EQUATIONS
A Dependent Pattern-Matching Suite

MATTHIEU SOZEAU

INRIA Paris - PPS

May 31 2012
IOC, Tallinn, Estonia

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 operatinality
- 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 $t, \tau ::= x \mid \lambda x : \tau, t \mid \Pi x : \tau, \tau' \mid \ldots$

binding $d ::= (x : \tau) \mid (x := t : \tau)$

context $\Gamma, \Delta ::= \rightarrow d$

user pattern $up ::= x \mid C up \mid ?(t)$

user node $n ::= := t \mid :=! x \mid \text{with } t ::= \{ \overrightarrow{c} \}$

user clause $c ::= f up n$

program $prog ::= f \Gamma : \tau ::= \overrightarrow{c}$
Searching for a splitting tree

For $f \Delta : \tau$ we define $f_{\text{comp}} \Delta := \tau$, so $f : \Pi \Delta, f_{\text{comp}} \Delta$. 
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 $p ::= x \mid \text{C } \overrightarrow{p} \mid ?(t)$
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 $f \Delta : f_{\text{comp}} \overline{\Delta} := \overrightarrow{p}$ ..., 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 \((n \ m : \text{nat}) : \{ n = m \} + \{ n \neq m \} :=
\)
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 : \text{nat} \vdash n m : n m : \text{nat}, n, [
Split(m : \text{nat} \vdash O m : n m : \text{nat}, m, [
  Compute(\vdash O O : n m : \text{nat}, \text{Program(left eq_refl))},
  Compute(m : \text{nat} \vdash O (S m) : n m : \text{nat}, \text{Program(right _))})
),
Split(n m : \text{nat} \vdash (S n) m : n m : \text{nat}, m, [
  Compute(n : \text{nat} \vdash (S n) O : n m : \text{nat}, \ldots),
  Compute(n m : \text{nat} \vdash (S n) (S m) : n m : \text{nat},
    \text{Refine(equal n m,}
      \text{idsubst(n m : \text{nat}, x : \{ n = m \} + \{ n \neq m \}), l, \ldots))]])]}
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 $\Pi \Delta, f_{\text{comp}} ps$.

- **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 $\Pi \Delta, f_{\text{comp}} ps$.

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

- **Program($t$):** witnessed by the term.
For each node with context map $\Delta \vdash ps : \Gamma$ we generate an obligation of type $\Pi \Delta, f_{\text{comp}} \ ps$.

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

- **Program($t$):** witnessed by the term.

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

$$\Delta_x \triangleq \Delta^t, x_t : \tau, \Delta_t[t/x_t] \text{ s.t. } \left\{ \begin{array}{l} \Delta^t, \Delta_t \sim \Delta \\ \Delta_x \vdash (f_{\text{comp}} \overrightarrow{p})[t/x_t] : \text{Type} \end{array} \right.$$
Consider a current problem $\Delta \vdash \overrightarrow{p} : \Gamma$ and a user clause $f \overrightarrow{u}$ with $t_{pre} := \{ e \}$ matching it. We typecheck $t_{pre}$ into $t : \tau$ and use strengthening and abstraction to find a new context

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

Using the clauses $e$ we then build a subcovering $s$ of the identity context map $c = \text{idsubst}(\Delta_x)$ and return $\text{Refine}(t, c, \ell.n, s)$. 
Consider a current problem $\Gamma \vdash \overrightarrow{p} : \Gamma$ and a user clause $f \overleftarrow{u} \overrightarrow{p}$ with $t_{pre} := \{ e \}$ matching it. We typecheck $t_{pre}$ into $t : \tau$ and use strengthening and abstraction to find a new context

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

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

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

we can then build:

$$
(\lambda \Delta, \ell.n \overrightarrow{\Delta^t} t \overrightarrow{\Delta_t}) : \prod \Delta, f_{comp} \overrightarrow{p}
$$
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!
Fixpoint Below_nat \((P : \text{nat} \rightarrow \text{Type}) (n : \text{nat}) : \text{Type} :=
\)
match \(n\) with
| 0 \Rightarrow ()
| S \(n'\) \Rightarrow (P \(n'\) \times \text{Below_nat} P \(n'\))
end\%\text{type}.\)
Fixpoint Below_nat \( (P : \text{nat} \rightarrow \text{Type}) (n : \text{nat}) : \text{Type} := \)
\[
\begin{align*}
\text{match } n \text{ with} \\
| 0 & \Rightarrow () \\
| S \ n' & \Rightarrow (P \ n' \times \text{Below_nat } P \ n')
\end{align*}
\] end%type.

Fixpoint below_nat \( (P : \text{nat} \rightarrow \text{Type}) \)
\( (\text{step} : \prod n : \text{nat}, \text{Below_nat } P \ n \rightarrow P \ n) \)
\( (n : \text{nat}) : \text{Below_nat } P \ n. \)
Fixpoint Below_nat \( (P : \text{nat} \rightarrow \text{Type}) (n : \text{nat}) : \text{Type} := \;\)
\[
\text{match } n \text{ with }
\mid 0 \Rightarrow ()
\mid S \; n' \Rightarrow (P \; n' \times \text{Below_nat} \; P \; n')
\text{end}\%	ext{type.}
\]

Fixpoint below_nat \( (P : \text{nat} \rightarrow \text{Type}) \)
\( (step : \Pi \; n : \text{nat}, \text{Below_nat} \; P \; n \rightarrow P \; n) \)
\( (n : \text{nat}) : \text{Below_nat} \; P \; n. \)

Definition rec_nat \( (P : \text{nat} \rightarrow \text{Type}) \)
\( (step : \Pi \; n : \text{nat}, \text{Below_nat} \; P \; n \rightarrow P \; n) \)
\( (n : \text{nat}) : P \; n := \; step \; n \; (\text{below_nat} \; P \; \text{step} \; n). \)
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 : vector (A \times B) n) : vector A n \times 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, f_{\text{comp}} \vec{t}) x.
Equations unzip \{A \ B \ n\} (v : vector (A \times B) n) : vector A n \times 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 \((\Pi \Delta, f_{\text{comp}} t) x\).
- Each recursive occurrence of \(f\) is transformed to a trivial projection \(f_{\text{comp-proj}} : \Pi \Delta \{p : f_{\text{comp}} \Delta\}, f_{\text{comp}} \Delta\).
Equations unzip \( \{ A \ B \ n \} (v : \text{vector} (A \times B) \ n) : \text{vector} A \ n \times \text{vector} B \ n \):

\[
\begin{align*}
\text{unzip} \ A \ B \ n \ v \ & \text{by rec} \ v := \\
\text{unzip} \ A \ B \ ?(O) \ Vnil & := (Vnil, Vnil) ; \\
\text{unzip} \ A \ B \ ?(S \ n) \ (Vcons \ (\text{pair} \ x \ y) \ n \ v) \ & \text{with} \ \text{unzip} \ v := \\
& | (\text{pair} \ xs \ ys) := (Vcons \ x \ xs, Vcons \ y \ ys) \} .
\end{align*}
\]

- **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 \( f_{\text{comp-proj}} : \Pi \ \Delta \ \{ p : f_{\text{comp}} \overrightarrow{\Delta} \}, f_{\text{comp}} \overrightarrow{\Delta} \).
- Proof search for \( f_{\text{comp}} \) goals appearing as obligations, unfolding **Below** hypotheses.
Subterm relations and well-founded recursion

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:
\[
I_{\text{sub}} : \Pi \Delta_l \Delta_r, I \overline{\Delta_l} \rightarrow I \overline{\Delta_r} \rightarrow \text{Prop}
\]
Subterm relations and well-founded recursion

Below is inefficient!

Use **well-founded** recursion on the subterm relation for inductive families \( \mathcal{I} : \prod \Delta \).

- **General definition of direct subterm:**
  \[
  \mathcal{I}_{sub} : \prod \Delta_l \Delta_r, \; \mathcal{I} \Delta_l \to \mathcal{I} \Delta_r \to \text{Prop}
  \]

- **Wrap the inductive type with its indices in a sigma and define an homogeneous relation on:**
  \[
  \mathcal{I}_{sub} : \text{relation} \left( \Sigma \Delta, \mathcal{I} \overline{\Delta} \right)
  \]
Below is inefficient!

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

- **General definition of direct subterm:**
  \[ I_{sub} : \Pi \Delta_l \Delta_r, I(\Delta_l) \rightarrow I(\Delta_r) \rightarrow \text{Prop} \]

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

- **Extracts efficiently, proof search only a tiny bit more complicated than for Below**
Derive Subterm for vector.
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 \((A : Type)\)

: \(\forall \ n \ m : \text{nat}, \ \text{vector} \ A \ n \to \text{vector} \ A \ m \to \text{Prop} :\)

\[
\text{vector\_strict\_subterm}\_1\_1 : \forall (a : A) (n : \text{nat}) (v : \text{vector} \ A \ n),
\]

vector\_strict\_subterm \(A \ n \ (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 \(A \ B \ n \ v \ \text{by rec} \ v :\)

| unzip \(A \ B \ ?(O) \ Vnil :\) \(= (Vnil, \ Vnil) ;\)

unzip \(A \ B \ ?(S \ n) \ (\text{Vcons} \ (\text{pair} \ x \ y) \ n \ v) \ \text{with unzip} \ v :\) \(= \{\)

| (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 $\text{CCI} + K$
- Equations for $\text{with}$ nodes are just proxies to the helper function $f.\ell$.
- 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

\[
\text{filter } \{ A \} (l : \text{list } A) (p : A \rightarrow \text{bool}) : \text{list } A :=
\]

\[
\text{filter } A \text{ nil } p := \text{nil } ;
\]

\[
\text{filter } A (\text{cons } a \ l) p \text{ with } p \ a := \{
\]
\[
| \text{true} := a :: \text{filter } l \ p ;
\]
\[
| \text{false} := \text{filter } l \ p \}.
\]

Check \((\text{filter\_elim})\):

\[
\forall (P : \forall (A : \text{Type}) (l : \text{list } A) (p : A \rightarrow \text{bool}), \text{filter\_comp } l \ p \rightarrow \text{Prop}),
\]

\[
\text{let } P0 := \text{fun (A : Type) (a : A) (l : list A) (p : A \rightarrow \text{bool})}
\]

\[
\text{(refine : bool) (res : filter\_comp (a :: l) p) \Rightarrow}
\]

\[
p \ a = \text{refine} \rightarrow P \ A \ (a :: l) \ p \ res
\]

in

\[
(\forall (A : \text{Type}) (p : A \rightarrow \text{bool}), P \ A [] p []) \rightarrow
\]

\[
(\forall (A : \text{Type}) (a : A) (l : \text{list } A) (p : A \rightarrow \text{bool}),
\]

\[
P \ A \ l \ p \ (\text{filter } l \ p) \rightarrow P0 \ A \ a \ l \ p \ \text{true (a :: filter } l \ p)) \rightarrow
\]

\[
(\forall (A : \text{Type}) (a : A) (l : \text{list } A) (p : A \rightarrow \text{bool}),
\]

\[
P \ A \ l \ p \ (\text{filter } l \ p) \rightarrow P0 \ A \ a \ l \ p \ \text{false (filter } l \ p)) \rightarrow
\]

\[
\forall (A : \text{Type}) (l : \text{list } A) (p : A \rightarrow \text{bool}), P \ A \ l \ p \ (\text{filter } l \ p)) .
\]

18
How to prove using \texttt{filter\_elim}?

\[ \Pi A \ (l : \text{list } A), \ \text{filter} \ (\lambda_, \ \text{false}) \ l = [] \]
Eliminating calls

How to prove using \texttt{filter\_elim}?

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

\[ \Rightarrow \ \text{Abstract by equalities:} \]

\[ (\lambda \ (l : \ \text{list} \ A) \ (p : \ A \ \to \ \text{bool}) \ (r : \ \text{filter}_{\text{comp}} \ l \ p), \]
\[ p = (\lambda \_, \ \text{false}) \ \to \ r = \ \text{filter} \ l \ p \ \to \ \text{filter} \ (\lambda \_, \ \text{false}) \ l = [] \]
\[ l \ (\lambda \_, \ \text{false}) \ (\text{filter} \ l \ (\lambda \_, \ \text{false})) \]
How to prove using \textit{filter\_elim}?

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

$$\Rightarrow$$ Abstract by equalities:

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

$$l \ (\lambda_, \ \text{false}) \ (\text{filter} \ l \ (\lambda_, \ \text{false}))$$

$$\Rightarrow$$ 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.
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 $K$. 
http://mattam.org/research/coq/equations.en.html