

# An Environment for Programming with Dependent Types (in Coq)

MATTHIEU SOZEAU  
under the direction of CHRISTINE PAULIN-MOHRING

LRI, Univ. Paris-Sud - DÉMONS Team & INRIA Saclay - PROVAL Project

ICIS Seminar  
January 20th 2009  
Nijmegen, Netherlands



- ▶ A higher-order, polymorphic logic:

$$\forall A (P Q : A \rightarrow \mathbf{Prop}), (\forall x, P x \leftrightarrow Q x) \rightarrow \forall x, P x \rightarrow Q x$$

$$\forall x y : \mathbb{N}, x + y = y + x$$

- ▶ A higher-order, polymorphic logic:

$$\forall A (P Q : A \rightarrow \mathbf{Prop}), (\forall x, P x \leftrightarrow Q x) \rightarrow \forall x, P x \rightarrow Q x$$

$$\forall x y : \mathbb{N}, x + y = y + x$$

- ▶ A purely functional programming language with dependent types:

$$\text{div} : \forall (a : \text{nat}) (b : \text{nat} \mid b \neq 0), \\ \{ (q, r) : \text{nat} \times \text{nat} \mid a = b \times q + r \wedge r < b \}$$

- ▶ A higher-order, polymorphic logic:

$$\forall A (P Q : A \rightarrow \mathbf{Prop}), (\forall x, P x \leftrightarrow Q x) \rightarrow \forall x, P x \rightarrow Q x$$

$$\forall x y : \mathbb{N}, x + y = y + x$$

- ▶ A purely functional programming language with dependent types:

$$\text{div} : \forall (a : \text{nat}) (b : \text{nat} \mid b \neq 0), \\ \{ (q, r) : \text{nat} \times \text{nat} \mid a = b \times q + r \wedge r < b \}$$

**Problem:** writing such programs is difficult.

# 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

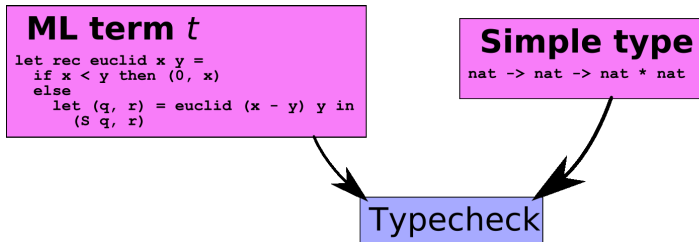
```
λ b (H : b > 0) a,  
gt_wf_rec a (λ n, { (q, r) : nat × nat | b > r ∧ n = q × b + r })  
  (λ n  
    (H0 : Π m : nat, n > m → { (q, r) : nat × nat | b > r ∧ m = q × b + r })),  
  sumbool_rec  
    (λ _ : {b ≤ n} + {b > n},  
      { (q, r) : nat × nat | b > r ∧ n = q × b + r })  
    (λ lebn : b ≤ n,  
      let (x, x0) := H0 (n - b) (lt_minus n b lebn H) in  
      (let (q, r) as p  
        return  
          ((let (q, r) := p in b > r ∧ n - b = q × b + r) →  
            { (q, r) : nat × nat | b > r ∧ n = q × b + r }) := x in  
      λ y : b > r ∧ n - b = q × b + r,  
      match y with  
      | conj g e ⇒  
        elt  
          (eq_ind (b + (q × b + r)) (λ n0, b > r ∧ n = n0)  
            (eq_ind (n - b) (λ n0, b > r ∧ n = b + n0)  
              (conj g (le_plus_minus b n lebn))  
                (q × b + r) e) (b + q × b + r) (plus_assoc b (q × b) r))  
        end) x0) (λ gtbn : b > n, elt (conj gtbn refl))  
    (le_gt_dec b n))
```

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

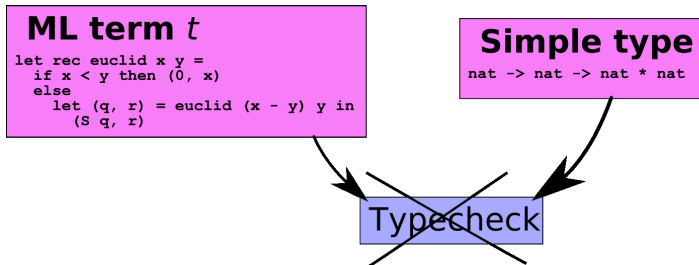
## ML term $t$

```
let rec euclid x y =  
  if x < y then (0, x)  
  else  
    let (q, r) = euclid (x - y) y in  
    (S q, r)
```

# The Big Picture



# The Big Picture



# The Big Picture

## ML term $t$

```
let rec euclid x y =  
  if x < y then (0, x)  
  else  
    let (q, r) = euclid (x - y) y in  
    (S q, r)
```

## Dependent type $T$

```
nat -> { y : nat | y > 0 } ->  
nat * nat
```

```
graph TD; A[ML term t] --> C[Typecheck]; B[Dependent type T] --> C;
```

Typecheck

# The Big Picture

## ML term $t$

```
let rec euclid x y =  
  if x < y then (0, x)  
  else  
    let (q, r) = euclid (x - y) y in  
    (S q, r)
```

## Dependent type $T$

```
nat -> { y : nat | y > 0 } ->  
nat * nat
```

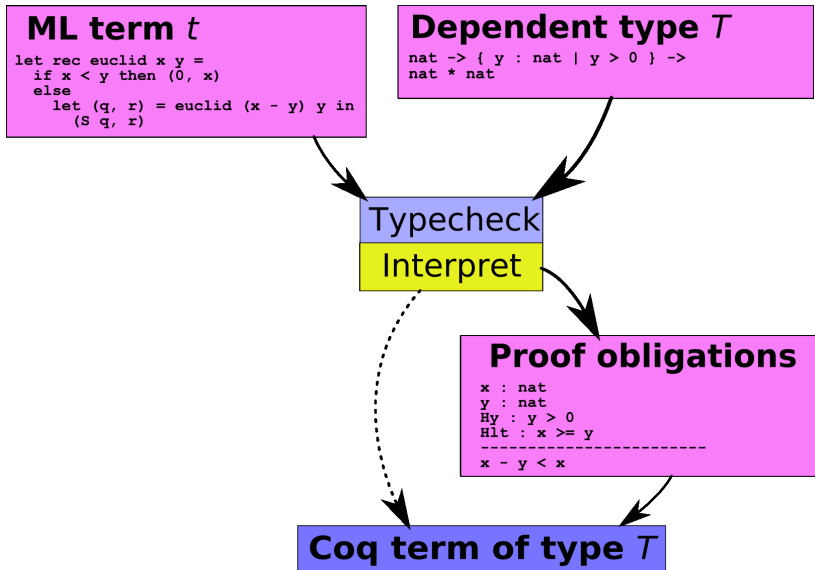
Typecheck

Interpret

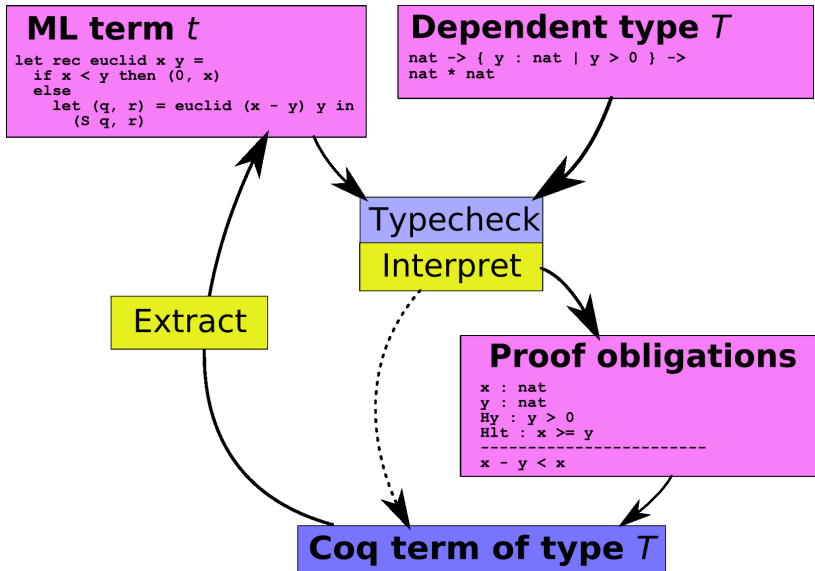
## Proof obligations

```
x : nat  
y : nat  
Hy : y > 0  
Hlt : x >= y  
-----  
x - y < x
```

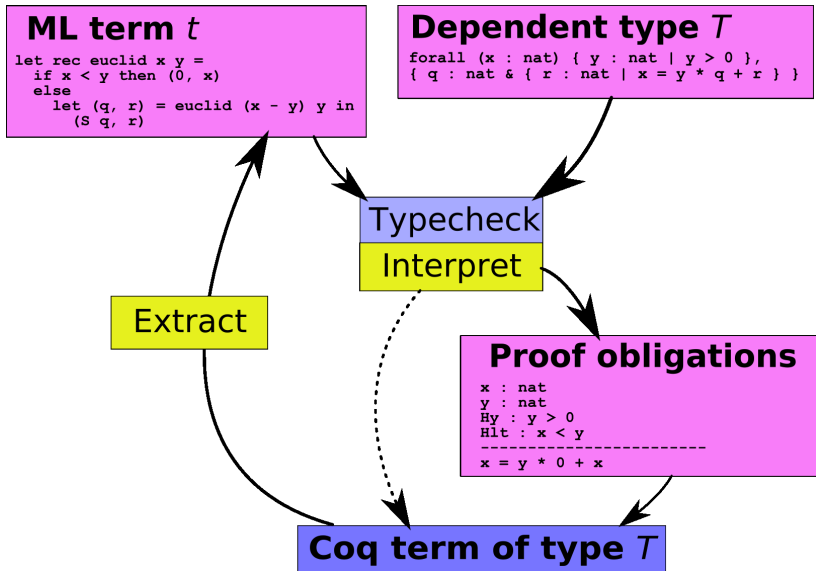
# The Big Picture



# The Big Picture



# The Big Picture



- 1 RUSSELL
  - Subset Coercions: A Simple Idea
  - Metatheory
  - Interpretation in COQ
  
- 2 PROGRAM
  - Hello World
  - Finger Trees in a Hurry
  
- 3 Type Classes
  - Type Classes from HASKELL
  - Type Classes in COQ
  
- 4 Conclusion
  - Extensions

## 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*.

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

## PVS

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

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

## PVS

- ▶ Specialized typing algorithm for subset types, generating *Type-checking conditions*.
- + Practical success ;
- No strong safety guarantee in PVS.

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

## PVS

- ▶ Specialized typing algorithm for subset types, generating *Type-checking conditions*.
- + Practical success ;
- No strong safety guarantee in PVS.

$$\frac{t : T \quad p : P[t/x]}{\text{elt } t \ p : \{ x : T \mid P \}} \qquad \frac{t : \{ x : T \mid P \}}{\sigma_1 t : T}$$

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

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

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

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

$$\frac{\Gamma \vdash t : \{ x : T \mid P \}}{\Gamma \vdash \sigma_1 t : T}$$

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

## ... 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.

$$\frac{\Gamma \vdash t : \{ x : T \mid P \}}{\Gamma \vdash \sigma_1 t : T}$$

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

Calculus of Constructions with

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

Calculus of Constructions with

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

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

Calculus of Constructions with

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

$$\frac{\Gamma \vdash U \triangleright V : \mathbf{Set} \quad \Gamma, x : U \vdash P : \mathbf{Prop}}{\Gamma \vdash \{ x : U \mid P \} \triangleright V : \mathbf{Set}}$$

$$\frac{\Gamma \vdash U \triangleright V : \mathbf{Set} \quad \Gamma, x : V \vdash P : \mathbf{Prop}}{\Gamma \vdash U \triangleright \{ x : V \mid P \} : \mathbf{Set}}$$

Calculus of Constructions with

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

$$\frac{\Gamma \vdash U \triangleright V : \mathbf{Set} \quad \Gamma, x : U \vdash P : \mathbf{Prop}}{\Gamma \vdash \{ x : U \mid P \} \triangleright V : \mathbf{Set}}$$

$$\frac{\Gamma \vdash U \triangleright V : \mathbf{Set} \quad \Gamma, x : V \vdash P : \mathbf{Prop}}{\Gamma \vdash U \triangleright \{ x : V \mid P \} : \mathbf{Set}}$$

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

Calculus of Constructions with

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

$$\frac{\Gamma \vdash U \triangleright V : \mathbf{Set} \quad \Gamma, x : U \vdash P : \mathbf{Prop}}{\Gamma \vdash \{ x : U \mid P \} \triangleright V : \mathbf{Set}}$$

$$\frac{\Gamma \vdash U \triangleright V : \mathbf{Set} \quad \Gamma, x : V \vdash P : \mathbf{Prop}}{\Gamma \vdash U \triangleright \{ x : V \mid P \} : \mathbf{Set}}$$

**Example**  $\frac{\Gamma \vdash 0 : \mathbb{N} \quad \Gamma \vdash \mathbb{N} \triangleright \{ x : \mathbb{N} \mid x \neq 0 \} : \mathbf{Set}}{\Gamma \vdash 0 : \{ x : \mathbb{N} \mid x \neq 0 \}}$   
 $\Gamma \vdash ? : 0 \neq 0$

Calculus of Constructions with

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

$$\frac{\Gamma \vdash U \triangleright V : \mathbf{Set} \quad \Gamma, x : U \vdash P : \mathbf{Prop}}{\Gamma \vdash \{ x : U \mid P \} \triangleright V : \mathbf{Set}}$$

$$\frac{\Gamma \vdash U \triangleright V : \mathbf{Set} \quad \Gamma, x : V \vdash P : \mathbf{Prop}}{\Gamma \vdash U \triangleright \{ x : V \mid P \} : \mathbf{Set}}$$

$$\frac{\Gamma \vdash U \triangleright T : s_1 \quad \Gamma, x : U \vdash V \triangleright W : s_2}{\Gamma \vdash \Pi x : T.V \triangleright \Pi x : U.W : s_2}$$

## Calculus of Constructions with

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

$$\frac{\Gamma \vdash U \triangleright V : \mathbf{Set} \quad \Gamma, x : U \vdash P : \mathbf{Prop}}{\Gamma \vdash \{ x : U \mid P \} \triangleright V : \mathbf{Set}}$$

$$\frac{\Gamma \vdash U \triangleright V : \mathbf{Set} \quad \Gamma, x : V \vdash P : \mathbf{Prop}}{\Gamma \vdash U \triangleright \{ x : V \mid P \} : \mathbf{Set}}$$

$$\frac{\Gamma \vdash U \triangleright T : s_1 \quad \Gamma, x : U \vdash V \triangleright W : s_2}{\Gamma \vdash \Pi x : T.V \triangleright \Pi x : U.W : s_2}$$

$\triangleright$  is symmetric!

## Theorem (Subject Reduction)

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



Using the TPOSR technique due to Robin Adams.

## Theorem (Subject Reduction)

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



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

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

$$\frac{\Gamma \vdash_{?} t : \{ x : T \mid P \}}{\Gamma \vdash_{?} \sigma_1 t : T} \quad \frac{\Gamma \vdash_{?} t : \{ x : T \mid P \}}{\Gamma \vdash_{?} \sigma_2 t : P[\sigma_1 t/x]}$$

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

We build an interpretation  $\llbracket - \rrbracket_{\Gamma}$  from RUSSELL to  $\text{CIC}_{?}$  terms.

The target system : CIC with metavariables

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

$$\frac{\Gamma \vdash_{?} t : \{ x : T \mid P \}}{\Gamma \vdash_{?} \sigma_1 t : T} \quad \frac{\Gamma \vdash_{?} t : \{ x : T \mid P \}}{\Gamma \vdash_{?} \sigma_2 t : P[\sigma_1 t/x]}$$

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

We build an interpretation  $\llbracket - \rrbracket_{\Gamma}$  from RUSSELL to  $\text{CIC}_{?}$  terms.

Our goal: proving soundness

$$\text{If } \Gamma \vdash t : T \text{ then } \llbracket \Gamma \rrbracket \vdash_{?} \llbracket t \rrbracket_{\Gamma} : \llbracket T \rrbracket_{\Gamma}.$$

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

## Interpretation of coercions

If  $\Gamma \vdash T \triangleright U : s$  then there exists  $c$  so that  $\Gamma \vdash_{\text{?}} c : T \triangleright U$ .

## Interpretation of coercions

If  $\Gamma \vdash T \triangleright U : s$  then there exists  $c$  so that  $\Gamma \vdash_{\gamma} c : T \triangleright U$ .

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

## Interpretation of coercions

If  $\Gamma \vdash T \triangleright U : s$  then there exists  $c$  so that  $\Gamma \vdash_{\gamma} c : T \triangleright U$ .

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

## Interpretation of coercions

If  $\Gamma \vdash T \triangleright U : s$  then there exists  $c$  so that  $\Gamma \vdash_{\gamma} c : T \triangleright U$ .

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

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

## Interpretation of coercions

If  $\Gamma \vdash T \triangleright U : s$  then there exists  $c$  so that  $\Gamma \vdash_{\gamma} c : T \triangleright U$ .

$$\frac{\Gamma \vdash_{\gamma} T \equiv_{\beta} U : s}{\Gamma \vdash_{\gamma} \bullet : T \triangleright U}$$
$$\Gamma \vdash_{\gamma} \sigma_1 \bullet : \{ x : T \mid P \} \triangleright T$$

## Interpretation of coercions

If  $\Gamma \vdash T \triangleright U : s$  then there exists  $c$  so that  $\Gamma \vdash_{\tau} c : T \triangleright U$ .

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

$$\Gamma \vdash_{\tau} \sigma_1 \bullet : \{ x : T \mid P \} \triangleright T$$

$$\Gamma \vdash_{\tau} \phantom{\sigma_1 \bullet} : T \triangleright \{ x : T \mid P \}$$

## Interpretation of coercions

If  $\Gamma \vdash T \triangleright U : s$  then there exists  $c$  so that  $\Gamma \vdash_{?} c : T \triangleright U$ .

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

$$\Gamma \vdash_{?} \sigma_1 \bullet : \{ x : T \mid P \} \triangleright T$$

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

## Interpretation of coercions

If  $\Gamma \vdash T \triangleright U : s$  then there exists  $c$  so that  $\Gamma \vdash_{\gamma} c : T \triangleright U$ .

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

$$\Gamma \vdash_{\gamma} \sigma_1 \bullet : \{ x : T \mid P \} \triangleright T$$

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

## Example

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

## Example (Application)

$$\frac{\Gamma \vdash f : T \quad \Gamma \vdash T \triangleright \Pi x : V.W : s \quad \Gamma \vdash u : U \quad \Gamma \vdash U \triangleright V : s'}{\Gamma \vdash (f u) : W[u/x]}$$

$$\llbracket f u \rrbracket_\Gamma = \text{let } \pi = \text{coerce}_\Gamma T (\Pi x : V.W) \text{ in} \\ \text{let } c = \text{coerce}_\Gamma U V \text{ in} \\ (\pi(\llbracket f \rrbracket_\Gamma)) (c(\llbracket u \rrbracket_\Gamma))$$

## Theorem (Soundness)

If  $\Gamma \vdash t : T$  then  $\llbracket \Gamma \rrbracket \vdash? \llbracket t \rrbracket_\Gamma : \llbracket T \rrbracket_\Gamma$ .

$\vdash_?$ 's equational theory:

$$\beta\text{-rules} \left\{ \begin{array}{l} (\beta) \quad (\lambda x : X.e) v \quad \equiv \quad e[v/x] \\ (\sigma_i) \quad \sigma_i (\text{elt}_{E,P} e_1 e_2) \quad \equiv \quad e_i \end{array} \right\} \text{CoQ}$$

$\vdash_?$ 's equational theory:

$$\beta\text{-rules} \left\{ \begin{array}{l} (\beta) \quad (\lambda x : X.e) v \quad \equiv \quad e[v/x] \\ (\sigma_i) \quad \sigma_i (\text{elt}_{E,P} e_1 e_2) \quad \equiv \quad e_i \end{array} \right\} \text{CoQ}$$

$$\eta\text{-rules} \left\{ \begin{array}{l} (\eta) \quad (\lambda x : X.e x) \quad \equiv \quad e \quad \text{if } x \notin \mathcal{FV}(e) \\ (\text{SP}) \quad \text{elt}_{E,P} (\sigma_1 e) (\sigma_2 e) \quad \equiv \quad e \end{array} \right.$$

$\vdash_?$ 's equational theory:

$$\beta\text{-rules} \left\{ \begin{array}{l} (\beta) \quad (\lambda x : X.e) v \quad \equiv \quad e[v/x] \\ (\sigma_i) \quad \sigma_i (\mathbf{elt}_{E,P} e_1 e_2) \quad \equiv \quad e_i \end{array} \right\} \text{CoQ}$$

$$\eta\text{-rules} \left\{ \begin{array}{l} (\eta) \quad (\lambda x : X.e x) \quad \equiv \quad e \quad \text{if } x \notin \mathcal{FV}(e) \\ (\text{SP}) \quad \mathbf{elt}_{E,P} (\sigma_1 e) (\sigma_2 e) \quad \equiv \quad e \end{array} \right.$$

$$\text{Proof Irrelevance} \quad \mathbf{elt}_{E,P} t p \quad \equiv \quad \mathbf{elt}_{E,P} t' p' \quad \text{if } t \equiv t'$$

$\vdash_?$ 's equational theory:

$$\beta\text{-rules} \left\{ \begin{array}{l} (\beta) \quad (\lambda x : X.e) v \quad \equiv \quad e[v/x] \\ (\sigma_i) \quad \sigma_i (\text{elt}_{E,P} e_1 e_2) \quad \equiv \quad e_i \end{array} \right\} \text{CoQ}$$

$$\eta\text{-rules} \left\{ \begin{array}{l} (\eta) \quad (\lambda x : X.e x) \quad \equiv \quad e \quad \text{if } x \notin \mathcal{FV}(e) \\ (\text{SP}) \quad \text{elt}_{E,P} (\sigma_1 e) (\sigma_2 e) \quad \equiv \quad e \end{array} \right.$$

$$\text{Proof Irrelevance} \quad \text{elt}_{E,P} t p \quad \equiv \quad \text{elt}_{E,P} t' p' \quad \text{if } t \equiv t'$$

... have practical effects

Difficulty to reason on code:  $t \equiv u \not\rightarrow \llbracket t \rrbracket_\Gamma \equiv_{\text{CoQ}} \llbracket u \rrbracket_\Gamma$ .

- 1 RUSSELL
  - Subset Coercions: A Simple Idea
  - Metatheory
  - Interpretation in COQ
- 2 PROGRAM
  - Hello World
  - Finger Trees in a Hurry
- 3 Type Classes
  - Type Classes from HASKELL
  - Type Classes in COQ
- 4 Conclusion
  - Extensions

# The PROGRAM vernacular

Replaces the Program tactic by Catherine Parent.

# The PROGRAM vernacular

Replaces the Program tactic by Catherine Parent.

## Architecture

Wrap around COQ's vernacular commands (Definition, Fixpoint, Lemma, ...).

# The PROGRAM vernacular

Replaces the Program tactic by Catherine Parent.

## Architecture

Wrap around COQ's vernacular commands (Definition, Fixpoint, Lemma, ...).

- 1 Use the COQ parser ;

Program Definition  $f : T := t$  .

# The PROGRAM vernacular

Replaces the Program tactic by Catherine Parent.

## Architecture

Wrap around COQ's vernacular commands (Definition, Fixpoint, Lemma, ...).

- 1 Use the COQ parser ;
- 2 Typecheck  $\Gamma \vdash t : T$  and generate  $\llbracket \Gamma \rrbracket \vdash? \llbracket t \rrbracket_{\Gamma} : \llbracket T \rrbracket_{\Gamma}$  ;

Program Definition  $f : \llbracket T \rrbracket_{\Gamma} := \llbracket t \rrbracket_{\Gamma}$  .

# The PROGRAM vernacular

Replaces the Program tactic by Catherine Parent.

## Architecture

Wrap around COQ's vernacular commands (Definition, Fixpoint, Lemma, ...).

- 1 Use the COQ parser ;
- 2 Typecheck  $\Gamma \vdash t : T$  and generate  $\llbracket \Gamma \rrbracket \vdash? \llbracket t \rrbracket_{\Gamma} : \llbracket T \rrbracket_{\Gamma}$  ;
- 3 Interactive proof of the obligations ;

Program Definition  $f : \llbracket T \rrbracket_{\Gamma} := \llbracket t \rrbracket_{\Gamma} + \text{obligations}$ .

# The PROGRAM vernacular

Replaces the Program tactic by Catherine Parent.

## Architecture

Wrap around COQ's vernacular commands (Definition, Fixpoint, Lemma, ...).

- 1 Use the COQ parser ;
- 2 Typecheck  $\Gamma \vdash t : T$  and generate  $\llbracket \Gamma \rrbracket \vdash? \llbracket t \rrbracket_{\Gamma} : \llbracket T \rrbracket_{\Gamma}$  ;
- 3 Interactive proof of the obligations ;
- 4 Final definition.

Definition  $f : \llbracket T \rrbracket_{\Gamma} := \llbracket t \rrbracket_{\Gamma} + \text{obligations}$ .

DEMO

# Dependent Finger Trees

A complex datastructure, with a reference implementation in `HASKELL` (Hinze & Paterson, JFP 2006).

- ▶ Using `PROGRAM` and type classes, we construct a faithful port which is proved **safe** (terminating and invariant preserving).

# Dependent Finger Trees

A complex datastructure, with a reference implementation in `HASKELL` (Hinze & Paterson, JFP 2006).

- ▶ Using `PROGRAM` and type classes, we construct a faithful port which is proved **safe** (terminating and invariant preserving).
- ▶ Through **dependent types**, it provides guarantees to be used by higher-level structures.

# Dependent Finger Trees

A complex datastructure, with a reference implementation in `HASKELL` (Hinze & Paterson, JFP 2006).

- ▶ Using `PROGRAM` and type classes, we construct a faithful port which is proved **safe** (terminating and invariant preserving).
- ▶ Through **dependent types**, it provides guarantees to be used by higher-level structures.
- ▶ **Instantiated** to (safe) ropes whose extraction has reasonable performance.

# Dependent Finger Trees

A complex datastructure, with a reference implementation in `HASKELL` (Hinze & Paterson, JFP 2006).

- ▶ Using `PROGRAM` and type classes, we construct a faithful port which is proved **safe** (terminating and invariant preserving).
- ▶ Through **dependent types**, it provides guarantees to be used by higher-level structures.
- ▶ **Instantiated** to (safe) ropes whose extraction has reasonable performance.

Required extensions of the tool to treat **inductive families** and **pattern-matching**.

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

- ▶ Already experimented by others, e.g. the Ynot project (Nanevsky, Morrisett & Birkedal).

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

- ▶ Already experimented by others, e.g. the Ynot project (Nanevsky, Morrisett & Birkedal).
- ▶ Independent of the theory, can be ported to other systems: Matita (by Enrico Tassi), Agda, Epigram.

- 1 RUSSELL
  - Subset Coercions: A Simple Idea
  - Metatheory
  - Interpretation in COQ
- 2 PROGRAM
  - Hello World
  - Finger Trees in a Hurry
- 3 Type Classes
  - Type Classes from HASKELL
  - Type Classes in COQ
- 4 Conclusion
  - Extensions

- ▶ **Intersection types**: closed overloading by declaring multiple signatures for a single constant (e.g.  $\mathbb{C}$ DUCE).
- ▶ **Bounded quantification** and **class-based** overloading. Overloading circumscribed by a subtyping relation (e.g. structural subtyping à la OCAML).

- ▶ **Intersection types**: closed overloading by declaring multiple signatures for a single constant (e.g. `CDUCE`).
- ▶ **Bounded quantification** and **class-based** overloading. Overloading circumscribed by a subtyping relation (e.g. structural subtyping à la `OCAML`).

Context:

- ▶ **Modularity**: separate definitions of the specializations.
- ▶ **Constrained by Coq**: a fixed kernel language!

# Solutions for overloading

- ▶ **Intersection types**: closed overloading by declaring multiple signatures for a single constant (e.g. `CDUCE`).
- ▶ **Bounded quantification** and **class-based** overloading. Overloading circumscribed by a subtyping relation (e.g. structural subtyping à la `OCAML`).

Context:

- ▶ **Modularity**: separate definitions of the specializations.
- ▶ **Constrained by COQ**: a fixed kernel language!

Solution:

**Elaborate** Type Classes, a kind of bounded quantification where the subtyping relation needs not be internalized.

Wadler & Blott, POPL'89.

```
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.

```
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
```

# Parameterized instances and superclasses

`instance (Eq a) => Eq [a] where`

`[] == [] = True`

`(x : xs) == (y : ys) = x == y && xs == ys`

`_ == _ = False`

# Parameterized instances and superclasses

```
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 ...
```

# Parameterized instances and superclasses

```
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 ...
```

## The MLer point of view

A system of modules and functors with sugar for implicit instantiation and functorization.

- 1** RUSSELL
  - Subset Coercions: A Simple Idea
  - Metatheory
  - Interpretation in COQ
- 2** PROGRAM
  - Hello World
  - Finger Trees in a Hurry
- 3** Type Classes
  - Type Classes from HASKELL
  - Type Classes in COQ
- 4** Conclusion
  - Extensions

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

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

**Class Eq A :=**

**eqb** :  $A \rightarrow A \rightarrow \text{bool}$  ;

**eq\_eqb** :  $\forall x y : A, x = y \leftrightarrow \text{eqb } x \ y = \text{true}$ .

- ▶ **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.
```

- ▶ Parameterized dependent records

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

- ▶ Parameterized dependent records

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

- ▶ Parameterized 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

- ▶ Parameterized 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**  $\overrightarrow{t_n}$ .

- ▶ Custom implicit arguments of projections

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

# A cheap implementation

- ▶ Parameterized 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:

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

- ▶ Bounded depth-first search
- Returns the first solution only
- + Extensible through  $\mathcal{L}_{\text{tac}}$

```
Class Num  $\alpha$  := zero :  $\alpha$  ; one :  $\alpha$  ; 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** ( $\lambda x : \text{Z}, x + (1 + 0 + x)$ ).

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

**Class** (Num  $\alpha$ )  $\Rightarrow$  Fractional  $\alpha :=$   
div :  $\alpha \rightarrow \{ y : \alpha \mid y \neq 0 \} \rightarrow \alpha.$

**Class** (Num  $\alpha$ )  $\Rightarrow$  Fractional  $\alpha$  :=  
div :  $\alpha \rightarrow \{ y : \alpha \mid y \neq 0 \} \rightarrow \alpha$ .

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

+ Special support for binding superclasses

Already tried and tested by Pierre Letouzey and Ralph Matthes on some contributions.

# Dependent classes

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

# Dependent classes

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

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

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

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

# Dependent classes

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

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

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

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

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

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

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

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

# Dependent classes

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

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

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

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

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

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

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

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

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

```
Lemma foo [ Reflexive nat R ] : R 0 0.
```

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

- ✓ A **lightweight** and **general** implementation of type classes, “available” in Coq v8.2.
- ✓ A type-theoretic **explanation** and **extension** of type classes concepts.

- ✓ A **lightweight** and **general** implementation of type classes, “available” in Coq v8.2.
- ✓ A type-theoretic **explanation** and **extension** of type classes concepts.

On top of that:

- ▶ **Realistic** test-case: a new setoid-rewriting tactic built on top of classes.
- ▶ A system to automatically infer instances by 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.

On top of that:

- ▶ **Realistic** test-case: a new setoid-rewriting tactic built on top of classes.
- ▶ A system to automatically infer instances by Matthias Puech.

Related to Coercive Subtyping and **Canonical Structures** by Amokrane Saïbi (POPL'97).

- 1 RUSSELL
  - Subset Coercions: A Simple Idea
  - Metatheory
  - Interpretation in COQ
- 2 PROGRAM
  - Hello World
  - Finger Trees in a Hurry
- 3 Type Classes
  - Type Classes from HASKELL
  - Type Classes in COQ
- 4 Conclusion
  - Extensions

Two **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.

Two **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 designed a **simple** yet powerful type class system as an additional **layer** on top of dependent type theory.

Five implementations!

- ▶ A **formal** proof of SR for RUSSELL ( $\sim$  20k lines of COQ)

## Five implementations!

- ▶ A **formal** proof of SR for RUSSELL ( $\sim$  20k lines of COQ)
- ▶ The PROGRAM tool to ease **programming** in COQ using the **full language** ( $\sim$  10k lines of OCAML)

## Five implementations!

- ▶ A **formal** proof of SR for RUSSELL ( $\sim 20$ k lines of COQ)
- ▶ The PROGRAM tool to ease **programming** in COQ using the **full language** ( $\sim 10$ k lines of OCAML)
- ▶ A **verified** implementation of Finger Trees and Ropes ( $\sim 4$ k lines of literate COQ).

## Five implementations!

- ▶ A **formal** proof of SR for RUSSELL ( $\sim 20$ k lines of COQ)
- ▶ The PROGRAM tool to ease **programming** in COQ using the **full language** ( $\sim 10$ k lines of OCAML)
- ▶ A **verified** implementation of Finger Trees and Ropes ( $\sim 4$ k lines of literate COQ).
- ▶ A **lightweight** implementation of type classes to support **overloading** in specifications and during proofs ( $\sim 2$ k lines of OCAML).

## Five implementations!

- ▶ A **formal** proof of SR for RUSSELL ( $\sim 20k$  lines of COQ)
- ▶ The PROGRAM tool to ease **programming** in COQ using the **full language** ( $\sim 10k$  lines of OCAML)
- ▶ A **verified** implementation of Finger Trees and Ropes ( $\sim 4k$  lines of literate COQ).
- ▶ A **lightweight** implementation of type classes to support **overloading** in specifications and during proofs ( $\sim 2k$  lines of OCAML).
- ▶ A new, **extensible** implementation of **generalized rewriting** based on classes ( $\sim 1k$  lines of OCAML,  $1k$  lines of COQ).

## RUSSELL, PROGRAM & COQ

- ▶ We hinted at the important **foundational issues** with  $\eta$ -rules and proof-irrelevance that need to be solved (with Benjamin Werner).
- ▶ **Automation** for discharging proof-obligations (Sean Wilson at Edinburgh).
- ▶ Handling **truly dependent pattern-matching** by an elaboration.

## RUSSELL, PROGRAM & COQ

- ▶ We hinted at the important **foundational issues** with  $\eta$ -rules and proof-irrelevance that need to be solved (with Benjamin Werner).
- ▶ **Automation** for discharging proof-obligations (Sean Wilson at Edinburgh).
- ▶ Handling **truly dependent pattern-matching** by an elaboration.

## Type Classes

- ▶ **Control** on instance declarations and better automated proof-search (with Matthias Puech), including automatic rewriting.
- ▶ Unification with **Structure**, interaction with implicit coercions.

## An Environment for Programming with Dependent Types (in Coq)

MATTHIEU SOZEAU  
under the direction of CHRISTINE PAULIN-MOHRING

LRI, Univ. Paris-Sud - DÉMONS Team & INRIA Saclay - PROVAL Project

ICIS Seminar  
January 20th 2009  
Nijmegen, Netherlands

Try



8.2!



# Pattern-matching revisited

Put **logic** into the terms.

Let  $e : \mathbb{N}$ :

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

# Pattern-matching revisited

Put **logic** into the terms.

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.

## 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.

## 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.

## 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.

## 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.

## 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)
```

## Obligations

Unresolved implicits (`_`) are turned into obligations, à la **refine**.

## Bang

`!` = (`False_rect _ _`) where `False_rect :  $\forall A : \text{Type}, \text{False} \rightarrow A$` . It corresponds to ML's `assert(false)`.

```
match x with 0 => ! | n => ... end
```

## Obligations

Unresolved implicits `(_)` are turned into obligations, à la **refine**.

## Bang

`!` = `(False_rect _ _)` where `False_rect :  $\forall A : \text{Type}, \text{False} \rightarrow A$` . It corresponds to ML's `assert(false)`.

```
match x with 0  $\Rightarrow$  ! | n  $\Rightarrow$  ... end
```

## Destruction

Let `let 'p := t in e = match t with p  $\Rightarrow$  e end`. `p` can be an arbitrary pattern.

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

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