

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



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
```



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

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)
```

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
```

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

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
```

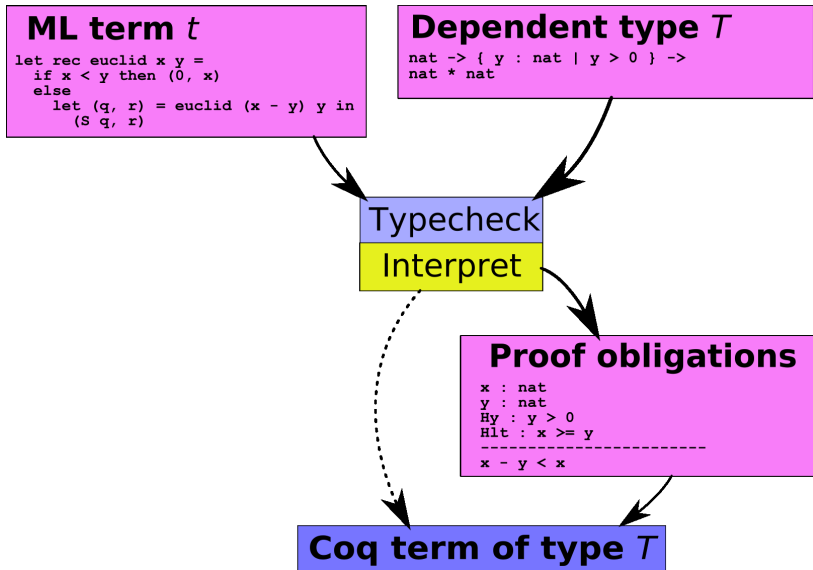
Typecheck

Interpret

Proof obligations

```
x : nat  
y : nat  
Hy : y > 0  
Hlt : x >= y  
-----  
x - y < x
```

The Big Picture



The Big Picture

Inductive *diveucl* $a b : \text{Set} :=$

divex : $\forall q r, b > r \rightarrow a = q \times b + r \rightarrow \text{diveucl } a b.$

Lemma *eucl_dev* : $\forall n, n > 0 \rightarrow \forall m:\text{nat}, \text{diveucl } m n.$

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.

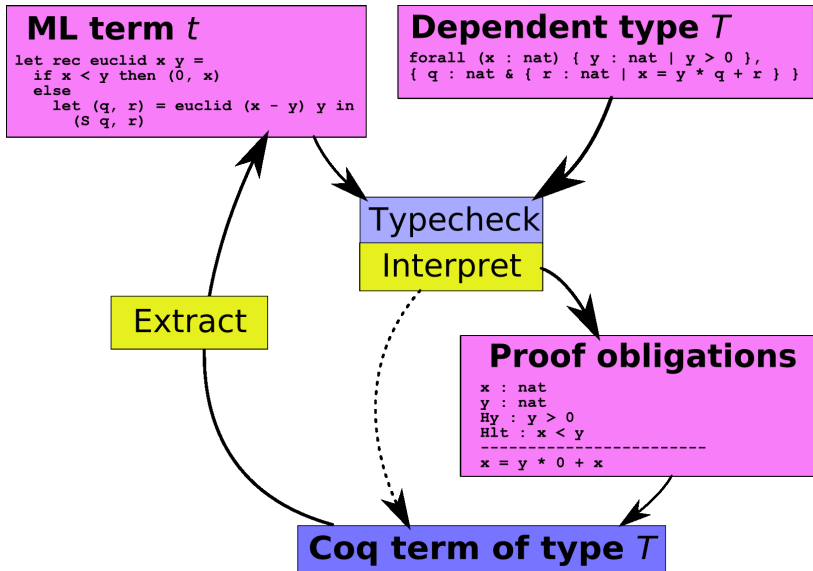
elim e; auto with arith.

intros gtbn.

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

Qed.

The Big Picture



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

Ω mega

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 \Rightarrow Extraction

~~Epigram~~

~~PVS~~

~~DML~~

~~Ω mega~~

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~~

Ω mega

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.
Total, no separation of terms and types..

Epigram

PVS

~~DML~~

~~Ω mega~~

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.
Total, no separation of terms and types..
- ▶ **Development style and proof automation** Interactive,
semi-automatic proof using tactics.

~~Epigram~~

PVS

~~DML~~

~~Ω mega~~

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.
Total, no separation of terms and types..
- ▶ **Development style and proof automation** Interactive,
semi-automatic proof using tactics.
- ▶ **Phase distinction** none

Epigram

~~PVS~~

DML

Ω mega

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.
Total, no separation of terms and types..
- ▶ **Development style and proof automation** Interactive,
semi-automatic proof using tactics.
- ▶ **Phase distinction** \Rightarrow in PROGRAM

~~Epigram~~

PVS

~~DML~~

~~Ω mega~~

- 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

- 4 Finger Trees
 - In HASKELL
 - In Coq

- 5 Conclusion

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.

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

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

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

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

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 $P[t/x]$

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 ;

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.

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

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

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

... to Subset coercions

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

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

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

... 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.

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

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

- 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
- 4 Finger Trees
 - In HASKELL
 - In Coq
- 5 Conclusion

$x \in \mathcal{V}$

$s, t, u, v ::= x$
| Set
| Prop
| Type

$x \in \mathcal{V}$

$s, t, u, v ::= x$
| Set
| Prop
| Type
| $\lambda x : s.t$
| $s t$
| $\Pi x : s.t$

$x \in \mathcal{V}$

$s, t, u, v ::= x$
| **Set**
| **Prop**
| **Type**
| $\lambda x : s.t$
| $s t$
| $\Pi x : s.t$
| $(u, v)_{\Sigma x : s.t}$
| $\pi_1 s \mid \pi_2 s$
| $\Sigma x : s.t$

$x \in \mathcal{V}$

$s, t, u, v ::= x$
| **Set**
| **Prop**
| **Type**
| $\lambda x : s.t$
| $s t$
| $\Pi x : s.t$
| $(u, v)_{\Sigma x : s.t}$
| $\pi_1 s \mid \pi_2 s$
| $\Sigma x : s.t$
| $\{ x : s \mid t \}$

Calculus of Constructions with

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

Calculus of Constructions with

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

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

Calculus of Constructions with

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

$$\frac{\Gamma \vdash U \triangleright V : \mathbf{Set} \quad \Gamma, x : U \vdash P : \mathbf{Prop}}{\Gamma \vdash \{ x : U \mid P \} \triangleright V : \mathbf{Set}}$$

$$\frac{\Gamma \vdash U \triangleright V : \mathbf{Set} \quad \Gamma, x : V \vdash P : \mathbf{Prop}}{\Gamma \vdash U \triangleright \{ x : V \mid P \} : \mathbf{Set}}$$

Calculus of Constructions with

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

$$\frac{\Gamma \vdash U \triangleright V : \mathbf{Set} \quad \Gamma, x : U \vdash P : \mathbf{Prop}}{\Gamma \vdash \{ x : U \mid P \} \triangleright V : \mathbf{Set}}$$

$$\frac{\Gamma \vdash U \triangleright V : \mathbf{Set} \quad \Gamma, x : V \vdash P : \mathbf{Prop}}{\Gamma \vdash U \triangleright \{ x : V \mid P \} : \mathbf{Set}}$$

Example
$$\frac{\Gamma \vdash 0 : \mathbb{N} \quad \Gamma \vdash \mathbb{N} \triangleright \{ x : \mathbb{N} \mid x \neq 0 \} : \mathbf{Set}}{\Gamma \vdash 0 : \{ x : \mathbb{N} \mid x \neq 0 \}}$$

Calculus of Constructions with

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

$$\frac{\Gamma \vdash U \triangleright V : \mathbf{Set} \quad \Gamma, x : U \vdash P : \mathbf{Prop}}{\Gamma \vdash \{ x : U \mid P \} \triangleright V : \mathbf{Set}}$$

$$\frac{\Gamma \vdash U \triangleright V : \mathbf{Set} \quad \Gamma, x : V \vdash P : \mathbf{Prop}}{\Gamma \vdash U \triangleright \{ x : V \mid P \} : \mathbf{Set}}$$

Example
$$\frac{\Gamma \vdash 0 : \mathbb{N} \quad \Gamma \vdash \mathbb{N} \triangleright \{ x : \mathbb{N} \mid x \neq 0 \} : \mathbf{Set}}{\Gamma \vdash 0 : \{ x : \mathbb{N} \mid x \neq 0 \}}$$

$\Gamma \vdash ? : 0 \neq 0$

Calculus of Constructions with

$$\begin{array}{c}
 \frac{\Gamma \vdash t : U \quad \Gamma \vdash U \triangleright T : s}{\Gamma \vdash t : T} \qquad \frac{\Gamma \vdash T \equiv_{\beta\pi} U : s}{\Gamma \vdash T \triangleright U : s} \\
 \\
 \frac{\Gamma \vdash U \triangleright V : \mathbf{Set} \quad \Gamma, x : U \vdash P : \mathbf{Prop}}{\Gamma \vdash \{ x : U \mid P \} \triangleright V : \mathbf{Set}} \\
 \\
 \frac{\Gamma \vdash U \triangleright V : \mathbf{Set} \quad \Gamma, x : V \vdash P : \mathbf{Prop}}{\Gamma \vdash U \triangleright \{ x : V \mid P \} : \mathbf{Set}} \\
 \\
 \frac{\Gamma \vdash U \triangleright T : s_1 \quad \Gamma, x : U \vdash V \triangleright W : s_2}{\Gamma \vdash \Pi x : T.V \triangleright \Pi x : U.W : s_2} \\
 \\
 \frac{\Gamma \vdash T \triangleright U : s \quad \Gamma, x : T \vdash V \triangleright W : s}{\Gamma \vdash \Sigma x : T.V \triangleright \Sigma y : U.W : s} \quad s \in \{\mathbf{Set}, \mathbf{Prop}\}
 \end{array}$$

Calculus of Constructions with

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

$$\frac{\Gamma \vdash U \triangleright V : \mathbf{Set} \quad \Gamma, x : U \vdash P : \mathbf{Prop}}{\Gamma \vdash \{ x : U \mid P \} \triangleright V : \mathbf{Set}}$$

$$\frac{\Gamma \vdash U \triangleright V : \mathbf{Set} \quad \Gamma, x : V \vdash P : \mathbf{Prop}}{\Gamma \vdash U \triangleright \{ x : V \mid P \} : \mathbf{Set}}$$

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

$$\frac{\Gamma \vdash T \triangleright U : s \quad \Gamma, x : T \vdash V \triangleright W : s}{\Gamma \vdash \Sigma x : T.V \triangleright \Sigma y : U.W : s} \quad s \in \{\mathbf{Set}, \mathbf{Prop}\}$$

\triangleright is symmetric!

Theorem (Decidability of type checking and type inference)

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

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

Theorem (Decidability of type checking and type inference)

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

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

Coq corner

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

- 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
- 4 Finger Trees
 - In HASKELL
 - In Coq
- 5 Conclusion

The target system : CIC with metavariables

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

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

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

We build an interpretation $\llbracket - \rrbracket_{\Gamma}$ from RUSSELL to CIC_? terms.

The target system : CIC with metavariables

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

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

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

We build an interpretation $\llbracket - \rrbracket_{\Gamma}$ from RUSSELL to $\text{CIC}_{?}$ terms.

Our goal

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

Interpretation of coercions

If $\Gamma \vdash T \triangleright U : s$ then $\Gamma \vdash_{?} c[\bullet] : T \triangleright U$ which implies
$$\llbracket \Gamma \rrbracket, x : \llbracket T \rrbracket_{\Gamma} \vdash_{?} c[x] : \llbracket U \rrbracket_{\Gamma}.$$

Interpretation of coercions

If $\Gamma \vdash T \triangleright U : s$ then $\Gamma \vdash_{?} c[\bullet] : T \triangleright U$ which implies
$$\llbracket \Gamma \rrbracket, x : \llbracket T \rrbracket_{\Gamma} \vdash_{?} c[x] : \llbracket U \rrbracket_{\Gamma}.$$

Definition

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

Interpretation of coercions

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

Definition

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

Interpretation of coercions

If $\Gamma \vdash T \triangleright U : s$ then $\Gamma \vdash_{?} c[\bullet] : T \triangleright U$ which implies
$$\llbracket \Gamma \rrbracket, x : \llbracket T \rrbracket_{\Gamma} \vdash_{?} c[x] : \llbracket U \rrbracket_{\Gamma}.$$

Definition

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

Interpretation of coercions

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

Definition

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

Interpretation of coercions

If $\Gamma \vdash T \triangleright U : s$ then $\Gamma \vdash_{?} c[\bullet] : T \triangleright U$ which implies
$$\llbracket \Gamma \rrbracket, x : \llbracket T \rrbracket_{\Gamma} \vdash_{?} c[x] : \llbracket U \rrbracket_{\Gamma}.$$

Definition

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

Interpretation of coercions

If $\Gamma \vdash T \triangleright U : s$ then $\Gamma \vdash_{?} c[\bullet] : T \triangleright U$ which implies
$$\llbracket \Gamma \rrbracket, x : \llbracket T \rrbracket_{\Gamma} \vdash_{?} c[x] : \llbracket U \rrbracket_{\Gamma}.$$

Definition

$$\frac{T \equiv_{\beta\pi} U}{\Gamma \vdash_{?} \bullet : T \triangleright U}$$
$$\Gamma \vdash_{?} \sigma_1 \bullet : \{ x : T \mid P \} \triangleright T$$
$$\Gamma \vdash_{?} \mathbf{elt} \ - \ - \ \bullet \ ?_{\llbracket P \rrbracket_{\Gamma, x:T}[\bullet/x]} : T \triangleright \{ x : T \mid P \}$$

Interpretation of coercions

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

Definition

$$\frac{T \equiv_{\beta\pi} U}{\Gamma \vdash_{?} \bullet : T \triangleright U}$$
$$\Gamma \vdash_{?} \sigma_1 \bullet : \{ x : T \mid P \} \triangleright T$$
$$\Gamma \vdash_{?} \mathbf{elt} _ _ \bullet \ ?_{[[P]]_{\Gamma, x:T}[\bullet/x]} : T \triangleright \{ x : T \mid P \}$$

Example

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

Example (Application)

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

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

Theorem (Soundness)

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

$\vdash_?$'s equational theory:

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

$\vdash_?$'s equational theory:

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

\Rightarrow Proof Irrelevance

$\vdash_?$'s equational theory:

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

\Rightarrow **Proof Irrelevance**

... have practical effects

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

Different representations

vector $n \triangleq \{ x : \text{list } A \mid \text{length } x = n \}$ or

vector $n \triangleq \text{vnil} : \text{vector } 0 \mid \text{vcons} : A \rightarrow \forall n, \text{vector } n \rightarrow \text{vector } (S n) ?$

Different representations

$\text{vector } n \triangleq \{ x : \text{list } A \mid \text{length } x = n \}$ or

$\text{vector } n \triangleq \text{vnil} : \text{vector } 0 \mid \text{vcons} : A \rightarrow \forall n, \text{vector } n \rightarrow \text{vector } (S n) ?$

$$\frac{\Gamma \vdash v : \text{vector } x \quad \Gamma \vdash \text{vector } x \triangleright \text{vector } y : \text{Set}}{\Gamma \vdash v : \text{vector } y}$$

Different representations

vector $n \triangleq \{ x : \text{list } A \mid \text{length } x = n \}$ or

vector $n \triangleq \text{vnil} : \text{vector } 0 \mid \text{vcons} : A \rightarrow \forall n, \text{vector } n \rightarrow \text{vector } (S n) ?$

$$\frac{\Gamma \vdash v : \text{vector } x \quad \Gamma \vdash \text{vector } x \triangleright \text{vector } y : \text{Set}}{\Gamma \vdash v : \text{vector } y}$$

A natural extension: reindexing

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

- 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
- 4 Finger Trees
 - In HASKELL
 - In Coq
- 5 Conclusion

Architecture

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

Architecture

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

- 1 Use the COQ parser.

Program Definition $f : T := t$.

Architecture

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

- 1 Use the COQ parser.
- 2 Typecheck $\Gamma \vdash t : T$ and generate $\llbracket \Gamma \rrbracket \vdash? \llbracket t \rrbracket_{\Gamma} : \llbracket T \rrbracket_{\Gamma}$;

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

Architecture

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

- 1 Use the COQ parser.
- 2 Typecheck $\Gamma \vdash t : T$ and generate $\llbracket \Gamma \rrbracket \vdash? \llbracket t \rrbracket_{\Gamma} : \llbracket T \rrbracket_{\Gamma}$;
- 3 Interactive proving of obligations ;

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

Architecture

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

- 1 Use the COQ parser.
- 2 Typecheck $\Gamma \vdash t : T$ and generate $\llbracket \Gamma \rrbracket \vdash? \llbracket t \rrbracket_{\Gamma} : \llbracket T \rrbracket_{\Gamma}$;
- 3 Interactive proving of obligations ;
- 4 Final definition.

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

Architecture

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

- 1 Use the COQ parser.
- 2 Typecheck $\Gamma \vdash t : T$ and generate $\llbracket \Gamma \rrbracket \vdash? \llbracket t \rrbracket_{\Gamma} : \llbracket T \rrbracket_{\Gamma}$;
- 3 Interactive proving of obligations ;
- 4 Final definition.

Restriction We assume $\Gamma \vdash_{CCI} \llbracket T \rrbracket_{\Gamma} : s$.

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

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
- 4 Finger Trees
 - In HASKELL
 - In Coq
- 5 Conclusion

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  
|  $S\ n \Rightarrow$  fun ( $H : S\ n = e$ )  $\Rightarrow t_1$   
|  $0 \Rightarrow$  fun ( $H : 0 = e$ )  $\Rightarrow t_2$   
end (refl_equal  $e$ )
```

Put **logic** into the terms.

Further refinements

- ▶ Each branch typed only once ;

Let $e : \mathbb{N}$:

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

Put **logic** into the terms.

Further refinements

- ▶ Each branch typed only once ;

Let $e : \mathbb{N}$:

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

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 ( $H : S (S n) = e$ )  $\Rightarrow t_1$   
|  $n \Rightarrow$  fun ( $H : n = e$ )  $\Rightarrow$  let  $H' : \forall n', n \neq S (S n')$  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 $e : \text{vector } n$:

```
match  $e$  return  $T$  with  
|  $\text{vnil} \Rightarrow t_1$   
|  $\text{vcons } x \ n' \ v' \Rightarrow t_2$   
end
```

Put **logic** into the terms.

Further refinements

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

Let $e : \text{vector } n$:

```
match  $e$  as  $t$  in  $\text{vector } n'$  return  $n' = n \rightarrow t \simeq e \rightarrow T$  with  
|  $\text{vnil} \Rightarrow$  fun ( $H : 0 = n$ )( $Hv : \text{vnil} \simeq e$ )  $\Rightarrow t_1$   
|  $\text{vcons } x \ n' \ v' \Rightarrow$  fun ( $H : S \ n' = n$ )( $Hv : \text{vcons } x \ n' \ v' \simeq e$ )  $\Rightarrow t_2$   
end( $\text{refl\_equal } n$ )( $\text{JMeq\_refl } e$ )
```

Obligations

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

Bang

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

match 0 with 0 \Rightarrow 0 | n \Rightarrow ! end

Obligations

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

Bang

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

match 0 with 0 \Rightarrow 0 | n \Rightarrow ! end

Destruction

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

Support for well-founded recursion and measures.

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

Support for well-founded recursion and measures.

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

$$\frac{\begin{array}{l} a : \mathbb{N} \\ f : \{x : \mathbb{N} \mid x < a\} \rightarrow \mathbb{N} \end{array}}{b : \mathbb{N}}$$

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
- 4 Finger Trees
 - In HASKELL
 - In Coq
- 5 Conclusion

- ▶ 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

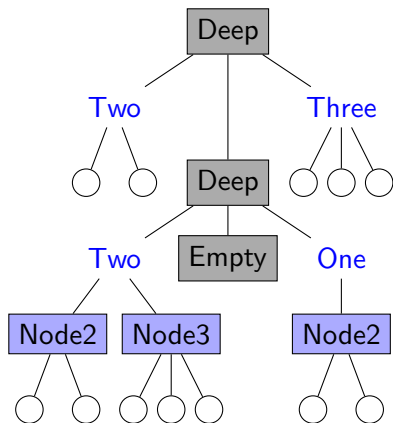
The Big Finger Tree Picture

data Digit $a = \text{One } a \mid \text{Two } a a \mid \text{Three } a a a \mid \text{Four } a a a a$

data Node $a = \text{Node2 } a a \mid \text{Node3 } a a a$

data FingerTree $a =$

- | Empty
- | Single a
- | Deep
 - (Digit a)
 - (FingerTree (Node a))
 - (Digit a)



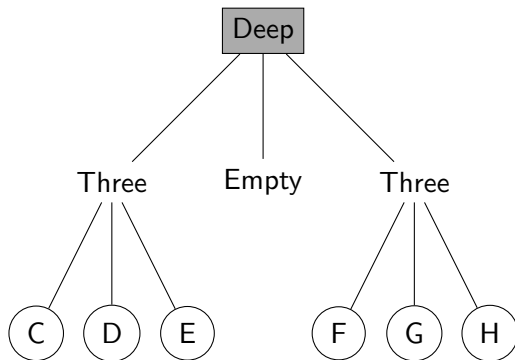
Operating on a Finger Tree

$add_left :: a \rightarrow FingerTree\ a \rightarrow FingerTree\ a$

$add_left\ a\ Empty = Single\ a$

$add_left\ a\ (Single\ b) = Deep\ (One\ a)\ Empty\ (One\ b)$

$add_left\ a\ (Deep\ pr\ m\ sf) = \dots$



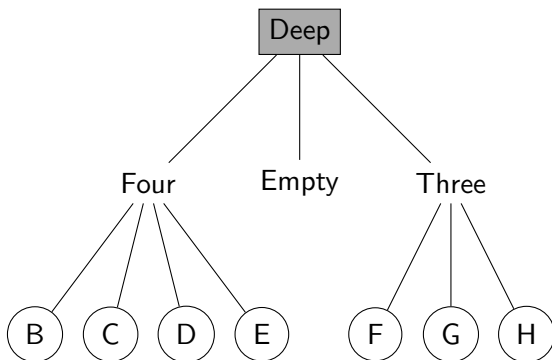
Operating on a Finger Tree

$add_left :: a \rightarrow FingerTree\ a \rightarrow FingerTree\ a$

$add_left\ a\ Empty = Single\ a$

$add_left\ a\ (Single\ b) = Deep\ (One\ a)\ Empty\ (One\ b)$

$add_left\ a\ (Deep\ pr\ m\ sf) = \dots$



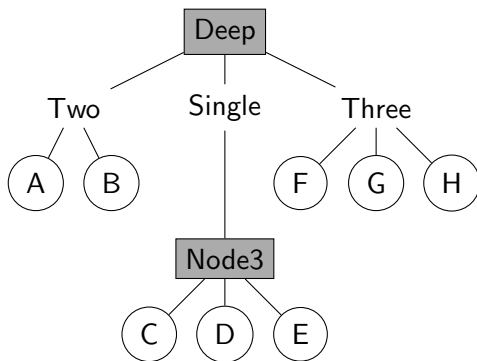
Operating on a Finger Tree

$add_left :: a \rightarrow FingerTree\ a \rightarrow FingerTree\ a$

$add_left\ a\ Empty = Single\ a$

$add_left\ a\ (Single\ b) = Deep\ (One\ a)\ Empty\ (One\ b)$

$add_left\ a\ (Deep\ pr\ m\ sf) = \dots$



Adding cached measures

class Monoid $v \Rightarrow$ Measured v a where

$\|-\| :: a \rightarrow v$

instance (Measured v a) \Rightarrow Measured v (Digit a) where ...

data Node v $a =$

Node2 v a a | Node3 v a a a

data FingerTree v $a =$

| Empty

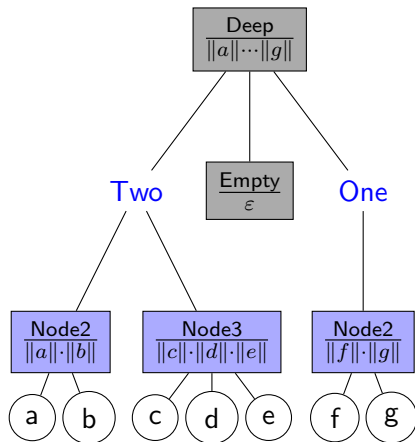
| Single a

| Deep v

(Digit a)

(FingerTree v (Node v a))

(Digit a)



- 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
- 4 Finger Trees
 - In HASKELL
 - In Coq
- 5 Conclusion

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

Variable A : Type.

Inductive digit : Type :=

| One : $A \rightarrow$ digit

| Two : $A \rightarrow A \rightarrow$ digit

| Three : $A \rightarrow A \rightarrow A \rightarrow$ digit

| Four : $A \rightarrow A \rightarrow A \rightarrow A \rightarrow$ digit.

Definition *full* x :=

match x with Four _ _ _ _ \Rightarrow True | _ \Rightarrow False end.

```
Program Definition add_digit_left
  (a : A) (d : digit |  $\neg$  full d) : digit :=
  match d with
  | One x  $\Rightarrow$  Two a x
  | Two x y  $\Rightarrow$  Three a x y
  | Three x y z  $\Rightarrow$  Four a x y z
  | Four _ _ _ _  $\Rightarrow$  !
  end.
```

Next Obligation.

```
intros ; simpl in n ; auto.
```

Qed.

Variables ($v : \text{Type}$) (*mono* : monoid v).

Variables ($v : \text{Type}$) ($mono : \text{monoid } v$).

Variables ($A : \text{Type}$) ($measure : A \rightarrow v$).

Inductive node : Type :=

| Node2 : $\forall x y, \{ s : v \mid s = \| x \| \cdot \| y \| \} \rightarrow \text{node}$

| Node3 : $\forall x y z, \{ s : v \mid s = \| x \| \cdot \| y \| \cdot \| z \| \} \rightarrow \text{node}.$

Variables ($v : \text{Type}$) ($mono : \text{monoid } v$).

Variables ($A : \text{Type}$) ($measure : A \rightarrow v$).

Inductive node : Type :=

| Node2 : $\forall x y, \{ s : v \mid s = \| x \| \cdot \| y \| \} \rightarrow \text{node}$

| Node3 : $\forall x y z, \{ s : v \mid s = \| x \| \cdot \| y \| \cdot \| z \| \} \rightarrow \text{node}$.

Program Definition $node2 (x y : A) : \text{node} :=$
Node2 x y ($\| x \| \cdot \| y \|$).

Program Definition $node_measure (n : \text{node}) : v :=$
match n with Node2 _ _ s \Rightarrow s | Node3 _ _ _ s \Rightarrow s end.

Dependent Finger Trees

Inductive `fingertree (A : Type) : Type :=`

| `Empty : fingertree A`

| `Single : $\forall x : A$, fingertree A`

| `Deep : $\forall (l : \text{digit } A) (m : v)$,
fingertree (node A) \rightarrow
 $\forall (r : \text{digit } A)$,
fingertree A.`

`node : $\forall (A : \text{Type}) (me : A \rightarrow v)$, Type`

Dependent Finger Trees

Inductive fingertree $(A : \text{Type})$ $(me : A \rightarrow v)$: Type :=

| Empty : fingertree A me

| Single : $\forall x : A$, fingertree A me

| Deep : $\forall (l : \text{digit } A)$ $(m : v)$,
fingertree (node A me) $(node_measure A me) \rightarrow$
 $\forall (r : \text{digit } A)$,
fingertree A me .

$node : \forall (A : \text{Type})$ $(me : A \rightarrow v)$, Type

$node_measure A (me : A \rightarrow v) : node A measure \rightarrow v$

Dependent Finger Trees

Inductive `fingertree` ($A : \text{Type}$) ($me : A \rightarrow v$) : $v \rightarrow \text{Type} :=$

| `Empty` : `fingertree A me ε`

| `Single` : $\forall x : A$, `fingertree A me ($me\ x$)`

| `Deep` : $\forall (l : \text{digit } A) (m : v)$,
 `fingertree (node A me) (node_measure A me) m` \rightarrow
 $\forall (r : \text{digit } A)$,
 `fingertree A me`
 (`digit_measure me l` \cdot `m` \cdot `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.
- ▶ ~ 1200 lines of specification, ~ 1400 of proof, mostly unchanged code.

	HASKELL	PROGRAM		
	Lines	L.o.C.	Obls	L.o.P.
<i>app</i>	200	200	100	auto
<i>split</i>	20	30	14	200
FingerTree	650	600	n.a.	400

The development

- ▶ Certified implementation of Finger Trees, sequences and ropes built on top of Finger Trees.
- ▶ ~ 1200 lines of specification, ~ 1400 of proof, mostly unchanged code.
- ▶ Extracts to HASKELL and OCAML (with magic).

Module version fast enough for the ICFP contest. DEMO!

- + 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.

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.

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