

Subset coercions in Coq

MATTHIEU SOZEAU

under direction of CHRISTINE PAULIN-MOHRING

Paris-Sud 11 University

LRI - DÉMONS Team & INRIA FUTURS - PROVAL Project

TYPES'06 Workshop

18-21 April 2006

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

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

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)
```

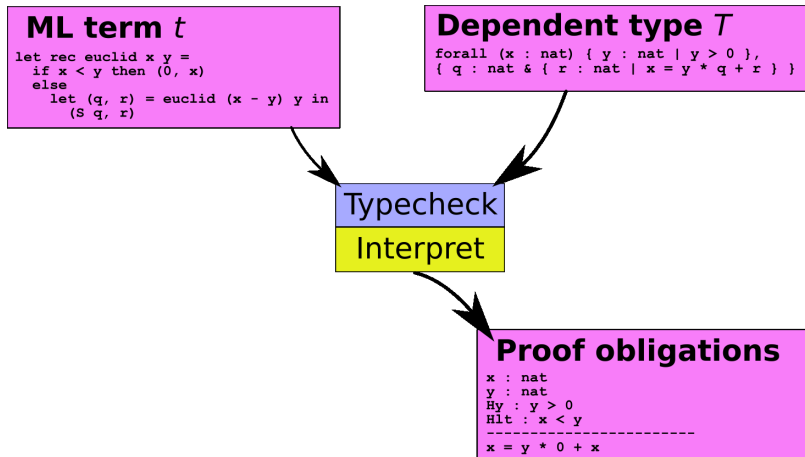
Dependent type T

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

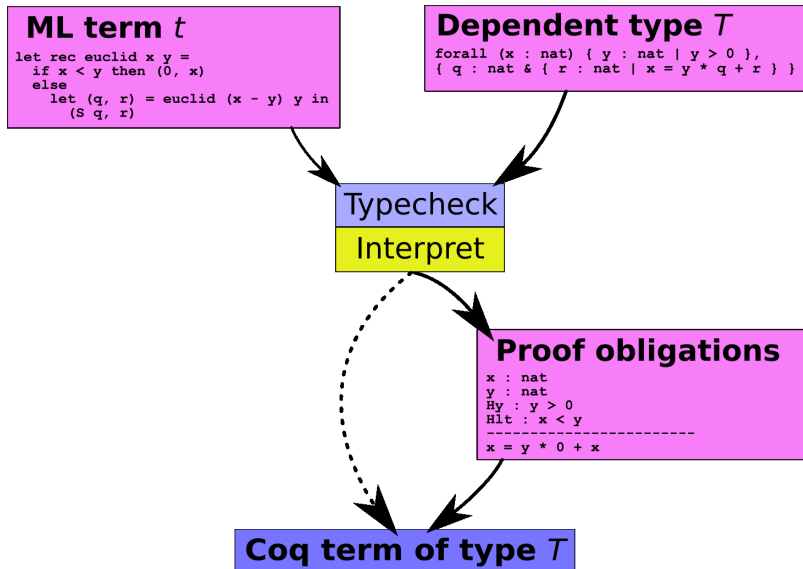
```
graph TD; A[ML term t] --> C[Typecheck]; B[Dependent type T] --> C;
```

Typecheck

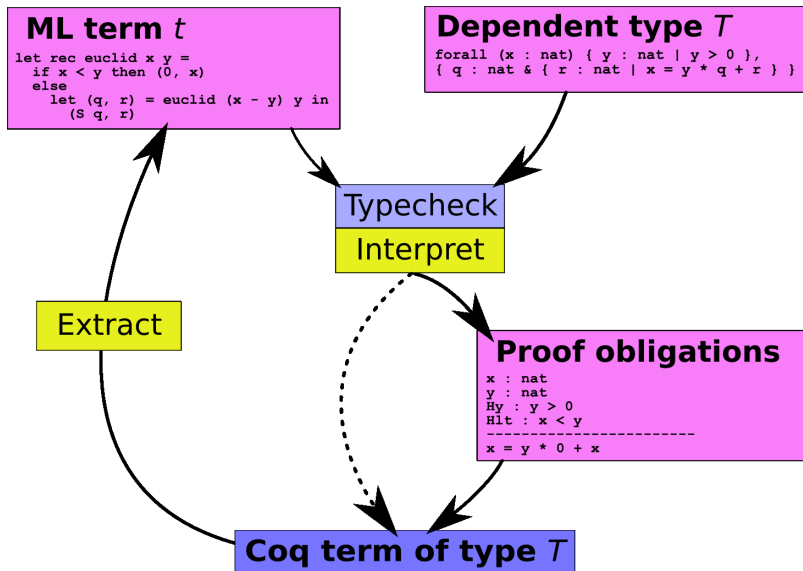
The Big Picture



The Big Picture



The Big Picture



- 1 The idea
- 2 Theoretical development
 - RUSSELL
 - Traduction in Coq
- 3 The implementation

A simple idea

Definition

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

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

A simple idea

Definition

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

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

Adapting the idea

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

From “*Predicate subtyping*”...

PVS

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

| | | | |
|--------------------------|---------|--------------------------|-------------|
| $t : \{ x : T \mid P \}$ | used as | $t : T$ | ok |
| $t : T$ | used as | $t : \{ x : T \mid P \}$ | if $P[t/x]$ |

From “*Predicate subtyping*”...

PVS

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

$t : \{ x : T \mid P \}$ used as $t : T$ ok

$t : T$ used as $t : \{ x : T \mid P \}$ if $P[t/x]$

- + Practical success ;

From “*Predicate subtyping*”...

PVS

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

$t : \{ x : T \mid P \}$ used as $t : T$ ok
 $t : T$ used as $t : \{ x : T \mid P \}$ if $P[t/x]$

- + Practical success ;
- No strong safety guarantee in PVS.

... to Subset coercions

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

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

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

... to Subset coercions

- 1 A property-irrelevant language (RUSSELL) with **decidable** typing ;
- 2 A total traduction to Coq terms with holes ;

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

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

... to Subset coercions

- 1 A property-irrelevant language (RUSSELL) with **decidable** typing ;
- 2 A total traduction to Coq terms with holes ;
- 3 A method to turn the holes into proof obligations.

Outline

- 1 The idea
- 2 Theoretical development
 - RUSSELL
 - Traduction in Coq
- 3 The implementation

RUSSELL's typing \vdash and coercion \triangleright

Calculus of Constructions - CONV +

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

RUSSELL's typing \vdash and coercion \triangleright

Calculus of Constructions - CONV +

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

$$\text{SUB-CONV} \frac{T \equiv_{\beta\pi} U}{T \triangleright U} \quad \text{SUB-TRANS} \frac{S \triangleright T \quad T \triangleright U}{S \triangleright U}$$

RUSSELL's typing \vdash and coercion \triangleright

Calculus of Constructions - CONV +

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

$$\text{SUB-CONV} \frac{T \equiv_{\beta\pi} U}{T \triangleright U} \quad \text{SUB-TRANS} \frac{S \triangleright T \quad T \triangleright U}{S \triangleright U}$$

$$\text{SUB-SUBSET} \frac{U \triangleright V}{\{x : U \mid P\} \triangleright V} \quad \text{SUB-PROOF} \frac{U \triangleright V}{U \triangleright \{x : V \mid P\}}$$

RUSSELL's typing \vdash and coercion \triangleright

Calculus of Constructions - CONV +

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

$$\text{SUB-CONV} \frac{T \equiv_{\beta\pi} U}{T \triangleright U} \quad \text{SUB-TRANS} \frac{S \triangleright T \quad T \triangleright U}{S \triangleright U}$$

$$\text{SUB-SUBSET} \frac{U \triangleright V}{\{x : U \mid P\} \triangleright V} \quad \text{SUB-PROOF} \frac{U \triangleright V}{U \triangleright \{x : V \mid P\}}$$

$$\frac{0 : \text{nat} \quad \text{nat} \triangleright \{x : \text{nat} \mid x \neq 0\}}{0 : \{x : \text{nat} \mid x \neq 0\}}$$

RUSSELL's typing \vdash and coercion \triangleright

Calculus of Constructions - CONV +

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

$$\text{SUB-CONV} \frac{T \equiv_{\beta\pi} U}{T \triangleright U} \quad \text{SUB-TRANS} \frac{S \triangleright T \quad T \triangleright U}{S \triangleright U}$$

$$\text{SUB-SUBSET} \frac{U \triangleright V}{\{x : U \mid P\} \triangleright V} \quad \text{SUB-PROOF} \frac{U \triangleright V}{U \triangleright \{x : V \mid P\}}$$

$$\frac{0 : \text{nat} \quad \text{nat} \triangleright \{x : \text{nat} \mid x \neq 0\}}{0 : \{x : \text{nat} \mid x \neq 0\}}$$

$?: 0 \neq 0$

RUSSELL's typing \vdash and coercion \triangleright

Calculus of Constructions - CONV +

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

$$\text{SUB-CONV} \frac{T \equiv_{\beta\pi} U}{T \triangleright U} \quad \text{SUB-TRANS} \frac{S \triangleright T \quad T \triangleright U}{S \triangleright U}$$

$$\text{SUB-SUBSET} \frac{U \triangleright V}{\{x : U \mid P\} \triangleright V} \quad \text{SUB-PROOF} \frac{U \triangleright V}{U \triangleright \{x : V \mid P\}}$$

$$\text{SUB-PROD} \frac{U \triangleright T \quad V \triangleright W}{\prod x : T.V \triangleright \prod x : U.W}$$

Results

Theorem (Decidability of type checking)

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

$$\text{APP} \frac{\Gamma \vdash f : T \quad T \triangleright \Pi x : A. B \quad \Gamma \vdash e : E \quad E \triangleright A}{\Gamma \vdash (f e) : B[e/x]}$$

Results

Theorem (Decidability of type checking)

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

Lemma (Elimination of transitivity)

If $T \triangleright U \wedge U \triangleright V$ then $T \triangleright V$.

$$\text{APP} \frac{\Gamma \vdash f : T \quad T \triangleright \Pi x : A. B \quad \Gamma \vdash e : E \quad E \triangleright A}{\Gamma \vdash (f e) : B[e/x]}$$

- 1 The idea
- 2 Theoretical development
 - RUSSELL
 - Traduction in Coq
- 3 The implementation

The target system

CIC with metavariables

$$\frac{\Gamma \vdash_? t : T \quad \Gamma \vdash_? p : P[t/x]}{\Gamma \vdash_? \text{elt } T P t p : \{x : T \mid P\}}$$
$$\frac{\Gamma \vdash_? t : \{x : T \mid P\}}{\Gamma \vdash_? \sigma_1 t : T} \quad \frac{\Gamma \vdash_? t : \{x : T \mid P\}}{\Gamma \vdash_? \sigma_2 t : P[\sigma_1 t/x]}$$
$$\frac{\Gamma \vdash_? P : \mathbf{Prop}}{\Gamma \vdash_? ?_P : P}$$

The easy way

$$(\sigma_1 t)^\circ = t^\circ$$

$$(\text{elt } T P t p)^\circ = t^\circ$$

$$(\sigma_2 t)^\circ = \perp$$

$$(?p)^\circ = \perp$$

If $\Gamma \vdash_? t : T$ then $\Gamma^\circ \vdash t^\circ : T^\circ$ if $()^\circ$ is defined on Γ, t and T .

From Coq to RUSSELL and back

The easy way

$$\begin{aligned}(\sigma_1 t)^\circ &= t^\circ \\(\text{elt } T P t p)^\circ &= t^\circ \\(\sigma_2 t)^\circ &= \perp \\(?p)^\circ &= \perp\end{aligned}$$

If $\Gamma \vdash_? t : T$ then $\Gamma^\circ \vdash t^\circ : T^\circ$ if $()^\circ$ is defined on Γ, t and T .

The hard way

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

Traduction : deriving explicit coercions

Traduction for coercions

If $T \triangleright U$ then $\Gamma \vdash? c[\bullet] : T \triangleright U$ which implies

$$\llbracket \Gamma \rrbracket \vdash? \lambda x : \llbracket T \rrbracket_{\Gamma}.c[x] : \llbracket T \rrbracket_{\Gamma} \rightarrow \llbracket U \rrbracket_{\Gamma}.$$

Traduction : deriving explicit coercions

Traduction for coercions

If $T \triangleright U$ then $\Gamma \vdash? c[\bullet] : T \triangleright U$ which implies

$$\llbracket \Gamma \rrbracket \vdash? \lambda x : \llbracket T \rrbracket_{\Gamma}.c[x] : \llbracket T \rrbracket_{\Gamma} \rightarrow \llbracket U \rrbracket_{\Gamma}.$$

Definition

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

Traduction : deriving explicit coercions

Traduction for coercions

If $T \triangleright U$ then $\Gamma \vdash? c[\bullet] : T \triangleright U$ which implies

$$\llbracket \Gamma \rrbracket \vdash? \lambda x : \llbracket T \rrbracket_{\Gamma}.c[x] : \llbracket T \rrbracket_{\Gamma} \rightarrow \llbracket U \rrbracket_{\Gamma}.$$

Definition

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

Traduction : deriving explicit coercions

Traduction for coercions

If $T \triangleright U$ then $\Gamma \vdash? c[\bullet] : T \triangleright U$ which implies

$$\llbracket \Gamma \rrbracket \vdash? \lambda x : \llbracket T \rrbracket_{\Gamma}.c[x] : \llbracket T \rrbracket_{\Gamma} \rightarrow \llbracket U \rrbracket_{\Gamma}.$$

Definition

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

Traduction : deriving explicit coercions

Traduction for coercions

If $T \triangleright U$ then $\Gamma \vdash_{\eta} c[\bullet] : T \triangleright U$ which implies

$$\llbracket \Gamma \rrbracket \vdash_{\eta} \lambda x : \llbracket T \rrbracket_{\Gamma}.c[x] : \llbracket T \rrbracket_{\Gamma} \rightarrow \llbracket U \rrbracket_{\Gamma}.$$

Definition

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

Traduction : deriving explicit coercions

Traduction for coercions

If $T \triangleright U$ then $\Gamma \vdash_? c[\bullet] : T \triangleright U$ which implies

$$\llbracket \Gamma \rrbracket \vdash_? \lambda x : \llbracket T \rrbracket_{\Gamma}.c[x] : \llbracket T \rrbracket_{\Gamma} \rightarrow \llbracket U \rrbracket_{\Gamma}.$$

Definition

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

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

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

Traduction : deriving explicit coercions

Traduction for coercions

If $T \triangleright U$ then $\Gamma \vdash_? c[\bullet] : T \triangleright U$ which implies

$$\llbracket \Gamma \rrbracket \vdash_? \lambda x : \llbracket T \rrbracket_{\Gamma}.c[x] : \llbracket T \rrbracket_{\Gamma} \rightarrow \llbracket U \rrbracket_{\Gamma}.$$

Definition

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

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

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

Traduction : deriving explicit coercions

Traduction for coercions

If $T \triangleright U$ then $\Gamma \vdash_? c[\bullet] : T \triangleright U$ which implies

$$\llbracket \Gamma \rrbracket \vdash_? \lambda x : \llbracket T \rrbracket_{\Gamma}. c[x] : \llbracket T \rrbracket_{\Gamma} \rightarrow \llbracket U \rrbracket_{\Gamma}.$$

Definition

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

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

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

Example

$$\frac{\Gamma \vdash_? 0 : \mathit{nat} \quad \Gamma \vdash_? c : \mathit{nat} \triangleright \{ x : \mathit{nat} \mid x \neq 0 \}}{\Gamma \vdash_? (\mathit{elt} \bullet ?_{(x \neq 0)[\bullet/x]})[0] = \mathit{elt} \ 0 ?_{0 \neq 0} : \{ x : \mathit{nat} \mid x \neq 0 \}}$$

Traduction : interpretation of terms $\llbracket \cdot \rrbracket_{\Gamma}$

Example (Application)

$$\frac{\Gamma \vdash f : T \quad T \triangleright \Pi x : V.W : s \quad \Gamma \vdash u : U \quad U \triangleright V}{\Gamma \vdash (f u) : W[u/x]}$$

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

Theorem (Soundness)

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

Theoretical matters

$\vdash_?$'s equational theory :

$$(\beta) \quad (\lambda x : X.e) v \quad \equiv \quad e[v/x]$$

$$(\sigma_i) \quad \sigma_i (\text{elt } E P e_1 e_2) \quad \equiv \quad e_i$$

$$(\eta) \quad (\lambda x : X.e x) \quad \equiv \quad e \quad \text{if } x \notin FV(e)$$

$$(\text{SP}) \quad \text{elt } E P (\sigma_1 e) (\sigma_2 e) \quad \equiv \quad e$$

Theoretical matters

$\vdash_?$'s equational theory :

$$\begin{array}{llll} (\beta) & (\lambda x : X.e) v & \equiv & e[v/x] \\ (\sigma_i) & \sigma_i (\text{elt } E P e_1 e_2) & \equiv & e_i \\ (\eta) & (\lambda x : X.e x) & \equiv & e \quad \text{if } x \notin FV(e) \\ (\text{SP}) & \text{elt } E P (\sigma_1 e) (\sigma_2 e) & \equiv & e \\ (\sigma) & \text{elt } E P t p & \equiv & \text{elt } E P t' p' \quad \text{if } t \equiv t' \end{array}$$

\Rightarrow **Proof Irrelevance**

Outline

- 1 The idea
- 2 Theoretical development
 - RUSSELL
 - Traduction in Coq
- 3 The implementation

The PROGRAM vernacular

Architecture

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

- 1 Use the Coq parser : Program **Definition** $f : T := t.$;

The PROGRAM vernacular

Architecture

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

- 1 Use the Coq parser : Program **Definition** $f : T := t$;
- 2 Typecheck $\Gamma^\circ \vdash t : T$ and generate $\llbracket \Gamma^\circ \rrbracket \vdash? \llbracket t \rrbracket_{\Gamma^\circ} : \llbracket T \rrbracket_{\Gamma^\circ}$;

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

The PROGRAM vernacular

Architecture

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

- 1 Use the Coq parser : Program **Definition** $f : T := t$;
- 2 Typecheck $\Gamma^\circ \vdash t : T$ and generate $\llbracket \Gamma^\circ \rrbracket \vdash? \llbracket t \rrbracket_{\Gamma^\circ} : \llbracket T \rrbracket_{\Gamma^\circ}$;
- 3 Interactive proving of obligations ;

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

The PROGRAM vernacular

Architecture

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

- 1 Use the Coq parser : Program **Definition** $f : T := t.$;
- 2 Typecheck $\Gamma^\circ \vdash t : T$ and generate $\llbracket \Gamma^\circ \rrbracket \vdash? \llbracket t \rrbracket_{\Gamma^\circ} : \llbracket T \rrbracket_{\Gamma^\circ}$;
- 3 Interactive proving of obligations ;
- 4 Final definition.

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

The PROGRAM vernacular

Architecture

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

- 1 Use the Coq parser : Program **Definition** $f : T := t.$;
- 2 Typecheck $\Gamma^\circ \vdash t : T$ and generate $\llbracket \Gamma^\circ \rrbracket \vdash? \llbracket t \rrbracket_{\Gamma^\circ} : \llbracket T \rrbracket_{\Gamma^\circ}$;
- 3 Interactive proving of obligations ;
- 4 Final definition.

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

Remark (Restriction)

We assume $\Gamma^\circ = \Gamma$ and $\Gamma \vdash_{CCI} \llbracket T \rrbracket_{\Gamma} : s.$

PROGRAM : The list example

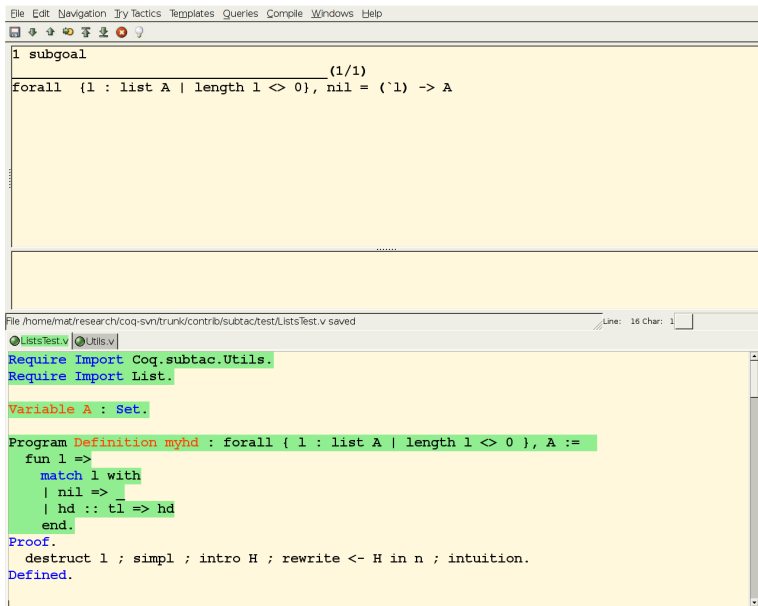
```
File Edit Navigation Try Tactics Templates Queries Compile Windows Help
Ready Line: 2 Char: 1
ListsTest.v Utilis.v

Notation ``t`` := (proj1_sig t) (at level 100) : core_scope.
Notation ''forall' { x : A | P }, Q'' :=
  (forall x:{x:A|P}, Q)
  (at level 200, x ident, right associativity).

Lemma subset_simpl : forall (A : Set) (P : A -> Prop)
  (t : sig P), P (` t).

Proof.
intros.
induction t.
simpl ; auto.
Qed.
```

PROGRAM : The list example



The screenshot shows a Coq IDE window with a menu bar (File, Edit, Navigation, Try Tactics, Templates, Queries, Compile, Windows, Help) and a toolbar. The main editor area is split into two panes. The top pane shows a proof goal:

```
1 subgoal
      (1/1)
forall { l : list A | length l <> 0 }, nil = (`l) -> A
```

The bottom pane shows the source code for ListsTest.v, with tabs for ListsTest.v and Utils.v. The code includes:

```
Require Import Coq.subtac.Utils.
Require Import List.

Variable A : Set.

Program Definition myhd : forall { l : list A | length l <> 0 }, A :=
  fun l =>
    match l with
    | nil => _
    | hd :: tl => hd
    end.

Proof.
  destruct l ; simpl ; intro H ; rewrite <- H in n ; intuition.
Defined.
```

PROGRAM : The list example

The screenshot shows a theorem prover interface with a menu bar (File, Edit, Navigation, Try Tactics, Templates, Queries, Compile, Windows, Help) and a toolbar. The main editor contains the following code:

```
(** val myhd : a list -> a **)

let myhd l =
  match proj1_sig l with
  | Nil -> assert false (* absurd case *)
  | Cons (hd, tl) -> hd
```

Below the code, the status bar shows "Ready" and "Line: 7 Char: 11". The interface also displays the extracted program and proof:

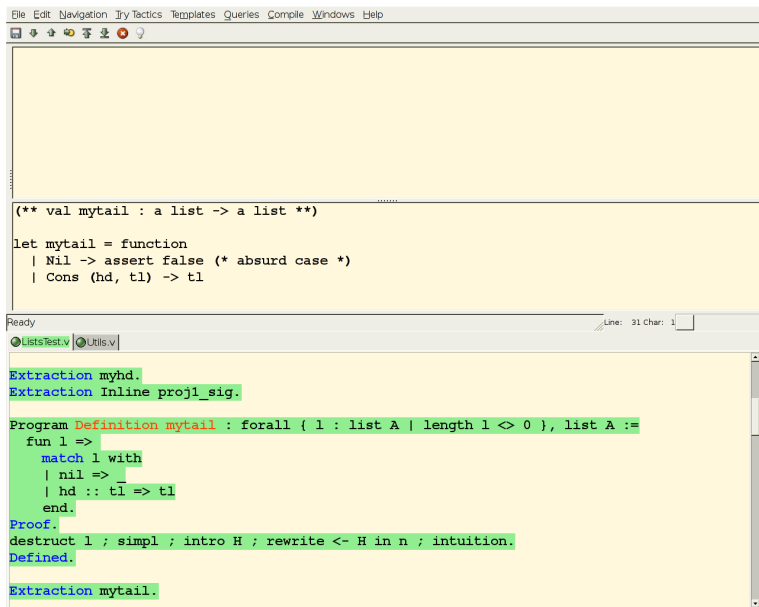
```
Variable A : Set.

Program Definition myhd : forall { l : list A | length l < 0 }, A :=
  fun l =>
    match l with
    | nil => _
    | hd :: tl => hd
  end.

Proof.
  destruct l ; simpl ; intro H ; rewrite <- H in n ; intuition.
Defined.

Extraction myhd.
Extraction Inline proj1_sig.
```


PROGRAM : The list example



The screenshot shows a proof assistant window with a menu bar (File, Edit, Navigation, Try Tactics, Templates, Queries, Compile, Windows, Help) and a toolbar. The main editor contains the following code:

```
(** val mytail : a list -> a list **)
.....
let mytail = function
  | Nil -> assert false (* absurd case *)
  | Cons (hd, tl) -> tl
```

Below the code, the status bar shows "Ready" and "Line: 31 Char: 1". The bottom panel shows the extracted code:

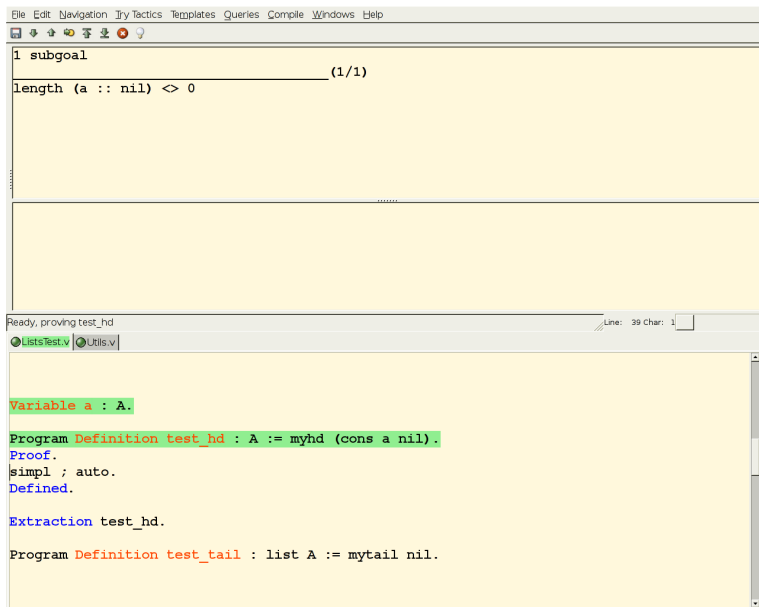
```
Extraction myhd.
Extraction Inline proj1_sig.

Program Definition mytail : forall { l : list A | length l < 0 }, list A :=
  fun l =>
    match l with
    | nil => _
    | hd :: tl => tl
  end.

Proof.
destruct l ; simpl ; intro H ; rewrite <- H in n ; intuition.
Defined.

Extraction mytail.
```

PROGRAM : The list example



The screenshot shows a proof assistant interface with a menu bar (File, Edit, Navigation, Try Tactics, Templates, Queries, Compile, Windows, Help) and a toolbar. The main window displays a subgoal:

```
1 subgoal
_____ (1/1)
length (a :: nil) <= 0
```

Below the subgoal, there is a status bar indicating "Ready, proving test_hd" and "Line: 39 Char: 1". The interface also shows a file explorer with "ListsTest.v" and "Utils.v" selected. The main text area contains the following code:

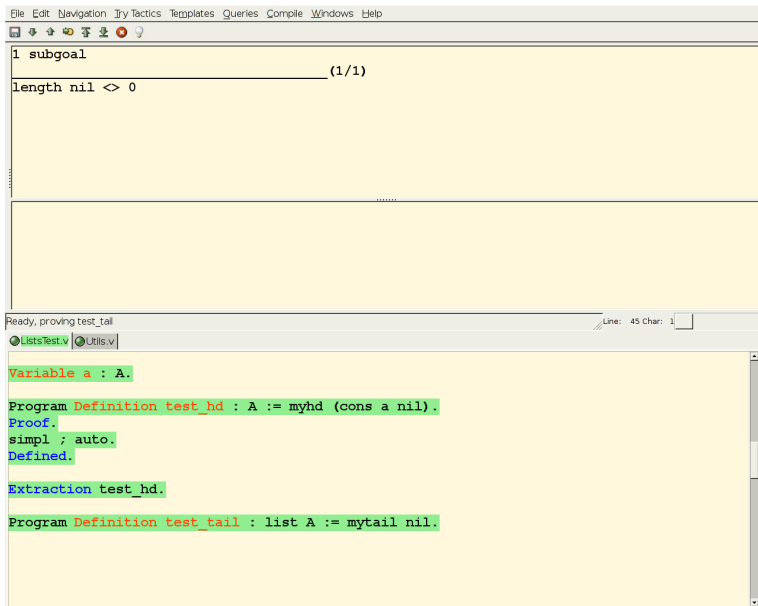
```
Variable a : A.

Program Definition test_hd : A := myhd (cons a nil).
Proof.
  simpl ; auto.
Defined.

Extraction test_hd.

Program Definition test_tail : list A := mytail nil.
```

PROGRAM : The list example



The image shows a screenshot of a proof assistant interface, likely Coq. The top window displays a subgoal:

```
1 subgoal
_____ (1/1)
length nil <> 0
```

The bottom window shows the code editor with the following content:

```
Variable a : A.

Program Definition test_hd : A := myhd (cons a nil).
Proof.
simpl ; auto.
Defined.

Extraction test_hd.

Program Definition test_tail : list A := mytail nil.
```

The interface includes a menu bar (File, Edit, Navigation, Try Tactics, Templates, Queries, Compile, Windows, Help), a toolbar with icons, and a status bar at the bottom right indicating "Line: 45 Char: 1".

PROGRAM : The list example

```
File Edit Navigation Try Tactics Templates Queries Compile Windows Help
[Icons]
2 subgoals
append : forall l l' : list A, (r : list A | length r = length l + length l')
l : list A
l' : list A
Heq1 : nil = l
_____ (1/2)
length l' = length l + length l'
.....
_____ (2/2)
length (hd0 :: (`append t10 l')) = length l + length l'
.....

Ready, proving append_and_proof Line: 57 Char: 46
[ListsTest.v] [Utils.v]

Program Fixpoint append (l : list A) (l' : list A) { struct l } :
{ r : list A | length r = length l + length l' } :=
match l with
| nil => l'
| hd :: tl => hd :: (append tl l')
end.
subst ; auto.
simpl ; rewrite (subset_simpl (append t10 l')).
simpl ; subst.
simpl ; auto.
Defined.

Extraction append.
```

PROGRAM : The list example

The screenshot shows a theorem prover interface with a menu bar (File, Edit, Navigation, Try Tactics, Templates, Queries, Compile, Windows, Help) and a toolbar. The main editor displays the following code:

```
(** val append : a list -> a list -> a list **)
let rec append l l' =
  match l with
  | Nil -> l'
  | Cons (hd, tl) -> Cons (hd, (append tl l'))
```

The status bar indicates "Ready" and "Line: 64 Char: 1". Below the editor, there are tabs for "ListsTest.v" and "Utils.v". The output window shows the following execution trace:

```
Program Fixpoint append (l : list A) (l' : list A) { struct l } :
{ r : list A | length r = length l + length l' } :=
match l with
| nil => l'
| hd :: tl => hd :: (append tl l')
end.
subst ; auto.
simpl ; rewrite (subset_simpl (append t10 l')).
simpl ; subst.
simpl ; auto.
Defined.

Extraction append.
```

PROGRAM : The list example

The screenshot shows a proof assistant interface with a menu bar (File, Edit, Navigation, Try Tactics, Templates, Queries, Compile, Windows, Help) and a toolbar. The main editor area is divided into two panes. The top pane shows a subgoal: "1 subgoal" followed by a horizontal line and "(1/1)", and the expression "forall l : list A, l = (`append l nil)". The bottom pane shows a proof script with two lemmas. The first lemma is "Program Lemma append_app' : forall l : list A, l = append nil l." followed by "Proof.", "simpl ; auto.", and "Qed.". The second lemma is "Program Lemma append_app : forall l : list A, l = append l nil." followed by "Proof.", "intros.", "induction l ; simpl ; auto.", "simpl in IH1.", "rewrite <- IH1.", "reflexivity.", and "Qed.". The status bar at the bottom indicates "Ready, proving append_app" and "Line: 71 Char: 1".

```
File Edit Navigation Try Tactics Templates Queries Compile Windows Help
1 subgoal
_____ (1/1)
forall l : list A, l = (`append l nil)

Ready, proving append_app Line: 71 Char: 1
ListsTest.v Util.v

Program Lemma append_app' : forall l : list A, l = append nil l.
Proof.
simpl ; auto.
Qed.

Program Lemma append_app : forall l : list A, l = append l nil.
Proof.
intros.
induction l ; simpl ; auto.
simpl in IH1.
rewrite <- IH1.
reflexivity.
Qed.
```

Conclusion

Our contribution

A more **flexible** programming language, (almost) **conservative** over CIC, **integrated** with the existing environment and a formal **justification** of “*Predicate subtyping*”.

Conclusion

Our contribution

A more **flexible** programming language, (almost) **conservative** over CIC, **integrated** with the existing environment and a formal **justification** of “*Predicate subtyping*”.

Future work

- Application to more constructs ((co-)inductive types) and commands.
- Improvements of Coq (existential variables, type inference, proof irrelevance).
- Complete and useful interpretation of ML languages.

Addendum : some practical enhancements

- Handling of dependent existential variables (WIP).

Addendum : some practical enhancements

- Handling of dependent existential variables (WIP).
- Pattern-matching and equalities.

match v **return** T **with** $p_1 \Rightarrow t_1 \cdots p_n \Rightarrow t_n$

(**match** μv **as** t' **return** $t' = \mu(v) \rightarrow T$) **with**
 $p_1 \Rightarrow$ **fun** h $\Rightarrow t_1$
 \vdots
 $p_n \Rightarrow$ **fun** h $\Rightarrow t_n$)
(**refl_equal** $\mu(v)$)

Addendum : some practical enhancements

- Handling of dependent existential variables (WIP).
- Pattern-matching and equalities.
- Well-founded recursion.

Program **Fixpoint** $f (a : nat) \{ \mathbf{wf} \text{ lt } a \} : nat := t$

Addendum : some practical enhancements

- Handling of dependent existential variables (WIP).
- Pattern-matching and equalities.
- Well-founded recursion.

Program **Fixpoint** $f (a : \text{nat}) \{ \mathbf{wf} \text{ lt } a \} : \text{nat} := t$

$a : \text{nat}$

$f : \{x : \text{nat} \mid x < a\} \rightarrow \text{nat}$

$t : \text{nat}$