Elaborations in Type Theory

Matthieu Sozeau

Harvard University

DTP'10
July 10th 2010
Edinburgh, UK

This talk

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

In the design spaces...

- Proof language design:
  - Separate tactic language $\mathcal{L}_{\mathsf{tac}}$.
  - 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).

Programming with tactics

```
Lemma eucl_dev : ∀ n, n > 0 → ∀ m : nat,
  { (q, r) : nat × nat | n > r ∧ m = q × 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 gtbn.
  ∃ (0, n); simpl; auto with arith.
Qed.
```

Programming directly

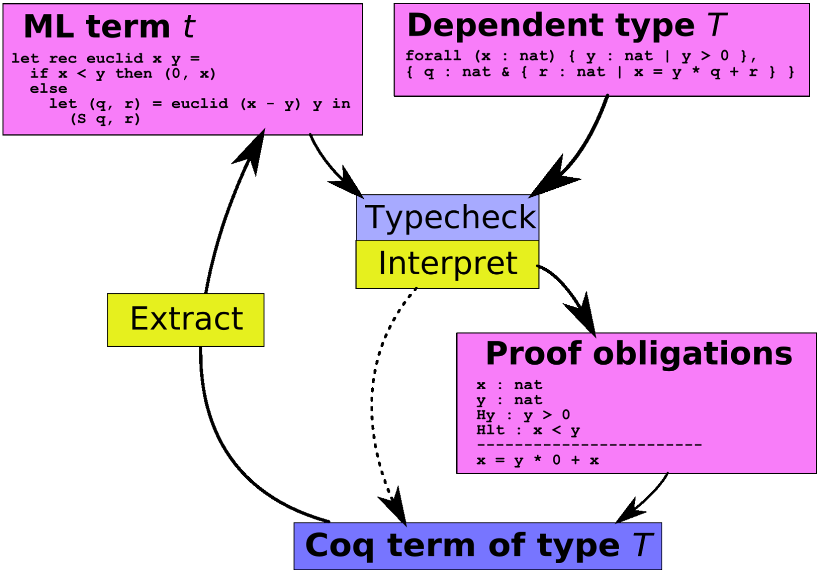
```
Check ((λ (b : nat) (H : b > 0) (a : nat),
Wf_nat.gt_wf_rec a (λ n : nat, diveucl n b)
  (λ (n : nat) (H0 : Π m : nat, n > m → diveucl m b),
   sumbool_rec (λ _ : {b ≤ n} + {b > n}, diveucl n b)
     (λ lebn : b ≤ n,
      diveucl_rec (n - b) b (λ _ : diveucl (n - b) b, diveucl n b)
        (λ (q 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).
```

Thesis

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



Hello World: Euclidean division

DEMO

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.

From "*Predicate subtyping*" ...

PVS

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

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

From “*Predicate subtyping*” . . .

PVS

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

+ Practical success ;

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

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).

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

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).

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

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

...to Subset Coercions

1. A property-irrelevant language (Russell) with decidable typing

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

$$\frac{\Gamma \vdash t : T \qquad \boxed{\Gamma, x : T \vdash P : \texttt{Prop}}}{\Gamma \vdash \qquad t \;\; : \{\, x : T \mid P \,\}}$$

. . . to Subset Coercions

1. A property-irrelevant language (Russell) with decidable typing
2. A total interpretation to Coq terms with holes

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

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

...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.

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

$$\frac{\Gamma \vdash t : T \qquad \Gamma, x : T \vdash P : \mathtt{Prop} \qquad \boxed{\Gamma \vdash p : P[t/x]}}{\Gamma \vdash \mathsf{exist}\ t\ p : \{\ x : T \mid P\ \}}$$

Russell typing $\vdash$ and coercion $\rhd$

Calculus of Constructions with

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

Calculus of Constructions with

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

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

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

RUSSELL typing $\vdash$ and coercion $\rhd$

Calculus of Constructions with

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

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

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

Example

$$\frac{\Gamma \vdash 0 : \mathbb{N} \qquad \Gamma \vdash \mathbb{N} \rhd \{ x : \mathbb{N} \mid x \neq 0 \}}{\Gamma \vdash 0 : \{ x : \mathbb{N} \mid x \neq 0 \}}$$

Russell typing $\vdash$ and coercion $\rhd$

Calculus of Constructions with

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

$$\Gamma \vdash \{\, x : U \mid P \,\} \rhd U$$

$$\Gamma \vdash U \rhd \{\, x : U \mid P \,\}$$

Example

$$\cfrac{\cfrac{\Gamma \vdash 0 : \mathbb{N} \qquad \Gamma \vdash \mathbb{N} \rhd \{\, x : \mathbb{N} \mid x \neq 0 \,\}}{\Gamma \vdash 0 : \{\, x : \mathbb{N} \mid x \neq 0 \,\}}}{\Gamma \vdash \,?\, : 0 \neq 0}$$

Metatheory

Theorem (Subject Reduction)

If $\Gamma \vdash t : T$ *and* $t \to_\beta u$ *then* $\Gamma \vdash u : T$

Proof.

Using the TPOSR technique due to Robin Adams. $\square$

Theorem (Decidability of type checking and type inference)

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

The target system : CIC with metavariables

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

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

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

From $\textsc{Russell}$ to $\textsc{Coq}$

The target system : CIC with metavariables

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

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

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

We can build a sound interpretation $[\![\_]\!]_\Gamma$ from $\textsc{Russell}$ to $\mathrm{CIC}_?$:

Theorem (Soundness)

If $\Gamma \vdash t : T$ *then* $[\![\Gamma]\!] \vdash_? [\![t]\!]_\Gamma : [\![T]\!]_\Gamma$ *(assuming proof-irrelevance).*

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

Interpretation of coercions

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

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

Interpretation of coercions

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

$$\frac{\Gamma \vdash_? T \equiv_\beta U : s}{\Gamma \vdash_? \quad : T \rhd U}$$

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

Interpretation of coercions

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

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

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

Interpretation of coercions

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

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

$$\Gamma \vdash_? \qquad : \{\, x : T \mid P \,\} \rhd T$$

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

Interpretation of coercions

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

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

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

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

Interpretation of coercions

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

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

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

$$\Gamma \vdash_? \qquad\qquad : T \rhd \{ x : T \mid P \}$$

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

Interpretation of coercions

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

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

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

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

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

Interpretation of coercions

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

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

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

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

Example

$$\frac{\Gamma \vdash_? 0 : \mathbb{N} \qquad \Gamma \vdash_? \textit{exist}\ \bullet\ ?_{\bullet \neq 0} : \mathbb{N} \rhd \{\ x : \mathbb{N} \mid x \neq 0\ \}}{\Gamma \vdash_? \textit{exist}\ 0\ ?_{0 \neq 0} : \{\ x : \mathbb{N} \mid x \neq 0\ \}}$$

Pattern-matching revisited

Put logic into the terms (think symbolic evaluation).

Let $e : \mathbb{N}$:

$$
\begin{array}{lll}
\texttt{match}\ e & \texttt{return} & T\ \texttt{with} \\
\mid \mathsf{S}\ n \Rightarrow & & \quad t_1 \\
\mid 0 \Rightarrow & & t_2 \\
\texttt{end} & &
\end{array}
$$

Pattern-matching revisited

Put logic into the terms (think symbolic evaluation).

Let $e : \mathbb{N}$:

$$\begin{array}{ll}
\texttt{match}\ e\ \texttt{as}\ t & \texttt{return}\ t = e \rightarrow T\ \texttt{with} \\
\mid \mathsf{S}\ n \Rightarrow & \texttt{fun}\ (H : \mathsf{S}\ n = e) \Rightarrow t_1 \\
\mid \mathsf{0} \Rightarrow & \texttt{fun}\ (H : \mathsf{0} = e) \Rightarrow t_2 \\
\texttt{end} & (\texttt{refl_equal}\ e)
\end{array}$$

Pattern-matching revisited

Put logic into the terms (think symbolic evaluation).

Further refinements

- Each branch typed only once ;

Let $e : \mathbb{N}$:

$$
\begin{array}{ll}
\texttt{match}\ e\ \texttt{as}\ t & \texttt{return}\ t = e \rightarrow T\ \texttt{with} \\
\mid \mathsf{S}\ (\mathsf{S}\ n) \Rightarrow & \texttt{fun}\ (H : \mathsf{S}\ (\mathsf{S}\ n) = e) \Rightarrow t_1 \\
\mid n \Rightarrow & \texttt{fun}\ (H : n = e) \Rightarrow\ t_2 \\
\texttt{end} & (\texttt{refl\_equal}\ e)
\end{array}
$$

Pattern-matching revisited

Put logic into the terms (think symbolic evaluation).

Further refinements

- Each branch typed only once ;

Let $e : \mathbb{N}$:

$$
\begin{array}{ll}
\texttt{match}\ e\ \texttt{as}\ t & \texttt{return}\ t = e \rightarrow T\ \texttt{with} \\
\mid \mathsf{S}\ (\mathsf{S}\ n) \Rightarrow & \texttt{fun}\ (H : \mathsf{S}\ (\mathsf{S}\ n) = e) \Rightarrow t_1 \\
\mid \mathsf{S}\ 0 \Rightarrow & \texttt{fun}\ (H : \mathsf{S}\ 0 = e) \Rightarrow\ t_2 \\
\mid 0 \Rightarrow & \texttt{fun}\ (H : 0 = e) \Rightarrow t_2 \\
\texttt{end} & (\texttt{refl\_equal}\ e)
\end{array}
$$

Pattern-matching revisited

Put logic into the terms (think symbolic evaluation).

Further refinements

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

Let $e : \mathbb{N}$:

$$
\begin{array}{ll}
\texttt{match}\ e\ \texttt{as}\ t & \texttt{return}\ t = e \to T\ \texttt{with} \\
\mid \mathsf{S}\ (\mathsf{S}\ n) \Rightarrow & \texttt{fun}\ (H : \mathsf{S}\ (\mathsf{S}\ n) = e) \Rightarrow t_1 \\
\mid n \Rightarrow & \texttt{fun}\ (H : n = e) \Rightarrow \texttt{let}\ H' : \forall n', n \neq \mathsf{S}\ (\mathsf{S}\ n')\ \texttt{in}\ t_2 \\
\texttt{end} & (\texttt{refl\_equal}\ e)
\end{array}
$$

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 : vector n:

```
match e                 return                     T with
| vnil ⇒                               t₁
| vcons x n' v' ⇒
                                  t₂
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 : vector n:

$$\begin{array}{l}
\texttt{match}\ e\ \texttt{as}\ t\ \texttt{in}\ \mathsf{vector}\ n'\ \texttt{return}\ n' = n \to \mathsf{JMeq}\ t\ e \to T\ \texttt{with} \\
\mid \mathsf{vnil} \Rightarrow \texttt{fun}\ (H : 0 = n)(Hv : \mathsf{JMeq}\ \mathsf{vnil}\ e) \Rightarrow t_1 \\
\mid \mathsf{vcons}\ x\ n'\ v' \Rightarrow \texttt{fun}\ (H : \mathsf{S}\ n' = n) \\
\qquad (Hv : \mathsf{JMeq}\ (\mathsf{vcons}\ x\ n'\ v')\ e) \Rightarrow t_2 \\
\texttt{end}\ (\mathsf{refl_equal}\ n)(\mathsf{JMeq_refl}\ e)
\end{array}$$

Extension to inductive families

Extend the coercion system to handle inductive families:

$$\dfrac{\dots \qquad \dfrac{\Gamma \vdash_{?} ?_{x=y} : x = y}{\Gamma \vdash_{?} \mathsf{eq\_rec}\ \mathbb{N}\ x\ \mathsf{vector}\quad \bullet\ \ y\ ?_{x=y} : \mathsf{vector}\ x \rhd \mathsf{vector}\ y}}{\Gamma \vdash_{?} \mathsf{eq\_rec}\ \mathbb{N}\ x\ \mathsf{vector}\ \ v\ y\ ?_{x=y} : \mathsf{vector}\ y}$$

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

Recursion

Support for well-founded recursion and measures.

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

Recursion

Support for well-founded recursion and measures.

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

$$\frac{\begin{array}{lcl} a & : & \mathbb{N} \\ f & : & \{x : \mathbb{N} \mid x < a\} \to \mathbb{N} \end{array}}{\begin{array}{lcl} b & : & \mathbb{N} \end{array}}$$

Summary

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

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

Summary

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)...

Summary

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.

Summary

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.

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

Compilation setup

Three phases:

1. Generation of a splitting tree from the clauses
2. Translation from the splitting tree to Coq terms with holes
3. Proofs of the obligations using a mix of ML and $\mathcal{L}_{\mathsf{tac}}$ code. Essentially `dependent destruction`.

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

Searching for a splitting tree

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

Goal Starting with $\mathsf{f}\ \Delta : \tau := \overrightarrow{\overrightarrow{p} \ldots}$, 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 : 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 $\vdash n\ m : n\ m$: nat, n, [
  Split(m : nat $\vdash O\ m : n\ m$: nat, m, [
    Compute($\vdash O\ O : n\ m$: nat, Program(left eq_refl)),
    Compute(m : nat $\vdash O\ (S\ m) : n\ m$: nat, Program(right _))]),
  Split($n\ m$: nat $\vdash (S\ n)\ m : n\ m$: nat, m, [
    Compute(n : nat $\vdash (S\ n)\ O : n\ m$: nat, $\ldots$),
    Compute($n\ m$: nat $\vdash (S\ n)\ (S\ m) : n\ m$: nat,
      Refine(equal $n\ m$,
        idsubst($n\ m$: nat$, x : \{n = m\} + \{n \neq m\}$), $\ldots$))])])

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.

Subterm relations and well-founded recursion

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

Subterm relations and well-founded recursion

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

- General definition of direct subterm:
  $\mathsf{I}_{subfull} : \Pi\ \Delta_l\ \Delta_r, \mathsf{I}\ \overline{\Delta_l} \rightarrow \mathsf{I}\ \overline{\Delta_r} \rightarrow \mathtt{Prop}$

Subterm relations and well-founded recursion

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

- General definition of direct subterm:
  $\mathsf{I}_{subfull} : \Pi\ \Delta_l\ \Delta_r, \mathsf{I}\ \overline{\Delta_l} \rightarrow \mathsf{I}\ \overline{\Delta_r} \rightarrow \texttt{Prop}$
- Wrap the inductive type in a sigma and define an homogeneous relation on the sigma type: $\mathsf{I}_{sub} : \mathrm{relation}\ (\Sigma\ \Delta, \mathsf{I}\ \overline{\Delta})$

Subterm relations and well-founded recursion

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

- General definition of direct subterm:
  $\mathsf{I}_{subfull} : \Pi\ \Delta_l\ \Delta_r, \mathsf{I}\ \overline{\Delta_l} \to \mathsf{I}\ \overline{\Delta_r} \to \mathtt{Prop}$
- Wrap the inductive type in a sigma and define an homogeneous relation on the sigma type: $\mathsf{I}_{sub} : \mathsf{relation}\ (\Sigma\ \Delta, \mathsf{I}\ \overline{\Delta})$
- 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 : Type)
  : ∀ H H0 : nat, vector A H → vector A H0 → Prop :=
    vector_strict_subterm_1_1 : ∀ (a : A) (n : nat) (H : vector A n),
      vector_strict_subterm A n (S n) H (Vcons a H).

Check vector_subterm : ∀ A : Type, relation {index : nat & vector A index}.
```

Subterm relation example: vectors

```
Derive Subterm for vector.

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

Check vector_subterm : ∀ A : Type, relation {index : nat & vector A index}.

Equations unzip {A B n} (v : vector (A × B) n)
  : vector A n × vector B n :=
unzip A B n v by rec v :=
unzip A B ?(O) Vnil := (Vnil, Vnil) ;
unzip A B ?(S n) (Vcons (pair x y) n v) with unzip v := {
  | (pair xs ys) := (Vcons x xs, Vcons y ys) }.
```

Equations

- Equations hold definitionally in $\mathrm{CCI} + \mathsf{K}$ (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.

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

Eliminating calls

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

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

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

$$\begin{array}{l}(\lambda\ (l\ l' : \mathsf{list}\ A)\ (r : \mathsf{app}_{\mathsf{comp}}\ l\ l'), \\ \qquad l' = [] \to r = \mathsf{app}\ l\ [] \to \mathsf{app}\ l\ [] = l) \\ \quad l\ []\ (\mathsf{app}\ l\ [])\end{array}$$

Directly apply the elimination principle and simplify the equations.

Conclusion

A function definition package handling:

- Full, nested dependent pattern-matching
- Structural and well-founded recursion on dependent types
- Generation of useful support lemmas for reasoning a posteriori

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.

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

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

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

Outline

Motivations

- Overloading in programs, specifications and proofs.

Motivations

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

  Class Eq A := {
  eqb : $A \rightarrow A \rightarrow$ bool ;
  eq_eqb : $\forall\ x\ y : A,\ x = y \leftrightarrow$ eqb $x\ y =$ true }.

Motivations

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

  Class Eq A := {
  eqb : $A \rightarrow A \rightarrow$ bool ;
  eq_eqb : $\forall\ x\ y : A,\ x = y \leftrightarrow$ eqb $x\ y =$ true }.

- Extensions Dependent types give new power to type classes.

  Class Reflexive A (R : relation A) :=
  reflexive : $\forall\ x,\ R\ x\ x$.

A cheap implementation

- Parametrized dependent records

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

A cheap implementation

- Parametrized dependent records

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

A cheap implementation

- Parametrized dependent records

  $\texttt{Record}\ \mathsf{Id}\ (\alpha_1 : \tau_1) \cdots (\alpha_n : \tau_n) :=$
  $\{\mathsf{f}_1 : \phi_1\ ;\ \cdots\ ; \mathsf{f}_m : \phi_m\}.$

  Instances are just definitions of type $\mathsf{Id}\ \overrightarrow{t_n}$.

A cheap implementation

- Parametrized dependent records

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

  Instances are just definitions of type $\mathsf{Id}\ \overrightarrow{t_n}$.

- Custom implicit arguments of projections

  $\mathsf{f}_1 : \forall\ \overrightarrow{\alpha_n : \tau_n}\ ,\ \mathsf{Id}\ \overrightarrow{\alpha_n}\ \rightarrow \phi_1$

A cheap implementation

- Parametrized dependent records

  $\texttt{Record}\ \mathsf{Id}\ (\alpha_1 : \tau_1) \cdots (\alpha_n : \tau_n) :=$
  $\{\mathsf{f}_1 : \phi_1\ ;\ \cdots ; \mathsf{f}_m : \phi_m\}.$

  Instances are just definitions of type $\mathsf{Id}\ \overrightarrow{t_n}$.

- Custom implicit arguments of projections

  $\mathsf{f}_1 : \forall \{\overrightarrow{\alpha_n : \tau_n}\}, \{\mathsf{Id}\ \overrightarrow{\alpha_n}\} \rightarrow \phi_1$

Elaboration with classes, an example

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

Elaboration with classes, an example

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

$\rightsquigarrow$ { Implicit arguments }

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

Elaboration with classes, an example

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

$\rightsquigarrow$ { Implicit arguments }

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

$\rightsquigarrow$ { Unification }

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

Elaboration with classes, an example

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

$\leadsto$ { Implicit arguments }

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

$\leadsto$ { Unification }

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

$\leadsto$ { Proof search for Eq bool returns Eq_bool }

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

Type Class resolution

Proof-search tactic with instances as lemmas:

$$A : \texttt{Type}, \mathit{eqa} : \mathsf{Eq}\ A \vdash ? : \mathsf{Eq}\ (\mathsf{list}\ A)$$

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

Numeric overloading

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

Numeric overloading

```
Class Num α := { zero : α ; one : α ; plus : α → α → α }.

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 α := { zero : α ; one : α ; plus : α → α → α }.

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 α := { zero : α ; one : α ; plus : α → α → α }.

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 (λ x : nat, x + (1 + 0 + x)).
Check (λ x : Z, x + (1 + 0 + x)).
```

Numeric overloading

```
Class Num α := { zero : α ; one : α ; plus : α → α → α }.

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 (λ x : nat, x + (1 + 0 + x)).
Check (λ x : Z, x + (1 + 0 + x)).
```

Defaulting

```
Check (λ x, x + 1).
```

Dependent classes

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

Dependent classes

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

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

Instance iff_refl : Reflexive iff.
Proof. red. tauto. Qed.

Goal ∀ P, P ↔ P.
Proof. apply refl. Qed.

Goal ∀ A (x : A), x = x.
Proof. intros A ; apply refl. Qed.
```

Dependent classes

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

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

Instance iff_refl : Reflexive iff.
Proof. red. tauto. Qed.

Goal ∀ P, P ↔ P.
Proof. apply refl. Qed.

Goal ∀ 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 α} :=
  { div : α → { y : α | y ≠ 0 } → α }.
Class Equivalence α :=
  { equiv_refl :> Reflexive α ;
    equiv_sym :> Symmetric α ;
    equiv_trans :> Transitive α }
```

+ 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)...

Summary

✓ 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).

Summary

- ✓ 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).

Summary

- ✓ 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 Structure`s 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.

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.

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.

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.

Ongoing and future work

Program, Equations & Coq

- We hinted at the important foundational issues with η-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 η-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.

Conclusions

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