Première journée de la

Fédération d'informatique fondamentale de Paris Diderot

Sémantique des preuves et des programmes

Thomas Ehrhard

10 février 2014

But de cet exposé

- Présenter quelques éléments du paysage mental de PPS : en particulier, la correspondance de Curry-Howard entre preuves et programmes.
- Sans entrer dans les détails techniques.
- Servir d'introduction à d'autres exposés à venir, sur des thèmes un peu plus ciblés.

Sémantique de la logique : point de vue standard

Sémantique signifie *sens*. On est habitué à l'idée qu'un énoncé logique ait un sens, une sémantique : c'est la notion tarskienne de modèle.

Sens d'une formule : sa valeur de vérité, vrai ou faux.

Sens d'une formule avec des variables libres : l'ensemble des objets (ou des tuples d'objets) où cette propriété est vraie.

Mais si on prend au sérieux la notion de preuve, la sémantique d'une formule devient une notion beaucoup plus riche que sa simple valeur de vérité.

Les preuves dévoilent leur contenu calculatoire.

Lien entre preuves et programmes : correspondance de Curry Howard.

En logique intutionniste, une preuve a un contenu calculatoire. C'est une logique *constructive*.

Logique intuitionniste : logique ordinaire, sans le tiers-exclu $(A \vee \neg A)$ ou autres principes classiques "équivalents".

Exemples de principes classiques

- raisonnement par contraposition

$$(\neg B \Rightarrow \neg A) \Rightarrow (A \Rightarrow B)$$

- raisonnement par l'absurde

$$((A \Rightarrow \bot) \Rightarrow \bot) \Rightarrow A$$

où $\bot$ représente le "faux" ($\bot \Rightarrow A$ est intuitionniste)

- loi de Peirce

$$((A \Rightarrow B) \Rightarrow A) \Rightarrow A$$

Rôle des règles structurelles

Girard a montré que le caractère constructif de la logique intuitionniste n'est pas fondamentalement dû l'absence de ces règles classiques.

Il est dû à une restriction sur l'utilisation des *règles structurelles* :

- contraction : une même hypothèse peut être utilisée plusieurs fois
- affaiblissement : on n'est pas obligé d'utiliser une hypothèse.

Cette remarque est à l'origine de la *logique linéaire* qui associe de nouveaux connecteurs logiques aux règles structurelles : les *exponentielles*.

Une arithmétique constructive

L'arithmétique de Heyting utilise :

- Les règles habituelles de la logique intuitionniste pour les connecteurs et les quantificateurs.
- L'égalité.
- Des constantes 0, S, $+$ et $\times$.
- Des axiomes de base sur l'égalité, le successeur etc, par exemple $\forall x \forall y\, (Sx) + y = S(x + y)$.
- Le schéma de récurrence

$$F[0] \wedge \forall x\, (F[x] \Rightarrow F[Sx]) \Rightarrow \forall x\, F[x]\,.$$

Elle est constructive :

- D'une preuve de $A \vee B$, on peut extraire une preuve de A ou une preuve de B. En logique classique $A \vee \neg A$ est toujours prouvable.
- D'une preuve de $\exists x\, A$, on peut extraire un entier n tel que $A[n]$ soit vrai. En logique classique

$$\exists x((x = 0 \wedge B) \vee (x = 1 \wedge \neg B))$$

est trivialement prouvable, même si B est la conjecture de Goldbach.
- D'une preuve de $\forall x \exists y\, A$, on peut extraire un programme f tel que pour chaque entier n, on ait $A[n, f(n)]$.

Une preuve classique

En *calcul des séquents* (Gentzen) :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{B \vdash B}}{\vdash B,\ \neg B}}{\vdash ((0 = 0 \wedge B) \vee (0 = 1 \wedge \neg B)),\ ((1 = 0 \wedge B) \vee (1 = 1 \wedge \neg B))}}{\vdash ((0 = 0 \wedge B) \vee (0 = 1 \wedge \neg B)),\ \exists x((x = 0 \wedge B) \vee (x = 1 \wedge \neg B))}}{\vdash \exists x((x = 0 \wedge B) \vee (x = 1 \wedge \neg B)),\ \exists x((x = 0 \wedge B) \vee (x = 1 \wedge \neg B))}}{\vdash \exists x((x = 0 \wedge B) \vee (x = 1 \wedge \neg B))}$$

NB : la ligne horizontale représente une étape de déduction. Ligne double : plusieurs étapes dont on ne veut pas donner les détails.

En logique intuitionniste, une preuve calcule !

On peut obtenir ces programmes par *interprétation fonctionnelle* (Gödel, *Dialectica*) des preuves de l'arithmétique de Heyting.

Langage cible de cette interprétation : le système **T** de Gödel.

C'est un langage de programmation fonctionnelle typé, un précurseur de ML !

Description succincte de **T**

Système **T** : *fonctions primitives récursives de type fini*

Types :

- ι est un type : le type des entiers
- si σ et τ sont des types alors $\sigma \Rightarrow \tau$ est un type.

Convention : $\sigma \Rightarrow \tau \Rightarrow \varphi \;=\; \sigma \Rightarrow (\tau \Rightarrow \varphi)$.

Intuitions sur les types

Intuitivement, un type représente un ensemble d'objets. Le type $\sigma \Rightarrow \tau$ est le type des fonctions de σ vers τ. Pas toutes les fonctions bien sûr, seulement celles de notre langage ou de notre modèle.

Un objet f de type $\sigma \Rightarrow \tau \Rightarrow \varphi$ est une fonction qui prend un objet de type σ et rend une fonction qui prend un objet de type τ et rend un objet de type φ.

Un tel f est vu comme une fonction à deux paramètres (un dans σ et un dans τ) et rendant un résultat de type φ. Cela évite d'introduire des types produit.

Termes :

- *Constructions de base sur les entiers :* 0 est un terme ; si t est un terme alors $\mathrm{S}(t)$ est un terme
- *Variables :* toute variable x est un terme
- *Récursion primitive :* si s, t et u sont des termes alors $\mathrm{Rec}(s, t, u)$ est un terme
- *Application :* si s et t sont des termes alors $s\,(t)$ est un terme (application de s à t)
- λ-*abstraction :* si s est un terme, x une variable et σ un type, alors $\lambda x^{\sigma}\, s$ est un terme.

NB : ne pas confondre l'application du *constructeur* S au terme t avec l'application du terme s au terme t. Un constructeur n'est pas un terme, juste un symbole de fonction pour construire des entiers, comme le symbole S de l'aritmétique de Peano ou de Heyting.

Tous les termes que l'on peut former dans cette syntaxe ne sont pas acceptables : il faut pouvoir les *typer*.

Pour cela on se donne des règles de typage qui permettent de prouver des *jugements de typage* de la forme

$$\Gamma \vdash t : \sigma$$

où

- σ est un type,
- t est un terme
- et $\Gamma = (x_1 : \sigma_1, \ldots, x_n : \sigma_n)$ est un contexte de typage.

Dans un contexte de typage, les variables $x_1, \ldots, x_n$ sont 2 à 2 distinctes ; il sert à indiquer le type qu'on veut leur donner.

Règles de typage

Une règle a un nombre fini de *prémisses* (au-dessus du trait horizontal) et une conclusion (en-dessous de ce trait). Un *arbre de déduction* de typage a une instance de ces règles à chacun de ses noeuds.

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma} \qquad \frac{}{\Gamma \vdash 0 : \iota} \qquad \frac{\Gamma \vdash s : \iota}{\Gamma \vdash \mathsf{S}(s) : \iota}$$

$$\frac{\Gamma \vdash s : \sigma \qquad \Gamma \vdash t : \iota \Rightarrow \sigma \Rightarrow \sigma \qquad \Gamma \vdash u : \iota}{\Gamma \vdash \mathsf{Rec}(s, t, u) : \sigma}$$

$$\frac{\Gamma \vdash s : \sigma \Rightarrow \tau \qquad \Gamma \vdash t : \sigma}{\Gamma \vdash s\,(t) : \tau} \qquad \frac{\Gamma, x : \sigma \vdash s : \tau}{\Gamma \vdash \lambda x^{\sigma}\, s : \sigma \Rightarrow \tau}$$

Exemple de déduction de typage

$$\dfrac{\dfrac{}{x : \iota, y : \iota \vdash y : \iota} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{x : \iota, y : \iota, n : \iota, z : \iota \vdash z : \iota}}{x : \iota, y : \iota, n : \iota, z : \iota \vdash \mathsf{S}(z) : \iota}}{x : \iota, y : \iota, n : \iota \vdash \lambda z^{\iota}\, \mathsf{S}(z) : \iota \Rightarrow \iota}}{x : \iota, y : \iota \vdash \lambda n^{\iota}\, \lambda z^{\iota}\, \mathsf{S}(z) : \iota \Rightarrow \iota \Rightarrow \iota} \quad \dfrac{}{x : \iota, y : \iota \vdash x : \iota}}{\dfrac{\dfrac{x : \iota, y : \iota \vdash \mathsf{Rec}(y,\ \lambda n^{\iota}\, \lambda z^{\iota}\, \mathsf{S}(z),\ x) : \iota}{x : \iota \vdash \lambda y^{\iota}\, \mathsf{Rec}(y,\ \lambda n^{\iota}\, \lambda z^{\iota}\, \mathsf{S}(z),\ x) : \iota \Rightarrow \iota}}{\vdash \lambda x^{\iota}\, \lambda y^{\iota}\, \mathsf{Rec}(y,\ \lambda n^{\iota}\, \lambda z^{\iota}\, \mathsf{S}(z),\ x) : \iota \Rightarrow \iota \Rightarrow \iota}}$$

Sémantique d'un programme ?

Le terme s qu'on vient de typer $\vdash s : \iota \Rightarrow \iota \Rightarrow \iota$ calcule la somme de deux entiers :

$$s = \lambda x^{\iota}\, \lambda y^{\iota}\, \mathrm{Rec}(y,\ \lambda n^{\iota}\, \lambda z^{\iota}\, \mathrm{S}(z),\ x)$$

Comment utilise-t-on un tel langage ? En quel sens permet-il de calculer ?

Autrement dit : quelle est sa *sémantique* ?

Il y a (au moins) deux réponses :

- sémantique opérationnelle : réécriture sur les termes, machines abstraites
- sémantique dénotationnelle : les termes sont des fonctions.

Sémantique opérationnelle : règles de réduction.

On se donne des règles de réécriture sur les termes :

- La β-reduction $(\lambda x^{\sigma}\, s)\,(t) \rightsquigarrow s\,[t/x]$ (s dans lequel toutes les occurrences de x sont remplacées par le terme t ; attention aux captures de variables !).
- Ce que Rec fait si son dernier argument est 0 : $\mathrm{Rec}(s, t, 0) \rightsquigarrow s$
- Ce que Rec fait si son dernier argument est un successeur : $\mathrm{Rec}(s, t, \mathrm{S}(u)) \rightsquigarrow (t\,(u))\,(\mathrm{Rec}(s, t, u))$.

Pour les 2 dernières règles, il faut se souvenir que $u : \iota$, $t : \iota \Rightarrow \sigma \Rightarrow \sigma$ et $s : \sigma$.

On peut appliquer ces règles n'importe où dans un terme.

Exemple : soit $s = \lambda x^\iota\, \lambda y^\iota\, \mathsf{Rec}(y,\ \lambda n^\iota\, \lambda z^\iota\, \mathsf{S}(z),\ x)$, $\underline{1} = \mathsf{S}(0)$, $\underline{2} = \mathsf{S}(\underline{1})$, on a

$$
\begin{aligned}
(s\,(\underline{2}))\,(\underline{1}) &\rightsquigarrow (\lambda y^\iota\, \mathsf{Rec}(y,\ \lambda n^\iota\, \lambda z^\iota\, \mathsf{S}(z),\ \underline{2}))\,(\underline{1}) \\
&\rightsquigarrow \mathsf{Rec}(\underline{1},\ \lambda n^\iota\, \lambda z^\iota\, \mathsf{S}(z),\ \mathsf{S}(\underline{1})) \\
&\rightsquigarrow ((\lambda n^\iota\, \lambda z^\iota\, \mathsf{S}(z))\,(\underline{1}))\,(\mathsf{Rec}(\underline{1},\ \lambda n^\iota\, \lambda z^\iota\, \mathsf{S}(z),\ \underline{1})) \\
&\rightsquigarrow \mathsf{S}(\mathsf{Rec}(\underline{1},\ \lambda n^\iota\, \lambda z^\iota\, \mathsf{S}(z),\ \mathsf{S}(0))) \\
&\rightsquigarrow \mathsf{S}(((\lambda n^\iota\, \lambda z^\iota\, \mathsf{S}(z))\,(0))\,(\mathsf{Rec}(\underline{1},\ \lambda n^\iota\, \lambda z^\iota\, \mathsf{S}(z),\ 0))) \\
&\rightsquigarrow \mathsf{S}(\mathsf{S}(\mathsf{Rec}(\underline{1},\ \lambda n^\iota\, \lambda z^\iota\, \mathsf{S}(z),\ 0))) \\
&\rightsquigarrow \mathsf{S}(\mathsf{S}(\underline{1})) = \underline{3}
\end{aligned}
$$

On peut représenter de cette façon des fonctions très compliquées (Ackermann, et beaucoup plus... mais pas Goodstein !) :

Théorème

Toute fonction récursive $f : \mathbb{N} \to \mathbb{N}$ *dont la totalité est prouvable dans l'arithmétique de Peano est représentable par un terme* s *tel que* $\vdash s : \iota \Rightarrow \iota$, *au sens où*

$$\forall n \in \mathbb{N} \quad s\,(\underline{n}) \rightsquigarrow^* \underline{f(n)}$$

D'autre part

Théorème

Tout terme s (bien typé) du système **T** *est fortement normalisable, autrement dit, il n'existe aucune suite infine de termes $(s_i)_{i\in\mathbb{N}}$ telle que $s_0 = s$ et $\forall i\ s_i \rightsquigarrow s_{i+1}$.*

Or, à travers l'interprétation fonctionnelle, ce théorème entraîne la cohérence de l'arithmétique de Heyting, et par suite de celle de Peano (PA) : on ne peut pas prouver que $0 = \mathrm{S}(0)$.

C'est pour ça que Gödel a introduit le système **T**.

Par conséquent ce Th n'est pas démontrable dans PA, à cause du 2ème Th d'incomplétude de Gödel.

On a vu qu'on peut associer à une preuve constructive dans l'arithmétique un terme d'un langage de programmation (un λ-calcul étendu).

On peut même voir les preuves elles-mêmes comme des λ-termes.

Exemple : le système **F** de Girard.

Le système **F**

A nouveau, il y a des types et des termes, mais les types sont un peu plus compliqués (et les termes, beaucoup plus simples !).

- Si α est une variable de type, α est un type.
- Si σ et τ sont des types alors $\sigma \Rightarrow \tau$ est un type.
- Si α est une variable de type et σ est un type alors $\forall \alpha \, \sigma$ est un type.

Les termes sont ceux du λ-*calcul pur*.

- Si x est une variable, x est un terme.
- Si s et t sont des termes, alors $s\,(t)$ est un terme.
- Si x est une variable et s est un terme alors $\lambda x\, s$ est un terme.

Les règles de typage :

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma}$$

$$\frac{\Gamma \vdash s : \sigma \Rightarrow \tau \qquad \Gamma \vdash t : \sigma}{\Gamma \vdash s\,(t) : \tau} \qquad \frac{\Gamma, x : \sigma \vdash s : \tau}{\Gamma \vdash \lambda x\, s : \sigma \Rightarrow \tau}$$

$$\frac{\Gamma \vdash s : \forall \alpha\, \sigma}{\Gamma \vdash s : \sigma\,[\tau/\alpha]} \qquad \frac{\Gamma \vdash s : \sigma}{\Gamma \vdash s : \forall \alpha\, \sigma}$$

Pour pouvoir appliquer la dernière règle, il faut que α ne soit pas libre dans les types qui apparaissent dans Γ.

Si on oublie les termes, on reconnaît les règles logiques de l'implication et de la quantification universelle propositionnelle du second ordre.

Les λ-termes sont des notations pour des démonstrations de la logique propositionnelle minimale du second ordre, en déduction naturelle intuitionniste.

En fait ce sont des squelettes de notation, car les règles du $\forall$ n'apparaissent pas dans les λ-termes.

Polymorphisme

On peut voir **F** comme un système de types *polymorphes*. Par exemple, $\forall\alpha\,(\alpha \Rightarrow \alpha)$ est le type de l'identité sur tous les types possibles. On a $\vdash \lambda x\, x : \forall\alpha\,(\alpha \Rightarrow \alpha)$.

F est à l'origine du polymorphisme de ML (Ocaml, F# etc), lié aussi à l'héritage dans les langages objets.

Réduction

Dans ce calcul, il n'y a qu'une seule règle de réduction, la β-réduction :

$$\lambda x\, s\,(t) \rightsquigarrow s\,[t/x]$$

C'est un calcul sur les preuves : *élimination des coupures* (Gentzen).

Alors que dans **T** il y avait un type de base (le type ι des entiers naturels), ici il n'y en a pas. Parce qu'on peut les définir, par exemple :

$$\iota = \forall\alpha\,((\alpha \Rightarrow \alpha) \Rightarrow \alpha \Rightarrow \alpha)$$

Pour chaque entier n, on définit un *entier de Church* :

$$\underline{n} = \lambda f\ \lambda x\ f\ (f\ (\cdots f\ (x))) \quad n \text{ occurrences de } f$$

On peut définir ainsi de nombreux autres types (ou constructions sur les types) :

- produit de 2 types : $\sigma \times \tau = \forall\alpha\,((\sigma \Rightarrow \tau \Rightarrow \alpha) \Rightarrow \alpha)$
- somme de 2 types : $\sigma + \tau = \forall\alpha\,((\sigma \Rightarrow \alpha) \Rightarrow (\tau \Rightarrow \alpha) \Rightarrow \alpha)$
- listes des éléments de type σ : $\forall\alpha\,((\sigma \Rightarrow \alpha \Rightarrow \alpha) \Rightarrow \alpha \Rightarrow \alpha)$
  Par exemple, la liste constituée de s_1, s_2 et s_3 (tous supposés de type σ) est représentée par le terme

$$\lambda f\,\lambda z\,f\,(s_1)\,f\,(s_2)\,f\,(s_3)\,z$$

- Arbres binaires ou unaires à feuilles dans σ :
  $\forall\alpha\,((\alpha \Rightarrow \alpha \Rightarrow \alpha) \Rightarrow (\alpha \Rightarrow \alpha) \Rightarrow (\sigma \Rightarrow \alpha) \Rightarrow \alpha)$
- Arbres à feuilles dans σ et à branchements dans τ :
  $\forall\alpha\,(((\tau \Rightarrow \alpha) \Rightarrow \alpha) \Rightarrow (\sigma \Rightarrow \alpha) \Rightarrow \alpha)$

Théorème

Toute fonction récursive $\mathbb{N} \to \mathbb{N}$ *dont la totalité est prouvable dans l'arithmétique du second ordre est représentable par un terme du système* **F**.

Théorème

Si $\Gamma \vdash s : \sigma$, *alors* s *est fortement normalisable (aucune suite infinie de* $\rightsquigarrow$*-réductions à partir de* s*).*

Par interprétation fonctionnelle, ce théorème entraîne la cohérence de l'arithmétique du second ordre.

Donc il n'est pas prouvable dans l'arithmétique du second ordre.

NB : la cohérence de l'arithmétique de Peano est prouvable dans l'arithmétique du second ordre.

C'est un théorème de Girard, qui utilise la *réalisabilité* (Kleene) et une idée nouvelle : les *candidats de réductibilité*.

On peut la voir comme la construction d'un modèle de vérité (mais pas du tout à deux valeurs) du calcul des propositions du second ordre.

Soit Λ l'ensembles des λ-termes, Λ_{SN} l'ensemble des termes fortement normalisables.

$\mathcal{S}$ est une partie de $\mathcal{P}(\Lambda_{\mathsf{SN}})$ dont les éléments ont une certaine propriété de saturation par rapport à $\rightsquigarrow$.

Valuation : fonction $\mathcal{I}$ des variables de types vers $\mathcal{S}$.

Pour chaque type σ, on définit $\sigma^{\bullet}_{\mathcal{I}} \in \mathcal{S}$.

$$\alpha^{\bullet}_{\mathcal{I}} = \mathcal{I}(\alpha) \qquad (\forall \alpha\, \sigma)^{\bullet}_{\mathcal{I}} = \bigcap_{S \in \mathcal{S}} \sigma^{\bullet}_{\mathcal{I}[\alpha \mapsto S]}$$

$$(\sigma \Rightarrow \tau)^{\bullet}_{\mathcal{I}} = \{u \in \Lambda \mid \forall s \in \sigma^{\bullet}_{\mathcal{I}} \quad u\,(s) \in \tau^{\bullet}_{\mathcal{I}}\}$$

On montre que $\vdash s : \sigma \Rightarrow s \in \sigma^\bullet$ (s et σ clos). Mais la réciproque n'est pas vraie.

Quand $s \in \sigma^\bullet$ on dit que s *réalise* σ.

Or $\sigma^\bullet \in \mathcal{S} \subseteq \mathcal{P}(\Lambda_{\mathsf{SN}})$.

Donc $\vdash s : \sigma \Rightarrow s \in \Lambda_{\mathsf{SN}}$.

Cette technique (réalisabilité, relations logiques etc) est extrêmement puissante et permet de prouver de nombreux résultats de terminaison, dont certains n'admettent aucune autre preuve connue.

Marche aussi pour **T**, mais on sait faire autrement : induction jusqu'à l'ordinal ε_0.

Pour **F**, on ne sait pas.

Réalisabilité classique

La réalisabilité s'étend à la logique classique, et on peut remplacer le calcul des propositions du second ordre par la théorie des ensembles ZF.

On obtient ainsi la *réalisabilité classique* que développe Jean-Louis Krivine à PPS, qu'on peut voir comme une généralisation du *forcing* (Cohen).

Les *conditions de forcing* sont remplacées par des termes d'un λ-calcul *pur* (non typé) étendu.

- Nouveaux modèles de ZF.
- Interprétation calculatoire des axiomes de ZF et de ses extensions.

Calcul des constructions

Une autre possibilité est d'étendre les types de **F**, qui sont purement propositionnels, pour autoriser des prédicats portant sur des types.

Et donc aussi des types portant sur des types (types dépendants, à la Martin-Löf).

En faisant attention à ne pas identifier types et propositions, on obtient le *Calcul des Constructions* (Coquand, Huet).

λ-calcul pur

Que reste-t-il si on abandonne les types ? Le λ-*calcul pur*.

Il n'y a plus garantie de terminaison des calculs :

$$\begin{aligned}\lambda x\, x\,(x)\,(\lambda x\, x\,(x)) &\rightsquigarrow (x\,(x))\,[\lambda x\, x\,(x)/x] \\ &= \lambda x\, x\,(x)\,(\lambda x\, x\,(x)) \\ &\rightsquigarrow \lambda x\, x\,(x)\,(\lambda x\, x\,(x)) \\ &\rightsquigarrow \cdots\end{aligned}$$

Les types étaient un garde-fou contre la divergence.

Le λ-calcul pur a d'excellentes propriétés :

Théorème (Church Rosser)

La réduction est confluente : si $s \rightsquigarrow^* s_1$ et $s \rightsquigarrow^* s_2$, alors il existe t tel que $s_1 \rightsquigarrow^* t$ et $s_2 \rightsquigarrow^* t$.

Donc s'il existe un terme normal (sans réduction possible) t_0 tel que $t \rightsquigarrow^* t_0$, ce terme normal est unique. C'est la *forme normale* de t.

Il y a une "stratégie standard" qui atteint toujours la forme normale quand elle existe.

Le λ-calcul est un modèle universel du calcul, comme les machines de Turing etc.

Théorème

Pour toute fonction récursive partielle $f : \mathbb{N} \to \mathbb{N}$ il existe un λ-terme s tel que, pour tout entier n

- *si $f(n) = k$ alors $s(\underline{n})$ admet $\underline{k}$ comme forme normale*
- *si $f(n)$ est indéfini alors $s(\underline{n})$ n'a pas de forme normale (la réduction standard ne termine pas).*

Le λ-calcul pur est un objet mathématique fascinant, mais n'est pas très crédible en tant que langage de programmation concret, notamment du point de vue de la complexité.

Par exemple, il n'y a pas de λ-terme qui calcule le prédécesseur d'un entier de Church en un nombre constant d'étapes.

Calcul des Constructions Inductives

Pour programmer vraiment, on ajoute au Calcul des Constructions une méthode générique pour définir des *types inductifs* comme le type des entiers, des listes etc, inspirée de la Théorie des Types de Martin-Löf (Paulin-Mohring, Werner).

Calcul des Constructions Inductives

- Langage permettant de formaliser des preuves mathématiques, y compris des preuves de programmes. Comporte des mécanismes automatisés de construction de preuves (tactiques).
- Langage de programmation avec un système de types extrêmement riche.
- C'est ce système logico-calculatoire qui est à la base du système Coq développé par l'équipe-projet INRIA πr^2 de PPS.

Autre langage de programmation fonctionnel : PCF

Programming with computable functionals PCF, un système introduit par Gordon Plotkin, inspiré du système LCF de Milner-Scott.

Ressemble au système **T**, mais autorise la définition de toutes les fonctions récursives (on a une boucle “while” au lieu de la boucle “for” qui correspond à la récursion primitive de **T**).

Types : ceux de **T**

- ι est un type : le type des entiers
- si σ et τ sont des types alors $\sigma \Rightarrow \tau$ est un type.

Termes, donnés avec leurs règles de typage.

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma} \qquad \frac{}{\Gamma \vdash \underline{k} : \iota} \qquad \frac{\Gamma \vdash s : \iota}{\Gamma \vdash \mathsf{S}(s) : \iota} \qquad \frac{\Gamma \vdash s : \iota}{\Gamma \vdash \mathsf{P}(s) : \iota}$$

$$\frac{\Gamma \vdash s : \iota \qquad \Gamma \vdash t : \sigma \qquad \Gamma \vdash u : \sigma}{\Gamma \vdash \mathsf{If}(s, t, u) : \sigma}$$

$$\frac{\Gamma, x : \sigma \vdash s : \tau}{\Gamma \vdash \lambda x^{\sigma}\, s : \sigma \Rightarrow \tau} \qquad \frac{\Gamma \vdash s : \sigma \Rightarrow \tau \qquad \Gamma \vdash t : \sigma}{\Gamma \vdash s\,(t) : \tau}$$

$$\frac{\Gamma \vdash s : \sigma \Rightarrow \sigma}{\Gamma \vdash \mathsf{Fix}(s) : \sigma}$$

On peut présenter la sémantique opérationnelle par des règles de réécriture comme pour **T**. Mais on peut aussi introduire une *machine à pile*.

Etat de la machine :

$$s \star \pi$$

où s est un terme de PCF (code) et π est une pile (contexte).

La pile

Eléments de pile :

- S (successeur)
- P (prédécessur)
- IF(t, u) où t et u sont des termes (branches de conditionnelle)
- ARG(t) où t est un terme (argument d'une application).

Piles :

- ε, la pile vide
- $e \cdot \pi$ où e est un élément de pile et π est une pile.

Evolution des états. Deux types de règle.

Règles passives. On se contente de procrastiner : on empile

- $\mathsf{S}(s) \star \pi \rightsquigarrow s \star \mathsf{S} \cdot \pi$
- $\mathsf{P}(s) \star \pi \rightsquigarrow s \star \mathsf{P} \cdot \pi$
- $\mathsf{If}(s,t,u) \star \pi \rightsquigarrow s \star \mathsf{IF}(t,u) \cdot \pi$
- $s\,(t) \star \pi \rightsquigarrow s \star \mathsf{ARG}(t) \cdot \pi$
- $\mathsf{Fix}(s) \star \pi \rightsquigarrow s \star \mathsf{ARG}(\mathsf{Fix}(s)) \cdot \pi$

Règles actives. Un jour, il faut quand même calculer : on dépile

- $\underline{k} \star \mathsf{S} \cdot \pi \rightsquigarrow \underline{k+1} \star \pi$
- $\underline{0} \star \mathsf{P} \cdot \pi \rightsquigarrow \underline{0} \star \pi$
- $\underline{k+1} \star \mathsf{P} \cdot \pi \rightsquigarrow \underline{k} \star \pi$
- $\underline{0} \star \mathsf{IF}(t,u) \cdot \pi \rightsquigarrow t \star \pi$
- $\underline{k+1} \star \mathsf{IF}(t,u) \cdot \pi \rightsquigarrow u \star \pi$
- $\lambda x^{\sigma}\, s \star \mathsf{ARG}(t) \cdot \pi \rightsquigarrow s\,[t/x] \star \pi$

Les autres états conduisent à des blocages, mais ils ne peuvent pas se présenter si on part d'un état $s \star \varepsilon$ où $\vdash s : \iota$.

Importance des machines à piles (et à environnements)

Dans une "vraie" machine, on ne ferait pas de substitution, il y aurait des *environnements* et on manipulerait des *clôtures*.

- Essentielles en réalisabilité classique (Krivine).
- Interprétation calculatoire de la logique classique (Griffin, puis $\lambda\mu$-calcul de Parigot et ses variantes : Curien, Herbelin. . .).
- L'implémentation de Ocaml est basée sur une machine à environnement (Leroy).
- Lien avec la sémantique des *jeux* (Danos, Herbelin, Regnier).
- Lien avec les *collapsible pushdown automata* (Salvati, Walukiewicz).

Bien sûr, PCF est Turing-complet :

Théorème

Pour toute fonction récursive partielle $f : \mathbb{N} \to \mathbb{N}$ il existe un terme s de PCF tel que $\vdash s : \iota \Rightarrow \iota$ et tel que, pour tout entier n :

- *si $f(n)$ est défini et a valeur k, alors $s \star \underline{n} \cdot \varepsilon \rightsquigarrow^* \underline{k} \cdot \varepsilon$*
- *sinon, la réduction de $s \star \underline{n} \cdot \varepsilon$ ne termine pas.*

Mais ce qui est plus intéressant, c'est qu'il admet de nombreuses *sémantiques dénotationnelles*.

Idée de la sémantique dénotationnelle

Associer à chaque type σ un objet, c'est-à-dire un ensemble $[\sigma]$ muni d'une structure adaptée.

A chaque terme s tel que

$$x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash s : \sigma$$

associer un morphisme

$$[s] : [\sigma_1] \times \cdots \times [\sigma_n] \to [\sigma]$$

Catégories

Le bon langage pour décrire ce genre de situation est celui des *catégories* (gros monoïdes éclatés) :

- classe d'objets et ensemble de "flèches" (morphismes) de X vers Y pour toute paire d'objets (X, Y)
- morphismes identité et opération de composition des morphismes
- certaines équations (comme un monoïde : associativité de la composition, neutralité des identités).

Catégories cartésiennes fermées

On a vu qu'on a besoin d'un produit (cartésien) "$\times$", et il faut aussi une opération $\Rightarrow$ sur les objets de la catégorie pour représenter l'opération $\Rightarrow$ sur les types.

Ce genre de catégorie est bien connu des mathématiciens : c'est une catégorie cartésienne fermée (CCC).

Modèle relationnel de PCF

Exemple de CCC et modèle de PCF : $\mathrm{Rel}_!$

$\mathcal{M}_{\mathrm{fin}}(X)$: les muti-ensembles finis sur X, notés $[a_1, \ldots, a_n]$

- objets : les ensembles
- morphismes : $\mathrm{Rel}_!(X, Y) = \mathcal{P}(\mathcal{M}_{\mathrm{fin}}(X) \times Y)$
- $\mathrm{Id}_X = \{([a], a) \mid a \in X)\}$
- si $f \in \mathrm{Rel}_!(X, Y)$ et $g \in \mathrm{Rel}_!(Y, Z)$, alors $g \circ f \in \mathrm{Rel}_!(X, Z)$ est défini par

$$g \circ f = \{(m_1 + \cdots + m_k, c) \mid \exists b_1, \ldots, b_k \in Y \\ ([b_1, \ldots, b_k], c) \in t \text{ et } \forall i \ (m_i, b_i) \in s\}$$

C'est une catégorie cartésienne fermée, avec $X \uplus Y$ comme produit cartésien et $X \Rightarrow Y = \mathcal{M}_{\mathrm{fin}}(X) \times Y$.

On interprète le type ι par $\mathbb{N}$.

Le successeur par $\{([n], n+1) \mid n \in \mathbb{N}\} \in \mathsf{Rel}_!(\mathbb{N}, \mathbb{N})$.

Etc.

Tout terme de PCF peut être interprété. En particulier,

$$\text{si} \quad \vdash s : \iota \quad \text{alors} \quad [s] \subseteq \mathbb{N}.$$

Exemple d'interprétation de terme

On peut voir les points du modèles comme des arbres d'exploration non déterministes des programmes.

Si $s = \lambda x^{\iota}\, \mathsf{If}(x, \mathsf{If}(x, \underline{0}, \underline{1}), \underline{2}) : \iota \Rightarrow \iota$

Alors

$$[s] = \{([0,0],0)\} \cup \{([0,n+1],1) \mid n \in \mathbb{N}\} \cup \{([n+1],2) \mid n \in \mathbb{N}\}$$

Intuitions sur le modèle relationnel

- Eléments de X, objet de $\mathrm{Rel}_!$: indéterminées (de séries entières) ou générateurs indépendants (d'espace vectoriel).
- Eléments de $\mathcal{P}(X)$: vecteurs à coefficients booléens.
- Eléments de $\mathcal{M}_{\mathrm{fin}}(X)$: monômes à indéterminées dans X.
- Eléments de $\mathrm{Rel}_!(X, Y) = \mathcal{P}(\mathcal{M}_{\mathrm{fin}}(X) \times Y)$: séries entières à plusieurs indéterminées $\in X$, et à valeur dans un espace engendré par Y, à coefficients booléens.

Morphismes comme séries entières

- La loi de composition qu'on a donnée correspond exactement à celle des séries entières.
- Les morphismes, et aussi les programmes de PCF, peuvent être décomposés en *série de Taylor* d'une façon qui est compatible avec cette interprétation du modèle à base de séries entières.
- Les exposant comptent des nombres d'étapes de calcul dans la machine de Krivine.

On montre assez facilement que

$$\text{si} \quad \vdash s : \iota \quad \text{et} \quad s \star \varepsilon \rightsquigarrow^* \underline{n} \star \varepsilon, \qquad \text{alors} \quad [s] = \{n\}.$$

La réciproque est aussi vraie.

Théorème

Si $n \in [s]$ *alors* $s \star \varepsilon \rightsquigarrow^* \underline{n} \star \varepsilon$.

Nettement moins facile.

C'est un théorème de normalisation, il se prouve par réductibilité, il y a aussi une preuve combinatoire qui utilise des développements de Taylor.

Logique linéaire

Ce modèle relationnel sert de base à beaucoup d'autres, dont certains ont de vrais coefficients.

Il se décrit mieux dans le langage de la *logique linéaire* LL.

LL introduit la construction de type $!\sigma$ qui est interpétée dans le modèle par l'opération $X \mapsto \mathcal{M}_{\text{fin}}(X)$, une comonade.

Elle oblige à introduire 2 types de conjonctions :

- $X \,\&\, Y = X \uplus Y$, similaire au produit direct d'espaces vectoriels
- $X \otimes Y = X \times Y$, similaire au produit tensoriel d'espaces vectoriels.

Intérêt des modèles dénotationnels

- Permettent de prouver, *de façon modulaire*, des propriétés des programmes.
- Donnent un point de vue intrinsèque sur la syntaxe, qui est pleine de choix totalement arbitraires.
- Suggèrent des extensions de la syntaxe.

Conclusion

On a fait un petit parcours parmi divers thèmes familiers aux membres de PPS. Au passage, on a vu plusieurs liens possibles avec des thèmes du LIAFA :

- la vérification (certification formelle de programmes)
- les automates (automates à piles et machines à environnement pour PCF)
- la combinatoire (importance des séries formelles et des développements de Taylor).

Beaucoup de sujets importants n'ont même pas été évoqués.

A suivre donc !