Assistant de preuve Coq

une boite à outils pour l'ingénierie des preuves

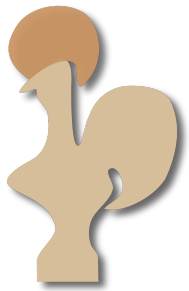
Christine Paulin-Mohring

Université Paris Sud & INRIA Saclay - Île-de-France

31 mars 2015

Objectifs

- Introduction à l'assistant de preuve Coq



  - Coq `http://coq.inria.fr`
  - s'applique à d'autres systèmes de la même famille HOL, Isabelle, PVS, Agda, Matita...
- Comprendre les concepts mis en œuvre : langage & architecture
- Intérêt d'une utilisation pour vos activités de recherche ?

Plan

- Introduction
- Langage
  - Objets & Types
  - Formules & Preuves
- Système de preuve
- Programmes aléatoires
  - Représentation
  - Bibliothèque ALEA
- Conclusion

Motivations

Pourquoi s'intéresser aux assistants de preuve ?

S'assurer que quelque chose est vrai

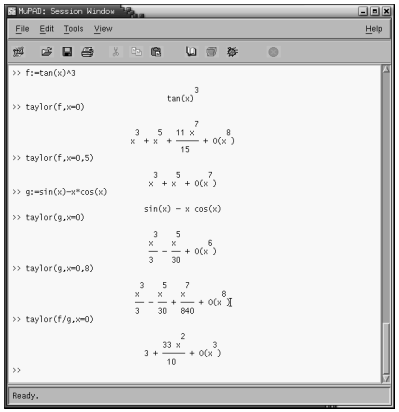
- Limitations liées à l'indécidabilité et la complexité
- Gérer des preuves infaisables à la main
  - ordinateurs : meilleurs que nous en activités routinières
  - favorisent le travail collaboratif
- Intégrer raisonnement et calcul
- Maximiser la confiance
  - parler aux humains autant qu'aux machines
- Emergence d'une nouvelle activité : Génie/Ingénierie des preuves

On a désappris à calculer

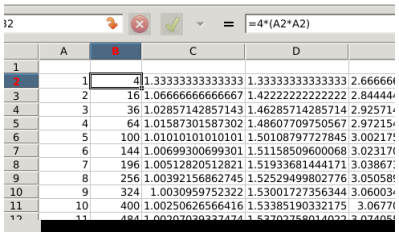
Pourra-t-on désapprendre la construction/vérification de preuves ?

Outils pour le calcul

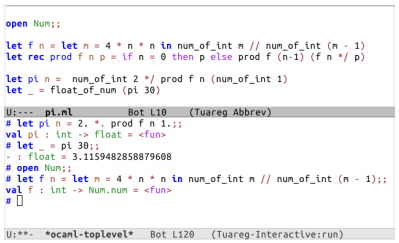
calculatrice
opérations de base



calcul formel
langage spécialisé,
fonctions avancées,
interface



feuille de calcul
présentation, édition



programmation
langage universel,
bibliothèques, spécialistes

problème : modéliser, combiner les méthodes

Plan

Propriété du langage

- Il ne suffit pas que le calcul soit juste, il faut avoir posé les bonnes équations
  - les spécifications doivent pouvoir être remises en question : relecture, confronter plusieurs définitions, tester, exécuter, prouver
- Un problème bien posé est à moitié résolu
  - un programme bien spécifié est à moitié prouvé...
  - choix des structures de données/définitions
  - le langage comme support à l'expression des besoins
  - compréhensible par l'homme et la machine

Choix du langage

La logique est un langage universel pour représenter des propriétés

- les constructions de base

  `vrai`($\top$), `faux`($\bot$),
  `non`($\neg$), `et` ($\wedge$), `ou` ($\vee$), `implique`($\Rightarrow$), `équivalent`($\Leftrightarrow$)...
  `pour tout` ($\forall$), `il existe` ($\exists$)

- Qu'en est-il des objets et propriétés atomiques ?

Représentation des objets

- Vision axiomatique
  - nouveaux symboles et axiomes
  - risques d'incohérence
- Théorie des ensembles

$$4 \equiv \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}\}$$

- Arithmétique de Peano $(0, S(\_), \_ + \_, \_ \times \_)$
  - codage des suites par des entiers
  - $2^n = m$

$$\exists e, |e| = n + 1 \wedge e(0) = 1 \wedge \forall i < n, e(i+1) = 2 \times e(i) \wedge e(n) = m$$

Analyse de la situation

- Machines de Turing comme modèle universel de calcul
  - oui mais les machines modernes sont plus sophistiquées
- Concevoir les langages modernes de la logique

Le point de vue informatique

- Mots binaires de taille bornée
- Code = suite finie d'instructions
  - une recette pour calculer des valeurs
  - moyen fini de décrire des objets potentiellement infinis
  - définition intentionnelle calculatoire
  - abstraction (lambda-calcul)/fermeture (langages fonctionnels)
- Organisation de la mémoire en blocs
  - entête : taille, étiquette pour organiser les blocs
  - termes, signature (symboles associés à des arités), algèbre libre

Types

Classifier finement les objets

- Des expressions (τ) pour représenter des ensembles d'objets
- Une relation $t : \tau$
  booléens, entiers, fonctions, arbres ...
- Reconnaissable statiquement (à la compilation, avant le calcul)
- En général un type canonique par objet
- Stable par calcul

  si $t : \tau$ et t s'évalue en u alors $u : \tau$
- limite les erreurs automatiquement (terminaison)
- Mais forcément des limitations

  $\mathbb{N}^*$ $\quad$ $1 : \mathbb{N}^*$ $\quad$ $x^2 - 2x + 1 : \mathbb{N}^*$?

les types ne sont pas des ensembles !

Types simples

- Types fonctionnels : $\tau \to \sigma$

$$\frac{f : \sigma \to \tau \qquad p : \sigma}{f\ p : \tau} \qquad [f(x) \stackrel{\text{def}}{=} t]\frac{x : \sigma \vdash t : \tau}{f : \sigma \to \tau}$$

- Types de base : `bool`, `nat`
  - constructeurs avec leur arité (typée)

```
Inductive bool := true : bool | false : bool.
Inductive nat := O : nat | S : nat → nat.
```

Définition de fonctions

- propriétés des types construits (algèbre initiale)
  - valeurs constructibles : viennent des constructeurs
  - constructeurs distincts
- analyse par cas

```
Definition isz (n:nat) :=
  match n with O ⇒ true | S m ⇒ false end.
```

- point-fixe structurel

```
Fixpoint exp2 (n:nat) :=
  match n with O ⇒ 1 | S m ⇒ 2 * exp2 m end.
```

Des objets qui calculent

```
Eval compute in exp2 4.
     = 16 : nat
```

- Tester (sur des valeurs closes)
- Si on peut calculer $f(n)$ en m alors $f(n) = m$ est trivial.

Limitations

- Toutes les fonctions ne peuvent pas se calculer
  - minimiser une fonction $f : \mathbb{N} \to \mathbb{N}$ $\quad \forall x : \mathbb{N}, f(\mathtt{min}(f)) \leq f(x)$
- Une valeur booléenne se calcule
  Une propriété logique se prouve

$bool \neq$ Prop

Quels programmes dans la logique ?

Pas n'importe quel calcul dans la logique

- problème de terminaison

$$\forall n, n = 0 \vee n \neq 0$$
$$0 \neq 1$$
$$f(x) = \textbf{if } f(x) = 0 \textbf{ then } 1 \textbf{ else } 0$$
$$f(x) = 0 \Leftrightarrow f(x) \neq 0$$

- calculs impurs
  - $p = p$ non valide si p fait des effets de bord

$$(x{+}{+} == x{+}{+}) \qquad \texttt{random} == \texttt{random}$$

Solutions

- fonctions représentées intentionnellement (relations logiques)
- expliciter les effets des constructions impératives (monades)
  - états, exception, entrées-sorties, aléatoire, non-déterminisme
- notations

Types étendus

- types polymorphes : `list` α, `tree` α

$$\frac{t : \forall \alpha, \tau[\alpha]}{t[\sigma] : \tau[\sigma]}$$

```
Inductive tree A :=
  leaf | node : A → tree A → tree A → tree A.
```

- types indicés : `vec` n

$$\frac{t : \forall x : \sigma, \tau[x] \qquad p : \sigma}{t\,p : \tau[p]}$$

```
Inductive vec A : nat → Type :=
  nil : vec A O | add : ∀ n,A → vec A n → vec A (S n).
```

- branchement indicé

```
Inductive BDT := T|F| var: nat → (bool → BDT) → BDT.
Inductive W A B := sup: ∀(a:A),(B a → W A B) → W A B.
```

Typage dépendant du match

- Forme élémentaire

```
match i in I x return P x i with
 c1 x1...xn1 ⇒ t1 (* : P u1 (c1 x1...xn1) *)
|...
|cp x1...xnp ⇒ tp (* : P up (cp x1...xnp) *)
end
```

- Exemple

```
Fixpoint app A n (v: vec A n) m (w:vec A m)
  : vec A (n+m)
  := match v in vec _ n return vec A (n+m) with
       nil ⇒ w
     | add p a v' ⇒ add a (app v' w)
     end.
```

Types dépendants

Définition récursive de types

```
Fixpoint prodn A (n:nat) : Type :=
  match n with O ⇒ unit | S m ⇒ A * prodn A m end.

Fixpoint pairn A (n:nat) (a:A) : prodn A n :=
  match n return prodn A n with
     O ⇒ tt | S m ⇒ (a,pairn m a)
  end.

Eval compute in pairn 4 true.
     = (true, (true, (true, (true, tt))))
     : prodn bool 4
```

Langage fonctionnel avancé

Récapitulatif objets et types

- Langage fonctionnel typé polymorphe
  - abstraction, application
- Types inductifs construits
  - constructeurs, filtrage, point-fixe
- Spécifique à Coq
  - les types sont des objets
  - types définis récursivement
  - une notion interne d'égalité modulo calcul

Preuves et vérité

S'il existe une preuve alors la formule est vraie
oui mais

- incomplétude : les preuves de cohérence sont inatteignables
  - systèmes logiques faux : paradoxe de Russel, système *Type* : *Type* (Martin-Löf)...
  - mais des systèmes rodés existent

    théorie des ensembles, arithmétique, logique d'ordre supérieur
- en pratique : si l'utilisateur introduit des hypothèses alors celles-ci peuvent être contradictoires
  - des axiomatiques manuelles incohérentes
  - interaction de plusieurs axiomes
  - raisonnement parfaitement correct mais qui ne sert à rien

Avoir un système assez puissant pour définir au lieu de supposer

Logique d'ordre supérieur

- type des propositions o, relations unaires $\tau \to o$, ...
- Deux connecteurs primitifs : $P \Rightarrow Q \quad \forall x : \tau, P$
- Quantification sur les propositions

$$\forall X : o, X \Rightarrow X \qquad \forall X : o, X$$

- Représentation des autres connecteurs logiques

$$P \wedge Q \stackrel{\text{def}}{=} \forall X : o, (P \Rightarrow Q \Rightarrow X) \Rightarrow X$$

- Quantification existentielle

$$\exists x : \tau, P \stackrel{\text{def}}{=} \forall X : o, (\forall x, P \Rightarrow X) \Rightarrow X$$

- Egalité

$$x =_\tau y \stackrel{\text{def}}{=} \forall X : \tau \to o, X\, x \Rightarrow X\, y$$

Propriétés en Coq

proposition = type de ses preuves

- une seule construction : implication et quantification universelle

$$A \Rightarrow B \stackrel{\text{def}}{=} A \to B \equiv \forall \_ : A, B$$

- autres connecteurs définis

```
Inductive True : Prop := I : True.
Inductive False : Prop :=.
Definition not (A:Prop) := A → False.
Inductive and (A B:Prop) : Prop :=
  | conj : A → B → A ∧ B
Inductive or (A B:Prop) : Prop :=
  | or_introl : A → A ∨ B | or_intror : B → A ∨ B
Inductive ex A (P:A → Prop) : Prop :=
  | ex_intro : ∀ x:A, P x → ex P.
```

Prédicats de base en Coq

```
Inductive eq A (x:A) : A → Prop := eq_refl : x = x.
```

$$\frac{t \equiv u}{t = u} \qquad \frac{t = u \quad P\,t}{P\,u}$$

```
Inductive ftrans A R (x y : A) : Prop :=
|  one : R x y → ftrans R x y
| join : ∀ z, ftrans R x z → ftrans R z y → ftrans x y.

Inductive Acc A R (x : A) : Prop :=
    Acc_intro : (∀ y:A, R y x → Acc R y) → Acc R x.
Definition well_founded A R := ∀ a:A, Acc R a.
```

Termes de preuves

- Proposition : type dans la sorte Prop.
- Preuve de P : terme de type P.
- Vérifier une preuve : typer un terme.
- Le typage est décidable
  - le terme contient tous les détails de la preuve
  - en Coq : intègre des calculs qui peuvent être complexes
- Le terme de preuve peut avoir d'autres applications
  - analyse des dépendances
  - explication
  - revérification indépendante

Exemple de preuve

```
∀ P : nat → Prop,
  P 0 → P 1 → (∀ n, P n → P (S (S n))) → ∀ n, P n
```

Preuve classique par récurrence

```
Variable P : nat → Prop.
Variables (p0:P 0) (p1:P 1) (p2:∀ n, P n → P (S (S n))).
Lemma nind2 : ∀ n, P n.
intro n; assert (P n ∧ P (S n)).
induction n; intuition.
destruct H; assumption.
Qed.
```

Preuves vues comme des programmes

```
Fixpoint nind2 (n:nat) : P n := match n with
   | 0 ⇒ p0 | 1 ⇒ p1 | S (S m) ⇒ p2 m (nind2 m) end.
```

Constructivité

- algorithme de décision de la propriété P
  $\Leftrightarrow$ preuve de $\forall x : A, P\,x \vee \neg P\,x$
- algorithme qui réalise la spécification Q
  $\Leftrightarrow$ preuve de $\forall x : A, \exists y : B, Q\,x\,y$

- Distinction $\mathrm{Prop} \neq \mathrm{Set}$
- Extraction de programmes exécutables (Ocaml)

Limites du langage de Coq

- Plusieurs égalités
  - $t \equiv u$: identiques par calcul $2+2 \equiv 4 \quad 0+x \equiv x$ décidable
  - $t = u$: prouvablement identiques par calcul $n+m = m+n$ échangeable dans n'importe quel contexte (bien typé)
  - $t \simeq u$: relation d'équivalence, substitution via congruence
- Langage intentionnel : $f\,x \stackrel{\text{def}}{=} x+0, g\,x \stackrel{\text{def}}{=} 0+x : \; f \neq g$
- utilisation des types dépendants délicate :

$$\texttt{vec}(n+m) \not\equiv \texttt{vec}(m+n)$$

- Théorie Types Homotopiques : meilleur traitement de l'égalité
- Condition de garde des points fixes

Récapitulatif langage logique

Logiques d'ordre supérieur

- ingrédients de base : fonctions et structures construites
- définition récursive par calcul ou définition implicite par règles
- preuve par récurrence sur les termes et sur les définitions

Coq

- langage fonctionnel calculatoire puissant
- interprétation constructive des propositions
- représentation des preuves par des termes typés

Plan

Assistant de preuve

- Passer d'un langage (la calculatrice) à un système
- Langage de plus haut niveau
  - Sucre syntaxique, notations
  - Précompilation de constructions avancées
    - arguments implicites
    - filtrage complexe
  - Théories prédéfinies (logique, entiers, booléens, listes . . .)
- Assistance à la construction de preuve
  - mode interactif dirigé par le but
  - tactiques automatiques
  - bibliothèques (propriétés, tactiques)
- Efficacité
  - modules (compilation séparée)
- Interface
  - chercher dans les bibliothèques

Noyau de confiance

- Etat : ensemble de définitions correctement typées
  - déclaration de types inductifs
  - hypothèses
  - termes représentant des objets
  - termes représentant des preuves
- Vérification
  - ensemble de règles de typage élémentaires
  - règles de calcul
- Extensions garanties
  - contributions distribuées
  - bibliothèques
  - tactiques

Automatisation

- Tactiques élémentaires qui se composent
- Programmation de procédures complexes
  - doivent produire un terme de preuve revérifiable
- Approche par trace
  - recherche de preuve externe
  - vérification d'un certificat
- Approche réflexive : stratégie programmée et vérifiée en Coq
  - vérifier la procédure plutôt que chacun des résultats
  - ramener la vérification au calcul
  - termes de preuves compacts

Exemple

Spécification inductive

```
Inductive pair : nat → Prop :=
  pair0 : pair 0
| pair2 : ∀ n, pair n → pair (S (S n)).
```

Spécification récursive

```
Fixpoint pairb (n:nat) : bool := match n with
  0 ⇒ true | 1 ⇒ false | S (S m) ⇒ pairb m end.
Lemma pairbok : ∀ n, pairb n = true → pair n.
```

Type des entiers pairs

```
Record deuxN := mk2N {val:>nat; isp : pair val}.
```

```
Notation "n ':2N'":= mk2N n (pairbok n (eq_refl true)).
Definition p400 : deuxN := 400 :2N.
```

Base du langage SSREFLECT (bibliothèques, tactiques).

Reflexion avancée

Stratégie de preuve programmée en Coq et utilisée sur les propositions Coq

- Type concret : C (formules, expressions, traces...)
- Interpétation : $I : C \to \mathrm{Prop}$
- Décision : $d : C \to \mathtt{bool}$
- Correction : $\mathtt{cor} : \forall x, d\,x = \mathtt{true} \Rightarrow I\,x$
- Utilisation pour prouver A
  - réification : p (clos) tel que $A \equiv I\,p$
  - si $d\,p \equiv \mathtt{true}$ alors $\mathtt{cor}\,p\,(\mathtt{eq\_refl\,true}) : A$

A propos de preuves par réflexion

- Possibilité de programmer les stratégies dans Coq
- La réification est un processus externe
- Langage de programmation limité
- Il faut prouver le programme *d*
- Exploite intensivement les capacités de calcul de Coq

Plan

Programmes aléatoires

- Important d'un point de vue théorique et pratique
- Théorie des probabilités
  - espace probabiliste
  - événement
  - variable aléatoire
- Programme probabilistes
  - fonction primitive de génération aléatoire
  - plusieurs appels indépendants
  - le programme représente la variable aléatoire

Langage

Un mini-langage fonctionnel avec des primitives aléatoires (random)

- constantes et fonctions primitives : c
- primitives aléatoires : `random`, `flip`, $(\_ +_p \_)$
- conditionnelle : **if** b **then** e_1 **else** e_2
- liaison locale : **let** $x = e_1$ **in** e_2
- abstraction : **fun** $(x : \tau) \to e$
- application : $(e_1\ e_2)$

deux évaluations du même programme peuvent donner des valeurs différentes

Exemples de programmes

```
let rec bernoulli p =
 let x = flip in
 if p < 1/2
 then if x then false else bernoulli (2*p)
 else if x then bernoulli (2*p-1) else true
```

```
let rec fair ()  =
 let x = biased () and y =  biased () in
 if x=y then fair () else x
```

Exemple de problème

Jeu de Monty-Hall

- un trésor est caché dans l'une des trois boîtes identiques *A*, *B*, *C*
- le joueur choisit une des boîtes
- le meneur du jeu ouvre une des boîtes restantes (vide)
- le joueur peut alors changer la boîte qu'il va ouvrir
- que doit-il faire ?
- quelles sont ses chances de gagner ?

```
type porte = A | B | C
let game strat =
 let tresor = porte1 () and choix1 = porte1 () in
 let ouvert =  porte2 tresor choix1 in
 let choix2 = strat choix1 ouvert in choix2 = tresor
```

Représentation logique

- plongement profond : représenter la syntaxe des programmes
- plongement superficiel : utiliser directement la sémantique
- approche monadique
  - expression aléatoire : représente un calcul de type τ
  - interprétation : expression purement fonctionnelle de type $[\tau]$
  - utilisation calculatoire dans des langages comme Haskell

Approche monadique

Opérateurs monadiques

$\texttt{return} : \tau \to [\tau]$ $\quad$ $\texttt{bind} : [\tau] \to (\tau \to [\sigma]) \to [\sigma]$

Primitive aléatoire

$\texttt{random} : \tau$ interprété comme un objet fonctionnel de type $[\tau]$.

Expressions

Calcul $p : \tau$	Valeur fonctionnelle $[p] : [\tau]$
let $x = a$ **in** b	$\texttt{bind}\ [a]\ (\textbf{fun}\ (x : \sigma) \to [b])$
fun $(x : \sigma) \to t$	**fun** $(x : \sigma) \to [t]$
$(t\ u)$	$\texttt{bind}\ [u]\ [t]$
if b **then** e_1 **else** e_2	$\texttt{bind}\ [b]$ $\quad$ **fun** $(x : \texttt{bool}) \to$ **if** x **then** $[e_1]$ **else** $[e_2]$

Point de vue opérationnel (simulation)

- générateur aléatoire global (tirages indépendants)
- accès à une suite infinie de valeurs aléatoires booléennes $\mathbb{B}^\infty$
- approche de Joe Hurd avec HOL
  http://www.cl.cam.ac.uk/~hurd
- variable globale $s \in \mathbb{B}^\infty$, consommée à chaque appel aléatoire

Expérience	$X : \alpha$	$[X] : \mathbb{B}^\infty \to \alpha \times \mathbb{B}^\infty$
Programme	$p : \alpha \to \beta$	$[p] : \alpha \to \mathbb{B}^\infty \to \beta \times \mathbb{B}^\infty$

- transformation monadique

$$[\alpha] = \mathbb{B}^\infty \to \alpha \times \mathbb{B}^\infty$$

```
return (a : α) : [α] = fun s → (a,s)
bind (a : [α])(f : α → [β]) : [β]
     = fun s → let (x,s') = a s in f x s'
flip : [bool] = fun s → (hd s,tl s)
```

Programme probabiliste dans HOL

- théorie de la mesure sur $\mathbb{B}^\infty$ formalisée dans HOL
- prob : $\mathcal{P}(\mathbb{B}^\infty) \to \mathbb{R}$
  - fonction partielle sur un ensemble $\mathcal{T}$ d'évènements clos par complément et union
- programme aléatoire $p : \alpha$
  $\to$ fonction HOL $[p] : [\alpha] = \mathbb{B}^\infty \to \alpha \times \mathbb{B}^\infty$
- probabilité que p satisfasse Q : prob($\{s \mid Q(\texttt{fst}\ ([p]\ s))\}$)

Bilan modélisation opérationnelle (HOL)

- Modélisation monadique simple
- Espace probabiliste clair, formalisé en HOL
- Possibilité de simulation (CAML) des programmes
- Raisonnement formel
  - Test de primalité de Miller-Rabin
- Modélisation des programmes qui terminent presque sûrement possible mais plus complexe à mettre en œuvre
- Extension dans les travaux de Sofiène Tahar et al. (Concordia U)
  - Variables aléatoires continues
  - Propriétés statistiques
  - Chaînes de Markov

Approche quantitative

Transformation de mesures

- voir travaux Dexter Kozen, Claire Jones and Gordon Plotkin, Annabelle McIver and Carroll Morgan...
- $p : \tau$ représente un ensemble de valeurs de type τ
- un programme $p : \tau$ correspond à une loi de probabilité μ_p sur τ
- si $Q \subseteq \tau$, alors $\text{prob}(p \in Q) = \mu_p(Q)$
- les primitives aléatoires exécutées déterminent l'espace de probabilité (implicite, dépend du chemin d'exécution)
- le programme est un ensemble d'observations sur cet espace

Quelle monade ?

- sortie du programme : loi de probabilité qui ne dépend que du programme.

$$e : \tau \rightsquigarrow [e] : 2^{\tau} \to [0,1]$$
$$\mathrm{prob}(e \in P) \rightsquigarrow ([e]\ P)$$

Approche directe pour une analyse quantitative.

$$\texttt{flip} : \texttt{bool} \rightsquigarrow \textbf{fun}\ P \to \tfrac{1}{2}\langle \texttt{true} \in P\rangle + \tfrac{1}{2}\langle \texttt{false} \in P\rangle$$

```
if flip then 0
else if flip then 1
else 2
```

$$\rightsquigarrow \textbf{fun}\ P \to \tfrac{1}{2}\langle 0 \in P\rangle + \tfrac{1}{4}\langle 1 \in P\rangle + \tfrac{1}{4}\langle 2 \in P\rangle$$

Construction de la monade

- ensembles $P : 2^\tau$ généralisés par des fonctions $f : \tau \to [0,1]$.
- $[\tau] = (\tau \to [0,1]) \to [0,1]$
- forme de double négation et de continuation

$$\begin{aligned}
&\texttt{return}\,(x:\tau):[\tau]\\
&\qquad = \textbf{fun}\,(f:\tau\to[0,1])\to\ (f\ x)\\
&\texttt{bind}\quad (\mu_a:[\tau])(\mu_b:\tau\to[\sigma]):[\sigma]\\
&\qquad = \textbf{fun}\,(f:\sigma\to[0,1])\to(\mu_a\,(\textbf{fun}\,(x:\tau)\to\ (\mu_b\ x\ f)))
\end{aligned}$$

- interprétation en terme de mesure
  - mesure de Dirac
  - indépendance des appels aléatoires
- expectation monad de Ramsey, Pfeffer en Haskell

Interprétation des primitives aléatoires

$$
\begin{array}{lcl}
\texttt{random}(n) & : & [\texttt{int}] \\
 & = & \textbf{fun}\, f \to \Sigma_{i=1}^{n} (f\ i)/n \\
\texttt{flip} & : & [\texttt{bool}] \\
 & = & \textbf{fun}\, f \to \frac{1}{2}(f\ \texttt{true}) + \frac{1}{2}(f\ \texttt{false}) \\
e_1 +_p e_2 & : & [\tau] \\
 & = & \textbf{fun}\, f \to p \times ([e_1]\ f) + (1-p) \times ([e_2]\ f)
\end{array}
$$

Calcul fonctionnel

- interprétation en Ocaml

```
type 'a distr = ('a → num) → num
let return a : 'a distr = fun f → f a
let bind (d:'a distr) (e:'a → 'b distr) : 'b distr
    = fun f → d (fun a → e a f)
let choice p a b = fun f → p * f a + (1 - p) * f b
let b2n b = num_of_int (if b then 1 else 0)
let prob (d:'a distr) (e:'a → bool)
    = d (fun x → b2n (e x))
```

- Application Monty-Hall
  - sans changer de porte : $1/3$
  - en changeant de porte : $2/3$

Récursion générale

Un programme qui termine seulement avec probabilité 1 :

```
let rec fix_test x =
        if flip then fix_test (x+1)
        else x
```

```
# fix_test 1;;
- : int = 1
# fix_test 1;;
- : int = 2
# fix_test 1;;
- : int = 2
# fix_test 1;;
- : int = 3
# fix_test 1;;
- : int = 1
```

Interprétation fonctionnelle

```
let rec fix_prob x =
bind flip
     (fun b → if b then fix_prob (x+1) else return x)
```

```
# fix_fun 1 (fun n -> 1.);;
Stack overflow during evaluation (looping recursion?).
```

- construire des limites pour analyser des programmes qui ne terminent pas forcément
- point fixe généraux en utilisant la structure de cpo

Raisonnement élémentaire

- Représentation Coq

```
Record distr (A:Type) : Type :=
  {mu : (A → U) -m-> U;
   mu_stable_inv  : stable_inv mu;
   mu_stable_plus : stable_plus mu;
   mu_stable_mult : stable_mult mu;
   mu_continuous  : continuous mu}.
```

- Opérateurs `bind`, `return` et `fix` sur `distr`
- Représenter la mesure $[e]$ associée au programme e
- Fonction caractéristique f associée à la propriété à tester
- On raisonne sur $[e](f)$ par calcul en utilisant les identités monadiques et les propriétés des primitives aléatoires
- Dériver des schémas avancés : sémantique axiomatique

Architecture de la bibliothèque

- bibliothèque relativement “autonome” (indépendante de $\mathbb{R}$)
  - 8000 lignes définitions
  - 10000 lignes de preuves
- principaux outils Coq utilisés
  - type classes (ordre, monotonie, continuité)
  - setoid rewrite
- plusieurs versions depuis environ 10 ans. . .
  - dernière version V8 (mai 2013)
  - disponible `http://www.lri.fr/~paulin/ALEA`

Machine de Turing probabiliste

```
Variables state alph : Type.
Variable accept : state → bool.
Inductive dir := L | R.
Record output := mkO {next:state;write:alph;move:dir}.
Variable trans : state → alph → distr output.
Record config := mkT {cur:state; tape:Z → alph; pos:Z}.
Definition update (m:config) (o:output) : config
  := mkT (next o)
         (fun z ⇒ if Zeq_bool z (pos m) then write o
                  else tape m z)
         (if move o then pos m - 1 else pos m + 1).
```

Machine de Turing probabiliste

Construction par point fixe

```
let rec run (m:config) : distr config =
if accept (cur m) then return m
else do out <- trans (cur m) (tape (pos m));
     run (update m out)
```

En Coq :

```
Definition Frun
 :(config → distr config) -m-> (config → distr config)
 := mon   (fun f m ⇒
          fif (accept (cur m)) (Munit m)
          (Mlet (trans (cur m) (tape m (pos m)))
                (fun out ⇒ f (update m out)))).
Definition run : config → distr config := Mfix Frun.
```

Bilan Alea

- sémantique formelle pour les programmes aléatoires
  - permet des calculs effectifs de probabilité dans des cas simples
  - adaptée à des raisonnements abstraits probabilistes (terminaison)
  - limitation aux variables aléatoires discrètes
- moins adaptée à la preuve de programmes spécifiques
  - les manipulations algébriques sur $[0, 1]$ peuvent être délicates (revenir à $\mathbb{R}$?)
  - automatiser les preuves de monotonie et de continuité
  - besoin d'automatisation des raisonnements combinatoires, simplification, approximation
  - primitives pour les programmes qui terminent presque sûrement

Plan

Applications de Coq

- Calcul scientifique (continu)
- Arithmétique des ordinateurs (flottants)
- Cryptographie (certicrypt)
- Mathématiques (constructives ou non)
- Sémantique des langages
- Algorithmes avancés
- Outils certifiés
  - compilateurs, analyse statique, outils de vérification
- Cible pour des outils de vérification (Why3, edukera)

Conclusion

Les ingrédients de la réussite de Coq

- Un langage avec un nombre limité de concepts mais puissant
- Intégration du calcul et du raisonnement
- Correspondance programme/preuve
- Architecture sécurisée mais ouverte
- Des outils matures (encore pour spécialistes)
- Une pénétration dans d'autres communautés

Questions ?