Sharing in Ynot

Gregory Malecha* Greg Morrisett

Harvard University SEAS

January 15, 2010

Outline

Outline

Gaining Assurance

Observation

If there's one thing that we've learned in the past 20 years it's that all software has bugs.

- Tried and are trying a lot of approaches to mitigate this problem:
    - (Unit) Testing
    - Bug Finding Tools
    - Static Type Systems
    - Model Checking
    - Theorem Proving

Gaining Assurance

Observation

If there's one thing that we've learned in the past 20 years it's that all software has bugs.

- Tried and are trying a lot of approaches to mitigate this problem:
  - (Unit) Testing
  - Bug Finding Tools
  - Static Type Systems
  - Model Checking
  - Theorem Proving

The Burden of Proofs

- Several projects have worked on verification.
  - Jahob - Verification in Java
  - *Spec#* - Verification in C#
  - *seL4* - Kernel verification in Agda
- Standard approach to verification:
  1. Write specifications.
  2. Write all of the code.
  3. Give specifications and code to VC Generator.
  4. Modify code/add annotations until and repeat until verification succeeds.

Type-Oriented Specifications: ML

- Most type systems don't express side-effects explicitly.

```
(* swap : 'a ref -> 'a ref -> unit *)
```

Type-Oriented Specifications: ML

- Most type systems don't express side-effects explicitly.

```
(* swap : 'a ref -> 'a ref -> unit *)
let swap a b =
  let t = !a in
  a := !b ;
  b := t
```

- Simplifies coding.
- But the types don't tell us whether a function is really a function!

Explicit IO: Haskell

- Haskell makes side-effects explicit using monads.

```
swap :: MVar a -> MVar a -> IO ()
```

Explicit IO: Haskell

- Haskell makes side-effects explicit using monads.

```
swap :: MVar a -> MVar a -> IO ()
swap p1 p2 =
  do { t1 <- takeMVar p1
     ; t2 <- takeMVar p2
     ; putMVar p1 t2
     ; putMVar p2 t1
     }
```

- Can now determine if a function doesn't have side effects.
- Only looking at the type, we know more, but not enough.

Specifications: Ynot

- Hoare logic-based specifications using dependent types.
- Index the IO monad by pre- and post-conditions.
  - Allows us to precisely specify the effects of a computation.

```
Definition swap : forall (p1 p2 : ptr) (v1 v2 : [nat]),
  Cmd (v1 ~~ v2 ~~ p1 ~~> v1 * p2 ~~> v2)
      (fun _ : unit => v1 ~~ v2 ~~ p1 ~~> v2 * p2 ~~> v1).
```

DEMO

Specifications: Ynot

- Hoare logic-based specifications using dependent types.
- Index the `IO` monad by pre- and post-conditions.
  - Allows us to precisely specify the effects of a computation.

```
Definition swap : forall (p1 p2 : ptr) (v1 v2 : [nat]),
  Cmd (v1 ~~ v2 ~~ p1 ~~> v1 * p2 ~~> v2)
      (fun _ : unit => v1 ~~ v2 ~~ p1 ~~> v2 * p2 ~~> v1).
  refine (fun p1 p2 v1 v2 =>
    t1 <- ! p1 ;
    t2 <- ! p2 ;
    p1 ::= t2 ;;
    {{ p2 ::= t1 }});
  sep fail auto. (** Proof **)
Qed.
```

Overview

1. Logic
   - Shallow embedding of separation logic.
   - Computational irrelevance.
2. Monad
   - `Cmd` monad indexed by pre- and post-conditions.
3. Tactics
   - *Ltac* automation for separation logic.

Separation Logic

```
(** Predicates over heaps **)
Definition heap := ptr -> option Dyn.
Definition hprop := heap -> Prop.
```

Separation Logic

```
(** Predicates over heaps **)
Definition heap := ptr -> option Dyn.
Definition hprop := heap -> Prop.

Definition emp : hprop := fun h => forall p, h p = None.
```

Separation Logic

```
(** Predicates over heaps **)
Definition heap := ptr -> option Dyn.
Definition hprop := heap -> Prop.

Definition emp : hprop := fun h => forall p, h p = None.

Definition cell p v : hprop := fun h =>
  forall p', if p = p' then h p = Some v
             else           h p = None.
```

Separation Logic

```
(** Predicates over heaps **)
Definition heap := ptr -> option Dyn.
Definition hprop := heap -> Prop.

Definition emp : hprop := fun h => forall p, h p = None.

Definition cell p v : hprop := fun h =>
  forall p', if p = p' then h p = Some v
             else           h p = None.

Definition hprop_sep (P Q : hprop) : hprop :=
  fun h => exists h1 h2, h ~> h1 * h2 /\ P h1 /\ Q h2.
```

Ynot Library : Command Monad

```
Axiom Cmd : forall (pre : hprop) {A} (post : A -> hprop), Set.
```

Ynot Library : Command Monad

```
Axiom Cmd : forall (pre : hprop) {A} (post : A -> hprop), Set.

Axiom CmdBind : forall pre1 T1 (post1 : T1 -> hprop)
  pre2 T2 (post2 : T2 -> hprop)
  (st1 : Cmd pre1 post1)
  (_ : forall v, post1 v ==> pre2 v)
  (st2 : forall v : T1, Cmd (pre2 v) post2)
  : Cmd pre1 post2.
```

Ynot Library : Command Monad

```
Axiom Cmd : forall (pre : hprop) {A} (post : A -> hprop), Set.

Axiom CmdBind : forall pre1 T1 (post1 : T1 -> hprop)
  pre2 T2 (post2 : T2 -> hprop)
  (st1 : Cmd pre1 post1)
  (_ : forall v, post1 v ==> pre2 v)
  (st2 : forall v : T1, Cmd (pre2 v) post2)
  : Cmd pre1 post2.

Axiom CmdRead : forall (T : Set) (p : ptr) (P : T -> hprop),
  Cmd (Exists v :@ T, p ~~> v * P v)
      (fun v => p ~~> v * P v).

(** ... and more ... **)
```

Outline

C-style Linked Lists

- Linked lists in ML.

```
module type LLIST =
struct
  type 'a t
  val new : unit -> 'a t
  (** ... **)
  val sub : 'a t -> int -> 'a option
end
```

- A type (`t`) and functions on it (`new`, `sub`).

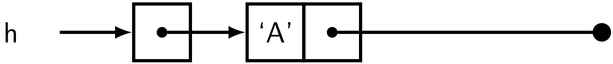
C-style Linked Lists

- Linked lists in ML.

```
module type LLIST =
struct
  type 'a t
  val new : unit -> 'a t
  (** ... **)
  val sub : 'a t -> int -> 'a option
end
```

- A type (`t`) and functions on it (`new`, `sub`).
- To reason about correctness, we need specifications.
  1. Relate the type `t` to a computationally irrelevant model.
  2. Provide a predicate that describes the heap in terms of model.
  3. Provide specifications as stronger types for the functions.

Representation Predicate



- Describe the heap computationally using a functional model.

.

Representation Predicate

pStart ———————————— pEnd

- Describe the heap computationally using a functional model.

.

```
Fixpoint llseg (pStart pEnd : optr) (ls : list T) : hprop :=
  match ls with
    | nil     => [pStart = pEnd]
```

Representation Predicate

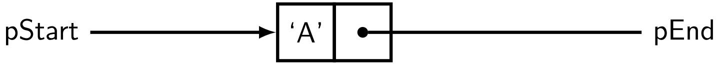
~~pStart~~ ~~pEnd~~

- Describe the heap computationally using a functional model.

.

```
Fixpoint llseg (pStart pEnd : optr) (ls : list T) : hprop :=
  match ls with
    | nil    => [pStart = pEnd]
    | a :: b => match pStart with
                  | None => [False]
```

Representation Predicate

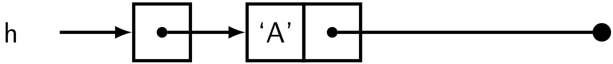


- Describe the heap computationally using a functional model.

```
Record llNode := mkNode { val : T ; next : optr }.

Fixpoint llseg (pStart pEnd : optr) (ls : list T) : hprop :=
  match ls with
    | nil    => [pStart = pEnd]
    | a :: b => match pStart with
                  | None => [False]
                  | Some p => Exists nx :@ option ptr,
                    p ~~> mkNode a nx * llseg nx pEnd b
  end end.
```

Representation Predicate



- Describe the heap computationally using a functional model.

```
Record llNode := mkNode { val : T ; next : optr }.

Fixpoint llseg (pStart pEnd : optr) (ls : list T) : hprop :=
  match ls with
    | nil    => [pStart = pEnd]
    | a :: b => match pStart with
                  | None => [False]
                  | Some p => Exists nx :@ option ptr,
                    p ~~> mkNode a nx * llseg nx pEnd b
  end end.

Definition tlst := ptr.
Definition llist (h : tlst) (m : list T) : hprop :=
  Exists st :@ option ptr, h ~~> st * llseg st None m.
```

Linked Lists: sub

```
Definition sub : forall (t : tlst) (i : nat) (m : [list T]),
  Cmd (m ~~ llist t m)
      (fun res : option T =>
         m ~~ llist t m * [res = nth_error m i]).
```

Linked Lists: sub

```
Definition sub : forall (t : tlst) (i : nat) (m : [list T]),
  Cmd (m ~~ llist t m)
      (fun res : option T =>
         m ~~ llist t m * [res = nth_error m i]).
  refine (fun t i m =>
    hd <- ! t ;
    {{ Fix3 (fun hd j m => m ~~ llseg hd None m)
            (fun hd j m (r : option T) =>
               m ~~ llseg hd None m * [r = nth_error m j])
            (fun self hd j m =>
              IfNull hd Then {{ Return None }}
              Else
                nde <- ! hd ;
                IfZero j Then
                  {{ Return (Some (val nde)) }}
                Else
                  {{ self (next nde) j (m ~~~ tail m) <@> _ }}
            ) hd i m <@> _ }});
  try clear self; sep' s tac.
Qed.
```

Linked Lists: sub

```
Definition sub : forall (t : tlst) (i : nat) (m : [list T]),
  Cmd (m ~~ llist t m)
      (fun res : option T =>
         m ~~ llist t m * [res = nth_error m i]).
  refine (fun t i m =>
    hd <- ! t ;
    {{ Fix3 (fun hd j m => m ~~ llseg hd None m)
            (fun hd j m (r : option T) =>
               m ~~ llseg hd None m * [r = nth_error m j])
            (fun self hd j m =>
              IfNull hd Then {{ Return None }}
              Else
                nde <- ! hd ;
                IfZero j Then
                  {{ Return (Some (val nde)) }}
                Else
                  {{ self (next nde) j (m ~~~ tail m) <@> _ }}
            ) hd i m <@> _ }});
  try clear self; sep' s tac.
Qed.
```

Linked Lists: sub

```
Definition sub : forall (t : tlst) (i : nat) (m : [list T]),
  Cmd (m ~~ llist t m)
      (fun res : option T =>
         m ~~ llist t m * [res = nth_error m i]).
  refine (fun t i m =>
    hd <- ! t ;
    {{ Fix3 (fun hd j m => m ~~ llseg hd None m)
            (fun hd j m (r : option T) =>
               m ~~ llseg hd None m * [r = nth_error m j])
            (fun self hd j m =>
              IfNull hd Then {{ Return None }}
              Else
                nde <- ! hd ;
                IfZero j Then
                  {{ Return (Some (val nde)) }}
                Else
                  {{ self (next nde) j (m ~~~ tail m) <@> _ }}
            ) hd i m <@> _ }});
  try clear self; sep' s tac.
Qed.
```

Linked Lists: sub

```
Definition sub : forall (t : tlst) (i : nat) (m : [list T]),
  Cmd (m ~~ llist t m)
      (fun res : option T =>
         m ~~ llist t m * [res = nth_error m i]).
  refine (fun t i m =>
    hd <- ! t ;
    {{ Fix3 (fun hd j m => m ~~ llseg hd None m)
            (fun hd j m (r : option T) =>
               m ~~ llseg hd None m * [r = nth_error m j])
            (fun self hd j m =>
              IfNull hd Then {{ Return None }}
              Else
                nde <- ! hd ;
                IfZero j Then
                  {{ Return (Some (val nde)) }}
                Else
                  {{ self (next nde) j (m ~~~ tail m) <@> _ }}
            ) hd i m <@> _ }});
  try clear self; sep' s tac.
Qed.
```

Linked Lists: mfold_left

- Can even write higher-order computations while maintaining abstraction.

```
Definition mfold_left : forall {U : Type} (t : tlst)
  (I : list T -> U -> hprop) (a : U) (m : [list T])
  (cmd : forall (c : T) (a : U) (m : [list T]),
     Cmd (m ~~ I m a)
         (fun a : U => m ~~ I (m ++ c :: nil) a)),
  Cmd (m ~~ llist t m * I nil a)
      (fun r : U => m ~~ llist t m * I m r).
```

- I is the invariant.
- a is the initial accumulator.
- cmd is the folded computation.

Outline

Adding Iterators

- Iterators and collections go hand-in-hand.

```
Class ListIterable (h : Set) (T : Type) : Type := {
  rep   : h -> list T -> nat -> hprop ;
```

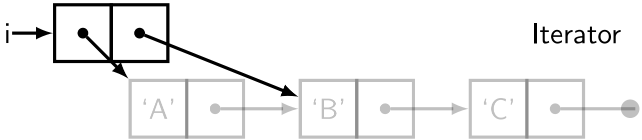
Adding Iterators

- Iterators and collections go hand-in-hand.

```
Class ListIterable (h : Set) (T : Type) : Type := {
  rep  : h -> list T -> nat -> hprop ;
  next : forall (t : h) (m : [list T]) (idx : [nat]),
    Cmd (m ~~ idx ~~ rep t m idx)
        (fun res : option T => m ~~ idx ~~
           rep t m (nextIndex idx (length m)) *
           [res = nth_error m idx])
}.
```

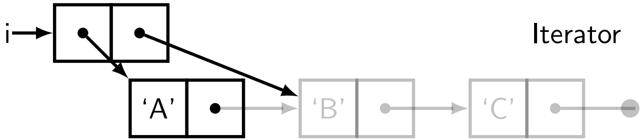
- `h` is the type of the iterator handle.
- `T` is the type of values being iterated over.
- Representation predicate (`rep`) and `next` command.

A Naïve Iterator



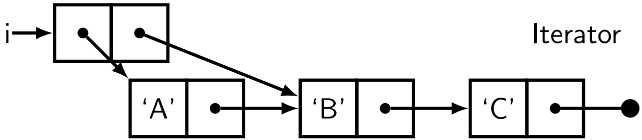

```
Definition titr := ptr.
(** Representation predicate **)
Definition liter (t : titr) (ls : list T) (idx : nat)
    : hprop :=
  Exists st :@ optr, Exists cur :@ optr,
  t ~~> (cur, st) *
```

A Naïve Iterator




```
Definition titr := ptr.
(** Representation predicate **)
Definition liter (t : titr) (ls : list T) (idx : nat)
    : hprop :=
  Exists st :@ optr, Exists cur :@ optr,
  t ~~> (cur, st) *
  llseg st cur (firstn idx ls) *
```

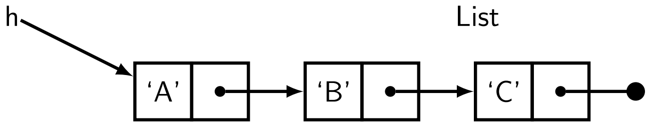
A Naïve Iterator




```
Definition titr := ptr.
(** Representation predicate **)
Definition liter (t : titr) (ls : list T) (idx : nat)
    : hprop :=
  Exists st :@ optr, Exists cur :@ optr,
  t ~~> (cur, st) *
  llseg st cur (firstn idx ls) *
  llseg cur None (skipn idx ls).
```

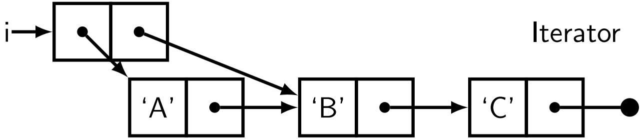
The Sharing Problem

- Requires access to the same memory as the underlying list.
  - Creating an iterator *consumes* the underlying list.
  - Can't have multiple iterators.



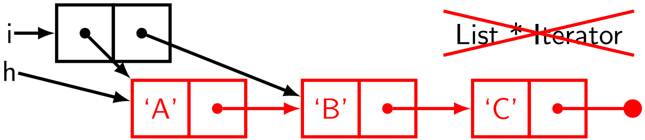
The Sharing Problem

- Requires access to the same memory as the underlying list.
  - Creating an iterator *consumes* the underlying list.
  - Can't have multiple iterators.



The Sharing Problem

- Requires access to the same memory as the underlying list.
  - Creating an iterator *consumes* the underlying list.
  - Can't have multiple iterators.



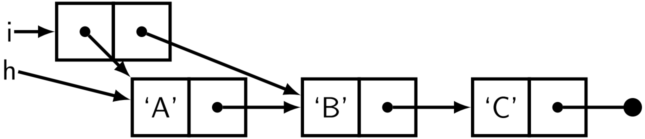
The Sharing Problem: Specifications

- Computations on iterators can't be called with the same underlying list.

```
Definition zip : forall (i1 i2 : titr)
    (l1 : [list T]) (l2 : [list U]),
  Cmd (l1 ~~ l2 ~~ liter i1 l1 0 * liter i2 l2 0 *
          [length l1 = length l2])
      (fun res : tlst => l1 ~~ l2 ~~
          liter i1 l1 (length l1) * liter i2 l2 (length l2) *
          llist res (zip l1 l2))
```

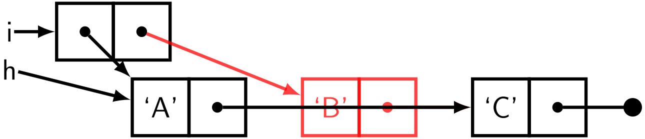
A Real Sharing Problem

- Who “owns” the list turns out to be a real problem.



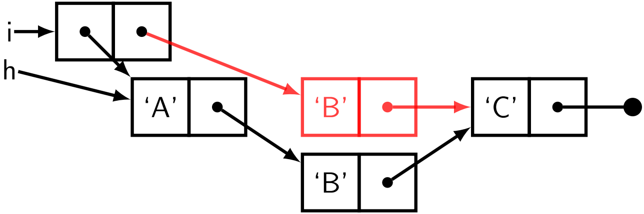
A Real Sharing Problem

- Who "owns" the list turns out to be a real problem.



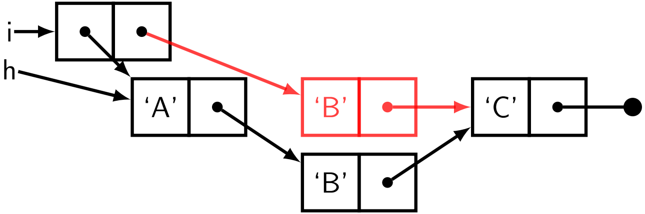
A Real Sharing Problem

- Who "owns" the list turns out to be a real problem.



A Real Sharing Problem

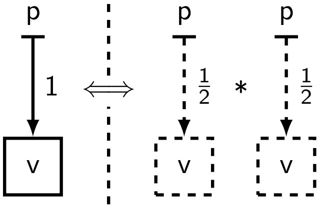
- Who "owns" the list turns out to be a real problem.



- Doesn't satisfy frame property!
  - Source of Java's `ConcurrentModificationException`.

Sharing with Fractional Permissions [1] (Boyland '03)

- Parameterize points-to by a fractional ownership.
  - `p ~[ q ]~> v`, `q` is the fraction.
- Ownership determines your capabilities:
  - Full permissions allows everything: read, write, free.
  - Partial permissions only allows reading.
  - Permissions can be split and joined.



[1] Ynot implementation by Avi Shinnar.

A Fractional Iterator

- Describe the iterator as owning a fraction of the whole list.

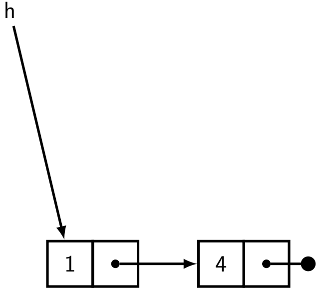
```
(** Representation predicate **)
Definition liter (owner : tlst) (q : Fp)
    (t : titr) (ls : list T) (idx : nat) : hprop :=
  Exists st :@ optr, Exists cur :@ optr,
  t ~~> (cur, st) *
  llseg st cur (firstn idx ls) q *
  llseg cur None (skipn idx ls) q.
```

- q is the fraction of the list that is owned.
- Allows multiple iterators over the same list.
  - As long as the fractions are compatible.

Outline

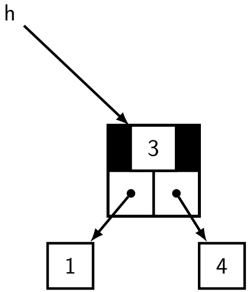
Aliasing

- Data structures with aliasing are more difficult to describe.



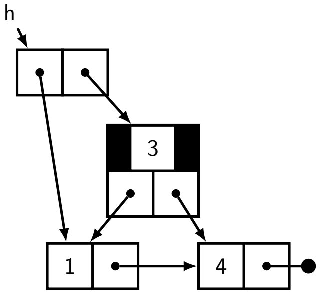
Aliasing

- Data structures with aliasing are more difficult to describe.



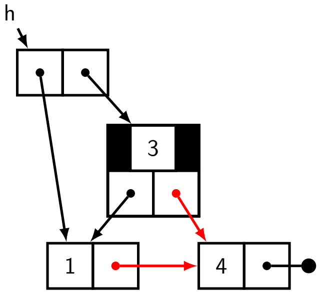
Aliasing

- Data structures with aliasing are more difficult to describe.



Aliasing

- Data structures with aliasing are more difficult to describe.



B+ Trees

- B+ trees are *n*-ary trees where the leaves are connected by a linked list.
  - Support fast lookup and in-order iteration.
  - Commonly used for database indices. (Malecha '10)
- Previous formalizations exist, but neither is mechanically verified:
  - Classical conjunction, $(\texttt{list} * \texttt{any}) \wedge (\texttt{tree})$. (Bornat '04)
  - B+ tree language. (Sexton '08)
- Both of these approaches seemed difficult to automate.

Difficulties of the Invariant

- Several difficulties describing this:
  - Have to encode pointer aliasing explicitly.
  - Many different B+ trees can describe the same finite map.
  - Enforce the tree balancedness.
  - Enforce the ordering of keys.
  - Invariants on the size of branches and leaves.

Difficulties of the Invariant

- Several difficulties describing this:
  - Have to encode pointer aliasing explicitly.
  - Many different B+ trees can describe the same finite map.
  - Enforce the tree balancedness.
  - Enforce the ordering of keys.
  - Invariants on the size of branches and leaves.

Representation Invariant

- Existentially quantify an irrelevant model (`tr`) of the tree which contains the pointers.
  - Avoids existentials in the representation invariant, simplifies automation.
  - Makes the heap predicate (`repTree`) very computational.

```
Definition rep (p : BptMap) (m : Model) : hprop :=
  Exists pRoot :@ ptr, Exists h :@ nat, Exists tr :@ ptree h,
  p ~~> (pRoot, existT (fun h:nat => [ptree h]) h [tr]) *
  repTree pRoot None tr *
```

Representation Invariant

- Existentially quantify an irrelevant model (`tr`) of the tree which contains the pointers.
  - Avoids existentials in the representation invariant, simplifies automation.
  - Makes the heap predicate (`repTree`) very computational.
  - Connect the logical model (`m`) to the physical model (`tr`).

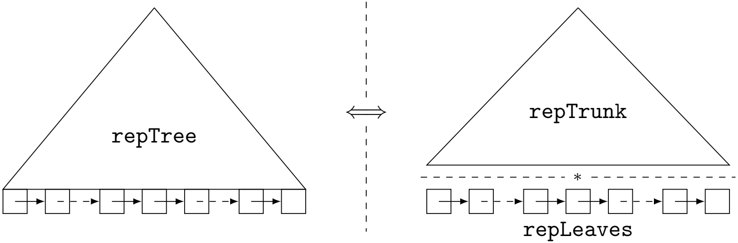
```
Definition rep (p : BptMap) (m : Model) : hprop :=
  Exists pRoot :@ ptr, Exists h :@ nat, Exists tr :@ ptree h,
  p ~~> (pRoot, existT (fun h:nat => [ptree h]) h [tr]) *
  repTree pRoot None tr *
  [eqlistA entry_eq m (as_map tr)] *
```

Representation Invariant

- Existentially quantify an irrelevant model (`tr`) of the tree which contains the pointers.
  - Avoids existentials in the representation invariant, simplifies automation.
  - Makes the heap predicate (`repTree`) very computational.
  - Connect the logical model (`m`) to the physical model (`tr`).
  - Consolidate pure facts about the model in `inv`.

```
Definition rep (p : BptMap) (m : Model) : hprop :=
  Exists pRoot :@ ptr, Exists h :@ nat, Exists tr :@ ptree h,
  p ~~> (pRoot, existT (fun h:nat => [ptree h]) h [tr]) *
  repTree pRoot None tr *
  [eqlistA entry_eq m (as_map tr)] *
  [inv _ tr MinK MaxK].
```

1 Model, 2 Views



- Can switch between views by proving and applying a lemma:

```
Lemma repTree_repTrunkLeaves : forall (h : nat)
    (p : ptr) (optr : optr) (m : ptree h),
  repTree p optr m
  <==>
  repTrunk p optr m *
  repLeaves (Some (firstPtr m)) (leaves m) optr.
```

Outline

Some Lessons

- Fractional permissions are necessary even for sequential code.
- Separation logic makes trees much easier than DAGs/graphs.
  - Can simplify things by re-ifying an irrelevant model.
  - Big win for automation.
- Higher-order ADT functions: `fold`
- Automation pays off when reasoning about separation logic.
- Higher-order abstraction simplifies specifications and proofs.

Other Projects & Outlook

Previous Projects

- Verified web application — trace-based I/O. (Wisnesky '09)
- Verified relational database. (Malecha '10)

Other Projects & Outlook

Previous Projects

- Verified web application — trace-based I/O. (Wisnesky '09)
- Verified relational database. (Malecha '10)

Future?

- Still a fair amount of work for a more realistic system.
  - Reasoning about concurrency.
    - Brookes '07, Appel '08, Nanevski '09
  - Reasoning about failures.
  - Proofs can still be tedious & long.
    - Domain specific external provers.

`http://ynot.cs.harvard.edu/`