Assistants de Preuve
Programmation avec types dépendants
Extraction

Christine Paulin-Mohring & Jean-Christophe Filliâtre

Université Paris-Sud 11, INRIA

20/01/09

Plan

Introduction

Types dépendants

Extraction
- Introduction
- Distinction `Prop` / `Set`
- Extraction dans le CCI
- Typage des termes extraits
- Autres méthodes d'analyse

Introduction

Coq permet de :

- Définir des programmes fonctionnels
- Définir des propriétés de programmes
- Raisonner sur les programmes

On a remarqué :

- Ecrire des fonctions totales requiert parfois d'ajouter des arguments logiques aux fonctions (préconditions)
- Les schémas de définition et de raisonnement sur les fonctions sont liés.

On va généraliser les types :

- Introduire des types inductifs calculatoires incluant des informations logiques.
- Utiliser les types dépendants pour inclure une post-condition dans le résultat de la fonction.
- Construire en une seule fois la fonction et sa preuve de correction.

Extraction

- Toute preuve de Coq est un lambda-term typé avec lequel on peut calculer.
- Distinction **Prop**/**Type**(**Set**) introduite par l'utilisateur pour séparer ce qui l'intéresse sur le plan du calcul de ce qui est logique.
- Le système de type empêche un calcul de dépendre de manière essentielle de la preuve d'une propriété marquée logique.
- L'extraction efface les preuves de propriétés logiques.
- Traduction vers des programmes ML qui peuvent être exécutés efficacement, intégrés dans des environnements complexes.

Type sous-ensemble

```
Inductive sig (A : Type) (P : A → Prop) : Type :=
    exist : ∀x : A, P x → sig P
```

On écrit $\{x : A|Q\}$ pour `sig` **fun** $x \Rightarrow Q$.

- Une preuve de $\{x : A|P\}$ est un couple (a, p) formé d'un objet $a : A$ et d'une preuve $p : P[x := a]$.
- A l'extraction, seul l'objet a est conservé.
- Le type sous-ensemble est le type dépendant de base du système PVS (sous-typage et type-checking conditions).

Application

Une fonction de type $A \rightarrow B$ avec une précondition P et une post-condition Q sera spécifiée comme un objet de type :

$$\forall x : A, P\, x \rightarrow \{y : B | Q\, x\, y\}$$

- A l'extraction, on retrouve une fonction de type $A \rightarrow B$.
- Tout terme de ce type est un programme correct (mais pas forcément efficace . . .)

Exemple

```
Lemma insert :
   ∀A a l, sorted l →
   {m : list A | sorted m ∧ ∀k, In k m ↔ In k l ∨ k=a}.
```

Types sommes

Test booléen

Spécification de fonctions booléennes.

```
Inductive sumbool (A : Prop) (B : Prop) : Set :=
    left : A → {A} + {B} | right : B → {A} + {B}
```

objet booléen :

- `left` (correspond à `true`) est associé à une preuve de *A*,
- `right` (correspond à `false`) est associé à une preuve de *B*.

Types sommes

Application du test booléen

Avec $P : A \rightarrow \mathbf{Prop}$, un terme p de type $\forall x, \{P\,x\} + \{\neg(P\,x)\}$ est une preuve de décidabilité de P qui correspond au test

Si $P\,x$ alors ... sinon ...

En Coq on pourra écrire ce programme :

```
match p x with left H ⇒ ... | right H ⇒ ... end
```

et utiliser l'hypothèse H de type $P\,x$ (resp. $\neg(P\,x)$) dans la branche.
Si ces hypothèses ne sont pas utiles à la construction du résultat, on écrit plus simplement :

```
if p x then ... else ...
```

L'analyse par cas sur ce programme fera apparaître les hypothèses $P\,x$ et $\neg(P\,x)$ dans chaque branche, ce qui facilite le raisonnement.

Exemple

```
Module Type Order.
Variable A:Type.
Variable le:A→A→Prop.
Hypothesis le_refl : ∀x, le x x.
Hypothesis le_trans : ∀x y z, le x y → le y z → le x z.
Variable le_total : ∀x y, {le x y}+{le y x}.
End Order.

Module SortedList (X:Order).
Export X.
Definition le_list a l := ∀k, In k l → le a k.
Inductive sorted : list A → Prop :=
  sort_nil : sorted nil
| sort_cons : ∀a l, le_list a l → sorted l → sorted (a::l)
Lemma sort_singl : ∀a, sorted (a::nil).
Lemma sort_cons_cons :
   ∀a b l, le a b → sorted (b::l) → sorted (a::b::l).
```

Fonction d'insertion

On peut construire une fonction d'insertion correcte par construction.

```
Definition insert0 : ∀a l, sorted l →
   {m : list A | sorted m ∧ ∀k, In k m ↔ In k l ∨ k=a}.
induction l.
∃(a::nil); simpl; intuition.
intro; case (le_total a a0); intro.
∃(a::a0::l); simpl; intuition.
case IHl.
inversion H; trivial.
intros m (Sm,Em).
∃(a0::m); split; intros.
apply sort_cons; auto.
...
Defined.
```

Types sommes

Exception

Un autre cas de type somme combine un objet calculatoire et une information non calculatoire :

```
Inductive sumor (A : Type) (B : Prop) : Type :=
    inleft : A → A + {B} | inright : B → A + {B}
```

correspond à un type option qui est soit une valeur correspondant à la spécification *A* soit un cas particulier dans le cas où *B* est vrai.

Ce schéma est utilisé pour modéliser des cas exceptionnels.
Par exemple, un programme de recherche d'un élément vérifiant une condition *P* dans une liste pourra avoir comme spécification :

```
∀l, {a:A | In a l ∧ P a}+{∀a, In a l → ¬ (P a)}
```

Paire dépendante

```
Inductive sigT (A : Type) (P : A → Type) : Type :=
    existT : ∀x : A, P x → sigT P.
```

Noté $\{x : A \& P\,x\}$.
Utilisé pour spécifier des fonctions qui calculent des produits.

```
Lemma exists_last:
  ∀(A : Type) (l : list A),
  l <> nil → {l' : list A & {a : A | l = l' ++ a :: nil}}
```

Autre type dépendant : ensemble fini

```
Variable A : Type.
Definition set := A → Prop.
Definition emptyset : set := fun x ⇒ False.
Definition add (a:A) (P:set) : set := fun x ⇒ x=a ∨ P x.
Definition equiv (P Q : set) := ∀x, P x ↔ Q x.
```

Definition inductive calculatoire de *P* est un ensemble fini :

```
Inductive finite (P : set) : Type :=
   fin0 : equiv P emptyset → finite P
 | finS : ∀a Q, finite Q → ¬Q a → equiv P (add a Q)
          → finite P.
```

Application au calcul de la taille d'un ensemble fini :

```
Fixpoint size P (F:finite P) {struct F} : nat :=
  match F with fin0 _ ⇒ 0
             | finS a Q FQ _ _ ⇒ S (size Q FQ) end.
```

Autre type dépendant : fermeture reflexive-transitive

```
Inductive transR (x:A) : A → Type :=
    trans_refl : transR x x
  | trans_step : ∀y z, transR x y → R y z → transR x z.
```

Une preuve de (`transR` x y) représente un chemin de x à y.

Difficultés avec les types dépendants

- Les familles inductives rendent le problème de complétude du filtrage complexe.
  On peut coder un problème de Post comme un problème de complétude de filtrage (N. Oury) pour des familles inductives simples.
- L'égalité intensionnelle sur les objets contenant des preuves est trop forte (besoin de proof-irrelevance, Barras & Bernardo)
- L'égalité de Leibniz est trop faible en présence de dépendance (égalité hétérogène)
- La construction de programme par preuve rend difficile le contrôle de l'efficacité de l'algorithme.

Program

- Thèse de M. Sozeau, implantée dans Coq.
- Langage de description de termes de preuves Coq à la ML.
- Un type sous-ensemble et une notion de sous-typage vue comme une coercion.
- Génération d'obligations de preuve.
- Extension à des types plus généraux.
- Exemple conséquent des finger trees.

Program - example

```
Program Fixpoint
  insert(a:A)(l:list A)(_:unit|sorted l) {struct l}
  : {m| sorted m ∧ ∀k, In k m ↔ In k l ∨ k=a}
  := match l with nil ⇒ (a::nil)
               | (b::m) ⇒ if le_total a b then (a::l)
                           else (b::insert a m tt) end.
Next Obligation.
intuition....
Defined.
```

Autres systèmes

Les types dépendants sont un sujet de recherche actif dans la communauté de la programmation fonctionnelle

- Extensions de Ocaml, Haskell avec des types de données généralisés (GADT, indexés par des types)
- Dépendances indexées par des entiers avec des contraintes simples : DML (Xi, Boston)
- Ajout d'assertions dans un langage fonctionnel pur : Pangolin (Régis-Gianas, Rocquencourt)

Théories des types pour la programmation fonctionnelle

- Agda (Coquand, Chalmers)
- Epigram (Altenkirch, Notthingham)

Principe

- Obtenir des programmes ML à partir des termes de preuves Coq
- Deux étapes :
  - Suppression des parties non calculatoires $p : P :$ **Prop**
  - Traduction vers Ocaml ou Haskell

Une nouvelle notion d'extraction [Letouzey 2002] implantée dans Coq 8.0

Difficultés

- L'utilisateur doit controler ce qui est calculatoire ou non.
- La cumulativité des univers (**Prop**,**Set** :**Type**) rend la modularité difficile.
- Différences entre le typage de Coq et de ML (polymorphisme, types dépendants).
- Possibilité d'extraire des programmes qui ne terminent pas ou qui sont inefficaces

Distinction Prop / Set : difficultés

une suppression brutale de tous les termes dans Prop conduit à des programmes erronés

```
Definition pred (n:nat) : n<>0 → nat :=
  match n return n<>0 → nat with
  | O ⇒ fun h ⇒ False_rec nat (h (refl_equal O))
  | S p ⇒ fun _ ⇒ p
  end.

Extraction pred.

(** val pred : nat → nat **)

let pred = function
  | O → assert false (* absurd case *)
  | S p → p
```

Mais alors quid de...

```
Definition predO := pred 0.    (* de type O<>O → nat *)

Extraction predO.
```

le code extrait

```
let predO = pred O    (* de type nat *)
```

produirait une erreur à l'exécution
de même on pourrait construire

- d'autres types d'erreurs à l'aide d'`eq_rec`
- des évaluations qui ne terminent pas avec `Acc_rec`

Solution

conserver les termes de sorte `Prop` sous une *forme dégénérée*

```
Definition predO := pred 0.

Extraction predO.

(** val predO : __ → nat **)

let predO _ = pred O
```

Complications

il existe des entorses à la règle « on ne peut construire quelque chose d'informatif à partir de quelque chose de logique »

- *inductif vide*

$$\frac{p : \texttt{False} : \texttt{Prop} \quad T : \texttt{Type}}{\texttt{case}(p, T, \emptyset) : T}$$

- *inductif singleton logique* (un seul constructeur ayant uniquement des arguments logiques)

$$\frac{p : x = y : \texttt{Prop} \quad q : P\ x : \texttt{Type}}{\texttt{case}(p, P, q) : P\ y : \mathit{Type}}$$

- la *garde* d'un point fixe informatif peut être un *inductif logique*

rappelez-vous `Acc_rec`

Extraction dans le CCI

$$\mathrm{CCI}_{\Box} \quad \begin{array}{rl} t ::= & \Box \mid s \mid x \mid c \mid C \mid I \\ \mid & \forall x : t,\ t \mid \lambda x : t,\ t \mid \texttt{let}\ x := t\ \texttt{in}\ t \mid t\ t \\ \mid & \texttt{case}(t, t, t, \ldots, t) \\ \mid & \texttt{fix}\ x_i\ \{x_1/k_1 : t := t, \ldots, x_n/k_n : t := t\} \end{array}$$

extraction vers un $\mathrm{CCI}_{\Box}$ *non typé* augmenté d'une constante spéciale $\Box$
arité : terme dont la forme normale est $\forall x_1 : X_1, \ldots \forall x_n : X_n, s$ avec s une sorte
schéma de types : terme dont le type est une arité

Fonction d'extraction

fonction d'extraction $\mathcal{E}$

- $\mathcal{E}(t) = \square$ si t est un schéma de types ou de sorte `Prop`

sinon, par récurrence sur le terme

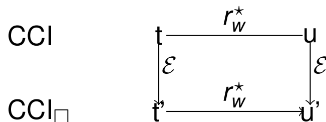
- $\mathcal{E}(x) = x$ si x variable, constante ou constructeur
- $\mathcal{E}(\lambda x : T, t) = \lambda x : \square, \mathcal{E}(t)$
- $\mathcal{E}(\texttt{let } x := t \texttt{ in } u) = \texttt{let } x := \mathcal{E}(t) \texttt{ in } \mathcal{E}(u)$
- $\mathcal{E}(u\ v) = \mathcal{E}(u)\ \mathcal{E}(v)$
- $\mathcal{E}(\texttt{case}(e, P, f_1, \ldots, f_n)) = \texttt{case}(\mathcal{E}(e), \square, \mathcal{E}(f_1), \ldots, \mathcal{E}(f_n))$
- $\mathcal{E}(\texttt{fix } f_i\ \{f_1/k_1 : A_1 := t_1, \ldots, f_n/k_n : A_n := t_n\})$
  $= \texttt{fix } f_i\ \{f_1/k_1 : \square := \mathcal{E}(t_1), \ldots, f_n/k_n : \square := \mathcal{E}(t_n)\}$

Réduction des termes extraits

dans $\mathrm{CCI}_\square$ on ajoute les réductions suivantes

- $\square\ u \rightarrow \square$
- $\texttt{case}_n(\square, P, f) \rightarrow f\ \underbrace{\square \cdots \square}_{n}$
  lorsqu'il s'agissait d'un inductif singleton avec n arguments à son constructeur
- $\texttt{fix}\ f_i\ \{F\}\ u_1 \ldots u_{k_i} \rightarrow t_i[f_j \leftarrow \texttt{fix}\ f_j\ \{F\}]_{\forall j}\ u_1 \ldots u_{k_i}$
  lorsque $u_{k_i} = \square$

alors on a la *bisimulation*



Correction de l'extraction

on montre que l'extraction de $t : T$ est un terme qui « vérifie le type T » par une notion de réalisabilité, i.e.

$$\mathcal{E}(t) \text{ r } T$$

voir [Letouzey 2002] pour les détails

Vers Ocaml et Haskell

les stratégies de réductions d'Ocaml et Haskell sont *faibles* (pas de réduction sous les λ) ; restent à régler

- élimination sur un *inductif vide*
  c'est du code qui ne sera jamais atteint $\Rightarrow$ remplacé par du code arbitraire (`assert false` en Ocaml, `error` en Haskell)
- les *singletons logiques* ; la règle

$$\texttt{case}_n(\Box, P, f) \to f\ \underbrace{\Box \cdots \Box}_{n}$$

  est intégrée à l'extraction

Vers Ocaml et Haskell

- les *arguments de* $\square$ (réduction $\square\ u \rightarrow u$)
  - Ocaml : point-fixe absorbant ses arguments `let rec f x = f`
  - Haskell : évaluation paresseuse $\Rightarrow$ réduction jamais utilisée $\Rightarrow$ $\square$ peut être implanté par un terme arbitraire
- les *gardes logiques* de points fixes
  - Ocaml : aucun problème, la réduction se moque du fait que l'argument de cadre commence par un constructeur
  - Haskell : plus délicat, car le dépliage d'un point fixe se fait sans réduction préalable des arguments

Typage des termes extraits

les « machines » d'Ocaml et d'Haskell conviennent (λ-calcul + inductifs)
mais on souhaite produire du *code source* (lisibilité, confiance, réutilisation, etc.)
problème : les termes extraits ne sont *pas toujours typables* car les systèmes de types d'Ocaml et d'Haskell sont *plus pauvres* que celui de Coq
remarque : on pourrait extraire vers des langages non typés comme Lisp ou Scheme (mais en pratique ce n'est pas fait)

Analyse des problèmes de typage

le système de types de Coq offre des possibilités sans contrepartie en Ocaml ou Haskell :

- filtrage au niveau des types
- points fixes au niveau des types
- polymorphisme non prénexe ou dans les types des constructeurs

Analyse des problèmes de typage : exemple

```
Definition P (b:bool) : Type := if b then nat else bool.

Definition p (b:bool) : P b :=
  match b return P b with
  | true ⇒ O
  | false ⇒ true
  end.
```

le terme extrait

```
let p = function
  | True → O
  | False → True
```

n'est typable ni en Ocaml ni en Haskell

Analyse des problèmes de typage : exemple

fonctions entières à `n` arguments

```
Fixpoint F (n:nat) : Type := match n with
  | O ⇒ nat
  | S n ⇒ nat → F n
  end.

Fixpoint f (n:nat) : F n := match n return F n with
  | O ⇒ O
  | S n ⇒ fun _ ⇒ f n
  end.
```

donne le terme extrait non typable

```
let rec f = function
  | O → O
  | S n → (fun _ → f n)
```

Analyse des problèmes de typage : exemple

types existentiels

```
Inductive any : Type := Any : ∀A:Type, A → any.
```

la solution

```
  type 'a any = Any of 'a
```

n'est pas satisfaisante (une variable de type s'échappe) ; pire encore

```
Inductive anyList : Type :=
  | AnyNil : anyList
  | AnyCons : ∀A:Type, A → anyList → anyList.

Definition l := AnyCons bool true (AnyCons nat O AnyNil).
```

est sans solution Ocaml / Haskell

Analyse des problèmes de typage : exemple

```
Definition distr_pair : (∀X:Type, X → X) → nat * bool :=
  fun f ⇒ (f nat O, f bool true).
```

ici c'est le polymorphisme *prénexe* d'Ocaml / Haskell qui est une limite
(il n'est là que pour rendre l'*inférence* de types décidable)

Correction des erreurs de typage

le typeur peut être « contourné » avec `Obj.magic` en Ocaml et `unsafeCoerce` en Haskell
problème : quand doit-on le faire ?
en pratique, *pas souvent* (jamais dans toute la bibliothèque standard, seulement 4 fois sur 73 contributions recensées des utilisateurs de Coq)

Des types pour les programmes extraits

1. définition d'un type attendu $\hat{\mathcal{E}}(T)$ pour l'extraction de $t : T$ tel que

$$\vdash_{CCI} t : T \quad \Rightarrow \quad \vdash_{ML} \mathcal{E}(t) : \hat{\mathcal{E}}(T)$$

2. on force $\mathcal{E}(t)$ à avoir le type $\hat{\mathcal{E}}(T)$ en insérant des `Obj.magic` / `unsafeCoerce`, mais le moins souvent possible

Extraction des types

- un type Ocaml pour les types Coq que l'on ne peut pas représenter

```
type __ = Obj.t
```

- une définition pour □

```
let __ = let rec f _ = Obj.repr f in Obj.repr f
```

Extraction des types

certains types peuvent être extraits à l'identique

- inductifs

```
Inductive list (A:Type) : Type :=
  nil : list A | cons : A → list A → list A
```

```
type 'a list = Nil | Cons of 'a * 'a list
```

- schémas de types

```
Definition sch : Type → Type := fun X:Type ⇒ X→X.
```

```
type 'x sch = 'x → 'x
```

Extraction des types

mais d'autres seront *approchés*

- inductifs

```
Inductive list2 : ∀A:Type,Type :=
  | nil2 : ∀A:Type, list2 A
  | cons2 : ∀A:Type, A → list2 A → list2 (A * A).
```

```
type 'a list2 = Nil2 | Cons2 of __ * __ list2
```

- schémas

```
Definition sch2 : (bool → Type) → Type :=
  fun (X:bool→Type) ⇒ X true → X false.
```

```
type 'x sch2 = 'x → 'x
```

L'insertion de `Obj.magic`

idée : utiliser un algorithme d'inférence / vérification de types et insérer des `Obj.magic` aux endroits où il aurait échouer
algorithme *W* de Damas-Milner : analyse *ascendante*

$$\mathrm{W} : \texttt{env} \star \texttt{expr} \rightarrow \texttt{type} \star \texttt{subst}$$

algorithme *M* de Lee-Yi : analyse *descendante*

$$\mathrm{M} : \texttt{env} \star \texttt{expr} \star \texttt{type} \rightarrow \texttt{subst}$$

L'algorithme M

$\mathrm{M}(\Gamma, x, \tau)$ =
  $\mathrm{mgu}(\tau, \mathrm{Inst}(\Gamma(x)))$

$\mathrm{M}(\Gamma, \mathrm{c}, \tau)$ =
  $\mathrm{mgu}(\tau, \mathrm{Inst}(\mathrm{type}(c)))$

$\mathrm{M}(\Gamma, \mathrm{fun}\ x \rightarrow a, \tau)$ =
  let $\sigma = \mathrm{mgu}(\tau, \alpha \rightarrow \beta)$ in α, β fraîches
  let $\phi = \mathrm{M}(\Gamma + x : \sigma(\alpha), a, \sigma(\beta))$ in
  $\phi \circ \sigma$

$\mathrm{M}(\Gamma, a_1\ a_2, \tau)$ =
  let $\phi_1 = \mathrm{M}(\Gamma, a_1, \alpha \rightarrow \tau)$ in α fraîche
  let $\phi_2 = \mathrm{M}(\phi_1(\Gamma), a_2, \phi_1(\alpha))$ in
  $\phi_2 \circ \phi_1$

$\mathrm{M}(\Gamma, \mathrm{let}\ x = a_1\ \mathrm{in}\ a_2, \tau)$ =
  let $\phi_1 = \mathrm{M}(\Gamma, a_1, \alpha)$ in α fraîche
  let $\phi_2 = \mathrm{M}(\phi_1(\Gamma) + x : Gen(\phi_1(\alpha), \phi_1(\Gamma)), a_2, \phi_1(\tau))$ in
  $\phi_2 \circ \phi_1$

L'algorithme M modifé

M(Γ, x, τ) =
  let σ = mgu(τ, Inst($\Gamma(x)$)) in
  if σ *= error then (id, Obj.magic* x*) else (*σ*,* x*)*

M(Γ, c, τ) =
  let σ = mgu(τ, Inst(type(c))) in
  if σ *= error then (id, Obj.magic c) else (*σ*, c)*

M(Γ, fun $x \rightarrow a$, τ) =
  let σ = mgu(τ, $\alpha \rightarrow \beta$) in α, β fraîches
  if σ *= error then*
    let (ϕ*,* a'*) = M(*$\Gamma + x : \alpha$*, a,* β*) in (*ϕ*, Obj.magic (fun* $x \rightarrow a'$*))*
  else
    let (ϕ, a') = M($\Gamma + x : \sigma(\alpha)$, a, $\sigma(\beta)$) in ($\phi \circ \sigma$, fun $x \rightarrow a'$)

M(Γ, a_1 a_2, τ) =
  ...inchangé...

M(Γ, let x = a_1 in a_2, τ) =
  ...inchangé...

L'extraction en pratique

- extraction des *modules*
  module (resp. signature) Coq $\longrightarrow$ module (resp. signature) Ocaml
  quelques problèmes de typage rédhibitoires cependant
- extraction des *types co-inductifs*
  - Haskell : trivial grâce à l'évaluation paresseuse
  - Ocaml : utilisation de du module `Lazy`

```
CoInductive stream (A:Type) : Type :=
  Cons : A → stream A → stream A.
```

```
type 'a stream = 'a __stream Lazy.t
and 'a __stream = Cons of 'a * 'a stream
```

+ `Lazy.force` dans les `match` et `lazy` devant les constructeurs

Efficacité du code extrait

l'utilisation quasi systématique des *principes de récurrence* venant avec les types inductifs conduit à un code inefficace dans un langage strict comme Ocaml
comparer les programmes Ocaml

```
let rec exists p = function
    | [] → false
    | x :: l → p x || exists p l
```

et

```
let exists p l = List.fold_left (fun b x → p x || b) false
```

le premier est efficace, le second non (parcourt toujours toute la liste)

Efficacité du code extrait

pour y remédier, l'extraction de Coq *déplie* systématiquement

- tous les principes de récurrence
- les fonctions
  - dont le corps n'est pas trop gros
  - dont certains arguments sont potentiellement non utiles car utilisés sous une seule branche d'un filtrage

on contrôle ce déliage globalement avec la commande

```
Type/Unset Extraction AutoInline.
```

et au cas par cas avec la commande

```
Extraction Inline/NoInline id.
```

Autres méthodes d'analyse

la méthode d'extraction de Coq ne permet pas de supprimer du programme certains arguments informatifs inutiles
ainsi, le constructeur `cons` des listes de longueur *n* a pour type

```
cons : (n:nat) A → list n → list (S n)
```

ce qui donne dans le code extrait

```
cons : nat → A → list → list
```

i.e. l'entier représentant la longueur de la liste est conservé

Autres méthodes d’analyse

deux solutions possibles

- déplacer les marques des sortes (`Prop`/`Type`) vers les *quantificateurs* [Takayama, Hayashi, ...]
  on a alors
  $$\forall x : A, B \qquad \text{et} \qquad \forall_\circ x : A, B$$
  avec
  $$\mathcal{E}(\forall_\circ x : A, B) = \mathcal{E}(B) \qquad \text{et} \qquad f \text{ r } \forall_\circ x : A, B = \forall x : A, f \text{ r } B$$
- analyses de *code mort* [Berardi, Boerio, ...]