

# Elaborations in Type Theory

MATTHIEU SOZEAU

Harvard University

DTP'10

July 10th 2010

Edinburgh, UK



Three elaborations:

- ▶ PROGRAM: a more flexible language for subset types.
- ▶ EQUATIONS: dependent pattern-matching and reasoning support.
- ▶ Type Classes: support for overloading and structuring.

- ▶ Full-spectrum dependent types
  - ▶ Single, unified term-type language, SN
  - ▶ Phase distinction issues (for runtime, see Brady, Barras)
- ▶ Core language design:
  - ▶ De Bruijn principle (“small” core, externally checkable terms)
  - ▶ Striving for minimality/purity and “accessibility” of models.
  - ▶ Open-world, generative. Powerful module system.
- ▶ External language design:
  - ▶ Unification is central (implicits, tactics) and incomplete.
  - ▶ Definitional coercion systems for accessibility of the language

- ▶ Proof language design:
  - ▶ Separate tactic language  $\mathcal{L}_{\text{tac}}$ .
  - ▶ Proving tools: proof search, tactics.
  - ▶ Development tools: derived definitions (FUNCTION, Schemes. . .).
- ▶ User interface and interaction: not discussed here.

Elaboration: compiling high-level constructs to the core language, using the metalanguage.

- ✓ Advantages: metatheory done once and for all (just kidding!). Freedom in the transformations, extensibility and modularity.
- ✗ Concerns: “abstraction leaks”, efficiency, correctness.

Compare with:

- ▶ Reflexion methods: less freedom, more assurance, full correctness, smaller scope (but see Epigram 2).
- ▶ “Axiomatic” methods, e.g. Agda’s built-in pattern-matching. Less assurance, more freedom.

**Acknowledgment** McBride and McKinna’s work (OLEG, EPIGRAM), KISS.

Defining and reasoning on functions with:

- ▶ Rich types while separating algorithms and proofs.
- ▶ Generic types, passing information implicitly.
- ▶ Rich data and control flow, keeping information transparently.
- ▶ Complex recursion behaviors and efficient evaluation.
- ▶ Support for reasoning on functions post-hoc: elimination principles and proof tools (search, rewriting).

- 1 PROGRAM
  - Hello World
  - The theory: Subset Coercions
- 2 EQUATIONS
  - Dependent pattern-matching
  - Recursion
  - Reasoning support
- 3 Type Classes
  - Type Classes from HASKELL
  - Type Classes in COQ
- 4 Conclusion

# Programming with tactics

**Lemma** `eucl_dev` :  $\forall n, n > 0 \rightarrow \forall m : \text{nat},$   
 $\{ (q, r) : \text{nat} \times \text{nat} \mid n > r \wedge m = q \times n + r \}.$

**Proof.**

```
intros b H a; pattern a; apply gt_wf_rec; intros n H0.
elim (le_gt_dec b n).
intro lebn.
case (H0 (n - b)); auto with arith.
intros [q r] [g e].
 $\exists$  (S q, r); simpl; auto with arith.
elim plus_assoc.
elim e; auto with arith.
intros gtbn.
 $\exists$  (0, n); simpl; auto with arith.
```

**Qed.**

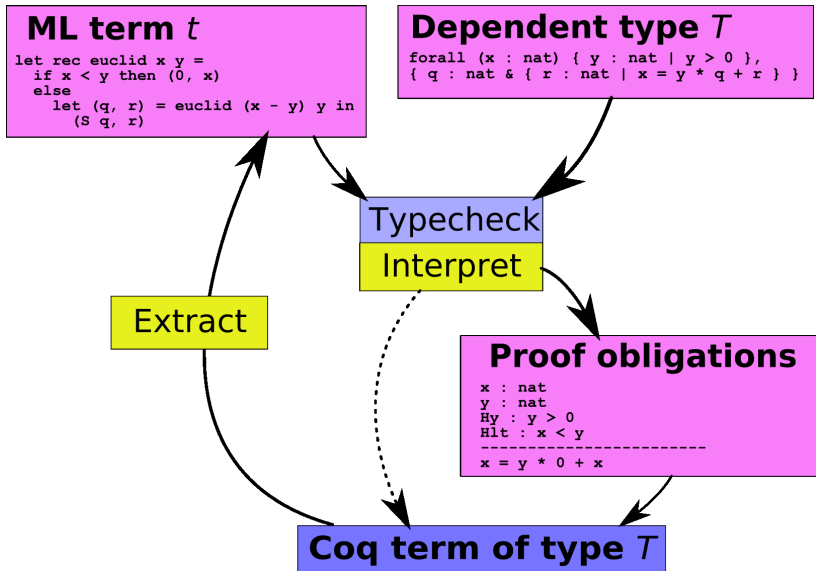
# Programming directly

```
Check ((λ (b : nat) (H : b > 0) (a : nat),
Wf_nat.gt_wf_rec a (λ n : nat, diveucl n b)
(λ (n : nat) (H0 : Π m : nat, n > m → diveucl m b),
sumbool_rec (λ _ : {b ≤ n} + {b > n}, diveucl n b)
(λ lebn : b ≤ n,
diveucl_rec (n - b) b (λ _ : diveucl (n - b) b, diveucl n b)
(λ (q r : nat) (g : b > r) (e : n - b = q × b + r),
divex n b (S q) r g
(eq_ind (b + (q × b + r)) (λ n0 : nat, n = n0)
(eq_ind (n - b) (λ n0 : nat, n = b + n0)
(Minus.le_plus_minus b n lebn) (q × b + r) e)
(b + q × b + r) (Plus.plus_assoc b (q × b) r))))
(H0 (n - b) (Minus.lt_minus n b lebn H)))
(λ gtbn : b > n, divex n b 0 n gtbn (refl_equal n))
(Compare_dec.le_gt_dec b n)))
: Π n : nat, n > 0 → Π m : nat, diveucl m n).
```

We can write programs as usual and still give them rich types.

```
Program Fixpoint div (a : nat) (b : nat | b ≠ 0) { wf lt a } :  
  { (q, r) : nat × nat | a = b × q + r ∧ r < b } :=  
  if less_than a b then (O, a)  
  else  
    let '(q', r) := div (a - b) b in  
      (S q', r).
```

# The Big Picture



DEMO

## 1 PROGRAM

- Hello World
- The theory: Subset Coercions

## 2 EQUATIONS

- Dependent pattern-matching
- Recursion
- Reasoning support

## 3 Type Classes

- Type Classes from HASKELL
- Type Classes in COQ

## 4 Conclusion

## Definition

The set  $\{x : T \mid P\}$  is the set of objects in  $T$  verifying property  $P$ .

- ▶ Useful for specifying, widely used in mathematics ;
- ▶ Links object and property.

## PVS

- ▶ Specialized typing algorithm for subset types, generating *Type-checking conditions*.

$$\frac{t : T \quad P[t/x]}{t : \{ x : T \mid P \}} \qquad \frac{t : \{ x : T \mid P \}}{t : T}$$

## PVS

- ▶ Specialized typing algorithm for subset types, generating *Type-checking conditions*.
- + Practical success ;

$$\frac{t : T \quad P[t/x]}{t : \{ x : T \mid P \}} \qquad \frac{t : \{ x : T \mid P \}}{t : T}$$

## PVS

- ▶ Specialized typing algorithm for subset types, generating *Type-checking conditions*.
- + Practical success ;
- Weaker safety guarantee in PVS (no De Bruijn principle).

$$\frac{t : T \quad P[t/x]}{t : \{ x : T \mid P \}} \qquad \frac{t : \{ x : T \mid P \}}{t : T}$$

## PVS

- ▶ Specialized typing algorithm for subset types, generating *Type-checking conditions*.
- + Practical success ;
- Weaker safety guarantee in PVS (no De Bruijn principle).

$$\frac{t : T \quad P[t/x]}{t : \{ x : T \mid P \}} \quad \frac{t : \{ x : T \mid P \}}{t : T}$$

Other languages based on refinement types/subset types/contracts:  
SAGE, F7, TYPED SCHEME ...

- 1 A property-irrelevant language (RUSSELL) with **decidable** typing

$$\frac{\Gamma \vdash t : \{ x : T \mid P \}}{\Gamma \vdash t : T}$$

$$\frac{\Gamma \vdash t : T \quad \Gamma, x : T \vdash P : \mathbf{Prop}}{\Gamma \vdash t : \{ x : T \mid P \}}$$

## ... to Subset Coercions

- 1 A property-irrelevant language (RUSSELL) with **decidable** typing
- 2 A **total** interpretation to COQ terms with **holes**

$$\frac{\Gamma \vdash t : \{ x : T \mid P \}}{\Gamma \vdash \pi_1 t : T}$$

$$\frac{\Gamma \vdash t : T \quad \Gamma, x : T \vdash P : \mathbf{Prop}}{\Gamma \vdash \mathbf{exist} \ t \ ?_{P[t/x]} : \{ x : T \mid P \}} \quad \Gamma \vdash ?_{P[t/x]} : P[t/x]$$

## ... to Subset Coercions

- 1 A property-irrelevant language (RUSSELL) with **decidable** typing
- 2 A **total** interpretation to COQ terms with **holes**
- 3 A mechanism to turn the holes into **proof obligations** and manage them.

$$\frac{\Gamma \vdash t : \{ x : T \mid P \}}{\Gamma \vdash \pi_1 t : T}$$

$$\frac{\Gamma \vdash t : T \quad \Gamma, x : T \vdash P : \mathbf{Prop} \quad \Gamma \vdash p : P[t/x]}{\Gamma \vdash \mathbf{exist} \ t \ p : \{ x : T \mid P \}}$$

Calculus of Constructions with

$$\frac{\Gamma \vdash t : U \quad \Gamma \vdash U \triangleright T}{\Gamma \vdash t : T}$$

$$\frac{T \equiv_{\beta} U}{\Gamma \vdash T \triangleright U}$$

Calculus of Constructions with

$$\frac{\Gamma \vdash t : U \quad \Gamma \vdash U \triangleright T}{\Gamma \vdash t : T} \quad \frac{T \equiv_{\beta} U}{\Gamma \vdash T \triangleright U}$$

$$\Gamma \vdash \{ x : U \mid P \} \triangleright U$$

$$\Gamma \vdash U \triangleright \{ x : U \mid P \}$$

Calculus of Constructions with

$$\frac{\Gamma \vdash t : U \quad \Gamma \vdash U \triangleright T}{\Gamma \vdash t : T} \quad \frac{T \equiv_{\beta} U}{\Gamma \vdash T \triangleright U}$$

$$\Gamma \vdash \{ x : U \mid P \} \triangleright U$$

$$\Gamma \vdash U \triangleright \{ x : U \mid P \}$$

Example

$$\frac{\Gamma \vdash 0 : \mathbb{N} \quad \Gamma \vdash \mathbb{N} \triangleright \{ x : \mathbb{N} \mid x \neq 0 \}}{\Gamma \vdash 0 : \{ x : \mathbb{N} \mid x \neq 0 \}}$$

Calculus of Constructions with

$$\frac{\Gamma \vdash t : U \quad \Gamma \vdash U \triangleright T}{\Gamma \vdash t : T} \qquad \frac{T \equiv_{\beta} U}{\Gamma \vdash T \triangleright U}$$

$$\Gamma \vdash \{ x : U \mid P \} \triangleright U$$

$$\Gamma \vdash U \triangleright \{ x : U \mid P \}$$

Example

$$\frac{\Gamma \vdash 0 : \mathbb{N} \quad \Gamma \vdash \mathbb{N} \triangleright \{ x : \mathbb{N} \mid x \neq 0 \}}{\Gamma \vdash 0 : \{ x : \mathbb{N} \mid x \neq 0 \}}$$

$$\Gamma \vdash ? : 0 \neq 0$$

## Theorem (Subject Reduction)

*If  $\Gamma \vdash t : T$  and  $t \rightarrow_{\beta} u$  then  $\Gamma \vdash u : T$*

## Proof.



Using the TPOSR technique due to Robin Adams.



## Theorem (Decidability of type checking and type inference)

*$\Gamma \vdash t : T$  is decidable.*

The target system : CIC with metavariables

$$\frac{\Gamma \vdash_? t : T \quad \Gamma \vdash_? p : P[t/x]}{\Gamma \vdash_? \text{exist}_{T,P} t p : \{ x : T \mid P \}}$$

$$\frac{\Gamma \vdash_? t : \{ x : T \mid P \}}{\Gamma \vdash_? \pi_1 t : T} \quad \frac{\Gamma \vdash_? t : \{ x : T \mid P \}}{\Gamma \vdash_? \pi_2 t : P[\pi_1 t/x]}$$

$$\frac{\Gamma \vdash_? P : \text{Prop}}{\Gamma \vdash_? ?_{\Gamma \vdash P} : P}$$

The target system : CIC with metavariables

$$\frac{\Gamma \vdash_? t : T \quad \Gamma \vdash_? p : P[t/x]}{\Gamma \vdash_? \text{exist}_{T,P} t p : \{ x : T \mid P \}}$$

$$\frac{\Gamma \vdash_? t : \{ x : T \mid P \}}{\Gamma \vdash_? \pi_1 t : T} \quad \frac{\Gamma \vdash_? t : \{ x : T \mid P \}}{\Gamma \vdash_? \pi_2 t : P[\pi_1 t/x]}$$

$$\frac{\Gamma \vdash_? P : \text{Prop}}{\Gamma \vdash_? ?_{\Gamma} P : P}$$

We can build a sound interpretation  $\llbracket - \rrbracket_{\Gamma}$  from RUSSELL to  $\text{CIC}_?$ :

Theorem (Soundness)

*If  $\Gamma \vdash t : T$  then  $\llbracket \Gamma \rrbracket \vdash_? \llbracket t \rrbracket_{\Gamma} : \llbracket T \rrbracket_{\Gamma}$  (assuming proof-irrelevance).*

# Deriving explicit coercions: $\Gamma \vdash_? c : T \triangleright U$

## Interpretation of coercions

If  $\Gamma \vdash T \triangleright U$  then there exists  $c$  such that  $\Gamma \vdash_? c : T \triangleright U$  and  
 $\Gamma, x : T \vdash_? c[x] : U$ .

# Deriving explicit coercions: $\Gamma \vdash_{\gamma} c : T \triangleright U$

## Interpretation of coercions

If  $\Gamma \vdash T \triangleright U$  then there exists  $c$  such that  $\Gamma \vdash_{\gamma} c : T \triangleright U$  and  $\Gamma, x : T \vdash_{\gamma} c[x] : U$ .

$$\frac{\Gamma \vdash_{\gamma} T \equiv_{\beta} U : s}{\Gamma \vdash_{\gamma} \quad : T \triangleright U}$$

# Deriving explicit coercions: $\Gamma \vdash_{\gamma} c : T \triangleright U$

## Interpretation of coercions

If  $\Gamma \vdash T \triangleright U$  then there exists  $c$  such that  $\Gamma \vdash_{\gamma} c : T \triangleright U$  and  $\Gamma, x : T \vdash_{\gamma} c[x] : U$ .

$$\frac{\Gamma \vdash_{\gamma} T \equiv_{\beta} U : s}{\Gamma \vdash_{\gamma} \bullet : T \triangleright U}$$

# Deriving explicit coercions: $\Gamma \vdash_? c : T \triangleright U$

## Interpretation of coercions

If  $\Gamma \vdash T \triangleright U$  then there exists  $c$  such that  $\Gamma \vdash_? c : T \triangleright U$  and  $\Gamma, x : T \vdash_? c[x] : U$ .

$$\frac{\Gamma \vdash_? T \equiv_{\beta} U : s}{\Gamma \vdash_? \bullet : T \triangleright U}$$
$$\Gamma \vdash_? \quad : \{ x : T \mid P \} \triangleright T$$

# Deriving explicit coercions: $\Gamma \vdash_? c : T \triangleright U$

## Interpretation of coercions

If  $\Gamma \vdash T \triangleright U$  then there exists  $c$  such that  $\Gamma \vdash_? c : T \triangleright U$  and  $\Gamma, x : T \vdash_? c[x] : U$ .

$$\frac{\Gamma \vdash_? T \equiv_{\beta} U : s}{\Gamma \vdash_? \bullet : T \triangleright U}$$
$$\Gamma \vdash_? \pi_1 \bullet : \{ x : T \mid P \} \triangleright T$$

# Deriving explicit coercions: $\Gamma \vdash_? c : T \triangleright U$

## Interpretation of coercions

If  $\Gamma \vdash T \triangleright U$  then there exists  $c$  such that  $\Gamma \vdash_? c : T \triangleright U$  and  $\Gamma, x : T \vdash_? c[x] : U$ .

$$\frac{\Gamma \vdash_? T \equiv_{\beta} U : s}{\Gamma \vdash_? \bullet : T \triangleright U}$$

$$\Gamma \vdash_? \pi_1 \bullet : \{ x : T \mid P \} \triangleright T$$

$$\Gamma \vdash_? \quad \quad \quad : T \triangleright \{ x : T \mid P \}$$

## Interpretation of coercions

If  $\Gamma \vdash T \triangleright U$  then there exists  $c$  such that  $\Gamma \vdash_? c : T \triangleright U$  and  $\Gamma, x : T \vdash_? c[x] : U$ .

$$\frac{\Gamma \vdash_? T \equiv_{\beta} U : s}{\Gamma \vdash_? \bullet : T \triangleright U}$$

$$\Gamma \vdash_? \pi_1 \bullet : \{ x : T \mid P \} \triangleright T$$

$$\Gamma \vdash_? \text{exist } \bullet \ ? \llbracket P \rrbracket_{\Gamma, x:T}[\bullet/x] : T \triangleright \{ x : T \mid P \}$$

# Deriving explicit coercions: $\Gamma \vdash_{?} c : T \triangleright U$

## Interpretation of coercions

If  $\Gamma \vdash T \triangleright U$  then there exists  $c$  such that  $\Gamma \vdash_{?} c : T \triangleright U$  and  
 $\Gamma, x : T \vdash_{?} c[x] : U$ .

$$\frac{\Gamma \vdash_{?} T \equiv_{\beta} U : s}{\Gamma \vdash_{?} \bullet : T \triangleright U}$$

$$\Gamma \vdash_{?} \pi_1 \bullet : \{ x : T \mid P \} \triangleright T$$

$$\Gamma \vdash_{?} \text{exist } \bullet \ ?_{\llbracket P \rrbracket_{\Gamma, x:T}[\bullet/x]} : T \triangleright \{ x : T \mid P \}$$

## Example

$$\frac{\Gamma \vdash_{?} 0 : \mathbb{N} \quad \Gamma \vdash_{?} \text{exist } \bullet \ ?_{\bullet \neq 0} : \mathbb{N} \triangleright \{ x : \mathbb{N} \mid x \neq 0 \}}{\Gamma \vdash_{?} \text{exist } 0 \ ?_{0 \neq 0} : \{ x : \mathbb{N} \mid x \neq 0 \}}$$

# Pattern-matching revisited

Put **logic** into the terms (think symbolic evaluation).

Let  $e : \mathbb{N}$ :

```
match e      return      T with
| S n ⇒      t1
| 0 ⇒      t2
end
```

# Pattern-matching revisited

Put **logic** into the terms (think symbolic evaluation).

Let  $e : \mathbb{N}$ :

```
match e as t   return t = e → T with
| S n ⇒       fun (H : S n = e) ⇒ t1
| 0 ⇒         fun (H : 0 = e) ⇒ t2
end           (refl_equal e)
```

# Pattern-matching revisited

Put **logic** into the terms (think symbolic evaluation).

## Further refinements

- ▶ Each branch typed only once ;

Let  $e : \mathbb{N}$ :

```
match e as t return t = e → T with
| S (S n) ⇒ fun (H : S (S n) = e) ⇒ t1
| n ⇒      fun (H : n = e) ⇒ t2
end       (refl_equal e)
```

# Pattern-matching revisited

Put **logic** into the terms (think symbolic evaluation).

## Further refinements

- ▶ Each branch typed only once ;

Let  $e : \mathbb{N}$ :

```
match e as t return t = e → T with
| S (S n) ⇒ fun (H : S (S n) = e) ⇒ t1
| S 0 ⇒ fun (H : S 0 = e) ⇒ t2
| 0 ⇒ fun (H : 0 = e) ⇒ t2
end (refl_equal e)
```

# Pattern-matching revisited

Put **logic** into the terms (think symbolic evaluation).

## Further refinements

- ▶ Each branch typed only once ;
- ▶ Add inequalities for intersecting patterns ;

Let  $e : \mathbb{N}$ :

```
match e as t return t = e → T with
| S (S n) ⇒ fun (H : S (S n) = e) ⇒ t1
| n ⇒ fun (H : n = e) ⇒ let H' : ∀n', n ≠ S (S n') in t2
end (refl_equal e)
```

# Pattern-matching revisited

Put **logic** into the terms (think symbolic evaluation).

## Further refinements

- ▶ Each branch typed only once ;
- ▶ Add inequalities for intersecting patterns ;
- ▶ Generalized to dependent inductive types.

Let  $e : \text{vector } n$ :

```
match e                                return                                T with
| vnil  $\Rightarrow$                          $t_1$ 
| vcons  $x n' v' \Rightarrow$                  $t_2$ 
end
```

# Pattern-matching revisited

Put **logic** into the terms (think symbolic evaluation).

## Further refinements

- ▶ Each branch typed only once ;
- ▶ Add inequalities for intersecting patterns ;
- ▶ Generalized to dependent inductive types.

Let  $e : \text{vector } n$ :

```
match e as t in vector n' return n' = n → JMeq t e → T with
| vnil ⇒ fun (H : 0 = n)(Hv : JMeq vnil e) ⇒ t1
| vcons x n' v' ⇒ fun (H : S n' = n)
    (Hv : JMeq (vcons x n' v') e) ⇒ t2
end (refl_equal n)(JMeq_refl e)
```

Extend the coercion system to handle inductive families:

$$\frac{\dots \quad \frac{\Gamma \vdash ?_{x=y} : x = y}{\Gamma \vdash ?_{x=y} : \mathbf{vector} \ x \triangleright \mathbf{vector} \ y}}{\Gamma \vdash ?_{x=y} : \mathbf{vector} \ v \ y}$$

**Any** object of an inductive family is coercible to another now.

Support for well-founded recursion and measures.

**Program Fixpoint**  $f (a : \mathbb{N}) \{wf < a\} : \mathbb{N} := b.$

Support for well-founded recursion and measures.

**Program Fixpoint**  $f (a : \mathbb{N}) \{wf < a\} : \mathbb{N} := b.$

$$\frac{\begin{array}{l} a : \mathbb{N} \\ f : \{x : \mathbb{N} \mid x < a\} \rightarrow \mathbb{N} \end{array}}{b : \mathbb{N}}$$

A **methodology** to build verified code in Coq, distributed since 2005:

- ▶ Realistic use-case: Finger Trees (ICFP'07).

A **methodology** to build verified code in Coq, distributed since 2005:

- ▶ Realistic use-case: Finger Trees (ICFP'07).
- ▶ Experimented by others, e.g. the Ynot project (Nanevsky, Morrisett & Birkedal), Swierstra (TPHOLs'09)...

A **methodology** to build verified code in Coq, distributed since 2005:

- ▶ Realistic use-case: Finger Trees (ICFP'07).
- ▶ Experimented by others, e.g. the Ynot project (Nanevsky, Morrisett & Birkedal), Swierstra (TPHOLs'09)...
- ▶ Independent of the theory, can be ported to other systems: Matita (by Enrico Tassi), Agda, Epigram.

A **methodology** to build verified code in Coq, distributed since 2005:

- ▶ Realistic use-case: Finger Trees (ICFP'07).
- ▶ Experimented by others, e.g. the Ynot project (Nanevsky, Morrisett & Birkedal), Swierstra (TPHOLs'09)...
- ▶ Independent of the theory, can be ported to other systems: Matita (by Enrico Tassi), Agda, Epigram.
- ▶ Still suffering from the proof-irrelevance abstraction leak.

- ▶ Limited support of dependent pattern-matching.
- ▶ Using coercions everywhere.
- ▶ No reasoning support for *a posteriori* proofs: “unstructured” code.
- ▶ Structural recursion on inductive families fragilized by the use of coercions.

- 1 PROGRAM
  - Hello World
  - The theory: Subset Coercions
- 2 EQUATIONS
  - Dependent pattern-matching
  - Recursion
  - Reasoning support
- 3 Type Classes
  - Type Classes from HASKELL
  - Type Classes in COQ
- 4 Conclusion

- ▶ EPIGRAM/AGDA-style pattern-matching definitions with `with` nodes.
- ▶ Purely logical handling of recursion.
- ▶ Propositional equations for definitional equalities and rewriting.
- ▶ Elimination principle and support for applying it.

Entirely elaborated to the vanilla kernel!

## DEMO

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. Essentially dependent destruction.

Heavily inspired from Goguen, McBride and McKinna (2006) and Norell (2007).

# Searching for a splitting tree

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  $rhs ::= \text{Program}(t) \mid \text{Refine}(c, t, spl)$

**Goal** Starting with  $f \Delta : \tau := \overrightarrow{\overrightarrow{p}} \dots$ , find a covering of the context map  $\Delta \vdash \overline{\Delta} : \Delta$ .

# 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}), ...)))]))])
```

- ▶ Syntactic guardness checks are fragile (and buggy)
- ▶ Do not work well with abstraction/modularity
- ▶ In Coq's case, restricted to structural recursion on a single argument, with no currying allowed

**Idea** Use the logic and well-founded recursion instead.

- ▶ Syntactic guardness checks are fragile (and buggy)
- ▶ Do not work well with abstraction/modularity
- ▶ In Coq's case, restricted to structural recursion on a single argument, with no currying allowed

**Idea** Use the logic and well-founded recursion instead.

In comparison with sized types:

- ✓ More general.
- ✓ Avoid extending the type system and the metatheory.
- ✗ Relies on the reduction of an hidden well-foundedness proof, necessary for SN.

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

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

- ▶ General definition of direct subterm:

$$I_{subfull} : \Pi \Delta_l \Delta_r, I \overline{\Delta_l} \rightarrow I \overline{\Delta_r} \rightarrow \mathbf{Prop}$$

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

- ▶ General definition of direct subterm:

$$I_{subfull} : \Pi \Delta_l \Delta_r, I \overline{\Delta_l} \rightarrow I \overline{\Delta_r} \rightarrow \mathbf{Prop}$$

- ▶ Wrap the inductive type in a sigma and define an homogeneous relation on the sigma type:  $I_{sub} : \mathbf{relation} (\Sigma \Delta, I \overline{\Delta})$

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

- ▶ General definition of direct subterm:  
 $I_{subfull} : \Pi \Delta_l \Delta_r, I \overline{\Delta_l} \rightarrow I \overline{\Delta_r} \rightarrow \mathbf{Prop}$
- ▶ Wrap the inductive type in a sigma and define an homogeneous relation on the sigma type:  $I_{sub} : \mathbf{relation} (\Sigma \Delta, I \overline{\Delta})$
- ▶ Extracts efficiently to a general fixpoint

# Subterm relation example: vectors

Derive Subterm for vector.

# Subterm relation example: vectors

Derive Subterm for vector.

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

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

vector\_strict\_subterm\_1\_1 :  $\forall (a : A) (n : \text{nat}) (H : \text{vector } A n),$   
vector\_strict\_subterm  $A n (\text{S } n) H (\text{Vcons } a H).$

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 ( $A : \text{Type}$ )  
:  $\forall H H0 : \text{nat}, \text{vector } A H \rightarrow \text{vector } A H0 \rightarrow \text{Prop} :=$   
vector\_strict\_subterm\_1\_1 :  $\forall (a : A) (n : \text{nat}) (H : \text{vector } A n),$   
vector\_strict\_subterm  $A n (\text{S } n) H (\text{Vcons } a H).$

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

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 PROGRAM

- Hello World
- The theory: Subset Coercions

## 2 EQUATIONS

- Dependent pattern-matching
- Recursion
- Reasoning support

## 3 Type Classes

- Type Classes from HASKELL
- Type Classes in COQ

## 4 Conclusion

- ▶ Equations hold definitionally in CCI + K (uses decidable instances if available).
- ▶ Equations for `with` nodes are just proxies to helper functions.
- ▶ All put together in a rewrite database, `f` can now be opacified.
- ▶ For well-founded definitions, we use a fixpoint unfolding lemma to prove the equations. This assumes proof-irrelevance and functional extensionally, but nothing if we have proof-irrelevance in the conversion (i.e. under contexts).

# 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.

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

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

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

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.

- ▶ 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.

- 1 PROGRAM
  - Hello World
  - The theory: Subset Coercions
- 2 EQUATIONS
  - Dependent pattern-matching
  - Recursion
  - Reasoning support
- 3 Type Classes
  - Type Classes from HASKELL
  - Type Classes in COQ
- 4 Conclusion

# Making *ad-hoc* polymorphism less *ad hoc*

Wadler & Blott, POPL'89. Also in ISABELLE since '91.

```
class Eq a where
  (==) :: a → a → Bool
instance Eq Bool where
  x == y = if x then y else not y
```

# Making *ad-hoc* polymorphism less *ad hoc*

Wadler & Blott, POPL'89. Also in ISABELLE since '91.

```
class Eq a where
  (==) :: a → a → Bool

instance Eq Bool where
  x == y = if x then y else not y

in :: Eq a ⇒ a → [a] → Bool
in x [] = False
in x (y : ys) = x == y || in x ys
```

# Parametrized instances and super-classes

```
instance (Eq a) => Eq [a] where
  [] == []           = True
  (x : xs) == (y : ys) = x == y && xs == ys
  _ == _            = False
```

# Parametrized instances and super-classes

```
instance (Eq a) => Eq [a] where
  [] == []           = True
  (x : xs) == (y : ys) = x == y && xs == ys
  _ == _            = False
```

```
class Num a where
  (+) :: a -> a -> a ...
```

```
class (Num a) => Fractional a where
  (/) :: a -> a -> a ...
```

## 1 PROGRAM

- Hello World
- The theory: Subset Coercions

## 2 EQUATIONS

- Dependent pattern-matching
- Recursion
- Reasoning support

## 3 Type Classes

- Type Classes from `HASKELL`
- Type Classes in `Coq`

## 4 Conclusion

- ▶ **Overloading** in programs, specifications and proofs.

- ▶ **Overloading** in programs, specifications and proofs.
- ▶ **A safer HASKELL** Proofs are part of instances.

```
Class Eq A := {  
  eqb : A → A → bool ;  
  eq_eqb : ∀ x y : A, x = y ↔ eqb x y = true }.
```

- ▶ **Overloading** in programs, specifications and proofs.
- ▶ **A safer HASKELL** Proofs are part of instances.

```
Class Eq A := {  
  eqb : A → A → bool ;  
  eq_eqb : ∀ x y : A, x = y ↔ eqb x y = true }.
```

- ▶ **Extensions** Dependent types give new power to type classes.

```
Class Reflexive A (R : relation A) :=  
  reflexive : ∀ x, R x x.
```

- ▶ Parametrized dependent records

**Class**  $\text{Id}$   $(\alpha_1 : \tau_1) \cdots (\alpha_n : \tau_n) :=$   
 $\{\mathbf{f}_1 : \phi_1 ; \cdots ; \mathbf{f}_m : \phi_m\}.$

- ▶ Parametrized dependent records

**Record** **ld**  $(\alpha_1 : \tau_1) \cdots (\alpha_n : \tau_n) :=$   
 $\{\mathbf{f}_1 : \phi_1 ; \cdots ; \mathbf{f}_m : \phi_m\}.$

- ▶ Parametrized dependent records

**Record** **ld**  $(\alpha_1 : \tau_1) \cdots (\alpha_n : \tau_n) :=$   
 $\{\mathbf{f}_1 : \phi_1 ; \cdots ; \mathbf{f}_m : \phi_m\}.$

Instances are just definitions of type **ld**  $\vec{t}_n$ .

# A cheap implementation

- ▶ Parametrized dependent records

**Record** **ld**  $(\alpha_1 : \tau_1) \cdots (\alpha_n : \tau_n) :=$   
 $\{\mathbf{f}_1 : \phi_1 ; \cdots ; \mathbf{f}_m : \phi_m\}.$

Instances are just definitions of type **ld**  $\vec{t}_n$ .

- ▶ Custom implicit arguments of projections

$\mathbf{f}_1 : \forall \vec{\alpha}_n : \vec{\tau}_n , \mathbf{ld} \vec{\alpha}_n \rightarrow \phi_1$

# A cheap implementation

- ▶ Parametrized dependent records

**Record** **ld**  $(\alpha_1 : \tau_1) \cdots (\alpha_n : \tau_n) :=$   
 $\{\mathbf{f}_1 : \phi_1 ; \cdots ; \mathbf{f}_m : \phi_m\}.$

Instances are just definitions of type **ld**  $\vec{t}_n$ .

- ▶ Custom implicit arguments of projections

$\mathbf{f}_1 : \forall \{\overrightarrow{\alpha_n : \tau_n}\}, \{\mathbf{ld} \overrightarrow{\alpha_n}\} \rightarrow \phi_1$

$(\lambda x y : \text{bool}. \text{eqb } x y)$

# Elaboration with classes, an example

$(\lambda x y : \text{bool}. \text{eqb } x y)$

$\rightsquigarrow \{ \text{Implicit arguments} \}$

$(\lambda x y : \text{bool}. @\text{eqb } (?_A : \text{Type}) (?_{eq} : \text{Eq } ?_A) x y)$

# Elaboration with classes, an example

$(\lambda x y : \text{bool}. \text{eqb } x y)$

$\rightsquigarrow \{ \text{Implicit arguments} \}$

$(\lambda x y : \text{bool}. @\text{eqb } (?_A : \text{Type}) (?_{eq} : \text{Eq } ?_A) x y)$

$\rightsquigarrow \{ \text{Unification} \}$

$(\lambda x y : \text{bool}. @\text{eqb } \text{bool } (?_{eq} : \text{Eq } \text{bool}) x y)$

# Elaboration with classes, an example

$(\lambda x y : \text{bool}. \text{eqb } x y)$

$\rightsquigarrow \{ \text{Implicit arguments} \}$

$(\lambda x y : \text{bool}. @\text{eqb } (?_A : \text{Type}) (?_{eq} : \text{Eq } ?_A) x y)$

$\rightsquigarrow \{ \text{Unification} \}$

$(\lambda x y : \text{bool}. @\text{eqb } \text{bool } (?_{eq} : \text{Eq } \text{bool}) x y)$

$\rightsquigarrow \{ \text{Proof search for Eq bool returns Eq\_bool} \}$

$(\lambda x y : \text{bool}. @\text{eqb } \text{bool } \text{Eq\_bool } x y)$

Proof-search tactic with instances as lemmas:

$A : \text{Type}, \text{eqa} : \text{Eq } A \vdash ? : \text{Eq } (\text{list } A)$

- ▶ SLD resolution, higher-order unification.
- Returns the first solution only
- + Extensible through  $\mathcal{L}_{\text{tac}}$

# Numeric overloading

**Class** Num  $\alpha := \{ \text{zero} : \alpha ; \text{one} : \alpha ; \text{plus} : \alpha \rightarrow \alpha \rightarrow \alpha \}$ .

# Numeric overloading

**Class** Num  $\alpha$  := { zero :  $\alpha$  ; one :  $\alpha$  ; plus :  $\alpha \rightarrow \alpha \rightarrow \alpha$  }.

**Instance** nat\_num : Num nat :=  
{ zero := 0%nat ; one := 1%nat ; plus := Peano.plus }.

**Instance** Z\_num : Num Z :=  
{ zero := 0%Z ; one := 1%Z ; plus := Zplus }.

# Numeric overloading

**Class** Num  $\alpha$  := { zero :  $\alpha$  ; one :  $\alpha$  ; plus :  $\alpha \rightarrow \alpha \rightarrow \alpha$  }.

**Instance** nat\_num : Num nat :=  
{ zero := 0%nat ; one := 1%nat ; plus := Peano.plus }.

**Instance** Z\_num : Num Z :=  
{ zero := 0%Z ; one := 1%Z ; plus := Zplus }.

*Notation* "0" := zero.

*Notation* "1" := one.

*Infix* "+" := plus.

# Numeric overloading

**Class** Num  $\alpha$  := { zero :  $\alpha$  ; one :  $\alpha$  ; plus :  $\alpha \rightarrow \alpha \rightarrow \alpha$  }.

**Instance** nat\_num : Num nat :=  
{ zero := 0%nat ; one := 1%nat ; plus := Peano.plus }.

**Instance** Z\_num : Num Z :=  
{ zero := 0%Z ; one := 1%Z ; plus := Zplus }.

*Notation* "0" := zero.

*Notation* "1" := one.

*Infix* "+" := plus.

**Check** ( $\lambda x : \text{nat}, x + (1 + 0 + x)$ ).

**Check** ( $\lambda x : \text{Z}, x + (1 + 0 + x)$ ).

# Numeric overloading

**Class** Num  $\alpha$  := { zero :  $\alpha$  ; one :  $\alpha$  ; plus :  $\alpha \rightarrow \alpha \rightarrow \alpha$  }.

**Instance** nat\_num : Num nat :=  
{ zero := 0%nat ; one := 1%nat ; plus := Peano.plus }.

**Instance** Z\_num : Num Z :=  
{ zero := 0%Z ; one := 1%Z ; plus := Zplus }.

*Notation* "0" := zero.

*Notation* "1" := one.

*Infix* "+" := plus.

**Check** ( $\lambda x : \text{nat}, x + (1 + 0 + x)$ ).

**Check** ( $\lambda x : \text{Z}, x + (1 + 0 + x)$ ).

Defaulting

**Check** ( $\lambda x, x + 1$ ).

# Dependent classes

**Class** Reflexive  $\{A\}$  ( $R$  : relation  $A$ ) :=  
 refl :  $\forall x, R\ x\ x$ .

# Dependent classes

```
Class Reflexive {A} (R : relation A) :=  
  refl :  $\forall x, R x x$ .
```

```
Instance eq_refl A : Reflexive (@eq A) := @refl_equal A.
```

```
Instance iff_refl : Reflexive iff.
```

```
Proof. red. tauto. Qed.
```

# Dependent classes

```
Class Reflexive {A} (R : relation A) :=  
  refl :  $\forall x, R x x$ .
```

```
Instance eq_refl A : Reflexive (@eq A) := @refl_equal A.
```

```
Instance iff_refl : Reflexive iff.
```

```
Proof. red. tauto. Qed.
```

```
Goal  $\forall P, P \leftrightarrow P$ .
```

```
Proof. apply refl. Qed.
```

```
Goal  $\forall A (x : A), x = x$ .
```

```
Proof. intros A ; apply refl. Qed.
```

# Dependent classes

```
Class Reflexive {A} (R : relation A) :=  
  refl :  $\forall x, R x x$ .
```

```
Instance eq_refl A : Reflexive (@eq A) := @refl_equal A.
```

```
Instance iff_refl : Reflexive iff.
```

```
Proof. red. tauto. Qed.
```

```
Goal  $\forall P, P \leftrightarrow P$ .
```

```
Proof. apply refl. Qed.
```

```
Goal  $\forall A (x : A), x = x$ .
```

```
Proof. intros A ; apply refl. Qed.
```

```
Ltac reflexivity' := apply refl.
```

```
Lemma foo '{Reflexive nat R} : R 0 0.
```

```
Proof. intros. reflexivity'. Qed.
```

# A structuring tool: super-classes and substructures

Building hierarchies of classes:

```
Class Fractional '{Num  $\alpha$ } :=  
  { div :  $\alpha \rightarrow \{ y : \alpha \mid y \neq 0 \} \rightarrow \alpha$  }.
```

```
Class Equivalence  $\alpha$  :=  
  { equiv_refl :> Reflexive  $\alpha$  ;  
    equiv_sym  :> Symmetric  $\alpha$  ;  
    equiv_trans :> Transitive  $\alpha$  }
```

+ Special support for binding super-classes

Tried and tested by P. Letouzey, S. Lescuyer on FSets (JFLA'10), B. Spitters and E. van der Weegen (ITP'10)...

- ✓ A **lightweight** and **general** implementation of type classes, available in Coq v8.2.
- ✓ A type-theoretic **explanation** and **extension** of type classes concepts (TPHOLs'08, with N. Oury).

- ✓ A **lightweight** and **general** implementation of type classes, available in Coq v8.2.
- ✓ A type-theoretic **explanation** and **extension** of type classes concepts (TPHOLs'08, with N. Oury).

On top of that:

- ▶ **Original** test-case: a new setoid-rewriting tactic (Coq-1, JFR'09). N. Benton and N. Tabareau (TLDI'10), T. Braibant and D. Pous (Coq-1, ITP'10).
- ▶ Automatic inference of instances (Matthias Puech).

- ✓ A **lightweight** and **general** implementation of type classes, available in Coq v8.2.
- ✓ A type-theoretic **explanation** and **extension** of type classes concepts (TPHOLs'08, with N. Oury).

On top of that:

- ▶ **Original** test-case: a new setoid-rewriting tactic (Coq-1, JFR'09). N. Benton and N. Tabareau (TLDI'10), T. Braibant and D. Pous (Coq-1, ITP'10).
- ▶ Automatic inference of instances (Matthias Puech).

Relation to Coercive Subtyping and **Canonical Structures** by Amokrane Saïbi (POPL'97), studied by Asperti *et al* (TPHOLs'09).

## Efficiency:

- ▶ No forward reasoning, risk of non-termination.
- ▶ Little sharing.

## Control:

- ▶ Scoping of instances.
- ▶ Research strategies.

**Hope** These are all researched in the logic programming community.

- 1 PROGRAM
  - Hello World
  - The theory: Subset Coercions
- 2 EQUATIONS
  - Dependent pattern-matching
  - Recursion
  - Reasoning support
- 3 Type Classes
  - Type Classes from HASKELL
  - Type Classes in COQ
- 4 Conclusion

Three **elaborations**:

- ▶ We studied a more **flexible** programming language, RUSSELL, providing a new **formal justification** of “*Predicate subtyping*” à la PVS, in a system with proof terms.

Three **elaborations**:

- ▶ We studied a more **flexible** programming language, RUSSELL, providing a new **formal justification** of “*Predicate subtyping*” à la PVS, in a system with proof terms.
- ▶ We implemented a **definitional** compiler for dependent pattern-matching and complex recursion schemes.

Three **elaborations**:

- ▶ We studied a more **flexible** programming language, RUSSELL, providing a new **formal justification** of “*Predicate subtyping*” à la PVS, in a system with proof terms.
- ▶ We implemented a **definitional** compiler for dependent pattern-matching and complex recursion schemes.
- ▶ We designed a **simple** yet powerful type class system as an additional **layer** on top of dependent type theory.

## PROGRAM, EQUATIONS & COQ

- ▶ We hinted at the important **foundational issues** with  $\eta$ -rules, K and proof-irrelevance that need to be solved (B. Werner, H. Herbelin).
- ▶ **Automation** for discharging proof obligations (S. Wilson).

## PROGRAM, EQUATIONS & COQ

- ▶ We hinted at the important **foundational issues** with  $\eta$ -rules, K and proof-irrelevance that need to be solved (B. Werner, H. Herbelin).
- ▶ **Automation** for discharging proof obligations (S. Wilson).

## Type Classes

- ▶ **Control** on instance declarations and better automated proof-search and rewriting.
- ▶ Unification with **Structure**, interaction with implicit coercions.

## Success of the elaboration point-of-view!

- ✓ Progress in accessibility and scalability of the tool.
- ▶ Still lots to do, e.g.: tactic language, declarative proof tools for proof search and rewriting.
- ✗ Practical shortcomings. **Youth!** Efficiency and controllability concerns.
- ✗ Foundational shortcomings: eta rules and K.

# Elaborations in Type Theory

MATTHIEU SOZEAU

Harvard University

DTP'10

July 10th 2010

Edinburgh, UK

