Program-ing in Coq

Matthieu Sozeau
under the direction of Christine Paulin-Mohring

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

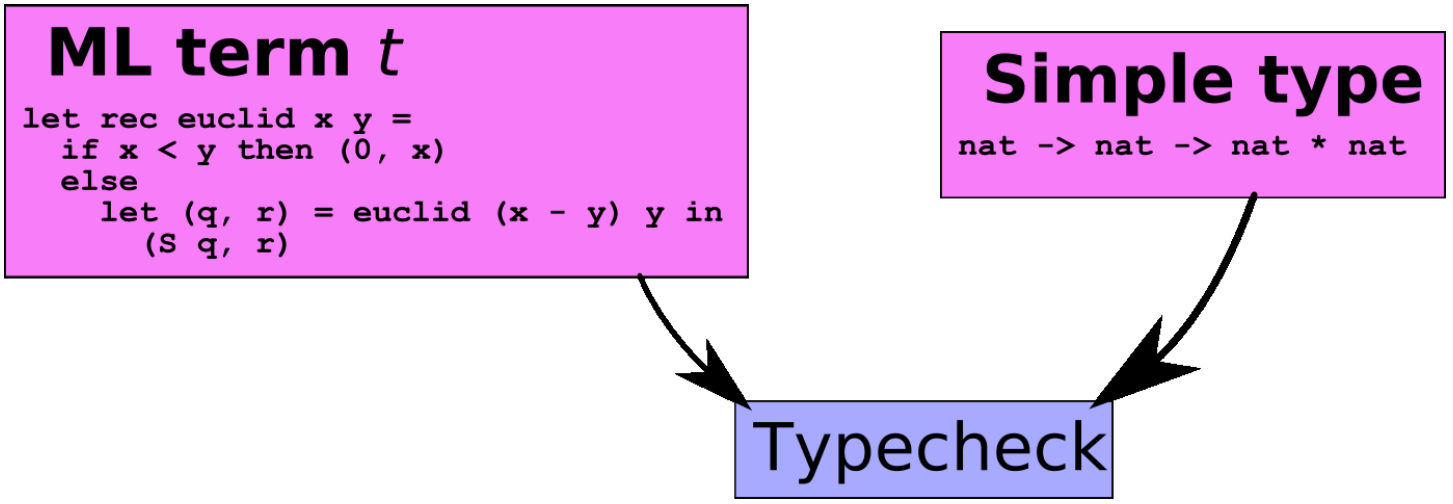
Portland State University Seminar
January 16th 2008
Portland, Oregon

The Big Picture

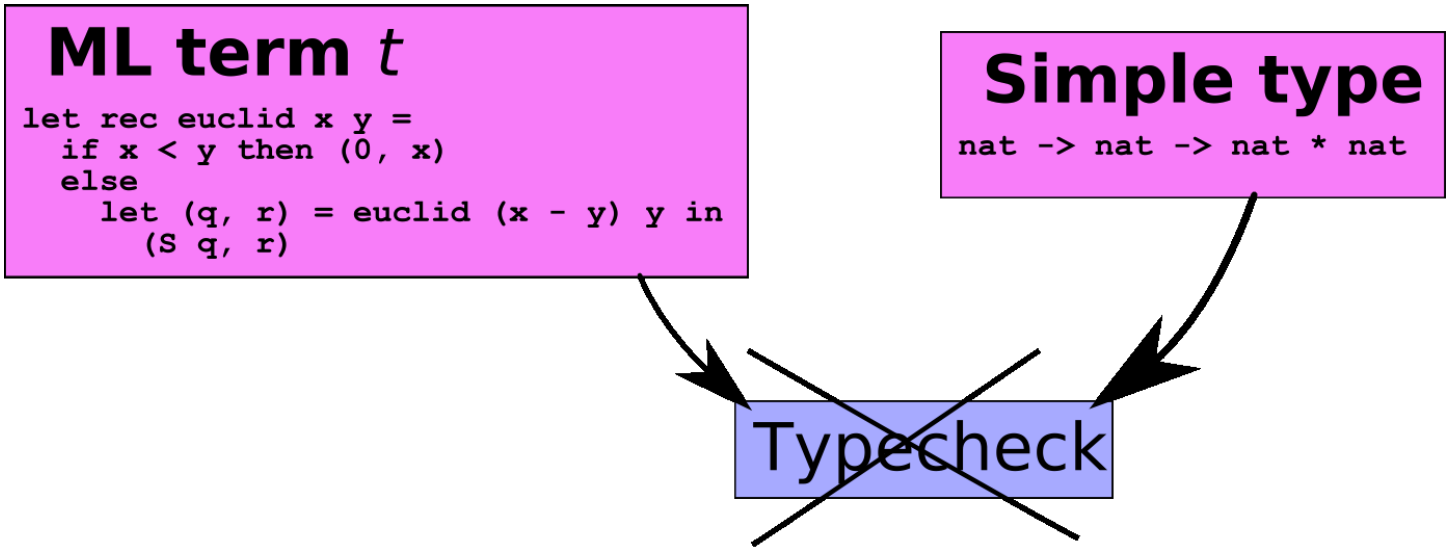
ML term t

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

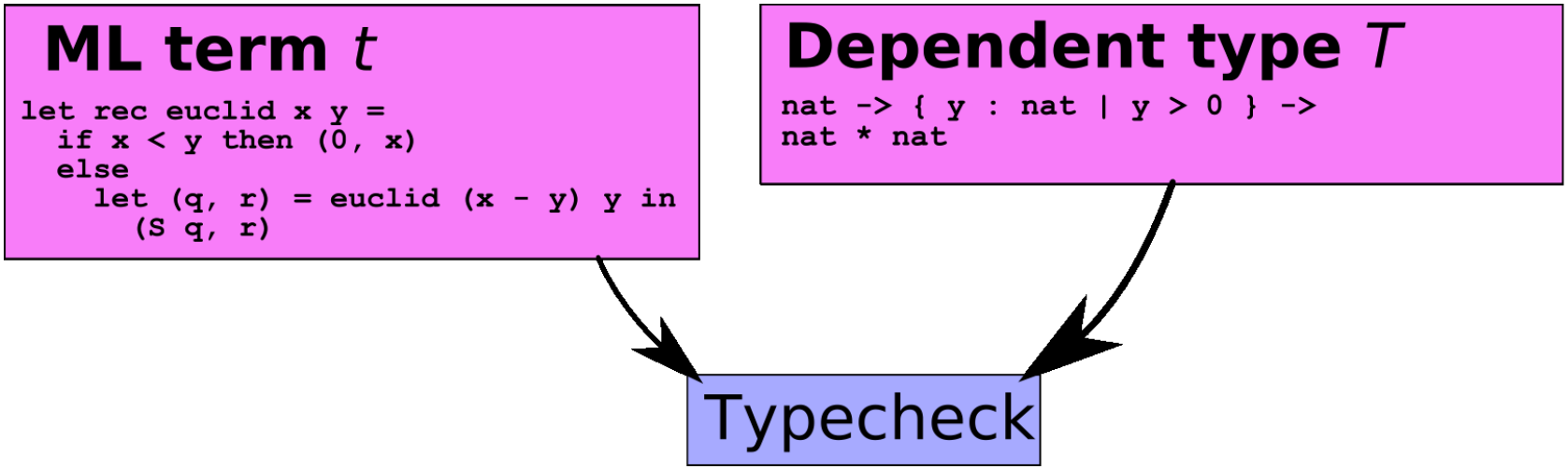
The Big Picture



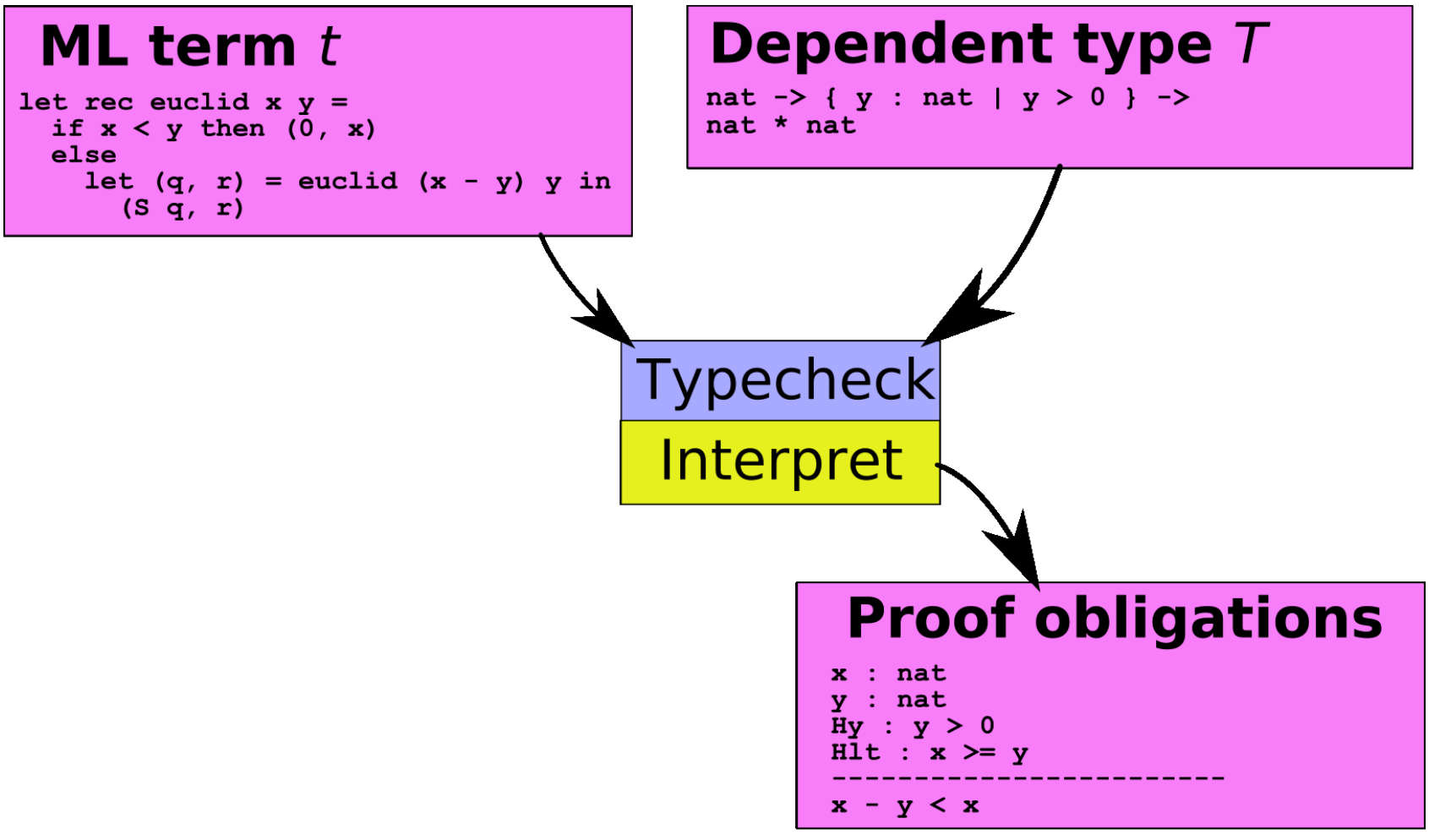
The Big Picture



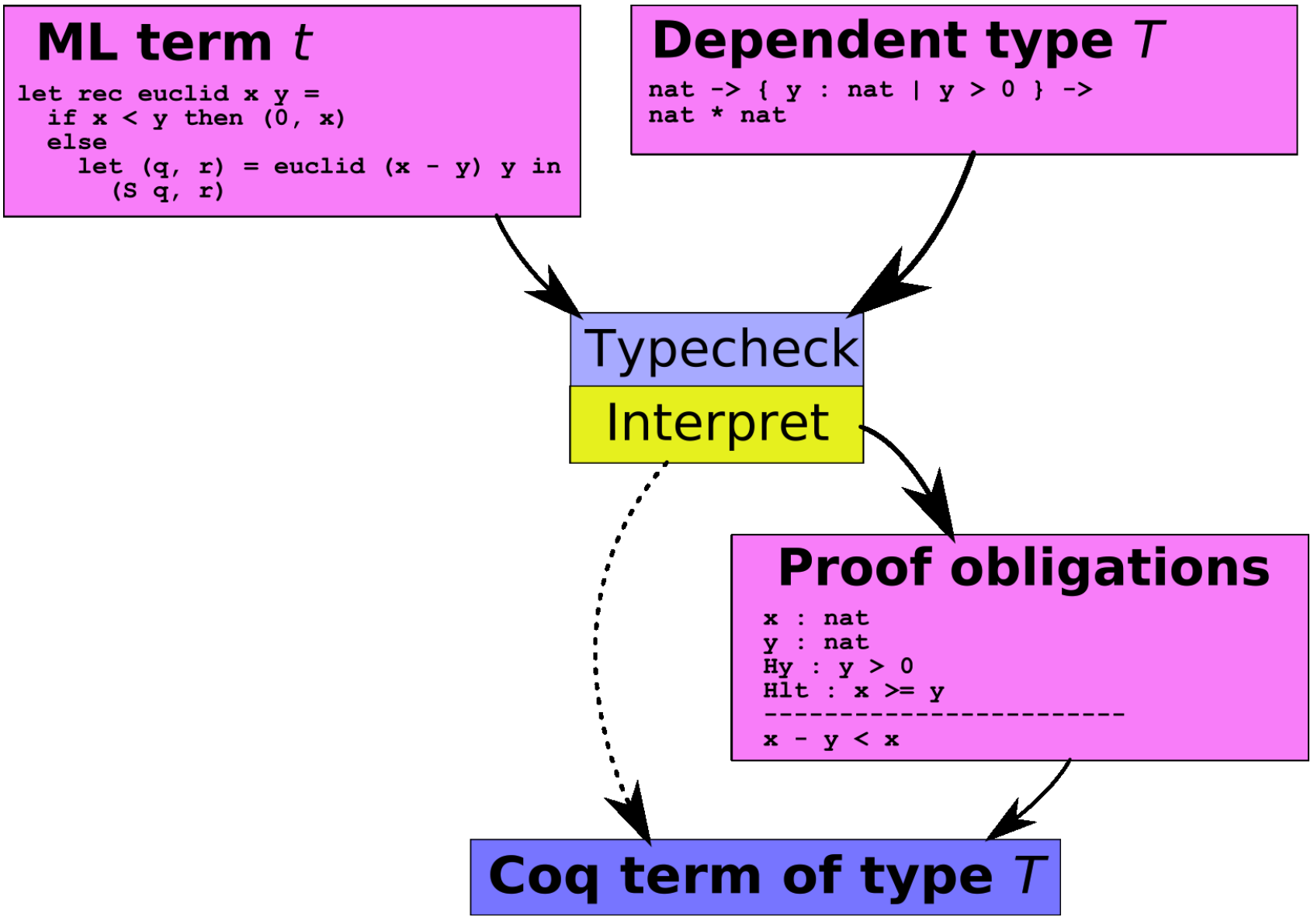
The Big Picture



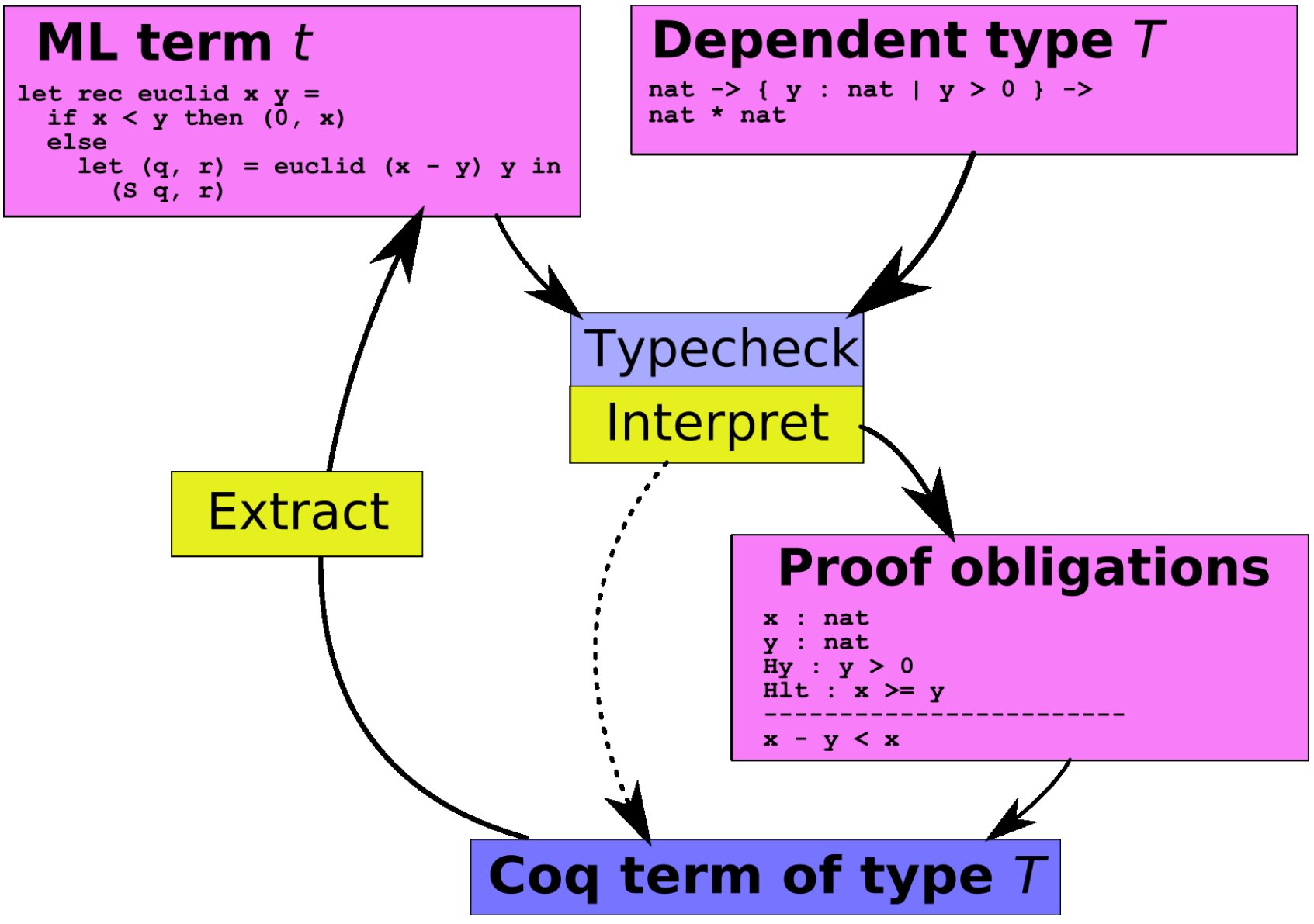
The Big Picture



The Big Picture



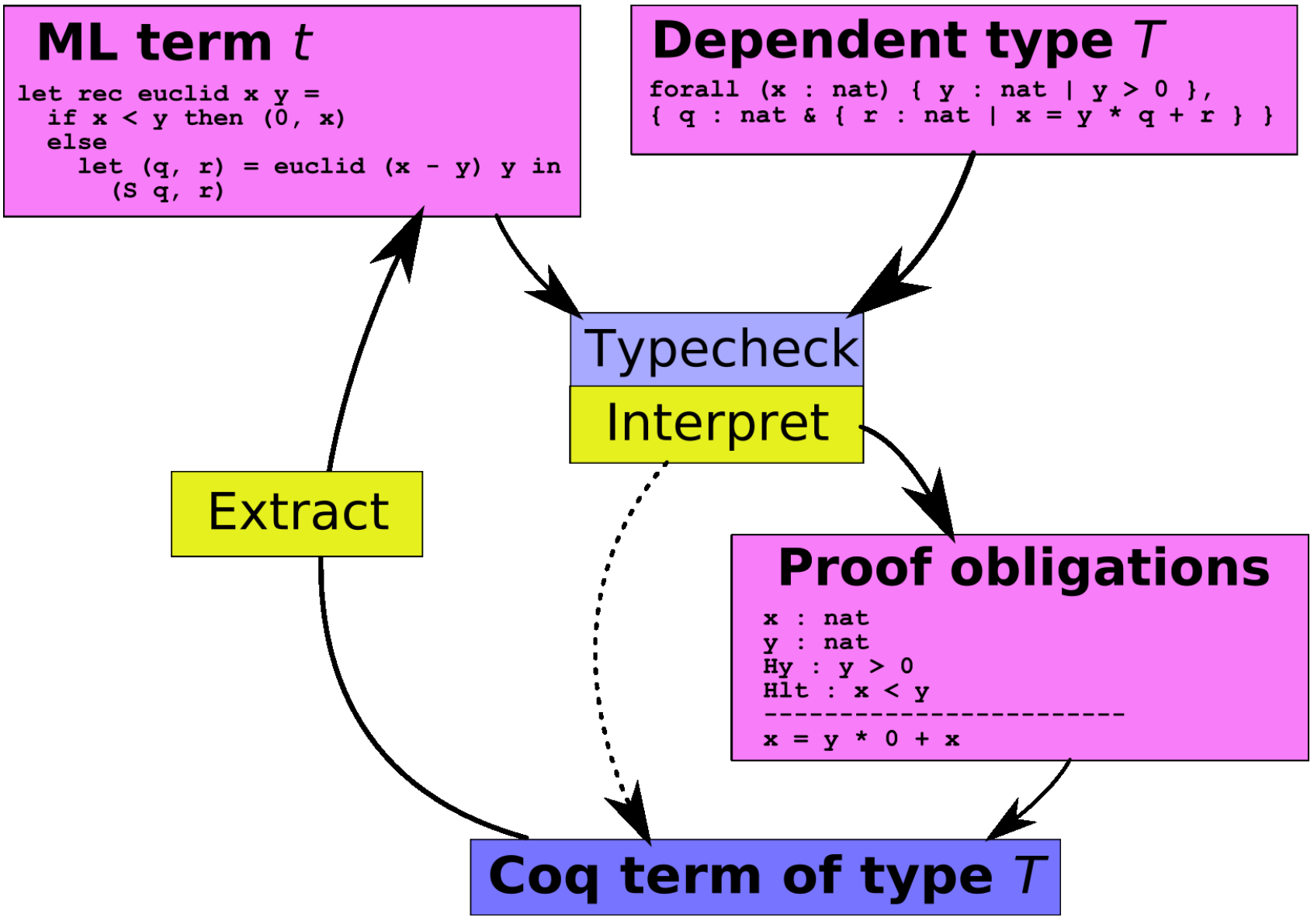
The Big Picture



The Big Picture

```
Inductive diveucl a b : Set :=
  divex : ∀ q r, b > r → a = q × b + r → diveucl a b.

Lemma eucl_dev : ∀ n, n > 0 → ∀ m:nat, diveucl m n.
Proof.
  intros b H a; pattern a in ⊢ ×; 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 ⊢ ×; auto with arith.
  elim plus_assoc.
  elim e; auto with arith.
  intros gtbn.
  apply divex with 0 n; simpl in ⊢ ×; auto with arith.
Qed.
```

The Big Picture



The Curry-Howard isomorphism

Programming language = Proof system

The Curry-Howard isomorphism

Programming language = Proof system

Program extends the Coq proof-assistant into a dependently-typed programming environment.

The Curry-Howard isomorphism

Programming language = Proof system

Program extends the Coq proof-assistant into a dependently-typed programming environment.

Epigram **PVS** **DML** **Ωmega**

The Curry-Howard isomorphism

Programming language = Proof system

Program extends the Coq proof-assistant into a dependently-typed programming environment.

- **Logical Framework** Type Theory.

Epigram **PVS** ~~**DML**~~ ~~**Ωmega**~~

The Curry-Howard isomorphism

Programming language = Proof system

Program extends the Coq proof-assistant into a dependently-typed programming environment.

- **Logical Framework** Type Theory.
  Separates proofs and programs using sorts ⇒ Extraction

~~**Epigram**~~ ~~**PVS**~~ ~~**DML**~~ ~~**Ωmega**~~

The Curry-Howard isomorphism

Programming language = Proof system

Program extends the Coq proof-assistant into a dependently-typed programming environment.

- **Logical Framework** Type Theory.
  Separates proofs and programs using sorts $\Rightarrow$ Extraction
- **Paradigm** Purely functional.

.

Epigram **PVS** ~~**DML**~~ **Ωmega**

The Curry-Howard isomorphism

Programming language = Proof system

Program extends the Coq proof-assistant into a dependently-typed programming environment.

- **Logical Framework** Type Theory.
  Separates proofs and programs using sorts ⇒ Extraction
- **Paradigm** Purely functional.
  Total, no separation of terms and types..

Epigram **PVS** ~~**DML**~~ ~~**Ωmega**~~

The Curry-Howard isomorphism

Programming language = Proof system

Program extends the Coq proof-assistant into a dependently-typed programming environment.

- **Logical Framework** Type Theory.
  Separates proofs and programs using sorts ⇒ Extraction
- **Paradigm** Purely functional.
  Total, no separation of terms and types..
- **Development style and proof automation** Interactive, semi-automatic proof using tactics.

~~Epigram~~ **PVS** ~~DML~~ ~~Ωmega~~

The Curry-Howard isomorphism

Programming language = Proof system

Program extends the Coq proof-assistant into a dependently-typed programming environment.

- **Logical Framework** Type Theory.
  Separates proofs and programs using sorts ⇒ Extraction
- **Paradigm** Purely functional.
  Total, no separation of terms and types..
- **Development style and proof automation** Interactive, semi-automatic proof using tactics.
- **Phase distinction** none

Epigram ~~**PVS**~~ **DML** **Ωmega**

The Curry-Howard isomorphism

Programming language = Proof system

Program extends the Coq proof-assistant into a dependently-typed programming environment.

- **Logical Framework** Type Theory.
  Separates proofs and programs using sorts ⇒ Extraction
- **Paradigm** Purely functional.
  Total, no separation of terms and types..
- **Development style and proof automation** Interactive, semi-automatic proof using tactics.
- **Phase distinction** ⇒ in Program

~~Epigram~~ | **PVS** | ~~DML~~ | ~~Ωmega~~

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

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

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 ;

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

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

. . . to Subset coercions

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

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

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

. . . to Subset coercions

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

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

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

Russell syntax

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

Russell syntax

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

Russell syntax

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

Russell syntax

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

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

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

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

Calculus of Constructions with

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

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

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

Calculus of Constructions with

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

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

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

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

Calculus of Constructions with

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

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

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

Example

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

Calculus of Constructions with

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

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

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

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

$$\frac{\Gamma \vdash T \rhd U : s \qquad \Gamma, x : T \vdash V \rhd W : s}{\Gamma \vdash \Sigma x : T.V \rhd \Sigma y : U.W : s} \; s \in \{\texttt{Set}, \texttt{Prop}\}$$

Calculus of Constructions with

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

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

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

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

$$\frac{\Gamma \vdash T \rhd U : s \qquad \Gamma, x : T \vdash V \rhd W : s}{\Gamma \vdash \Sigma x : T.V \rhd \Sigma y : U.W : s} \; s \in \{\texttt{Set}, \texttt{Prop}\}$$

$\rhd$ is symmetric!

Theorem (Decidability of type checking and type inference)

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

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

Results

Theorem (Decidability of type checking and type inference)

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

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

Coq corner

Mechanised proofs of Subject Reduction and equivalence between declarative and algorithmic presentations of the system.

The target system : Cic with metavariables

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

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

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

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

The target system : Cic with metavariables

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

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

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

We build an interpretation $[\![\_]\!]_\Gamma$ from Russell to 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

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

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

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

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

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

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

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

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

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

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

Example

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

Interpretation of terms

Example (Application)

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

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

Theorem (Soundness)

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

Theoretical matters ...

$\vdash_?$'s equational theory:

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

Theoretical matters ...

$\vdash_?$'s equational theory:

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

$\Rightarrow$ Proof Irrelevance

Theoretical matters ...

$\vdash_?$'s equational theory:

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

$\Rightarrow$ Proof Irrelevance

... have practical effects

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

Inductive types

Different representations

vector $n \triangleq \{\ x : \text{list}\ A \mid \textit{length}\ x = n\ \}$ or
vector $n \triangleq$ vnil : vector $0 \mid$ vcons : $A \rightarrow \forall n,$ vector $n \rightarrow$ vector $(\text{S}\ n)$?

Different representations

vector $n \triangleq \{\ x : \mathsf{list}\ A \mid \mathit{length}\ x = n\ \}$ or
vector $n \triangleq \mathsf{vnil} : \mathsf{vector}\ 0 \mid \mathsf{vcons} : A \rightarrow \forall n, \mathsf{vector}\ n \rightarrow \mathsf{vector}\ (\mathsf{S}\ n)$?

$$\frac{\Gamma \vdash v : \mathsf{vector}\ x \qquad \Gamma \vdash \mathsf{vector}\ x \rhd \mathsf{vector}\ y : \mathtt{Set}}{\Gamma \vdash v : \mathsf{vector}\ y}$$

Inductive types

Different representations

vector $n \triangleq \{\ x : \text{list}\ A \mid \textit{length}\ x = n\ \}$ or
vector $n \triangleq$ vnil : vector 0 | vcons : $A \to \forall n,$ vector $n \to$ vector (S n) ?

$$\frac{\Gamma \vdash v : \text{vector}\ x \qquad \Gamma \vdash \text{vector}\ x \rhd \text{vector}\ y : \texttt{Set}}{\Gamma \vdash v : \text{vector}\ y}$$

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 $\overrightarrow{x} = \overrightarrow{y}$.

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 $[\![\Gamma]\!] \vdash_? [\![t]\!]_\Gamma : [\![T]\!]_\Gamma$;

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

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.

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

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.

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

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

Hello world: Euclidean division

DEMO

Pattern-matching revisited

Put logic into the terms.

Let $e : \mathbb{N}$:

match e **return** T **with**
| S $n \Rightarrow$ t_1
| 0 $\Rightarrow$ t_2
end

Pattern-matching revisited

Put logic into the terms.

Let $e : \mathbb{N}$:

match e **as** t	**return** $t = e \rightarrow T$ **with**
$\mid$ S $n \Rightarrow$	**fun** $(H :$ S $n = e) \Rightarrow t_1$
$\mid 0 \Rightarrow$	**fun** $(H : 0 = e) \Rightarrow t_2$
end	(**refl_equal** e)

Pattern-matching revisited

Put logic into the terms.

Further refinements

- Each branch typed only once ;

Let $e : \mathbb{N}$:

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

Pattern-matching revisited

Put logic into the terms.

Further refinements

- Each branch typed only once ;

Let $e : \mathbb{N}$:

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

Pattern-matching revisited

Put logic into the terms.

Further refinements

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

Let $e : \mathbb{N}$:

match e **as** t **return** $t = e \rightarrow T$ **with**
| S (S n) $\Rightarrow$ **fun** $(H : \mathsf{S}\ (\mathsf{S}\ n) = e) \Rightarrow t_1$
| $n \Rightarrow$ **fun** $(H : n = e) \Rightarrow$ **let** $H' : \forall n', n \neq \mathsf{S}\ (\mathsf{S}\ n')$ **in** t_2
end (**refl_equal** e)

Pattern-matching revisited

Put logic into the terms.

Further refinements

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

Let e : vector n:

match e **return** T **with**
| vnil $\Rightarrow$ t_1
| vcons x n' v' $\Rightarrow$ t_2
end

Pattern-matching revisited

Put logic into the terms.

Further refinements

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

Let e : vector n:

match e **as** t **in** vector n' **return** $n' = n \rightarrow t \simeq e \rightarrow T$ **with**
| vnil $\Rightarrow$ **fun** $(H : 0 = n)(Hv :$ vnil $\simeq e) \Rightarrow t_1$
| vcons $x\ n'\ v' \Rightarrow$ **fun** $(H : S\ n' = n)(Hv :$ vcons $x\ n'\ v' \simeq e) \Rightarrow t_2$
end(refl_equal n)(JMeq_refl e)

Sugar

Obligations

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

Bang

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

match 0 **with** $0 \Rightarrow 0 \mid n \Rightarrow$! **end**

Sugar

Obligations

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

Bang

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

match 0 **with** $0 \Rightarrow 0 \mid n \Rightarrow !$ **end**

Destruction

Let **dest** t **as** p **in** $e \triangleq$ **match** t **with** $p \Rightarrow e$ **end**. p can be an arbitrary pattern.

Recursion

Support for well-founded recursion and measures.

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

Recursion

Support for well-founded recursion and measures.

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

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

DEMO

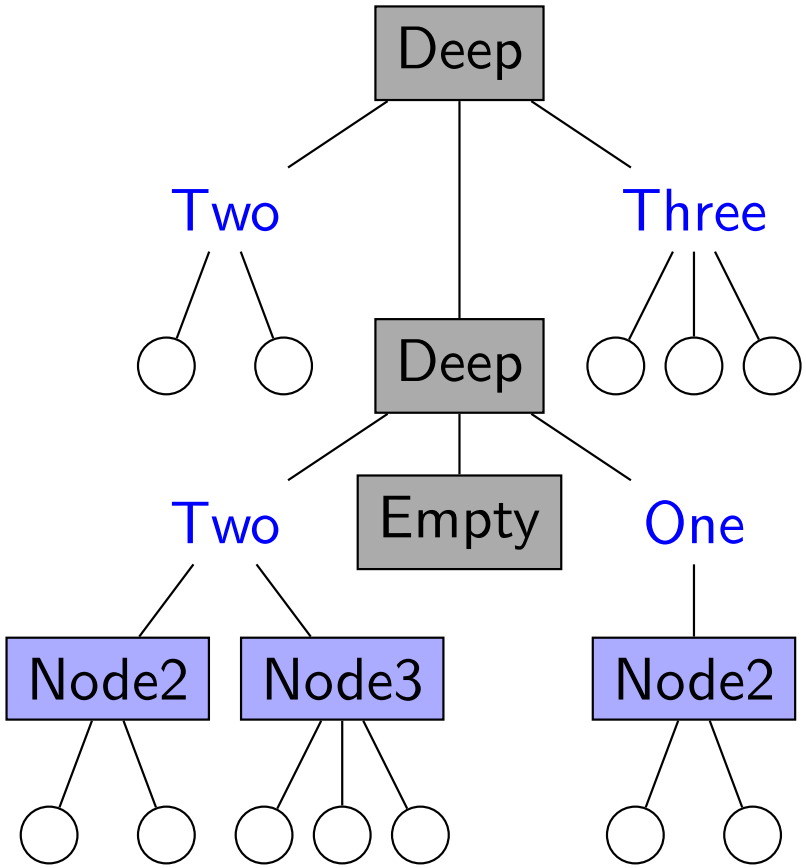
A quick tour of Finger Trees

- A Simple General Purpose Data Structure (Hinze & Paterson, JFP 2006)
- Purely functional, nested datatype
- Parameterized data structure
- Efficient deque operations, concatenation and splitting
- Comparable to Kaplan & Tarjan's catenable deques

The Big Finger Tree Picture

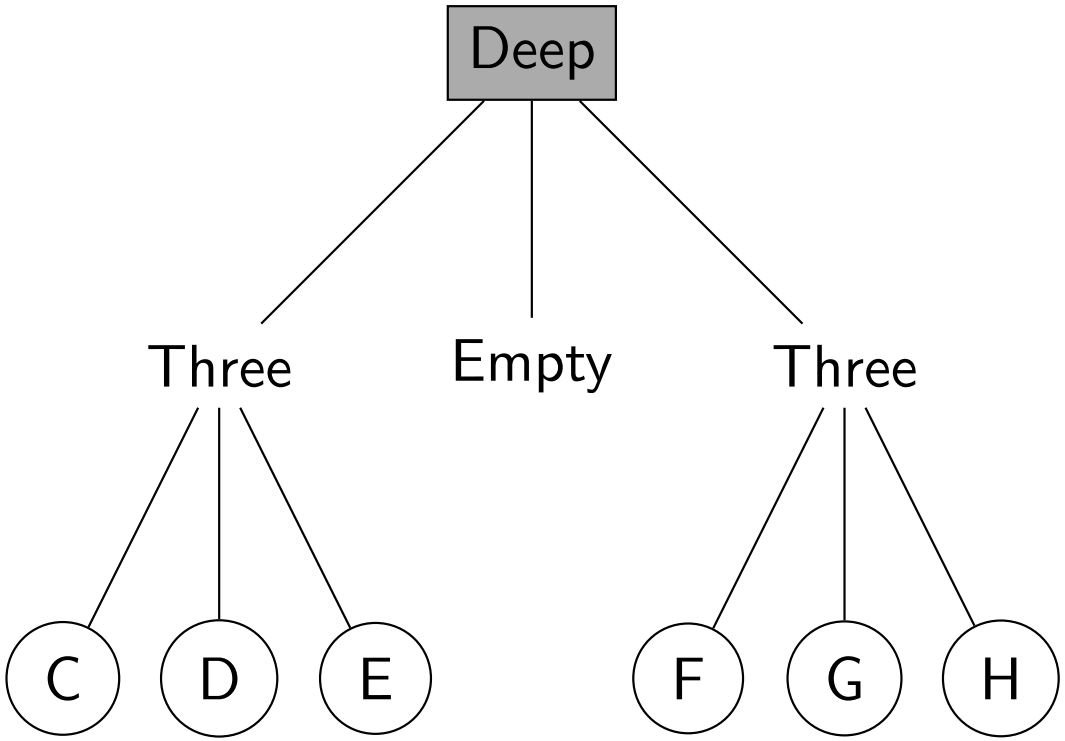
```
data Digit a = One a | Two a a | Three a a a | Four a a a a

data Node a = Node2 a a | Node3 a a a
```

```
data FingerTree a =
  | Empty
  | Single a
  | Deep
    (Digit a)
    (FingerTree (Node a))
    (Digit a)
```



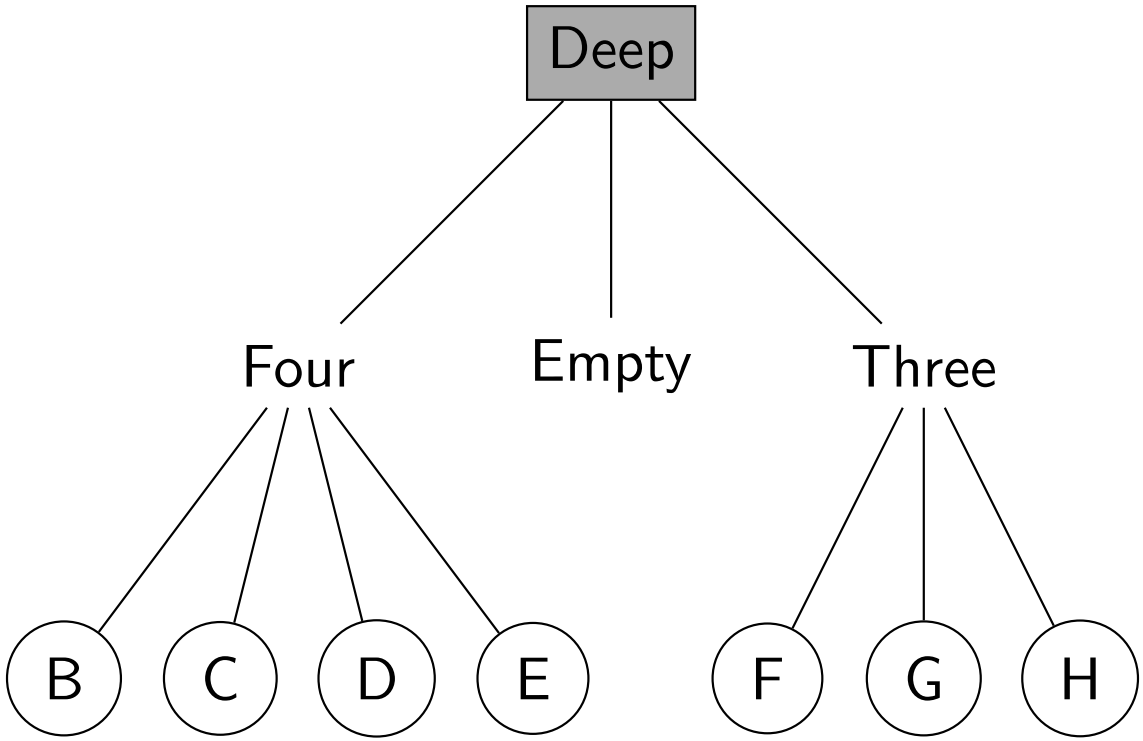
Operating on a Finger Tree

add_left :: $a \rightarrow$ FingerTree $a \rightarrow$ FingerTree a
add_left a Empty = Single a
add_left a (Single b) = Deep (One a) Empty (One b)
add_left a (Deep pr m sf) = ...



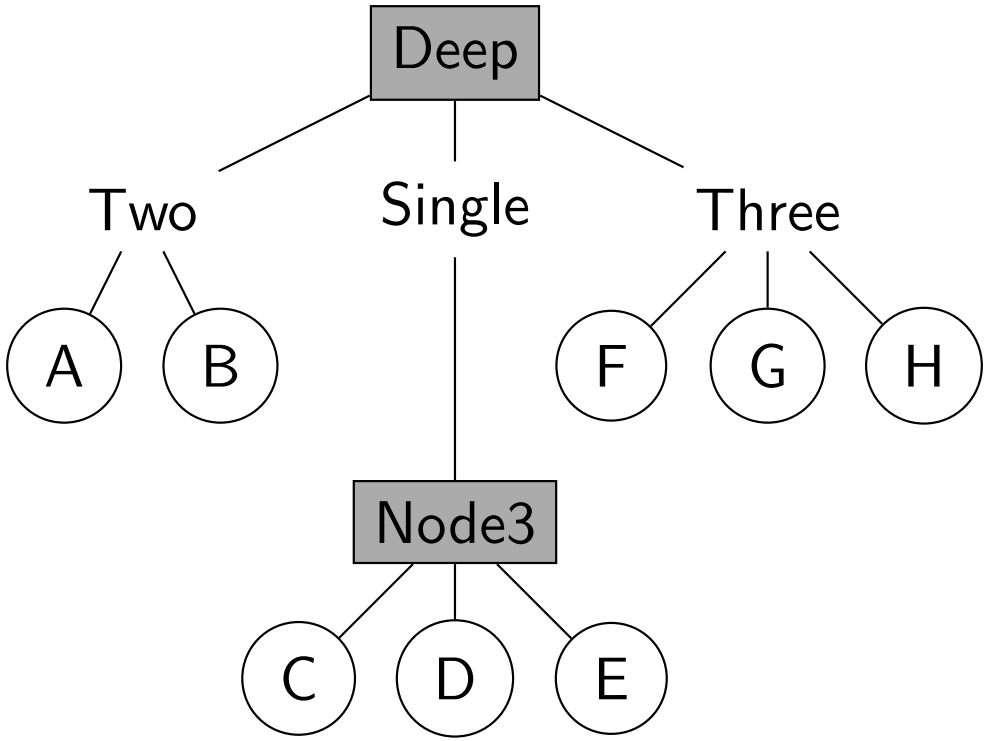
Operating on a Finger Tree

add_left :: $a \rightarrow$ FingerTree $a \rightarrow$ FingerTree a
add_left a Empty = Single a
add_left a (Single b) = Deep (One a) Empty (One b)
add_left a (Deep pr m sf) = ...



Operating on a Finger Tree

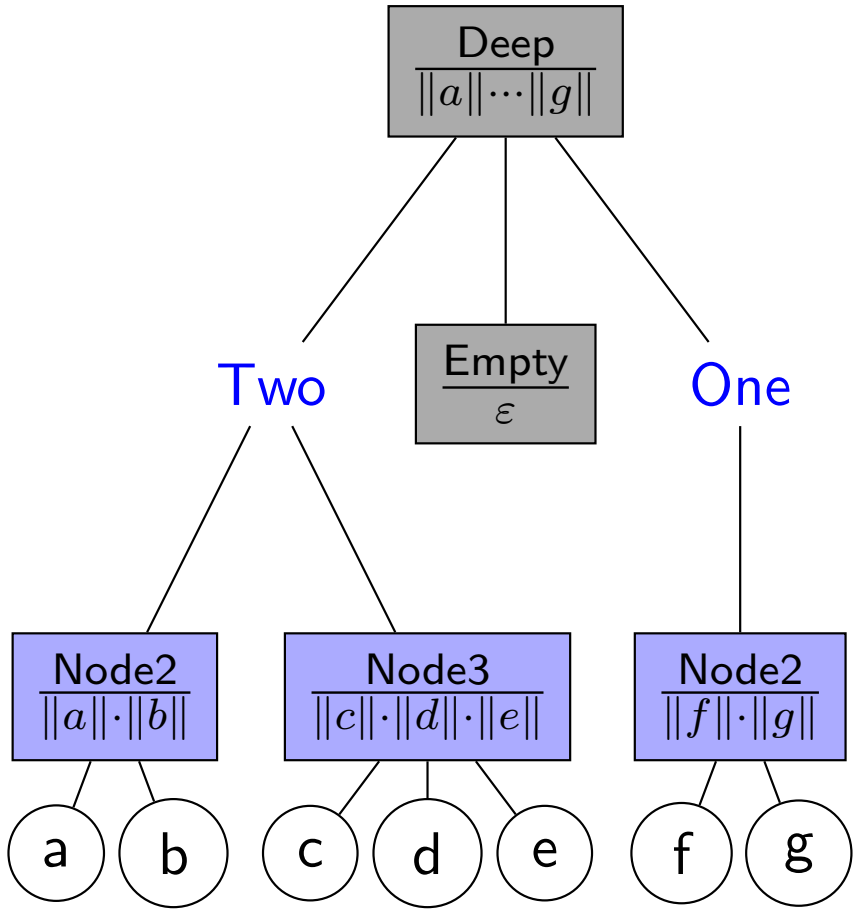
add_left :: $a \rightarrow$ FingerTree $a \rightarrow$ FingerTree a
add_left a Empty = Single a
add_left a (Single b) = Deep (One a) Empty (One b)
add_left a (Deep pr m sf) = ...



Adding cached measures

```
class Monoid v ⇒ Measured v a where
  ‖_‖ :: a → v

instance (Measured v a) ⇒ Measured v (Digit a) where ···
```

```
data Node v a =
  Node2 v a a | Node3 v a a a

data FingerTree v a =
  | Empty
  | Single a
  | Deep v
    (Digit a)
    (FingerTree v (Node v a))
    (Digit a)
```



Outline

Why do this ?

- Generally useful, non-trivial structure
- Abstraction power needed to ensure coherence of measures
- Makes dependent types (subsets and indexed datatypes) shine
- Fun ! Helps solve the ICFP contest using Coq

Digits

```
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 cont'd

```
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.
```

Nodes

```
Variables (v : Type) (mono : monoid v).
```

Nodes

```
Variables (v : Type) (mono : monoid v).

Variables (A : Type) (measure : A → v).

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

Nodes

```
Variables (v : Type) (mono : monoid v).

Variables (A : Type) (measure : A → v).

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) : Type :=
| Empty : fingertree A
| Single : ∀ x : A, fingertree A
| Deep : ∀ (l : digit A) (m : v),
  fingertree (node A) →
  ∀ (r : digit A),
  fingertree A.
```

$$node : \forall\ (A : \mathtt{Type})\ (me : A \to v),\ \mathtt{Type}$$

Dependent Finger Trees

```
Inductive fingertree (A : Type) (me : A → v) : Type :=
| Empty : fingertree A me
| Single : ∀ x : A, fingertree A me
| Deep : ∀ (l : digit A) (m : v),
  fingertree (node A me) (node_measure A me) →
  ∀ (r : digit A),
  fingertree A me.
```

$$node : \forall\ (A : \mathtt{Type})\ (me : A \to v),\ \mathtt{Type}$$

$$node\_measure\ A\ (me : A \to v) : node\ A\ measure \to v$$

Dependent Finger Trees

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

Adding to the left

```
Program Fixpoint add_left A (me : A → v)
  (a : A) (s : v) (t : fingertree me s) {struct t} :
  fingertree me (me a · s) :=
```

Adding to the left

```
Program Fixpoint add_left A (me : A → v)
  (a : A) (s : v) (t : fingertree me s) {struct t} :
  fingertree me (me a · s) :=
  match t with
    | Empty ⇒ Single a ← measure a = measure a · ε
    | Single b ⇒ Deep (One a) Empty (One b)
    | Deep pr st' t' sf ⇒
      ...
  end.
```

Adding to the left

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

The development

- Certified implementation of Finger Trees, sequences and ropes built on top of Finger Trees.
- $\sim$ 1200 lines of specification, $\sim$ 1400 of proof, mostly unchanged code.

	Haskell	Program		
	Lines	L.o.C.	Obls	L.o.P.
app	200	200	100	auto
split	20	30	14	200
FingerTree	650	600	n.a.	400

A glimpse of Finger Trees in Coq

The development

- Certified implementation of Finger Trees, sequences and ropes built on top of Finger Trees.
- $\sim$ 1200 lines of specification, $\sim$ 1400 of proof, mostly unchanged code.
- Extracts to Haskell and OCaml (with magic).

Module version fast enough for the ICFP contest. DEMO!

Experiment conclusions

+ Program scales ;
+ Subset types arise naturally ;
+ Dependent types are a powerful specification tool ;
– Need more language technology, e.g: overloading ;
– Some difficulties with reasoning 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.

Ongoing and future work

- Reasoning support through tactics
- Implementation of proof-irrelevance in COQ's kernel
- Overloading support through a typeclass mechanism.

The End

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