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

\textbf{Elaboration into CIC + K}

Three phases:

1. Generation of a splitting tree from the clauses
2. Translation from the splitting tree to \texttt{Coq} terms with holes
3. Proofs of the obligations using a mix of ML and $\mathcal{L}_{\text{tac}}$ code
For $f \Delta : \tau$ we define $f_{\text{comp}} \Delta := \tau$, so $f : \Pi \Delta, f_{\text{comp}} \Delta$. 
### Goal

Find a covering of the context map $\Delta \vdash \overline{\Delta} : \Delta$. This will compile to a term of type $\Pi \Delta, f_{\text{comp}} \overline{\Delta}$
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 _))]),
  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 $\Pi \Delta, f_{\text{comp}} ps$.

- **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 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 $f \overrightarrow{up}$ with $t_{pre} := \{ e \}$ matching it. We typecheck $t_{pre}$ into $t : \tau$ and use strenghtening and abstraction to find a new context $\Delta^t, x_t : \tau, \Delta_t[t/x_t]$ such that $\Delta^t, \Delta_t \sim \Delta$.
Consider a current problem $\Delta \vdash \vec{p} : \Gamma$ and a user clause $f \rightleftharpoons \vec{w} \text{ with } t_{\text{pre}} := \{ e \} \text{ matching it. We typecheck } t_{\text{pre}} \text{ into } t : \tau \text{ and use strengthening and abstraction to find a new context}$

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

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 $\text{Refine}(t, c, \ell.n, s)$. 
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 $\text{Refine}(t, c, \ell.n, s)$.

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

$$
(\lambda \Delta, \ell.n \overrightarrow{\Delta^t t \Delta_t}) : \Pi \Delta, f_{\text{comp} \overrightarrow{p}}
$$
Dependent pattern-matching compilation

- Case analysis
- With in detail

Recursion

- The Below way
- Subterm relations

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!
Introduced by McBride and McKinna.

\[
\text{Fixpoint } \text{Below}_\text{nat} \ (P : \text{nat} \to \text{Type}) \ (n : \text{nat}) : \text{Type} :=
\begin{align*}
\text{match } n \ \text{with} \\
| \ 0 \Rightarrow () \\
| \ S \ n' \Rightarrow (P \ n' \times \text{Below}_\text{nat} \ P \ n')
\end{align*}
\text{end}\%^\text{type}.
\]
The Below way

Introduced by McBride and McKinna.

\[ \text{Fixpoint Below\_nat} \ (P : \text{nat} \rightarrow \text{Type}) \ (n : \text{nat}) : \text{Type} := \]
match \( n \) with
\[ \mid 0 \Rightarrow () \]
\[ \mid S \ n' \Rightarrow (P \ n' \times \text{Below\_nat} \ P \ n') \]
end%type.

\[ \text{below\_nat} : \Pi \ (P : \text{nat} \rightarrow \text{Type}) \]
\[ (\text{step} : \Pi \ n : \text{nat}, \text{Below\_nat} \ P \ n \rightarrow P \ n) \]
\[ (n : \text{nat}) : \text{Below\_nat} \ P \ n \]
The Below way

Introduced by McBride and McKinna.

Fixpoint Below_nat \( (P : \text{nat} \to \text{Type}) (n : \text{nat}) : \text{Type} := \)
\[
\begin{align*}
\text{match } n \text{ with} \\
\phantom{\text{match } n} | 0 & \Rightarrow () \\
\phantom{\text{match } n} | S \ n' & \Rightarrow (P \ n' \times \text{Below_nat} \ P \ n')
\end{align*}
\]
end%type.

below_nat : \(\Pi\) \( (P : \text{nat} \to \text{Type}) \)
\( (\text{step} : \Pi \ n : \text{nat}, \text{Below_nat} \ P \ n \to P \ n) \)
\( (n : \text{nat}) : \text{Below_nat} \ P \ n \)

Definition rec_nat \( (P : \text{nat} \to \text{Type}) \)
\( (\text{step} : \Pi \ n : \text{nat}, \text{Below_nat} \ P \ n \to P \ n) \)
\( (n : \text{nat}) : P \ n := \text{step} \ n \ (\text{below_nat} \ P \ \text{step} \ n). \)
Equations unzip \{A \times B\} (v : vector \(A \times B\) 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 \times B \} \ (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) 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 \( \text{Below} \ (\Pi \ \Delta, f_{\text{comp}} \overrightarrow{t} ) \ x \).
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}} \overrightarrow{t}) x.

- Each recursive occurrence of f is transformed to a trivial projection \( f_{\text{comp-proj}} : \Pi \Delta \{p : f_{\text{comp}} \overrightarrow{\Delta}\}, f_{\text{comp}} \overrightarrow{\Delta}.\)
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_{comp} \stackrel{t}{\rightarrow}) x.

▶ Each recursive occurrence of f is transformed to a trivial
projection \( f_{comp-proj} : \Pi \Delta \ \{p : f_{comp} \overline{\Delta}\}, f_{comp} \overline{\Delta} \).

▶ 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 $I : \Pi \Delta, s$. 
The **Below** construction is inefficient!

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

- Same setup, the recursor is now of type
  \[
  \prod \Delta (y : I \, \overline{\Delta}), \, R \, y \, x \to f_{\text{comp}} \, y.
  \]
The **Below** construction is inefficient!

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

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

- General definition of direct subterm:
  \( l_{\text{sub}} : \Pi \Delta_l \ Delta_r, l \overline{\Delta_l} \rightarrow l \overline{\Delta_r} \rightarrow \text{Prop} \)
The **Below** construction is inefficient!
Use **well-founded** recursion on the subterm relation for inductive families $l : \Pi \ \Delta, s$.

- Same setup, the recursor is now of type
  \[ \Pi \ \Delta \ (y : l \ \overline{\Delta}), R \ y \ x \rightarrow f_{\text{comp}} \ y. \]
- General definition of direct subterm:
  \[ l_{\text{sub}} : \Pi \ \Delta_l \ \Delta_r, l \ \overline{\Delta_l} \rightarrow l \ \overline{\Delta_r} \rightarrow \text{Prop} \]
- Wrap the inductive type in a sigma and define an homogeneous relation on the sigma type from the heterogeneous subterm relation.
The **Below** construction is inefficient! Use **well-founded** recursion on the subterm relation for inductive families $I : \Pi \Delta, s$.

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

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

- 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**.
Derive Subterm for vector.
Derive Subterm for vector.

**Inductive vector\_strict\_subterm (A : Type)**

: \( \forall H H0 : \text{nat}, \text{vector} A H \rightarrow \text{vector} A H0 \rightarrow \text{Prop} := \)

vector\_strict\_subterm\_1_1 : \( \forall (a : A) (n : \text{nat}) (H : \text{vector} A n), \)

vector\_strict\_subterm A n (S n) H (\text{Vcons} a H).

**Check vector\_subterm : \( \forall A : \text{Type}, \text{relation} \{ \text{index} : \text{nat} \& \text{vector} A \text{index} \}. \)**
Derive Subterm for vector.

**Inductive** \texttt{vector\_strict\_subterm} \((A : \text{Type})\)

\[ \forall \, H \, H0 : \text{nat}, \, \text{vector} \, A \, H \rightarrow \text{vector} \, A \, H0 \rightarrow \text{Prop} \,:=\]

\texttt{vector\_strict\_subterm\_1\_1} : \forall \ (a : A) \ (n : \text{nat}) \ (H : \text{vector} \, A \, n),

\texttt{vector\_strict\_subterm} \, A \, n \, (S \, n) \, H \, (\text{Vcons} \, a \, H).

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

**Equations** \texttt{unzip} \{ \text{A} \, \text{B} \, \text{n} \} \ (\text{v} : \text{vector} \, (A \times B) \, n)

\[ : \text{vector} \, A \, n \times \text{vector} \, B \, n \,:=\]

\texttt{unzip} \, A \, B \, n \, v \, \text{by rec} \, v \,:=

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

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

| \, (\text{pair} \, xs \, ys) \,:=\, (\text{Vcons} \, x \, xs, \, \text{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 hold definitionally in CCI + K

Equations for \texttt{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.
For $f.\ell : \Pi \Delta, f_{\text{comp}} \overset{\rightarrow}{\tau}$ we generate $f.\ell_{\text{ind}} : \Pi \Delta, f_{\text{comp}} \overset{\rightarrow}{\tau} \rightarrow \text{Prop}$ and prove $\Pi \Delta, f.\ell_{\text{ind}} \overset{\Delta}{\rightarrow} (f.\ell \Delta)$.

$\text{AbsRec}(f, t)$ 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 $x : \Pi \Delta, f_{\text{comp}} \overset{\tau}{\rightarrow}$ for all the recursive calls.

Define $\text{HypS}(\Gamma)$ by a map to produce the corresponding inductive hyps of the form $H_x : \Pi \Delta, f_{\text{ind}} \overset{\tau}{\rightarrow} (x \Delta)$. 
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**\((\Delta \vdash \overrightarrow{p} : \Gamma, \text{rhs})\) : By case on \(\text{rhs}\):
  - **Program**\((t)\) : Compute \(\Psi \vdash t' = \text{AbsRec}(f, t)\) and return the statement
    \[
    \Pi \Delta \Psi \text{HYPS}(\Psi), \ f.\ell_{\text{ind}} \overrightarrow{p} \ t'
    \]
  - **Refine**\((t, \Delta' \vdash \overrightarrow{u}^x, x, \overrightarrow{u}_x : \Delta^x, x : \tau, \Delta_x, \ell.n, s)\) : Compute \(\Psi \vdash t' = \text{AbsRec}(f, t)\) and return:
    \[
    \Pi \Delta \Psi \text{HYPS}(\Psi) \left( res : f_{\text{comp}} \overrightarrow{p} \right) \\
    f.\ell.n_{\text{ind}} \Delta^x t' \Delta_x res \rightarrow f.\ell_{\text{ind}} \overrightarrow{p} res
    \]

We continue with the generation of the \(f.\ell.n_{\text{ind}}\) graph.
Elimination principle

Equations \( \text{filter} \{ \! \! A \! \! \} (l : \text{list } A) (p : A \rightarrow \text{bool}) : \text{list } A := \)
\[
\text{filter } A \text{ nil } p := \text{ nil } ;
\]
\[
\text{filter } A (\text{cons } a l) p \text{ with } p a := \{ \text{ true } := a :: \text{filter } l p ; \}
\text{ false } := \text{filter } l p \}.
\]
Elimination principle

Equations filter \( \{ A \} \) \((l : \text{list } A) \) \((p : A \rightarrow \text{bool}) : \text{list } A := \)
\[
\text{filter } A \ \text{nil } p := \text{nil} ;
\]
\[
\text{filter } A \ (\text{cons } a \ l) \ p \ \text{with} \ p \ a := \{
\begin{align*}
| \text{true} & := a :: \text{filter } l \ p ; \\
| \text{false} & := \text{filter } l \ p \}
\end{align*}
\]

Check (filter_elim :
\[
\forall P : \forall (A : \text{Type}) (l : \text{list } A) (p : A \rightarrow \text{bool}), \text{filter_comp } l \ p \rightarrow \text{Prop},
\]
\[
\text{let } P0 := \text{fun } (A : \text{Type}) (a : A) (l : \text{list } A) (p : A \rightarrow \text{bool})
\]
\[
(\text{refine } : \text{bool}) (H : \text{filter_comp (a :: l) } p) \Rightarrow
\]
\[
p \ a = \text{refine} \rightarrow P A (a :: l) \ p \ H
\]
in
\[
(\forall (A : \text{Type}) (p : A \rightarrow \text{bool}), P A [] \ p []) \rightarrow
\]
\[
(\forall (A : \text{Type}) (a : A) (l : \text{list } A) (p : A \rightarrow \text{bool}), \ P A l \ p \ (\text{filter } l \ p) \rightarrow P0 A a l \ p \ \text{true} \ (a :: \text{filter } l \ p)) \rightarrow
\]
\[
(\forall (A : \text{Type}) (a : A) (l : \text{list } A) (p : A \rightarrow \text{bool}),
\]
\[
P A l \ p \ (\text{filter } l \ p) \rightarrow P0 A a l \ p \ \text{false} \ (\text{filter } l \ p)) \rightarrow
\]
\[
(\forall (A : \text{Type}) (l : \text{list } A) (p : A \rightarrow \text{bool}), P A l \ p \ (\text{filter } l \ p)).
\]
Generated mutual induction principle

\(\text{Check}(\text{filter\_ind\_mut} : \forall (P : \forall (A : \text{Type}) (l : \text{list} A) (p : A \rightarrow \text{bool}), \text{filter\_comp} l p \rightarrow \text{Prop}) \quad (P0 : \forall (A : \text{Type}) (a : A) (l : \text{list} A) (p : A \rightarrow \text{bool}), \text{bool} \rightarrow \text{filter\_comp} (a :: l) p \rightarrow \text{Prop}),\)

\((\forall A p, P A [] p []) \rightarrow \)

\((\forall A a l p, \text{filter\_ind\_1} A a l p (p a) (\text{filter\_obligation\_2} (@\text{filter}) A a l p (p a)) \rightarrow P0 A a l p (p a) (\text{filter\_obligation\_2} (@\text{filter}) A a l p (p a)) \rightarrow P A (a :: l) p (\text{filter\_obligation\_2} (@\text{filter}) A a l p (p a))) \rightarrow \)

\((\forall A a l p, \text{filter\_ind} A l p (\text{filter} l p) \rightarrow P A l p (\text{filter} l p) \rightarrow P0 A a l p \text{ true} (a :: \text{filter} l p)) \rightarrow (\forall A a l p, \text{filter\_ind} A l p (\text{filter} l p) \rightarrow P A l p (\text{filter} l p) \rightarrow P0 A a l p \text{ false} (\text{filter} l p)) \rightarrow \)

\(\forall A l p (f3 : \text{filter\_comp} l p), \text{filter\_ind} A l p f3 \rightarrow P A l p f3).\)
The elimination principle can only be applied usefully to calls with solely variable arguments.

\[ \Pi A \ (l : \text{list} \ A), \ \text{app} \ l \ [] = l \]
The elimination principle can only be applied usefully to calls with solely variable arguments.

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

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

\[ (\lambda (l \ l' : \text{list } A) \ (r : \text{app}_{\text{comp}} l \ l'), \ l' = [] \rightarrow \text{app } l \ l' = l) \]

\[ l \ [] \ (\text{app } l \ []) \]

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