l'ensemble $Expr$ des expressions du $\lambda_{\bigcirc}$ sont définis par les grammaires suivantes :*

$Term$	::=	| true		*booléens*
		| false		
		| if a then b else c	$(a, b, c \in Term)$	*test*
		| v	$(v \in \tau,\ \tau \in T_0)$	*valeur de base*
		| x	$(x \in \mathcal{V})$	*variable*
		| (a, b)	$(a, b \in Term)$	*couple*
		| fst a	$(a \in Term)$	*projection gauche*
		| snd a	$(a \in Term)$	*projection droite*
		| $\lambda x.\ c$	$(x \in \mathcal{V},\ c \in Term)$	*abstraction*
		| $a\ b$	$(a, b \in Term)$	*application*
		| prob e	$(e \in Expr)$	*distribution de probabilité*
		| fix $x.\ a$	$(x \in \mathcal{V},\ a \in Term)$	*terme récursif*
$Expr$	::=	| a	$(a \in Term)$	*terme (déterministe)*
		| $\mathcal{S}$		*extraction d'un élément du flot*
		| sample x from a in e	$(x \in \mathcal{V},\ a \in Term,\ e \in Expr)$	*échantillonnage d'une distribution*

Dans cette définition, les valeurs booléennes sont particularisées à cause du test if, qui exige explicitement leur présence.

[3]Ou encore ALICE, pour «**Al**ea-**I**ncluding **C**omputations and **E**valuations», «calculs et évaluations avec aléa». – Sources et mise en œuvre sur une machine : cf. annexe p. 53

On distingue deux types de réduction : l'*évaluation* (réduction des termes, n'utilisant pas le flot) et le *calcul* (réduction des expressions, pouvant utiliser le flot).
La réduction d'un terme ou expression doit aboutir à un terme appelé *valeur* au sens de la :

Définition 1.2.3 (Valeur) *L'ensemble Val des valeurs est un sous-ensemble de l'ensemble $Term$ des termes. Cet ensemble est défini par la grammaire suivante :*

Val	$::=$	$\mid$	true	*booléens*
		$\mid$	false	
		$\mid$	$v \quad (v \in \tau,\ \tau \in T_0)$	*valeur de base*
		$\mid$	$\lambda x.\ c \quad (x \in \mathcal{V},\ c \in Term)$	*abstraction*
		$\mid$	$\mathsf{prob}\ e \quad (e \in Expr)$	*distribution de probabilité*
		$\mid$	$(a, b) \quad (a, b \in Val)$	*couple de valeurs*

On présente ici les règles de réduction en un pas : la sémantique proposée est une sémantique de réduction *à petits pas*. On notera $\mapsto$ un pas d'évaluation, $\Rightarrow$ un pas de calcul, et $\mapsto^*, \Rightarrow^*$ les clôtures réflexives et transitives correspondantes.

1.2.1 Evaluation des termes

Les termes sont évalués selon le principe de l'appel *par valeur*. Le flot n'intervient pas : la relation $\mapsto$ est donc simplement une relation entre termes.

Test : le booléen est évalué d'abord. Lorsque l'on tombe sur une valeur booléenne, on peut alors effectuer le branchement.

$$\frac{a \mapsto a'}{\mathsf{if}\ a\ \mathsf{then}\ b\ \mathsf{else}\ c \ \mapsto\ \mathsf{if}\ a'\ \mathsf{then}\ b\ \mathsf{else}\ c} \qquad \frac{}{\mathsf{if\ true\ then}\ b\ \mathsf{else}\ c \ \mapsto b} \qquad \frac{}{\mathsf{if\ false\ then}\ b\ \mathsf{else}\ c \ \mapsto c}$$

Couple : le membre gauche est évalué d'abord.

$$\frac{a \mapsto a'}{(a, b) \mapsto (a', b)} \qquad \frac{v \in Val \quad b \mapsto b'}{(v, b) \mapsto (v, b')}$$

Projections : elles ne sont évaluées qu'une fois que les deux membres du couple l'ont été.

$$\frac{c \mapsto c'}{\mathsf{fst}\ c \ \mapsto\ \mathsf{fst}\ c'} \qquad \frac{c \mapsto c'}{\mathsf{snd}\ c \ \mapsto\ \mathsf{snd}\ c'} \qquad \frac{v_g, v_d \in Val}{\mathsf{fst}\ (v_g, v_d) \ \mapsto v_g} \qquad \frac{v_g, v_d \in Val}{\mathsf{snd}\ (v_g, v_d) \ \mapsto v_d}$$

Application : c'est la règle d'*élimination* de l'abstraction.
La fonction est évaluée d'abord. Puis l'argument est évalué, avant que n'intervienne la β-réduction.

$$\frac{a \mapsto a'}{a\ b \ \mapsto\ a'\ b} \qquad \frac{b \mapsto b'}{(\lambda x.\ c)\ b \ \mapsto\ (\lambda x.\ c)\ b'} \qquad \frac{v \in Val}{(\lambda x.\ c)\ v \ \mapsto c\,[x := v]}$$

où la substitution est définie de manière évidente (moyennant les α-renommages). On voit ici que la β-réduction n'apparaît que lorsque l'argument est une valeur. C'est dire que les termes suivent un schéma d'évaluation en appel par valeur.

Point fixe : un dépliage («unfold»).

$$\frac{}{\mathsf{fix}\ x.\ a \quad \mapsto a\,[x \ := \ \mathsf{fix}\ x.\ a]}$$

On montre facilement qu'à partir d'un terme a il existe au plus un terme a' tel que $a \mapsto a'$. On montre alors, par induction sur la structure du terme, qu'entre un terme et une valeur, il existe au plus un chemin d'évaluation.

1.2.2 Calcul des expressions

Ici, le flot intervient. La relation $\Rightarrow$ est donc une relation entre les couples expression-flot, on écrira $e \ @\ s \quad \Rightarrow \quad e' \ @\ s'$.

Terme en tant qu'expression : le calcul d'un terme correspond à son évaluation, sans consommer le flot.

$$\frac{a \mapsto a'}{a \ @\ s \quad \Rightarrow \quad a' \ @\ s}$$

Consommation d'un élément du flot.

$$\frac{}{\mathcal{S} \ @\ x :: t \quad \Rightarrow \quad x \ @\ t}$$

Echantillonnage : c'est la règle correspondant à l'*élimination* du terme prob.
On évalue d'abord le terme correspondant à la distribution de probabilité. Puis, lorsqu'on tombe sur une valeur prob, on évalue son «expression-corps», en modifiant au besoin le flot, jusqu'à tomber sur une valeur que l'on substitue à la variable d'échantillonnage, en renvoyant le nouveau flot.

$$\frac{a \mapsto a'}{\mathsf{sample}\ x\ \mathsf{from}\ a\ \mathsf{in}\ e \ \ @\ s \quad \Rightarrow \quad \mathsf{sample}\ x\ \mathsf{from}\ a'\ \mathsf{in}\ e \ \ @\ s}$$

$$\frac{e_x \ @\ s \quad \Rightarrow \quad e'_x \ @\ s'}{\mathsf{sample}\ x\ \mathsf{from\ prob}\ e_x\ \mathsf{in}\ e \ \ @\ s \quad \Rightarrow \quad \mathsf{sample}\ x\ \mathsf{from\ prob}\ e'_x\ \mathsf{in}\ e \ \ @\ s'}$$

$$\frac{v \in Val}{\mathsf{sample}\ x\ \mathsf{from\ prob}\ v\ \mathsf{in}\ e \ \ @\ s' \quad \Rightarrow \quad e\,[x := v] \ @\ s'}$$

Ici encore, on montre qu'à flot fixé s, à partir d'une expression e il existe au plus une expression e' et un flot s' tels que $e \ @\ s \quad \Rightarrow \quad e' \ @\ s'$. Donc, à flot fixé, entre une expression et une valeur, il existe au plus un chemin de calcul.
On montre facilement, par induction sur la structure de l'expression, que le flot résultant est un sous-flot du flot d'entrée et que seuls sont utilisés les éléments extraits.

En λ-calcul la pierre angulaire de l'évaluation est la β-réduction. En fait, on constate qu'en $\lambda_{\bigcirc}$ elle intervient à deux niveaux : à côté de la β-réduction classique sur les termes, on peut considérer que les règles de calcul du sample constituent une β-réduction au niveau des expressions, où le sample joue le rôle d'application et le prob joue le rôle d'abstraction. En somme, comme les règles de l'application correspondent à l'*élimination* de l'abstraction, celles du sample correspondent à l'élimination du terme prob. Cf. fig. 1 p. 10.

	Termes	**Expressions**
Abstraction (terme)	$\lambda x.\ c$	prob c
Corps c	Terme	*Expression*
Application «$f(a)$»	$f\ a$	sample y from f in e
Argument	Explicite : terme a	Implicite : flot
Nature de l'application	Terme	*Expression*
Résultat	Valeur renvoyée	Valeur utilisée dans e sous la variable y Flot de sortie passé en entrée de e
β-réduction	$f[x := a]$	Calcul du corps de la prob sous le flot d'entrée du sample

FIG. 1 – Les deux niveaux de β-réduction en $\lambda_{\bigcirc}$

1.2.3 Exemple de réduction

Soit l'expression suivante :

sample x from prob $\mathcal{S}$ in
  sample y from
  if $x > 18$ then prob 0 else prob $\mathcal{S}$
  in $x + y$

(On suppose que le type τ_0 des éléments de flot est le type des entiers naturels.)
Etudions cet exemple sous le flot d'entrée $15 :: 1500 :: 18 :: \varphi'$ (avec φ' quelconque mais fixé).

1. L'expression est un sample, évaluons le terme introduit par from : c'est prob $\mathcal{S}$. Comme c'est un terme prob, calculons son expression-corps, qui est $\mathcal{S}$: extraction de flot. On obtient alors l'expression suivante :

   sample x from prob 15 in
     sample y from
      if $x > 18$ then prob 0 else prob $\mathcal{S}$
     in $x + y$

   et le flot est maintenant $1500 :: 18 :: \varphi'$.

2. L'expression-corps est 15, c'est donc une valeur que l'on peut substituer à x dans l'expression introduite par in. On obtient l'expression suivante :

sample y from
  if $15 > 18$ then prob 0 else prob $\mathcal{S}$
in $15 + y$

et le flot est toujours $1500 :: 18 :: \varphi'$.

3. L'expression est un sample : évaluons le terme introduit par from. C'est un terme if : évaluons le booléen, $15 > 18$. Considérant qu'il s'agit d'un opérateur de base, et que 15 et 18 sont deux valeurs, ne faisons pas de détail sur la façon dont il est évalué, et disons directement que le booléen obtenu est false :

   sample y from
     if false then prob 0 else prob $\mathcal{S}$
   in $15 + y$

   Comme on est en train d'évaluer un terme, le flot est inchangé.

4. On peut maintenant effectuer le branchement :

   sample y from prob $\mathcal{S}$ in $15 + y$

5. C'est donc un terme prob dont on va pouvoir calculer l'expression-corps, qui une nouvelle fois est une extraction de flot . On obtient donc :

   sample y from prob 1500 in $15 + y$

   et le nouveau flot est $18 :: \varphi'$.
6. Maintenant, l'expression sous le prob est une valeur que l'on peut substituer à y dans l'expression $15 + y$:

   $15 + 1500$

7. C'est désormais un terme. Les deux arguments du + étant des valeurs, là encore, pas de détail. On obtient donc finalement :

   1515

   et le flot de sortie est $18 :: \varphi'$.

Il y a donc eu deux consommations. Notons que si le flot d'entrée avait été $1500 :: 15 :: 18 :: \varphi'$, alors on aurait obtenu le résultat 1500 et le flot de sortie $15 :: 18 :: \varphi'$ après une seule consommation.

1.2.4 Typage

Le $\lambda_{\bigcirc}$ est simplement typé. Mais on distinguera deux notations pour le typage des termes et celui des expressions :
– $a : t$ indique que le terme a est déterministe dans le type t
– on écrira plutôt $e \div t$ pour dire que l'expression e est une *variable aléatoire* sur le type t.

Définition 1.2.4 (Types) *On confond un type de base et l'ensemble de base $\tau \in T_0$ auquel il correspond. L'ensemble Ty des types du $\lambda_{\bigcirc}$ sur les types de base T_0 est défini par la grammaire suivante :*

$$\begin{array}{rcll} Ty & ::= & \mid \ \mathsf{bool} & \\ & & \mid \ \tau & (\tau \in T_0) \\ & & \mid \ t_1 \to t_2 & (t_1, t_2 \in Ty) \\ & & \mid \ t_1 \times t_2 & (t_1, t_2 \in Ty) \\ & & \mid \ \bigcirc\, t & (t \in Ty) \end{array}$$

Dans cette définition des types, le type bool est particularisé car, comme on l'a vu, le terme de test if exige la présence explicite des booléens.

Soit Γ un *contexte de typage* : une partie de $\mathcal{V} \times Ty$, associant un type à chaque variable du contexte.
Attention : le typage est *déterministe*[4].

Typage des termes.
Booléens et test : introduction et élimination du type bool.

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}} \quad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}} \quad \frac{\Gamma \vdash a : \mathsf{bool} \quad \Gamma \vdash b : t \quad \Gamma \vdash c : t}{\Gamma \vdash \mathsf{if}\ a\ \mathsf{then}\ b\ \mathsf{else}\ c\ : t}$$

Valeur de base et variable :

$$\frac{\tau \in T_0 \quad v \in \tau}{\Gamma \vdash v : \tau} \quad \frac{(x,t) \in \Gamma}{\Gamma \vdash x : t}$$

Couple et projections : introduction et élimination du type couple $\times$.

$$\frac{\Gamma \vdash a : t_a \quad \Gamma \vdash b : t_b}{\Gamma \vdash (a,b) : t_a \times t_b} \quad \frac{\Gamma \vdash c : t_g \times t_d}{\Gamma \vdash \mathsf{fst}\ c\ : t_g} \quad \frac{\Gamma \vdash c : t_g \times t_d}{\Gamma \vdash \mathsf{snd}\ c\ : t_d}$$

Abstraction et application : introduction et élimination du type flèche $\to$.

$$\frac{\Gamma \cup \{(x,t_x)\} \vdash c : t_c}{\Gamma \vdash \lambda x : t_x.\ c\ : t_x \to t_c} \quad \frac{\Gamma \vdash a : t_b \to t \quad \Gamma \vdash b : t_b}{\Gamma \vdash a\ b\ : t}$$

Distribution de probabilité : introduction du type $\bigcirc$.

$$\frac{\Gamma \vdash e \div t}{\Gamma \vdash \mathsf{prob}\ e\ : \bigcirc\, t}$$

[4]On pourrait le rendre en quelque sorte «probabiliste» en autorisant les types somme ; mais au lieu de parler de «probabilité que le résultat soit d'un certain type», on parlerait plutôt de «probabilité que le résultat commence par un certain constructeur de type somme», le type somme étant déterminé par les règles de typage, qui restent tout à fait déterministes.

Point fixe :

$$\frac{\Gamma \cup \{(x,t)\} \vdash a : t}{\Gamma \vdash \mathsf{fix}\ x : t.\ a\ \ : t}$$

Typage des expressions. On considère que les éléments du flot sont d'un type $\tau_0 \in T_0 \cup \{\mathsf{bool}\}$ fixé.
Terme en tant qu'expression :

$$\frac{\Gamma \vdash a : t}{\Gamma \vdash a \div t}$$

Extraction d'un élément du flot :

$$\frac{}{\Gamma \vdash \mathcal{S} \div \tau_0}$$

Echantillonnage d'une distribution de probabilité : élimination du type $\bigcirc$.

$$\frac{\Gamma \vdash a : \bigcirc\ t_x \quad \Gamma \cup \{(x,t_x)\} \vdash e \div t}{\Gamma \vdash \mathsf{sample}\ x\ \mathsf{from}\ a\ \mathsf{in}\ e\ \ \div t}$$

On montre facilement, par induction sur la structure de la relation de réduction correspondante, que l'évaluation et le calcul conservent le type.
On montre également que si un terme (resp. une expression) s'évalue (resp. se calcule) en une valeur, alors il (resp. elle) est bien typé(e) (et bien entendu son type correspond à celui de la valeur obtenue).

1.3 Représentation d'un algorithme en $\lambda_\bigcirc$

Un algorithme, déterministe ou probabiliste, peut être modélisé par un terme du $\lambda_\bigcirc$.

Si l'algorithme a besoin d'un certain nombre de *paramètres déterministes*, alors on les déclare au moyen d'*abstractions* ; lors de la mise en œuvre de l'algorithme, on fournira ces paramètres au moyen d'*applications*. Dans un premier temps, donc, l'algorithme est évalué de façon *déterministe*, sans utiliser le flot.
Evaluons donc un tel terme de mise en œuvre de l'algorithme, et regardons la valeur obtenue.

- Abstraction : cela signifie que l'on n'a pas donné tous les paramètres déterministes dont l'algorithme a besoin pour tourner.
- Valeur de type de base : l'algorithme a tourné et la valeur retournée est déterministe, ne dépend pas du flot.
- Structure prob : il s'agit d'un *algorithme probabiliste*, qu'il faut mettre en œuvre au moyen d'une *expression*.

Prenons par exemple le terme suivant, qui modélise l'algorithme de simulation de la loi de Bernoulli de paramètre p donné de façon déterministe lors de sa mise en œuvre (en supposant que les flots soient des «suites de variables aléatoires réelles uniformément distribuées sur $[0,1]$») :

$$\text{bernoulli} \ := \ \lambda p.\ \mathsf{prob\ sample}\ x\ \mathsf{from\ prob}\ \mathcal{S}\ \mathsf{in}\ x \leq p$$

L'instanciation se fera donc en deux coups : d'abord un terme d'application pour renseigner le paramètre déterministe p, puis une expression sample pour mettre en œuvre effectivement

l'algorithme probabiliste. Le tout est donc réalisé par l'expression suivante (étant donné un réel p_1) :

$$\mathsf{sample}\ x\ \mathsf{from}\ \text{bernoulli}\ p_1\ \mathsf{in}\ x$$

Conformément aux règles de calcul, le terme introduit par from est évalué en premier : ici, cela correspond à la donnée, par application, du paramètre déterministe p_1. On obtient une valeur prob, qui permet enfin le calcul proprement dit de l'expression correspondant à l'algorithme probabiliste instancié.

2 Mécanisation du raisonnement sur des algorithmes probabilistes

Etant donné un algorithme probabiliste, sous la forme d'un terme du $\lambda_{\bigcirc}$ (que l'on appellera *programme*), on veut énoncer et prouver des propriétés sur l'algorithme en considérant le comportement du terme une fois instancié.
Par exemple, pour reprendre l'exemple de la loi de Bernoulli :

$$\lambda p.\ \mathsf{prob\ sample}\ x\ \mathsf{from\ prob}\ \mathcal{S}\ \mathsf{in}\ x \leqslant p$$

on veut pouvoir prouver que si l'algorithme est instancié avec un paramètre $p \in [0,1]$, alors le terme prob représente un algorithme de simulation de la loi de Bernoulli de paramètre p.

2.1 Formules associées à un programme : le transformateur Π

L'idée est d'associer à un programme une formule logique qui décrive son comportement, afin de pouvoir raisonner directement en Coq avec cette formule. Ces formules, décrivant comment le programme s'évalue ou se calcule et utilise le flot, sont toutes *déterministes*. L'aspect probabiliste vient plus tard, dans le raisonnement sur la formule générée en considérant des probabilités sur le flot d'entrée (dont seule l'utilisation «brute» par le programme est exprimée par la formule générée).

Je construis la formule par *induction sur la structure* du terme ou de l'expression, et suivant la *sémantique opérationnelle* du $\lambda_{\bigcirc}$.

Le procédé que j'ai choisi est analogue à celui du typage : c'est d'ailleurs en modifiant la procédure de typage de mon interpréteur *Proba Caml* que j'ai écrit, toujours en OCaml [OCa], le programme de génération de la formule Coq à partir du code $\lambda_{\bigcirc}$ donné en entrée, programme que j'ai appelé «transformateur Π» (car, par exemple, pour un terme m, je désigne par Π_m la «**p**ropriété vérifiée par le résultat de l'évaluation du terme m»).

En toute rigueur, pour chaque élément de langage, terme ou expression, on doit disposer d'informations non seulement sur sa correction, mais aussi sur sa terminaison. Ce sont donc deux formules qu'il faut générer.

Supposons que l'on se place dans un contexte de variables Γ. Soient e une expression et P et Q deux formules. Alors, lorsque l'on écrit le *jugement* suivant :

$$\Gamma \ \vdash\ \{P\}\ e\ \{Q\}$$

on dit en fait que P est une *condition suffisante de terminaison* et Q est une *propriété de correction* vérifiée par le résultat et le flot de sortie *sous l'hypothèse que le programme termine.* Un tel jugement se lit : «si P est vérifiée, alors le calcul de e termine et Q est vérifiée».

Remarque : l'espace des variables libres de P est $\Gamma \cup \{\sigma\}$, où σ désigne le flot d'entrée, tandis que l'espace des variables libres de Q est $\Gamma \cup \{\sigma, \_, \sigma'\}$, où σ' désigne le flot de sortie et $\_$ le résultat du calcul de e. Si e est un terme, alors P et Q ne dépendent ni de σ, ni de σ'. *Dans toute la suite, nous utiliserons ces conventions de notation pour désigner le flot d'entrée, la valeur résultat et le flot de sortie du calcul d'une expression.*

2.1.1 Formules spéciales : présentation informelle

Ce qu'il ne fallait surtout pas faire, c'est d'autoriser l'apparition de termes et d'expressions tels quels dans les formules générées. En effet, ces formules sont destinées à être utilisées dans Coq et le calcul des constructions inductives. Traduire directement les termes en autorisant par exemple l'apparition de l'application nécessiterait en fait que j'adopte les règles de réduction de Coq lui-même sur les termes. Or, je souhaite raisonner *uniquement dans la logique d'ordre supérieur* sur les formules, et en particulier exprimer dans la logique la réduction elle-même.

Il a donc fallu créer des formules spéciales pour le couple, l'application et le sample. Ces formules permettent une approche *relationnelle* des structures du langage $\lambda_{\bigcirc}$, à la manière de ce que l'on peut faire en PROLOG.

De façon informelle, le langage des formules sera engendré par les formules «élémentaires» suivantes :

– pour un couple (a, b), on introduit la relation notée :

$$a \underset{\otimes}{\rightleftharpoons} b$$

qui exprime que le résultat _ de l'évaluation de ce terme est un couple dont la partie gauche (resp. droite) est le résultat de l'évaluation de a (resp. b).

– pour une application $f\ a$, on introduit la relation notée :

$$a \underset{\triangleright}{\overset{f}{\longrightarrow}} \_$$

qui exprime que le résultat _ de l'évaluation de ce terme est le résultat de l'application de f à a.

– pour un sample x from m in e, on introduit la relation notée :

$$s \underset{\square}{\overset{m}{\longrightarrow}} \langle x, s' \rangle$$

qui exprime que l'évaluation de m donne un terme prob et que le calcul de l'expression-corps de ce prob sous le flot d'entrée s donne une valeur x et un flot de sortie s' (qui sera passé en entrée au calcul de e).

Bien entendu, il s'agit ici d'une présentation informelle, et dans la pratique, les termes et expressions apparaissant dans les relations seront remplacés par des variables ou des valeurs de base (ou plus généralement des «termes de logique» obtenus à partir de variables et valeurs de base par des opérations de base fixées une fois pour toutes, par exemple l'addition sur les réels).

2.1.2 Formules spéciales : première définition

Je présente ici de manière plus ou moins formelle le mécanisme que j'ai adopté et employé pour les formules de correction, mécanisme adapté aux programmes ne contenant pas de point fixe.

Voici le mécanisme d'*introduction* des flèches $\underset{\otimes}{\rightleftharpoons}$, $\underset{\triangleright}{\longrightarrow}$ et $\underset{\square}{\longrightarrow}$.

– pour les projections $\mathsf{fst}\ c$ et $\mathsf{snd}\ c$, que nous noterons a et b, on génère la formule $\Pi_c(\_)$ correspondant à c et dépendant du résultat $\_$ de son évaluation, et alors on obtient :

$$\Pi_a(\_) \ := \ \exists c', \exists b : \ \_ \overset{c'}{\underset{\otimes}{\rightleftharpoons}} b \ \wedge \ \Pi_c(c')$$

$$\Pi_b(\_) \ := \ \exists c', \exists a : \ a \overset{c'}{\underset{\otimes}{\rightleftharpoons}} \_ \ \wedge \ \Pi_c(c')$$

– pour une application $f\ a$, que nous noterons y, on génère les formules $\Pi_f(\_)$ et $\Pi_a(\_)$ correspondant aux formules vérifiées par f et par a, dépendant chacune du résultat $\_$ de leur évaluation, et alors on obtient pour y :

$$\Pi_y(\_) \ := \ \exists f', \exists a' : \ \Pi_f(f') \ \wedge \ \Pi_a(a') \ \wedge \ a' \underset{\triangleright}{\xrightarrow{f'}} \_$$

– pour une expression $\mathsf{sample}\ x\ \mathsf{from}\ m\ \mathsf{in}\ e$, que nous noterons g, on génère les formules $\Pi_m(\_)$, correspondant à m et dépendant du résultat $\_$ de l'évaluation de m, et $\Pi_e(x, \sigma, \_, \sigma')$, correspondant à e et dépendant de la variable x libre dans e, du flot d'entrée σ passé à e, et du résultat $\_$ et du flot de sortie σ' obtenus lors du calcul de e. Et alors on obtient :

$$\Pi_g(\sigma, \_, \sigma') \ := \ \exists s, \exists x, \exists m' : \ \Pi_m(m') \ \wedge \ \sigma \underset{\square}{\xrightarrow{m'}} \langle x, s \rangle \ \wedge \ \Pi_e(x, s, \_, \sigma')$$

où, cette fois, σ désigne le flot d'entrée, $\_$ le résultat et σ' le flot de sortie lors du calcul de *toute l'expression* sample.

Pour pouvoir raisonner avec de telles formules, il est nécessaire de disposer également d'un mécanisme d'*élimination* de telles formules :

– pour un couple (a, b), que nous noterons c, on génère les formules $\Pi_a(\_)$ et $\Pi_b(\_)$ correspondant à a et b et dépendant du résultat $\_$ de leurs évaluations respectives, et alors on obtient :

$$\Pi_c(\_) \ := \ \forall a', \forall b' : \ a' \overset{}{\underset{\otimes}{\rightleftharpoons}} b' \ \Rightarrow \ \Pi_a(a') \ \wedge \ \Pi_b(b')$$

où $\_$ représente le résultat de l'évaluation du couple c.

– pour une abstraction $\lambda x.\ c$, que nous noterons f, on génère la formule $\Pi_c(x, \_)$, dépendant de la variable x libre dans c et du résultat $\_$ de l'évaluation de c à x fixé, et alors on obtient :

$$\Pi_f(\_) \ := \ \forall x, \forall y : \ x \underset{\triangleright}{\Rightarrow} y \ \Rightarrow \ \Pi_c(x, y)$$

En effet, dans cette formule, $\_$, résultat de l'évaluation du terme abstraction f, désigne le terme abstraction lui-même (d'où son occurrence sur la flèche $\underset{\triangleright}{\Rightarrow}$), car c'est une valeur.

– pour un terme $\mathsf{prob}\ e$, que nous noterons m, on génère la formule $\Pi_e(\sigma, \_, \sigma')$ correspondant à e et dépendant du flot d'entrée σ passé à e et du résultat $\_$ et du flot de sortie σ' obtenus lors du calcul de e sous σ, et alors on obtient :

$$\Pi_m(\_) \;:=\; \forall s, \forall y, \forall s' :\; s \xrightarrow[\square]{=} \langle y, s' \rangle \;\Rightarrow\; \Pi_e(s, y, s')$$

Là encore, _ représente le terme prob lui-même, car c'est une valeur.

Une quatrième structure qui aurait pu faire l'objet de la création d'une nouvelle formule logique est l'expression $\mathcal{S}$ de consommation de l'élément de flot. Seulement, si j'avais créé une telle relation, le mécanisme d'introduction aurait été évident puisqu'il aurait correspondu à la formule pour $\mathcal{S}$, mais il n'aurait pas existé *a priori* de mécanisme d'élimination d'une telle relation. C'est pourquoi j'ai opté cette fois-ci pour l'ajout d'un «terme de formule», le *conse* ::, qui, en fait, correspond plutôt à un terme de théorie des flots :

$$\Pi_{\mathcal{S}}(\sigma, \_, \sigma') \;:=\; \sigma = \_ :: \sigma'$$

Exemple. Supposons que l'on dispose des réels avec les opérations et relations élémentaires. Supposons que le type τ_0 des éléments de flot soit le type R des réels. Considérons le terme suivant :

$$\lambda p : \mathsf{R}.\ \mathsf{prob\ sample}\ x\ \mathsf{from\ prob}\ \mathcal{S}\ \mathsf{in}\ x \leqslant p$$

C'est une abstraction. Alors, générons la formule $\Pi_1(p, \_)$ de son terme-corps :

$$m_1 :=\ \mathsf{prob\ sample}\ x\ \mathsf{from\ prob}\ \mathcal{S}\ \mathsf{in}\ x \leqslant p$$

Le terme m_1 est un prob, alors, générons la formule $\Pi_{11}(p, \sigma, \_, \sigma')$ de son expression-corps :

$$e_{11} :=\ \mathsf{sample}\ x\ \mathsf{from\ prob}\ \mathcal{S}\ \mathsf{in}\ x \leqslant p$$

L'expression e_{11} est un sample. Alors, générons d'abord la formule $\Pi_{111}(p, \_)$ du terme introduit par from :

$$m_{111} :=\ \mathsf{prob}\ \mathcal{S}$$

Le terme m_{111} est un prob, alors, générons la formule $\Pi_{1111}(p, \sigma, \_, \sigma')$ de son expression-corps :

$$e_{1111} :=\ \mathcal{S}$$

L'expression e_{1111} est un $\mathcal{S}$, sa formule vient immédiatement :

$$\Pi_{1111}(p, \sigma, \_, \sigma') :=\ \sigma = \_ :: \sigma'$$

On revient alors à la formule $\Pi_{111}(p, \_)$ du terme m_{111} qui est un prob : c'est donc

$$\begin{aligned} \Pi_{111}(p, \_) \;&:=\; \forall s_{111}, \forall y_{111}, \forall s'_{111} :\; s_{111} \xrightarrow[\square]{=} \langle y_{111}, s'_{111} \rangle \;\Rightarrow \Pi_{1111}(p, s_{111}, y_{111}, s'_{111}) \\ &\equiv\; \forall s_{111}, \forall y_{111}, \forall s'_{111} :\; s_{111} \xrightarrow[\square]{=} \langle y_{111}, s'_{111} \rangle \;\Rightarrow s_{111} = y_{111} :: s'_{111} \end{aligned}$$

Revenons donc à l'expression e_{11} : il faut maintenant générer la formule $\Pi_{112}(p, x, \sigma, \_, \sigma')$ de l'expression introduite par in :

$$e_{112} :=\ x \leqslant p$$

C'est un terme : il y aura donc $\sigma' = \sigma$, et comme on a un terme spécial $\leqslant$, on peut dire que la formule obtenue est :

$$\Pi_{112}(p, x, \sigma, \_, \sigma') := \ (\_ = \mathsf{true} \Leftrightarrow x \leqslant p) \ \wedge \ \sigma' = \sigma$$

Revenant à e_{11}, on obtient désormais :

$$\begin{array}{rcl}
\Pi_{11}(p, \sigma, _, \sigma') & := & \exists s_{11}, \exists x_{11}, \exists m_{111} : \\
& & \Pi_{111}(p, m_{111}) \\
& & \wedge \ \sigma \xrightarrow[\square]{m_{111}} \langle x_{11}, s_{11} \rangle \\
& & \wedge \ \Pi_{112}(p, x_{11}, s_{11}, _, \sigma') \\
& & \\
& \equiv & \exists s_{11}, \exists x_{11}, \exists m_{111} : \\
& & \left(\forall s_{111}, \forall y_{111}, \forall s'_{111} : \ s_{111} \xrightarrow[\square]{m_{111}} \langle y_{111}, s'_{111} \rangle \ \Rightarrow s_{111} = y_{111} :: s'_{111}\right) \\
& & \wedge \ \sigma \xrightarrow[\square]{m_{111}} \langle x_{11}, s_{11} \rangle \\
& & \wedge \ ((_ = \mathsf{true} \Leftrightarrow x_{11} \leqslant p) \ \wedge \ \sigma' = s_{11})
\end{array}$$

qui est déjà difficile à lire.
Pour en revenir au terme m_1, qui est un prob, on obtient la formule suivante :

$$\begin{array}{rcl}
\Pi_1(p, _) & := & \forall s_1, \forall y_1, \forall s'_1 : \\
& & s_1 \xRightarrow[\square]{} \langle y_1, s'_1 \rangle \ \Rightarrow \\
& & \exists s_{11}, \exists x_{11}, \exists m_{111} : \\
& & \left(\forall s_{111}, \forall y_{111}, \forall s'_{111} : \ s_{111} \xrightarrow[\square]{m_{111}} \langle y_{111}, s'_{111} \rangle \ \Rightarrow s_{111} = y_{111} :: s'_{111}\right) \\
& & \wedge \ s_1 \xrightarrow[\square]{m_{111}} \langle x_{11}, s_{11} \rangle \\
& & \wedge \ ((y_1 = \mathsf{true} \Leftrightarrow x_{11} \leqslant p) \ \wedge \ s'_1 = s_{11})
\end{array}$$

Et enfin, la formule pour notre terme abstraction de départ :

$$\begin{array}{rcl}
\Pi(_) & := & \forall p, \forall m_1 : \\
& & p \xRightarrow[\triangleright]{} m_1 \ \Rightarrow \\
& & \forall s_1, \forall y_1, \forall s'_1 : \\
& & s_1 \xrightarrow[\square]{m_1} \langle y_1, s'_1 \rangle \ \Rightarrow \\
& & \exists s_{11}, \exists x_{11}, \exists m_{111} : \\
& & \left(\forall s_{111}, \forall y_{111}, \forall s'_{111} : \ s_{111} \xrightarrow[\square]{m_{111}} \langle y_{111}, s'_{111} \rangle \ \Rightarrow s_{111} = y_{111} :: s'_{111}\right) \\
& & \wedge \ s_1 \xrightarrow[\square]{m_{111}} \langle x_{11}, s_{11} \rangle \\
& & \wedge \ ((y_1 = \mathsf{true} \Leftrightarrow x_{11} \leqslant p) \ \wedge \ s'_1 = s_{11})
\end{array}$$

2.1.3 Langage de formules $\mathcal{L}_{\bigcirc}$: définition formelle

Maintenant que nous avons présenté informellement les trois relations spéciales de notre logique : $\underset{\otimes}{\rightleftharpoons}$ pour les projections, $\underset{\triangleright}{\rightarrow}$ pour les applications, et $\underset{\square}{\rightarrow}$ pour les expressions sample, nous pouvons les décrire formellement en définissant de manière rigoureuse le langage $\mathcal{L}_{\bigcirc}$ des formules où elles vont apparaître, avant d'établir le schéma définitif des règles de génération.

Définition 2.1.1 (Langage de formules $\mathcal{L}_{\bigcirc}$ sur un contexte de variables) *Fixons une fois pour toutes un ensemble T_0 de types de base, et un ensemble $\mathcal{R}$ de relations et un ensemble $\mathcal{O}$ d'opérations sur ces types de base ainsi que les booléens.*

Soit Γ un ensemble non vide de variables, ou contexte. Alors, l'ensemble TF^Γ des termes de formule et l'ensemble $\mathcal{L}^\Gamma_\bigcirc$ des formules logiques sur le contexte Γ décrivant sont définis par la grammaire suivante :

$$
\begin{array}{lll}
TF^\Gamma ::= & & \\
\quad | \; v & (v \textit{ valeur de base}) & \\
\quad | \; x & (x \in \Gamma) & \textit{Variable} \\
\quad | \; x :: t & (x, t \in TF^\Gamma) & \textit{Consommation du flot} \\
\quad | \; O(x_1, \ldots, x_n) & (O \in \mathcal{O},\ x_1, \ldots, x_n \in TF^\Gamma) & \textit{Opération} \\
\\
\mathcal{L}^\Gamma_\bigcirc ::= & & \\
\quad | \; \top & & \textit{formule vraie} \\
\quad | \; R(x_1, \ldots, x_n) & (R \in \mathcal{R},\ x_1, \ldots, x_n \in TF^\Gamma) & \textit{Relation} \\
\quad | \; a \underset{\otimes}{\overset{c}{\rightleftharpoons}} b & (a, b, c \in TF^\Gamma) & \textit{Couple} \\
\quad | \; a \underset{\triangleright}{\overset{f}{\longrightarrow}} y & (a, f, y \in TF^\Gamma) & \textit{Application} \\
\quad | \; s \underset{\square}{\overset{m}{\longrightarrow}} \langle x, t \rangle & (s, m, x, t \in TF^\Gamma) & \textit{Echantillonnage} \\
\quad | \; s = t & (s, t \in TF^\Gamma) & \textit{Egalité} \\
\quad | \; \neg F & (F \in \mathcal{L}^\Gamma_\bigcirc) & \textit{Opérations logiques} \\
\quad | \; F \wedge G & (F, G \in \mathcal{L}^\Gamma_\bigcirc) & \\
\quad | \; F \vee G & (F, G \in \mathcal{L}^\Gamma_\bigcirc) & \\
\quad | \; F \Rightarrow G & (F, G \in \mathcal{L}^\Gamma_\bigcirc) & \\
\quad | \; F \Leftrightarrow G & (F, G \in \mathcal{L}^\Gamma_\bigcirc) & \\
\quad | \; \exists v : F & (v \notin \Gamma,\ F \in \mathcal{L}^{\Gamma \cup \{v\}}_\bigcirc) & \textit{Quantifications} \\
\quad | \; \forall v : F & (v \notin \Gamma,\ F \in \mathcal{L}^{\Gamma \cup \{v\}}_\bigcirc) &
\end{array}
$$

Comme nous allons le voir dans la description formelle du système de génération des formules : pour le raisonnement avec des formules du langage $\mathcal{L}_\bigcirc$, le raisonnement avec les formules spéciales ne nécessite pas d'autres règles logiques que celles de la logique «classique» des prédicats (logique du second ordre).

2.1.4 Règles de génération des formules : termes (sauf points fixes)

Dans cette sous-section ainsi que les deux suivantes, je présente les règles définitives de génération des formules suivant la syntaxe du $\lambda_\bigcirc$, avec les points fixes, ce qui nécessite de parler de la terminaison, sous la forme des jugements $\Gamma \vdash \{P\}\ m\ \{Q\}$ avec P condition suffisante de terminaison et Q prédicat de correction vérifiée par le résultat sous l'hypothèse que le programme termine.

Attention : la notation utilisée ici, sous forme de règles d'inférence, *ne correspond pas* à une éventuelle sémantique axiomatique, car la génération des formules suivant la sémantique opérationnelle conduit à propager les informations *vers la racine* et non les feuilles.

Les formules pour les termes ne dépendent pas des flots.
Dans cette section, je reprends les règles de génération des formules de correction de la section 2.1.2 mais en y intégrant la terminaison.

true, false ou valeur de base : une telle valeur termine toujours.

$$\frac{v \text{ valeur de base}}{\Gamma \ \vdash \ \{\top\}\ v\ \{\_ = v\}}$$

Variable : lorsque l'on tente d'évaluer une variable, en fait, au niveau où on réduit le terme ou l'expression où elle figure, elle est remplacée par un terme dont l'évaluation termine toujours (ce qui est vrai dans le cas où la variable désigne le point fixe dans son corps, sous les restrictions annoncées plus loin). Donc, elle termine toujours.

$$\frac{x \in \Gamma}{\Gamma \ \vdash \ \{\top\}\ x\ \{\_ = x\}}$$

Test : si le booléen termine, alors on peut effectuer le branchement, et la condition de terminaison varie en fonction de la valeur du booléen.

$$\frac{\Gamma \ \vdash \ \{P_a\}\ a\ \{Q_a(\_)\} \quad \Gamma \ \vdash \ \{P_b\}\ b\ \{Q_b(\_)\} \quad \Gamma \ \vdash \ \{P_c\}\ c\ \{Q_c(\_)\}}{\Gamma \ \vdash \ \left\{ \begin{array}{ll} & P_a \\ \wedge & \forall a' : \\ & P_a \wedge Q_a(a') \ \Rightarrow \\ & (a' = \mathsf{true} \ \Rightarrow \ P_b) \\ \wedge & (a' = \mathsf{false} \ \Rightarrow \ P_c) \end{array} \right\} \ \mathsf{if}\ a\ \mathsf{then}\ b\ \mathsf{else}\ c \ \left\{ \begin{array}{ll} & \exists a' : \\ & Q_a(a') \\ \wedge & (a' = \mathsf{true} \ \Rightarrow \ Q_b(\_)) \\ \wedge & (a' = \mathsf{false} \ \Rightarrow \ Q_c(\_)) \end{array} \right\}}$$

Application : il faut d'abord assurer la terminaison de la fonction et de l'argument. Mais la formule de correction de la fonction contient aussi la condition pour que l'application termine «après la β-réduction», en quelque sorte, condition qui doit être entraînée par la formule de correction de l'argument.

$$\frac{\Gamma \ \vdash \ \{P_a\}\ a\ \{Q_a(\_)\} \quad \Gamma \ \vdash \ \{P_b\}\ b\ \{Q_b(\_)\}}{\Gamma \ \vdash \ \left\{ \begin{array}{ll} & P_a \wedge P_b \\ \wedge & \forall a', \forall b' : \\ & P_a \wedge P_b \wedge Q_a(a') \wedge Q_b(b') \ \Rightarrow \\ & \exists y : \ b' \xrightarrow[\triangleright]{a'} y \end{array} \right\} \ a\ b \ \left\{ \begin{array}{ll} & \exists a', \exists b' : \\ & Q_a(a') \wedge Q_b(b') \\ \wedge & b' \xrightarrow[\triangleright]{a'} \_ \end{array} \right\}}$$

Abstraction : elle-même termine toujours (c'est une valeur). C'est dans la correction que l'on doit insérer la propriété de «terminaison de la fonction une fois appliquée». A α-renommage près, on peut supposer que la variable $x \notin \Gamma$.

$$\frac{\Gamma \cup \{x\} \ \vdash \ \{P_c(x)\}\ c\ \{Q_c(x, \_)\}}{\Gamma \ \vdash \ \{\top\}\ \lambda x.\ c \ \left\{ \begin{array}{ll} & \left(\forall x : \ P_c(x) \ \Rightarrow \ \exists y : \ x \xRightarrow[\triangleright]{} y\right) \\ \wedge & \left(\forall x, \forall y : \ x \xRightarrow[\triangleright]{} y \ \Rightarrow \ Q_c(x, y)\right) \end{array} \right\}}$$

Couple : termine si et seulement si ses deux membres terminent.

$$\frac{\Gamma \vdash \{P_a\}\ a\ \{Q_a(\_)\} \quad \Gamma \vdash \{P_b\}\ b\ \{Q_b(\_)\}}{\Gamma \vdash \{P_a \wedge P_b\}\ (a,b)\ \left\{ \begin{array}{l} \forall a', \forall b' : \\ a' \underset{\otimes}{\rightleftharpoons} b' \ \Rightarrow\ Q_a(a') \wedge Q_b(b') \end{array} \right\}}$$

Projections : terminent si et seulement si le couple dont on veut une projection termine.

$$\frac{\Gamma \vdash \{P_c\}\ c\ \{Q_c(\_)\}}{\Gamma \vdash \{P_c\}\ \mathsf{fst}\ c\ \left\{ \begin{array}{l} \exists c', \exists b : \\ \_ \overset{c'}{\underset{\otimes}{\rightleftharpoons}} b\ \wedge\ Q_c(c') \end{array} \right\}} \qquad \frac{\Gamma \vdash \{P_c\}\ c\ \{Q_c(\_)\}}{\Gamma \vdash \{P_c\}\ \mathsf{snd}\ c\ \left\{ \begin{array}{l} \exists c', \exists a : \\ a \overset{c'}{\underset{\otimes}{\rightleftharpoons}} \_\ \wedge\ Q_c(c') \end{array} \right\}}$$

Terme prob **:** lui-même termine toujours (c'est une valeur). Mais lorsque l'on génère les formules de terminaison et de correction de son expression-corps, celles-ci dépendent des flots d'entrée σ et de sortie σ'. Cette dépendance est «capturée» par la relation $\underset{\square}{\longrightarrow}$.
De plus, comme pour l'abstraction, la terminaison de l'expression e, une fois le prob échantillonné par un sample, apparaît dans la formule de correction du prob.

$$\frac{\Gamma \vdash \{P_e(\sigma)\}\ e\ \{Q_e(\sigma, \_, \sigma')\}}{\Gamma \vdash \{\top\}\ \mathsf{prob}\ e\ \left\{ \begin{array}{ll} & \Big(\forall s :\ P_e(s)\ \Rightarrow\ \exists y, \exists s' :\ s \underset{\square}{\Rightarrow} \langle y, s' \rangle\Big) \\ \wedge & \Big(\forall s, \forall y, \forall s' :\ s \underset{\square}{\Rightarrow} \langle y, s' \rangle\ \Rightarrow\ Q_e(s, y, s')\Big) \end{array} \right\}}$$

2.1.5 Règles de génération des formules : expressions

Cette fois, les formules dépendent des flots d'entrée σ et de sortie σ' de l'expression.

Terme en tant qu'expression : il faut générer les formules du terme, mais aussi exprimer que le flot est inchangé. En ce sens, en toute rigueur, il faudrait distinguer les deux sortes de générations de formules, les termes d'un côté et les expressions de l'autre, comme on fait pour le typage.

$$\frac{\Gamma \vdash \{P_m\}\ m\ \{Q_m(\_)\}}{\Gamma \vdash \{P_m\}\ m\ \{Q_m(\_)\ \wedge\ \sigma' = \sigma\}}$$

Extraction $\mathcal{S}$ d'un élément de flot : elle termine toujours.

$$\overline{\Gamma \vdash \{\top\}\ \mathcal{S}\ \{\sigma = \_ :: \sigma'\}}$$

Echantillonnage sample x from m in e **:** Là, la formule de terminaison est plus compliquée, car il faut s'assurer de trois choses :
- S'assurer que l'évaluation du terme m termine.
- S'assurer, une fois que le terme m est réduit à un prob e_m, que sous le flot d'entrée σ du sample, l'expression e_m termine
- S'assurer, une fois que l'expression e_m donne un flot de sortie t et une valeur v, que l'expression $e[x := v]$ termine sous le flot d'entrée t.

Là encore, à α-renommage près, on peut supposer que $x \notin \Gamma$.

$$\frac{\Gamma \vdash \{P_m\}\ m\ \{Q_m(\_)\} \quad \Gamma \cup \{x\} \vdash \{P_e(x,\sigma)\}\ e\ \{Q_e(x,\sigma,\_,\sigma')\}}{\Gamma \vdash \left\{ \begin{array}{ll} & P_m \\ \wedge & (\forall m' : \\ & P_m \wedge Q_m(m') \Rightarrow \\ & \exists x, \exists t : \ \sigma \xrightarrow[\square]{m'} \langle x, t\rangle) \\ \wedge & (\forall m', \forall x, \forall t : \\ & P_m \wedge Q_m(m') \wedge \sigma \xrightarrow[\square]{m'} \langle x, t\rangle \Rightarrow \\ & P_e(x,t)) \end{array} \right\} \text{sample } x \text{ from } m \text{ in } e \left\{ \begin{array}{ll} & \exists m', \exists x, \exists t : \\ & Q_m(m') \\ \wedge & \sigma \xrightarrow[\square]{m'} \langle x, t\rangle \\ \wedge & Q_e(x,t,\_,\sigma') \end{array} \right\}}$$

2.1.6 Règles de génération des formules : points fixes

La terminaison d'un point fixe est une vaste question. Comme il est naturellement hors de portée de vouloir la traiter en toute généralité, je me suis borné à étudier les deux formes récursives suivantes, qui me semblent les plus courantes dans les algorithmes probabilistes :

– Les points fixes «déterministes» : il s'agit d'algorithmes dont on peut exhiber un *variant déterministe*. Le plus souvent, l'algorithme est une fonction et le variant est son argument récursif. Le cas classique est, bien entendu, celui de la *factorielle*, qui appartient à la classe très connue des programmeurs impératifs des *boucles* `for`. Une telle boucle, qui s'écrirait en gros en impératif :

```
result := init;
for n = 1 to fin
do
  result := corps;
done;
return result;
```

s'écrirait, en $\lambda_{\bigcirc}$, dans le cas où *corps* est un terme de type $\mathsf{nat} \to t \to \bigcirc\ t$ (permettant de calculer une valeur aléatoire en fonction du compteur n et du résultat de l'itération précédente) :

```
sample return from (
  fix boucle : nat → ◯ t. λn : nat. prob
    if n = 0
    then prob init
    else prob
      sample result_prev from boucle (n − 1) in
      sample result from corps n result_prev in
        result
) fin
in return
```

– Les points fixes «probabilistes» : il s'agit d'algorithmes dont le variant est probabiliste. Dans le cadre des simulations de lois de probabilité, on rencontre souvent ce que l'on appelle en langage impératif des *boucles* `repeat...until`. Une telle boucle, qui s'écrirait en gros en impératif :

```
repeat
  result := corps;
until cond;
return result;
```

s'écrirait, en $\lambda_{\bigcirc}$, dans le cas où *corps* est un terme de type $\bigcirc\ t$ (permettant de calculer une valeur aléatoire) et *cond* est un terme de type $t \to \bigcirc$ bool (condition aléatoire, calcul d'un booléen aléatoire en fonction du résultat obtenu) :

```
sample return from
  fix boucle : ◯ t. prob
    sample result from corps in
    sample b from cond result in
    sample return_0 from
      if b
      then prob result
      else boucle
    in return_0
in return
```

Je n'ai donc considéré que ces deux classes de points fixes, et j'ai donc restreint à ces deux formes la définition du terme fix du $\lambda_{\bigcirc}$:

$Term$ ::=	...		termes non récursifs
	$\mid$ fix $x.\ \lambda n.\ c$	$(x, n \in \mathcal{V},\ c \in Term)$	Point fixe déterministe
	$\mid$ fix $x.$ prob e	$(x \in \mathcal{V},\ e \in Expr)$	Point fixe probabiliste

Ainsi, l'évaluation du point fixe en lui-même termine toujours, car elle consiste en un seul dépliage : $\lambda n.\ c[x := \text{fix}\dots]$ ou prob $e[x := \text{fix}\dots]$, lesquels termes obtenus après ce dépliage sont des valeurs.
En outre, cela justifie, sous ces restrictions, que «l'évaluation d'une variable termine toujours», car la variable x est remplacée par le terme point fixe, dont on vient de montrer qu'il termine.

Formules pour un point fixe déterministe. L'approche choisie est tout à fait classique : on a besoin de spécifier un *variant* et un *ordre bien fondé* selon lequel le variant décroît strictement. *On convient que le variant est l'argument n dans* fix $x.\ \lambda n.\ c$. En effet, on peut toujours se ramener à ce cas :
– Si le variant est constitué de plusieurs arguments (par exemple, avec l'ordre lexicographique), on «décurryfie» la fonction pour qu'elle ne prenne qu'un seul argument de type couple.
– Si le variant est constitué du résultat d'une opération F fixée sur l'argument n, il suffit de modifier la définition de l'ordre pour y «intégrer» l'opération F.

L'ordre est alors indiqué en décorant le terme.

Dans le cadre de la génération des formules : comme on a vu que le point fixe en lui-même termine toujours, la condition de terminaison du «point fixe une fois appliqué» apparaît à l'intérieur de la formule de correction du point fixe. Etant donné un n, pour que l'application termine au point n, il suffit qu'elle termine pour les éléments plus petits que n et que la condition de terminaison du corps soit vérifiée pour n. Charge alors à l'utilisateur de prouver la terminaison de l'application pour n en exhibant la preuve de bonne fondation de l'ordre sur le variant.
Cette formule de terminaison est alors simplement conjointe à la formule de correction de l'abstraction comme s'il ne s'agissait pas d'un point fixe (étant donné que cette formule *suppose* la terminaison de l'application).

$$\frac{\Gamma \cup \{x, n\} \ \vdash \ \{P_c(x,n)\}\ c\ \{Q_c(x,n,\_)\}}{\Gamma \ \vdash \ \{\top\}\ \mathsf{fix}\ x.\ \lambda_{\prec} n.\ c \left\{ \begin{array}{l} \left(\forall n : \begin{array}{l} P_c(\_,n) \ \wedge \ \left(\forall m : \ m \prec n \ \Rightarrow \ \exists z : \ m \underset{\triangleright}{\rightrightarrows} z \right) \\ \Rightarrow \ \exists y : \ n \underset{\triangleright}{\rightrightarrows} y \end{array} \right) \\ \wedge \\ \left(\forall n, \forall y : \ n \underset{\triangleright}{\rightrightarrows} y \ \Rightarrow \ Q_c(\_,n,y) \right) \end{array} \right\}$$

où $\prec$ est l'ordre sur le variant, supposé bien fondé.

Formules pour un point fixe probabiliste. L'idée fondamentale est de dire qu'un point fixe probabiliste est simplement un point fixe dont *le variant est le flot de données aléatoires lui-même.*
Cependant, ce n'est pas suffisant. En effet, nombreux sont les algorithmes probabilistes qui ne terminent pas de façon déterministe. Un des plus simples est la simulation suivante de la distribution géométrique de paramètre p :

```
fix boucle : ◯ nat. prob
  sample x from prob S in
  sample return from
    if x ⩽ p
    then prob 0
    else prob sample y from boucle in y + 1
  in return
```

Un tel algorithme peut boucler indéfiniment si le flot d'entrée est par exemple constitué uniquement de valeurs plus grandes que p.
Il faut donc que l'utilisateur spécifie un ensemble sur lequel il sait que l'algorithme termine, c'est-à-dire tel que l'appartenance du flot d'entrée à cet ensemble soit une condition suffisante de terminaison, cette condition étant déterministe. Par conséquent, l'utilisateur pourra montrer plus tard que l'algorithme termine avec probabilité plus grande que la mesure de l'ensemble indiqué. Pour montrer la terminaison avec probabilité 1, il suffit donc d'exhiber un tel ensemble de mesure 1.
Supposant que le flot d'entrée est dans cet ensemble, pour montrer la terminaison de l'algorithme, l'utilisateur fera comme s'il s'agissait d'un point fixe déterministe : indiquer un ordre bien fondé sur les flots appartenant à cet ensemble.
Aussi les formules générées pour un tel point fixe sont-elles très voisines de celles générées pour un point fixe déterministe.

$$\frac{\Gamma \cup \{x\} \ \vdash \ \{P_c(x,\sigma)\}\ c\ \{Q_c(x,\sigma,\_,\sigma')\}}{\Gamma \ \vdash \ \{\top\}\ \mathsf{fix}\ x.\ \mathsf{prob}_{S,\prec}\ c\ \left\{ \begin{array}{l} \left(\begin{array}{ll} & P_c(\_,s) \ \wedge \ s \in S \ \wedge \\ \forall s: & \left(\forall t: \ t \in S \ \wedge \ t \prec s \ \Rightarrow \ \exists z,t': \ t \underset{\square}{\Rightarrow} \langle z,t'\rangle\right) \\ \Rightarrow & \exists y, \exists s': \ s \underset{\square}{\Rightarrow} \langle y,s'\rangle \end{array} \right) \\ \wedge \\ \left(\forall s, \forall y, \forall s': \ s \underset{\square}{\Rightarrow} \langle y,s'\rangle \ \Rightarrow \ Q_c(\_,s,y,s')\right) \end{array} \right\}}$$

où S est l'«ensemble suffisant» pour la terminaison de l'algorithme, et $\prec$ l'ordre bien fondé sur cet ensemble.

2.1.7 Mise en œuvre du transformateur Π[5]

Nous venons de présenter un système de génération de deux formules, l'une de terminaison, l'autre de correction, pour chaque élément du langage $\lambda_{\bigcirc}$. Nous obtenons donc en sortie provisoire, pour un programme m passé en entrée de Π, un jugement de la forme :

$$\Gamma \ \vdash \ \{P\}\ m\ \{Q(\_)\}$$

Mais sous cette forme, le jugement généré n'est pas encore exploitable dans un système de preuve, pas plus que ne l'est le couple (P,Q) des formules obtenues.

Il y a donc une dernière étape à accomplir, au niveau *global* cette fois : réunir ces deux formules en une seule, *que l'on pourra utiliser pour raisonner en Coq.*
Rappelons qu'un tel jugement se lit : «si P est vérifiée, alors le calcul de e termine et Q est vérifiée». C'est suivant cette lecture que le transformateur génère la formule finale globale à partir des deux formules de terminaison P et de correction Q.

On rappelle que le programme m est un *terme* ; on peut considérer que ce terme va être stocké dans une variable «globale». Sous le contexte Γ des variables globales déclarées avant le terme m, le transformateur Π fonctionne alors, en supposant que le terme soit stocké dans la variable v_m, de la manière suivante :

1. Π génère les formules P et $Q(\_)$ telles que
$$\Gamma \ \vdash \{P\}\ m\ \{Q(\_)\}$$
2. et alors la formule *globale* obtenue pour le programme est $P \ \Rightarrow \ Q(v_m)$.

En fait, la variable v_m existe toujours en Coq, mais la terminaison est quand même «comprise» dans cette formule en ce sens que si P n'est pas vérifiée, alors *le comportement du programme est indéterminé.* Cependant, on ne peut pas dire *a priori* que le programme ne termine pas, car P, on le rappelle, est une *condition suffisante* de terminaison.
Toutefois, en pratique, le terme m est une abstraction ou un prob : c'est donc une valeur, qui «termine toujours» au sens où $P \equiv \top$, donc le problème ne se pose pas.

[5]Pour des informations d'ordre pratique (comment utiliser effectivement Π sur une machine), on se réfèrera à l'annexe, page 53.

2.2 Formalisation en Coq

J'ai mis en œuvre dans le système Coq la représentation des données aléatoires par une formalisation sous deux aspects :
- l'axiomatique des espaces mesurables et des probabilités en général (fichier `proba.v`)
- l'application aux flots de données aléatoires, à partir d'un espace élémentaire donné en paramètre (à cet égard, le système de modules de Coq est d'utilisation très commode), et le langage de formules $\mathcal{L}_{\bigcirc}$ (fichier `lo_axiom.v`)

Ces formalisations ont donc fait l'objet de deux fichiers Coq. Mais j'ai dû regrouper dans un seul fichier les modules sur les flots et ceux sur le langage de formules $\mathcal{L}_{\bigcirc}$, car ils sont intimement liés. Voici comment ce fichier est structuré :

- Signature `PROBA_FLOTS_PARAM` d'un module de paramètres : pour spécifier l'espace des éléments de flot (et au passage le type de données τ_0 des éléments de flot).
- Foncteur `Principal` prenant en entrée un tel module de paramètres.
  - Définition d'un flot (pas de sous-module)
  - Sous-module `Lambda_o` du langage de formules $\mathcal{L}_{\bigcirc}$ (qui se rapporte à $\lambda_{\bigcirc}$, d'où le nom du module).
  - Sous-module `Proba_flots` de probabilités sur les flots.
  - Sous-module `Proba_locale` : probabilités locales sur les flots (cf. 2.3)

Et alors, à partir de ces modules de base, je dégage une méthodologie particulière pour structurer la mécanisation d'un raisonnement probabiliste en Coq.

Pour écrire mes modules Coq, je tire parti des librairies standard sur les réels, les entiers naturels, les ordres bien fondés, les ensembles, etc. Tous mes modules Coq sont disponibles sur le Web (voir annexe page 53).

2.2.1 Axiomatisation des probabilités en Coq

J'ai d'abord formalisé en Coq, dans le fichier `proba.v`, la théorie des probabilités sur un ensemble quelconque.
Cet «ensemble» de départ sur lequel je greffe des structures de probabilités est en fait un objet U de sorte `Type` en Coq. En effet, j'ai besoin d'être aussi général que possible sur l'ensemble de base.
Pourtant, lorsque j'ai besoin de parler des parties de cet ensemble, ainsi que de toute autre structure ensembliste par-dessus U, je me sers du module `Ensembles` de la librairie standard, qui définit les ensembles à partir des prédicats vérifiés par leurs éléments, prédicats de type $U \to$ `Prop` en Coq ; pour montrer l'égalité de deux ensembles, il est suffisant, grâce à l'axiome d'extensionnalité, de montrer que leurs prédicats caractéristiques sont équivalents.

Pour exprimer les définitions, axiomes et théorèmes de la théorie des probabilités, j'ai suivi l'approche de [Hur01]. En revanche, j'ai choisi de ne pas changer de terminologie entre la théorie des mesures et celle des probabilités. Tout au long de mon stage, je continue de parler de partie mesurable au lieu d'événement, etc. Certes, il aurait été intéressant, dans l'absolu, de différencier les deux terminologies afin de pouvoir savoir, dans une preuve, de quelle théorie l'on a réellement besoin. Cependant, en pratique, tous les raisonnements que j'ai faits ont nécessité

un dépliage des définitions, de sorte qu'un renommage aurait entraîné une étape de dépliage inutile.
J'obtiens donc correctement, et moyennant quelques gros théorèmes posés en axiomes[6], la définition d'une algèbre, et d'un espace mesuré[7].

Remarque : pour la définition de la tribu (ou σ-algèbre) engendrée par une famille de parties, je propose deux définitions Coq : l'une par formule ensembliste (`sigma` : l'intersection de toutes les tribus contenant cette famille), l'autre par induction Coq (`sigma2` : la plus petite tribu contenant cette famille). Je montre ensuite que ces deux définitions donnent les mêmes objets.

```
Inductive sigma2 (U : Type) (G : Ensemble (Ensemble U)) :
Ensemble (Ensemble U) :=

| sigma_0 : forall (e : Ensemble U),
In (Ensemble U) G e -> In (Ensemble U) (sigma2 U G) e

| sigma_vide : In (Ensemble U) (sigma2 U G) (Empty_set U)

| sigma_complement : forall (e : Ensemble U),
In (Ensemble U) (sigma2 U G) e ->
In (Ensemble U) (sigma2 U G) (Complement U e)

| sigma_union : forall (e1 e2 : Ensemble U),
In (Ensemble U) (sigma2 U G) e1 -> In (Ensemble U) (sigma2 U G) e2 ->
In (Ensemble U) (sigma2 U G) (Union U e1 e2)

| sigma_union_denomb : forall (A : Familles_parties.Famille_parties U nat),
(forall n :nat, In (Ensemble U) (sigma2 U G) (A n)) ->
In (Ensemble U) (sigma2 U G) (Familles_parties.Union U nat A).


Definition sigma (U :Type) (G : Ensemble (Ensemble U)) :
Ensemble (Ensemble U) :=
let e := Ensemble U in let E := Ensemble e in
let F : Ensemble E := fun f :E => (Sigma_algebre U f /\ Included e G f) in
Operations_ensembles.Intersection e F.

Theorem sigma_est_sigma2 : forall (U : Type) (G : Ensemble (Ensemble U)),
sigma2 U G = sigma U G.
```

Par exemple, je définis l'espace des boréliens, en posant comme axiome l'existence de la mesure de Lebesgue :

[6]Par exemple le théorème de Carathéodory, dont la formalisation en HOL par Hurd [Hur01] a nécessité deux semaines.

[7]Dans les sources, je parle à tort d'«espace mesurable» pour désigner un espace mesuré.

```
Definition Boreliens (E : Ensemble R) : Ensemble (Ensemble R) :=
sigma R (fun A => exists u :  R, exists v :  R,
Intersection R E A = Intervalle_ferme u v).

Axiom Existe_Lebesgue :
forall E : Ensemble R, exists lambda : Ensemble R -> R,
Espace_mesurable R (Boreliens E) lambda /\
forall u v : R,
u <= v -> Included R (Intervalle_ferme u v) E ->
lambda (Intervalle_ferme u v) = v - u.
```

Remarque : pour construire la tribu borélienne sur un intervalle[8] I (noté `E` dans le code Coq), je ne me restreins pas aux parties de I. Je considère *toutes* les parties de $\mathbb{R}$, et simplement en termes de mesurabilité et de mesure, je ne m'intéresse qu'à l'intersection avec I des parties.
On pourra obtenir des résultats farfelus comme celui-ci : si Z est une partie de $\mathbb{R}$ non mesurable au sens de l'espace des boréliens sur tout $\mathbb{R}$ (on sait qu'il existe un tel Z, voir par exemple [Hur01]), alors, pour tout intervalle I, $Z \backslash I$ n'est pas mesurable au sens des boréliens sur $\mathbb{R}$, mais l'est au sens des boréliens sur I, car, intersecté avec I, c'est la partie vide !

2.2.2 Les flots

Je considère comme un paramètre du développement l'ensemble E des éléments de flot : c'est une clause `Parameter` de la signature `PROBA_FLOTS_PARAM`, qui permet en fait, grâce à la structure de foncteur `Principal` qui prend en entrée un module de cette signature, de créer des modules différents en fonction du type choisi pour les éléments de flot.
Je définis les flots sous la forme la plus commune en informatique : la forme co-inductive, au lieu d'adopter l'approche mathématique d'une famille indicée par $\mathbb{N}$. Ceci me permet de considérer le *conse* :: de manière structurelle, et non plus comme un alias. De plus, ceci permet de mieux modéliser la consommation d'un élément de flot.
Pour utiliser la clause `CoInductive`, afin de définir mon type des flots dépendant du paramètre de type E, il a fallu que ce paramètre, que j'ai nommé `Element_flot` en Coq, soit de sorte `Set` en Coq, et non de sorte `Type`.

Notons que [PPT05] choisit d'emblée les réels pour l'ensemble E des éléments de flot, tandis que [Hur01] choisit les booléens. En considérant E comme un paramètre, je permets à l'utilisateur de se prononcer le plus tard possible, et je montre des propriétés très générales sur les flots et les probabilités sur les flots indépendamment de la nature de leurs éléments.

2.2.3 Langage de formules $\mathcal{L}_{\bigcirc}$

Comme on l'a vu, je n'ai pas voulu mapper les structures du $\lambda_{\bigcirc}$ vers les structures du Calcul des Constructions Inductives (CIC)[BC04] de Coq, afin de ne pas utiliser les règles de réduction du CIC sur les structures du langage $\lambda_{\bigcirc}$.
Par conséquent, j'ai dû créer des types Coq spéciaux pour modéliser les types flèche et couple du $\lambda_{\bigcirc}$, au lieu d'utiliser les types fournis par Coq. (Par contre, pour les types de base, pas question de réinventer la roue.) Ces types sont simplement définis en Coq par des clauses

[8] Ces définitions me permettent même de construire un espace des boréliens sur une partie autre qu'un intervalle.

`Variable` ; j'introduis ensuite, pour faciliter la lecture, des notations plus proches de celles de la grammaire.

Les relations spéciales $\underset{\otimes}{\rightleftharpoons}$ pour les projections $\mathsf{fst}, \mathsf{snd}$, $\underset{\triangleright}{\rightarrow}$ pour les applications et $\underset{\square}{\rightarrow}$ pour les expressions sample sont simplement définies par des clauses `Variable` et portent le nom, respectivement, de `Relation_croix`, `Relation_triangle`, `Relation_carre`. Puis, je définis des notations afin, là encore, de me rapprocher de la grammaire.

A côté de ces relations, je définis un «raccourci» pour les formules permettant leur élimination : les formules de correction pour les projections, l'abstraction et le terme prob.

	Notation	**Notation Coq**	**Signification**
Couple	$c \otimes P$	`Croix c P`	$\forall a, \forall b : \ a \overset{c}{\underset{\otimes}{\rightleftharpoons}} b \ \Rightarrow P(a,b)$
Abstraction	$f \triangleright P$	`Triangle f P`	$\forall x, \forall y : \ x \overset{f}{\underset{\triangleright}{\rightarrow}} y \ \Rightarrow \ P(x,y)$
Terme prob	$m \square P$	`Carre m P`	$\forall s, \forall y, \forall s' : s \overset{m}{\underset{\triangleright}{\rightarrow}} \langle x, s' \rangle \ \Rightarrow \ P(s,y,s')$

En réalité, j'ai commencé mon stage en introduisant ces formules *avant même* de parler des relations spéciales. Ceci explique l'origine des symboles $\otimes, \triangleright, \square$ sous les flèches.
Quant à l'origine de ces symboles même pour les formules de correction ci-dessus, on pourrait l'expliquer par leur similitude avec les symboles des types correspondants : $\otimes$ pour le type couple $\times$, mais aussi $\triangleright$ pour le type flèche $\rightarrow$, et enfin $\square$ pour le type $\bigcirc$. Pour ce dernier symbole, le lien avec le symbole du type est difficile à voir. Mais il suffit de se dire que dans les trois cas, on arrive au symbole de formule par une *déformation* du symbole du type correspondant. Cela illustre à quel point sont similaires le typage et la génération des formules.

Par la suite, il a fallu rajouter des formules de terminaison : par exemple, pour la formule de terminaison de l'application d'une abstraction (qui est une partie de la formule de correction de l'abstraction), j'introduis, pour une abstraction f et un prédicat $P(x)$ portant sur l'argument x de l'abstraction, le terme Coq `Terminaison_triangle f P`, correspondant à la formule «si $P(x)$ est vérifiée, alors l'application $f(x)$ termine» :

$$\forall x : \ P(x) \ \Rightarrow \ \exists y : \ x \overset{f}{\underset{\triangleright}{\rightarrow}} y$$

J'ai défini de même les termes Coq `Terminaison_croix` et `Terminaison_carre`. Enfin, je combine terminaison et correction en introduisant des termes Coq tels que `Double_carre m P Q` correspondant à `Terminaison_carre m P /\ Carre m Q`.

Toutes ces définitions servent à rendre la formule relativement plus lisible. De plus, comme les formules générées par Π sont, pour la plupart, sous cette forme, un éventuel changement de leur implémentation sous Coq n'affecterait pas la validité des formules générées par le transformateur Π, sous réserve que l'implémentation sous Coq de ces définitions soit elle-même valide.

Cependant, pour pouvoir modéliser correctement le comportement des programmes en $\lambda_{\bigcirc}$, il a fallu introduire des axiomes cruciaux :

– des axiomes de fonctionnalité, exprimant l'unicité sous réserve d'existence, du résultat d'une fonction, ou d'un échantillonnage *à flot fixé* :

$$\forall c, \forall a_1, \forall a_2, \forall b_1, \forall b_2 :\ a_1 \underset{\otimes}{\overset{c}{\rightleftharpoons}} b_1 \ \wedge\ a_2 \underset{\otimes}{\overset{c}{\rightleftharpoons}} b_2 \ \Rightarrow\ a_1 = a_2 \ \wedge\ b_1 = b_2$$

$$\forall f, \forall x, \forall y_1, \forall y_2 :\ x \underset{\triangleright}{\overset{f}{\longrightarrow}} y_1 \ \wedge\ x \underset{\triangleright}{\overset{f}{\longrightarrow}} y_2 \ \Rightarrow\ y_1 = y_2$$

$$\forall m, \forall s, \forall y_1, \forall y_2, \forall s'_1, \forall s'_2 :\ s \underset{\square}{\overset{m}{\longrightarrow}} \langle y_1, s'_1 \rangle \ \wedge\ s \underset{\square}{\overset{m}{\longrightarrow}} \langle y_2, s'_2 \rangle \ \Rightarrow\ y_1 = y_2 \ \wedge\ s'_1 = s'_2$$

– un axiome énonçant que le flot de sortie d'un échantillonnage est une queue du flot d'entrée.

2.2.4 Probabilités sur les flots

Pour la définition de l'espace des flots, j'ai d'abord adapté la définition suivante de [Hur01], et je l'ai mise en œuvre en Coq. J'ai ensuite appliqué la théorie des probabilités à cet espace.

Définition 2.2.1 (Espace des flots) *Soit E un ensemble non vide, l'ensemble des éléments de flot. Supposons que l'on dispose d'un espace de probabilités $(\mathcal{E}, \mu)$ sur E, l'espace des éléments de flot. Alors, on définit l'espace des flots sur $\mathcal{E}$ comme suit :*
– *La tribu $\mathcal{E}^\infty$ des flots sur $\mathcal{E}$ est la σ-algèbre engendrée par les rectangles de flots (c'est-à-dire les flots dont un nombre fini d'extractions est dans un produit fini d'espaces $\mathcal{E}$-mesurables).*

$$\mathcal{E}^\infty = \sigma\left\{\left\{\varphi \in E^{\mathbb{N}} :\ \forall n < N, \varphi_n \in E_n\right\};\ N \in \mathbb{N},\ E_1, \ldots, E_N \in \mathcal{E}\right\}$$

– *La mesure μ^∞ des flots sur $\mathcal{E}$ selon la mesure μ est la mesure qui vaut, pour un rectangle formé par N ensembles $E_1, \ldots, E_n$ $\mathcal{E}$-mesurables, le produit des mesures des E_i au sens de la mesure μ sur les ensembles d'éléments de flot.*

$$\forall N \in \mathbb{N}, \forall E_1, \ldots, E_n \in \mathcal{E} :\ \mu^\infty\left\{\varphi \in E^{\mathbb{N}} :\ \forall n < N, \varphi_n \in E_n\right\} = \prod_i E_i$$

L'existence d'une telle mesure se démontre comme pour la mesure de Lebesgue ; pour ma part, je l'ai posée en axiome.
L'axiome de la mesure d'un rectangle exprime la propriété que deux extractions distinctes sont indépendantes.

2.3 Probabilités sur des événements de programme

Reprenons l'exemple de la distribution de Bernoulli.

$$\lambda p.\ \mathsf{prob\ sample}\ x\ \mathsf{from\ prob}\ \mathcal{S}\ \mathsf{in}\ x \leqslant p$$

Comment énoncer formellement qu'une fois ce terme appliqué sous le paramètre p, la probabilité que j'aie un échantillonnage me retournant le résultat true est p ?
Il y a deux façons de le faire :

– soit écrire formellement la mesure de l'ensemble des flots σ vérifiant que l'échantillonnage du terme prob sous le flot d'entrée σ donne true :

$$\mu^\infty \left\{ \sigma \in E^{\mathbb{N}} : \ \exists y, \exists \sigma' : \ \sigma \xrightarrow[\square]{} \langle y, \sigma' \rangle \ \wedge \ y = \mathsf{true} \right\} = p$$

Pour raisonner avec cette probabilité «globale», on procède en deux étapes :

1. D'abord, on se ramène à un mesurable «connu», c'est-à-dire que l'on montre que l'ensemble à mesurer s'exprime comme un ensemble mesurable de l'espace des flots.
2. Ensuite, on calcule la mesure de cet ensemble à partir de l'axiome sur la mesure μ^∞.

– soit parler de ce que j'ai appelé *probabilité locale* sur le flot s. Je n'ai pas formalisé mathématiquement ce concept, mais je l'ai introduit en tant qu'outil de raisonnement en Coq. On écrira en Coq :

```
forall s, Carre s (fun y t => Proba_locale s (y = true) = p)
```

qui exprime que si s est passé en flot d'entrée, alors la probabilité qu'il donne un échantillon $y = \mathsf{true}$ est p.
Cette formule exprime en fait une probabilité sur un objet *déjà défini* et dont on considère qu'il est distribué comme un flot d'entrée de tout le programme. Mais on doit alors supposer que l'on ne dispose d'*aucune autre propriété* sur le flot. A cet égard, le manque de formalisme autour de cette notion fait qu'il faut la manipuler avec circonspection.

Toutefois, la probabilité locale est surtout utile lorsque l'on raisonne avec des distributions de probabilité auxiliaires. Par exemple, beaucoup d'algorithmes de simulation ont besoin de simuler localement la loi de Bernoulli (comme la loi binomiale, fig. 6 p. 44). A cet égard, la probabilité locale permet d'exprimer localement que tel fragment de code $\lambda_\bigcirc$ simule la loi de Bernoulli .
En réalité, on peut dire que la probabilité locale sur un flot s exprime une probabilité sur le flot d'entrée σ de tout le programme, *conditionnée* par les consommations qui ont eu lieu sur σ pour obtenir s.

De plus, on constate qu'aucune mention n'est faite des opérations sur les flots. Mais pour des algorithmes à récursivité probabiliste comme la simulation de la loi géométrique, il faut pouvoir raisonner ainsi, sur les flots eux-mêmes, donc cette approche est inadaptée pour ce genre de programme.
En outre, on aura remarqué que le terme `Proba_locale` est «sous» le carré : donc, cela suppose l'existence du résultat y et du flot de sortie s'. C'est dire que la probabilité locale ne peut être utilisée que dans un raisonnement de correction partielle, en supposant la terminaison.

Pour raisonner avec les probabilités locales, j'ai introduit deux axiomes :

Axiome 2.3.1 (Probabilité locale des flots équivalents) *Sous le même flot d'entrée s, deux propriétés équivalentes ont même probabilité locale.*

Axiome 2.3.2 (Probabilité locale et globale) *Pour un prédicat $\psi(s)$, la probabilité locale sur le flot s que s vérifie $\psi(s)$ est la mesure $\mu^\infty \{s : \psi(s)\}$*

On constate qu'ils constituent des étapes précédant celles du calcul de probabilité «globale» avec μ^∞ : dans un premier temps, on utilise le premier axiome pour réécrire la propriété dont on veut calculer la probabilité sous la forme $\psi(s)$, puis grâce au deuxième axiome, on se ramène au premier cas (global).

Remarque : lorsque je raisonne sur un programme en $\lambda_\bigcirc$, je n'ai jamais besoin de considérer des tribus sur autre chose que les flots ou les éléments de flot. En effet, pour calculer une probabilité, je me ramène toujours à un ensemble de flots à mesurer.

2.4 Structure d'une preuve sur un algorithme probabiliste

Pour prouver en Coq une propriété sur un programme en $\lambda_\bigcirc$, il est nécessaire de suivre une discipline toute particulière en matière de structure du module de preuve, que j'ai mise en place au travers des exemples de preuves formelles que j'ai menés à bien, surtout pour des algorithmes récursifs comme pour la loi binomiale ou la loi géométrique.

Le transformateur Π produit un fichier Coq contenant
– la déclaration des variables libres du programme
– la formule Π correspondant au programme, sous la forme d'une hypothèse (clause `Hypothesis`)
sous la forme d'une section Coq comme sur la figure 2 page 37.

Ce fichier doit être complété en rajoutant les *hypothèses sur les variables libres* du programme, évidemment, car Π ne sait rien *a priori* sur les variables libres, qui en fait représentent simplement l'utilisation d'algorithmes auxiliaires.
D'ailleurs, cela permet d'avoir une certaine forme de *compositionnalité* du raisonnement de ce point de vue-là : dans ce genre de situation, on n'indique que les propriétés dont on a besoin sur ces algorithmes auxiliaires.
Mais ces hypothèses ne sont pas nécessairement des formules du langage $\mathcal{L}_\bigcirc$ telles que les produirait le transformateur Π. Il peut très bien s'agir de formules probabilistes, avec la mesure μ^∞, ou même, et c'est ici qu'il est intéressant d'en reparler, la probabilité locale, car elle permet par exemple de parler de l'algorithme auxiliaire uniquement de manière qualitative, comme une «spécification», sans parler de la consommation du flot par exemple. C'est l'approche souvent choisie lorsque l'on utilise de façon auxiliaire un algorithme simulant la loi de Bernoulli, par exemple pour la loi binomiale (voir figure 6, page 44).

Une fois indiquées toutes les hypothèses, que ce soit les hypothèses sur les algorithmes auxiliaires ou la formule Π du programme, on peut alors énoncer et prouver la ou les propriétés voulues sur le programme. Ces propriétés peuvent traiter de la terminaison seulement, de la correction sous hypothèse de terminaison seulement, ou des deux à la fois.
En tous cas, il faut, autant que possible, *séparer formules de terminaison et formules de correction* et commencer par prouver les formules de *terminaison*. En effet, certaines preuves de formules de correction exigent que la terminaison ait été prouvée.

On aboutira donc à la structure suivante pour un module de preuve :

1. Variables libres (données par Π)
2. Hypothèses sur les variables libres

3. Variable du programme (donnée par Π), représentant l'algorithme sur lequel porteront les preuves
4. Hypothèse sur le programme : formule donnée par le transformateur Π
5. Preuves de terminaison : prouver des conditions suffisantes déterministes pour qu'un algorithme termine (on ne calculera pas ici la «probabilité qu'un algorithme termine»), ou alors ramener ces conditions à des ensembles mesurables de flots (réécrire cet événement de programme sous la forme d'une propriété sur les flots d'entrée définissant un ensemble mesurable)
6. Preuves de correction :
   – si l'on veut prouver des conditions déterministes sur les paramètres, ou si l'on raisonne avec les probabilités locales, on peut raisonner directement
   – si l'on raisonne sur des conditions probabilistes avec la probabilité «globale», ramener les propriétés de correction à des ensembles mesurables de flots
7. Preuves de mesurabilité des ensembles à mesurer
8. Mesures de ces ensembles

Ainsi, on effectue deux séparations :
– séparer terminaison et correction
– séparer mesurabilité et probabilité
Ces séparations rendent les preuves plus claires et permettent également de réutiliser des fragments de preuves concernant terminaison ou correction, dans d'autres preuves sur d'autres programmes.
En outre, elles permettent de rendre *indépendants* les preuves sur les programmes, et des preuves que l'on pourra considérer comme «annexes» (des lemmes, en quelque sorte), sur des structures «intermédiaires» dont les propriétés ne parlent plus du comportement du programme.
De plus, de telles structures peuvent également apparaître dans d'autres occasions, et à ce titre, elles pourront être définies et traitées dans d'autres fichiers Coq que l'on chargera dynamiquement. Par exemple, on trouvera dans le fichier `lo_axiom.v` une section `Ensf_ord` qui définit de manière générale un ensemble et un ordre bien fondé sur les flots ; j'ai rajouté cette structure au moment où j'ai voulu prouver la correction d'un algorithme de simulation de la loi géométrique (cf. 3.3 pour plus de détails), mais je l'ai inclus dans le fichier `lo_axiom.v` car je pense que cette structure générale peut être utilisée dans d'autres contextes.

2.5 Technique : comment se débrouiller avec les formules Π dans Coq

Les formules générées par le transformateur Π sont souvent très longues : ne serait-ce que pour la simulation de la loi de Bernoulli, on se reportera à la figure 2, page 37, pour s'en convaincre ! Aussi m'a-t-il paru nécessaire d'inclure dans ce rapport quelques éléments techniques permettant l'utilisation des modules Coq que j'ai écrits pour prouver des propriétés sur d'autres algorithmes probabilistes, et décrivant en particulier l'usage des *tactiques* qui a été mien pour mes exemples de preuves.
Toutefois, cette section rend compte de l'incomplétude de mon travail en matière d'*automatisation* de la preuve : tous les éléments que je vais donner ici pourraient donner lieu à l'écriture de tactiques en $\mathcal{L}_{tac}$, le langage des tactiques de Coq (cf. [BC04], chap. 7, «Tactics and Automation»).

Tout d'abord, je déplie toutes les définitions des termes Coq du triangle, carré, croix, etc. en utilisant la tactique `generalize` sur toutes les hypothèses fournies (ce qui me les injecte dans

le but à prouver), puis en effectuant `unfold` sur le but pour chaque terme, et enfin `intros` pour réintroduire les hypothèses dépliées.

Les formules Π contiennent beaucoup de conjonctions et de quantifications existentielles (toutefois souvent cachées sous des universels). J'ai écrit une tactique très utile, `casser_les_cailloux`, permettant :
– de remplacer l'hypothèse `P /\ Q` par les hypothèses `P` et `Q`
– de remplacer l'hypothèse `exists x, P` par les hypothèses `x` et `P`

Le gros du travail de manipulation a lieu lors des «réductions» : dès lors que j'ai deux hypothèses des formes suivantes (correspondant à la relation $s \xrightarrow[\square]{m} \langle y, t \rangle$) :
– `H1 : forall s y t, { s >> m ~ y >> t } -> P`
– `H2 : { s >> m ~ y >> t }`
Je peux alors obtenir `P` en exécutant la tactique :

```
generalize (H1 _ _ _ H2); intros
```

De même, dès lors que j'ai deux hypothèses des formes suivantes (correspondant à la relation $x \xrightarrow[\triangleright]{f} y$) :
– `H1 : forall x y, x -- f --> y -> P`
– `H2 : x -- f --> y`
Je peux alors obtenir `P` en exécutant la tactique :

```
generalize (H1 _ _ H2); intros
```

En combinant toutes ces tactiques, je peux en fait «exécuter le programme pas à pas» en décrivant en logique les différentes étapes d'exécution.

Il serait intéressant d'automatiser ces tactiques avec $\mathcal{L}_{tac}$: en effet, dans mes raisonnements, je perds beaucoup de temps à chercher les numéros des hypothèses correspondantes, d'une part parce qu'elles sont très nombreuses et très complexes, d'autre part parce qu'une modification de la preuve en amont «décale» complètement toute la numérotation.
Mais il faut faire attention aux points fixes : une tactique d'automatisation mal écrite peut faire boucler le raisonnement !

3 Application : distributions simples

La formalisation que j'ai réalisée en Coq de la théorie des probabilités ainsi que du langage de formules $\mathcal{L}_{\bigcirc}$ m'a permis de traiter des exemples de lois simples :
- des distributions «de base» qui se simulent par des algorithmes non récursifs, comme la loi de Bernoulli ou la loi exponentielle
- la loi binomiale, se simulant par un algorithme à récursivité déterministe
- la loi géométrique, par un algorithme à récursivité probabiliste

Pour traiter ces simulations, je me suis placé dans le cadre de [PPT05] : des flots de réels uniformément distribués sur $[0, 1]$. On supposera donc que l'on dispose des réels, ainsi que des entiers naturels (quasi incontournables en Coq), avec leurs opérations arithmétiques et relations élémentaires.

Remarque : j'ai préféré traiter les boréliens sur $[0, 1]$ au lieu de $]0, 1]$ car il m'a semblé plus simple de raisonner avec des inégalités larges. Toutefois, nous verrons que cela m'a posé quelques problèmes pour la loi exponentielle.

Tous les exemples cités ici sont disponibles sur mon site Web, sous-dossier `coq` (pour des informations sur le téléchargement, voir annexe page 53).

3.1 Simulations par algorithmes non récursifs

3.1.1 Loi de Bernoulli

La loi de Bernoulli est une des lois de base en probabilités, et même en statistiques. C'est même une des lois fondamentalement utilisées, et peut-être même la brique de base, pour la représentation des lois discrètes (ce qui n'a pas échappé à [Hur01], en considérant des flots de booléens suivant la loi de Bernoulli de paramètre $1/2$).
Du point de vue de la simulation, l'algorithme est très simple, et ce sont ces deux arguments, importance et simplicité, qui m'ont conduit à citer cet exemple en de nombreux points de mon rapport. En $\lambda_{\bigcirc}$, on écrit donc :

$$\lambda p.\ \mathsf{prob\ sample}\ x\ \mathsf{from\ prob}\ \mathcal{S}\ \mathsf{in}\ x \leqslant p$$

Lors du passage au transformateur Π, on obtient une formule très «lourde» en comparaison de la simplicité du code $\lambda_{\bigcirc}$ représentant l'algorithme, comme on peut le constater sur les figures 2 (page 37) et 3 (page 38).
On veut donc prouver que cet algorithme simule la loi de Bernoulli. Il faut donc prouver les deux :

Théorème 3.1.1 (Loi de Bernoulli, probabilité du true**)** *Soit $p \in [0, 1]$. La mise en œuvre de l'algorithme ci-dessus avec le paramètre p donne un algorithme probabiliste tel que si l'on l'échantillonne, alors il termine et on obtient* true *avec la probabilité p.*

Théorème 3.1.2 (Loi de Bernoulli, probabilité du false**)** *Soit $p \in [0, 1]$. La mise en œuvre de l'algorithme ci-dessus avec le paramètre p donne un algorithme probabiliste tel que si l'on l'échantillonne, alors il termine et on obtient* false *avec la probabilité $1 - p$.*

FIG. 2 – La formule générée par le transformateur Π pour la simulation de la loi de Bernoulli, prête à être copiée dans Coq.

FIG. 3 – La même formule, copiée dans Coq, et imprimée par le *pretty-printer* de Coq.

Pour ma part, j'ai prouvé le premier théorème, dans le fichier `bernoulli.v`. Mais comme il s'agissait d'un des premiers algorithmes que j'ai prouvés, je n'ai pas dissocié terminaison et correction. Toutefois, on pourra constater, au vu de la figure 5, page 41, une «coupure» dans la preuve du théorème, entre terminaison et correction, signalée par un simple commentaire. C'est ce qui m'a donné l'*intuition* de structurer mes raisonnements (intuition confirmée par la situation analogue que j'ai rencontrée ensuite pour la loi exponentielle), que j'ai concrétisée plus tard pour les raisonnements sur la loi binomiale et la loi géométrique.
Pour démontrer ce théorème, j'ai choisi de le faire par probabilités locales (cf. section 2.3). En effet, comme on l'a vu, la loi de Bernoulli est souvent utilisée de manière auxiliaire dans d'autres lois. J'énonce le théorème en Coq comme suit :

```
Theorem Concl_bernoulli :
Double_triangle bernoulli (fun _ => True) (fun p f =>
0 <= p <= 1 ->
Double_carre f (fun _ => True) (fun s y _ =>
Proba_locale s (y = true) = p)
).
```

Les termes Coq `Double_triangle`, `Double_carre` expriment le fait que je n'aie pas séparé terminaison et correction. Les termes Coq `(fun _ => True)` indiquent que l'algorithme termine toujours.
La terminaison se démontre facilement, en dépliant les définitions et en «suivant» les réductions à la manière de la technique développée à la section 2.5.
Pour la correction, je me ramène à mesurer l'ensemble des flots $\alpha :: \varphi$ tels que la tête $\alpha \leqslant p$. Pour cela, j'ai recours à un lemme, `Bernoulli1`, que j'ai démontré dans le fichier `lo_axiom.v` (mais que j'aurais pu inclure dans le fichier `bernoulli.v`), selon lequel cet ensemble de flots est de mesure p. Donc, en injectant simplement, par une tactique `rewrite` (cf. fig. 5 p. 41), l'égalité fournie par ce lemme, j'obtiens alors une égalité de la forme `Proba_locale s P = Ef_mesure Q`, où `Q` est une fonction (prédicat à une variable). Alors, je réécris `Ef_mesure Q = Proba_locale s (Q s)`, en utilisant l'axiome de la probabilité locale et globale (2.3.2, page 32), et enfin, je me ramène à montrer que les propriétés `P` et `(Q s)` sont équivalentes (en appliquant 2.3.1, page 32).

J'ai omis de démontrer le second théorème, concernant la probabilité du false. Toutefois, on pourrait le faire par probabilité locale (en supposant la terminaison) et en disant que je mesure le complémentaire de l'ensemble des flots tels que j'obtienne true.

3.1.2 Loi exponentielle de paramètre 1

La loi exponentielle de paramètre 1 se simule par un algorithme similaire à celui de la loi de Bernoulli. En $\lambda_{\bigcirc}$ le code correspondant est :

$$\mathsf{prob\ sample}\ x\ \mathsf{from\ prob}\ \mathcal{S}\ \mathsf{in}\ 0 - \ln x$$

On veut prouver que cet algorithme simule la loi exponentielle de paramètre 1.
Il faut donc prouver le :

Théorème 3.1.3 (Loi exponentielle de paramètre 1) *Si on échantillonne l'algorithme probabiliste ci-dessus, alors pour tout $p \geqslant 0$, la probabilité qu'il termine et renvoie une valeur $y \geqslant p$ est égale à e^{-p}.*

FIG. 4 – Simulation de la loi de Bernoulli prête à être prouvée sous CoqIDE

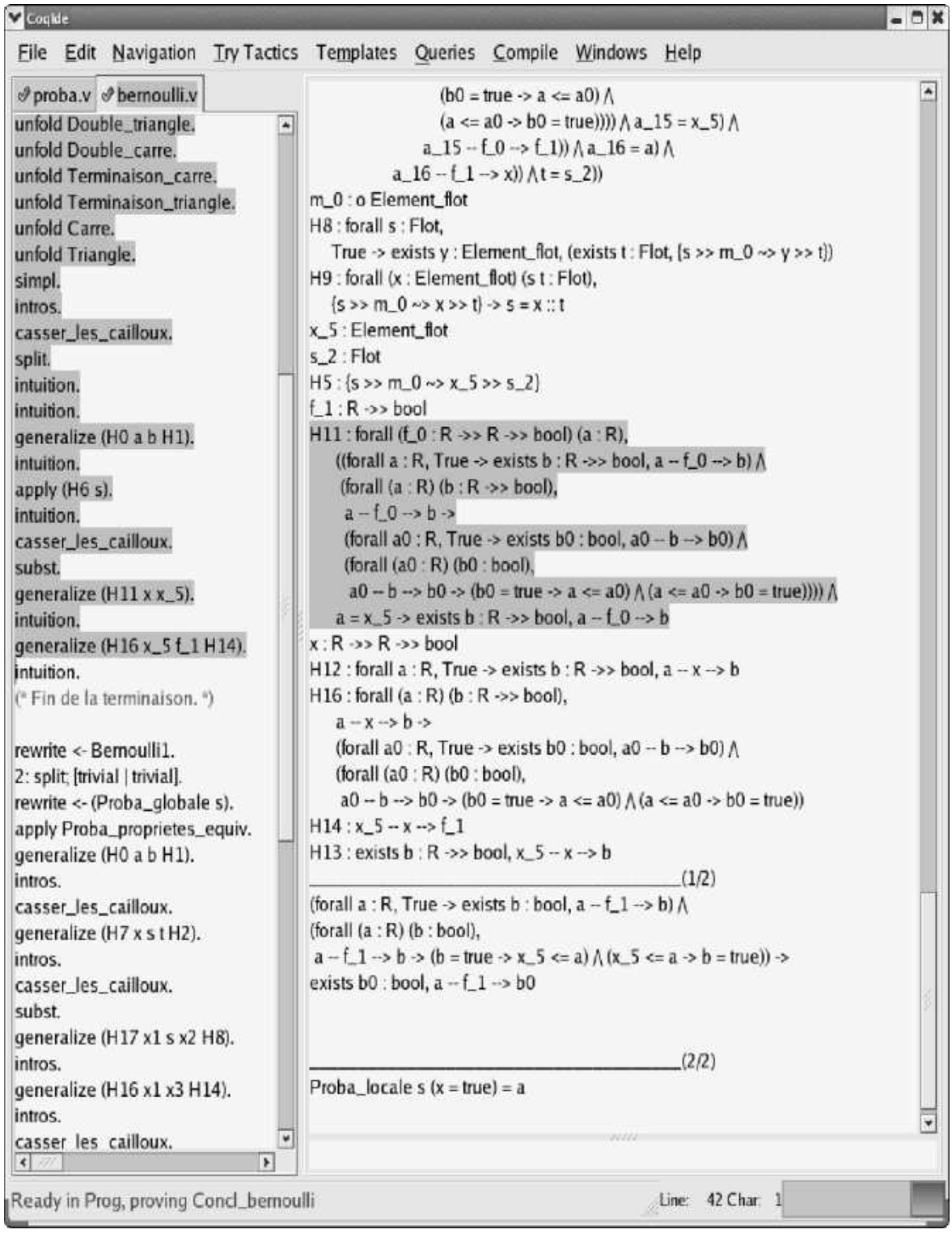

FIG. 5 – Preuve en cours sous CoqIDE de la simulation de la loi de Bernoulli

Ce théorème est suffisant, car il décrit la loi par sa *fonction de distribution*.
Cependant, un problème va se poser pour la terminaison : étant donné que j'autorise qu'une valeur de flot soit nulle, je risque de vouloir évaluer un $\ln 0$, où le zéro est la valeur de la tête du flot. Je considère donc que si le flot a une tête strictement positive, alors le programme termine. Pour ma part, j'ai montré une proposition légèrement plus faible : l'algorithme termine pour les flots de tête non nulle, et la probabilité, *conditionnée par la terminaison*, que la valeur retournée soit plus grande que p est e^{-p}. On pourrait par la suite compléter cette démonstration en montrant que l'ensemble de terminaison est de probabilité 1.

La terminaison sur l'ensemble des flots de tête non nulle se fait par «extraction» de l'hypothèse correspondante (en gros, une tactique `apply`) et également en suivant les réductions au passage.
Pour la correction, je fais exactement comme pour la loi de Bernoulli. Pour l'énoncer, j'utilise les probabilités locales ; pour la démontrer, j'ai recours à un lemme, `Expo1`, selon lequel la mesure de l'ensemble des flots $\alpha :: \varphi$ tels que $0 - \ln \alpha \geqslant p$ est e^{-p}. Je m'y ramène comme pour Bernoulli, par réécriture d'égalités.
Le point plus subtil est celui de la démonstration du lemme `Expo1`. En gros, je veux transformer l'inégalité $-\ln \alpha \geqslant p$ en $\alpha \leqslant e^{-p}$, «en appliquant le logarithme des deux côtés de la deuxième inégalité», afin de me ramener au lemme `Bernoulli1` (ce qui, au passage, montre l'intérêt de l'avoir placé directement dans le fichier `lo_axiom.v` au lieu de le cantonner dans `bernoulli.v`) pour prouver que la mesure de l'ensemble est e^{-p}. Mais pour effectuer cette transformation, je dois m'assurer que $\alpha > 0$. C'est ici que je me sers d'un lemme (que j'ai démontré en Coq) selon lequel en retirant un point (ici le point 0) d'un intervalle, on ne change pas sa mesure. Pour montrer le théorème 3.1.3 de façon plus rigoureuse avec les probabilités globales, on pourrait aussi se servir de ce lemme.

3.2 Récursivité déterministe : loi binomiale

En impératif, la loi binomiale de paramètres n et p est simulée par une «boucle *for*» :

```
procedure binomial (p : R, n : nat) : nat
begin
 result := 0;
 for i = 1 to n
 do
  if bernoulli(p) then result := result + 1;
 done;
 return result;
end.
```

qui s'écrit, en $\lambda_{\bigcirc}$:

λp : R. fix *boucle* : nat $\to \bigcirc$ nat. λn : nat.
  if $n = 0$
  then prob 0
  else prob
   sample $result_{prev}$ from *boucle* $(n-1)$ in
   sample b from *bernoulli* p in

if b then $result_{prev} + 1$ else $result_{prev}$

Comme on le constate dans cet algorithme, apparaît une variable libre, *bernoulli* : en fait, cet algorithme fait appel à une simulation de la loi de Bernoulli comme loi auxiliaire.
Il faut donc spécifier, en Coq, les hypothèses nécessaires à propos de cette variable *bernoulli* : selon la figure 6, page 44, j'indique que cette variable est un algorithme prenant en paramètre un réel p et tel que sous ce paramètre, il simule la loi de Bernoulli ; je formule cette dernière propriété *en utilisant les probabilités locales*, et *sans avoir à spécifier comment l'algorithme utilise le flot* de données aléatoires.

A partir de ces hypothèses, on veut montrer le :

Théorème 3.2.1 (Loi binomiale) *Soit $p \in [0, 1]$. Alors, sous le premier paramètre p, et pour tous entiers naturels n et $r \leqslant n$, si n est passé en second paramètre, alors l'échantillonnage termine toujours et la probabilité que le résultat soit r est $\begin{pmatrix} n \\ r \end{pmatrix} p^r (1-p)^{n-r}$.*

Ici, fort de mes expériences lors de la démonstration de la simulation des lois de Bernoulli et exponentielle, j'ai commencé à dégager une structure dans mon module de preuve : j'ai clairement séparé terminaison et correction. Voici les trois énoncés Coq que j'ai isolés :

```
Theorem Concl_binomial_termine :
Double_triangle binomial (fun _ => True) (fun p binp =>
0 <= p -> p <= 1 ->
Double_triangle binp (fun _ => True) (fun n f =>
Terminaison_carre f (fun _ => True)
)).

Axiom Concl_binomial_mesurable :
Triangle binomial (fun p binp => 0 <= p -> p <= 1 ->
Triangle binp (fun n f =>
Carre f (fun _ y _ =>
forall r :nat, (Le r n) -> In _ Espace_flots (fun _ => y = r)
))).

Theorem Concl_binomial :
Triangle binomial (fun p binp => 0 <= p -> p <= 1 ->
Triangle binp (fun n f =>
Carre f (fun s y t =>
forall r :nat, (Le r n) ->
Proba_locale s (y = r) = C n r * p ^ r * (1 - p) ^ (n - r)
))).
```

Comme on l'aura constaté, j'ai posé en axiome le second énoncé. Mais on pourrait le démontrer en suivant, en fait, la structure de la preuve du théorème `Concl_binomial`.

Pour démontrer la terminaison, j'ai besoin d'exhiber deux fois la preuve de bonne fondation de la relation $<$ (`Lt`) sur les entiers naturels : une fois pour prouver que `binp` (la boucle elle-même) termine, et une fois pour prouver que `f` (le terme prob contenu dans la boucle) termine.

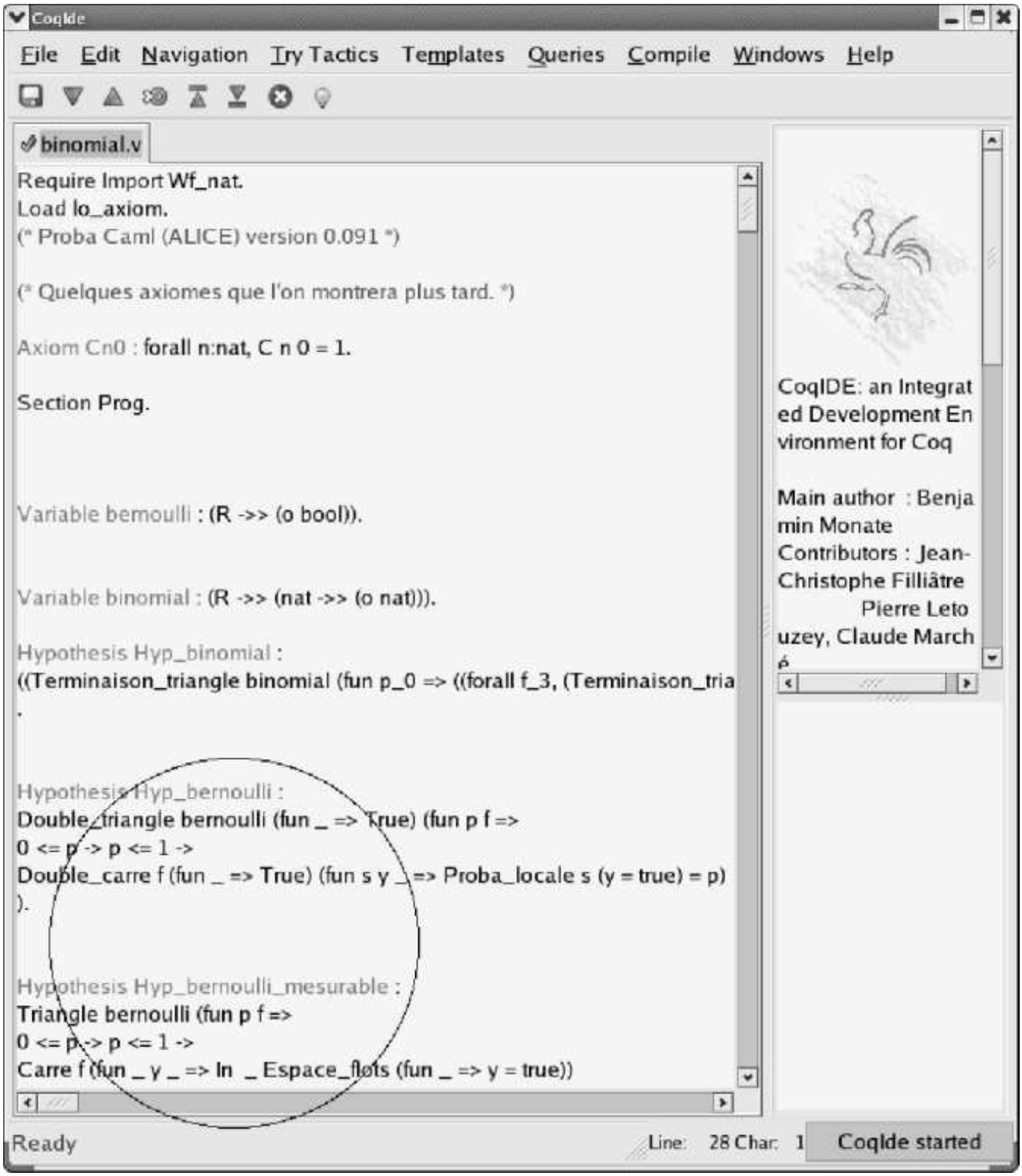

FIG. 6 – Simulation de la loi binomiale : hypothèses nécessaires à la preuve

La variable libre `bernoulli` *représente l'algorithme de simulation de la loi de Bernoulli, utilisé comme algorithme auxiliaire, c'est pourquoi on a mis en évidence, en les entourant grossièrement, les hypothèses* `Hyp_bernoulli` *et* `Hyp_bernoulli_mesurable` *sur cette variable.*

Pour démontrer la correction, il faut d'abord démontrer un lemme : la sortie de l'échantillonnage est toujours inférieure ou égale au paramètre n. En effet, je souhaite pouvoir raisonner *par induction sur* n pour prouver le théorème de correction, or celui-ci exige que $r \leqslant n$. La raison de cette exigence est que les coefficients binomiaux de Coq (le terme `C n r`) sont mal définis pour $r > n$. La convention en mathématiques est qu'ils vaillent zéro, mais ce n'est pas le cas en Coq.
C'est d'ailleurs la raison pour laquelle, pour montrer ce théorème de correction, j'ai besoin, dans le cas inductif, de distinguer trois cas :
– le cas $r = n$: tous les échantillonnages de Bernoulli donnent false, correspondant à une probabilité $(1-p)^n$, le coefficient binomial valant 1.
– le cas $r = 0$: tous les échantillonnages de Bernoulli donnent true, correspondant à une probabilité p^n, le binomial valant 1.
– le cas $0 < r < n$: les deux branches du test sont explorées. Ce cas entraîne l'utilisation de la formule de Pascal (définie dans la librairie standard de Coq) :

$$\begin{pmatrix} n \\ r \end{pmatrix} + \begin{pmatrix} n \\ r+1 \end{pmatrix} = \begin{pmatrix} n+1 \\ r+1 \end{pmatrix}$$

Cette formule exprime en elle-même le passage du cas n au cas $n+1$. La somme traduit le fait que le test donne lieu à deux branches disjointes ; elle apparaît grâce à la propriété d'additivité de la mesure.
En effet, si les coefficients binomiaux étaient définis correctement en Coq, alors j'aurais pu utiliser directement la formule de Pascal, pour laquelle les cas $r = 0$ et $r = n$ conduisent à l'annulation de l'un des coefficients binomiaux.

On remarquera, pour conclure, que les méthodes de preuve sont très différentes pour la terminaison et pour la correction. En effet, la terminaison se démontre avec l'ordre bien fondé, tandis que la correction se démontre par induction structurelle sur n.

3.3 Récursivité probabiliste : loi géométrique

En impératif, la loi géométrique de paramètre p est simulée par une «boucle *while*» probabiliste :

```
procedure geometric (p: R) : nat
begin
 result := 0;
 while not bernoulli(p)
 do
  result := result + 1;
 done;
 return result;
end.
```

qui s'écrit, en $\lambda_{\bigcirc}$:

λp : R. fix *boucle* : $\bigcirc$ nat. prob
  sample b from *bernoulli* p in
  sample *return* from

```
  if b
  then prob 0
  else prob
    sample result_prev from boucle in result_prev + 1
in return
```

Ici aussi, la variable libre *bernoulli* apparaît. Mais nous avons affaire à une récursivité probabiliste, dont le variant est *le flot lui-même*. Nous allons donc devoir parler en détail des opérations effectuées sur le flot. Donc, première conséquence, *pas d'utilisation des probabilités locales*. L'hypothèse sur l'algorithme de simulation de la loi de Bernoulli doit être exprimée avec précision quant à la consommation du flot. Ici, l'algorithme auxiliaire choisi agit en une consommation :

```
Hypothesis Hyp_bernoulli_conso :
Triangle bernoulli (fun p f => 0 <= p -> p <= 1 ->
Carre f (fun s y t => exists x, s = x :: t /\
(y = true -> x <= p) /\ (y = false -> ~ (x <= p))
)).
```

On veut alors montrer le :

Théorème 3.3.1 (Loi géométrique) *Soient $p \in [0,1]$, $n \in \mathbb{N}$. Alors, sous le paramètre p, la probabilité que l'échantillonnage de l'algorithme probabiliste ci-dessus termine et donne le résultat n est $(1-p)^n p$.*

Comme je l'ai annoncé à la section 2.1.6, pour pouvoir raisonner sur cet algorithme, j'ai besoin de considérer un ensemble de flots sur lequel je sais que l'algorithme termine, et un ordre bien fondé sur cet ensemble.
Je «vois» dans l'algorithme une «porte de sortie» de la boucle : lorsque le booléen b obtenu par échantillonnage de la loi de Bernoulli est true. Au niveau du flot, cela se traduit par le fait de tomber sur un élément de tête plus petit que p. Donc, une condition *suffisante* pour que l'algorithme termine est qu'il comporte au moins un élément plus petit que p.
Je «vois» aussi dans l'algorithme la situation d'appel récursif : lorsque ce même booléen b est false, on refait une itération. Au niveau du flot, cela se traduit par le fait de tomber sur un élément de tête plus grand que p. A ce moment-là, le point fixe rappelé avec la queue du flot. Donc, le flot avec sa tête plus grande que p doit être «plus grand» que sa queue, au sens de l'ordre sur le variant.
En somme :
- si je considère l'ensemble des flots tels que l'on tombe sur un élément $x \leqslant n$ au bout d'un nombre *fini* d'itérations, alors il est suffisant que le flot soit dans cet ensemble pour que l'algorithme termine.
- sur cet ensemble, l'ordre bien fondé $\prec$ doit vérifier que si je soumets à l'algorithme le flot d'entrée $\alpha :: \varphi$, avec $\alpha > p$, alors je vais le rappeler avec φ, donc il faut que $\varphi \prec \alpha :: \varphi$.

Or, il se trouve qu'un tel ensemble peut être assez facilement muni d'un tel ordre, et cela de façon générale : la propriété $\alpha \leqslant p$ peut être remplacée par n'importe quel prédicat $P(\alpha)$. C'est pourquoi j'ai défini directement dans le fichier `lo_axiom.v` (section `Ensf_ord`) un tel ensemble et un ordre bien fondé paramétrés par le prédicat P.

Définition 3.3.1 *Soit P un prédicat à une variable de l'ensemble E des éléments de flot, notons $E(P)$ l'ensemble des flots φ tels que $P(\varphi_n)$ pour un certain n.*

En Coq, il est défini inductivement : la tête vérifie P, ou alors, dans le cas contraire, la queue est dans l'ensemble.

```
Inductive Ensf : Ensemble Flot :=
| ensf_0 : forall (x : Element_flot) (t : Flot),
P x -> In _ Ensf (x :: t)

| ensf_S : forall (x : Element_flot) (t : Flot),
~ P x -> In _ Ensf t -> In _ Ensf (x :: t)
.
```

Il se trouve que la définition inductive de $E(P)$ est très proche de celle de n (ce qui est indiqué implicitement dans les clauses d'induction) : les flots commençant par un élément vérifiant P sont «des zéros», tandis que les autres, commençant par un élément ne vérifiant pas P, sont «les successeurs de leurs queues».
Aussi peut-on se servir de cette similitude pour définir l'ordre. Rappelons que l'ordre sur $\mathbb{N}$ est défini de la manière suivante :
– pour tout n, $0 < S(n)$
– pour tous n, n', si $n < n'$, alors $S(n) < S(n')$
Alors, définissons l'ordre sur $E(P)$ en suivant ce modèle :

Définition 3.3.2 *On dit que deux flots $\alpha :: \varphi \prec \alpha' :: \varphi'$ si, et seulement si :*
– *soit $P(\alpha)$ et $\neg P(\alpha')$ et $\varphi' \in E(P)$ (cas zéro)*
– *soit $\neg P(\alpha)$ et $\neg P(\alpha')$ et $\varphi \prec \varphi'$ (cas successeur).*

Cet ordre est défini *a priori* sur tous les flots, mais on montre facilement que deux flots comparables sont nécessairement dans $E(P)$.
Il se trouve qu'il vérifie la propriété voulue :

Lemme 3.3.1 *Si α ne vérifie pas P et si $\varphi \in E(P)$, alors $\varphi \prec \alpha :: \varphi$.*

que j'ai démontrée en Coq. En fait, ce lemme est similaire à celui que l'on connaît sur les entiers naturels : $n < S(n)$.

Pour montrer que $\prec$ est bien fondé, je procède alors par étapes :

1. je définis d'abord une famille, indicée par $\mathbb{N}$, de flots *de référence*. Mais, pour pouvoir les définir, j'ai besoin d'un élément $\top \in E$ vérifiant P et d'un élément $\bot \in E$ ne vérifiant pas P. Etant donnés deux tels éléments, je définis $\Phi(0)$ comme étant le flot constant $\top$, et par induction sur n, $\Phi(n+1) = \bot :: \Phi(n)$. Ces flots sont dans $E(P)$, et $\Phi(n) \prec \Phi(n+1)$
2. je démontre alors que tout flot de $E(P)$ est majoré par un certain flot de référence $\Phi(n)$ au sens de $\prec$
3. je montre ensuite la propriété très importante suivante : si $\varphi \prec \varphi' \prec \Phi(n+1)$, alors $\varphi \prec \Phi(n)$. En d'autres termes, il n'y a «rien entre $\Phi(n)$ et $\Phi(n+1)$»
4. cette propriété très importante me permet de montrer que tout flot majoré par un flot de référence est accessible

5. or, je sais que tout flot de $E(P)$ est majoré par un tel flot de référence. Donc, tous les flots de $E(P)$ sont accessibles. Comme deux flots comparables sont nécessairement dans $E(P)$, je sais alors que les flots hors de cet ensemble sont accessibles (par vacuité). Donc, $\prec$ est bien fondé.

Et voilà, j'ai défini un ensemble $E(P)$ et un ordre $\prec$ bien fondé sur les flots, et vérifiant la propriété voulue. Il ne me reste plus qu'à l'appliquer au prédicat $P(\alpha) := \alpha \leqslant p$, et, moyennant les éléments $\top := p$ vérifiant P et $\bot := p + 1$ ne vérifiant pas P, j'ai une instance de cet ensemble et de cet ordre bien fondé pour le problème qui m'intéresse ici, à savoir l'algorithme de simulation de la loi géométrique, dont je pourrai prouver la terminaison grâce à cet ensemble.

Pour montrer la correction, ne pouvant pas utiliser l'outil des probabilités locales, je suis obligé de raisonner directement en probabilité globale. Alors je montre, par induction sur n, que l'ensemble des flots φ tels que la mise en œuvre de l'algorithme termine et donne n est l'ensemble E_n (noté en Coq `Ens_geom_n p`) des flots φ commençant par n éléments plus grands que p :

```
Fixpoint Ens_geom_n (p : R) (n :nat) struct n : Ensemble Flot :=
match n with
| Zero[9] => Produit_ensembles_flots (fun x => x <= p) (Full_set _)
| Succ n' => Produit_ensembles_flots (fun x => ~(x <= p)) (Ens_geom_n p n')
end.

Theorem Concl_geometric_ensemble :
Triangle geometric (fun p g => 0 <= p -> p <= 1 -> forall n : nat,
((fun s => exists y, exists t, {s >> g ~> y >> t} /\ y = n) : Ensemble Flot) =
Ens_geom_n p n
).
```

La notation `{s >> g ~> y >> t}` correspond en fait à la relation $s \xrightarrow[\square]{g} \langle y, t\rangle$.

C'est cet ensemble E_n dont je vais montrer qu'il est mesurable et dont je vais calculer la mesure, toujours par induction sur n.

Remarque : au passage, on pourrait montrer que si l'algorithme termine, alors nécessairement le flot d'entrée est dans un des E_n ; en montrant alors que $E(P) = \bigcup_n E_n$, alors on aura montré que l'appartenance à $E(P)$ est aussi une condition nécessaire à la terminaison.

Ici encore, comme pour la loi binomiale, la terminaison se démontre en utilisant l'ordre bien fondé sur les flots, tandis que pour la correction, on raisonne par induction structurelle sur le résultat n.

[9] `Zero` est un alias pour `0%nat` et `Succ` pour `S`.

FIG. 7 – Simulation de la loi géométrique : hypothèses nécessaires à la preuve

Conclusion

L'approche que j'ai proposée durant ce stage me permet d'avoir une vision formelle des algorithmes probabilistes me permettant de raisonner correctement dessus au moyen d'un système d'aide à la preuve tel que Coq.
Le fait d'avoir transporté le programme en logique, par le biais du transformateur Π, me permet de raisonner sur les algorithmes probabilistes à partir de formules déterministes sur les programmes. Les formules probabilistes, que je ne rencontre que par la suite dans le raisonnement lui-même, font intervenir des calculs de probabilités qui se ramènent en fait à des mesures d'ensembles, mesures qui s'effectuent en se ramenant directement à des ensembles mesurables de flots. Ainsi, je ne parle d'aucun autre espace de probabilités que celui des flots ou celui des éléments de flots.
Par cette approche, je peux montrer qu'un algorithme, donné sous la forme d'un programme en $\lambda_{\bigcirc}$, respecte sa «spécification» probabiliste, donnée sous la forme d'un théorème à prouver sous l'hypothèse de la formule Π modélisant le comportement du programme, et sous les hypothèses sur les variables libres, concernant les algorithmes auxiliaires.

Ce qu'il reste à faire

Par manque de temps (deux mois, c'est très court), je suis allé à l'essentiel, droit au but ; par conséquent, il reste à compléter certaines formalisations en Coq plus périphériques comme le théorème de Carathéodory et la mesure de Lebesgue.
Mais une formalisation beaucoup plus importante et sur laquelle il faut réellement approfondir la réflexion est la notion de *probabilités locales*. En effet, je n'ai défini et utilisé cette notion qu'en tant qu'outil de raisonnement en Coq. Pour justifier l'approche proposée, il faut donner à cette notion de bonnes bases mathématiques qui n'ont pas pu être jetées au cours de ce stage.

En plus ces deux points permettant de compléter la formalisation de la théorie sous-jacente à ce stage, il pourrait être intéressant d'améliorer des points touchant à la mise en pratique des méthodes de raisonnement mécanisé proposées dans ce stage. Ces améliorations portent principalement sur les formules générées par le transformateur Π. En effet, on aura constaté l'extrême complexité de ces formules. La section 2.5 pourrait constituer une première piste dans l'écriture de tactiques Coq en $\mathcal{L}_{tac}$ pour la manipulation des formules. Mais on pourrait aussi songer à optimiser la génération même des formules par le transformateur. Lors d'une telle optimisation, on prendra bien sûr garde à ne pas perdre en expressivité, c'est-à-dire qu'il ne faudra pas perdre d'informations sur le programme.

Elargissements possibles

Un élargissement important réside dans la réflexion sur la principale restriction du modèle que j'ai choisi : la restriction à deux classes de points fixes (fix *boucle*. λn et fix *boucle*. prob), à variant uniquement déterministe ou uniquement probabiliste. Ce modèle ne me permet pas de raisonner sur des algorithmes probabilistes récursifs plus complexes, ceux dont le variant est «mixte», à la fois un argument récursif «déterministe» et le flot de données aléatoires (du genre fix *boucle*. λn. prob). Cet élargissement nécessiterait de considérer une classe plus générale de variants, et permettrait de réunifier les deux classes de points fixes que j'ai distinguées.

Enfin, si mon approche me permet de relier un algorithme et sa spécification (probabiliste) lorsqu'ils sont *donnés*, en *vérifiant* (et c'est précisément le titre du stage) que l'algorithme respecte

sa spécification, en revanche je n'ai rien évoqué sur la notion d'*extraction* d'un programme probabiliste à partir de la seule spécification. C'est un élargissement qui pourrait se révéler intéressant. En effet, une telle approche permet l'élaboration presque *ex nihilo* de programmes déjà certifiés et qu'il n'est plus nécessaire de vérifier.

Comparaison avec d'autres approches

– La génération des formules correspondant à un programme (par le transformateur Π) suit la sémantique opérationnelle. Mais il serait possible d'aborder plutôt une approche axiomatique pour décrire le comportement des programmes. Cependant, cette approche axiomatique impose le développement d'un concept d'interprétation des types, termes et expressions du $\lambda_{\bigcirc}$, et, rapporté au thème des algorithmes probabilistes, il faudrait pouvoir parler de tribus sur des ensembles très généraux donnés par les interprétations sur les types. Mais je considère cela comme un niveau *méta*, raisonnement probabiliste sur l'ensemble des programmes eux-mêmes, alors que l'approche que j'ai proposée raisonne au niveau d'un programme donné, et ne parle que des tribus sur les flots et les éléments de flot.

– Il aurait également été intéressant d'aborder la terminaison et la correction des algorithmes probabilistes suivant une approche similaire à celle de Why [Why]. Cette approche par obligations de preuves pourrait proposer une approche plus «locale» du raisonnement sur les algorithmes probabilistes. En effet, l'approche que j'ai proposée nécessite un raisonnement global sur une formule donnant le comportement global de tout un programme en $\lambda_{\bigcirc}$, alors que classiquement, on raisonne sur un programme en montrant plusieurs lemmes, les obligations de preuves, portant sur des fragments locaux de programme.

Bilan du stage

Ce stage m'a permis de découvrir de façon pratique le monde de la recherche, au sein des deux équipes *Marelle* et *Everest*. Je dois dire que la méthode de travail que j'ai personnellement mise en œuvre au cours de ce stage était très voisine de celle des chercheurs que j'ai côtoyés (en partie étudiants en thèse de doctorat). J'ai pu mettre en évidence, en gros, deux approches différentes de la recherche :

– s'aider de documents tout au long du chemin de recherche, en guise de «balises»
– à partir de documents de base, tracer son chemin de recherche de façon quasi autonome, éventuellement en reconstruisant des théories déjà existantes

C'est plutôt la seconde méthode, moins fréquente que la première, que j'ai appliquée durant ce stage : elle facilite la compréhension du sujet en m'impliquant à fond dans tous ses détails, mais demande peut-être un peu plus de temps que l'approche «balisée».

Ce stage a confirmé mes ambitions de poursuivre mes études en informatique fondamentale dans le domaine de la sémantique et de la vérification de programmes, en m'en faisant découvrir pour ainsi dire les entrailles, les différents domaines en exploitation : on aborde entre autres la sémantique de compilateurs ou de machines virtuelles comme JVM...
Mais le travail de recherche de l'équipe *Everest* (qui travaille dans le même bâtiment que l'équipe *Marelle* au sein de laquelle je travaillais) montre que je peux aussi combiner efficacement la sémantique et d'autres domaines de l'informatique fondamentale comme la cryptographie.

Annexe : utilisation des fichiers relatifs au stage

Téléchargement depuis le Web

Tous les fichiers du stage se trouvent sur ma page Web, à l'adresse suivante :

`http://www.eleves.ens.fr/home/ramanana/travail/lo/`

Les fichiers que j'ai créés durant ce stage se classent en plusieurs sous-dossiers :
- `pcaml` : les sources OCaml de l'interpréteur *Proba Caml*
- `pi` : les sources OCaml du transformateur Π
- `coq` : les sources Coq des théories formalisées (probabilités, flots, formules $\mathcal{L}_{\bigcirc}$, etc.) ainsi que des exemples traités au chapitre 3 avec leurs sources $\lambda_{\bigcirc}$ (dans la syntaxe du $\lambda_{\bigcirc}$ de Π)
- `tex` : les sources $\TeX$ des rapports de ce stage (dont le présent rapport technique)

En plus de ces sous-dossiers, on trouvera les archives compressées `*.tar.gz` (et `*.zip` pour Windows) des sous-dossiers `pcaml` et `pi`, ainsi que les diapositives de la présentation (aux formats MS PowerPoint 97 mais aussi PS et PDF).

Interpréteur *Proba Caml*

J'ai écrit l'interpréteur *Proba Caml* en OCaml. On en trouvera les sources dans le sous-dossier `pcaml`. On peut les consulter directement en ligne, ou bien télécharger l'archive `pcaml.tar.gz` que l'on décompressera en tapant successivement les deux commandes suivantes :

```
gunzip pcaml.tar.gz
tar xvf pcaml.tar
```

Sous Windows, on téléchargera l'archive `pcaml.zip` que l'on décompressera soit avec l'outil des Dossiers compressés de l'Explorateur Windows (sur des versions récentes comme Windows XP), soit avec des utilitaires tiers comme WinZip (sur des versions plus anciennes comme Windows 95).

J'ai volontairement omis de proposer une distribution directement exécutable, en raison de la très grande variété de plates-formes existant dans le monde informatique (moi-même, je disposais de trois plates-formes différentes pour travailler durant ce stage). C'est pourquoi, pour utiliser l'interpréteur, il faudra recompiler soi-même les sources fournies.
Pour ce faire, il faut bien entendu disposer d'un compilateur OCaml. On en trouve plusieurs distributions sur le site Internet officiel du langage Objective Caml [OCa].

Les fichiers source ont chacun leur rôle :

Fichier	**Description**
`lo.ml`	L'interpréteur proprement dit («syntaxe abstraite»)
`lo_parse.mly`	Le parseur («syntaxe concrète», format `ocamlyacc`)
`lo_lex.mll`	Le lexeur (format `ocamllex`)
`lo_print.ml`	L'imprimeur, permettant d'afficher à l'écran les éléments de langage
`lo_top.ml`	Le «point d'entrée» : entrée interactive ou lecture d'un fichier $\lambda_{\bigcirc}$.

La compilation s'effectue à la main :

```
ocamlyacc lo_parse.mly
ocamllex lo_lex.mll
ocamlopt -o pcaml nums.cmxa lo.ml lo_parse.mli lo_parse.ml \
  lo_lex.ml lo_print.ml lo_top.ml
```

Remarque : certaines distributions d'Objective Caml ne disposent pas d'un compilateur vers code natif (`ocamlopt`), comme la distribution Cygwin. Dans ce cas, il faut utiliser le compilateur *bytecode* (`ocamlc`), en utilisant le fichier librairie `nums.cma` au lieu de `nums.cmxa` :

```
ocamlc -o pcaml nums.cma lo.ml lo_parse.mli lo_parse.ml \
  lo_lex.ml lo_print.ml lo_top.ml
```

On obtient alors un programme nommé `pcaml`. Lancé sans paramètre, il lance simplement l'invite d'entrée interactive. Sinon, on peut spécifier une suite de fichiers source, que l'interpréteur lira dans l'ordre, un fichier pouvant dépendre de tous les fichiers précédents ; dans cette liste, on peut intercaler les options suivantes :

`-v`	Affiche les erreurs sur la sortie erreur standard (défaut)
`-q`	N'affiche pas les erreurs
`-l`	Affiche le résultat produit sur la sortie standard (défaut)
`-s`	N'affiche pas le résultat produit
`-o` *fichier*	Enregistre le résultat produit dans un fichier
`-e` *fichier*	Enregistre les erreurs dans un fichier
`--` *fichier*	Permet de lire un fichier d'entrée dont le nom commence par -
`-k`	Passe la main à l'utilisateur (invite clavier)
`-h`, `--help`	Affiche le message d'aide des options acceptées.

L'interpréteur prend les fichiers et les options *dans l'ordre où ils apparaissent* : les options `-v`, `-q`, etc. valent pour tous les fichiers indiqués *après* leur déclaration, jusqu'à l'indication d'une autre telle option. Par exemple, si l'on veut exécuter un fichier `fich1.lo` et sortir les résultats dans `fich1.out`, puis exécuter un second fichier `fich2.lo` et sortir les résultats à l'écran, alors on écrira :

```
pcaml -o fich1.out fich1.lo -l fich2.lo
```

A l'entrée interactive, ou dans un fichier «source dans le langage $\lambda_{\bigcirc}$ de *Proba Caml*», on peut entrer les *instructions* suivantes :

– *expr* `.`
  Une expression à calculer (ou un terme à évaluer)

– `let` [`rec`] *var* `:` *ty* `:=` *terme* `.`
  Déclaration d'un algorithme auxiliaire.

– `let` *var* `:` *ty* `:=` *terme* `and` *var* `:` *ty* `:=` *terme* [`and` ...].
  Déclaration de plusieurs algorithmes auxiliaires *non récursifs*.

Ne pas oublier le point à la fin de chaque instruction.

Remarque : l'entrée interactive ne reconnaît pas plusieurs instructions s'enchaînant sur une même ligne. (En revanche, on peut procéder ainsi si on passe par un fichier.)

La syntaxe du $\lambda_{\bigcirc}$ de *Proba Caml* a été dernièrement unifiée avec celle du $\lambda_{\bigcirc}$ du transformateur Π. Le rôle de Π étant de produire une formule Coq à partir d'un programme en $\lambda_{\bigcirc}$, cette syntaxe est donc plus proche de celle du «sous-ensemble Caml de *Gallina*[10]» que du «vrai» Caml, par l'utilisation, par exemple, du symbole `=>` à la place de `->` pour le constructeur de fonctions.

Variables. Dans le $\lambda_{\bigcirc}$ de *Proba Caml*, les variables sont des identificateurs uniquement composés des lettres latines, majuscules ou minuscules, non accentuées, ainsi que des chiffres de 0 à 9. Ils doivent commencer par une minuscule.

Types. Les parenthèses pour le type couple `*` sont obligatoires ; elles sont facultatives pour le type flèche `->` qui est associatif à droite :

$$ty ::= \quad \texttt{bool} \mid \texttt{nat} \mid \texttt{R} \mid ty \texttt{ -> } ty \mid \texttt{(} ty \texttt{ * } ty \texttt{)} \mid \texttt{O} \quad ty$$

où `nat` (resp. `R`) représente le type des entiers naturels (resp. des réels), comme en Coq.

Termes et expressions. Les parenthèses pour le couple sont obligatoires. L'application est associative à gauche et prioritaire devant l'abstraction `fun`. Un flottant est un nombre «à virgule» (en fait un point), comme en Caml ; un entier est un nombre «sans virgule» ; les entiers et toutes les opérations infixes doivent nécessairement être «scopés à la Coq», c'est-à-dire que le type de leurs arguments doit être indiqué au moyen d'une clause `%` (ce sont les termes notés $terme^{\%}$). Le type correspond à la clause `%` «la plus interne». Les parenthèses sont obligatoires là où elles apparaissent (en particulier pour les opérations infixes).

[10]Le «langage» de Coq lui-même, dit *vernaculaire*, à ne pas confondre avec $\mathcal{L}_{tac}$, le langage des tactiques de Coq.

$$
\begin{array}{lcl}
\textit{terme} & ::= & \texttt{true} \mid \texttt{false} \mid \textit{special} \mid \textit{flottant} \mid \textit{var} \\
& & \mid \texttt{fun}\ \textit{var}\ \texttt{:}\ \textit{ty}\ \texttt{=>}\ \textit{terme} \mid \textit{terme}\ \textit{terme} \mid \texttt{prob}\ \textit{expr} \\
& & \mid \texttt{fix}\ \textit{var}\ \texttt{:}\ \textit{ty}\ \texttt{:=}\ \textit{terme} \\
& & \mid \texttt{if}\ \textit{terme}\ \texttt{then}\ \textit{terme}\ \texttt{else}\ \textit{terme} \\
& & \mid \texttt{(}\ \textit{terme}\ \texttt{,}\ \textit{terme}\ \texttt{)} \mid \texttt{fst}\ \textit{terme} \mid \texttt{snd}\ \textit{terme} \\
& & \mid (\textit{terme}^{\%}\ \texttt{\%}\ \textit{ty}) \\
\\
\textit{terme}^{\%} & ::= & \texttt{true} \mid \texttt{false} \mid \textit{special} \mid \textit{flottant} \mid \textit{var} \\
& & \mid \texttt{fun}\ \textit{var}\ \texttt{:}\ \textit{ty}\ \texttt{=>}\ \textit{terme}^{\%} \mid \textit{terme}^{\%}\ \textit{terme}^{\%} \mid \texttt{prob}\ \textit{expr}^{\%} \\
& & \mid \texttt{fix}\ \textit{var}\ \texttt{:}\ \textit{ty}\ \texttt{:=}\ \textit{terme}^{\%} \\
& & \mid \texttt{if}\ \textit{terme}^{\%}\ \texttt{then}\ \textit{terme}^{\%}\ \texttt{else}\ \textit{terme}^{\%} \\
& & \mid \texttt{(}\ \textit{terme}^{\%}\ \texttt{,}\ \textit{terme}^{\%}\ \texttt{)} \mid \texttt{fst}\ \textit{terme}^{\%} \mid \texttt{snd}\ \textit{terme}^{\%} \\
& & \mid (\textit{terme}^{\%}\ \texttt{\%}\ \textit{ty}) \\
& & \mid \textit{entier} \mid \texttt{(}\ \textit{terme}^{\%}\ \textit{op}\ \textit{terme}^{\%}\ \texttt{)} \\
\\
\textit{expr} & ::= & \textit{terme} \mid \texttt{S} \mid \texttt{sample}\ \textit{var}\ \texttt{from}\ \textit{terme}\ \texttt{in}\ \textit{expr} \\
\\
\textit{expr}^{\%} & ::= & \textit{terme}^{\%} \mid \texttt{S} \mid \texttt{sample}\ \textit{var}\ \texttt{from}\ \textit{terme}^{\%}\ \texttt{in}\ \textit{expr}^{\%} \\
\\
\textit{op} & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
\textit{special} & ::= & \texttt{Rplus} \mid \texttt{Rminus} \mid \texttt{Rmult} \mid \texttt{Rdiv} \mid \texttt{Rlt} \mid \texttt{Rle} \mid \texttt{Rgt} \mid \texttt{Rge} \\
& & \mid \texttt{Plus} \mid \texttt{Minus} \mid \texttt{Mult} \mid \texttt{Lt} \mid \texttt{Le} \mid \texttt{Gt} \mid \texttt{Ge} \\
& & \mid \texttt{Ln} \mid \texttt{Exp} \mid \texttt{Succ} \mid \texttt{Zero} \mid \texttt{Pred}
\end{array}
$$

Sucres syntaxiques. En plus des structures du $\lambda_{\bigcirc}$, le $\lambda_{\bigcirc}$ de *Proba Caml* définit également les «sucres syntaxiques» suivants :

$$
\begin{array}{c}
\texttt{let}\ \textit{var}\ \texttt{:}\ \textit{ty}\ \texttt{:=}\ \textit{terme}\ \texttt{in}\ \textit{terme}' \\
\texttt{let rec}\ \textit{var}\ \texttt{:}\ \textit{ty}\ \texttt{:=}\ \textit{terme}\ \texttt{in}\ \textit{terme}'
\end{array}
$$

respectivement équivalents à :

$$
\begin{array}{c}
(\texttt{fun}\ \textit{var}\ \texttt{:}\ \textit{ty}\ \texttt{=>}\ \textit{terme}')\ \textit{terme} \\
(\texttt{fun}\ \textit{var}\ \texttt{:}\ \textit{ty}\ \texttt{=>}\ \textit{terme}')\ (\texttt{fix}\ \textit{var}\ \texttt{:}\ \textit{ty}\ \texttt{:=}\ \textit{terme})
\end{array}
$$

ainsi que les sucres syntaxiques de Coq pour les arguments des abstractions `fun` :

$$
\begin{array}{rcl}
\texttt{fun}\ (\textit{var}\ \texttt{:}\ \textit{ty})\ (\textit{var}'\ \texttt{:}\ \textit{ty}')\ \texttt{=>}\ \textit{terme} & \equiv & \texttt{fun}\ \textit{var}\ \texttt{:}\ \textit{ty}\ \texttt{=>}\ \texttt{fun}\ \textit{var}'\ \texttt{:}\ \textit{ty}'\ \texttt{=>}\ \textit{terme} \\
\texttt{fun}\ \textit{var}\ \textit{var}'\ \texttt{:}\ \textit{ty}\ \texttt{=>}\ \textit{terme} & \equiv & \texttt{fun}\ \textit{var}\ \texttt{:}\ \textit{ty}\ \texttt{=>}\ \texttt{fun}\ \textit{var}'\ \texttt{:}\ \textit{ty}\ \texttt{=>}\ \textit{terme}
\end{array}
$$

Opérations prédéfinies. Outre les opérations infixes classiques, *Proba Caml* reconnaît les fonctions de base sur les naturels `Pred`, `Succ`, ainsi que sur les réels `Exp`, `Ln`. Ces fonctions sont déclarées dans le module `lo.ml` lui-même (ce qui est une assez mauvaise stratégie).

Transformateur Π

J'ai également écrit le transformateur Π en OCaml (tout comme *Proba Caml*, dont, d'ailleurs, il dérive directement). On en trouvera les sources dans le sous-dossier `pi`. On peut les consulter

directement en ligne, ou bien télécharger l'archive `pi.tar.gz` que l'on décompresse comme pour PCaml avec `gunzip` puis `tar xvf` (ou, sous Windows, télécharger et décompresser `pi.zip`).
Les fichiers source jouent exactement les mêmes rôles que pour *Proba Caml*, avec le transformateur proprement dit dans le fichier `lo.ml`.
Tout comme pour *Proba Caml*, j'ai volontairement omis de proposer une distribution directement exécutable de Π. Il faut donc là encore compiler à la main, exactement comme pour *Proba Caml*, en changeant, si l'on veut, le nom de l'exécutable `pcaml` en `pi`.

Le transformateur Π est d'utilisation assez simple.
De deux choses l'une : soit l'utilisateur lance l'invite interactif de Π et tape directement son code $\lambda_{\bigcirc}$, soit il l'écrit dans un fichier qu'il demande alors au transformateur de lire, en le passant en paramètre. Dans les deux cas, Π écrit la formule Coq obtenue directement à l'écran, ou, sur demande de l'utilisateur (option `-o`), sous la forme d'un module source Coq.
En fait, la syntaxe de ligne de commande de `pi` est la même que celle de `pcaml`. Précisons que le résultat est la formule Coq elle-même, sur la sortie standard par défaut, tout autre message étant envoyé par défaut vers la sortie erreur standard.

Contrairement à Proba Caml, le transformateur Π ne reconnaît que les définitions de variables :
– `let` [`rec`]... pour la définition d'algorithmes
– ou bien, pour déclarer une variable libre sans donner de code :

`Variable` *var* `:` *ty* `.`

La syntaxe du $\lambda_{\bigcirc}$ de Π suit celle du $\lambda_{\bigcirc}$ de Proba Caml (voir ci-avant), avec laquelle elle a été dernièrement «unifiée» ; cependant, à la différence de l'interpréteur, le transformateur n'accepte que la définition des points fixes conformément à la restriction suivant les deux classes que j'ai distinguées en 2.1.6 :
– pour les points fixes déterministes :

`fix` *var* `(` *var′* `:` *ty′* `) variant "` *ordre* `" :` *ty* `:=` *terme*

correspondant au terme de point fixe :

$$\mathsf{fix}\ var :\ ty' \to ty.\ \lambda var' : ty'.\ terme$$

où *ordre* est l'ordre sur le variant, supposé bien fondé, à spécifier *en Coq* (on donnera par exemple `Lt` pour l'ordre classique sur les entiers naturels).

– pour les points fixes probabilistes :

`fix` *var* `prob "` *ens* `" variant "` *ordre* `" :` *ty* `:=` *expr*

correspondant au terme de point fixe :

$$\mathsf{fix}\ var :\ \bigcirc\ ty.\ \mathsf{prob}\ expr$$

où *ens* est un ensemble de flots tels qu'un échantillonnage sur ces flots termine, et *ordre* est l'ordre sur le variant flot, supposé bien fondé, tous deux à spécifier *en Coq*.

Le `let rec` (aussi bien en local `let rec ... in` que l'instruction `let rec`) est alors restreint à ces deux structures (en remplaçant simplement `fix` par `let rec`).

Remarque : pour préserver la «compatibilité entre *Proba Caml* et Π» dans le sens «naturel» (tout programme en $\lambda_{\bigcirc}$ de Π est accepté par *Proba Caml*), *Proba Caml* reconnaît ces syntaxes restreintes en ignorant les chaînes Coq.

Références

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art : The Calculus of Inductive Constructions*. Texts in Theoretical Computer science, EATCS. Springer Verlag, 03 2004.

[Coq] Le système d'aide à la preuve Coq. `http://coq.inria.fr`.

[Hur01] Joe Hurd. *Formalizing Mathematics to Verify Probabilistic Algorithms*. PhD thesis, Trinity College, University of Cambridge, 08 2001.

[OCa] Le langage de programmation Objective Caml. `http://caml.inria.fr/ocaml`.

[PPT05] Sungwoo Park, Frank Pfenning, and Sebastian Thrun. A probabilistic language based upon sampling functions. In *POPL'05*. ACM, 01 2005.

[Why] L'outil de vérification formelle de logiciels Why. `http://why.lri.fr`.

[Yca02] Bernard Ycart. *Modèles et Algorithmes Markoviens*. Number 39 in Société de Mathématiques Appliquées et Industrielles, coll. Mathématiques et Applications. Springer Verlag, 2002.

Table des figures