

EQUATIONS: a dependent pattern-matching compiler

MATTHIEU SOZEAU

INRIA Paris

TYPES'10

October 15th 2010

Warsaw, Poland



- ▶ 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!

Examples

Equations $\text{vmap} \{A B\} (f : A \rightarrow B) \{n\} (v : \text{vector } A n) :$
 $\text{vector } B n :=$
 $\text{vmap } A B f ?(0) \text{Vnil} := \text{Vnil} ;$
 $\text{vmap } A B f ?(S n) (\text{Vcons } a n v) := \text{Vcons } (f a) (\text{vmap } f v).$

Examples

Equations $\text{vmap} \{A B\} (f : A \rightarrow B) \{n\} (v : \text{vector } A \ n) :$
 $\text{vector } B \ n :=$
 $\text{vmap } A \ B \ f \ ?(0) \ \text{Vnil} := \text{Vnil} ;$
 $\text{vmap } A \ B \ f \ ?(S \ n) (\text{Vcons } a \ n \ v) := \text{Vcons } (f \ a) (\text{vmap } f \ v).$

Equations $\text{vtail} \{A \ n\} (v : \text{vector } A \ (S \ n)) : \text{vector } A \ n :=$
 $\text{vtail } A \ n (\text{Vcons } a \ n \ v') := v'.$

Examples

Equations $\text{vmap} \{A B\} (f : A \rightarrow B) \{n\} (v : \text{vector } A n) :$
 $\text{vector } B n :=$
 $\text{vmap } A B f ?(0) \text{Vnil} := \text{Vnil} ;$
 $\text{vmap } A B f ?(S n) (\text{Vcons } a n v) := \text{Vcons } (f a) (\text{vmap } f v).$

Equations $\text{vtail} \{A n\} (v : \text{vector } A (S n)) : \text{vector } A n :=$
 $\text{vtail } A n (\text{Vcons } a n v') := v'.$

Equations $\text{diag} \{A n\} (v : \text{vector } (\text{vector } A n) n) : \text{vector } A n :=$
 $\text{diag } A 0 \text{Vnil} := \text{Vnil} ;$
 $\text{diag } A (S n) (\text{Vcons } (\text{Vcons } a n v) n v') :=$
 $\text{Vcons } a (\text{diag } (\text{vmap } \text{vtail } v')).$

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}_{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, ...)))]])])
```

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 ℓ .

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

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

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** $(\Pi\ \Delta, f_{\text{comp}}\ \vec{t})\ x$.
- ▶ 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_{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 $CCI + K$
- ▶ 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.

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>