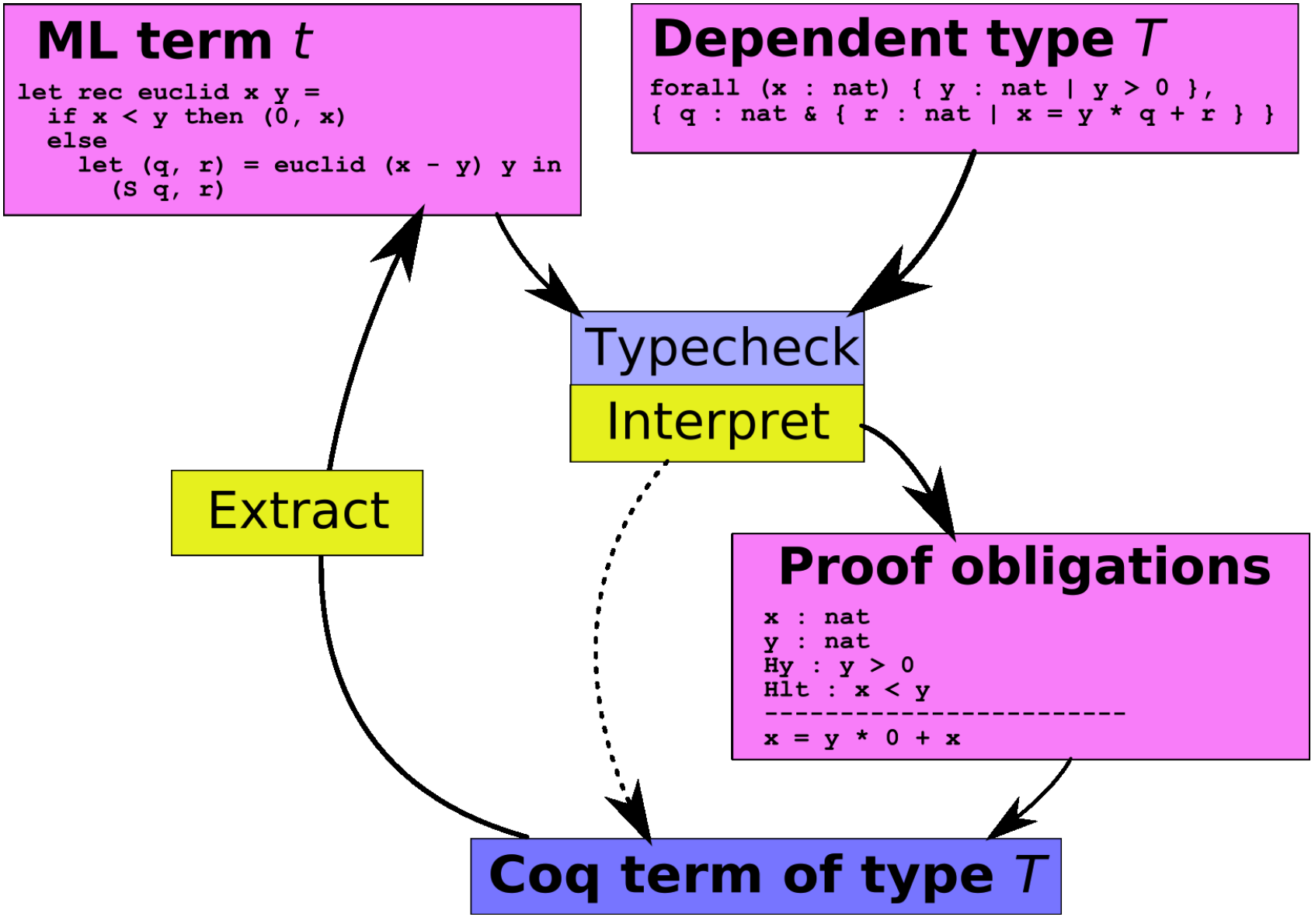
A journey with Russell: Programing Dependent Finger Trees

Matthieu Sozeau

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

TYPES'07
2-5 May 2007
Cividale del Friuli, Italy

Flashback


ML term t
let rec euclid x y =
if x < y then (0, x)
else
let (q, r) = euclid (x - y) y in
(S q, r)
Dependent type T
forall (x : nat) { y : nat | y > 0 },
{ q : nat & { r : nat | x = y * q + r } }
Typecheck
Interpret
Extract
Proof obligations
x : nat
y : nat
Hy : y > 0
Hlt : x < y
---
x = y * 0 + x
Coq term of type T

What I did since last summer

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

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

.

What I did since last summer

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

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

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

What I did since last summer

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

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

Formalization in Coq, SR using (Adams, JFP 2006, 16:2, pp. 219-246).

What I did since last summer

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

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

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

.

What I did since last summer

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

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

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

.

$$\frac{\Gamma \vdash v : \mathsf{vector}\ (n + 0) \qquad \Gamma \vdash \mathsf{vector}\ (n + 0) \rhd \mathsf{vector}\ n : \texttt{Type}}{\Gamma \vdash v : \mathsf{vector}\ n}$$

What I did since last summer

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

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

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

.

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

What I did since last summer

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

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

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

.

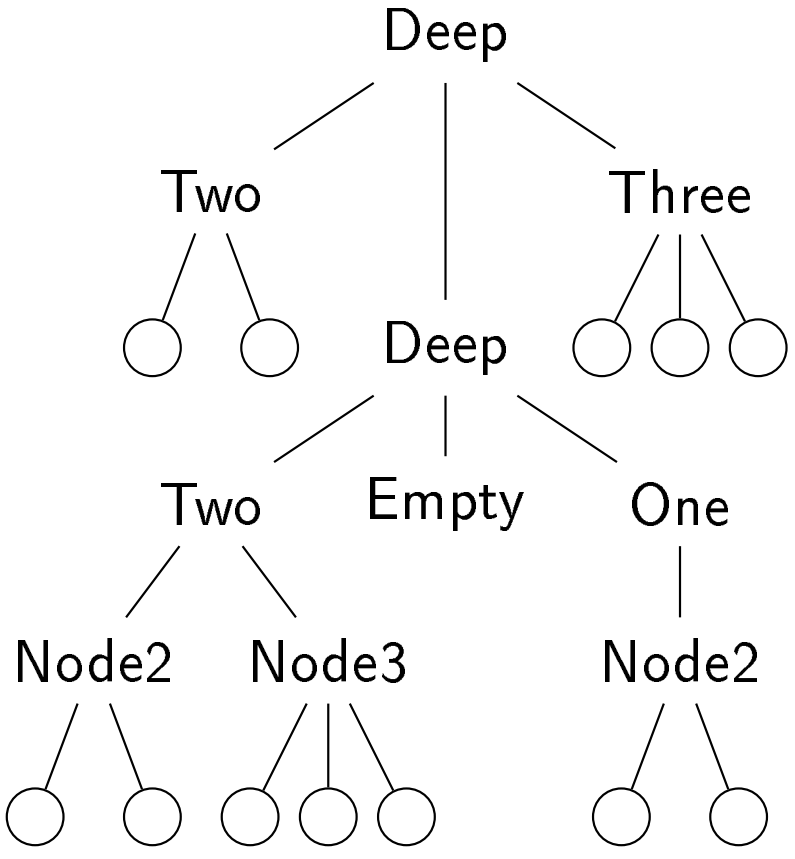
```
match e as t          return t = e → T with
| S n ⇒               fun (H : S n = e) ⇒ t1
| 0 ⇒                 fun (H : 0 = e) ⇒ t2
end                   (refl_equal e)
```

Outline

A quick tour of Finger Trees

- A simple general purpose data structure (Hinze & Paterson, JFP 2006, 16:2, pp. 197-217) ;
- Purely functional, nested datatype ;
- General purpose, parameterized data structure (gives sequences, priority queues, search trees...) ;
- Efficient deque operations (amortized constant time) and concatenation and splitting (logarithmic in smallest argument's size).
- Simple implementation compared to Kaplan & Tarjan's catenable dequeues, efficient in practice.

The Big Finger Tree Picture



A *nested* datatype and its companions

```
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)
```

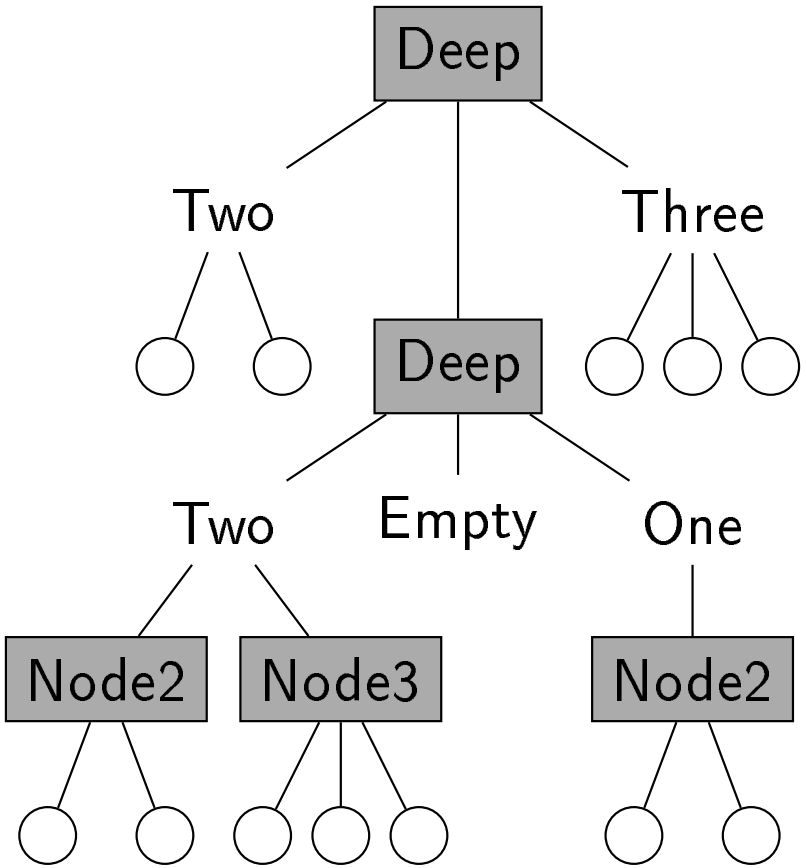
A *nested* datatype and its companions

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

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)
```

A sample FingerTree

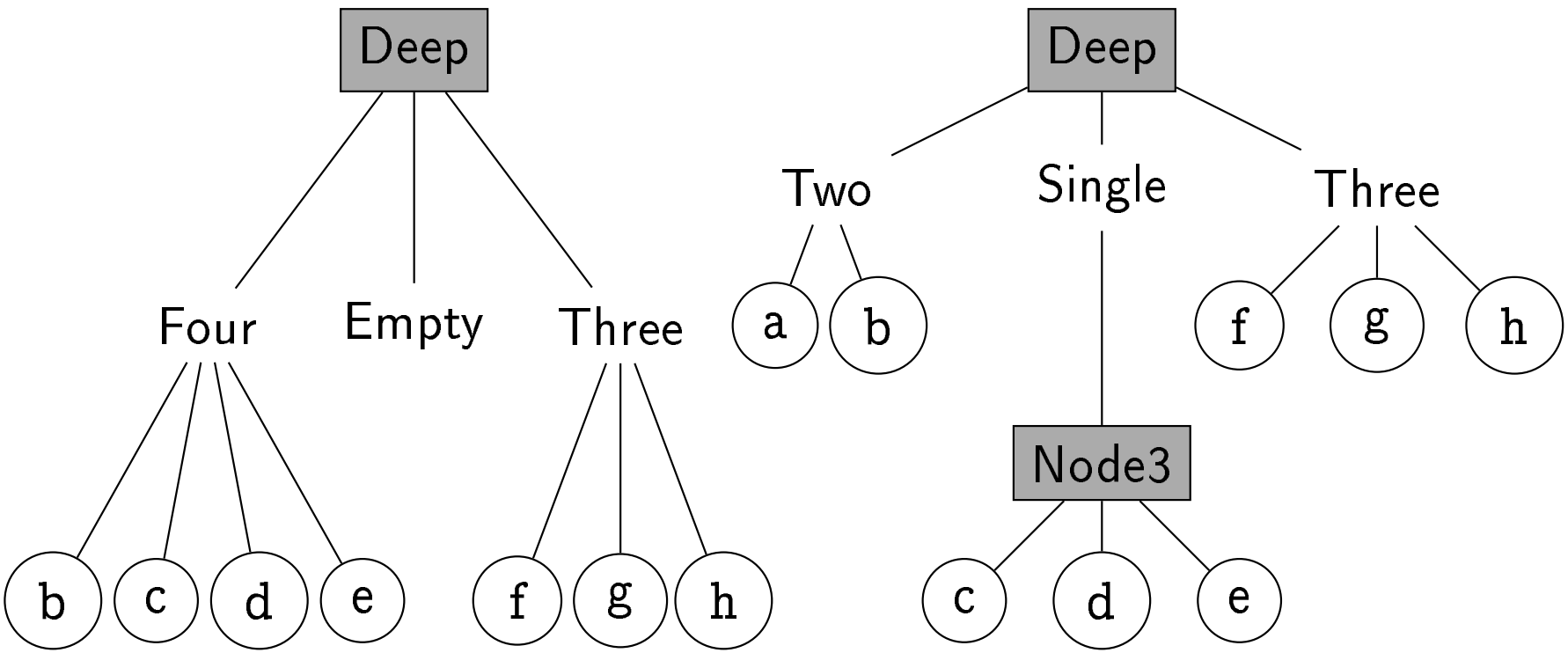


Operating on a Finger Tree

$$
\begin{aligned}
&\mathit{add\_left} :: a \to \mathsf{FingerTree}\ v\ a \to \mathsf{FingerTree}\ v\ a \\
&\mathit{add\_left}\ a\ \mathsf{Empty} = \mathsf{Single}\ a \\
&\mathit{add\_left}\ a\ (\mathsf{Single}\ b) = \mathsf{Deep}\ (\mathsf{One}\ a)\ \mathsf{Empty}\ (\mathsf{One}\ b) \\
&\mathit{add\_left}\ a\ (\mathsf{Deep}\ v\ \mathit{pr}\ m\ \mathit{sf}) = \ldots
\end{aligned}
$$

Operating on a Finger Tree

$$add\_left :: a \to \text{FingerTree } v\ a \to \text{FingerTree } v\ a$$
$$add\_left\ a\ \text{Empty} = \text{Single } a$$
$$add\_left\ a\ (\text{Single } b) = \text{Deep (One } a)\ \text{Empty (One } b)$$
$$add\_left\ a\ (\text{Deep } v\ pr\ m\ sf) = \ldots$$



A good set of primitive operations

$$\texttt{class Monoid } v \Rightarrow \text{Measured } v\ a\ \texttt{where}$$
$$\quad \mathit{measure} :: a \to v$$

$$\mathit{add\_left}, \mathit{add\_right},$$
$$\mathit{head}, \mathit{last} :: \text{Measured } v\ a \Rightarrow \text{FingerTree } v\ a \to a$$
$$\mathit{tail}, \mathit{liat} :: \text{Measured } v\ a \Rightarrow \text{FingerTree } v\ a \to \text{FingerTree } v\ a$$

A good set of primitive operations

```
class Monoid v ⇒ Measured v a where
  measure :: a → v
```

$add\_left, add\_right,$
$head, last$ :: Measured v $a \Rightarrow$ FingerTree v $a \rightarrow a$
$tail, liat$ :: Measured v $a \Rightarrow$ FingerTree v $a \rightarrow$ FingerTree v a

app :: Measured v $a \Rightarrow$ FingerTree v $a \rightarrow$ FingerTree v $a \rightarrow$
  FingerTree v a

A good set of primitive operations

$$\texttt{class Monoid } v \Rightarrow \text{Measured } v\ a \texttt{ where}$$
$$\quad \mathit{measure} :: a \to v$$

$$\mathit{add\_left}, \mathit{add\_right},$$
$$\mathit{head}, \mathit{last} :: \text{Measured } v\ a \Rightarrow \text{FingerTree } v\ a \to a$$
$$\mathit{tail}, \mathit{liat} :: \text{Measured } v\ a \Rightarrow \text{FingerTree } v\ a \to \text{FingerTree } v\ a$$

$$\mathit{app} :: \text{Measured } v\ a \Rightarrow \text{FingerTree } v\ a \to \text{FingerTree } v\ a \to$$
$$\quad \text{FingerTree } v\ a$$

$$\mathit{split} :: \text{Measured } v\ a \Rightarrow \text{FingerTree } v\ a \to (v \to \text{bool}) \to v \to$$
$$\quad \text{Maybe (FingerTree } v\ a) * a * \text{Maybe (FingerTree } v\ a)$$

1. Original Finger Trees
   - An introduction
   - In Haskell

2. Dependent Finger Trees
   - Digits & Nodes
   - A nested indexed inductive datatype
   - Instantiation: random-access sequences

Digits

```
Section Digit.
  Variable A : Type.

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

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

Digits 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

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

Notation "'ε'" := mempty mono.
Infix "·" := mappend mono.
```

Nodes

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

Notation "'ε'" := mempty mono.
Infix "·" := mappend mono.

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

Nodes cont'd

$$\texttt{Inductive node : Type} :=$$
$$\mid \text{Node2} : \forall\ x\ y, \{\ s : v \mid s = \|\ x\ \| \cdot \|\ y\ \|\ \} \to \text{node}$$
$$\mid \text{Node3} : \forall\ x\ y\ z, \{\ s : v \mid s = \|\ x\ \| \cdot \|\ y\ \| \cdot \|\ z\ \|\ \} \to \text{node}.$$

Nodes cont'd

```
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),
  fingertree (node A) →
  ∀ (r : digit A),
  fingertree A.
```

Dependent Finger Trees

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

  node_measure A (measure : A → v) : node A measure → v
```

A nested indexed inductive datatype

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

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

Adding to the left

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

Concatenation

Two hundred lines, one hundred obligations concatenation

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

Splitting nodes

```
Program Definition splitNode (p : v → bool) (i : v)
  (n : node 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)} := ...
```

A “simple” version

```
Program Definition sizeMonoid :=
  mkMonoid (mempty:=0) (mappend:=plus) _ _ _.

Program Definition measure (x : A) : v := 1.
```

A "simple" version

```
Program Definition sizeMonoid :=
  mkMonoid (mempty:=0) (mappend:=plus) _ _ _.

Program Definition measure (x : A) : v := 1.

Definition seq :=
  { n : nat & fingertree sizeMonoid measure n }.
```

A "simple" version

```
Program Definition sizeMonoid :=
  mkMonoid (mempty:=0) (mappend:=plus) _ _ _.

Program Definition measure (x : A) : v := 1.

Definition seq :=
  { n : nat & fingertree sizeMonoid measure n }.

Definition length (x : seq) := let (s, seq) := x in s.
```

A "simple" version

```
Program Definition sizeMonoid :=
  mkMonoid (mempty:=0) (mappend:=plus) _ _ _.

Program Definition measure (x : A) : v := 1.

Definition seq :=
  { n : nat & fingertree sizeMonoid measure n }.

Definition length (x : seq) := let (s, seq) := x in s.

Program Definition nth x (i : nat | i < length x) : A :=
  dest x as existT s t in
  dest split_tree (lt i) 0 t as mkTreeSplit _ _ x _ _ in x.
```

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

A dependent version: the measure gives the semantics

$$\texttt{Definition } \mathit{below}\ i := \{\ x : \mathsf{nat} \mid x < i\ \}.$$

$$\texttt{Definition } v := \{\ i : \mathsf{nat}\ \&\ (\mathit{below}\ i \to A)\ \}.$$

A dependent version: the measure gives the semantics

```
Definition below i := { x : nat | x < i }.

Definition v := { i : nat & (below i → A) }.

Program Definition ε : v := 0 ≺ (fun _ ⇒ !).
Program Definition append (xs ys : v) : v := ···

Program Definition seqMonoid : monoid v :=
  mkMonoid (mempty:=ε) (mappend:=append) _ _ _.
```

A dependent version: the measure gives the semantics

```
Definition below i := { x : nat | x < i }.

Definition v := { i : nat & (below i → A) }.

Program Definition ε : v := 0 ≺ (fun _ ⇒ !).
Program Definition append (xs ys : v) : v := ···

Program Definition seqMonoid : monoid v :=
  mkMonoid (mempty:=ε) (mappend:=append) _ _ _.

Program Definition measure (x : A) : v := 1 ≺ (fun _ ⇒ x).
```

The sequence and its operations

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

Program Fixpoint make (i : nat) (v : A) { struct i } :
  seq (i ≺ (fun _ ⇒ v)).
```

The sequence and its operations

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

Program Fixpoint make (i : nat) (v : A) { struct i } :
  seq (i ≺ (fun _ ⇒ v)).

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

The sequence and its operations

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

Program Fixpoint make (i : nat) (v : A) { struct i } :
  seq (i ≺ (fun _ ⇒ v)).

Program Definition get (i : nat) (m : below i → A)
  (x : seq (i ≺ m)) (j : below i) : { value : A | m j = value }.

Program Definition set (i : nat) (m : below i → A)
  (x : seq (i ≺ m)) (j : below i) (value : A)
  : seq (i ≺ (fun idx : below i ⇒
    if eq_nat_dec idx j then value else m idx)).
```

Theorems for free!

$$\texttt{Program Lemma } \mathit{get\_set}\ i\ m\ (x : \mathit{seq}\ (i \prec m))\ (j : \mathit{below}\ i)\ (\mathit{value} : A) : \mathit{value} = \mathit{get}\ (\mathit{set}\ x\ j\ \mathit{value})\ j.$$

Theorems for free!

$$\texttt{Program Lemma } \mathit{get\_set}\ i\ m\ (x : \mathit{seq}\ (i \prec m))\ (j : \mathit{below}\ i)\ (\mathit{value} : A) : \mathit{value} = \mathit{get}\ (\mathit{set}\ x\ j\ \mathit{value})\ j.$$

$$\texttt{Program Lemma } \mathit{get\_set\_diff}\ i\ m\ (x : \mathit{seq}\ (i \prec m))\ (j : \mathit{below}\ i)\ (\mathit{value} : A)\ (k : \mathit{below}\ i) : (j : \mathsf{nat}) \neq k \to (\mathit{get}\ x\ k : A) = \mathit{get}\ (\mathit{set}\ x\ j\ \mathit{value})\ k.$$

Finger Trees in Russell

The development

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

Finger Trees in Russell

The development

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

Conclusions

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

The End

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

Dependent Inductive coercion rule

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

$$\frac{\dots \qquad \dfrac{\Gamma \vdash_{CCI} ?_{x=y} : x = y}{\Gamma \vdash_{CCI} \mathit{eq\_rect}\ \mathbb{N}\ x\ \mathsf{vector}\ \bullet\ y\ ?_{x=y} : \mathsf{vector}\ x \rhd \mathsf{vector}\ y}}{\Gamma \vdash_{CCI} \mathit{eq\_rect}\ \mathbb{N}\ x\ \mathsf{vector}\ \nu\ y\ ?_{x=y} : \mathsf{vector}\ y}$$

$$\mathtt{eq\_rect} : \forall (A : \mathtt{Type})(x : A)(P : A \to \mathtt{Type}), P\ x \to \forall y, x = y \to P\ y$$