PROGRAM-ing Dependent Finger Trees In COQ

MATTHIEU SOZEAU

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

DTP'08
February 18–20 2008
Nottingham, UK

PROGRAM-ing with subsets

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

where:

less_than : $\forall\ x\ y$: **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 : $\forall\, x\, y$: **nat**, $\{ x < y \} + \{ x \geq y \}$

Enriched type equality

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

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

Program-ing with inductive families

```
def head n (v : vector A (S n)) : A :=
  match v with
  | vcons a n' v' ⇒ a
  | vnil ⇒ !
  end

fix prod n (v : vector A n) (w : vector B n)
  : vector (A × B) n :=
  match v, w with
  | vnil, vnil ⇒ vnil
  | vcons a n' v', vcons b _ w' ⇒ vcons (a, b) (prod v' w')
  | _, _ ⇒ !
  end
```

Finger Trees highlights

- A useful, complex data structure formalized in a dependently-typed style
- Modular instantiation of the structure to get certified specializations.
- Reasonnably efficient extracted code.

Claims

- Useful phase distinction between programming and proving ;
- General methods to put logic in the terms are needed to reason in the abstract ;
- A flexible source language, a powerful elaboration.

Outline

1. Finger Trees

2. Dependent Finger Trees

3. Specializations
   - Ropes
   - Random-access sequences

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

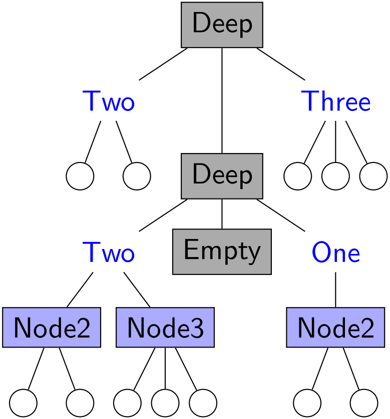
`data` **Node** a = Node2 a a | Node3 a a a

The Big Finger Tree Picture

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

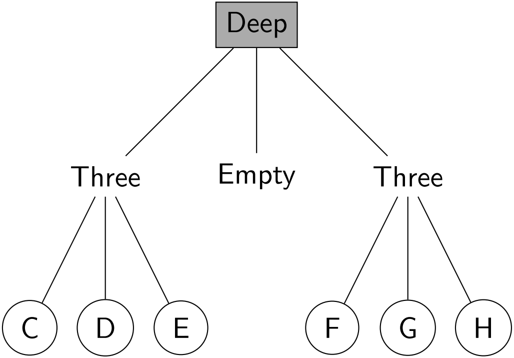
`data` **Node** *a* = Node2 *a a* | Node3 *a a a*

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

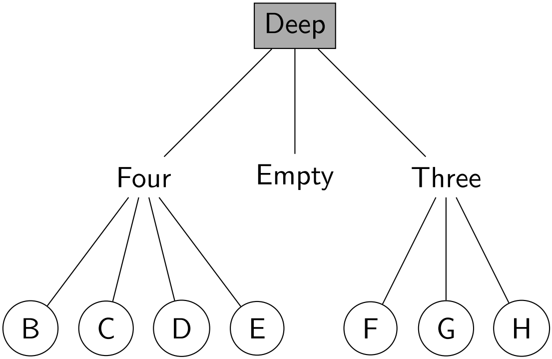


Operating on a Finger Tree

add_left :: $a \rightarrow$ **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) = ...

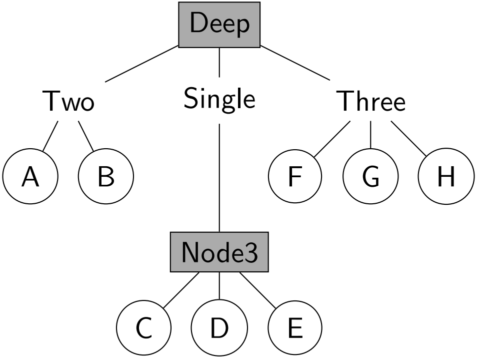


add_left :: $a \rightarrow$ **FingerTree** $a \rightarrow$ **FingerTree** a
add_left a Empty = Single a
add_left a (Single b) = Deep (One a) Empty (One b)
add_left a (Deep pr m sf) = ...



Operating on a Finger Tree

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



Adding cached measures

$\texttt{class}$ **Monoid** $v \Rightarrow$ **Measured** $v\ a$ $\texttt{where}$

$\quad \|\_\| :: a \rightarrow v$

Adding cached measures

$\texttt{class}$ **Monoid** $v \Rightarrow$ **Measured** $v\ a$ $\texttt{where}$
$\quad \|\_\| :: a \rightarrow v$

$\texttt{instance}$ (**Measured** $v\ a$) $\Rightarrow$ **Measured** v (**Digit** a) $\texttt{where} \cdots$

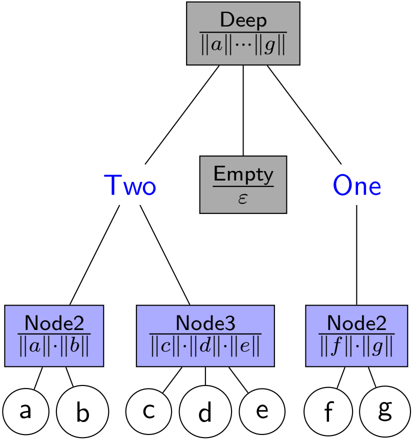
Adding cached measures

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

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

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

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



Outline

1. Finger Trees

2. Dependent Finger Trees

3. Specializations
   - Ropes
   - Random-access sequences

Why do this ?

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

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

```
def 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.
```

Nodes

Variables (v : Type) (*mono* : **monoid** v).
Variables (A : Type) (*measure* : $A \to v$).

Nodes

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

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

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.

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

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

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

node : ∀ (*A* : Type) (*measure* : *A* → *v*), Type

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) (m : v),
    fingertree (node A measure) (node_measure A measure) →
    ∀ (r : digit A),
    fingertree A measure.
```

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

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

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

$|$ Empty : **fingertree** $A\ measure\ \varepsilon$

$|$ Single : $\forall\ x : A,$ **fingertree** $A\ measure\ (measure\ x)$

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

Adding to the left

$\texttt{fix}$ **add_left** A $(measure : A \to v)$
$(a : A)$ $(s : v)$ $(t : \textbf{fingertree}\ measure\ s)$ $\{\texttt{struct}\ t\}$:
$\textbf{fingertree}\ measure\ (measure\ a \cdot s)$:=

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.
```

$\texttt{def}$ **app** $(A : \texttt{Type})\ (measure : A \rightarrow v)$
$(xs : v)\ (x : \textbf{fingertree}\ measure\ xs)$
$(ys : v)\ (y : \textbf{fingertree}\ measure\ ys) :$
$\textbf{fingertree}\ measure\ (xs \cdot ys)$.

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)} := ...
```

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

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

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 Finger Trees

2 Dependent Finger Trees

3 Specializations
- Ropes
- Random-access sequences

Ingredients:

- A := **string** × **int** × **int** (substrings)
- v := **int** (the length, computationally relevant)
- **measure** (*str*, *start*, *len*) := *len*
- **mono** := $(0, +)$
- Implement **substring**, **get**

Ropes on top of Finger Trees

Ingredients:

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

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

Ingredients:

- A := **string** × **int** × **int** (substrings)
- v := **int** (the length, computationally relevant)
- **measure** (*str*, *start*, *len*) := *len*
- **mono** := $(0, +)$
- Implement **substring**, **get**

⇒ Extracted code comparable to an optimized rope implementation.

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

Certified Sequences: the monoid

def **below** $i := \{\ x : \textbf{nat} \mid x < i\ \}$.

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

Certified Sequences: the monoid

$\texttt{def}$ **below** $i := \{ x : \textbf{nat} \mid x < i \}$.

$\texttt{def}$ **v** $:= \{ i : \textbf{nat} \ \& \ (\textbf{below}\ i \rightarrow A) \}$.

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

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

Instantiation

$\texttt{def}$ **measure** $(x : A) : \mathbf{v} := 1 \prec (\texttt{fun}\ \_ \Rightarrow x)$.

$\texttt{def}$ **seq** $(x : \mathbf{v}) :=$ **fingertree** *seqMonoid measure x*.

Instantiation

def **measure** $(x : A) : \mathbf{v} := 1 \prec (\texttt{fun}\ \_ \Rightarrow x)$.

def **seq** $(x : \mathbf{v}) :=$ **fingertree** *seqMonoid measure x*.

tail $n\ f : \mathbf{seq}\ (n \prec f) \rightarrow \mathbf{seq}\ (\mathrm{pred}\ n \prec (\texttt{fun}\ i \Rightarrow f\ (\mathrm{S}\ i)))$

app $n\ fx\ m\ fy : \mathbf{seq}\ (n \prec fx) \rightarrow \mathbf{seq}\ (m \prec fy) \rightarrow$
$\mathbf{seq}\ (n + m \prec (\texttt{fun}\ i \Rightarrow \texttt{if}\ \mathbf{lt\_ge\_dec}\ i\ n\ \texttt{then}\ fx\ i\ \texttt{else}\ fy\ (i - n)))$

The sequence and its operations

$\texttt{fix}$ **make** $(i : \mathbf{nat})$ $(\mathbf{v} : A)$ $\{ \, \mathit{struct} \ i \, \}$: seq $(i \prec (\texttt{fun} \ \_ \Rightarrow v))$.

The sequence and its operations

$\texttt{fix}\ \textbf{make}\ (i : \textbf{nat})\ (\mathbf{v} : A)\ \{\ \mathit{struct}\ i\ \} : \mathit{seq}\ (i \prec (\texttt{fun}\ \_ \Rightarrow v)).$

$\texttt{def}\ \textbf{get}\ (i : \textbf{nat})\ (m : \textbf{below}\ i \rightarrow A)$
$\quad (x : \mathit{seq}\ (i \prec m))\ (j : \textbf{below}\ i) : \{\ \mathit{value} : A \mid \mathit{value} = m\ j\ \}.$

fix **make** (i : **nat**) (**v** : A) { *struct* i } : seq ($i \prec$ (fun _ $\Rightarrow v$)).

def **get** (i : **nat**) (m : **below** $i \to A$)
(x : seq ($i \prec m$)) (j : **below** i) : { $value$: A | $value = m\ j$ }.

def **set** (i : **nat**) (m : **below** $i \to A$)
(x : seq ($i \prec m$)) (j : **below** i) ($value$: A)
: seq ($i \prec$ (fun $idx \Rightarrow$ if $idx = j$ then $value$ else $m\ idx$)).

$\texttt{fix}$ **make** $(i : \mathbf{nat})\ (\mathbf{v} : A)\ \{\ struct\ i\ \} : seq\ (i \prec (\texttt{fun}\ \_ \Rightarrow v)).$

$\texttt{def}$ **get** $(i : \mathbf{nat})\ (m : \mathbf{below}\ i \rightarrow A)$
$(x : seq\ (i \prec m))\ (j : \mathbf{below}\ i) : \{\ value : A \mid value = m\ j\ \}.$

$\texttt{def}$ **set** $(i : \mathbf{nat})\ (m : \mathbf{below}\ i \rightarrow A)$
$(x : seq\ (i \prec m))\ (j : \mathbf{below}\ i)\ (value : A)$
$: seq\ (i \prec (\texttt{fun}\ idx \Rightarrow \texttt{if}\ idx = j\ \texttt{then}\ value\ \texttt{else}\ m\ idx)).$

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

Theorems for free!

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

`Program Lemma` $\textbf{get\_set\_diff}\ i\ m\ (x : \textbf{seq}\ (i \prec m))$
$(j : \textbf{below}\ i)\ (value : A)\ (k : \textbf{below}\ i) :$
$j \neq k \rightarrow \textbf{get}\ x\ k = \textbf{get}\ (\textbf{set}\ x\ j\ value)\ k.$

Conclusions

- ✓ PROGRAM scales, thanks to the phase distinction.
- ✓ Use abstract indices !
- × Need more language technology, e.g: overloading
- × Difficulties with reasoning and computing.