

# EQUATIONS

## A Dependent Pattern-Matching Suite

MATTHIEU SOZEAU

INRIA Paris - PPS

May 31 2012  
IOC, Tallinn, Estonia



*Barber shop's red rooster (Eugene, OR)*



- ▶ 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

## **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  $\mathcal{L}_{\text{tac}}$  code

term, type	$t, \tau$	$::=$	$x \mid \lambda x : \tau, t \mid \Pi x : \tau, \tau' \mid \dots$
binding	$d$	$::=$	$(x : \tau) \mid (x := t : \tau)$
context	$\Gamma, \Delta$	$::=$	$\vec{d}$
user pattern	$up$	$::=$	$x \mid \mathbf{C} \vec{up} \mid ?(t)$
user node	$n$	$::=$	$:= t \mid :=! x \mid \mathbf{with} t := \{ \vec{c} \}$
user clause	$c$	$::=$	$\mathbf{f} \vec{up} n$
program	$prog$	$::=$	$\mathbf{f} \Gamma : \tau := \vec{c}$

## Searching for a splitting tree

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

# Searching for a splitting tree

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

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

**Goal** Starting with  $f \Delta : f_{\text{comp}} \bar{\Delta} := \vec{p} \dots$ , find a covering of the context map  $\text{idsubst}(\Delta) = \Delta \vdash \bar{\Delta} : \Delta$ .

# 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 ⊢ n m : n m : nat, n, [  
  Split(m : nat ⊢ O m : n m : nat, m, [  
    Compute(⊢ O O : n m : nat, Program(left eq_refl)),  
    Compute(m : nat ⊢ O (S m) : n m : nat, Program(right _))]),  
  Split(n m : nat ⊢ (S n) m : n m : nat, m, [  
    Compute(n : nat ⊢ (S n) O : n m : nat, ...),  
    Compute(n m : nat ⊢ (S n) (S m) : n m : nat,  
      Refine(equal n m,  
        idsubst(n m : nat, x : {n = m} + {n ≠ m}), ℓ, ...)))]])
```



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

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

- ▶  $\text{Split}(c, x, s)$ : witnessed by applying a dependent elimination (dependent destruction, using [JMeq](#)) and using the compiled terms for  $s$ . Empty nodes are translated to empty splittings.

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

- ▶  $\text{Split}(c, x, s)$ : witnessed by applying a dependent elimination (dependent destruction, using [JMeq](#)) and using the compiled terms for  $s$ . Empty nodes are translated to empty splittings.
- ▶  $\text{Program}(t)$ : witnessed by the term.

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

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

# With nodes in detail

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

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

# With nodes in detail

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

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

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

# With nodes in detail

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

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

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

$$\ell.n : \Pi \Delta_x, (f_{\text{comp}} \vec{p})[t/x_t]$$

we can then build:

$$(\lambda \Delta, \ell.n \overline{\Delta^t} t \overline{\Delta_t}) : \Pi \Delta, f_{\text{comp}} \vec{p}$$

1 Dependent pattern-matching compilation

2 Recursion

- The Below way
- Subterm relations

3 Reasoning support

- Equations
- Elimination principle
- Eliminating calls



- ▶ 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).
```

```
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`  $\{A\ B\ n\}$   $(v : \text{vector } (A \times B)\ n) : \text{vector } A\ n \times \text{vector } B\ n$   
:=

`unzip`  $A\ B\ n\ v$  **by rec**  $v$  :=

`unzip`  $A\ B\ ?(O)\ \text{Vnil} := (\text{Vnil}, \text{Vnil}) ;$

`unzip`  $A\ B\ ?(S\ n)\ (\text{Vcons } (\text{pair } x\ y)\ n\ v)$  **with** `unzip`  $v$  := {  
|  $(\text{pair } xs\ ys) := (\text{Vcons } x\ xs, \text{Vcons } y\ ys)$  }.

- ▶ **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`  $(\Pi\ \Delta, \text{f}_{\text{comp}} \overrightarrow{t})\ x$ .

# Integration into EQUATIONS

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

- ▶ **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`  $(\Pi\ \Delta, \text{f}_{\text{comp}} \overrightarrow{t})\ x$ .
- ▶ Each recursive occurrence of  $f$  is transformed to a trivial projection  $\text{f}_{\text{comp\_proj}} : \Pi\ \Delta\ \{p : \text{f}_{\text{comp}}\ \overline{\Delta}\}, \text{f}_{\text{comp}}\ \overline{\Delta}$ .

# Integration into EQUATIONS

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

- ▶ **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`  $(\Pi\ \Delta, f_{\text{comp}} \overrightarrow{t})\ x$ .
- ▶ Each recursive occurrence of  $f$  is transformed to a trivial projection  $f_{\text{comp\_proj}} : \Pi\ \Delta\ \{p : f_{\text{comp}}\ \overline{\Delta}\}, f_{\text{comp}}\ \overline{\Delta}$ .
- ▶ Proof search for  $f_{\text{comp}}$  goals appearing as obligations, unfolding `Below` hypotheses.



Below is inefficient!

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

Below is inefficient!

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

- ▶ General definition of direct subterm:  
 $I_{sub} : \Pi \Delta_l \Delta_r, I \overline{\Delta_l} \rightarrow I \overline{\Delta_r} \rightarrow \mathbf{Prop}$

Below is inefficient!

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

- ▶ General definition of direct subterm:

$$I_{sub} : \Pi \Delta_l \Delta_r, I \overline{\Delta_l} \rightarrow I \overline{\Delta_r} \rightarrow \mathbf{Prop}$$

- ▶ Wrap the inductive type with its indices in a sigma and define an homogeneous relation on:  $I_{sub} : \mathbf{relation} (\Sigma \Delta, I \overline{\Delta})$

Below is inefficient!

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

- ▶ General definition of direct subterm:

$$I_{sub} : \Pi \Delta_l \Delta_r, I \overline{\Delta_l} \rightarrow I \overline{\Delta_r} \rightarrow \mathbf{Prop}$$

- ▶ Wrap the inductive type with its indices in a sigma and define an homogeneous relation on:  $I_{sub} : \mathbf{relation} (\Sigma \Delta, I \overline{\Delta})$
- ▶ 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 ( $A : \text{Type}$ )

:  $\forall n m : \text{nat}, \text{vector } A n \rightarrow \text{vector } A m \rightarrow \text{Prop} :=$

vector\_strict\_subterm\_1\_1 :  $\forall (a : A) (n : \text{nat}) (v : \text{vector } A n),$   
vector\_strict\_subterm  $A n (\text{S } n) v (\text{Vcons } a v).$

Check vector\_subterm :  $\forall A : \text{Type}, \text{relation } \{index : \text{nat} \ \& \ \text{vector } A \ index\}.$

# Subterm relation example: vectors

Derive Subterm for vector.

Inductive `vector_strict_subterm` ( $A : \text{Type}$ )  
:  $\forall n m : \text{nat}, \text{vector } A n \rightarrow \text{vector } A m \rightarrow \text{Prop} :=$   
  `vector_strict_subterm_1_1` :  $\forall (a : A) (n : \text{nat}) (v : \text{vector } A n),$   
    `vector_strict_subterm`  $A n (\text{S } n) v (\text{Vcons } a v).$

Check `vector_subterm` :  $\forall A : \text{Type}, \text{relation } \{ \text{index} : \text{nat} \ \& \ \text{vector } A \ \text{index} \}.$

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

1 Dependent pattern-matching compilation

2 Recursion

- The Below way
- Subterm relations

3 Reasoning support

- Equations
- Elimination principle
- Eliminating calls



**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.

```
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)).
```

How to prove using `filter_elim` ?

$$\prod A (l : \text{list } A), \text{filter } (\lambda_, \text{false}) l = []$$

How to prove using `filter_elim` ?

$$\prod A (l : \text{list } A), \text{filter } (\lambda_, \text{false}) l = []$$

⇒ Abstract by equalities:

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

How to prove using `filter_elim` ?

$$\prod A (l : \text{list } A), \text{filter } (\lambda_, \text{false}) l = []$$

⇒ Abstract by equalities:

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

⇒ Apply the elimination principle and simplify the equations.

A function definition package handling:

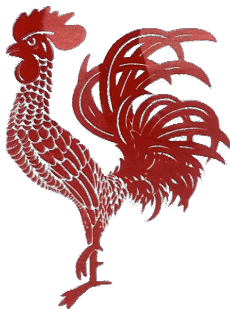
- ▶ Full, nested dependent pattern-matching
- ▶ Structural and well-founded recursion on inductive families
- ▶ Generation of useful support lemmas for reasoning a posteriori

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.

- ▶ 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`.





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