Preuves Interactives et Applications

Christine Paulin & Burkhart Wolff
http://www.lri.fr/~paulin/PreuvesInteractives

Université Paris-Saclay

Master Informatique FIIL, 2016–17

Objectifs du cours

- Comprendre et savoir utiliser des *assistants de preuves interactives*
  - Langage de description d'objets mathématiques et de propriétés
  - Environnement pour construire interactivement des preuves de propriétés
    - facilité de *notation*
    - *bibliothèques* réutilisables
    - des procédures de preuves *automatiques*
    - des outils (*tactiques*) de décomposition de preuve (analyse par cas, récurrence)
    - un noyau de vérification *sûr*
- Découvrir deux systèmes importants
  - *Isabelle/HOL* : `https://isabelle.in.tum.de/`
  - *Coq* : `http://coq.inria.fr`

Déroulement du cours

- séances de cours/TD/TP
- projet à réaliser
- page Web du cours
  `https://www.lri.fr/~paulin/PreuvesInteractives`

Plan du cours

1. (CP) Introduction, motivations, lambda-calcul simplement typé.
2. (BW) Logique d'ordre supérieur, représentation des termes et des formules en Coq et Isabelle/HOL, preuves élémentaires.
3. (CP) Définitions inductives de types de données et de relations, définitions récursives et preuves par récurrence, représentation en Coq et Isabelle/HOL.
4. (BW) Exemple de modélisation en Isabelle/HOL.
5. (CP) Exemple de modélisation en Coq.
6. (BW) Techniques de démonstration automatique, cas de la réécriture (système de réécriture, unification, confluence et terminaison), mise en œuvre dans les assistants de preuve.
7. (CP & BW) Applications avancées.

Cours 1

Démarche

- Objectif : contrôler le comportement des systèmes informatiques
- Utiliser des outils mathématiques
  - modélisation (abstraction)
    - machine, programme,
    - environnement,
    - comportement attendu
  - méthodes de construction de programmes
  - validation
    - simulation
    - preuves de correction
- Outils informatiques pour faciliter les traitements mathématiques
  - valider les modèles et les méthodes
  - construire des preuves complexes

Ordinateur, calcul et preuve

- Les ordinateurs effectuent des calculs sur des objets binaires.
- Les mathématiques définissent des notions et établissent la vérité d'énoncés via du calcul et des preuves.
- Peut-on représenter une notion mathématique comme un objet binaire ?
- Peut-on calculer si un énoncé est vrai ?
- Peut-on vérifier qu'une preuve est correcte ?

Motivations

Pourquoi s'intéresser aux assistants de preuve ?

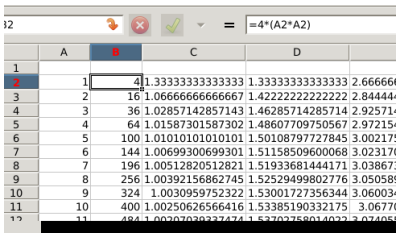
- Limitations liées à l'indécidabilité et à la complexité
- Gérer des preuves *infaisables* à la main
- Transformer les objets mathématiques en objets informatiques
- Intégrer raisonnement et calcul
- Maximiser la confiance

Emergence d'une nouvelle activité
Génie/Ingénierie des preuves

Outils pour le calcul

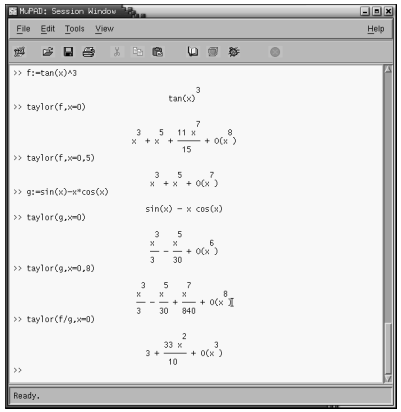
calculatrice

opérations de base



feuille de calcul

présentation, édition

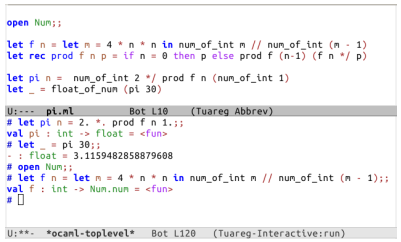


calcul formel

langage spécialisé,

fonctions avancées,

interface



programmation

langage universel,

bibliothèques, spécialistes

problème : modéliser, combiner les méthodes

Calcul et Preuve

Calcul	Assistant de preuve
Calculatrice	Noyau de vérification
Feuille de Calcul	Edition des preuves, tactiques
Calcul formel	Bibliothèques spécialisées
Programmation	Extension via plugin

Les machines font mieux les calculs que nous

Elles peuvent aussi nous aider à construire/vérifier des preuves

Quelques exemples d'applications

- Programmes prouvés
  - SEL4 (Isabelle/HOL, NICTA), micro-noyau sécurisé de système d'exploitation
  - Compcert (Coq, Inria), compilateur C optimisant
- Sécurité : modélisation des plateformes JavaCard en Isabelle/HOL et Coq
- Mathématiques : théorème 4 couleurs, conjecture Kepler, Feit-Thompson. . .
- Preuves formelles en informatique
  - arithmétique des ordinateurs (nombres flottants)
  - cryptographie, combinatoire
  - sémantique des langages de programmation
- Back-end pour d'autres environnements
  - revérification de traces,
  - résolution d'obligations de preuve,
  - couverture de tests

Historique assistants de preuve

Automath Automating Mathematics, Nicolaas G. de Bruijn, Eindhoven, 1967

Mizar (théorie des ensembles) Andrzej Trybulec, University of Białystok, 1973

LCF Logic for Computable Functions, Robin Milner, Edimbourg et Stanford, 1972

HOL (HOL4, HOL-light) Higher-Order Logic, Mike Gordon, Cambridge, 1988

Isabelle (/HOL, /ZF) Larry Paulson, Cambridge, 1989 (Tobias Nipkow, Münich)

Theorie des types de Martin-Löf 1980

- NuPrl : Bob Constable, Cornell U, 1984
- Agda et ses prédecesseurs : Thierry Coquand, Chalmers U, Göteborg, 1990

Coq Gérard Huet et Thierry Coquand, Inria Paris, 1984 (Epigram, OTT, Matita)

ACL2 A Computational Logic for Applicative Common Lisp, Robert S. Boyer et J Strother Moore, 1990 (NqThm, 1971)

PVS Prototype Verification System, Sam Owre, Shankar et John Rushby, SRI, Stanford, 1992

Coq versus Isabelle/HOL

- Deux systèmes matures, applications variées
  - multi-plateformes
  - commandes : `Isabelle2015`, `coqide`
- Expertise locale (BW Isabelle, CP Coq)
- Un large sous-ensemble commun pour la modélisation
  - logique d'ordre supérieur
  - définitions inductives
  - tactiques
- Des choix différents
  - logique classique versus intuitionniste
  - objets définissables, systèmes de types
  - choix architecturaux

Enjeux actuels

Situation

- une technologie qui se développe depuis plus de 30 ans
- des applications phares qui concernent des communautés variées
- un besoin général d'assistance dans les preuves
- des outils qui nécessitent une certaine expertise et du temps

Recherches actuelles

- Evolution des langages (expressivité, style)
- Environnements
  - langages de tactique
  - bibliothèques
  - parallélisation
- Preuves automatiques et sûres

Débouchés industriels

- Trusted Logic/Labs $\longrightarrow$ Gemalto
- ClearSy (atelier B)
- Adacore
- TrustInSoft
- Internet of trust
- Prove & Run
- SafeRiver
- Edukera
- ...

Activités

- Production de logiciels critiques
  - cartes à puce, aéronautique, ferroviaire ...
  - automobiles, santé, énergie ...
- Conception d'outils

Cours 1

Logique du premier ordre

- Termes ($t, u, \ldots$)
  - variables ($x, y, \ldots$)
  - signature : constantes et symboles de fonction d'arité fixée

$$x \qquad c \qquad f(t_1, \ldots, t_n)$$

- Formules logiques

$$t = u \qquad \bot \qquad A \Rightarrow B \qquad \forall x, A$$

- Autres connecteurs

$$\begin{array}{llll} \neg A & \equiv A \Rightarrow \bot & \exists x, A & \equiv \neg \forall x, \neg A \\ A \vee B & \equiv \neg A \Rightarrow B & A \wedge B & \equiv \neg (A \Rightarrow \neg B) \end{array}$$

- Symboles de prédicat avec arité pour des concepts « atomiques »

$$P(t_1, \ldots, t_n)$$

- Règles de précédence

$$\begin{array}{llll} \neg A \wedge B & = (\neg A) \wedge B & A \vee B \wedge C & = A \vee (B \wedge C) \\ A \vee B \Rightarrow C & = (A \vee B) \Rightarrow C & A \Rightarrow B \Rightarrow C & = A \Rightarrow (B \Rightarrow C) \\ \forall x, A \Rightarrow B & = \forall x, (A \Rightarrow B) & \exists x, A \Rightarrow B & = \exists x, (A \Rightarrow B) \end{array}$$

Exemples

- théorie des paires :
  - signature : `pair` (arité 2) et `fst`, `snd` (arité 1)
  - axiomes : $\forall x\, y, \mathtt{fst}(\mathtt{pair}(x, y)) = x \quad \forall x\, y, \mathtt{snd}(\mathtt{pair}(x, y)) = y$
- x est un multiple de 3
  - signature : constante 3 et fonction $\times$ (arité 2)
  - formule : $\exists y, x = 3 \times y$
- x est la somme de deux carrés
- notions atomiques :
  - x a les yeux bleus : $\mathtt{couleur\text{-}yeux}(x) = \mathtt{bleu} \quad \mathtt{yeux\text{-}bleus}(x)$
  - x est l'ami de y : $\mathtt{ami}(x, y)$

Preuves sous hypothèses (séquents)

$$\underbrace{A_1, \ldots, A_n}_{\text{hypothèses}} \quad \underbrace{\vdash}_{\text{déduction}} \quad \underbrace{B}_{\text{conclusion}}$$

- Notation Γ, Δ pour les hypothèses (ensemble de formules)
- Validité : dans toute situation qui rend les hypothèses vraies, la conclusion est également vraie
- Equivalent à $A_1 \wedge \ldots \wedge A_n \Rightarrow B$ ou $A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow B$
- Exemples (dire si les séquents suivants sont vrais)
  - $A \Rightarrow B, A \vdash B$,
  - $A \Rightarrow B, B \vdash A$
  - $A \Rightarrow B, B \vdash \neg A$

Logique du premier ordre sortée

- La logique *sortée* considère plusieurs univers séparés pour les objets
- Les sortes sont des symboles (en nombre fini) comme `nat`, `bool`...
- Les symboles de constantes, de fonctions ou de prédicats sont déclarés avec une arité qui donne les sortes des arguments et du résultat.
- Les variables dans les quantifications sont annotées par leur sorte
- Exemple (yeux bleux) :
  - sortes : individus et couleurs ι, c
  - constantes et fonctions $\texttt{bleu} : c$, $\texttt{couleur-yeux} : [\iota] \to c$
  - prédicat $=$ a pour arité $[c; c]$
  - il existe une personne aux yeux bleus

$$\exists x : \iota, \texttt{couleur-yeux}(x) = \texttt{bleu}$$

- Les sortes évitent l'usage de prédicats « ensemblistes » comme `est-couleur` ou `est-individu`
- Les sortes ne permettent pas de représenter n'importe quel ensemble : l'appartenance à la sorte est une propriété syntaxique de l'objet

Programmation fonctionnelle

- utiliser des fonctions au sens mathématique pour représenter des calculs
  - abbréviation : $f(x) = x^2 + x + 1$
  - définition récursive : $f(0) = 1 \quad f(n+1) = (n+1) \times f(n)$
  - calculer : $f(3)$
- les fonctions sont des objets comme les autres
  - fonction d'ordre supérieur : $f \circ g$ définie par $(f \circ g)(x) = f(g(x))$
- raisonnement mathématique sur les fonctions

Mélange logique et programmes

Il n'est pas immédiat de raisonner sur des programmes quelconques

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

    $(x{+}{+} == x{+}{+})$ `random == random`
- problème de terminaison $f(x) = \textbf{if } f(x) = 0 \textbf{ then } 1 \textbf{ else } 0$

  on prouve $f(x) = 0 \Leftrightarrow f(x) = 1$, d'où $0 = 1$

Cours 1

Lambda-calcul

- Un formalisme « minimal » pour parler de la notion de fonction, d'application et de calcul
- Introduit par Alonzo Church (1930)
- Base des langages de programmation fonctionnels
- Base des langages logiques d'ordre supérieur

Définitions des termes

Termes $t \in \Lambda$

- Ensemble de variables : $x \in \mathcal{V}$
- Ensemble de constantes : $c \in \mathcal{C}$
- Application : $(t\ t')$
- Abstraction : $\lambda x.t$
  - $f(x) = x^2$ versus $f \stackrel{\text{def}}{=} \lambda x.x^2$

$$\begin{array}{ll} \lambda x\ y.t & \stackrel{\text{def}}{=} \lambda x.\lambda y.t \\ t\ u\ v & \stackrel{\text{def}}{=} ((t\ u)\ v) \\ t + u & \stackrel{\text{def}}{=} (" + "\ t\ u) \end{array}$$

Exemples

$$\begin{array}{ll} \lambda x.x & f(x) \stackrel{\text{def}}{=} x \\ \lambda x.\lambda y.x & f(x,y) \stackrel{\text{def}}{=} x \\ \lambda f.\lambda x.f\ (f\ x) & F(f,x) \stackrel{\text{def}}{=} f(f(x)) \\ \lambda f.\lambda g.\lambda x.(f\ (g\ x)) & F(f,g,x) \stackrel{\text{def}}{=} f(g(x)) \\ \lambda x.(x\ x) & f(x) \stackrel{\text{def}}{=} ??? \end{array}$$

Renommage

Variables libres et liées

- toutes les occurrences de x dans t sont liées dans $\lambda x.t$
- une occurrence qui n'est pas liée est libre
- définition de l'ensemble $\mathrm{FV}(t)$ des variables libres de t

$$\begin{array}{lll} \mathrm{FV}(x) & = \{x\} & \mathrm{FV}(c) = \emptyset \\ \mathrm{FV}(\lambda x.t) & = \mathrm{FV}(t) \backslash \{x\} & \mathrm{FV}(t\,u) = \mathrm{FV}(t) \cup \mathrm{FV}(u) \end{array}$$

- exercice : $\mathrm{FV}((\lambda x.\lambda y.(x\,y))\ (\lambda x.\lambda z.(y\,z)))$
- *α-equivalence*
  deux termes sont égaux modulo le renommage des variables liées

$$\lambda x.\lambda y.(x\,y) \equiv \lambda t.\lambda u.(t\,u) \equiv \lambda y.\lambda x.(y\,x)$$

Substitution

- $t[x \leftarrow u]$ représente le terme t dans lequel la variable libre x a été remplacée par le terme u
- problèmes dûs aux variables liées ($\lambda x.x = \lambda y.y$)
  - $(\lambda x.x)[x \leftarrow 3] = (\lambda y.y)[x \leftarrow 3] = \lambda x.x$
  - $(\lambda y.x + y)[x \leftarrow u] =? \lambda y.u + y$
    - $(\lambda y.x + y)[x \leftarrow 3] = \lambda y.3 + y$
    - $(\lambda y.x + y)[x \leftarrow y^2] \neq \lambda y.y^2 + y$
    - $(\lambda y.x + y)[x \leftarrow y^2] = (\lambda z.x + z)[x \leftarrow y^2] = (\lambda z.y^2 + z)$
- Définition

$$
\begin{array}{lll}
x[x \leftarrow u] & = u & \\
y[x \leftarrow u] & = y & y \in \mathcal{V}, y \neq x \\
c[x \leftarrow u] & = c & c \in \mathcal{C} \\
(t\ v)[x \leftarrow u] & = (t[x \leftarrow u]\ v[x \leftarrow u]) & \\
(\lambda x.t)[x \leftarrow u] & = \lambda x.t & \\
(\lambda y.t)[x \leftarrow u] & = \lambda y.(t[x \leftarrow u]) & y \notin \mathsf{FV}(u)
\end{array}
$$

Réduction

β-réduction

- Appliquer une fonction $f(x) = t$ à un argument u revient à remplacer le paramètre x par le terme u dans le corps t de la définition.
  - $f(x) = x^2 + x + 1$, $f(3) = 3^2 + 3 + 1$
  - $(\lambda x.x^2 + x + 1)\,3 = 3^2 + 3 + 1$
- Dans le cadre du λ-calcul $(\lambda x.t)\,u \longrightarrow_\beta t[x \leftarrow u]$
- Congruence (le calcul peut avoir lieu n'importe où dans le terme) si $t \longrightarrow_\beta t'$ alors
  - $t\,u \longrightarrow_\beta t'\,u$
  - $u\,t \longrightarrow_\beta u\,t'$
  - $\lambda x.t \longrightarrow_\beta \lambda x.t'$
- Fermeture réflexive-transitive (suite de calculs)
  - $t \longrightarrow_\beta^* t$
  - $t \longrightarrow_\beta^* u$ et $u \longrightarrow_\beta^* v$ alors $t \longrightarrow_\beta^* v$
- β-équivalence ($\equiv$) : fermeture réflexive-transitive et symétrique

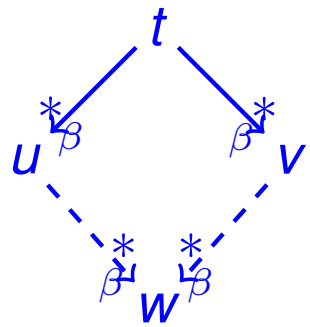
Propriétés

- Certains calculs ne terminent pas

$$\Delta \stackrel{\text{def}}{=} (\lambda x.(x\,x))\,(\lambda x.(x\,x))$$

- Un terme à partir duquel on ne peut plus calculer est dit *en forme normale*
- Les calculs peuuvent toujours être réconciliés (confluence)

  A partir du terme t on effectue deux suites de calculs : l'un vers u et l'un vers v
  alors il est toujours possible de poursuivre les calculs pour revenir vers un même terme.



- Si $t \longrightarrow_\beta^* u$ et u est en forme normale, alors u est appelée la forme normale de t.
- La confluence permet de montrer que la forme normale est unique.

Typage

- Les types représentent des *ensembles de valeurs*, ils permettent de classifier les objets
- Langage pour les types $\alpha \quad c(\tau_1, \ldots, \tau_p) \quad \tau \to \sigma$
  - constantes de type : `nat`, `bool`
  - constructeurs de type : `list(nat)`, `bool` × `nat`
- Relation de typage entre un terme et un type $t : \tau$
- Jugement de typage $x_1 : \tau_1, \ldots, x_p : \tau_p \vdash t : \tau$
- Règles de typage pour établir le jugement
  - Vérifier un jugement de typage (en général décidable)
  - Inférence de type : trouver un type valide pour un programme
- Stabilité du typage lors du calcul (subject-reduction)
- Certains systèmes de types garantissent la terminaison des programmes (même leur complexité)

Règles pour le calcul simplement typé

- τ_c est le type de la constante c
- environnement : $\rho = x_1 : \tau_1, \ldots, x_p : \tau_p$
- règles

$$\frac{}{\rho \vdash c : \tau_c} \qquad \frac{}{x_1 : \tau_1, \ldots, x_p : \tau_p \vdash x_i : \tau_i}$$

$$\frac{\rho, x : \tau \vdash t : \sigma}{\rho \vdash \lambda x.t : \tau \to \sigma} \qquad \frac{\rho \vdash f : \tau \to \sigma \quad \rho \vdash t : \tau}{\rho \vdash f\ t : \sigma}$$

- Inférence
  - utiliser des variables de type et des contraintes d'unification (ML, Isabelle)
  - les constantes peuvent être polymorphes (utilisables dans plusieurs contextes) `[]` : α `list`
  - ajouter le type des variables dans l'abstraction $\lambda x : \tau.t$ (Coq)

Algorithme d'inférence de types

- L'algorithme prend en arguments un environnement ρ et un terme t il calcule un type τ et une substitution s la plus générale telle que $s(\rho) \vdash t : \tau$

$$\frac{}{\rho \vdash c : \tau_c(\alpha_1, \ldots, \alpha_p) \rightsquigarrow \mathtt{id}} \text{si } \alpha_i \notin \rho$$

$$\frac{}{x_1 : \tau_1, \ldots, x_p : \tau_p \vdash x_i : \tau_i \rightsquigarrow \mathtt{id}}$$

$$\frac{\rho, x : \alpha \vdash t : \tau \rightsquigarrow s}{\rho \vdash \lambda x.t : s(\alpha) \to \tau \rightsquigarrow s} \text{si } \alpha \notin \rho$$

$$\frac{\rho \vdash f : \sigma \rightsquigarrow s_1 \qquad s_1(\rho) \vdash t : \tau \rightsquigarrow s_2}{\rho \vdash f\ t : s_3(\alpha) \rightsquigarrow s_3 \circ s_2 \circ s_1} \text{si } s_3 \circ s_2(\sigma) = s_3(\tau \to \alpha), \alpha \notin \rho$$

Exemples

- $\lambda x.(x + 3)$
- $(\lambda x.x) + 3$
- $\lambda x.x$
- $(\lambda x.x)\,3 \qquad (\lambda x.x)\,(\lambda x.(x + 3))$

Propriétés

- Si $t : \tau(\alpha)$ alors pour tout type σ, on a $t : \tau(\sigma)$
- Tout terme t s'il est typable admet un type le plus général $\tau(\alpha_1, \ldots, \alpha_p)$

$$t : \sigma \quad \Leftrightarrow \quad \exists \sigma_1 \ldots \sigma_p . \sigma = \tau(\sigma_1, \ldots, \sigma_p)$$

- la substitution d'une variable $x : \tau$ par un terme de type τ préserve le typage
- la β-réduction préservent le typage
- tous les termes (simplement) typables sont fortement normalisables (n'importe quelle suite de calculs termine)

Cours 1

Notations

Cours	Coq	Isabelle
	Types	
$T \to U$	$T \to U$ | $T - > U$	$T \Rightarrow U$ | $T => U$
	Termes	
$\lambda x.t$ $t\ u$	**fun** $x \Rightarrow t$ $t\ u$	$\lambda x.t$ | \<*lambda*\> $x.t$ $t\ u$
	Propositions	
	Prop	`bool`
$t = u$	$t = u$	$t = u$
$\bot$	`False`	`False`
$\neg P$	~P | `not` P	$\neg P$ | \<*not*\> P
$P \wedge Q$	P/\Q | `and` $P\ Q$	$P \wedge Q$ | P\<*and*\> Q
$P \vee Q$	P\/Q | `or` $P\ Q$	$P \vee Q$ | P\<*or*\> Q
$P \Rightarrow Q$	$P \to Q$	$P \longrightarrow Q$ | $P - - > Q$
$\forall x, P$	**forall** x, t	$\forall x.t$ | \<*forall*\> $x.t$
$\exists x, P$	**exists** x, t	$\exists x.t$ | \<*exists*\> $x.t$

Commandes

Coq	Isabelle
Vérifier un type	
Check T. (* Set/Type *)	**typ** " T "
Vérifier une formule	
Check T. (* Prop *)	**prop** " T "
Vérifier un terme	
Check t.	**term** " t "
Nommer un objet	
Definition *name* : T := t.	**definition** *name* : : "T" **where** "*name*=t"
Evaluer un terme	
Eval compute in t.	**value** " t "
Commentaires	
(* ... *)	(* ... *)

Isabelle

```
theory Scratch imports Main
begin
typ "bool"
term "¬ a ⟶False ∧b"

typ "nat"
typ "nat ⇒nat ⇒bool"

term "λ f x. f (f x)"
term "λ x. x + 3::nat"

prop "∀ x. x=x"
prop "∃ x. x=x"

value "False ∧True"
value "False ⟶True"

definition double where "double f = (λ x. f (f x))"

typ "'a list"
term "[]"
term "[1,2::nat]"
end
```

Coq

```
Check bool.
Check true.
Check Prop.
Check True.

Variables a b : Prop.
Check (~ a → False /\ b).

Section bool.
Variables a b : bool.
Check (andb (implb (negb a) false) b).
End bool.

Check nat → nat → bool.
```

Coq

```
Check (fun f x ⇒ f (f x)).
Check (fun A f (x:A) ⇒ f (f x)).
Check (fun x ⇒ (x + 3)%nat).

Check (∀ x : nat, x=x).
Check (∀ A (x : A), x=x).
Check (∀ A, (∃ x : A, x=x)).

Eval compute in (False /\ True).
Eval compute in (false && true)%bool.
Eval compute in (false || true)%bool.

Definition double A f := fun x : A ⇒ f (f x).

Require Import List.
Check list.
Check nil.
Check (nil : list nat).
Check 1::2::nil.
```

Cours 1

Logique du premier ordre

- On se donne une signature pour modéliser la réduction sur les λ-termes avec un ensemble *L* et une relation `betas` qui représente la fermeture reflexive et transitive de la β-réduction
- Donnez les formules logiques corrrespondant aux notions introduites en cours
  1. le terme *t* est en forme normale
  2. la relation β satisfait la propriété de confluence
  3. le terme *t* admet une forme normale
  4. il existe des termes qui n'ont pas de forme normale
- En utilisant les squelettes `Beta.v` et `Beta.thy` écrire dans Coq et Isabelle les formules correspondantes

Lambda-calcul

1. Explorer les différentes réductions possibles à partir du terme $t \stackrel{\text{def}}{=} (\lambda x\, y.y)\, \Delta$, que constatez-vous ?
2. Soit le terme $Y_f \stackrel{\text{def}}{=} (\lambda x.f\,(x\,x))(\lambda x.f\,(x\,x))$
   - montrer que Y_f se réduit en $f\ Y_f$.
     On dit que Y_f est un *point-fixe* de f.
   - le terme Y_f peut-il être typé dans le λ-calcul simplement typé ?

Lambda-calcul typé

1. Soit le terme $\lambda f\, x.f\,(n\, f\, x)$
   - dire quelles sont ses variables libres et liées
   - donner son type le plus général
   - ce terme est-il en forme normale ?
2. On note N le type $(\alpha \to \alpha) \to (\alpha \to \alpha)$. Pour tout entier naturel, $n \in \mathbb{N}$, on note $\tilde{n}$ le terme $\lambda f\, x.\underbrace{(f \ldots (f}_{n \text{ fois}}\, x) \ldots)$.
   - donner les termes correspondant à $\tilde{0}, \tilde{1}$ et $\tilde{2}$.
   - montrer que pour tout entier n, on a $\tilde{n} : N$
   - montrer que le terme $\lambda n\, f\, x.f\,(n\, f\, x)$ a pour type $N \to N$ (on construira la dérivation).
     Dans la suite ce terme est noté s
   - quelle est la forme normale du terme $s\,\tilde{n}$?
3. utiliser Isabelle pour vérifier vos résultats

Modélisation

On modélise un système de droit d'accès inspiré de celui des fichiers Unix. On introduit des sortes pour représenter les individus (`ind`), les groupes d'individus (`grpe`), les ressources (fichiers, répertoires) (`res`) et les actions à réaliser (`act`). On introduit deux actions particulières `Lire` et `Ecrire` et des symboles de prédicat :

- `dans`(x, g) : l'individu x est dans le groupe g ;
- `proprio`(x, r) : l'individu x est le propriétaire de la ressource r ;
- `droit`(g, a, r) : le groupe g est autorisé à effectuer l'action a sur la ressource r ;
- `peut`(x, a, r) : l'individu x peut effectuer l'action a sur la ressource r ;

Modélisation : questions

1. Traduire en langage naturel les formules suivantes, donner les types des variables :
   1. $\forall x\, r.\texttt{peut}(x, \texttt{Ecrire}, r) \Rightarrow \texttt{peut}(x, \texttt{Lire}, r)$
   2. $\exists x, \forall a\, r.\texttt{peut}(x, a, r)$
2. Exprimer comme des formules logiques les propriétés suivantes :
   1. Le propriétaire d'une ressource peut effectuer toutes les actions sur cette ressource ;
   2. Toute ressource a un et un seul propriétaire ;
   3. Aucun groupe n'est vide et toute personne appartient à (au moins) un groupe.
   4. Lorsqu'un groupe est autorisé à effectuer une action sur une ressource alors tous les membres de ce groupe peuvent effectuer l'action.
3. Traduire le contexte et les formules précédentes dans Coq et dans Isabelle.