

# EQUATIONS: a dependent pattern-matching compiler

MATTHIEU SOZEAU

Harvard University

UPenn PL club  
September 24th 2010  
Philadelphia, PA



- ▶ 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 ensure operability

## **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

# 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  $f : \Pi \Delta, f_{\text{comp}} \overline{\Delta}$ .

pattern	$p$	$::=$	$x \mid \mathbf{C} \overrightarrow{p} \mid ?(t)$
context map	$c$	$::=$	$\Delta \vdash \overrightarrow{p} : \Gamma$
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  $f \Delta : f_{\text{comp}} \overline{\Delta} := \overrightarrow{p} \dots$ , find a covering of the context map  $\text{idsubst}(\Delta) = \Delta \vdash \overline{\Delta} : \Delta$ .

# 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 ⊢ 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, ...)))]])])
```

# 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

$$\Delta^t, x_t : \tau, \Delta_t[t/x_t] \text{ such that } \Delta^t, \Delta_t \sim \Delta$$

# 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

$$\Delta^t, x_t : \tau, \Delta_t[t/x_t] \text{ such that } \Delta^t, \Delta_t \sim \Delta$$

Using the clauses  $e$  we then build a subcovering  $s$  of the identity context map

$$c = \text{idsubst}(\Delta^t, x_t : \tau_\Delta, \Delta_t[t/x_t])$$

and return  $\text{Refine}(t, c, l.n, s)$ .

# 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 strengthening and abstraction to find a new context

$$\Delta^t, x_t : \tau, \Delta_t[t/x_t] \text{ such that } \Delta^t, \Delta_t \sim \Delta$$

Using the clauses  $e$  we then build a subcovering  $s$  of the identity context map

$$c = \text{idsubst}(\Delta^t, x_t : \tau_\Delta, \Delta_t[t/x_t])$$

and return  $\text{Refine}(t, c, l.n, s)$ .

Compilation produces  $l.n : \Pi \Delta^t (x_t : \tau_\Delta) \Delta_t[t/x_t], (f_{\text{comp}} \overrightarrow{p})[t/x_t]$ , we build

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

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  $\ell$ .

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

Introduced by 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

Introduced by 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 :  $\prod n : \text{nat}, \text{Below\_nat } P n \rightarrow P n$ )  
  (n : nat) : Below_nat P n.
```

# The Below way

Introduced by 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).

**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**  $v := \{$   
 $\quad | (\text{pair } xs\ ys) := (\text{Vcons } x\ xs, \text{Vcons } y\ ys) \}$ .

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

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

**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.

The **Below** construction is inefficient!

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

The **Below** construction is inefficient!

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

- ▶ Same setup, the recursor is now of type  $\Pi \Delta (y : I \overline{\Delta}), R y x \rightarrow f_{\text{comp}} y$ .

The **Below** construction is inefficient!

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

- ▶ Same setup, the recursor is now of type  $\Pi \Delta (y : I \overline{\Delta}), R y x \rightarrow f_{\text{comp}} y$ .
- ▶ General definition of direct subterm:  
 $I_{\text{sub}} : \Pi \Delta_l \Delta_r, I \overline{\Delta_l} \rightarrow I \overline{\Delta_r} \rightarrow \text{Prop}$

The **Below** construction is inefficient!

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

- ▶ Same setup, the recursor is now of type  $\Pi \Delta (y : I \overline{\Delta}), R y x \rightarrow f_{\text{comp}} y$ .
- ▶ General definition of direct subterm:  
 $I_{\text{sub}} : \Pi \Delta_l \Delta_r, I \overline{\Delta_l} \rightarrow I \overline{\Delta_r} \rightarrow \text{Prop}$
- ▶ Curry/uncurry transformation to use the usual fixpoint combinator on homogeneous relations.

The **Below** construction is inefficient!

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

- ▶ Same setup, the recursor is now of type  $\Pi \Delta (y : I \overline{\Delta}), R y x \rightarrow f_{\text{comp}} y$ .
- ▶ General definition of direct subterm:  
 $I_{\text{sub}} : \Pi \Delta_l \Delta_r, I \overline{\Delta_l} \rightarrow I \overline{\Delta_r} \rightarrow \text{Prop}$
- ▶ Curry/uncurry transformation to use the usual fixpoint combinator on homogeneous relations.
- ▶ Extracts efficiently, but proof search a bit more complicated than **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  $\{A B n\} (v : \text{vector } (A \times B) n)$   
:  $\text{vector } A n \times \text{vector } B n :=$   
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  $v := \{$   
|  $(\text{pair } xs ys) := (\text{Vcons } x xs, \text{Vcons } y ys) \}.$

1 Dependent pattern-matching compilation

2 Recursion

- The Below way
- Subterm relations

3 Reasoning support

- Equations
- Elimination principle
- Eliminating calls

- ▶ Equations hold definitionally in  $\text{CCI} + \text{K}$
- ▶ 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.

# 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 }.
```

# Generated mutual induction principle

```
Check(filter_ind_mut :  
  ∀ (P : ∀ (A : Type) (l : list A) (p : A → bool), filter_comp l p → Prop)  
  (P0 : ∀ (A : Type) (a : A) (l : list A) (p : A → bool),  
    bool → filter_comp (a :: l) p → Prop),  
  
  (∀ A p, P A [] p []) →  
  
  (∀ A a l p,  
    filter_ind_1 A a l p (p a) (filter_obligation_2 (@filter) A a l p (p a)) →  
    P0 A a l p (p a) (filter_obligation_2 (@filter) A a l p (p a)) →  
    P A (a :: l) p (filter_obligation_2 (@filter) A a l p (p a))) →  
  
  (∀ A a l p, filter_ind A l p (filter l p) →  
    P A l p (filter l p) → P0 A a l p true (a :: filter l p)) →  
  (∀ A a l p, filter_ind A l p (filter l p) →  
    P A l p (filter l p) → P0 A a l p false (filter l p)) →  
  
  ∀ A l p (f3 : filter_comp l p), filter_ind A l p f3 → P A l p f3).
```

# 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 }.
```

# 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)).
```

The elimination principle can only be applied usefully to calls with solely variable arguments.

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

# Eliminating calls

The elimination principle can only be applied usefully to calls with solely variable arguments.

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

Use abstraction by equalities again:

$$\begin{aligned} & (\lambda (l \ l' : \text{list } A) (r : \text{app}_{\text{comp}} l \ l'), \\ & \quad l' = [] \rightarrow r = \text{app } l [] \rightarrow \text{app } l [] = l) \\ & \quad l [] (\text{app } l []) \end{aligned}$$

Directly apply the elimination principle and simplify the equations.

A function definition package handling:

- ▶ Full, nested dependent pattern-matching
- ▶ Structural and well-founded recursion on dependent types
- ▶ 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: 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, support for measures
- ▶ Efficiency, a primitive handling of dependent elimination internalizing K would help.
- ▶ Move to `eq_dep` instead of `JMeq`? Necessary to use decidable instances of K.



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

# Elimination principle: inductive graph

For  $f.l : \Pi \Delta, f_{\text{comp}} \overrightarrow{t}$  we generate  $f.l_{\text{ind}} : \Pi \Delta, f_{\text{comp}} \overrightarrow{t} \rightarrow \text{Prop}$   
and prove  $\Pi \Delta, f.l_{\text{ind}} \overline{\Delta} (f.l \overline{\Delta})$ .

$\text{ABSREC}(f, t)$  abstracts all the calls to  $f_{\text{comp\_proj}}$  from the term  $t$ ,  
returning a new derivation  $\Gamma' \vdash t'$  where  $\Gamma'$  contains bindings of the  
form  $x : \Pi \Delta, f_{\text{comp}} \overrightarrow{t}$  for all the recursive calls.

Define  $\text{HYPS}(\Gamma)$  by a map to produce the corresponding inductive  
hyps of the form  $H_x : \Pi \Delta, f_{\text{ind}} \overrightarrow{t} (x \overline{\Delta})$ .

Direct translation from the splitting tree:

- ▶  $\text{Split}(c, x, s), \text{Rec}(v, s)$  : collect the constructors for the subsplitting(s)  $s$ , if any.
- ▶  $\text{Compute}(\Delta \vdash \vec{p} : \Gamma, rhs)$  : By case on  $rhs$ :
  - ▶  $\text{Program}(t)$  : Compute  $\Psi \vdash t' = \text{ABSREC}(f, t)$  and return the statement

$$\Pi \Delta \Psi \text{HYPS}(\Psi), \mathbf{f.l.ind} \vec{p} t'$$

- ▶  $\text{Refine}(t, \Delta' \vdash \vec{v}^x, x, \vec{v}_x : \Delta^x, x : \tau, \Delta_x, l.n, s)$  :  
Compute  $\Psi \vdash t' = \text{ABSREC}(f, t)$  and return:

$$\Pi \Delta \Psi \text{HYPS}(\Psi) (\mathbf{res} : \mathbf{f.comp} \vec{p}) \\ \mathbf{f.l.n.ind} \overline{\Delta^x} t' \overline{\Delta_x} \mathbf{res} \rightarrow \mathbf{f.l.ind} \vec{p} \mathbf{res}$$

We continue with the generation of the  $\mathbf{f.l.n.ind}$  graph.