PROGRAM-ing Finger Trees In COQ or How To Morph Endo Using Type Theory

MATTHIEU SOZEAU

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

ICFP'07
October 1–3 2007
Freiburg, Germany

The Curry-Howard isomorphism

Programming language = Proof system

The Curry-Howard isomorphism

Programming language = Proof system

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

The Curry-Howard isomorphism

Programming language = Proof system

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

Epigram **PVS** **DML** **Ωmega**

The Curry-Howard isomorphism

Programming language = Proof system

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

- **Logical Framework** Type Theory.

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

The Curry-Howard isomorphism

Programming language = Proof system

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

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

~~**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.
- **Paradigm** Purely functional.

.

Epigram **PVS** ~~**DML**~~ **Ωmega**

The Curry-Howard isomorphism

Programming language = Proof system

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

- **Logical Framework** Type Theory.
  Separates proofs and programs using sorts.
- **Paradigm** Purely functional.
  No separation of terms and types.

Epigram PVS ~~DML~~ ~~Ωmega~~

The Curry-Howard isomorphism

Programming language = Proof system

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

- **Logical Framework** Type Theory.
  Separates proofs and programs using sorts.
- **Paradigm** Purely functional.
  No separation of terms and types.
- **Proof automation** Semi-automatic, using tactics.

~~Epigram~~ PVS ~~DML~~ ~~Ωmega~~

The Curry-Howard isomorphism

Programming language = Proof system

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

- **Logical Framework** Type Theory.
  Separates proofs and programs using sorts.
- **Paradigm** Purely functional.
  No separation of terms and types.
- **Proof automation** Semi-automatic, using tactics.
- **Phase distinction** none

Epigram ~~**PVS**~~ **DML** **Ωmega**

The Curry-Howard isomorphism

Programming language = Proof system

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

- **Logical Framework** Type Theory.
  Separates proofs and programs using sorts.
- **Paradigm** Purely functional.
  No separation of terms and types.
- **Proof automation** Semi-automatic, using tactics.
- **Phase distinction** $\Rightarrow$ in Program

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

PROGRAM-ing with subsets

$$\texttt{Fixpoint}\ \mathit{div}\ (a : \mathbf{nat})\ (b : \mathbf{nat} \mid b \neq 0)\ \{\ \texttt{wf}\ \mathit{lt}\ \} :$$
$$\{\ (q, r) : \mathbf{nat} \times \mathbf{nat} \mid a = b \times q + r \wedge r < b\ \} :=$$
$$\texttt{if}\ \mathit{less\_than}\ a\ (\mathit{proj}\ b)\ \texttt{then}\ ((0, a), ?)$$
$$\texttt{else}\ \texttt{dest}\ \mathit{div}\ (a\ \text{-}\ \mathit{proj}\ b)\ b\ \texttt{as}\ (q', r)\ \texttt{in}\ ((\mathrm{S}\ q', r), ?).$$

where:

$$\mathit{less\_than} : \forall\ x\ y : \mathbf{nat}, \{\ x < y\ \} + \{\ x \geq y\ \}$$

Program-ing with subsets

```
Program Fixpoint div (a : nat) (b : nat | b ≠ 0) { wf lt } :
  { (q, r) : nat × nat | a = b × q + r ∧ r < b } :=
  if less_than a b then (0, a)
  else dest div (a - b) b as (q', r) in (S q', r).
```

where:

less_than : ∀ x y : **nat**, { x < y } + { x ≥ y }

Enriched type equality

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

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

Outline

1 Program

2 Finger Trees

3 In Coq

4 A Rope for Endo

A quick tour of Finger Trees

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

The Big Finger Tree Picture

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

The Big Finger Tree Picture

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

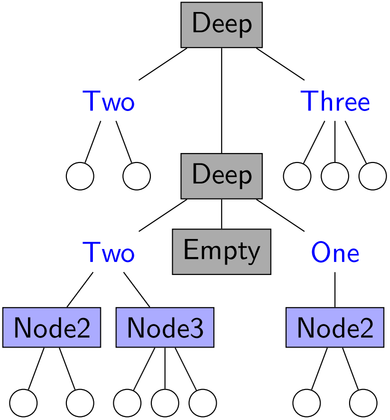
data **Node** a = Node2 a a | Node3 a a a

The Big Finger Tree Picture

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

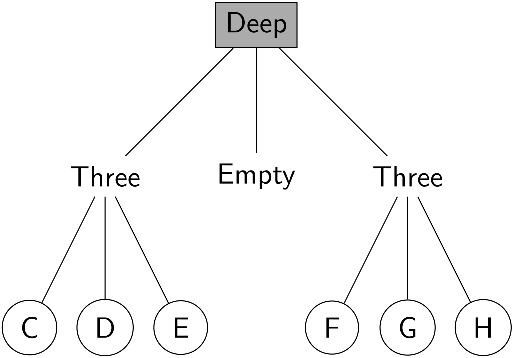
data **Node** *a* = Node2 *a a* | Node3 *a a a*

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



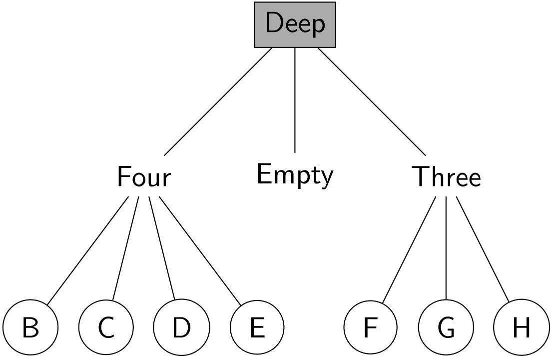
Operating on a Finger Tree

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



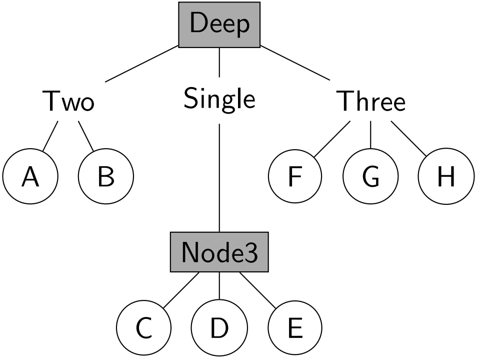
Operating on a Finger Tree

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



Operating on a Finger Tree

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



Adding cached measures

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

Adding cached measures

`class` **Monoid** $v \Rightarrow$ **Measured** $v\ a$ `where`
  $\|\_\| :: a \rightarrow v$

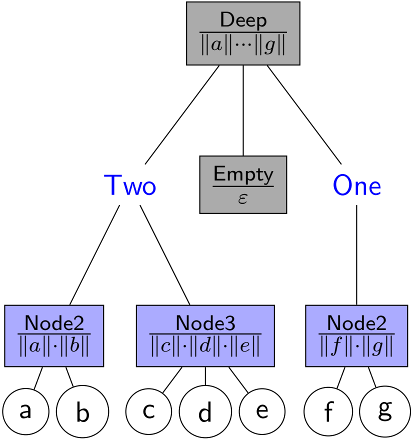
`instance` (**Measured** $v\ a$) $\Rightarrow$ **Measured** v (**Digit** a) `where` $\cdots$

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

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

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

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



Outline

1 Program

2 Finger Trees

3 In Coq

4 A Rope for Endo

Why do this ?

- Generally useful, non-trivial structure

Why do this ?

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

Why do this ?

- Generally useful, non-trivial structure
- Abstraction power needed to ensure coherence of measures
- Makes dependent types (subsets and indexed datatypes) shine

Why do this ?

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

Digits

```
Variable A : Type.

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

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

Digits cont'd

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

Next Obligation.
  intros ; simpl in n ; auto.
Qed.
```

Nodes

`Variables` $(v : \texttt{Type})$ $(\mathit{mono} : \textbf{monoid}\ v)$.
`Variables` $(A : \texttt{Type})$ $(\mathit{measure} : A \rightarrow v)$.

Nodes

$\texttt{Variables}\ (v : \texttt{Type})\ (\mathit{mono} : \textbf{monoid}\ v).$
$\texttt{Variables}\ (A : \texttt{Type})\ (\mathit{measure} : A \rightarrow v).$

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

Nodes

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

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

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

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

Dependent Finger Trees

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

| Empty : **fingertree** A

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

| Deep : $\forall$ (l : **digit** A) ($m : v$),
  fingertree (**node** A) $\rightarrow$
  $\forall$ (r : **digit** A),
  fingertree A.

node : $\forall$ (A : Type) (*measure* : $A \rightarrow v$), Type

Dependent Finger Trees

$\texttt{Inductive}$ **fingertree** $(A : \texttt{Type})\ (\mathit{measure} : A \rightarrow v) : \texttt{Type} :=$

$|$ Empty : **fingertree** A $\mathit{measure}$

$|$ Single : $\forall\ x : A,$ **fingertree** A $\mathit{measure}$

$|$ Deep : $\forall\ (l :$ **digit** $A)\ (m : v),$
fingertree (**node** A $\mathit{measure}$) $(\mathit{node\_measure}\ A\ \mathit{measure}) \rightarrow$
$\forall\ (r :$ **digit** $A),$
fingertree A $\mathit{measure}$.

$$\mathit{node} : \forall\ (A : \texttt{Type})\ (\mathit{measure} : A \rightarrow v),\ \texttt{Type}$$

$$\mathit{node\_measure}\ A\ (\mathit{measure} : A \rightarrow v) : \mathit{node}\ A\ \mathit{measure} \rightarrow v$$

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

$\texttt{Program Fixpoint}\ \mathit{add\_left}\ A\ (\mathit{measure} : A \rightarrow v)$
$(a : A)\ (s : v)\ (t : \textbf{fingertree}\ \mathit{measure}\ s)\ \{\texttt{struct}\ t\} :$
$\textbf{fingertree}\ \mathit{measure}\ (\mathit{measure}\ a \cdot s) :=$

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 ← 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 (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.
```

Summary

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

Summary

- 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	600	n.a.	400

- 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	600	n.a.	400

- Non-dependent interface, specializations

Summary

- 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	600	n.a.	400

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

Outline

1 Program

2 Finger Trees

3 In Coq

4 A Rope for Endo

Ropes on top of Finger Trees

Ingredients:

- A := **string** $\times$ **int** $\times$ **int**
- *measure* (*str*, *start*, *len*) := *len*
- v := **int**
- *mono* := $(0, +)$

Ropes on top of Finger Trees

Ingredients:

- $A :=$ **string** $\times$ **int** $\times$ **int**
- $measure\ (str, start, len) := len$
- $v :=$ **int**
- $mono := (0, +)$
- Implement *substring*, *get*

Ropes on top of Finger Trees

Ingredients:

- $A := \textbf{string} \times \textbf{int} \times \textbf{int}$
- $\mathit{measure}\ (\mathit{str}, \mathit{start}, \mathit{len}) := \mathit{len}$
- $v := \textbf{int}$
- $\mathit{mono} := (0, +)$
- Implement *substring*, *get*

Demo

Ingredients:

- $A :=$ **string** $\times$ **int** $\times$ **int**
- $measure\ (str, start, len) := len$
- $v :=$ **int**
- $mono := (0, +)$
- Implement *substring*, *get*

Demo

$\Rightarrow$ Extracted code comparable to an optimized rope implementation:

4min vs. 1min30 for the empty prefix.

Conclusions

- ✓ PROGRAM scales
- ✓ Subset types arise naturally
- ✓ Dependent types are a powerful and manageable tool, get some !
- × Difficulties with reasoning and computing

`lri.fr/~sozeau/research/russell/fingertrees.en.html`