Programmer avec Coq: filtrage dépendant et récursion

Matthieu Sozeau, $\pi.r^2$, Inria Paris & IRIF

Séminaire du cours "Programmer = Démontrer?"

12 Décembre 2018

Collège de France

Programmer = Démontrer ?

"En théorie des types, on n'écrit pas de vrais programmes"
- P.E. Dagand

Pourquoi?

Programmer = Démontrer ?

"En théorie des types, on n'écrit pas de vrais programmes"
- P.E. Dagand

Pourquoi? ...et pourquoi pas?

Pourquoi pas de vrais programmes

"Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs"
John Backus, Turing Award Lecture, 1977

"Total Functional Programming"
D.A. Turner, développeur de Miranda, un ancètre d'Haskell

On bannit deux concepts du calcul:

- La partialité: les fonctions sont définies sur tout leur domaine
- La non-terminaison: on ne peut pas boucler indéfiniment sans rien produire

On perd de l'expressivité, mais on gagne le raisonnement équationel familier des mathématiques sur les programmes.

La partialité

Exemple: la fonction “nth” sur les listes.
Programme Caml:

```
val nth : 'a list -> int -> 'a

let rec nth l n =
  match l, n with
  | nil, _ -> assert false
  | cons x _, 0 -> x
  | cons _ xs, n -> nth xs (pred n)

val bug : 'a
let bug = nth [] 0
```

Modéliser la partialité

On peut modéliser `nth` en théorie des types en utilisant:

- Une valeur par défaut de type `'a`
- Un type option (à la `KO`/`OK`) ou une valeur exceptionelle $\bot$.

Ces deux solutions changent le programme et son sens!

La non-terminaison

En λ-calcul pur, on a un combinateur de point fixe:

$$Y := \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x, f\ (x\ x))$$

tel que:

$$YF \equiv F(YF)$$

- Permet d'encoder toute fonction récursive!
- Essentiel pour la thèse de Church-Turing: toute fonction calculable $f : \mathbb{N} \to \mathbb{N}$ est représentable par un λ-terme ou une machine de Turing.
- Impossible à accepter en théorie des types! (c.f. cours 3)

Programme Caml:

```
let rec fix f x = f (fix f) x
let loop : 'a = fix (fun f x -> f x) 0
```

La récursion

On se restreint à des récursions *structurelles* sur les types inductifs (c.f. cours 3):

- Via leurs éliminateurs seulement dans Lean;
- Avec une *condition de garde* syntaxique dans Coq.
- Avec des *sized-types*, en annotant les types par une notion de taille, en Agda.
- La méthode du carburant/gaz.

Assez contraignant en pratique:

- La terminaison est une propriété difficile à montrer en général
- Elle peut dépendre de la correction même du programe!

La récursion

On se restreint à des récursions *structurelles* sur les types inductifs (c.f. cours 3):

- Via leurs éliminateurs seulement dans Lean;
- Avec une *condition de garde* syntaxique dans Coq.
- Avec des *sized-types*, en annotant les types par une notion de taille, en Agda.
- La méthode du carburant/gaz.

Assez contraignant en pratique:

- La terminaison est une propriété difficile à montrer en général
- Elle peut dépendre de la correction même du programe!

Question: peut-on encore définir toute fonction $f : \mathbb{N} \to \mathbb{N}$ "intéressante" en théorie des types?
Hormis la prouvabilité en théorie des types (incomplétude de Gödel).

Slogan

On peut refléter les fonctions partielles, ou dont la terminaison n'est pas structurelle, en théorie des types dépendants.

Slogan

On peut refléter les fonctions partielles, ou dont la terminaison n'est pas structurelle, en théorie des types dépendants.

Sans modifier essentiellement leur comportement calculatoire!

Idée: mettre de la logique dans les types et les termes.

Une distinction preuves/programmes

`Prop`/`Type` et l'extraction

Preuves et programmes

La distinction `Prop`/`Type` de Coq (Paulin-Mohring, 1989; Letouzey, 2004):

- À rebours de Curry-Howard, on va distinguer les parties logique des parties calculatoires de nos termes.
- Le calcul empêche la dépendance d'une partie calculatoire sur une preuve.

$$p : x < y \ \vee \ x > y \not\vdash \quad \begin{array}{l} \texttt{match}\ p\ \texttt{with} \\ \mid \text{in\_left}\ H \Rightarrow \text{true} \\ \mid \text{in\_right}\ H \Rightarrow \text{false} \\ \texttt{end} : \text{bool} \end{array}$$

- Mais pas la dépendance d'une preuve sur une autre:

$$p : P \vee Q \vdash \ldots : Q \vee P$$

- Principe validé par `Prop`: *indifférence des preuves* (raisonnement classique)

$$\forall (p\ q : A \vee \neg A), p = q$$

Des ponts entre `Prop` et `Type`

La restriction des dépendences des termes informatifs vis-à-vis des preuves n'est pas totale!

Des ponts entre Prop et Type

La restriction des dépendences des termes informatifs vis-à-vis des preuves n'est pas totale!

On peut éliminer à partir d'une proposition P : Prop pour produire un objet calculatoire $t : T$: Type si:

1. P a au plus un constructeur;
2. Tout ses arguments sont des propositions.

Cela inclut True, False, eq, Acc

Intuition: ces propositions valident naturellement l'indifférence aux preuves (elles en ont au plus une possible!)

Des ponts entre Prop et Type

La restriction des dépendences des termes informatifs vis-à-vis des preuves n'est pas totale!

On peut éliminer à partir d'une proposition P : Prop pour produire un objet calculatoire $t : T$: Type si:

1. P a au plus un constructeur;
2. Tout ses arguments sont des propositions.

Cela inclut True, False, eq, Acc

Intuition: ces propositions valident naturellement l'indifférence aux preuves (elles en ont au plus une possible!)

```
Inductive eq {A} (a : A) : A → Prop := eq_refl : eq a a.
Def coerce {A x y} {P : A → Type} (e : eq x y) : P x → P y
:= match e with eq_refl ⇒ fun x ⇒ x end.
```

Extraction

Cette distinction permet l'extraction de programmes.

L'extraction $\mathcal{E}(t)$ produit à partir d'un terme $t : T : \texttt{Type}$ un terme du λ-calcul pur correspondant au contenu calculatoire de t.

- On élimine types et preuves.
- C'est un compilateur pour Coq!

Extraction

Cette distinction permet l'extraction de programmes.

L'extraction $\mathcal{E}(t)$ produit à partir d'un terme $t : T : \texttt{Type}$ un terme du λ-calcul pur correspondant au contenu calculatoire de t.

- On élimine types et preuves.
- C'est un compilateur pour Coq!

Exemples:

$$\mathcal{E}(\lambda(X : \texttt{Type})(x : X).x) = \lambda x.x$$

$$\mathcal{E}(\lambda(m\ n : \mathbb{N})(H : n > 0).\texttt{div}\ m\ n\ H) = \lambda m\ n.\mathcal{E}(\texttt{div})\ m\ n$$

$$\mathcal{E}(\texttt{match}\ e\ \texttt{with}\ \mathsf{eq\_refl} \Rightarrow t\ \texttt{end}) = \mathcal{E}(t)$$

Extraction des types existentiels

Témoin	Prédicat	Type existentiel	Sorte	Introduction
$A : \texttt{Type}$	$P : A \to \texttt{Type}$	$\Sigma x : A.P\ x$	$: \texttt{Type}$	$(x; y)$
$A : \texttt{Type}$	$P : A \to \texttt{Prop}$	$\{x : A \mid P\ x\}$	$: \texttt{Type}$	`exist` $x\ p$
$A : \texttt{Type}$	$P : A \to \texttt{Prop}$	$\exists x : A, P\ x$	$: \texttt{Prop}$	`ex_intro` $x\ p$

Extraction des types existentiels

Témoin	Prédicat	Type existentiel	Sorte	Introduction
$A : \texttt{Type}$	$P : A \to \texttt{Type}$	$\Sigma x : A.P\ x$	$: \texttt{Type}$	$(x; y)$
$A : \texttt{Type}$	$P : A \to \texttt{Prop}$	$\{x : A \mid P\ x\}$	$: \texttt{Type}$	$\texttt{exist}\ x\ p$
$A : \texttt{Type}$	$P : A \to \texttt{Prop}$	$\exists x : A, P\ x$	$: \texttt{Prop}$	$\texttt{ex\_intro}\ x\ p$

En pratique, on remplace $p : P : \texttt{Prop}$ ou $T : \texttt{Type}$ par un "dummy" $\square$.

Exemples:

$$\mathcal{E}(\lambda(n : \mathbb{N})(v : \texttt{vector}\ A\ n).(n; v)) \quad = \quad \lambda(n : \square)(v : \square).(n; v)$$

Extraction des types existentiels

Témoin	Prédicat	Type existentiel	Sorte	Introduction
$A : \texttt{Type}$	$P : A \to \texttt{Type}$	$\Sigma x : A.P\ x$	$: \texttt{Type}$	$(x; y)$
$A : \texttt{Type}$	$P : A \to \texttt{Prop}$	$\{x : A \mid P\ x\}$	$: \texttt{Type}$	$\texttt{exist}\ x\ p$
$A : \texttt{Type}$	$P : A \to \texttt{Prop}$	$\exists x : A, P\ x$	$: \texttt{Prop}$	$\texttt{ex\_intro}\ x\ p$

En pratique, on remplace $p : P : \texttt{Prop}$ ou $T : \texttt{Type}$ par un "dummy" $\square$.

Exemples:

$$\begin{aligned}
\mathcal{E}(\lambda(n : \mathbb{N})(v : \texttt{vector}\ A\ n).(n; v)) &= \lambda(n : \square)(v : \square).(n; v) \\
\mathcal{E}(\lambda(n : \mathbb{N})(H : n > 0).\texttt{exist}\ n\ H) &= \lambda n.\texttt{exist}\ n\ \square \\
\mathcal{E}(\lambda(n : \mathbb{N})(H : n > 0).\texttt{ex_intro}\ n\ H) &= (\lambda n.\square) = \square
\end{aligned}$$

Extraction

Définition: Par contenu calculatoire d'un terme Coq on fait maintenant référence au λ-terme extrait associé.

Informellement:

- Si $t : P :$ `Prop` (t est une preuve), alors $\mathcal{E}(t) = \square$
- Si $t :$ `Type` (t est un type), alors $\mathcal{E}(t) = \square$
- Sinon t est informatif (i.e. un entier naturel) et on extrait récursivement ses sous termes.

Formellement:

$$\text{Si } \vdash t : \mathbb{N} \text{ et } t \to^* \underline{n} \text{ alors } \mathcal{E}(t) \to^* \underline{n}.$$

Plan

La fonction `nth` sous toutes ses formes

Démonstration

http://www.irif.fr/~sozeau/teaching/CDF-nth.v

Point d'étape

- Les types dépendants permettent de spécifier le code que l'on désire, à l'aide de types sous-ensembles ou de familles inductives.
- Les choix de représentation sont essentiels!
- On a réflechi dans la logique le contenu calculatoire du programme pour pouvoir implémenter les fonctions dépendantes.
- Par extraction, cette décoration logique disparaît!

Plan

La division euclidienne

Programe Caml:

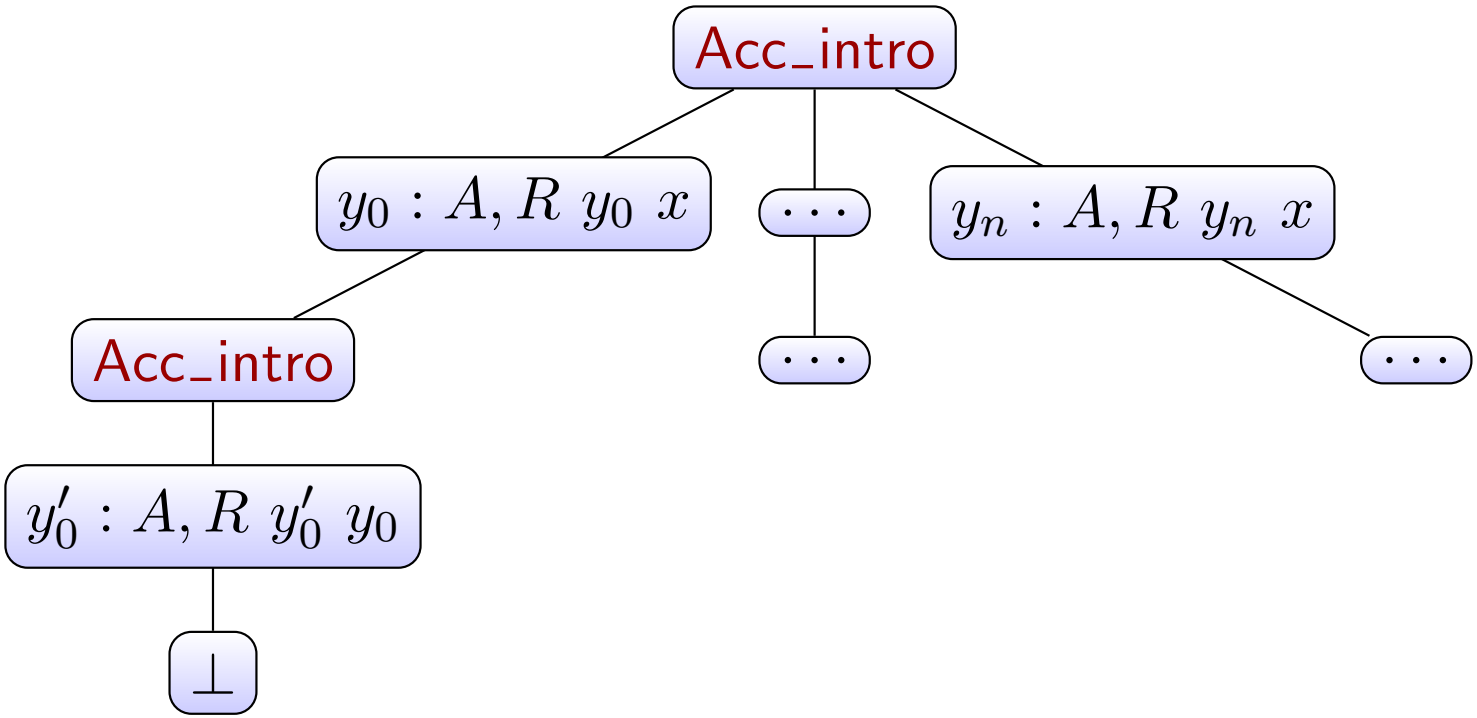
```
val euclid : int -> int -> (int * int)
let rec euclid n m =
  if n < m then (0, n)
  else let (q, r) = euclid (n - m) m in
       (q + 1, r)
```

Démonstration

http://www.irif.fr/~sozeau/teaching/CDF-euclid.v

Accessibilité

```
Inductive Acc (A : Type) (R : A → A → Prop) (x : A) : Prop :=
  Acc_intro : (∀ y : A, R y x → Acc R y) → Acc R x
```



Toute valeur inductive est finie.

Récursion bien fondée

Définition du récurseur sur Acc:

```
Section FixWf.
  Context (A : Type) (R : A → A → Prop) (P : A → Type).
  Context (F : ∀ x : A, (∀ y : A, R y x → P y) → P x).

  Definition well_founded := ∀ x, Acc R x.
```

Récursion bien fondée

Définition du récurseur sur Acc:

```
Section FixWf.
  Context (A : Type) (R : A → A → Prop) (P : A → Type).
  Context (F : ∀ x : A, (∀ y : A, R y x → P y) → P x).

  Definition well_founded := ∀ x, Acc R x.

  Fixpoint Fix (x : A) (a : Acc R x) {struct a} : P x :=
    F x (fun (y : A) (h : R y x) ⇒ Fix y (Acc_inv a h)).
```

Récursion bien fondée

Définition du récurseur sur Acc:

```
Section FixWf.
  Context (A : Type) (R : A → A → Prop) (P : A → Type).
  Context (F : ∀ x : A, (∀ y : A, R y x → P y) → P x).

  Definition well_founded := ∀ x, Acc R x.

  Fixpoint Fix (x : A) (a : Acc R x) {struct a} : P x :=
    F x (fun (y : A) (h : R y x) ⇒ Fix y (Acc_inv a h)).

  Definition FixWf (wf : well_founded) : ∀ x : A, P x :=
    fun x ⇒ Fix x (wf x).

End FixWf.
```

Comparer au combinateur Y tel que $YF \equiv F(YF)$.

Récursion bien fondée

On a maintenant tout le pouvoir de la logique pour démontrer la terminaison!

- Ordres lexicographiques
- Mesures
- Ordinaux
- Récursions imbriquées

Plan

Filtrage et unification

Idée: pendant le filtrage dépendant, on raisonne modulo la théorie de l'égalité et des constructeurs.

Exemple: si on élimine t : vector A m, on va unifier chaque type de constructeur possible:

1. vector A O pour vnil
2. vector A (S n) pour vcons

L'unification $t \equiv u \rightsquigarrow Q$ peut répondre:

- $Q = \texttt{Fail}$ si l'unification échoue;
- $Q = \texttt{Success}\ \sigma$ si l'unification réussit (avec une substitution σ);
- $Q = \texttt{Stuck}\ t$ si elle ne peut pas progresser à cause du terme t.

Dans cet exemple on aura deux succès: un pour $[m := 0]$ et un autre pour $[m := \text{S}\ n]$.

Règles d'unification

$$\frac{x \notin \mathcal{FV}(t)}{x \equiv t \rightsquigarrow \texttt{Success}\ \sigma[x := t]}\ \textsc{Solution}$$

$$\frac{}{\mathsf{C}\ \_ \equiv \mathsf{D}\ \_ \rightsquigarrow \texttt{Fail}}\ \textsc{Discrimination}$$

$$\frac{t_1 \ldots t_n \equiv u_1 \ldots u_n \rightsquigarrow Q}{\mathsf{C}\ t_1 \ldots t_n \equiv \mathsf{C}\ u_1 \ldots u_n \rightsquigarrow Q}\ \textsc{Injectivity}$$

$$\frac{p_1 \equiv q_1 \rightsquigarrow \texttt{Success}\ \sigma \quad (p_2 \ldots p_n)\sigma \equiv (q_2 \ldots q_n)\sigma \rightsquigarrow Q}{p_1 \ldots p_n \equiv q_1 \ldots q_n \rightsquigarrow Q \cup \sigma}\ \textsc{Patterns}$$

$$\frac{}{t \equiv t \rightsquigarrow \texttt{Success}\ []}\ \textsc{Deletion} \qquad \frac{\text{Dans les autres cas}}{t \equiv u \rightsquigarrow \texttt{Stuck}\ u}\ \textsc{Stuck}$$

Exemples d'unifications

- $\mathsf{O} \equiv \mathsf{S}\ n \rightsquigarrow \texttt{Fail}$
- $\mathsf{S}\ m \equiv \mathsf{S}\ (\mathsf{S}\ n) \rightsquigarrow \texttt{Success}\ [m := \mathsf{S}\ n]$
- $\mathsf{O} \equiv m + \mathsf{O} \rightsquigarrow \texttt{Stuck}\ (m + \mathsf{O})$

Les cas `Stuck` indiquent qu'il faut peut-être éliminer une nouvelle variable pour pouvoir décider du succès ou de l'échec (ici la variable m).

L'algorithme de filtrage utilise l'unification pour:

- Décider quelle clause du programme choisir
- Déterminer quels constructeurs s'appliquent quand on élimine une variable

Compilation du filtrage

Les clauses peuvent se superposer: on choisit la première qui unifie dans l'ordre du texte du programme.

```
Equations equal (m n : nat) : bool :=
  equal O O := true;
  equal (S m') (S n') := equal m' n';
  equal m n := false.
```

cover($m\ n$: nat $\vdash$ $m\ n$)

Compilation du filtrage

Les clauses peuvent se superposer: on choisit la première qui unifie dans l'ordre du texte du programme.

```
Equations equal (m n : nat) : bool :=
  equal O O := true;
  equal (S m') (S n') := equal m' n';
  equal m n := false.
```

cover$(m\ n : \mathsf{nat} \vdash m\ n) \rightarrow \mathsf{O}\ \mathsf{O} \equiv m\ n \rightsquigarrow \mathtt{Stuck}\ m$

Compilation du filtrage

Les clauses peuvent se superposer: on choisit la première qui unifie dans l'ordre du texte du programme.

```
Equations equal (m n : nat) : bool :=
  equal O O := true;
  equal (S m') (S n') := equal m' n';
  equal m n := false.
```

Split($m\ n$: nat $\vdash$ $m\ n$, m, [ ])

Compilation du filtrage

Les clauses peuvent se superposer: on choisit la première qui unifie dans l'ordre du texte du programme.

```
Equations equal (m n : nat) : bool :=
  equal O O := true;
  equal (S m') (S n') := equal m' n';
  equal m n := false.
```

```
Split(m n : nat ⊢ n m, m, [
  cover(n : nat    ⊢ O n)
  cover(m' n : nat ⊢ (S m') n)])
```

Compilation du filtrage

Les clauses peuvent se superposer: on choisit la première qui unifie dans l'ordre du texte du programme.

```
Equations equal (m n : nat) : bool :=
  equal O O := true;
  equal (S m') (S n') := equal m' n';
  equal m n := false.
```

```
Split(m n : nat ⊢ m n, m, [
  Split(n : nat ⊢ O n, n, [
    Compute(⊢ O O ⇒ true),
    Compute(n' : nat ⊢ O (S n') ⇒ false)]),
  cover(m' n : nat ⊢ (S m') n)])
```

Compilation du filtrage

Les clauses peuvent se superposer: on choisit la première qui unifie dans l'ordre du texte du programme.

```
Equations equal (m n : nat) : bool :=
  equal O O := true;
  equal (S m') (S n') := equal m' n';
  equal m n := false.
```

```
Split(m n : nat ⊢ m n, m, [
  Split(n : nat ⊢ O n, n, [
    Compute(⊢ O O ⇒ true),
    Compute(n' : nat ⊢ O (S n') ⇒ false)]),
  Split(m' n : nat ⊢ (S m') n, n, [
    Compute(m' : nat ⊢ (S m') O ⇒ false),
    Compute(m' n' : nat ⊢ (S m') (S n') ⇒ equal m' n')])])
```

Filtrage dépendant

```
Inductive vector (A : Type) : nat → Type :=
| vnil : vector A 0
| vcons : A → ∀ (n : nat), vector A n → vector A (S n).

Equations vtail A n (v : vector A (S n)) : vector A n :=
  vtail A n (vcons _ ?(n) v) := v.
```

Chaque variable doit apparaître une seule fois, excepté dans les arguments forcés.

cover($A\ n\ v$: vector A (S n)) $\vdash A\ n\ v$)

Filtrage dépendant

```
Inductive vector (A : Type) : nat → Type :=
| vnil : vector A 0
| vcons : A → ∀ (n : nat), vector A n → vector A (S n).

Equations vtail A n (v : vector A (S n)) : vector A n :=
  vtail A n (vcons _ ?(n) v) := v.
```

Chaque variable doit apparaître une seule fois, excepté dans les arguments forcés.

```
Split(A n (v : vector A (S n)) ⊢ A n v, v, [
  Fail; // O ≠ S n
  cover(A n' a (v' : vector A n') ⊢ A n' (vcons a ?(n') v'))])
```

Filtrage dépendant

```
Inductive vector (A : Type) : nat → Type :=
| vnil : vector A 0
| vcons : A → ∀ (n : nat), vector A n → vector A (S n).

Equations vtail A n (v : vector A (S n)) : vector A n :=
  vtail A n (vcons _ ?(n) v) := v.
```

Chaque variable doit apparaître une seule fois, excepté dans les arguments forcés.

```
Split(A n (v : vector A (S n)) ⊢ A n v, v, [
 Fail; // S n ≠ O
 Compute(A n' a (v' : vector A n') ⊢ A n' (vcons a ?(n') v')
    ⇒ v')])
```

Filtrage dépendant et axiome K

Pour compiler le filtrage, on utilise la non-confusion sur les familles inductives pour résoudre, e.g.:

$$\mathsf{fs}\ n\ f =_{\mathsf{fin}\ (\mathsf{S}\ n)} \mathsf{fs}\ n\ f'$$

Filtrage dépendant et axiome K

Pour compiler le filtrage, on utilise la non-confusion sur les familles inductives pour résoudre, e.g.:

$$\begin{array}{rl} & \mathsf{fs}\ n\ f =_{\mathsf{fin}\ (\mathsf{S}\ n)} \mathsf{fs}\ n\ f' \\ \simeq & (n; f) =_{\Sigma x{:}\mathsf{nat}.\mathsf{fin}\ x} (n; f') \end{array}$$

Filtrage dépendant et axiome K

Pour compiler le filtrage, on utilise la non-confusion sur les familles inductives pour résoudre, e.g.:

$$
\begin{array}{rl}
 & \mathsf{fs}\ n\ f =_{\mathsf{fin}\ (\mathsf{S}\ n)} \mathsf{fs}\ n\ f' \\
\simeq & (n; f) =_{\Sigma x:\mathsf{nat}.\mathsf{fin}\ x} (n; f') \\
\simeq & \Sigma(e : n =_{\mathsf{nat}} n).\mathrm{coerce}\ e\ f =_{\mathsf{fin}\ n} f'
\end{array}
$$

Filtrage dépendant et axiome K

Pour compiler le filtrage, on utilise la non-confusion sur les familles inductives pour résoudre, e.g.:

$$\begin{array}{rl} & \mathsf{fs}\ n\ f =_{\mathsf{fin}\ (\mathsf{S}\ n)} \mathsf{fs}\ n\ f' \\ \simeq & (n; f) =_{\Sigma x:\mathsf{nat}.\mathsf{fin}\ x} (n; f') \\ \simeq & \Sigma(e : n =_{\mathsf{nat}} n).\mathrm{coerce}\ e\ f =_{\mathsf{fin}\ n} f' \end{array}$$

Pour simplifier cette égalité et obtenir $f = f'$, il faudrait savoir que

$$(e : n =_{\mathsf{nat}} n) = \mathsf{eq\_refl}$$

Ce n'est pas le cas en général!

Un peu d'histoire

- Coquand (1992) a introduit la notion de filtrage dépendant comme une nouvelle primitive à la théorie des types, introduisant par la même occasion l'axiome K.

$$K : \forall A\ (x : A)\ (e : x = x), e = \mathsf{eq\_refl}$$

Un peu d'histoire

- Coquand (1992) a introduit la notion de filtrage dépendant comme une nouvelle primitive à la théorie des types, introduisant par la même occasion l'axiome K.

$$K : \forall A\ (x : A)\ (e : x = x), e = \mathsf{eq\_refl}$$

- Cet axiome a été montré indépendant de la théorie des types (MLTT et CIC) par Hofmann et Streicher (1994).
  - Il découle de l'axiome d'indifférence aux preuves.

Un peu d'histoire

- Coquand (1992) a introduit la notion de filtrage dépendant comme une nouvelle primitive à la théorie des types, introduisant par la même occasion l'axiome K.

$$K : \forall A\ (x : A)\ (e : x = x), e = \mathsf{eq\_refl}$$

- Cet axiome a été montré indépendant de la théorie des types (MLTT et CIC) par Hofmann et Streicher (1994).
  - Il découle de l'axiome d'indifférence aux preuves.
  - Il est incompatible avec l'axiome d'univalence (c.f. cours 8 sur l'égalité et séminaire de Thierry Coquand).

Un peu d'histoire

- Coquand (1992) a introduit la notion de filtrage dépendant comme une nouvelle primitive à la théorie des types, introduisant par la même occasion l'axiome K.

$$K : \forall A\ (x : A)\ (e : x = x), e = \mathsf{eq\_refl}$$

- Cet axiome a été montré indépendant de la théorie des types (MLTT et CIC) par Hofmann et Streicher (1994).
  - Il découle de l'axiome d'indifférence aux preuves.
  - Il est incompatible avec l'axiome d'univalence (c.f. cours 8 sur l'égalité et séminaire de Thierry Coquand).
- McBride (1999); Goguen et al. (2006) introduisent l'idée d'"internaliser" le filtrage dépendant en utilisant les constructions de base du langage (éliminateurs, manipulations de l'égalité), toujours en supposant une variante de K.

Axiome K

Question: peut-on restreindre le filtrage pour ne pas valider K?

Cockx (2017) propose un algorithme d'unification basé sur la simplification d'égalités, qui évite l'utilisation de K, en restreignant la règle d'effacement aux preuves de reflexivité:

$$\frac{}{\mathsf{eq\_refl} : t =_T t \rightsquigarrow \texttt{Success}\ []}\ \textsc{Deletion}$$

Axiome K

Question: peut-on restreindre le filtrage pour ne pas valider K?

Cockx (2017) propose un algorithme d'unification basé sur la simplification d'égalités, qui évite l'utilisation de K, en restreignant la règle d'effacement aux preuves de reflexivité:

$$\frac{}{\mathsf{eq\_refl} : t =_T t \rightsquigarrow \texttt{Success}\ []}\ \textsc{Deletion}$$

On doit modifier aussi la règle d'injectivité des constructeurs pour les types indexés, par exemple pour le type fin:

$$\frac{\mathrm{noconf}\ e : n =_{\mathsf{nat}} n \rightsquigarrow Q}{e : \mathsf{fz}\ n =_{\mathsf{fin}\ (\mathsf{S}\ n)} \mathsf{fz}\ n \rightsquigarrow Q}\ \textsc{Injectivity}$$

On aurait encore une fois besoin d'effacement dans la prémisse...

Filtrage sans K

Brady et al. (2003) ont proposé la notion d'*argument forcé* d'un constructeur pour justifier une optimisation de la représentation des constructeurs. Pour fin:

```
Inductive fin : nat → Set :=
| fz : ∀ n : nat, fin (S n)
| fs : ∀ n : nat, fin n → fin (S n).
```

On peut remarquer que l'argument n de chaque constructeur est "forcé" par l'index S n. Non-confusion raffinée:

```
Equations NoConf {n} (f f' : fin n) : Prop :=
  NoConf (fz ?(n)) (fz n) := True;
  NoConf (fs ?(n) f) (fs n f') := f = f';
  NoConf _ _ := False.
```

Non-confusion

```
Equations noconf {n} (f f' : fin n) (e : f = f') : NoConf f f' :=
  ...
Equations noconfeq {n} (f f' : fin n) (e : NoConf f f') : f = f'
:=
  noconfeq (fz ?(n)) (fz n) _ := eq_refl;
  ...
```

- Ces deux fonctions forment une équivalence de types (isomorphisme).
- Cela signifie que le type des égalités $f = f'$ est égal au type $\mathrm{NoConf}\ f\ f'$
- $\mathrm{noconfeq}\ p \equiv \mathrm{eq\_refl}$ quelque soit la preuve p de $\mathrm{NoConf}\ f\ f'$

Filtrage sans K

Cela justifie la règle d'injectivité:

$$\frac{\mathrm{noconfeq}\ (\mathrm{noconf}\ e) \equiv \mathsf{eq\_refl} : (\mathsf{fz}\ n) =_{\mathsf{fin}\ (\mathsf{S}\ n)} (\mathsf{fz}\ n) \rightsquigarrow \mathtt{Success}\ []}{e : \mathsf{fz}\ n =_{\mathsf{fin}\ (\mathsf{S}\ n)} \mathsf{fz}\ n \rightsquigarrow \mathtt{Success}\ [e := \mathsf{eq\_refl}]}$$

$\Rightarrow$ Les arguments forcés n'ont pas besoin d'être unifiés (ils sont égaux par construction).

- Dans Agda, cela justifie l'unification utilisée dans le mode `--without-K`
- Pour Coq, Equations defini la non-confusion en utilisant cette variante et compile le filtrage vers les constructions de base, sans utilier d'axiome.

Conclusion

- La programmation avec types dépendants permet de justifier la partialité, la terminaison et la correction des programmes en leur donnant des spécifications fortes.
- Le contenu calculatoire ne change pas, mais on introduit des "décorations" logiques, mélangeant preuves et programmes.
- Ces décorations peuvent venir compliquer (rustines!) ou simplifier les preuves (pas de cas inutiles).
- L'extraction efface ces décorations pour obtenir le contenu calculatoire.

Directions de recherche

- Une théorie avec indifférence aux preuves définitionelle (Gilbert et al., 2019)
- Certifier l'extraction, chaîne de compilation certifiée de Coq vers C (CompCert): le projet CertiCoq (Anand et al., 2017).
- Optimiser l'extraction: certains arguments "informatifs" comme les indices de familles inductives peuvent parfois être effacés: Calcul des Constructions Implicite (Barras et Bernardo, 2008).

Bibliographie

Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, et Matthew Weaver. CertiCoq: A verified compiler for Coq. In *CoqPL*, Paris, France, 2017.

Bruno Barras et Bruno Bernardo. The Implicit Calculus of Constructions as a Programming Language with Dependent Types. In Roberto M. Amadio, editor, *FoSSaCS*, volume 4962 of *Lecture Notes in Computer Science*, pages 365–379. Springer, 2008.

Edwin Brady, Conor McBride, et James McKinna. Inductive Families Need Not Store Their Indices. In Stefano Berardi, Mario Coppo, et Ferruccio Damiani, editors, *TYPES*, volume 3085 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2003.

Jesper Cockx. *Dependent Pattern Matching and Proof-Relevant Unification*. PhD thesis, Katholieke Universiteit Leuven, Belgium, 2017.

Thierry Coquand. Pattern Matching with Dependent Types, 1992. Proceedings of the Workshop on Logical Frameworks.

Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, et Nicolas Tabareau. Definitional Proof-Irrelevance without K. In *46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2019)*, POPL, Lisbon, Portugal, January 2019.

Healfdene Goguen, Conor McBride, et James McKinna. Eliminating Dependent Pattern Matching. In Kokichi Futatsugi, Jean-Pierre Jouannaud, et José Meseguer, editors, *Essays Dedicated to Joseph A. Goguen*, volume 4060 of *Lecture Notes in Computer Science*, pages 521–540. Springer, 2006.

Martin Hofmann et Thomas Streicher. A Groupoid Model Refutes Uniqueness of Identity Proofs. In *LICS*, pages 208–212. IEEE Computer Society, 1994.

Pierre Letouzey. *Programmation fonctionnelle certifiée: l'extraction de programmes dans l'assistant Coq*. Thèse de doctorat, Université Paris-Sud, July 2004.

Conor McBride. *Dependently Typed Functional Programs and Their Proofs*. PhD thesis, University of Edinburgh, 1999.

Christine Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions. (Program Extraction in the Calculus of Constructions)*. PhD thesis, Paris Diderot University, France, 1989.