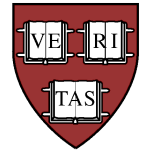
Equations: a dependent pattern-matching compiler

Matthieu Sozeau

Harvard University

GT Types & Réalisabilité
January 13th 2010
Paris, France



Overview

- EPIGRAM-style pattern-matching definitions with `with` and `rec` nodes
- Propositional equations for definitional equalities
- Elimination principle and support for applying it

DEMO

1. Dependent pattern-matching compilation
   - Case analysis
   - `with` in detail

2. Recursion
   - The Below way
   - Subterm relations

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

Compilation setup

Elaboration into $\mathrm{CIC} + \mathsf{K}$

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	$::=$	$\overrightarrow{d}$
`program`	$prog$	$::=$	$\mathrm{f}\ \Gamma\ :\ \tau\ :=\ \overrightarrow{c}$
`user clause`	c	$::=$	$\mathrm{f}\ \overrightarrow{up}\ n$
`user pattern`	up	$::=$	$x \mid \mathsf{C}\ \overrightarrow{up} \mid \texttt{?}(t)$
`user node`	n	$::=$	$:=\ t \mid :=\texttt{!}\ x \mid \texttt{with}\ t := \{\ \overrightarrow{c}\ \}$

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

Searching for a splitting tree

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

Goal Find a covering of the context map $\Delta \vdash \overline{\Delta} : \Delta$. This will compile to a term of type $\Pi\Delta, \mathrm{f}_{\mathsf{comp}}\ \overline{\Delta}$

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, ...))])])
```

Translation from the splitting to Coq

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

Translation from the splitting to Coq

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

- 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 $\Pi\ \Delta, \mathrm{f}_{\mathsf{comp}}\ ps$.

- 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 $\Pi\ \Delta, \mathrm{f}_{\mathsf{comp}}\ ps$.

- 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.
- 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 $\mathrm{f}\ \overrightarrow{up}$ `with` $t_{pre} := \{\ e\ \}$ matching it. We typecheck t_{pre} into $t : \tau$ and use strenghtening and abstraction to find a new context

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

With nodes in detail

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

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

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

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

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 $\mathrm{f}\ \overrightarrow{up}$ `with` $t_{pre} := \{\ e\ \}$ matching it. We typecheck t_{pre} into $t : \tau$ and use strenghtening and abstraction to find a new context

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

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

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

and return Refine$(t, c, \ell.n, s)$.
Compilation produces $\ell.n : \Pi\ \Delta^t\ (x_t : \tau_\Delta)\ \Delta_t[t/x_t], (\mathrm{f}_{\mathsf{comp}}\ \overrightarrow{p})[t/x_t]$, we build

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

1. Dependent pattern-matching compilation
   - Case analysis
   - `with` in detail

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

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.

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

below_nat : Π (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 $\mathrm{Below}\ (\Pi\ \Delta, \mathrm{f}_{\mathsf{comp}}\ \overrightarrow{t})\ x$.

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 $\mathrm{Below}\ (\Pi\ \Delta, \mathrm{f}_{\mathsf{comp}}\ \overrightarrow{t})\ x$.
- Each recursive occurence of f is transformed to a trivial projection $\mathrm{f}_{\mathsf{comp\_proj}} : \Pi\ \Delta\ \{p : \mathrm{f}_{\mathsf{comp}}\ \overline{\Delta}\}, \mathrm{f}_{\mathsf{comp}}\ \overline{\Delta}$.

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

Subterm relations and well-founded recursion

The Below construction is inefficient!
Use **well-founded** recursion on the subterm relation for inductive families $\mathsf{I} : \Pi\ \Delta, s$.

Subterm relations and well-founded recursion

The Below construction is inefficient!
Use **well-founded** recursion on the subterm relation for inductive families $\mathsf{I} : \Pi\ \Delta, s$.

- Same setup, the recursor is now of type $\Pi\ \Delta\ (y : \mathsf{I}\ \overline{\Delta}), R\ y\ x \to \mathrm{f}_{\mathsf{comp}}\ y$.

Subterm relations and well-founded recursion

The Below construction is inefficient!
Use **well-founded** recursion on the subterm relation for inductive families $\mathsf{I} : \Pi\ \Delta, s$.

- Same setup, the recursor is now of type $\Pi\ \Delta\ (y : \mathsf{I}\ \overline{\Delta}), R\ y\ x \to \mathrm{f}_{\mathsf{comp}}\ y$.
- General definition of direct subterm: $\mathsf{I}_{sub} : \Pi\ \Delta_l\ \Delta_r, \mathsf{I}\ \overline{\Delta_l} \to \mathsf{I}\ \overline{\Delta_r} \to \mathtt{Prop}$

Subterm relations and well-founded recursion

The Below construction is inefficient!
Use **well-founded** recursion on the subterm relation for inductive families $\mathsf{I} : \Pi\ \Delta, s$.

- Same setup, the recursor is now of type $\Pi\ \Delta\ (y : \mathsf{I}\ \overline{\Delta}), R\ y\ x \to \mathrm{f}_{\mathsf{comp}}\ y$.
- 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 in a sigma and define an homogeneous relation on the sigma type from the heterogeneous subterm relation.

Subterm relations and well-founded recursion

The Below construction is inefficient!
Use **well-founded** recursion on the subterm relation for inductive families $\mathsf{I} : \Pi\ \Delta, s$.

- Same setup, the recursor is now of type $\Pi\ \Delta\ (y : \mathsf{I}\ \overline{\Delta}), R\ y\ x \to \mathrm{f}_{\mathsf{comp}}\ y$.
- 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 in a sigma and define an homogeneous relation on the sigma type from the heterogeneous subterm relation.
- 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 : Type)
  : ∀ H H0 : nat, vector A H → vector A H0 → Prop :=
    vector_strict_subterm_1_1 : ∀ (a : A) (n : nat) (H : vector A n),
      vector_strict_subterm A n (S n) H (Vcons a H).

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)
  : ∀ H H0 : nat, vector A H → vector A H0 → Prop :=
    vector_strict_subterm_1_1 : ∀ (a : A) (n : nat) (H : vector A n),
      vector_strict_subterm A n (S n) H (Vcons a H).

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
   - Case analysis
   - `with` in detail
2. Recursion
   - The Below way
   - Subterm relations
3. Reasoning support
   - Equations
   - Elimination principle
   - Eliminating calls

Equations

- Equations hold definitionally in $\mathrm{CCI} + \mathrm{K}$
- Equations for `with` nodes are just proxies to the helper function $\mathrm{f}.\ell$.
- 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.

Elimination principle: inductive graph

For $\mathrm{f}.\ell : \Pi\ \Delta, \mathrm{f}_{\mathsf{comp}}\ \overrightarrow{t}$ we generate $\mathrm{f}.\ell_{\mathsf{ind}} : \Pi\ \Delta, \mathrm{f}_{\mathsf{comp}}\ \overrightarrow{t} \to \mathtt{Prop}$ and prove $\Pi\ \Delta, \mathrm{f}.\ell_{\mathsf{ind}}\ \overline{\Delta}\ (\mathrm{f}.\ell\ \overline{\Delta})$.

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

Define $\textsc{Hyps}(\Gamma)$ by a map to produce the corresponding inductive hyps of the form $H_x : \Pi\ \Delta, \mathrm{f}_{\mathsf{ind}}\ \overrightarrow{t}\ (x\ \overline{\Delta})$.

Inductive graph constructors

Direct translation from the splitting tree:

- Split(c, x, s), Rec(v, s) : collect the constructors for the subsplitting(s) s, if any.
- Compute$(\Delta \vdash \overrightarrow{p} : \Gamma, rhs)$: By case on rhs:
  - Program(t) : Compute $\Psi \vdash t' = \textsc{AbsRec}(f, t)$ and return the statement
    $$\Pi\ \Delta\ \Psi\ \textsc{Hyps}(\Psi),\ \mathrm{f}.\ell_{\mathsf{ind}}\ \overrightarrow{p}\ t'$$
  - Refine$(t, \Delta' \vdash \overrightarrow{v}^x, x, \overrightarrow{v}_x : \Delta^x, x : \tau, \Delta_x, \ell.n, s)$:
    Compute $\Psi \vdash t' = \textsc{AbsRec}(f, t)$ and return:
    $$\Pi\ \Delta\ \Psi\ \textsc{Hyps}(\Psi)\ (res : \mathrm{f}_{\mathsf{comp}}\ \overrightarrow{p})\ \mathrm{f}.\ell.\mathrm{n}_{\mathsf{ind}}\ \overline{\Delta^x}\ t'\ \overline{\Delta_x}\ res \to \mathrm{f}.\ell_{\mathsf{ind}}\ \overrightarrow{p}\ res$$
    We continue with the generation of the $\mathrm{f}.\ell.\mathrm{n}_{\mathsf{ind}}$ graph.

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) (H : filter_comp (a :: l) p) ⇒
      p a = refine → P A (a :: l) p H
  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)).
```

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).
```

Eliminating calls

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

$$\Pi\ A\ (l : \mathsf{list}\ A),\ \mathrm{app}\ l\ [] = l$$

Eliminating calls

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

$$\Pi\ A\ (l : \mathsf{list}\ A),\ \mathrm{app}\ l\ [] = l$$

Use the same "abstraction by equalities" technique used in dependent elimination to solve this. We can abstract:

$$(\lambda\ (l\ l' : \mathsf{list}\ A)\ (r : \mathrm{app}_{\mathsf{comp}}\ l\ l'),\ l' = [] \to \mathrm{app}\ l\ l' = l)\ l\ []\ (\mathrm{app}\ l\ [])$$

Directly apply the elimination principle and simplify the equations.

Conclusion

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

Tested on a bit-fiddling library: less boilerplate, shorter proofs.

Perspectives

- Treatment of non-constructor indices and constraints
- Mutual recursion, support for measures
- Efficiency, a primitive handling of dependent elimination internalizing K would help (hint !)
- Move to eq_dep instead of JMeq?

The End