Mechanized Verification with Sharing

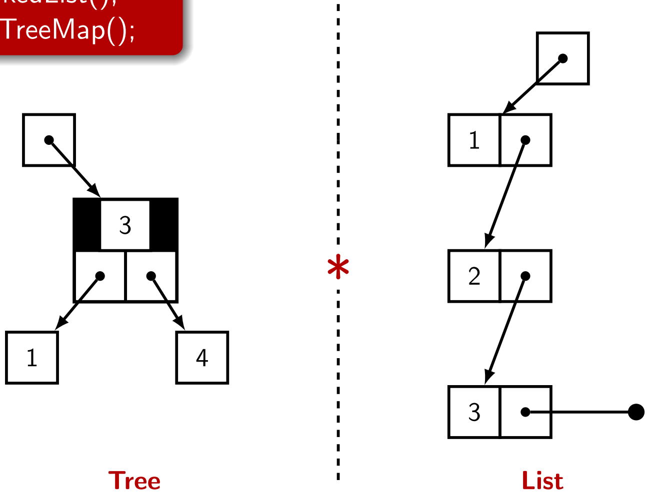
Gregory Malecha Greg Morrisett

Harvard SEAS

September 2, 2010

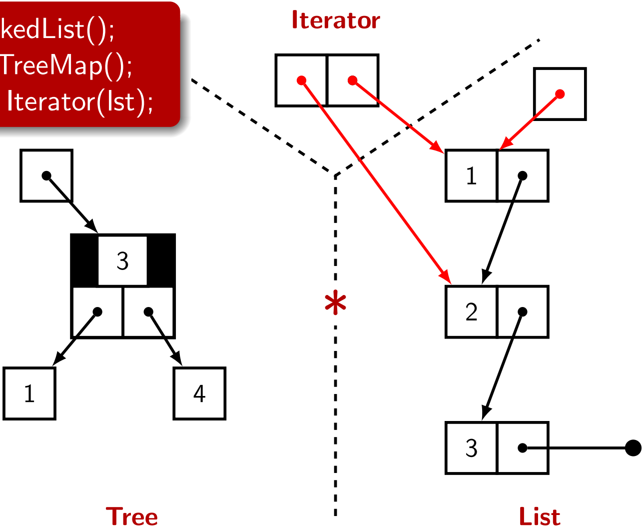
Separation Logic is Great

List lst = LinkedList();
Map map = TreeMap();



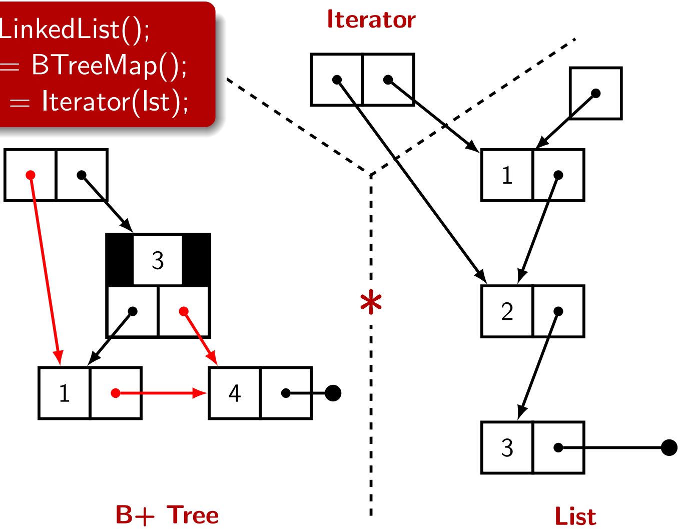
Aliasing Breaks Everything: External Sharing

```
List lst = LinkedList();
Map map = TreeMap();
Iterator itr = Iterator(lst);
```



Aliasing Breaks Everything: **Internal** Sharing

List lst = LinkedList();
Map map = BTreeMap();
Iterator itr = Iterator(lst);



Outline

Outline

Where have all the effects gone?

- Most type systems don't express effects.

$$\boxed{\text{swap} : \text{int } \textbf{ref} \rightarrow \text{int } \textbf{ref} \rightarrow \text{unit}}$$

- Don't know whether the function has effects.
- Must break encapsulation to reason about programs that call `swap`.

Black-box Effects

- Monads mitigate this problem to some extent.

$$\boxed{\text{swap} :: \text{MVar int} \rightarrow \text{MVar int} \rightarrow \textbf{IO}\ ()}$$

- Can make explicit the **absence** of effects.
- Does **not** explicitly express their presence...or exactly what they are.

Hoare Type Theory: Ynot

- We rely on *program logics* to express effects.

$$\{P\}c\{r : T \Rightarrow Q\}$$

Hoare Type Theory: Ynot

- We rely on *program logics* to express effects.
- Build the logic into the type system.
    - Express side-effects using a dependent monad.

$$\{P\}c\{r : T \Rightarrow Q\} \quad \equiv \quad \texttt{c : Cmd P (r : T} \Rightarrow \texttt{Q)}$$

Hoare Type Theory: Ynot

- We rely on *program logics* to express effects.
- Build the logic into the type system.
  - Express side-effects using a dependent monad.

$$\{P\}c\{r : T \Rightarrow Q\} \quad \equiv \quad \texttt{c : Cmd P (r : T} \Rightarrow \texttt{Q)}$$

Pre-condition

Hoare Type Theory: Ynot

- We rely on *program logics* to express effects.
- Build the logic into the type system.
  - Express side-effects using a dependent monad.

$$\{P\}c\{r : T \Rightarrow Q\} \quad \equiv \quad \texttt{c : Cmd P (r : T} \Rightarrow \texttt{Q)}$$

Return Type

Hoare Type Theory: Ynot

- We rely on *program logics* to express effects.
- Build the logic into the type system.
  - Express side-effects using a dependent monad.

$$\{P\}c\{r : T \Rightarrow Q\} \quad \equiv \quad \texttt{c : Cmd P (r : T} \Rightarrow \texttt{Q)}$$

Post-condition (depends on return value)

Hoare Type Theory: Ynot

- We rely on *program logics* to express effects.
- Build the logic into the type system.
  - Express side-effects using a dependent monad.

$$\{P\}c\{r : T \Rightarrow Q\} \quad \equiv \quad \texttt{c : Cmd P (r : T} \Rightarrow \texttt{Q)}$$

swap : $\forall$ p q (v u : int),
Cmd (p $\mapsto$ v $*$ q $\mapsto$ u)
(_ : unit $\Rightarrow$ p $\mapsto$ u $*$ q $\mapsto$ v)

Basic Typing Rule: Read a Pointer

- Hoare Logic

$$\frac{P \Longrightarrow p \mapsto v * P'\ v}{\{P\}!p\{r \Rightarrow p \mapsto r * P'\ r\}}\ \textsc{Read}$$

- Hoare Type

```
read : ∀ p P',
  Cmd (∃ v. p ↦ v ∗ P' v)
      (r ⇒ p ↦ r ∗ P' r).
```

Basic Typing Rule: Read a Pointer

- Hoare Logic

$$\frac{\boxed{P \implies p \mapsto v * P'\ v}}{\{P\}!p\{r \Rightarrow p \mapsto r * P'\ r\}}\ \textsc{Read}$$

- Hoare Type

```
read : ∀ p P',
  Cmd (∃ v. p ↦ v * P' v)
      (r ⇒ p ↦ r * P' r).
```

Basic Typing Rule: Read a Pointer

- Hoare Logic

$$\frac{P \implies p \mapsto v * P'\ v}{\{P\}!p\{r \Rightarrow \boxed{p \mapsto r * P'\ r}\}}\ \textsc{Read}$$

- Hoare Type

```
read  : ∀ p P',
  Cmd (∃ v. p ↦ v ∗ P' v)
        (r  ⇒ p ↦ r ∗ P' r).
```

Outline

C-style Linked Lists

```
module type LLIST =
struct
  type tlst (** = list int **)
  val empty : unit → tlst
  (** ... **)
  val sub   : tlst → int → int option
end
```

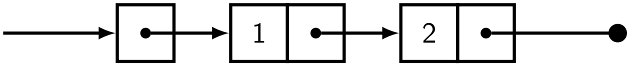
- An abstract type (`tlst`) and functions on it (`empty`, `sub`).

C-style Linked Lists

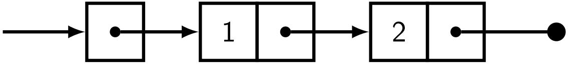
module type LLIST $=$
struct
  type tlst $(** = list\ \ int\ **)$
  val empty : unit $\rightarrow$ tlst
  $(**\ \ ...\ \ **)$
  val sub : tlst $\rightarrow$ int $\rightarrow$ int option
end

- An abstract type (`tlst`) and functions on it (`empty`, `sub`).
- To reason about correctness, we need specifications.
  1. Relate the type `tlst` to a functional model.
  2. Describe the heap in terms of the model.
  3. Provide specifications for functions.

(1&2) Representation Predicate



(1&2) Representation Predicate



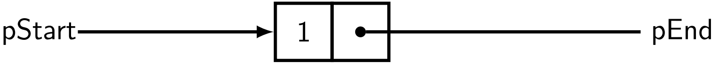
Record llNode := mkNode { val : int ; next : optr }.

(1&2) Representation Predicate


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

Equations llseg (pStart pEnd : optr) (ls : list int) :=
 llseg pStart pEnd nil :=
   pStart = pEnd
```

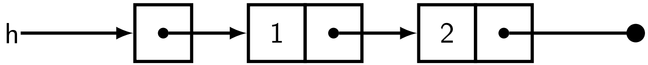
(1&2) Representation Predicate




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

Equations llseg (pStart pEnd : optr) (ls : list int) :=
 llseg pStart pEnd nil :=
   pStart = pEnd
 llseg (Ptr st) pEnd (a :: b) :=
   ∃ nx : optr, st ↦ mkNode a nx ∗ llseg nx pEnd b
```

(1&2) Representation Predicate




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

Equations llseg (pStart pEnd : optr) (ls : list int) :=
 llseg pStart pEnd nil :=
   pStart = pEnd
 llseg (Ptr st) pEnd (a :: b) :=
   ∃ nx : optr, st ↦ mkNode a nx ∗ llseg nx pEnd b

Definition tlst := ptr.
Definition llist (h : tlst) (m : list T) :=
   ∃ st : optr, h ↦ st ∗ llseg st Null m.
```

(3) Correctness Specifications

$$
\begin{array}{l}
\text{empty} : \textbf{Cmd}\ (\text{emp}) \\
\qquad\qquad\qquad (\text{r} : \text{tlst} \Rightarrow \text{llist r nil}) \\
\\
\text{sub} : \forall\ (\text{t} : \text{tlst})\ (\text{i} : \text{int})\ (\text{m} : \text{list int}), \\
\quad \textbf{Cmd}\ (\text{llist t m}) \\
\qquad\qquad (\text{r} : \text{option int} \Rightarrow \text{llist t m} * \text{r} = \text{nth m i})
\end{array}
$$

Outline

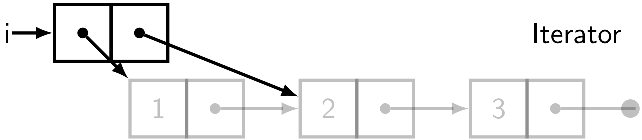
Expressing Iterators

```
Class ListIterable (titr : Type) :=
{ rep : titr → list int → nat → hprop
```

Expressing Iterators

```
Class ListIterable (titr : Type) :=
{ rep : titr → list int → nat → hprop
; next : ∀ (t : titr) (m : list int) (idx : nat),
    Cmd (rep t m idx)
        (res : option int ⇒ res = nth m idx *
         rep t m (idx + 1))
}.
```

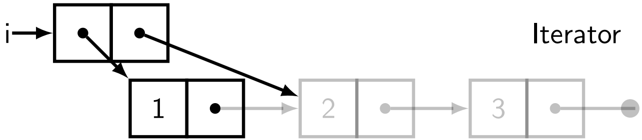
A Simple Iterator




```
Definition titr := ptr.

Definition liter (t : titr) (ls : list int) (n : nat) :=
  ∃ st : optr, ∃ cur : optr,
  t ↦ (st, cur) ∗
```

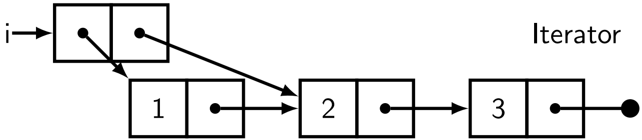
A Simple Iterator




```
Definition titr := ptr.

Definition liter (t : titr) (ls : list int) (n : nat) :=
  ∃ st : optr, ∃ cur : optr,
  t ↦ (st, cur) ∗
  llseg st cur (firstn n ls) ∗
```

A Simple Iterator

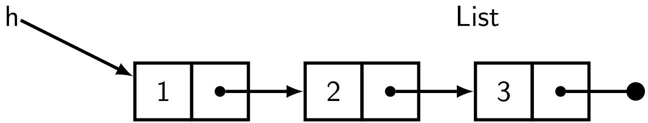



```
Definition titr := ptr.

Definition liter (t : titr) (ls : list int) (n : nat) :=
  ∃ st : optr, ∃ cur : optr,
  t ↦ (st, cur) ∗
   llseg st cur (firstn n ls) ∗
   llseg cur Null (skipn n ls).
```

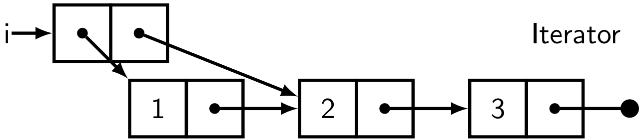
The Sharing Problem

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



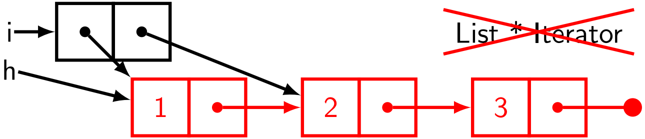
The Sharing Problem

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



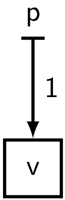
The Sharing Problem

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



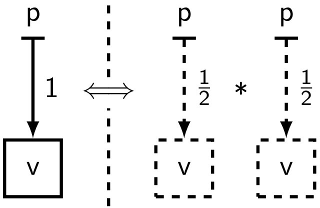
Sharing with Fractional Permissions (Boyland '03)

- Parameterize points-to by a fractional ownership.
  - p $\stackrel{q}{\mapsto}$ v, q is the fraction.



Sharing with Fractional Permissions (Boyland '03)

- Parameterize points-to by a fractional ownership.
  - p $\overset{q}{\mapsto}$ v, q is the fraction.



A Fractional Iterator

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

```
Definition liter (owner : tlst ) (q : Fp)
    (t : titr ) ( ls : list int ) (n : nat) : hprop :=
  ∃ st : optr , ∃ cur : optr ,
  t ↦ (st, cur) ∗
  llseg st cur ( firstn n ls ) q ∗
  llseg cur Null (skipn n ls ) q.
```

- During construction, only consume a fraction of the list.

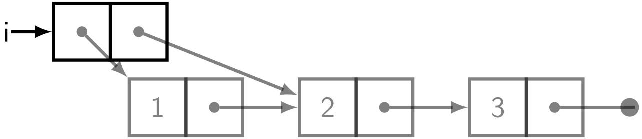
```
Definition iterator : ∀ (t : tlst ) (m : list int ) (q : Fp),
  Cmd (llist t m q)
      (res : titr ⇒ liter t q res m 0).
```

The Sharing Problem

- Multiple objects can share the underlying list.

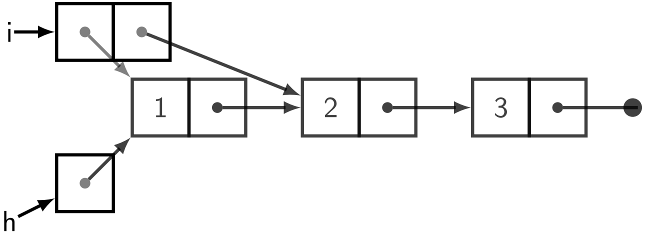
The Sharing Problem

- Multiple objects can share the underlying list.



The Sharing Problem

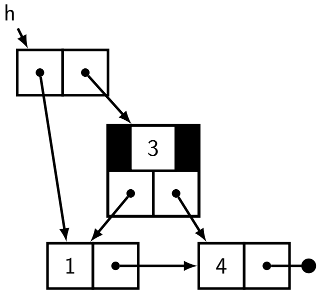
- Multiple objects can share the underlying list.



Outline

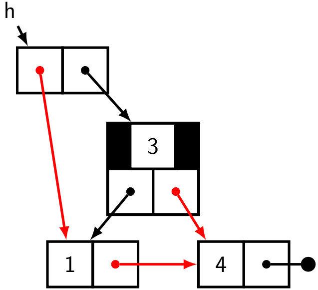
Aliasing

- Aliasing makes describing data structure more difficult.



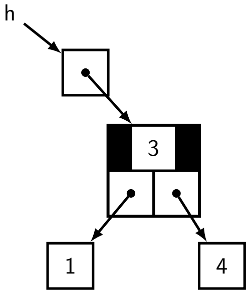
Aliasing

- Aliasing makes describing data structure more difficult.



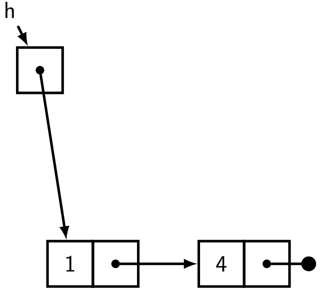
Aliasing

- Aliasing makes describing data structure more difficult.



Aliasing

- Aliasing makes describing data structure more difficult.



Difficulties of the Invariant

- Many different B+ trees can describe the same finite map.

Difficulties of the Invariant

- Many different B+ trees can describe the same finite map.
- Have to encode pointer aliasing explicitly.

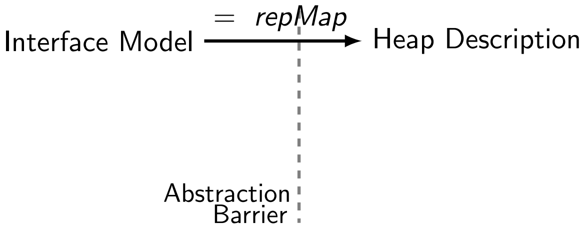
Difficulties of the Invariant

- Many different B+ trees can describe the same finite map.
- Have to encode pointer aliasing explicitly.
- Enforce pure properties:
    - Balanced structure.
    - Ordered keys.
    - Leaf & branch sizes.

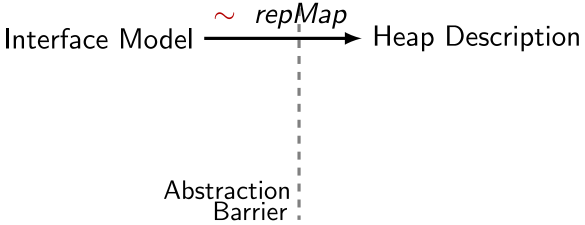
Difficulties of the Invariant

- **Many different B+ trees can describe the same finite map.**
- **Have to encode pointer aliasing explicitly.**
- Enforce pure properties:
  - Balanced structure.
  - Ordered keys.
  - Leaf & branch sizes.

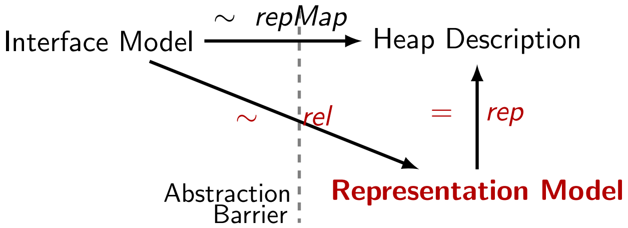
Representation Predicate



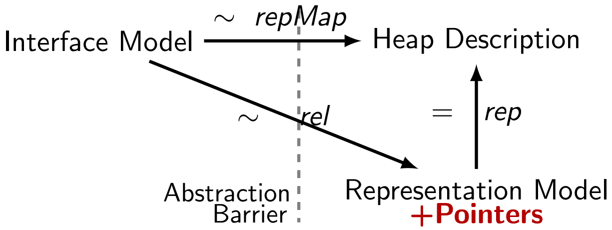
Representation Predicate



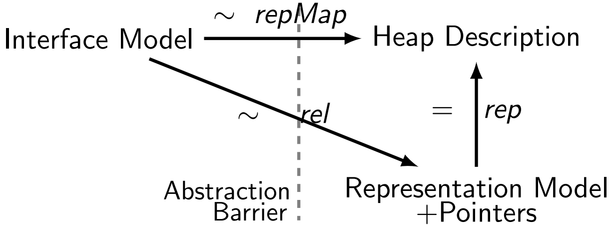
Representation Predicate



Representation Predicate

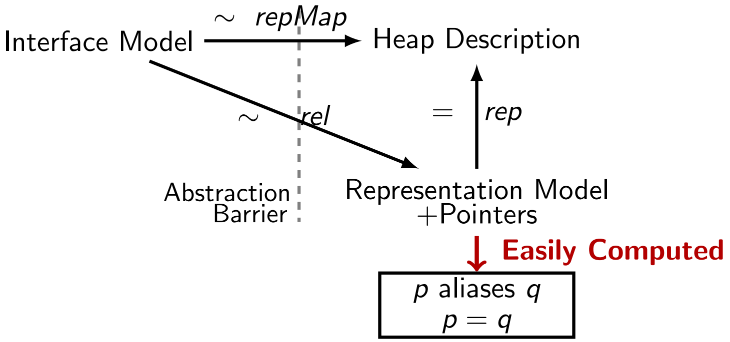


Representation Predicate



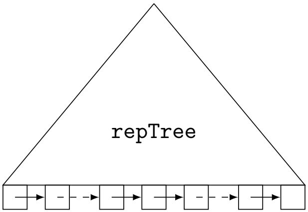
$\boxed{\begin{array}{c} p \textbf{ aliases } q \\ p = q \end{array}}$

Representation Predicate



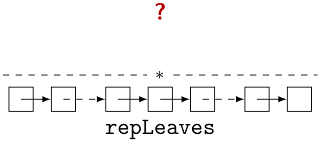
Implementing the Interface

- lookup, insert



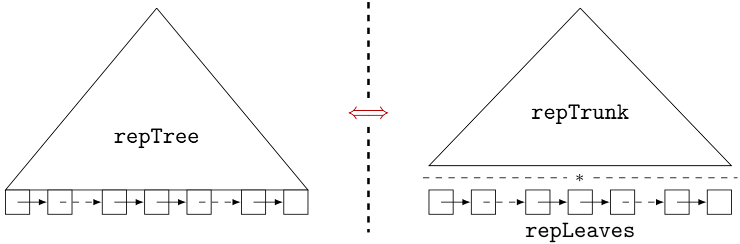
Implementing the Interface

- fold



Implementing the Interface

- lookup, insert
- fold



Lemma repTree_repTrunkLeaves : ∀ h p optr (m : ptree h),
repTree p optr m ⟺
repTrunk p optr m ∗ repLeaves (firstPtr m) optr (leaves m).

Outline

Take Away

1. **Sequential Sharing** — Even sequential code has sharing problems.
2. **Fractional permissions** — Expose sharing at interfaces.
3. **Internal Aliasing** – Hide details of efficient implementation.