

Programming with Dependent Types in Coq

MATTHIEU SOZEAU

LRI, Univ. Paris-Sud - DÉMONS Team & INRIA Saclay - PROVAL Project

PPS Seminar
February 26th 2009
Paris, France



- ▶ A higher-order, polymorphic logic:

$$\forall A (P Q : A \rightarrow \mathbf{Prop}), (\forall x, P x \leftrightarrow Q x) \rightarrow \forall x, P x \rightarrow Q x$$

$$\forall x y : \mathbb{N}, x + y = y + x$$

- ▶ A higher-order, polymorphic logic:

$$\forall A (P Q : A \rightarrow \mathbf{Prop}), (\forall x, P x \leftrightarrow Q x) \rightarrow \forall x, P x \rightarrow Q x$$

$$\forall x y : \mathbb{N}, x + y = y + x$$

- ▶ A purely functional programming language with dependent types:

$$(\lambda A (x : A), x) : \forall A, A \rightarrow A$$

$$\text{div} : \forall (a : \text{nat}) (b : \text{nat} \mid b \neq 0), \\ \{ (q, r) : \text{nat} \times \text{nat} \mid a = b \times q + r \wedge r < b \}$$

- ▶ A higher-order, polymorphic logic:

$$\forall A (P Q : A \rightarrow \mathbf{Prop}), (\forall x, P x \leftrightarrow Q x) \rightarrow \forall x, P x \rightarrow Q x$$

$$\forall x y : \mathbb{N}, x + y = y + x$$

- ▶ A purely functional programming language with dependent types:

$$(\lambda A (x : A), x) : \forall A, A \rightarrow A$$

$$\text{div} : \forall (a : \text{nat}) (b : \text{nat} \mid b \neq 0), \\ \{ (q, r) : \text{nat} \times \text{nat} \mid a = b \times q + r \wedge r < b \}$$

Problem: writing such programs is difficult.

Programming with tactics

Lemma `eucl_dev` : $\forall n, n > 0 \rightarrow \forall m : \text{nat},$
 $\{ (q, r) : \text{nat} \times \text{nat} \mid n > r \wedge m = q \times n + r \}.$

Proof.

```
intros b H a; pattern a; apply gt_wf_rec; intros n H0.
elim (le_gt_dec b n).
intro lebn.
case (H0 (n - b)); auto with arith.
intros [q r] [g e].
 $\exists$  (S q, r); simpl; auto with arith.
elim plus_assoc.
elim e; auto with arith.
intros gtbn.
 $\exists$  (0, n); simpl; auto with arith.
```

Qed.

Programming directly

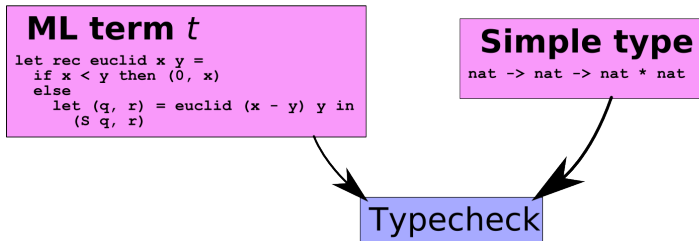
```
λ b (H : b > 0) a,
gt_wf_rec a (λ n, { (q, r) : nat × nat | b > r ∧ n = q × b + r })
(λ n
  (H0 : Π m : nat, n > m → { (q, r) : nat × nat | b > r ∧ m = q × b + r })),
sumbool_rec
(λ _ : {b ≤ n} + {b > n},
 { (q, r) : nat × nat | b > r ∧ n = q × b + r })
(λ lebn : b ≤ n,
  let (x, x0) := H0 (n - b) (lt_minus n b lebn H) in
  (let (q, r) as p
    return
      ((let (q, r) := p in b > r ∧ n - b = q × b + r) →
       { (q, r) : nat × nat | b > r ∧ n = q × b + r }) := x in
  λ y : b > r ∧ n - b = q × b + r,
  match y with
  | conj g e ⇒
    elt
      (eq_ind (b + (q × b + r)) (λ n0, b > r ∧ n = n0)
        (eq_ind (n - b) (λ n0, b > r ∧ n = b + n0)
          (conj g (le_plus_minus b n lebn))
            (q × b + r) e) (b + q × b + r) (plus_assoc b (q × b) r))
    end) x0) (λ gtbn : b > n, elt (conj gtbn refl))
(le_gt_dec b n))
```

We can write programs as usual and still give them rich types.

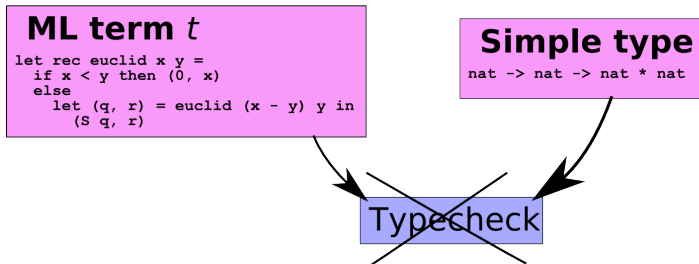
ML term t

```
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

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)
```

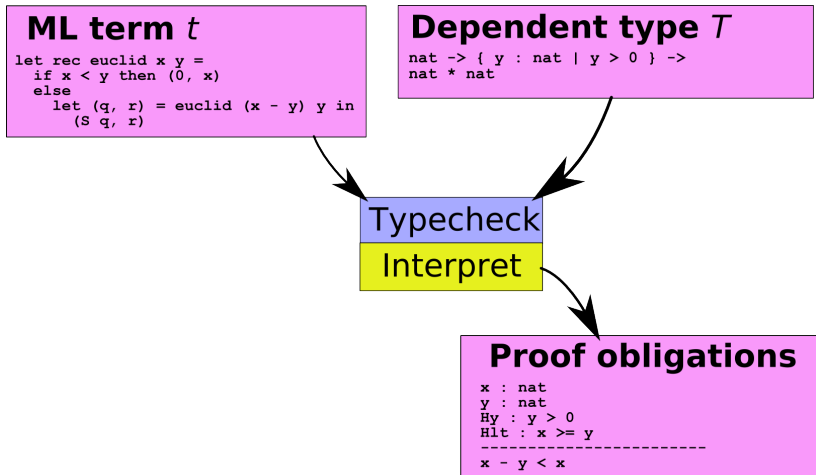
Dependent type T

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

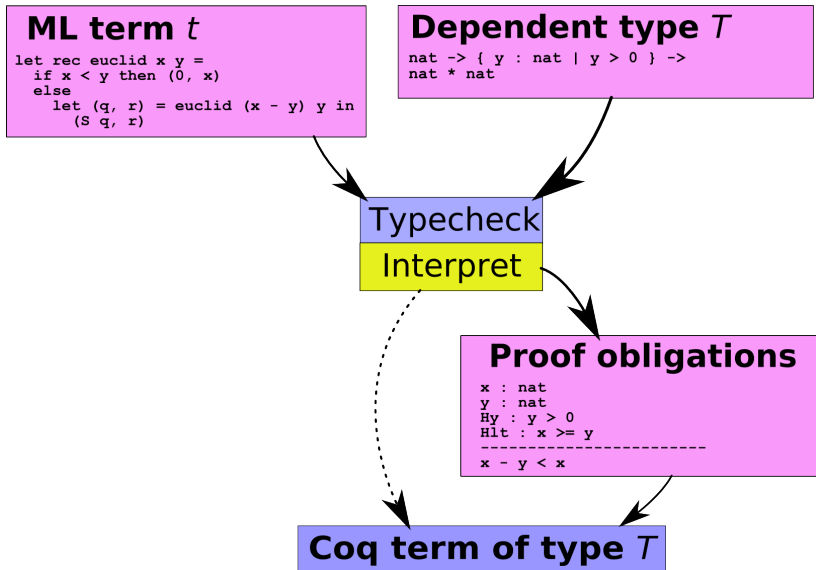
Typecheck

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

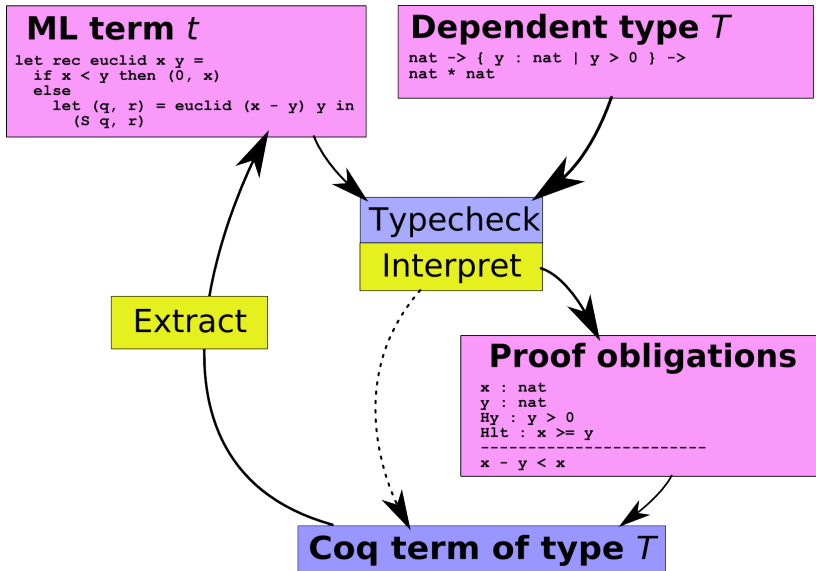
The Big Picture



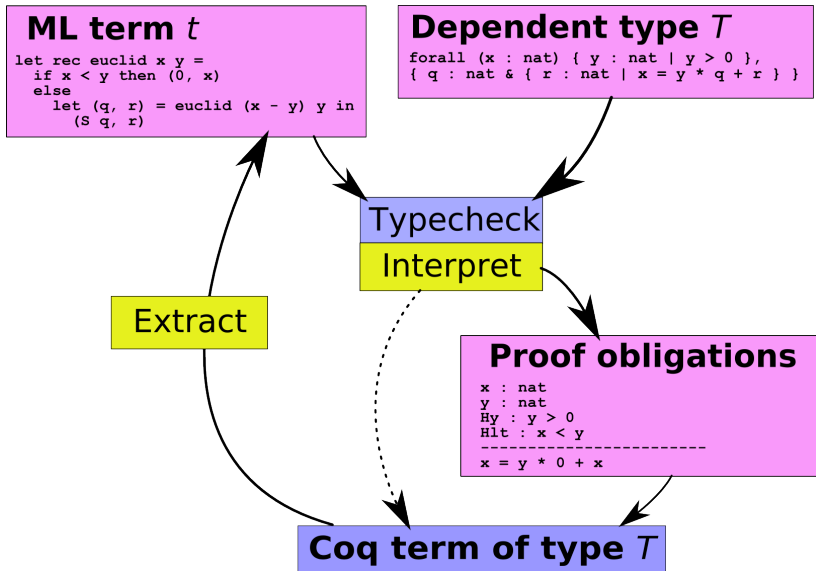
The Big Picture



The Big Picture



The Big Picture



- 1 RUSSELL
 - Subset Coercions: A Simple Idea
 - Metatheory
 - Interpretation in COQ

- 2 PROGRAM
 - Extensions
 - Hello World

- 3 Finger Trees
 - Finger Trees in HASKELL
 - Dependent Finger Trees
 - Specializations
 - Ropes
 - Random-access sequences

- 4 Conclusion

Definition

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

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

PVS

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

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

PVS

- ▶ Specialized typing algorithm for subset types, generating *Type-checking conditions*.
- + Practical success ;

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

PVS

- ▶ Specialized typing algorithm for subset types, generating *Type-checking conditions*.
- + Practical success ;
- No strong safety guarantee in PVS.

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

PVS

- ▶ Specialized typing algorithm for subset types, generating *Type-checking conditions*.
- + Practical success ;
- No strong safety guarantee in PVS.

$$\frac{t : T \quad p : P[t/x]}{\text{elt } t \ p : \{ x : T \mid P \}} \qquad \frac{t : \{ x : T \mid P \}}{\sigma_1 t : T}$$

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

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

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

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

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

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

- 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.

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

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

Calculus of Constructions with

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

Calculus of Constructions with

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

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

Calculus of Constructions with

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

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

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

Calculus of Constructions with

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

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

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

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

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

Calculus of Constructions with

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

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

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

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

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

\triangleright is symmetric!

Theorem (Subject Reduction)

If $\Gamma \vdash t : T$ and $t \rightarrow_{\beta} u$ then $\Gamma \vdash u : T$



Using the TPOSR technique due to Robin Adams.

Theorem (Subject Reduction)

If $\Gamma \vdash t : T$ and $t \rightarrow_{\beta} u$ then $\Gamma \vdash u : T$



Using the TPOSR technique due to Robin Adams.

Theorem (Decidability of type checking and type inference)

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

The target system : CIC with metavariables

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

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

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

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

The target system : CIC with metavariables

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

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

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

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

Our goal: proving soundness

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

Interpretation of coercions

If $\Gamma \vdash T \triangleright U : s$ then there exists c so that $\Gamma \vdash_{\text{?}} c : T \triangleright U$.

Interpretation of coercions

If $\Gamma \vdash T \triangleright U : s$ then there exists c so that $\Gamma \vdash_{\gamma} c : T \triangleright U$.

$$\frac{\Gamma \vdash_{\gamma} T \equiv_{\beta} U : s}{\Gamma \vdash_{\gamma} \quad : T \triangleright U}$$

Interpretation of coercions

If $\Gamma \vdash T \triangleright U : s$ then there exists c so that $\Gamma \vdash_{\gamma} c : T \triangleright U$.

$$\frac{\Gamma \vdash_{\gamma} T \equiv_{\beta} U : s}{\Gamma \vdash_{\gamma} \bullet : T \triangleright U}$$

Interpretation of coercions

If $\Gamma \vdash T \triangleright U : s$ then there exists c so that $\Gamma \vdash_{\gamma} c : T \triangleright U$.

$$\frac{\Gamma \vdash_{\gamma} T \equiv_{\beta} U : s}{\Gamma \vdash_{\gamma} \bullet : T \triangleright U}$$

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

Interpretation of coercions

If $\Gamma \vdash T \triangleright U : s$ then there exists c so that $\Gamma \vdash_{\gamma} c : T \triangleright U$.

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

Interpretation of coercions

If $\Gamma \vdash T \triangleright U : s$ then there exists c so that $\Gamma \vdash_{\gamma} c : T \triangleright U$.

$$\frac{\Gamma \vdash_{\gamma} T \equiv_{\beta} U : s}{\Gamma \vdash_{\gamma} \bullet : T \triangleright U}$$

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

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

Interpretation of coercions

If $\Gamma \vdash T \triangleright U : s$ then there exists c so that $\Gamma \vdash_{?} c : T \triangleright U$.

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

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

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

Interpretation of coercions

If $\Gamma \vdash T \triangleright U : s$ then there exists c so that $\Gamma \vdash_{\text{?}} c : T \triangleright U$.

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

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

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

Example

$$\frac{\Gamma \vdash_{\text{?}} 0 : \mathbb{N} \quad \Gamma \vdash_{\text{?}} \text{elt } \bullet \text{ ?}_{\bullet \neq 0} : \mathbb{N} \triangleright \{ x : \mathbb{N} \mid x \neq 0 \}}{\Gamma \vdash_{\text{?}} \text{elt } 0 \text{ ?}_{0 \neq 0} : \{ x : \mathbb{N} \mid x \neq 0 \}}$$

Example (Application)

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

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

Theorem (Soundness)

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

$\vdash_?$'s equational theory:

$$\beta\text{-rules} \left\{ \begin{array}{l} (\beta) \quad (\lambda x : X.e) v \quad \equiv \quad e[v/x] \\ (\sigma_i) \quad \sigma_i (\text{elt}_{E,P} e_1 e_2) \quad \equiv \quad e_i \end{array} \right\} \text{CoQ}$$

$\vdash_?$'s equational theory:

$$\beta\text{-rules} \left\{ \begin{array}{l} (\beta) \quad (\lambda x : X.e) v \quad \equiv \quad e[v/x] \\ (\sigma_i) \quad \sigma_i (\text{elt}_{E,P} e_1 e_2) \quad \equiv \quad e_i \end{array} \right\} \text{CoQ}$$

$$\eta\text{-rules} \left\{ \begin{array}{l} (\eta) \quad (\lambda x : X.e x) \quad \equiv \quad e \quad \text{if } x \notin \mathcal{FV}(e) \\ (\text{SP}) \quad \text{elt}_{E,P} (\sigma_1 e) (\sigma_2 e) \quad \equiv \quad e \end{array} \right.$$

$\vdash_?$'s equational theory:

$$\beta\text{-rules} \left\{ \begin{array}{l} (\beta) \quad (\lambda x : X.e) v \quad \equiv \quad e[v/x] \\ (\sigma_i) \quad \sigma_i (\mathbf{elt}_{E,P} e_1 e_2) \quad \equiv \quad e_i \end{array} \right\} \text{CoQ}$$

$$\eta\text{-rules} \left\{ \begin{array}{l} (\eta) \quad (\lambda x : X.e x) \quad \equiv \quad e \quad \text{if } x \notin \mathcal{FV}(e) \\ (\text{SP}) \quad \mathbf{elt}_{E,P} (\sigma_1 e) (\sigma_2 e) \quad \equiv \quad e \end{array} \right.$$

$$\text{Proof Irrelevance} \quad \mathbf{elt}_{E,P} t p \quad \equiv \quad \mathbf{elt}_{E,P} t' p' \quad \text{if } t \equiv t'$$

$\vdash_?$'s equational theory:

$$\beta\text{-rules} \left\{ \begin{array}{l} (\beta) \quad (\lambda x : X.e) v \quad \equiv \quad e[v/x] \\ (\sigma_i) \quad \sigma_i (\mathbf{elt}_{E,P} e_1 e_2) \quad \equiv \quad e_i \end{array} \right\} \text{CoQ}$$

$$\eta\text{-rules} \left\{ \begin{array}{l} (\eta) \quad (\lambda x : X.e x) \quad \equiv \quad e \quad \text{if } x \notin \mathcal{FV}(e) \\ (\text{SP}) \quad \mathbf{elt}_{E,P} (\sigma_1 e) (\sigma_2 e) \quad \equiv \quad e \end{array} \right.$$

$$\text{Proof Irrelevance} \quad \mathbf{elt}_{E,P} t p \quad \equiv \quad \mathbf{elt}_{E,P} t' p' \quad \text{if } t \equiv t'$$

... have practical effects

Difficulty to reason on code: $t \equiv u \not\rightarrow \llbracket t \rrbracket_\Gamma \equiv_{\text{CoQ}} \llbracket u \rrbracket_\Gamma$.

- 1 RUSSELL
 - Subset Coercions: A Simple Idea
 - Metatheory
 - Interpretation in COQ

- 2 PROGRAM
 - Extensions
 - Hello World

- 3 Finger Trees
 - Finger Trees in HASKELL
 - Dependent Finger Trees
 - Specializations
 - Ropes
 - Random-access sequences

- 4 Conclusion

The PROGRAM vernacular

Replaces the Program tactic by Catherine Parent.

The PROGRAM vernacular

Replaces the Program tactic by Catherine Parent.

Architecture

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

The PROGRAM vernacular

Replaces the Program tactic by Catherine Parent.

Architecture

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

- 1 Use the COQ parser ;

Program Definition $f : T := t$.

The PROGRAM vernacular

Replaces the Program tactic by Catherine Parent.

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

The PROGRAM vernacular

Replaces the Program tactic by Catherine Parent.

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 proof of the obligations ;

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

The PROGRAM vernacular

Replaces the Program tactic by Catherine Parent.

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 proof of the obligations ;
- 4 Final definition.

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

Obligations

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

Bang

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

```
match x with 0 => ! | n => ... end
```

Obligations

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

Bang

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

```
match x with 0  $\Rightarrow$  ! | n  $\Rightarrow$  ... end
```

Destruction

Let `let 'p := t in e = match t with p \Rightarrow e end`. `p` can be an arbitrary pattern.

Support for well-founded recursion and measures.

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

Support for well-founded recursion and measures.

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

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

DEMO

Pattern-matching revisited

Put **logic** into the terms.

Let $e : \mathbb{N}$:

```
match e      return      T with
| S n ⇒      t1
| 0 ⇒      t2
end
```

Pattern-matching revisited

Put **logic** into the terms.

Let $e : \mathbb{N}$:

```
match e as t   return t = e → T with
| S n ⇒       fun (H : S n = e) ⇒ t1
| 0 ⇒         fun (H : 0 = e) ⇒ t2
end            (refl_equal e)
```

Pattern-matching revisited

Put **logic** into the terms.

Further refinements

- ▶ Each branch typed only once ;

Let $e : \mathbb{N}$:

```
match e as t return t = e → T with
| S (S n) ⇒ fun (H : S (S n) = e) ⇒ t1
| n ⇒      fun (H : n = e) ⇒ t2
end       (refl_equal e)
```

Pattern-matching revisited

Put **logic** into the terms.

Further refinements

- ▶ Each branch typed only once ;

Let $e : \mathbb{N}$:

```
match e as t return t = e → T with
| S (S n) ⇒ fun (H : S (S n) = e) ⇒ t1
| S 0 ⇒ fun (H : S 0 = e) ⇒ t2
| 0 ⇒ fun (H : 0 = e) ⇒ t2
end (refl_equal e)
```

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 → T with
| S (S n) ⇒ fun (H : S (S n) = e) ⇒ t1
| n ⇒ fun (H : n = e) ⇒ let H' : ∀n', n ≠ S (S n') in t2
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
| 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 $e : \text{vector } n$:

```
match e as t in vector n' return n' = n → JMeq t e → T with
| vnil ⇒ fun (H : 0 = n)(Hv : JMeq vnil e) ⇒ t1
| vcons x n' v' ⇒ fun (H : S n' = n)
  (Hv : JMeq (vcons x n' v') e) ⇒ t2
end (refl_equal n)(JMeq_refl e)
```

- 1 RUSSELL
 - Subset Coercions: A Simple Idea
 - Metatheory
 - Interpretation in COQ

- 2 PROGRAM
 - Extensions
 - Hello World

- 3 Finger Trees
 - Finger Trees in HASKELL
 - Dependent Finger Trees
 - Specializations
 - Ropes
 - Random-access sequences

- 4 Conclusion

A complex datastructure, with a reference implementation in `HASKELL` (Hinze & Paterson, JFP 2006).

- ▶ Using `PROGRAM` and type classes, we construct a faithful port which is proved **safe** (terminating and invariant preserving).

Dependent Finger Trees

A complex datastructure, with a reference implementation in HASKELL (Hinze & Paterson, JFP 2006).

- ▶ Using PROGRAM and type classes, we construct a faithful port which is proved **safe** (terminating and invariant preserving).
- ▶ Through **dependent types**, it provides guarantees to be used by higher-level structures.

Dependent Finger Trees

A complex datastructure, with a reference implementation in `HASKELL` (Hinze & Paterson, JFP 2006).

- ▶ Using `PROGRAM` and type classes, we construct a faithful port which is proved **safe** (terminating and invariant preserving).
- ▶ Through **dependent types**, it provides guarantees to be used by higher-level structures.
- ▶ **Instantiated** to (safe) ropes whose extraction has reasonable performance.

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

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`

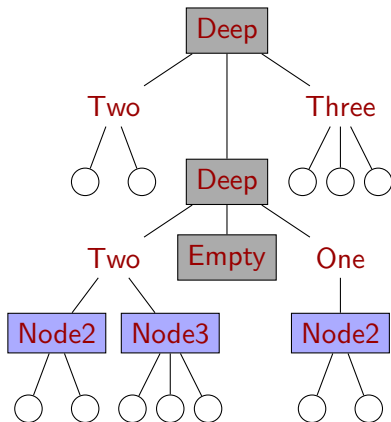
The Big Finger Tree Picture

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)



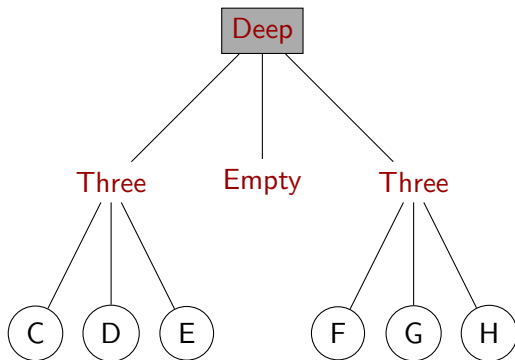
Operating on a Finger Tree

`add_left` :: $a \rightarrow \text{FingerTree } a \rightarrow \text{FingerTree } a$

`add_left` a `Empty` = `Single` a

`add_left` a (`Single` b) = `Deep` (`One` a) `Empty` (`One` b)

`add_left` a (`Deep` pr mf) = ...



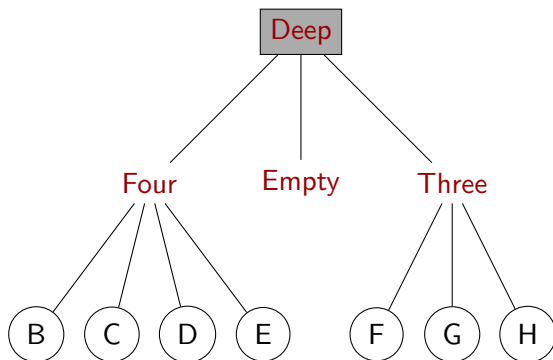
Operating on a Finger Tree

`add_left` :: $a \rightarrow \text{FingerTree } a \rightarrow \text{FingerTree } a$

`add_left` a `Empty` = `Single` a

`add_left` a (`Single` b) = `Deep` (`One` a) `Empty` (`One` b)

`add_left` a (`Deep` pr m sf) = ...



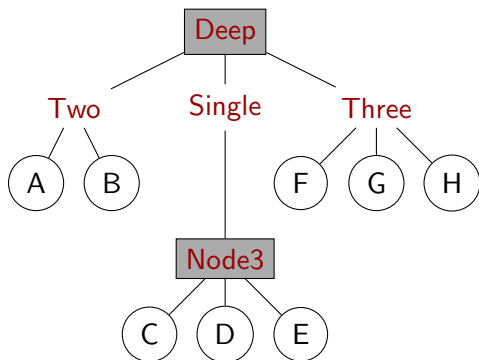
Operating on a Finger Tree

`add_left` :: $a \rightarrow \text{FingerTree } a \rightarrow \text{FingerTree } a$

`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
```

Adding cached measures

```
class Monoid v ⇒ Measured v a where  
  ||-|| :: a → v  
instance (Measured v a) ⇒ Measured v (Digit a) where ...
```

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`

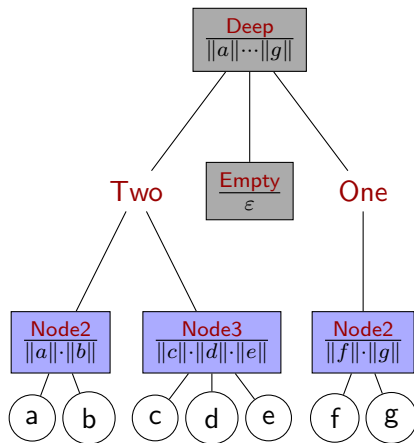
`| Single a`

`| Deep v`

`(Digit a)`

`(FingerTree v (Node v a))`

`(Digit a)`



- 1** RUSSELL
 - Subset Coercions: A Simple Idea
 - Metatheory
 - Interpretation in COQ
- 2** PROGRAM
 - Extensions
 - Hello World
- 3** Finger Trees
 - Finger Trees in HASKELL
 - **Dependent Finger Trees**
 - Specializations
 - Ropes
 - Random-access sequences
- 4** Conclusion

Why do this ?

- ▶ Generally useful, non-trivial structure
- ▶ Abstraction and specification power needed to ensure coherence of measures

Variable A : Type.

Inductive digit : Type :=

| One : $A \rightarrow \text{digit}$

| Two : $A \rightarrow A \rightarrow \text{digit}$

| Three : $A \rightarrow A \rightarrow A \rightarrow \text{digit}$

| Four : $A \rightarrow A \rightarrow A \rightarrow A \rightarrow \text{digit}$.

Definition $\text{full } x$:=

 match x with Four _ _ _ _ \Rightarrow True | _ \Rightarrow 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.
```

```
Variables (v : Type) (mono : monoid v).  
Variables (A : Type) (measure : A → v).
```

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

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

Inductive $node : \text{Type} :=$

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

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

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

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

Inductive $node : \text{Type} :=$

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

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

Program Definition $node2 (x y : A) : node :=$

$Node2 \ x \ y \ (\| x \| \cdot \| y \|).$

Program Definition $node_measure (n : node) : v :=$

$match \ n \ with \ Node2 \ _ \ _ \ s \Rightarrow s \mid \ Node3 \ _ \ _ \ _ \ s \Rightarrow s \ end.$

Dependent Finger Trees

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

| `Empty` : `fingertree A`

| `Single` : $\forall x : A, \text{fingertree } A$

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

`node` : $\forall (A : \text{Type}) (\text{measure} : A \rightarrow \nu), \text{Type}$

Dependent Finger Trees

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

| `Empty` : `fingertree A measure`

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

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

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

`node_measure` $A (measure : A \rightarrow v) : \text{node } A \text{ measure} \rightarrow v$

Dependent Finger Trees

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

Adding to the left

```
fix add_left A (measure : A → v)
  (a : A) (s : v) (t : fingertree measure s) {struct t} :
  fingertree measure (measure a · s) :=
```

Adding to the left

```
fix 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 ← measure a = measure a · ε
  | Single b ⇒ Deep (One a) Empty (One b)
  | Deep pr st' t' sf ⇒
    ...
end.
```

Adding to the left

```
fix 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 ← 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 measure c d e) t' in
      Deep (Two a b) sub sf
    | x ⇒ Deep (add_digit_left a pr) t' sf
  end
end.
```

Two hundred lines, one hundred obligations concatenation

```
def app (A : Type) (measure : A → v)
  (xs : v) (x : fingertree measure xs)
  (ys : v) (y : fingertree measure ys) :
  fingertree measure (xs · ys).
```

Splitting nodes

```
def splitNode (p : v → bool) (i : v)
  (n : node A measure) :
  { (l, x, r) : option (digit A) × A × option (digit A) |
    let ls := option_digit_measure measure l in
    let rs := option_digit_measure measure r in
    node_measure n = ls · || x || · rs ∧
    (l = None ∨ p (i · ls) = false) ∧
    (r = None ∨ p (i · ls · || x ||) = true)} := ...
```

```
Program Definition split_node (p : v → bool) i (n : node A) :
  { (l, x, r) : option (digit A) × A × option (digit A) |
    || n || = || l || · || x || · || r || ∧
    (l = None ∨ p (i · || l ||) = false) ∧
    (r = None ∨ p (i · || l || · || x ||) = true)} := ...
```

- ▶ We proved that all the functions from the original paper:
 - ▶ are terminating and total
 - ▶ respect the measures
 - ▶ respect the invariants given in the paper

	HASKELL	PROGRAM		
	Lines	L.o.C.	Obls	L.o.P.
app	200	200	100	auto
split	20	30	14	200
FingerTree	650	700	n.a.	400

- ▶ We proved that all the functions from the original paper:
 - ▶ are terminating and total
 - ▶ respect the measures
 - ▶ respect the invariants given in the paper

	HASKELL	PROGRAM		
	Lines	L.o.C.	Obls	L.o.P.
app	200	200	100	auto
split	20	30	14	200
FingerTree	650	700	n.a.	400

- ▶ Non-dependent interface, specializations
- ▶ A version with modules for a better extraction to OCaml

- 1** RUSSELL
 - Subset Coercions: A Simple Idea
 - Metatheory
 - Interpretation in COQ
- 2** PROGRAM
 - Extensions
 - Hello World
- 3** Finger Trees
 - Finger Trees in HASKELL
 - Dependent Finger Trees
 - Specializations
 - Ropes
 - Random-access sequences
- 4** Conclusion

Ropes on top of Finger Trees

Ingredients:

- ▶ $A := \text{string} \times \text{int} \times \text{int}$ (substrings)
- ▶ $v := \text{int}$ (the length, computationally relevant)
- ▶ $\text{measure}(\text{str}, \text{start}, \text{len}) := \text{len}$
- ▶ $\text{mono} := (0, +)$
- ▶ Implement `substring`, `get`

Ropes on top of Finger Trees

Ingredients:

- ▶ $A := \text{string} \times \text{int} \times \text{int}$ (substrings)
- ▶ $v := \text{int}$ (the length, computationally relevant)
- ▶ $\text{measure}(\text{str}, \text{start}, \text{len}) := \text{len}$
- ▶ $\text{mono} := (0, +)$
- ▶ Implement `substring`, `get`

⇒ **Extracted** code comparable to an optimized rope implementation.

Ropes on top of Finger Trees

Ingredients:

- ▶ $A := \text{string} \times \text{int} \times \text{int}$ (substrings)
- ▶ $v := \text{int}$ (the length, computationally relevant)
- ▶ $\text{measure}(\text{str}, \text{start}, \text{len}) := \text{len}$
- ▶ $\text{mono} := (0, +)$
- ▶ Implement `substring`, `get`

⇒ **Extracted** code comparable to an optimized rope implementation.

Relies on implicit invariants of the monoid, measure and code.

```
def below  $i := \{ x : \text{nat} \mid x < i \}$ .
```

```
def v :=  $\{ i : \text{nat} \ \& \ (\text{below } i \rightarrow A) \}$ .
```

```
def below  $i := \{ x : \text{nat} \mid x < i \}$ .
```

```
def  $v := \{ i : \text{nat} \ \& \ (\text{below } i \rightarrow A) \}$ .
```

```
def epsilon :  $v := 0 \prec (\text{fun } \_ \Rightarrow !)$ .
```

```
def append ( $n, fx$ ) ( $m, fy$ ) :  $v :=$   
  ( $n + m$ )  $\prec$   
    ( $\text{fun } i \Rightarrow \text{if } \text{lt\_ge\_dec } i \ n \ \text{then } fx \ i \ \text{else } fy \ (i - n)$ ).
```

```
def measure (x : A) : v := 1 < (fun _ => x).
```

```
def seq (x : v) := fingertree seqMonoid measure x.
```

```
def measure (x : A) : v := 1 < (fun _ => x).
```

```
def seq (x : v) := fingertree seqMonoid measure x.
```

```
tail n f : seq (n < f) → seq (pred n < (fun i => f (S i)))
```

```
app n fx m fy : seq (n < fx) → seq (m < fy) →
```

```
seq (n + m < (fun i => if lt_ge_dec i n then fx i else fy (i - n)))
```

The sequence and its operations

```
fix make (i : nat) (v : A) { struct i } : seq (i < (fun _ => v)).
```

The sequence and its operations

```
fix make (i : nat) (v : A) { struct i } : seq (i < (fun _ => v)).  
def get (i : nat) (m : below i → A)  
  (x : seq (i < m)) (j : below i) : { value : A | value = m j }.
```

The sequence and its operations

```
fix make (i : nat) (v : A) { struct i } : seq (i < (fun _ => v)).
```

```
def get (i : nat) (m : below i → A)  
  (x : seq (i < m)) (j : below i) : { value : A | value = m j }.
```

```
def set (i : nat) (m : below i → A)  
  (x : seq (i < m)) (j : below i) (value : A)  
  : seq (i < (fun idx => if idx = j then value else m idx)).
```

The sequence and its operations

```
fix make (i : nat) (v : A) { struct i } : seq (i < (fun _ => v)).
```

```
def get (i : nat) (m : below i → A)  
  (x : seq (i < m)) (j : below i) : { value : A | value = m j }.
```

```
def set (i : nat) (m : below i → A)  
  (x : seq (i < m)) (j : below i) (value : A)  
  : seq (i < (fun idx => if idx = j then value else m idx)).
```

- ▶ **Modularity**: only the specifications are used!
- ▶ **Efficiency**: Irrelevance of m currently not specified

Program Lemma `get_set` $i\ m\ (x : \text{seq } (i < m))\ (j : \text{below } i)$
 $(value : A) : value = \text{get } (\text{set } x\ j\ value)\ j.$

Program Lemma `get_set_diff` $i\ m\ (x : \text{seq } (i < m))$
 $(j : \text{below } i)\ (value : A)\ (k : \text{below } i) :$
 $j \neq k \rightarrow \text{get } x\ k = \text{get } (\text{set } x\ j\ value)\ k.$

- ✓ A certified implementation of Finger Trees and random-access sequences.
- ✓ PROGRAM scales, thanks to the phase distinction.
- ✓ Typeclasses to solve the overloading issues.

- 1 RUSSELL
 - Subset Coercions: A Simple Idea
 - Metatheory
 - Interpretation in COQ
- 2 PROGRAM
 - Extensions
 - Hello World
- 3 Finger Trees
 - Finger Trees in HASKELL
 - Dependent Finger Trees
 - Specializations
 - Ropes
 - Random-access sequences
- 4 Conclusion

- ▶ We studied a more **flexible** programming language, RUSSELL, providing a new **formal justification** of “*Predicate subtyping*” à la PVS, in a system with proof terms.

- ▶ We studied a more **flexible** programming language, RUSSELL, providing a new **formal justification** of “*Predicate subtyping*” à la PVS, in a system with proof terms.
- ▶ We implemented the PROGRAM tool to ease **programming** in COQ using the **full language** ($\sim 10\text{k}$ lines of OCAML).

- ▶ We studied a more **flexible** programming language, RUSSELL, providing a new **formal justification** of “*Predicate subtyping*” à la PVS, in a system with proof terms.
- ▶ We implemented the PROGRAM tool to ease **programming** in COQ using the **full language** ($\sim 10\text{k}$ lines of OCAML).
- ▶ We experimented it on a **verified** implementation of Finger Trees and ropes ($\sim 4\text{k}$ lines of literate COQ).

A **methodology** to build certified code in Coq, distributed since 2005:

- ▶ Already experimented by others, e.g. the Ynot project (Nanevsky, Morrisett & Birkedal) implementing Hoare Type Theory on top of Coq.
- ▶ Independent of the theory, can be ported to other systems: Matita (by Enrico Tassi), Agda, Epigram.

- ▶ We hinted at the important **foundational issues** with η -rules and proof-irrelevance that need to be solved (with Benjamin Werner).
- ▶ **Automation** for discharging proof-obligations (Sean Wilson at Edinburgh).
- ▶ Handling **truly dependent** pattern-matching by an elaboration.

Programming with Dependent Types in Coq

MATTHIEU SOZEAU

LRI, Univ. Paris-Sud - DÉMONS Team & INRIA Saclay - PROVAL Project

PPS Seminar
February 26th 2009
Paris, France

Try



8.2!

