Programing in Coq

Matthieu Sozeau
under the direction of Christine Paulin-Mohring

LRI, Univ. Paris-Sud - Démons Team & INRIA Futurs - ProVal Project

Démons-ProVal-LogiCal seminar
9 march 2007

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)
```

The Big Picture

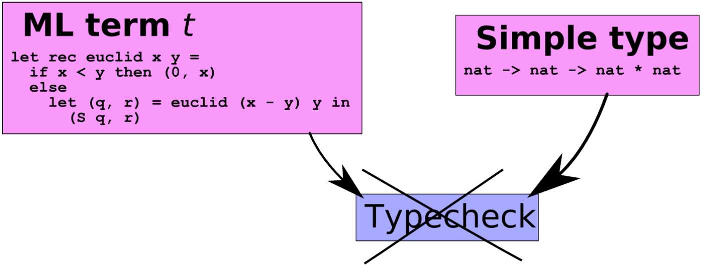
ML term ***t***

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

Simple type

```
nat -> nat -> nat * nat
```

Typecheck


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)
Simple type
nat -> nat -> nat * nat
~~Typecheck~~

The Big Picture

ML term t

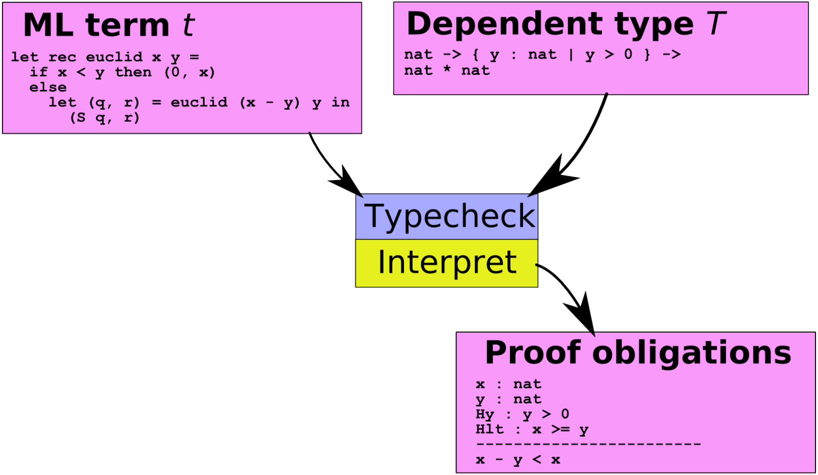
```
let rec euclid x y =
  if x < y then (0, x)
  else
    let (q, r) = euclid (x - y) y in
      (S q, r)
```

Dependent type T

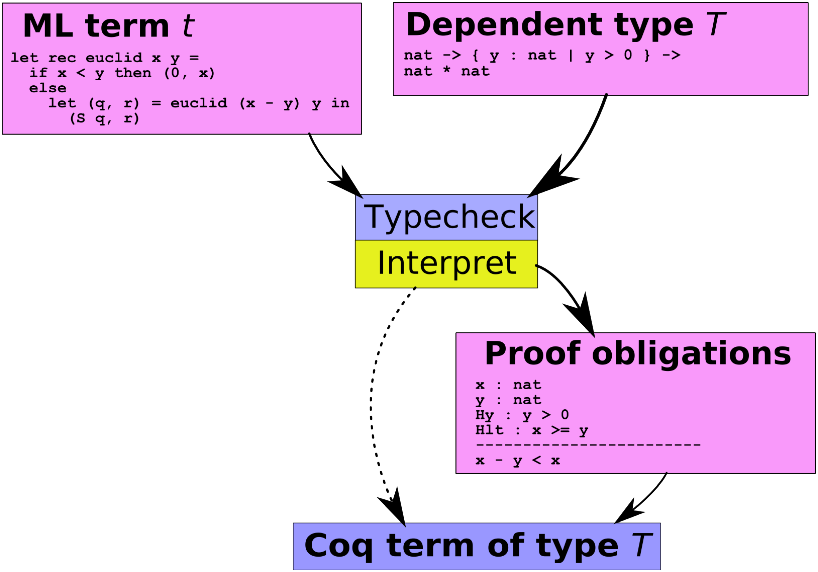
```
nat -> { y : nat | y > 0 } ->
nat * nat
```

Typecheck

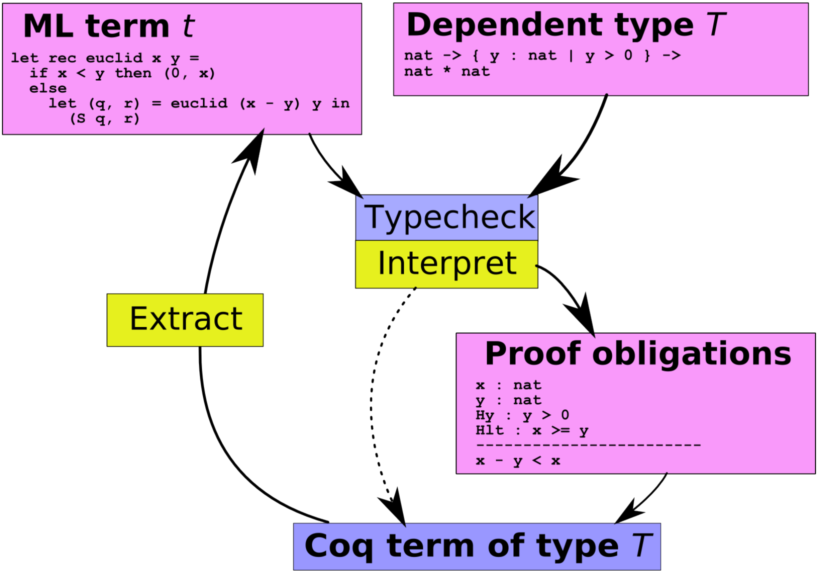
The Big Picture



The Big Picture



The Big Picture



The Big Picture

Inductive $diveucl\ a\ b$: Set :=
  $divex : \forall\ q\ r,\ b > r \rightarrow a = q \times b + r \rightarrow diveucl\ a\ b.$

Lemma $eucl\_dev : \forall\ n,\ n > 0 \rightarrow \forall\ m{:}nat,\ diveucl\ m\ n.$

Proof.

  $intros\ b\ H\ a;\ pattern\ a$ in $\vdash \times;\ apply\ gt\_wf\_rec;\ intros\ n$ $H0.$

  $elim\ (le\_gt\_dec\ b\ n).$

  $intro\ lebn.$

  $elim\ (H0\ (n - b));\ auto$ with $arith.$

  $intros\ q\ r\ g\ e.$

  $apply\ divex$ with $(S\ q)\ r;\ simpl$ in $\vdash \times;\ auto$ with $arith.$

  $elim\ plus\_assoc.$

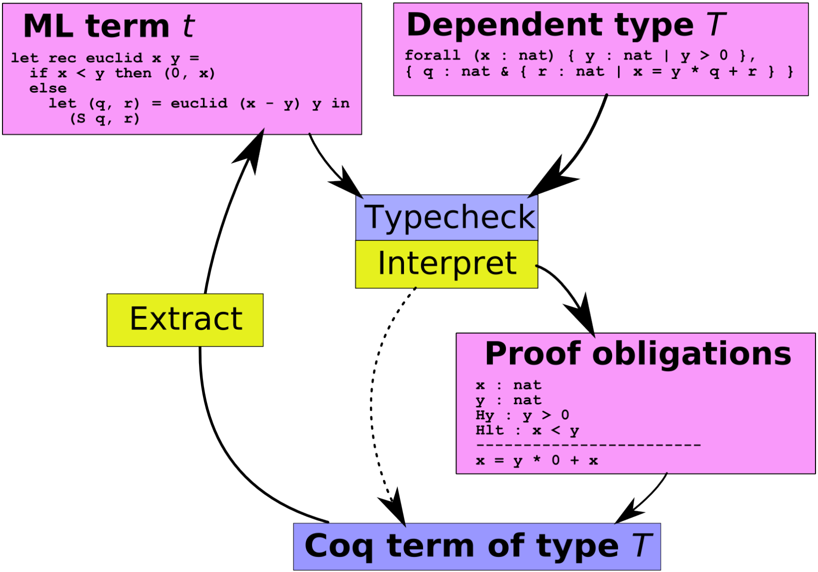
  $elim\ e;\ auto$ with $arith.$

  $intros\ gtbn.$

  $apply\ divex$ with $0\ n;\ simpl$ in $\vdash \times;\ auto$ with $arith.$

Qed.

The Big Picture



1. The idea
   - A simple idea
   - From PVS to Coq
2. Theoretical development
   - Russell
   - Interpretation in Coq
   - Inductive types
3. Program
   - Architecture
   - Hello world
   - Extensions
   - Finger Trees
4. Conclusion

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 ;
- Links 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 ;
- Links object and property.

Adapting the idea

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

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 ;
- Links object and property.

Adapting the idea

$$\frac{t : T \qquad p : P[t/x]}{(t, p) : \{\ x : T \mid P\ \}} \qquad \frac{t : \{\ x : T \mid P\ \}}{\texttt{proj}\ t : T}$$

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 ;

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

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

…to Subset coercions

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

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

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

. . . to Subset coercions

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

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

$$\frac{\Gamma \vdash t : T \qquad \Gamma, x : T \vdash P : \texttt{Prop} \qquad \Gamma \vdash p : P[t/x]}{\Gamma \vdash (t, p) : \{ x : T \mid P \}}$$

1. The idea
   - A simple idea
   - From PVS to Coq
2. Theoretical development
   - Russell
   - Interpretation in Coq
   - Inductive types
3. Program
   - Architecture
   - Hello world
   - Extensions
   - Finger Trees
4. Conclusion

Russell syntax

$$x \in \mathcal{V}$$

$$\begin{array}{rcl} s, t, u, v & ::= & x \\ & \mid & \mathtt{Set} \\ & \mid & \mathtt{Prop} \\ & \mid & \mathtt{Type} \end{array}$$

Russell syntax

$$
\begin{array}{lcl}
x & \in & \mathcal{V} \\
\\
s, t, u, v & ::= & x \\
 & | & \mathtt{Set} \\
 & | & \mathtt{Prop} \\
 & | & \mathtt{Type} \\
 & | & \lambda x : s.t \\
 & | & s\ t \\
 & | & \Pi x : s.t
\end{array}
$$

Russell syntax

$$
\begin{array}{lcl}
x & \in & \mathcal{V} \\
\\
s, t, u, v & ::= & x \\
& | & \mathtt{Set} \\
& | & \mathtt{Prop} \\
& | & \mathtt{Type} \\
& | & \lambda x : s.t \\
& | & s\ t \\
& | & \Pi x : s.t \\
& | & (u, v)_{\Sigma x:s.t} \\
& | & \pi_1\ s \mid \pi_2\ s \\
& | & \Sigma x : s.t
\end{array}
$$

Russell syntax

$$
\begin{array}{lcl}
x & \in & \mathcal{V} \\
\\
s, t, u, v & ::= & x \\
& \mid & \mathtt{Set} \\
& \mid & \mathtt{Prop} \\
& \mid & \mathtt{Type} \\
& \mid & \lambda x : s.t \\
& \mid & s\ t \\
& \mid & \Pi x : s.t \\
& \mid & (u, v)_{\Sigma x : s.t} \\
& \mid & \pi_1\ s \mid \pi_2\ s \\
& \mid & \Sigma x : s.t \\
& \mid & \{\ x : s \mid t\ \}
\end{array}
$$

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

Calculus of Constructions with

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

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

Calculus of Constructions with

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

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

Calculus of Constructions with

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

$$\frac{\Gamma \vdash S \rhd T : s \qquad \Gamma \vdash T \rhd U : s}{\Gamma \vdash S \rhd U : s}$$

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

Calculus of Constructions with

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

$$\frac{\Gamma \vdash S \rhd T : s \qquad \Gamma \vdash T \rhd U : s}{\Gamma \vdash S \rhd U : s}$$

$$\frac{\Gamma \vdash U \rhd V : \mathtt{Set} \qquad \Gamma, x : U \vdash P : \mathtt{Prop}}{\Gamma \vdash \{ x : U \mid P \} \rhd V : \mathtt{Set}}$$

$$\frac{\Gamma \vdash U \rhd V : \mathtt{Set} \qquad \color{red}{\Gamma, x : V \vdash P : \mathtt{Prop}}}{\Gamma \vdash U \rhd \{ x : V \mid P \} : \mathtt{Set}}$$

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

Calculus of Constructions with

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

$$\frac{\Gamma \vdash S \rhd T : s \qquad \Gamma \vdash T \rhd U : s}{\Gamma \vdash S \rhd U : s}$$

$$\frac{\Gamma \vdash U \rhd V : \mathsf{Set} \qquad \Gamma, x : U \vdash P : \mathsf{Prop}}{\Gamma \vdash \{\, x : U \mid P \,\} \rhd V : \mathsf{Set}}$$

$$\frac{\Gamma \vdash U \rhd V : \mathsf{Set} \qquad \color{red}{\Gamma, x : V \vdash P : \mathsf{Prop}}}{\Gamma \vdash U \rhd \{\, x : V \mid P \,\} : \mathsf{Set}}$$

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

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

Calculus of Constructions with

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

$$\frac{\Gamma \vdash S \rhd T : s \qquad \Gamma \vdash T \rhd U : s}{\Gamma \vdash S \rhd U : s}$$

$$\frac{\Gamma \vdash U \rhd V : \mathtt{Set} \qquad \Gamma, x : U \vdash P : \mathtt{Prop}}{\Gamma \vdash \{\, x : U \mid P \,\} \rhd V : \mathtt{Set}}$$

$$\frac{\Gamma \vdash U \rhd V : \mathtt{Set} \qquad \Gamma, x : V \vdash P : \mathtt{Prop}}{\Gamma \vdash U \rhd \{\, x : V \mid P \,\} : \mathtt{Set}}$$

$$\frac{\dfrac{\Gamma \vdash 0 : \mathbb{N} \qquad \Gamma \vdash \mathbb{N} \rhd \{\, x : \mathbb{N} \mid x \neq 0 \,\} : \mathtt{Set}}{\Gamma \vdash 0 : \{\, x : \mathbb{N} \mid x \neq 0 \,\}}}{\Gamma \vdash ? : 0 \neq 0}$$

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

Calculus of Constructions with

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

$$\frac{\Gamma \vdash S \rhd T : s \qquad \Gamma \vdash T \rhd U : s}{\Gamma \vdash S \rhd U : s}$$

$$\frac{\Gamma \vdash U \rhd V : \mathtt{Set} \qquad \Gamma, x : U \vdash P : \mathtt{Prop}}{\Gamma \vdash \{\, x : U \mid P \,\} \rhd V : \mathtt{Set}}$$

$$\frac{\Gamma \vdash U \rhd V : \mathtt{Set} \qquad \color{red}{\Gamma, x : V \vdash P : \mathtt{Prop}}}{\Gamma \vdash U \rhd \{\, x : V \mid P \,\} : \mathtt{Set}}$$

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

Results

Theorem (Decidability of type checking and type inference)

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

$$\textsc{App}\ \frac{\Gamma \vdash f : T \quad \Gamma \vdash T \rhd \Pi x : A.B : s \qquad \Gamma \vdash e : E \qquad \Gamma \vdash E \rhd A : s'}{\Gamma \vdash (f\ e) : B[e/x]}$$

Results

Theorem (Decidability of type checking and type inference)

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

Lemma (Elimination of transitivity)

If $T \rhd U \wedge U \rhd V$ *then* $T \rhd V$.

Theorem (Subject Reduction)

If $\Gamma \vdash t : T$ *and* $t \rightarrow_{\beta\pi} t'$ *then* $\Gamma \vdash t' : T$.

$$\textsc{App}\ \frac{\Gamma \vdash f : T \quad \Gamma \vdash T \rhd \Pi x : A.B : s \qquad \Gamma \vdash e : E \qquad \Gamma \vdash E \rhd A : s'}{\Gamma \vdash (f\ e) : B[e/x]}$$

1. The idea
   - A simple idea
   - From PVS to Coq
2. Theoretical development
   - Russell
   - Interpretation in Coq
   - Inductive types
3. Program
   - Architecture
   - Hello world
   - Extensions
   - Finger Trees
4. Conclusion

From Russell to Coq

The target system : Cic with metavariables

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

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

$$\frac{\Gamma \vdash_? P : \mathtt{Prop}}{\Gamma \vdash_? ?_P : P}$$

We build an interpretation $[\![\_]\!]_\Gamma$ from Russell to $\text{Cic}_?$ terms.

From Russell to Coq

The target system : Cic with metavariables

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

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

$$\frac{\Gamma \vdash_? P : \mathtt{Prop}}{\Gamma \vdash_? ?_P : P}$$

We build an interpretation $[\![\_]\!]_\Gamma$ from Russell to $\text{Cic}_?$ terms.

Our goal

If $\Gamma \vdash t : T$ then $[\![\Gamma]\!] \vdash_? [\![t]\!]_\Gamma : [\![T]\!]_\Gamma$.

Deriving explicit coercions

Interpretation of coercions

If $\Gamma \vdash T \rhd U : s$ then $\Gamma \vdash_? c[\bullet] : T \rhd U$ which implies
$$[\![\Gamma]\!], x : [\![T]\!]_\Gamma \vdash_? c[x] : [\![U]\!]_\Gamma.$$

Deriving explicit coercions

Interpretation of coercions

If $\Gamma \vdash T \rhd U : s$ then $\Gamma \vdash_? c[\bullet] : T \rhd U$ which implies $[\![\Gamma]\!], x : [\![T]\!]_\Gamma \vdash_? c[x] : [\![U]\!]_\Gamma$.

Definition

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

Deriving explicit coercions

Interpretation of coercions

If $\Gamma \vdash T \rhd U : s$ then $\Gamma \vdash_? c[\bullet] : T \rhd U$ which implies $[\![\Gamma]\!], x : [\![T]\!]_\Gamma \vdash_? c[x] : [\![U]\!]_\Gamma$.

Definition

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

Deriving explicit coercions

Interpretation of coercions

If $\Gamma \vdash T \rhd U : s$ then $\Gamma \vdash_? c[\bullet] : T \rhd U$ which implies $[\![\Gamma]\!], x : [\![T]\!]_\Gamma \vdash_? c[x] : [\![U]\!]_\Gamma$.

Definition

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

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

Deriving explicit coercions

Interpretation of coercions

If $\Gamma \vdash T \rhd U : s$ then $\Gamma \vdash_? c[\bullet] : T \rhd U$ which implies $[\![\Gamma]\!], x : [\![T]\!]_\Gamma \vdash_? c[x] : [\![U]\!]_\Gamma$.

Definition

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

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

Deriving explicit coercions

Interpretation of coercions

If $\Gamma \vdash T \rhd U : s$ then $\Gamma \vdash_? c[\bullet] : T \rhd U$ which implies $[\![\Gamma]\!], x : [\![T]\!]_\Gamma \vdash_? c[x] : [\![U]\!]_\Gamma$.

Definition

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

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

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

Deriving explicit coercions

Interpretation of coercions

If $\Gamma \vdash T \rhd U : s$ then $\Gamma \vdash_? c[\bullet] : T \rhd U$ which implies $[\![\Gamma]\!], x : [\![T]\!]_\Gamma \vdash_? c[x] : [\![U]\!]_\Gamma$.

Definition

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

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

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

Deriving explicit coercions

Interpretation of coercions

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

$$[\![\Gamma]\!], x : [\![T]\!]_\Gamma \vdash_? c[x] : [\![U]\!]_\Gamma.$$

Definition

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

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

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

Example

$$\frac{\Gamma \vdash_? 0 : \mathbb{N} \qquad \Gamma \vdash_? \mathit{elt}\ \_\ \_\ \bullet\ ?_{(x \neq 0)[\bullet/x]} : \mathbb{N} \rhd \{\ x : \mathbb{N} \mid x \neq 0\ \}}{\Gamma \vdash_? \mathit{elt}\ \_\ \_\ 0\ ?_{0 \neq 0} : \{\ x : \mathbb{N} \mid x \neq 0\ \}}$$

Interpretation of terms

Example (Application)

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

$$\begin{array}{rcl} [\![f\ u]\!]_\Gamma & = & \textbf{\textit{let}}\ \pi = \textbf{\textit{coerce}}_\Gamma\ T\ (\Pi x : V.W)\ \textbf{\textit{in}} \\ & & \textbf{\textit{let}}\ c = \textbf{\textit{coerce}}_\Gamma\ U\ V\ \textbf{\textit{in}} \\ & & (\pi[[\![f]\!]_\Gamma])\ (c[[\![u]\!]_\Gamma]) \end{array}$$

Theorem (Soundness)

If $\Gamma \vdash t : T$ *then* $[\![\Gamma]\!] \vdash_? [\![t]\!]_\Gamma : [\![T]\!]_\Gamma$.

Theoretical matters . . .

$\vdash_?$'s equational theory:

$$
\begin{array}{llcll}
(\beta) & (\lambda x : X.e)\ v & \equiv & e[v/x] & \\
(\pi_i) & \pi_i\ (e_1, e_2)_T & \equiv & e_i & \\
(\sigma_i) & \sigma_i\ (\mathsf{elt}\ E\ P\ e_1\ e_2) & \equiv & e_i & \\
(\eta) & (\lambda x : X.e\ x) & \equiv & e & \text{if } x \notin FV(e) \\
(SP) & \mathsf{elt}\ E\ P\ (\sigma_1\ e)\ (\sigma_2\ e) & \equiv & e &
\end{array}
$$

Theoretical matters . . .

$\vdash_?$'s equational theory:

(β)	$(\lambda x : X.e)\ v$	$\equiv$	$e[v/x]$	
(π_i)	$\pi_i\ (e_1, e_2)_T$	$\equiv$	e_i	
(σ_i)	$\sigma_i\ (\mathsf{elt}\ E\ P\ e_1\ e_2)$	$\equiv$	e_i	
(η)	$(\lambda x : X.e\ x)$	$\equiv$	e	if $x \notin FV(e)$
(SP)	$\mathsf{elt}\ E\ P\ (\sigma_1\ e)\ (\sigma_2\ e)$	$\equiv$	e	
(σ)	$\mathsf{elt}\ E\ P\ t\ p$	$\equiv$	$\mathsf{elt}\ E\ P\ t'\ p'$	if $t \equiv t'$

$\Rightarrow$ Proof Irrelevance

Theoretical matters ...

$\vdash_?$'s equational theory:

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

$\Rightarrow$ Proof Irrelevance

... have practical effects

Difficulty to reason on code: $f(\mathsf{elt}\ T\ P\ x\ p_1) \not\equiv f(\mathsf{elt}\ T\ P\ x\ p_2)$.

Inductive types

A natural extension: Reindexing

Let $I\ \overrightarrow{x}$ be an inductive type with *real* arguments $\overrightarrow{x}$. We can coerce any object of this type to an object of $I\ \overrightarrow{y}$ provided $\overrightarrow{x} = \overrightarrow{y}$.

Inductive types

A natural extension: Reindexing

Let $I\ \overrightarrow{x}$ be an inductive type with *real* arguments $\overrightarrow{x}$. We can coerce any object of this type to an object of $I\ \overrightarrow{y}$ provided $\overrightarrow{x} = \overrightarrow{y}$.

$$\frac{\Gamma \vdash v : \mathsf{vec}\ (m+0) \qquad \Gamma \vdash \mathsf{vec}\ (m+0) \rhd \mathsf{vec}\ m : \mathsf{Set}}{\Gamma \vdash v : \mathsf{vec}\ m}$$

Inductive types

A natural extension: Reindexing

Let $I\ \overrightarrow{x}$ be an inductive type with *real* arguments $\overrightarrow{x}$. We can coerce any object of this type to an object of $I\ \overrightarrow{y}$ provided $\overrightarrow{x} = \overrightarrow{y}$.

$$\frac{\Gamma \vdash v : \mathsf{vec}\ (m+0) \qquad \Gamma \vdash \mathsf{vec}\ (m+0) \rhd \mathsf{vec}\ m : \mathsf{Set}}{\Gamma \vdash v : \mathsf{vec}\ m}$$

$$\frac{\dots \qquad \dfrac{\Gamma \vdash_? ?_{m+0=m} : m+0=m}{\Gamma \vdash_? \mathsf{eq\_rec}\ \mathbb{N}\ (m+0)\ \mathsf{vec}\ \bullet\ m\ ?_{m+0=m} : \mathsf{vec}\ (m+0) \rhd \mathsf{vec}\ m}}{\Gamma \vdash_? \mathsf{eq\_rec}\ \mathbb{N}\ (m+0)\ \mathsf{vec}\ \ v\ m\ ?_{m+0=m} : \mathsf{vec}\ m}$$

$$\mathsf{eq\_rec} : \forall (A : \mathsf{Set})(x : A)(P : A \to \mathsf{Set}), P\ x \to \forall y, x = y \to P\ y$$

1. The idea
   - A simple idea
   - From PVS to COQ
2. Theoretical development
   - RUSSELL
   - Interpretation in COQ
   - Inductive types
3. PROGRAM
   - Architecture
   - Hello world
   - Extensions
   - Finger Trees
4. Conclusion

The Program vernacular

Architecture

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

The Program vernacular

Architecture

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

1. Use the Coq parser.

`Program Definition f : T := t .`

The PROGRAM vernacular

Architecture

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

1. Use the COQ parser.
2. Typecheck $\Gamma \vdash t : T$ and generate $[\![\Gamma]\!] \vdash_? [\![t]\!]_\Gamma : [\![T]\!]_\Gamma$;

$$\texttt{Program Definition f} : [\![T]\!]_\Gamma := [\![t]\!]_\Gamma \quad .$$

The Program vernacular

Architecture

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

1. Use the Coq parser.
2. Typecheck $\Gamma \vdash t : T$ and generate $[\![\Gamma]\!] \vdash_? [\![t]\!]_\Gamma : [\![T]\!]_\Gamma$;
3. Interactive proving of obligations ;

`Program Definition f` $: [\![T]\!]_\Gamma := [\![t]\!]_\Gamma +$ obligations.

The PROGRAM vernacular

Architecture

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

1. Use the COQ parser.
2. Typecheck $\Gamma \vdash t : T$ and generate $[\![\Gamma]\!] \vdash_{?} [\![t]\!]_\Gamma : [\![T]\!]_\Gamma$;
3. Interactive proving of obligations ;
4. Final definition.

$$\texttt{Definition } f \; : [\![T]\!]_\Gamma := [\![t]\!]_\Gamma + \text{ obligations.}$$

The PROGRAM vernacular

Architecture

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

1. Use the COQ parser.
2. Typecheck $\Gamma \vdash t : T$ and generate $[\![\Gamma]\!] \vdash_? [\![t]\!]_\Gamma : [\![T]\!]_\Gamma$;
3. Interactive proving of obligations ;
4. Final definition.

$$\texttt{Definition f} : [\![T]\!]_\Gamma := [\![t]\!]_\Gamma + \text{ obligations.}$$

Remark (Restriction)

We assume $\Gamma \vdash_{CCI} [\![T]\!]_\Gamma : s$.

Hello world: Euclidian division

DEMO

1. The idea
   - A simple idea
   - From PVS to Coq
2. Theoretical development
   - Russell
   - Interpretation in Coq
   - Inductive types
3. Program
   - Architecture
   - Hello world
   - Extensions
   - Finger Trees
4. Conclusion

Pattern-matching revisited

From patterns to equations:

$$
\begin{array}{llll}
\textbf{match} & e & \textbf{return} & T\ \textbf{with} \\
\mid p_1 & & \Rightarrow & t_1 \\
 & & \vdots & \\
\mid p_n & & \Rightarrow & t_n \\
\textbf{end} & & &
\end{array}
$$

where $\nu(e)$ coerces e to an inductive object.

Pattern-matching revisited

From patterns to equations:

$$
\begin{array}{lcl}
\textbf{match } \nu(e) \textbf{ as } t & \textbf{return} & t = \nu(e) \to T \textbf{ with} \\
\mid p_1 & \Rightarrow & \textbf{fun } H : p_1 = \nu(e) \Rightarrow t_1 \\
 & \vdots & \\
\mid p_n & \Rightarrow & \textbf{fun } H : p_n = \nu(e) \Rightarrow t_n \\
\textbf{end} & & \\
(\textbf{refl\_equal } \nu(e)) & &
\end{array}
$$

where $\nu(e)$ coerces e to an inductive object.

Pattern-matching revisited

From patterns to equations:

$$
\begin{array}{lcl}
\textbf{match } \nu(e) \textbf{ as } t & \textbf{return} & t = \nu(e) \to T \textbf{ with} \\
\mid p_1 & \Rightarrow & \textbf{fun } H : p_1 = \nu(e) \Rightarrow t_1 \\
 & \vdots & \\
\mid p_n & \Rightarrow & \textbf{fun } H : p_n = \nu(e) \Rightarrow t_n \\
\textbf{end} & & \\
(\textbf{refl\_equal } \nu(e)) & &
\end{array}
$$

where $\nu(e)$ coerces e to an inductive object.

Further refinements

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

Sugar

Obligations

Unresolved implicits (_) are turned into obligations.

Bang

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

Example

$\textit{match}\ 0\ \textit{with}\ 0 \Rightarrow 0 \mid n \Rightarrow !\ \textit{end}$

Recursion

Support for well-founded recursion (and measures).

$$\texttt{Program Fixpoint}\ f\ (a : \mathbb{N})\ \{\mathbf{wf}\ \mathrm{lt}\ a\} : \mathbb{N} := b$$

Recursion

Support for well-founded recursion (and measures).

$$\texttt{Program Fixpoint } f\ (a : \mathbb{N})\ \{\mathbf{wf}\ \mathrm{lt}\ a\} : \mathbb{N} := b$$

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

1. The idea
   - A simple idea
   - From PVS to COQ
2. Theoretical development
   - RUSSELL
   - Interpretation in COQ
   - Inductive types
3. PROGRAM
   - Architecture
   - Hello world
   - Extensions
   - Finger Trees
4. Conclusion

Digits

```
Section Digit.
  Variable A : Type.

  Inductive digit : Type :=
  | One : A → digit
  | Two : A → A → digit
  | Three : A → A → A → digit
  | Four : A → A → A → A → digit.

  Definition full x := match x with Four _ _ _ _ ⇒ True | _ ⇒ False end.
```

Digits

```
Section Digit.
  Variable A : Type.

  Inductive digit : Type :=
  | One : A → digit
  | Two : A → A → digit
  | Three : A → A → A → digit
  | Four : A → A → A → A → digit.

  Definition full x := match x with Four _ _ _ _ ⇒ True | _ ⇒ False end.

  Program Definition add_digit_left (a : A) (d : digit | ¬ full d) : digit :=
    match d with
      | One x ⇒ Two a x
      | Two x y ⇒ Three a x y
      | Three x y z ⇒ Four a x y z
      | Four _ _ _ _ ⇒ !
    end.
  Next Obligation.
    intros ; simpl in n ; auto.
  Qed.
```

Monoids & Nodes

```
Variable v : Type.
Variable mono : monoid v.

Notation "'ε'" := mempty mono.
Infix "·" := mappend mono (right associativity, at level 20).

Program Definition listMonoid (A : Type) : monoid (list A) :=
  mkMonoid nil app _ _ _.
```

Monoids & Nodes

```
Variable v : Type.
Variable mono : monoid v.

Notation "'ε'" := mempty mono.
Infix "·" := mappend mono (right associativity, at level 20).

Program Definition listMonoid (A : Type) : monoid (list A) :=
  mkMonoid nil app _ _ _.

Section Nodes.
  Variable A : Type.
  Variable measure : A → v.
  Notation "'‖' x '‖'" := (measure x).

  Inductive node : Type :=
  | Node2 : ∀ x y, { s : v | s = ‖ x ‖ · ‖ y ‖ } → node
  | Node3 : ∀ x y z, { s : v | s = ‖ x ‖ · ‖ y ‖ · ‖ z ‖ } → node.
```

Monoids & Nodes

```
Variable v : Type.
Variable mono : monoid v.

Notation "'ε'" := mempty mono.
Infix "·" := mappend mono (right associativity, at level 20).

Program Definition listMonoid (A : Type) : monoid (list A) :=
  mkMonoid nil app _ _ _.

Section Nodes.
  Variable A : Type.
  Variable measure : A → v.
  Notation "'‖' x '‖'" := (measure x).

  Inductive node : Type :=
  | Node2 : ∀ x y, { s : v | s = ‖ x ‖ · ‖ y ‖ } → node
  | Node3 : ∀ x y z, { s : v | s = ‖ x ‖ · ‖ y ‖ · ‖ z ‖ } → node.

  Program Definition node2 (x y : A) : node :=
    Node2 x y (‖ x ‖ · ‖ y ‖).

  Program Definition node_measure (n : node) : v :=
   match n with | Node2 _ _ s ⇒ s | Node3 _ _ _ s ⇒ s end.
```

Dependent Finger Trees

```
Inductive fingertree (A : Type) (measure : A → v) : v → Type :=
| Empty : fingertree measure ε
| Single : ∀ x : A, fingertree measure (measure x)
| Deep : ∀ (l : digit A) (m : v),
  @fingertree (node measure) nodeMeasure m → ∀ r : digit A,
  fingertree measure (digit_measure measure l · m · digit_measure measure r).
```

Dependent Finger Trees

```
Inductive fingertree (A : Type) (measure : A → v) : v → Type :=
| Empty : fingertree measure ε
| Single : ∀ x : A, fingertree measure (measure x)
| Deep : ∀ (l : digit A) (m : v),
  @fingertree (node measure) nodeMeasure m → ∀ r : digit A,
  fingertree measure (digit_measure measure l · m · digit_measure measure r).

  Program Fixpoint add_left A (measure : A → v) (a : A) (s : v)
    (t : fingertree measure s) {struct t} :
    fingertree measure (measure a · s) :=
    match t with
      | Empty ⇒ Single a
      | Single b ⇒ Deep (One a) Empty (One b)
      | Deep pr st' t' sf ⇒
        match pr with
          | Four b c d e ⇒ let sub := add_left (node3 measure c d e) t' in
            Deep (Two a b) sub sf
          | x ⇒ Deep (add_digit_left a pr) t' sf
        end
    end.
```

Dependent Finger Trees - cont'd

Two hundred lines, one hundred obligations concatenation

Definition $app\ (A : \mathrm{Type})\ (measure : A \rightarrow v)$
$(xs : v)\ (x : fingertree\ measure\ xs)$
$(ys : v)\ (y : fingertree\ measure\ ys)$:
$fingertree\ measure\ (xs \cdot ys)$.

Splitting, views can be defined in a similar way.

A glimpse of Finger Trees in Coq

The development

- Certified implementation of Finger Trees. Certified implementation of sequences built on top of Finger Trees.
- $\sim$ 1200 lines of specification, $\sim$ 1400 of proof, mostly unchanged code.
- Extracts to Haskell and ML (uses dependent records).

A glimpse of Finger Trees in COQ

The development

- Certified implementation of Finger Trees. Certified implementation of sequences built on top of Finger Trees.
- $\sim$ 1200 lines of specification, $\sim$ 1400 of proof, mostly unchanged code.
- Extracts to HASKELL and ML (uses dependent records).

Conclusions

- PROGRAM scales ;
- Need more language technology, e.g: overloading ;
- Subset types arise naturally ;
- Dependent types are a powerfull specification tool ;
- Some difficulties with reasonning and computing.

Conclusion

Our contributions

- A more flexible programming language, (almost) conservative over Cic, integrated with the existing environment and a formal justification of "*Predicate subtyping*".
- A tool to make programming in Coq using the full language possible, which can effectively be used for non-trivial developments.

Future work

Reasonning support through tactics, implementation of proof-irrelevance in Coq's kernel.

The End

http://www.lri.fr/~sozeau/research/russell.en.html

Why eta rules and proof irrelevance ?

Let $A, \mathrm{B} : \mathrm{Type}$, $\mathrm{c} : A \rhd \mathrm{B}$, $\mathrm{d} : \mathrm{B} \rhd A$.

Why eta rules and proof irrelevance ?

Let $A, B : \mathtt{Type}$, $c : A \rhd B$, $d : B \rhd A$.

Let $P : A \rightarrow \mathtt{Prop}$ with $p : \Pi x : A, P\ x$ and $q : \Pi x : B, P\ x$.

Why eta rules and proof irrelevance ?

Let $A, B : \mathsf{Type}$, $c : A \rhd B$, $d : B \rhd A$.

Let $P : A \rightarrow \mathsf{Prop}$ with $p : \Pi x : A, P\ x$ and $q : \Pi x : B, P\ x$.

Consider: $p\ x =_{(P\ x)} q\ x$ in context $\Gamma \triangleq x : A$.

By $[\![\_]\!]_\Gamma$: $[\![P]\!]_\Gamma \triangleq P$, $[\![p]\!]_\Gamma \triangleq p$ and $[\![q]\!]_\Gamma \triangleq q : \Pi x : B, P\ d[x]$,

Why eta rules and proof irrelevance ?

Let $A, B : \mathsf{Type}$, $c : A \rhd B$, $d : B \rhd A$.

Let $P : A \to \mathsf{Prop}$ with $p : \Pi x : A, P\ x$ and $q : \Pi x : B, P\ x$.

Consider: $p\ x =_{(P\ x)} q\ x$ in context $\Gamma \triangleq x : A$.

By $[\![\_]\!]_\Gamma$: $[\![P]\!]_\Gamma \triangleq P$, $[\![p]\!]_\Gamma \triangleq p$ and $[\![q]\!]_\Gamma \triangleq q : \Pi x : B, P\ d[x]$,

hence: $[\![p\ x =_{(P\ x)} q\ x]\!]_\Gamma \triangleq p\ x =_{(P\ x)} q\ c[x]$.

We have $p\ x : P\ x$ but $q\ c[x] : P\ d[c[x]]$, so $x \equiv d[c[x]]$ is necessary. This means eta rules for all constructs are needed.

Why eta rules and proof irrelevance ?

Let $A, B : \mathtt{Type}$, $c : A \rhd B$, $d : B \rhd A$.
Let $P : A \to \mathtt{Prop}$ with $p : \Pi x : A, P\ x$ and $q : \Pi x : B, P\ x$.
Consider: $p\ x =_{(P\ x)} q\ x$ in context $\Gamma \triangleq x : A$.
By $[\![\_]\!]_\Gamma$: $[\![P]\!]_\Gamma \triangleq P$, $[\![p]\!]_\Gamma \triangleq p$ and $[\![q]\!]_\Gamma \triangleq q : \Pi x : B, P\ d[x]$,
hence: $[\![p\ x =_{(P\ x)} q\ x]\!]_\Gamma \triangleq p\ x =_{(P\ x)} q\ c[x]$.
We have $p\ x : P\ x$ but $q\ c[x] : P\ d[c[x]]$, so $x \equiv d[c[x]]$ is necessary. This means eta rules for all constructs are needed.
Now: $c \triangleq \mathrm{elt}\ A\ P\ \ \cdot\ \ ?_{P[\cdot/x]} : A \rhd \{\ x : A \mid P\ \}$ and
$d \triangleq \sigma_1 \cdot : \{\ x : A \mid P\ \} \rhd A$, so $d[c[x]] \triangleq x$ but what about $c[d[x]]$?

Why eta rules and proof irrelevance ?

Let $A, B : \mathtt{Type}$, $c : A \rhd B$, $d : B \rhd A$.

Let $P : A \rightarrow \mathtt{Prop}$ with $p : \Pi x : A, P\ x$ and $q : \Pi x : B, P\ x$.

Consider: $p\ x =_{(P\ x)} q\ x$ in context $\Gamma \triangleq x : A$.

By $[\![\_]\!]_\Gamma$: $[\![P]\!]_\Gamma \triangleq P$, $[\![p]\!]_\Gamma \triangleq p$ and $[\![q]\!]_\Gamma \triangleq q : \Pi x : B, P\ d[x]$,

hence: $[\![p\ x =_{(P\ x)} q\ x]\!]_\Gamma \triangleq p\ x =_{(P\ x)} q\ c[x]$.

We have $p\ x : P\ x$ but $q\ c[x] : P\ d[c[x]]$, so $x \equiv d[c[x]]$ is necessary. This means eta rules for all constructs are needed.

Now: $c \triangleq \mathrm{elt}\ A\ P\ \ \cdot\ \ ?_{P[\cdot/x]} : A \rhd \{\ x : A \mid P\ \}$ and

$d \triangleq \sigma_1 \cdot : \{\ x : A \mid P\ \} \rhd A$, so $d[c[x]] \triangleq x$ but what about $c[d[x]]$?

$c[d[x]] \triangleq \mathrm{elt}\ A\ P\ (\sigma_1\ x)\ ?_{P[\sigma_1 x/x]} \not\equiv x$ except if PI is included in $\equiv$.

Dependent Inductive coercion rule

$$\frac{\Gamma, \overrightarrow{(x_i : X_i \simeq y_i : Y_i)}_0^{j-1} \vdash ?_j : x_j : X_j \simeq y_j : Y_j \quad \forall j \in [0..i]}{\Gamma \vdash c(\overrightarrow{?_i}) : I\ \overrightarrow{x_i} \rhd I\ \overrightarrow{y_i} : s}$$

Splitting nodes

$$
\begin{array}{l}
\texttt{Program Definition } \mathit{splitNode}\ (p : v \rightarrow \mathit{bool})\ (i : v)\ (n : \mathit{node\ measure}) : \\
\{\ s : \mathit{split}\ (\texttt{fun}\ A \Rightarrow \mathit{option}\ (\mathit{digit}\ A))\ A \mid \texttt{let}\ (l, x, r) := s\ \texttt{in} \\
\quad \texttt{let}\ \mathit{ls} := \mathit{option\_digit\_measure}\ \mathit{measure}\ l\ \texttt{in} \\
\quad \texttt{let}\ \mathit{rs} := \mathit{option\_digit\_measure}\ \mathit{measure}\ r\ \texttt{in} \\
\quad \mathit{node\_measure}\ n = \mathit{ls} \cdot \|\ x\ \| \cdot \mathit{rs}\ \wedge \\
\quad \mathit{node\_to\_list}\ n = \mathit{option\_digit\_to\_list}\ l\ +\!\!+\ [\mathrm{x}]\ +\!\!+\ \mathit{option\_digit\_to\_list}\ r\ \wedge \\
\quad (l = \mathit{None} \vee p\ (i \cdot \mathit{ls}) = \mathit{false})\ \wedge \\
\quad (r = \mathit{None} \vee p\ (i \cdot \mathit{ls} \cdot \|\ x\ \|) = \mathit{true})\} := \ldots
\end{array}
$$

Instanciation: sequences

The monoid and measure: executable semantics

Definition $below\ i$:= { x : nat | $x < i$ }.
Definition v := { i : nat & ($below\ i \to A$) }.
Program Definition ε : v := 0 $\prec$ (fun _ $\Rightarrow$!).
Program Definition $append$ (xs ys : v) : v :=
  let (n, fx) := xs in let (m, fy) := ys in
    ($n + m$) $\prec$
      (fun i $\Rightarrow$ if lt_ge_dec i n then fx i else fy (i - n)).
Program Definition $seqMonoid$:= @$mkMonoid$ v ε $append$ _ _ _.
Program Definition $measure$ (x : A) : v := 1 $\prec$ (fun _ $\Rightarrow$ x).

Specialization

Definition seq (x : v) := $fingertree$ $seqMonoid$ $measure$ x.
Program Fixpoint $make$ (i : nat) (v : A) { $struct$ i } : seq ($i \prec$ (fun _ $\Rightarrow$ v)).
Program Definition set (i : nat) (m : $below\ i \to A$) (x : seq ($i \prec m$))
  (j : $below\ i$) ($value$: A)
  : seq ($i \prec$ (fun idx : $below\ i$ $\Rightarrow$ if eq_nat_dec idx j then $value$ else m idx)).
Program Definition get (i : nat) (m : $below\ i \to A$) (x : seq ($i \prec m$))
  (j : $below\ i$) : { $value$: A | $m\ j = value$ }.