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.
In the design spaces...

- 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.
Our angle

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.
Our focus

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
Lemma eucl_dev : ∀ n, n > 0 → ∀ m : nat, 

\{ (q, r) : \text{nat} \times \text{nat} \mid n > r \land 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].
∃ (S q, r); simpl; auto with arith.
elim plus_assoc.
elim e; auto with arith.
intros gtnb.
∃ (0, n); simpl; auto with arith.

Qed.
Programming directly

Check \(((\lambda (b : \text{nat}) (H : b > 0) (a : \text{nat}),\nWf\_nat.gt\_wf\_rec a (\lambda n : \text{nat}, \text{diveucl} n b)\n(\lambda (n : \text{nat}) (H0 : \prod m : \text{nat}, n > m \rightarrow \text{diveucl} m b),\nsymbol\_bool\_rec (\lambda \_ : \{b \leq n\} + \{b > n\}, \text{diveucl} n b)\n(\lambda lebn : b \leq n,\ndiveucl\_rec (n - b) b (\lambda \_ : \text{diveucl} (n - b) b, \text{diveucl} n b)\n(\lambda (q r : \text{nat}) (g : b > r) (e : n - b = q \times b + r),\ndivex n b (S q) r g
(eq\_ind (b + (q \times b + r))) (\lambda n0 : \text{nat}, n = n0)\n(eq\_ind (n - b) (\lambda n0 : \text{nat}, n = b + n0)\n(Minus.le\_plus\_minus b n lebn) (q \times b + r) e)\n(b + q \times b + r) (Plus.plus\_assoc b (q \times b) r))))\n(H0 (n - b) (Minus.lt\_minus n b lebn H)))(\lambda gtbn : b > n, \text{divex} n b 0 n gtbn (\text{refl\_equal} n))\n(\text{Compare\_dec.le\_gt\_dec b n)))) : \prod n : \text{nat}, n > 0 \rightarrow \prod m : \text{nat}, \text{diveucl} m n).
We can write programs as usual and still give them rich types.

```plaintext
Program Fixpoint \texttt{div} (a : \texttt{nat}) (b : \texttt{nat} \mid b \neq 0) \{ \texttt{wf \_lt} \ a \} : 
\{ (q, r) : \texttt{nat} \times \texttt{nat} \mid a = b \times q + r \land r < b \} :=
\text{if} \ \texttt{less\_than} \ a \ b \ \text{then} \ (O, a)
\text{else}
\text{let} ' (q', r) := \texttt{div} (a - b) b \ \text{in}
\ (S \ q', r).
```
**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** $T$

```
forall (x : nat) { y : nat | y > 0 },
{ q : nat & { r : nat | x = y * q + r } }
```

---

**Extract**

**Typecheck**

**Interpret**

**Coq term of type** $T$

---

**Proof obligations**

$x : \text{nat}$

$y : \text{nat}$

$Hy : y > 0$

$Hlt : x < y$

-----------------------------------

$x = y * 0 + x$
Hello World: Euclidean division

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
A simple idea

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.

\[
\frac{t : T}{t : \{ x : T \mid P \}} \quad \frac{t : \{ x : T \mid P \}}{t : T}
\]
From “Predicate subtyping” . . .

PVS

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

+ Practical success ;

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

\[
\frac{t : \{ x : T \mid P \}}{t : T}
\]
From “Predicate subtyping” . . .

PVS

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

+ Practical success ;
- Weaker safety guarantee in PVS (no De Bruijn principle).

\[
\frac{t : T \quad P[t/x]}{t : \{ x : T \mid P \}} \quad \frac{t : \{ x : T \mid P \}}{t : T}
\]
From “Predicate subtyping”…

**PVS**

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

  + Practical success;
  
  - Weaker safety guarantee in PVS (no De Bruijn principle).

\[
\begin{align*}
\frac{t : T \quad P[t/x]}{t : \{ x : T \mid P \}} & \quad \frac{t : \{ x : T \mid P \}}{t : T}
\end{align*}
\]

Other languages based on refinement types/subset types/contracts: *Sage, F7, Typed Scheme* …
A property-irrelevant language (Russell) with decidable typing

\[ \Gamma \vdash t : \{ x : T \mid P \} \]

\[ \Gamma \vdash t : T \]

\[ \Gamma \vdash t : T \quad \Gamma, x : T \vdash P : \text{Prop} \]

\[ \Gamma \vdash t : \{ x : T \mid P \} \]
A property-irrelevant language (Russell) with decidable typing

A total interpretation to Coq terms with holes

\[ \Gamma \vdash t : \{ x : T \mid P \} \]
\[ \Gamma \vdash \pi_1 t : T \]

\[ \Gamma \vdash t : T \]
\[ \Gamma, x : T \vdash P : \text{Prop} \]
\[ \Gamma \vdash \text{exist } t \ ?_{P[t/x]} : \{ x : T \mid P \} \]
\[ \Gamma \vdash ?_{P[t/x]} : P[t/x] \]
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.

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

\[
\begin{align*}
\Gamma \vdash t : T & \quad \Gamma, x : T \vdash P : \text{Prop} & \quad \Gamma \vdash p : P[t/x] \\
\hline
\Gamma \vdash \text{exist } t \, p : \{x : T \mid P\}
\end{align*}
\]
Calculus of Constructions with

\[
\begin{align*}
\Gamma &\vdash t : U \\
\Gamma &\vdash U \triangleright T \\
\hline
\Gamma &\vdash t : T
\end{align*}
\]

\[
\begin{align*}
T &\equiv_{\beta} U \\
\hline
\Gamma &\vdash T \triangleright U
\end{align*}
\]
Russell typing \( \vdash \) and coercion \( \triangleright \)

Calculus of Constructions with

\[
\begin{align*}
\Gamma \vdash t : U & \quad \Gamma \vdash U \triangleright T \\
\hline
\Gamma \vdash t : T \quad T \equiv_{\beta} U \\
\hline
\Gamma \vdash T \triangleright U \\
\Gamma \vdash \{ x : U \mid P \} \triangleright U \\
\Gamma \vdash U \triangleright \{ x : U \mid P \}
\end{align*}
\]
Calculus of Constructions with

\[
\frac{\Gamma \vdash t : U \quad \Gamma \vdash U \to T}{\Gamma \vdash t : T} \quad \frac{T \equiv_{\beta} U}{\Gamma \vdash T \to U}
\]

\[
\Gamma \vdash \{ x : U \mid P \} \to U
\]

\[
\Gamma \vdash U \to \{ x : U \mid P \}
\]

Example

\[
\frac{\Gamma \vdash 0 : \mathbb{N} \quad \Gamma \vdash \mathbb{N} \to \{ x : \mathbb{N} \mid x \neq 0 \}}{\Gamma \vdash 0 : \{ x : \mathbb{N} \mid x \neq 0 \}}
\]
Russell typing ⊢ and coercion ⊢

Calculus of Constructions with

\[
\frac{\Gamma \vdash t : U \quad \Gamma \vdash U \supset T}{\Gamma \vdash t : T} \quad \frac{T \equiv_\beta U}{\Gamma \vdash T \supset U} \\
\Gamma \vdash \{ x : U \mid P \} \supset U \\
\Gamma \vdash U \supset \{ x : U \mid P \}
\]

Example

\[
\frac{\Gamma \vdash 0 : \mathbb{N} \quad \Gamma \vdash \mathbb{N} \supset \{ x : \mathbb{N} \mid x \neq 0 \}}{\Gamma \vdash 0 : \{ x : \mathbb{N} \mid x \neq 0 \}} \\
\frac{\Gamma \vdash 0 : \{ x : \mathbb{N} \mid x \neq 0 \}}{\Gamma \vdash ? : 0 \neq 0}
\]
Metatheory

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

\[
\begin{align*}
\Gamma \vdash ? \ t : T & \quad \Gamma \vdash ? \ p : P[t/x] \\
\Gamma \vdash ? \ \text{exist}_{T,P} \ t \ p : \{ x : T \mid P \} \\
\Gamma \vdash ? \ t : \{ x : T \mid P \} & \quad \Gamma \vdash ? \ t : \{ x : T \mid P \} \\
\Gamma \vdash ? \ \pi_1 \ t : T & \quad \Gamma \vdash ? \ \pi_2 \ t : P[\pi_1 \ t/x] \\
\Gamma \vdash ? \ P : \text{Prop} & \\
\Gamma \vdash ? \ ? \Gamma \vdash ? \ P : P
\end{align*}
\]
The target system: CIC with metavariables

\[
\begin{align*}
\Gamma \vdash ? \; t : T & \quad \Gamma \vdash ? \; p : P[t/x] \\
\Gamma \vdash ? \text{exist}_{T,P} \; t \; p : \{ \, x : T \mid P \, \} \\
\Gamma \vdash ? \; t : \{ \, x : T \mid P \, \} & \quad \Gamma \vdash ? \; t : \{ \, x : T \mid P \, \} \\
\Gamma \vdash ? \; \pi_1 \; t : T & \quad \Gamma \vdash ? \; \pi_2 \; t : P[\pi_1 \; t/x] \\
\Gamma \vdash ? \; P : \text{Prop} & \\
\Gamma \vdash ? ? \Gamma \vdash P : P
\end{align*}
\]

We can build a sound interpretation \([\_\_]_{\Gamma}\) from \textsc{Russell} to \textsc{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 $\Gamma, x : T \vdash ? \ c[x] : U$. 

Example

$\Gamma \vdash ? \ 0 : N$

$\Gamma \vdash ? \exist \ c \neq 0 : N \triangleright \{x : N \mid x \neq 0\}$

$\Gamma \vdash ? \exist 0 ? 0 \neq 0 : \{x : N \mid x \neq 0\}$
Interpretation of coercions

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

\[
\Gamma \vdash ? \ T \equiv_{\beta} U : s \\
\hline
\Gamma \vdash ? \ : T \triangleright U
\]

Example

\( \Gamma \vdash ? \ 0 : N \)

\( \Gamma \vdash ? \ \exists x \quad x \neq 0 : N \triangleright \{ x : N \mid x \neq 0 \} \)

\( \Gamma \vdash ? \ \exists 0 ? \ 0 \quad x \neq 0 \triangleright \{ x : N \mid x \neq 0 \} \)
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 $\Gamma, x : T \vdash ? \ c[x] : U$.

\[
\frac{\Gamma \vdash ? \ T \equiv_{\beta} U : s}{\Gamma \vdash ? \ \bullet : T \triangleright U}
\]
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 $\Gamma, x : T \vdash \, ? \, c[x] : U$.

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

$$\Gamma \vdash \, ? \, : \{ x : T \mid P \} \triangleright T$$
Interpretation of coercions

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

\[
\frac{\Gamma \vdash T \equiv_{\beta} U : s}{\Gamma \vdash \bullet : T \triangleright U}
\]

$\Gamma \vdash \pi_1 \bullet : \{ x : T \mid P \} \triangleright T$
Interpretation of coercions

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

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

$\Gamma \vdash ? \bullet : T \triangleright U$

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

$\Gamma \vdash ? : T \triangleright \{ x : T \mid P \}$
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.$

\[
\frac{\Gamma \vdash ? \ T \equiv_{\beta} U : s}{\Gamma \vdash ? \ \bullet : T \triangleright U}
\]

\[
\Gamma \vdash ? \ \pi_1 \ \bullet : \{ x : T \mid P \} \triangleright T
\]

\[
\Gamma \vdash ? \ \text{exist} \ \bullet \ ?^{[P]}_{\Gamma, x : T[\bullet/x]} : T \triangleright \{ x : T \mid P \}
\]
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

$\Gamma, x : T \vdash c[x] : U$.

\[
\begin{align*}
\Gamma \vdash T \equiv_\beta U : s \\
\Gamma \vdash \bullet : T \triangleright U
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash \pi_1 \bullet : \{ x : T \mid P \} \triangleright T
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash \text{exist } \bullet ?_{[P]_{\Gamma, x : T[\bullet/x]}} : T \triangleright \{ x : T \mid P \}
\end{align*}
\]

**Example**

\[
\begin{align*}
\Gamma \vdash 0 : \mathbb{N} & \quad \Gamma \vdash \text{exist } \bullet ?_{\neq 0} : \mathbb{N} \triangleright \{ x : \mathbb{N} \mid x \neq 0 \} \\
\Gamma \vdash \text{exist } 0 ?_{0\neq 0} : \{ x : \mathbb{N} \mid x \neq 0 \}
\end{align*}
\]
Put logic into the terms (think symbolic evaluation).

Let $e : \mathbb{N}$:

\[
\text{match } e \quad \text{return } T \text{ with }
\]
\[
| \text{S } n \Rightarrow t_1 \\
| 0 \Rightarrow t_2 \\
\text{end}
\]
Put logic into the terms (think symbolic evaluation).

Let $e : \mathbb{N}$:

\[
\text{match } e \text{ as } t \quad \text{return } t = e \rightarrow T \text{ with }
\]
\[
| S \ n \Rightarrow \quad \text{fun } (H : S \ n = e) \Rightarrow t_1
\]
\[
| 0 \Rightarrow \quad \text{fun } (H : 0 = e) \Rightarrow t_2
\]

\[
\text{end} \quad (\text{refl\_equal } e)
\]
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
| S (S n) ⇒ fun (H : S (S n) = e) ⇒ t_1
| n ⇒ fun (H : n = e) ⇒ t_2
end (refl_equal e)
```
Put logic into the terms (think symbolic evaluation).

Further refinements

- Each branch typed only once;

Let $e : \mathbb{N}$:

```plaintext
match e as t
| S (S n) ⇒ fun (H : S (S n) = e) ⇒ t_1
| S 0 ⇒ fun (H : S 0 = e) ⇒ t_2
| 0 ⇒ fun (H : 0 = e) ⇒ t_2
end
return t = e → T with
(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 $e : \mathbb{N}$:

\[
\text{match } e \text{ as } t \quad \text{return } t = e \to T \text{ with } \\
\mid S (S \ n) \Rightarrow \quad \text{fun } (H : S (S \ n) = e) \Rightarrow t_1 \\
\mid n \Rightarrow \quad \text{fun } (H : n = e) \Rightarrow \text{let } H' : \forall n', n \neq S (S \ n') \text{ in } t_2 \\
\text{end} \quad (\text{refl\_equal } e)
\]
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 $e : \text{vector } n$:

```plaintext
match e
| vnil ⇒ return $t_1$
| vcons $x\ n\ v\ v'$ ⇒ $t_2$
end
```

$T$ with $t_1$ $t_2$
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 \rightarrow \text{JMeq } t \ e \rightarrow T \ with

| \text{vnil} \Rightarrow \text{fun} \ (H : 0 = n)(Hv : \text{JMeq } \text{vnil } e) \Rightarrow t_1
| \text{vcons } x \ n' \ v' \Rightarrow \text{fun} \ (H : S \ n' = n)
  \ (Hv : \text{JMeq } (\text{vcons } x \ n' \ v') \ e) \Rightarrow t_2
end \ (\text{refl_equal } n)(\text{JMeq_refl } e)
```
Extension to inductive families

Extend the coercion system to handle inductive families:

\[ \Gamma \vdash ?\, \text{eq} \, \text{rec} \, \mathbb{N} \, x \, \text{vector} \, \bullet \, y \, \text{?}\, \text{eq} \, \text{rec} \, \mathbb{N} \, x \, \text{vector} \, v \, y \, \text{?}\, \text{eq} \, \text{rec} \, \mathbb{N} \, x \, \text{vector} \, y \]

Any object of an inductive family is coercible to another now.
Support for well-founded recursion and measures.

Program Fixpoint $f (a : \mathbb{N}) \{\text{wf } < a\} : \mathbb{N} := b$. 
Support for well-founded recursion and measures.

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

\[
\begin{align*}
  a & : \mathbb{N} \\
  f & : \{ x : \mathbb{N} \mid x < a \} \rightarrow \mathbb{N} \\
  b & : \mathbb{N}
\end{align*}
\]
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.**
Problems with PROGRAM

- 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
Overview of Equations

- **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 \texttt{Coq} terms with holes
3. Proofs of the obligations using a mix of \texttt{ML} and \texttt{Ltac} code. Essentially dependent destruction.

Searching for a splitting tree

pattern $p ::= x \mid C \overrightarrow{p} \mid ?(t)$

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 $f \Delta : \tau ::= \overrightarrow{p} \ldots$, find a covering of the context map $\Delta \vdash \overline{\Delta} : \Delta$. 
Proof search example

Overlapping clauses with first-match semantics.

Equations equal \( (n \ m : \text{nat}) : \{ n = m \} + \{ n \neq m \} \) :=
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 _ .

\[
\begin{align*}
\text{Split}(n \ m : \text{nat} \vdash n \ m : n \ m : \text{nat}, \ n, [ \\
\text{Split}(m : \text{nat} \vdash O \ m : n \ m : \text{nat}, \ m, [ \\
\quad \text{Compute}(\vdash O O : n \ m : \text{nat}, \ \text{Program}(\text{left eq_refl})), \\
\quad \text{Compute}(m : \text{nat} \vdash O (S m) : n \ m : \text{nat}, \ \text{Program}(\text{right _ })))], \\
\text{Split}(n \ m : \text{nat} \vdash (S n) \ m : n \ m : \text{nat}, \ m, [ \\
\quad \text{Compute}(n : \text{nat} \vdash (S n) O : n \ m : \text{nat}, \ldots), \\
\quad \text{Compute}(n \ m : \text{nat} \vdash (S n) (S m) : n \ m : \text{nat}, \\
\quad \quad \text{Refine(equal n m,} \\
\quad \quad \quad \text{idsubst}(n \ m : \text{nat}, x : \{ n = m \} + \{ n \neq m \}, \ldots))))])
\end{align*}
\]
Recursion

- 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.
Recursion

- 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 $I : \prod \Delta, s$. 

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

- General definition of direct subterm:
  \[
  \text{I}_{\text{subfull}} : \prod \Delta_l \Delta_r, I \Delta_l \rightarrow I \Delta_r \rightarrow \text{Prop}
  \]
Use **well-founded** recursion on the subterm relation for inductive families $I : \prod \Delta, s$.

- General definition of direct subterm:
  $$I_{subfull} : \prod \Delta_l \Delta_r, I \Delta_l \to I \Delta_r \to \text{Prop}$$

- Wrap the inductive type in a sigma and define an homogeneous relation on the sigma type: $I_{sub} : \text{relation} (\Sigma \Delta, I \Delta)$
Use **well-founded** recursion on the subterm relation for inductive families $I : \Pi \Delta, s$.

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

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

- Extracts efficiently to a general fixpoint
Derive Subterm for vector.
Subterm relation example: vectors

Derive Subterm for vector.

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

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

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

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

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

Derive Subterm for vector.

Inductive vector.strict_subterm (A : Type)

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

\( \text{vector.strict_subterm.1.1} \) : \( \forall (a : A) (n : \text{nat}) (H : \text{vector} \ A \ n), \) vector.strict_subterm \( A \ n \ (S \ n) \ H \ (\text{Vcons} \ a \ H) \).

Check vector.subterm : \( \forall A : \text{Type}, \ \text{relation} \ \{ \text{index} : \text{nat} \& \ \text{vector} \ A \ \text{index} \} \).

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) Vnil := (Vnil, Vnil) ;

unzip A B ?(S n) (Vcons (pair x y) n v) with unzip v := {

\ | (pair xs ys) := (Vcons x xs, Vcons y ys) \}.
1 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 \rightarrow 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
\begin{align*}
\text{filter} & \{ A \} \ (l : \text{list} \ A) \ (p : A \to \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 \}
\end{align*}

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

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

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

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

(\forall A a l p, \text{filter\_ind} A l p (\text{filter\_l\_p}) \to \\
\quad P A l p (\text{filter\_l\_p}) \to P0 A a l p \text{true} (a :: \text{filter\_l\_p}) \to \\
(\forall A a l p, \text{filter\_ind} A l p (\text{filter\_l\_p}) \to \\
\quad P A l p (\text{filter\_l\_p}) \to P0 A a l p \text{false} (\text{filter\_l\_p}) \to \\

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

$$\Pi \ A \ (l : \ list \ A), \ app \ l \ [] = l$$
Eliminating calls

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

\[\Pi A \ l : \ list \ A, \ app \ l \ [] = l\]

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

\[(\lambda \ l \ l' : \ list \ A) \ (r : app_{\text{comp}} \ l \ l'),
\]

\[l' = [] \rightarrow r = app \ l \ [] \rightarrow app \ l \ [] = l\]

\[l \ [] \ (app \ l \ [])\]

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.
Perspectives

- 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
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
```
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
instance (Eq a) ⇒ Eq [a] where

  [] == [] = True

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

  _ == _ = False
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 ...
Outline

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.
Motivations

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

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

- **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

\[ \text{Class } \text{Id } (\alpha_1 : \tau_1) \cdots (\alpha_n : \tau_n) := \{ f_1 : \phi_1 ; \cdots ; f_m : \phi_m \}. \]
A cheap implementation

- Parametrized dependent records

Record \texttt{Id} \ (\alpha_1 : \tau_1) \cdots (\alpha_n : \tau_n) := \\
\{ f_1 : \phi_1 ; \cdots ; f_m : \phi_m \}.
Parametrized dependent records

\[
\text{Record } \text{Id} \ (\alpha_1 : \tau_1) \cdots (\alpha_n : \tau_n) := \\
\{ f_1 : \phi_1 ; \cdots ; f_m : \phi_m \}.
\]

Instances are just definitions of type \( \text{Id} \rightarrow \tau_n \).
A cheap implementation

- Parametrized dependent records

  Record \( \text{Id} \ (\alpha_1 : \tau_1) \cdots (\alpha_n : \tau_n) := \{ f_1 : \phi_1 ; \cdots ; f_m : \phi_m \} \).

  Instances are just definitions of type \( \text{Id} \rightarrow t_n \).

- Custom implicit arguments of projections

  \( f_1 : \forall \alpha_n : \tau_n \), \( \text{Id} \alpha_n \rightarrow \phi_1 \)
Parametrized dependent records

\[
\text{Record } \text{ld} \ (\alpha_1 : \tau_1) \cdots (\alpha_n : \tau_n) := \{ f_1 : \phi_1 ; \cdots ; f_m : \phi_m \}.
\]

Instances are just definitions of type \( \text{ld} \overrightarrow{t_n} \).

Custom implicit arguments of projections

\[
f_1 : \forall \{ \alpha_n : \tau_n \}, \{ \text{ld} \overrightarrow{\alpha_n} \} \rightarrow \phi_1
\]
Elaboration with classes, an example

\[(\lambda x \ y \ : \ \text{bool} . \ \text{eqb} \ x \ y)\]
Elaboration with classes, an example

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

$$\leadsto \{ \text{Implicit arguments} \}$$

$$(\lambda x \ y : \text{bool}. \ @\text{eqb} (\ ?_A : \text{Type}) (\ ?_eq : \text{Eq} \ ?_A) \ x \ y)$$
Elaboration with classes, an example

\[(\lambda x \ y : \text{bool}. \ \text{eqb} \ x \ y)\]
\[\leadsto \{ \text{Implicit arguments} \}\]
\[\left(\lambda x \ y : \text{bool}. \ \text{@eqb} \ (\text{?}_A : \text{Type}) \ (\text{?}_{eq} : \text{Eq} \ ?_A) \ x \ y\right)\]
\[\leadsto \{ \text{Unification} \}\]
\[\left(\lambda x \ y : \text{bool}. \ \text{@eqb} \ \text{bool} \ (\text{?}_{eq} : \text{Eq} \ \text{bool}) \ x \ y\right)\]
Elaboration with classes, an example

\[(\lambda x \; y : \text{bool}. \; \text{eqb} \; x \; y)\]
\[\leadsto \{ \text{Implicit arguments} \} \]
\[\quad (\lambda x \; y : \text{bool}.\; \@\text{eqb} \; (?_A : \text{Type}) \; (?_{eq} : \text{Eq} \; ?_A) \; x \; y)\]
\[\leadsto \{ \text{Unification} \} \]
\[\quad (\lambda x \; y : \text{bool}.\; \@\text{eqb} \; \text{bool} \; (?_{eq} : \text{Eq} \; \text{bool}) \; x \; y)\]
\[\leadsto \{ \text{Proof search for Eq bool returns Eq_bool} \} \]
\[\quad (\lambda x \; y : \text{bool}.\; \@\text{eqb} \; \text{bool} \; \text{Eq_bool} \; x \; y)\]
Type Class resolution

Proof-search tactic with instances as lemmas:

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

- SLD resolution, higher-order unification.
- Returns the first solution only
- Extensible through $L_{\text{tac}}$
Numeric overloading

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

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

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 \textbf{Num} \( \alpha \) := \{ zero : \( \alpha \); one : \( \alpha \); plus : \( \alpha \to \alpha \to \alpha \) \}.

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

\textbf{Instance} Z_num : \textbf{Num} \( \mathbb{Z} \) :=
\{ zero := 0\%Z ; one := 1\%Z ; plus := Zplus \}.

\texttt{Notation} “0” := zero.
\texttt{Notation} “1” := one.
\texttt{Infix} “+” := plus.
Numeric overloading

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

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

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

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

\textit{Notation}”0” := zero.
\textit{Notation}”1” := one.
\textit{Infix}”+” := plus.

\textbf{Check (}$\lambda$ \textit{x : nat, x + (1 + 0 + x))}.
\textbf{Check (}$\lambda$ \textit{x : Z, x + (1 + 0 + x))}.

\textbf{Defaulting}
\textbf{Check (}$\lambda$ \textit{x, x + 1).}
Dependent classes

Class Reflexive \{ A \} (R : relation A) :=

refl : \forall \ x, R x x.
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.
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 \(\iff\) P.
Proof. apply refl. Qed.

Goal \(\forall\) A (x : A), x = x.
Proof. intros A ; apply refl. Qed.
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 \('\{\text{Num} \ \alpha\}\) :=
\{ \text{div} : \alpha \rightarrow \{ y : \alpha \mid y \neq 0 \} \rightarrow \alpha \}.

Class Equivalence \(\alpha\) :=
\{ \text{equiv_refl} \triangleright \text{Reflexive} \ \alpha \;
\text{equiv_sym} \triangleright \text{Symmetric} \ \alpha \;
\text{equiv_trans} \triangleright \text{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:


- 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:


- 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).
Current Issues

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.
Summary

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).
Ongoing and future work

**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.
The End

Elaborations in Type Theory

Matthieu Sozeau

Harvard University

DTP’10
July 10th 2010
Edinburgh, UK