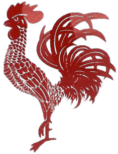
Equations
A Dependent Pattern-Matching Suite

Matthieu Sozeau

INRIA Paris - PPS

September 15th 2011
Shonan Village Center, Japan



Barber shop's red rooster (Eugene, OR)

Overview

- Epigram/Agda-style pattern-matching definitions with `with`
- Purely logical handling of recursion for inductive families
- Propositional equations for definitional equalities
- Elimination principle and support for applying it

Entirely elaborated to the vanilla kernel!

DEMO

Dependent Pattern-Matching

- Patterns = **well-typed** refinements of the signature
- We refine the **entire** context at each node (correct dependency tracking)
- Internalizes "Uniqueness of Identity Proofs" (axiom K)
- **Inaccessible** patterns + first-match semantics ensure operationality
- Empty nodes ensure decidability of coverage

Compilation setup

Elaboration into CIC + K

Three phases:

1. *Generation* of a splitting tree from the clauses
2. *Translation* from the splitting tree to Coq terms with holes
3. *Proofs* of the obligations using a mix of ML and $\mathcal{L}_{\mathsf{tac}}$ code

Input language

`term, type`	$t,\ \tau$	$::=$	$x \mid \lambda x : \tau, t \mid \Pi x : \tau, \tau' \mid \ldots$
`binding`	d	$::=$	$(x\ :\ \tau) \mid (x\ :=\ t\ :\ \tau)$
`context`	Γ, Δ	$::=$	$\overrightarrow{d}$
`user pattern`	up	$::=$	$x \mid \mathsf{C}\ \overrightarrow{up} \mid \texttt{?}(t)$
`user node`	n	$::=$	$:=\ t \mid :=!\ x \mid \texttt{with}\ t := \{\ \overrightarrow{c}\ \}$
`user clause`	c	$::=$	$\mathsf{f}\ \overrightarrow{up}\ n$
`program`	$prog$	$::=$	$\mathsf{f}\ \Gamma\ :\ \tau\ :=\ \overrightarrow{c}$

Searching for a splitting tree

For $\mathsf{f}\ \Delta : \tau$ we define $\mathsf{f}_{\mathsf{comp}}\ \Delta := \tau$, so $\mathsf{f} : \Pi\ \Delta, \mathsf{f}_{\mathsf{comp}}\ \overline{\Delta}$.

Searching for a splitting tree

For f $\Delta : \tau$ we define $\mathsf{f_{comp}}\ \Delta := \tau$, so $\mathsf{f} : \Pi\ \Delta, \mathsf{f_{comp}}\ \overline{\Delta}$.

`pattern`	p	$::=$	$x \mid \mathsf{C}\ \overrightarrow{p} \mid \texttt{?}(t)$
`context map`	c	$::=$	$\Delta \vdash \overrightarrow{p} : \Gamma$
`splitting`	spl	$::=$	$\mathsf{Split}(c, x, (spl?)^n) \mid \mathsf{Compute}(c, rhs)$
`node`	rhs	$::=$	$\mathsf{Program}(t) \mid \mathsf{Refine}(c, t, \ell, spl)$

Goal Starting with f $\Delta : \mathsf{f_{comp}}\ \overline{\Delta} := \overrightarrow{\overrightarrow{p}\ \ldots}$, find a covering of the context map $\texttt{idsubst}(\Delta) = \Delta \vdash \overline{\Delta} : \Delta$.

Proof search example

Overlapping clauses with first-match semantics.

```
Equations equal (n m : nat) : { n = m } + { n ≠ m } :=
equal O O := left eq_refl ;
equal (S n) (S m) with equal n m := {
  equal (S n) (S ?(n)) (left eq_refl) := left eq_refl ;
  equal (S n) (S m) (right p) := right _ } ;
equal x y := right _.
```

Split($n\ m$: nat $\vdash n\ m : n\ m$: nat, n, [
  Split(m : nat $\vdash O\ m : n\ m$: nat, m, [
    Compute($\vdash O\ O : n\ m$: nat, Program(left eq_refl)),
    Compute(m : nat $\vdash O\ (S\ m) : n\ m$: nat, Program(right _))]),
  Split($n\ m$: nat $\vdash (S\ n)\ m : n\ m$: nat, m, [
    Compute(n : nat $\vdash (S\ n)\ O : n\ m$: nat, $\ldots$),
    Compute($n\ m$: nat $\vdash (S\ n)\ (S\ m) : n\ m$: nat,
      Refine(equal $n\ m$,
        idsubst($n\ m$: nat, $x : \{n = m\} + \{n \neq m\}$), ℓ, $\ldots$))])])

Translation from the splitting to Coq

For each node with context map $\Delta \vdash ps : \Gamma$ we generate an obligation of type $\Pi\ \Delta, \mathsf{f}_{\mathsf{comp}}\ ps$.

Translation from the splitting to Coq

For each node with context map $\Delta \vdash ps : \Gamma$ we generate an obligation of type $\Pi\ \Delta, \mathsf{f}_{\mathsf{comp}}\ ps$.

- Split(c, x, s): witnessed by applying a dependent elimination (`dependent destruction`, using JMeq) and using the compiled terms for s. Empty nodes are translated to empty splittings.

Translation from the splitting to Coq

For each node with context map $\Delta \vdash ps : \Gamma$ we generate an obligation of type $\Pi\ \Delta, \mathsf{f}_{\mathsf{comp}}\ ps$.

- Split(c, x, s): witnessed by applying a dependent elimination (`dependent destruction`, using JMeq) and using the compiled terms for s. Empty nodes are translated to empty splittings.
- Program(t): witnessed by the term.

Translation from the splitting to Coq

For each node with context map $\Delta \vdash ps : \Gamma$ we generate an obligation of type $\Pi\ \Delta, \mathsf{f_{comp}}\ ps$.

- Split(c, x, s): witnessed by applying a dependent elimination (`dependent destruction`, using JMeq) and using the compiled terms for s. Empty nodes are translated to empty splittings.
- Program(t): witnessed by the term.
- Refine(t, c, ℓ, s): witnessed by inserting a let-definition in the context, strengthening, abstracting and clearing its body, then applying the compiled term for label ℓ.

With nodes in detail

Consider a current problem $\Delta \vdash \overrightarrow{p} : \Gamma$ and a user clause f $\overrightarrow{up}$ `with` t_{pre} := { e } matching it. We typecheck t_{pre} into $t : \tau$ and use strenghtening and abstraction to find a new context

$$\Delta_x \triangleq \Delta^t, x_t : \tau, \Delta_t[t/x_t] \text{ s.t. } \begin{cases} \Delta^t, \Delta_t \sim \Delta \\ \Delta_x \vdash (\mathsf{f}_{\mathsf{comp}}\ \overrightarrow{p})[t/x_t] : \mathtt{Type} \end{cases}$$

With nodes in detail

Consider a current problem $\Delta \vdash \overrightarrow{p} : \Gamma$ and a user clause f $\overrightarrow{up}$ `with` $t_{pre} := \{\ e\ \}$ matching it. We typecheck t_{pre} into $t : \tau$ and use strenghtening and abstraction to find a new context

$$\Delta_x \triangleq \Delta^t, x_t : \tau, \Delta_t[t/x_t] \text{ s.t. } \begin{cases} \Delta^t, \Delta_t \sim \Delta \\ \Delta_x \vdash (\mathsf{f}_{\mathsf{comp}}\ \overrightarrow{p})[t/x_t] : \mathtt{Type} \end{cases}$$

Using the clauses e we then build a subcovering s of the identity context map $c = \mathtt{idsubst}(\Delta_x)$ and return Refine$(t, c, \ell.n, s)$.

With nodes in detail

Consider a current problem $\Delta \vdash \overrightarrow{p} : \Gamma$ and a user clause f $\overrightarrow{up}$ `with` $t_{pre} := \{\ e\ \}$ matching it. We typecheck t_{pre} into $t : \tau$ and use strenghtening and abstraction to find a new context

$$\Delta_x \triangleq \Delta^t, x_t : \tau, \Delta_t[t/x_t] \text{ s.t. } \begin{cases} \Delta^t, \Delta_t \sim \Delta \\ \Delta_x \vdash (\mathsf{f}_{\mathsf{comp}}\ \overrightarrow{p})[t/x_t] : \mathtt{Type} \end{cases}$$

Using the clauses e we then build a subcovering s of the identity context map $c = \mathtt{idsubst}(\Delta_x)$ and return Refine$(t, c, \ell.n, s)$. Compilation will produce:

$$\ell.n : \Pi\ \Delta_x, (\mathsf{f}_{\mathsf{comp}}\ \overrightarrow{p})[t/x_t]$$

we can then build:

$$(\lambda\Delta, \ell.n\ \overline{\Delta^t}\ t\ \overline{\Delta_t}) : \Pi\ \Delta, \mathsf{f}_{\mathsf{comp}}\ \overrightarrow{p}$$

1. Dependent pattern-matching compilation

2. Recursion
   - The Below way
   - Subterm relations

3. Reasoning support
   - Equations
   - Elimination principle
   - Eliminating calls

Recursion

- Syntactic guardness checks are too fragile (and buggy)
- Do not work well with abstraction/modularity
- Restricted to structural recursion on a single argument, with no currying allowed

Idea Use the logic instead !

The Below way (McBride and McKinna)

```
Fixpoint Below_nat (P : nat → Type) (n : nat) : Type :=
  match n with
    | 0 ⇒ ()
    | S n′ ⇒ (P n′ × Below_nat P n′)
  end%type.
```

The Below way (McBride and McKinna)

```
Fixpoint Below_nat (P : nat → Type) (n : nat) : Type :=
  match n with
    | 0 ⇒ ()
    | S n′ ⇒ (P n′ × Below_nat P n′)
  end%type.

Fixpoint below_nat (P : nat → Type)
  (step : Π n : nat, Below_nat P n → P n)
  (n : nat) : Below_nat P n.
```

The Below way (McBride and McKinna)

```
Fixpoint Below_nat (P : nat → Type) (n : nat) : Type :=
  match n with
    | 0 ⇒ ()
    | S n′ ⇒ (P n′ × Below_nat P n′)
  end%type.

Fixpoint below_nat (P : nat → Type)
  (step : Π n : nat, Below_nat P n → P n)
  (n : nat) : Below_nat P n.

Definition rec_nat (P : nat → Type)
  (step : Π n : nat, Below_nat P n → P n)
  (n : nat) : P n := step n (below_nat P step n).
```

Integration into EQUATIONS

```
Equations unzip {A B n} (v : vector (A×B) n) : vector A n × vector B n
:=
unzip A B n v by rec v :=
unzip A B ?(O) Vnil := (Vnil, Vnil) ;
unzip A B ?(S n) (Vcons (pair x y) n v) with unzip v := {
  | (pair xs ys) := (Vcons x xs, Vcons y ys) }.
```

- by rec v applies the elimination principle associated to the type of v (found using typeclass resolution).

Integration into EQUATIONS

```
Equations unzip {A B n} (v : vector (A×B) n) : vector A n × vector B n
:=
unzip A B n v by rec v :=
unzip A B ?(O) Vnil := (Vnil, Vnil) ;
unzip A B ?(S n) (Vcons (pair x y) n v) with unzip v := {
  | (pair xs ys) := (Vcons x xs, Vcons y ys) }.
```

- by rec v applies the elimination principle associated to the type of v (found using typeclass resolution).
- Introduce **hidden** variables in the problem to carry recursion hypotheses of the form Below $(\Pi\ \Delta, \mathsf{f}_{\mathsf{comp}}\ \overrightarrow{t})\ x$.

Integration into Equations

```
Equations unzip {A B n} (v : vector (A×B) n) : vector A n × vector B n
:=
unzip A B n v by rec v :=
unzip A B ?(O) Vnil := (Vnil, Vnil) ;
unzip A B ?(S n) (Vcons (pair x y) n v) with unzip v := {
  | (pair xs ys) := (Vcons x xs, Vcons y ys) }.
```

- by rec v applies the elimination principle associated to the type of v (found using typeclass resolution).
- Introduce **hidden** variables in the problem to carry recursion hypotheses of the form Below $(\Pi\ \Delta, \mathsf{f}_{\mathsf{comp}}\ \overrightarrow{t})\ x$.
- Each recursive occurrence of f is transformed to a trivial projection $\mathsf{f}_{\mathsf{comp\_proj}} : \Pi\ \Delta\ \{p : \mathsf{f}_{\mathsf{comp}}\ \overline{\Delta}\}, \mathsf{f}_{\mathsf{comp}}\ \overline{\Delta}$.

Integration into EQUATIONS

```
Equations unzip {A B n} (v : vector (A×B) n) : vector A n × vector B n
:=
unzip A B n v by rec v :=
unzip A B ?(O) Vnil := (Vnil, Vnil) ;
unzip A B ?(S n) (Vcons (pair x y) n v) with unzip v := {
  | (pair xs ys) := (Vcons x xs, Vcons y ys) }.
```

- by rec v applies the elimination principle associated to the type of v (found using typeclass resolution).
- Introduce **hidden** variables in the problem to carry recursion hypotheses of the form Below $(\Pi\ \Delta, \mathsf{f}_{\mathsf{comp}}\ \overrightarrow{t})\ x$.
- Each recursive occurrence of f is transformed to a trivial projection $\mathsf{f}_{\mathsf{comp\_proj}} : \Pi\ \Delta\ \{p : \mathsf{f}_{\mathsf{comp}}\ \overline{\Delta}\}, \mathsf{f}_{\mathsf{comp}}\ \overline{\Delta}$.
- Proof search for $\mathsf{f}_{\mathsf{comp}}$ goals appearing as obligations, unfolding Below hypotheses.

Below is inefficient!

Use **well-founded** recursion on the subterm relation for inductive families $\mathsf{I} : \Pi\ \Delta,$

Below is inefficient!

Use **well-founded** recursion on the subterm relation for inductive families $\mathsf{I} : \Pi\ \Delta$,

- General definition of direct subterm:
  $\mathsf{I}_{sub} : \Pi\ \Delta_l\ \Delta_r, \mathsf{I}\ \overline{\Delta_l} \to \mathsf{I}\ \overline{\Delta_r} \to \texttt{Prop}$

Below is inefficient!

Use **well-founded** recursion on the subterm relation for inductive families $\mathsf{I} : \Pi\ \Delta$,

- General definition of direct subterm: $\mathsf{I}_{sub} : \Pi\ \Delta_l\ \Delta_r, \mathsf{I}\ \overline{\Delta_l} \to \mathsf{I}\ \overline{\Delta_r} \to \mathtt{Prop}$
- Wrap the inductive type with its indices in a sigma and define an homogeneous relation on: I_{sub} : relation $(\Sigma\ \Delta, \mathsf{I}\ \overline{\Delta})$

Subterm relations and well-founded recursion

Below is inefficient!

Use **well-founded** recursion on the subterm relation for inductive families $\mathsf{I} : \Pi\ \Delta,$

- General definition of direct subterm:
  $\mathsf{I}_{sub} : \Pi\ \Delta_l\ \Delta_r, \mathsf{I}\ \overline{\Delta_l} \rightarrow \mathsf{I}\ \overline{\Delta_r} \rightarrow \texttt{Prop}$
- Wrap the inductive type with its indices in a sigma and define an homogeneous relation on: I_{sub} : relation $(\Sigma\ \Delta, \mathsf{I}\ \overline{\Delta})$
- Extracts efficiently, proof search only a tiny bit more complicated than for Below

Subterm relation example: vectors

```
Derive Subterm for vector.
```

Subterm relation example: vectors

```
Derive Subterm for vector.

Inductive vector_strict_subterm (A : Type)
  : ∀ n m : nat, vector A n → vector A m → Prop :=
    vector_strict_subterm_1_1 : ∀ (a : A) (n : nat) (v : vector A n),
      vector_strict_subterm A n (S n) v (Vcons a v).

Check vector_subterm : ∀ A : Type, relation {index : nat & vector A index}.
```

Subterm relation example: vectors

```
Derive Subterm for vector.

Inductive vector_strict_subterm (A : Type)
  : ∀ n m : nat, vector A n → vector A m → Prop :=
    vector_strict_subterm_1_1 : ∀ (a : A) (n : nat) (v : vector A n),
      vector_strict_subterm A n (S n) v (Vcons a v).

Check vector_subterm : ∀ A : Type, relation {index : nat & vector A index}.

Equations unzip {A B n} (v : vector (A × B) n)
  : vector A n × vector B n :=
unzip A B n v by rec v :=
unzip A B ?(O) Vnil := (Vnil, Vnil) ;
unzip A B ?(S n) (Vcons (pair x y) n v) with unzip v := {
  | (pair xs ys) := (Vcons x xs, Vcons y ys) }.
```

1. Dependent pattern-matching compilation

2. Recursion
   - The Below way
   - Subterm relations

3. Reasoning support
   - Equations
   - Elimination principle
   - Eliminating calls

Reasoning support: equations

Goal: keep an abstract view of definitions.

- Equations hold definitionally in CCI + K
- Equations for `with` nodes are just proxies to the helper function f.ℓ.
- All put together in a rewrite database, f can now be opacified.
- For well-founded definitions, we use the unfolding lemma to prove the equations.

Reasoning support: elimination principle

```
Equations filter {A} (l : list A) (p : A → bool) : list A :=
filter A nil p := nil ;
filter A (cons a l) p with p a := {
  | true := a :: filter l p ;
  | false := filter l p }.
```

Reasoning support: elimination principle

```
Equations filter {A} (l : list A) (p : A → bool) : list A :=
filter A nil p := nil ;
filter A (cons a l) p with p a := {
  | true := a :: filter l p ;
  | false := filter l p }.
```

```
Check (filter_elim :
  ∀ (P : ∀ (A : Type) (l : list A) (p : A → bool), filter_comp l p → Prop),
  let P0 := fun (A : Type) (a : A) (l : list A) (p : A → bool)
        (refine : bool) (res : filter_comp (a :: l) p) ⇒
        p a = refine → P A (a :: l) p res
  in
  (∀ (A : Type) (p : A → bool), P A [] p []) →
  (∀ (A : Type) (a : A) (l : list A) (p : A → bool),
     P A l p (filter l p) → P0 A a l p true (a :: filter l p)) →
  (∀ (A : Type) (a : A) (l : list A) (p : A → bool),
     P A l p (filter l p) → P0 A a l p false (filter l p)) →
  ∀ (A : Type) (l : list A) (p : A → bool), P A l p (filter l p)).
```

Eliminating calls

How to prove using filter_elim ?

$$\Pi\ A\ (l : \mathsf{list}\ A),\ \mathsf{filter}\ (\lambda\_,\ \mathsf{false})\ l = []$$

Eliminating calls

How to prove using filter_elim ?

$$\Pi\ A\ (l : \mathsf{list}\ A),\ \mathsf{filter}\ (\lambda\_,\ \mathsf{false})\ l = []$$

⇒ Abstract by equalities:

$$\begin{array}{l}(\lambda\ (l : \mathsf{list}\ A)\ (p : A \rightarrow \mathsf{bool})\ (r : \mathsf{filter}_{\mathsf{comp}}\ l\ p), \\ \quad p = (\lambda\_,\ \mathsf{false}) \rightarrow r = \mathsf{filter}\ l\ p \rightarrow \mathsf{filter}\ (\lambda\_,\ \mathsf{false})\ l = []) \\ \quad l\ (\lambda\_,\ \mathsf{false})\ (\mathsf{filter}\ l\ (\lambda\_,\ \mathsf{false}))\end{array}$$

Eliminating calls

How to prove using filter_elim ?

$$\Pi\ A\ (l : \text{list}\ A),\ \text{filter}\ (\lambda\_,\ \text{false})\ l = []$$

$\Rightarrow$ Abstract by equalities:

$$\begin{aligned}&(\lambda\ (l : \text{list}\ A)\ (p : A \to \text{bool})\ (r : \text{filter}_{\text{comp}}\ l\ p),\\ &\quad p = (\lambda\_,\ \text{false}) \to r = \text{filter}\ l\ p \to \text{filter}\ (\lambda\_,\ \text{false})\ l = [])\\ &\quad l\ (\lambda\_,\ \text{false})\ (\text{filter}\ l\ (\lambda\_,\ \text{false}))\end{aligned}$$

$\Rightarrow$ Apply the elimination principle and simplify the equations.

Summary

A function definition package handling:

- Full, nested dependent pattern-matching
- Structural and well-founded recursion on inductive families
- Generation of useful support lemmas for reasoning a posteriori

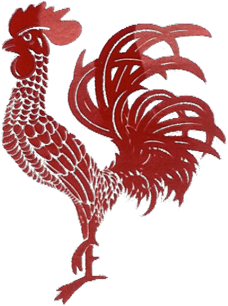
Compared to FUNCTION, mainly adds support for inductive families and a more robust implementation.

Tested on a bit-fiddling library and a formalization of LF: less boilerplate, shorter proofs.

Perspectives

- Treatment of non-constructor indices and unsolved constraints, e.g.: $0 = x + y$, with a subsequent splitting on x.
- Mutual recursion (structural and well-founded)
- Move to eq_dep instead of JMeq? Necessary to use decidable instances of K.
- Efficiency, primitive handling of K.

The End



http://mattam.org/research/coq/equations.en.html