Verification with Sharing and Aliasing

Gregory Malecha

Harvard University SEAS

March 5, 2010

Why Verify?

Observation

All large, complex systems have bugs.

- Hardware design – Intel floating point bug ($300 million)
- Mars rover – Priority inversion
- Security – Internet viruses and worms
- Voting machines – Hacking voting machines (an election?)
- Safety – control systems for airplanes, power plants, space shuttle

A problem that isn't going away...

- Just waiting won't solve this problem...
  - A computer that runs twice as fast will just trigger twice as many bugs per second.

A problem that isn't going away...

- Just waiting won't solve this problem...
  - A computer that runs twice as fast will just trigger twice as many bugs per second.
- ...actually time is making it harder.
  - Multicore and multiprocessor means reasoning about concurrency.
  - Lax memory models make low-level reasoning more difficult.

A problem that isn't going away...

- Just waiting won't solve this problem...
  - A computer that runs twice as fast will just trigger twice as many bugs per second.
- ...actually time is making it harder.
  - Multicore and multiprocessor means reasoning about concurrency.
  - Lax memory models make low-level reasoning more difficult.
- And, we're trying to solve bigger problems than before...
  - Data integrity and security
  - Scientific simulation

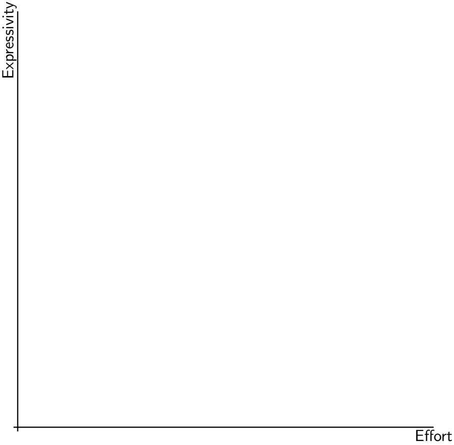
Outline

1. Techniques for Gaining Confidence
2. Software Verification with Types
   - Modularity and Abstraction
3. My Work: Addressing Sharing and Aliasing
   - Sharing: Iterators
   - Aliasing: B+ Trees
4. Conclusions

Outline

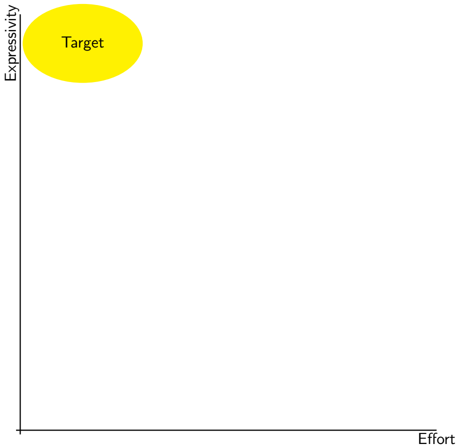
1. Techniques for Gaining Confidence
2. Software Verification with Types
   - Modularity and Abstraction
3. My Work: Addressing Sharing and Aliasing
   - Sharing: Iterators
   - Aliasing: B+ Trees
4. Conclusions

Techniques for Reasoning About Code



- Basic trade-off between the amount of effort required and the expressivity of the properties.

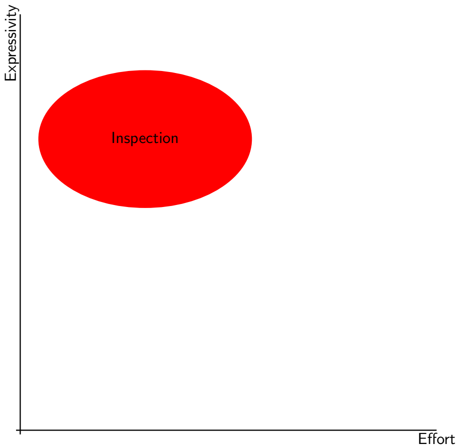
Techniques for Reasoning About Code



Goal

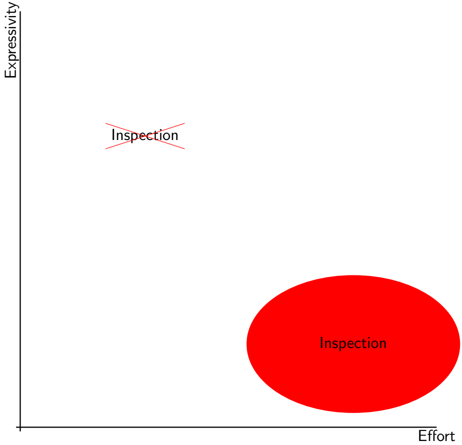
- Strong guarantees about complex properties.
- Scalable and modular.

Techniques for Reasoning About Code



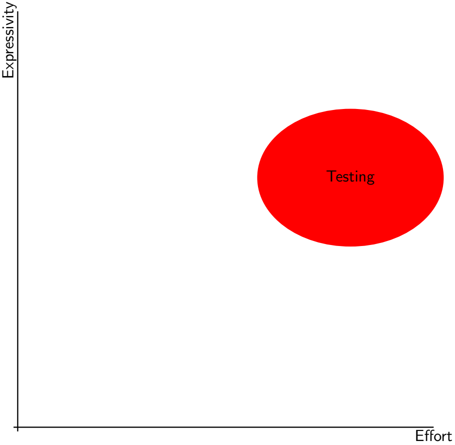
- We would love it if just looking at the code was here...

Techniques for Reasoning About Code



- We would love it if just looking at the code was here...
- But we all know it's more like here...

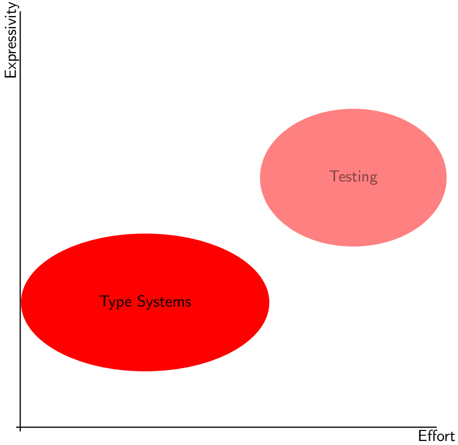
Techniques for Reasoning About Code



Testing

- **Pro** Direct and intuitive methodology.
- **Con** Large amount of code, very manual.

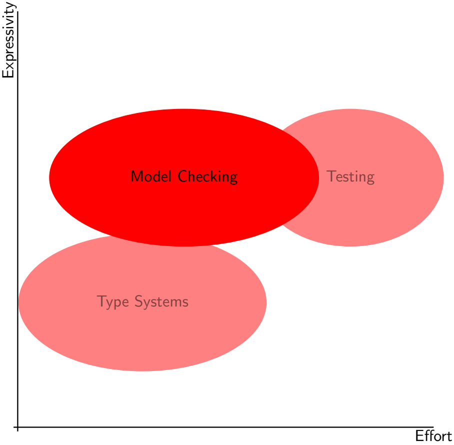
Techniques for Reasoning About Code



Type Systems

- **Pro** Fast, (can be) provably correct and compositional.
- **Con** Limited properties, "restricted programming".

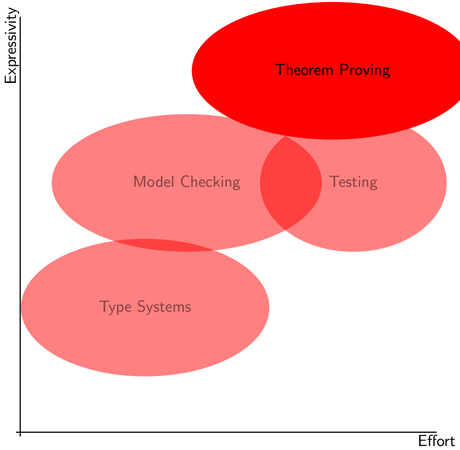
Techniques for Reasoning About Code



Model Checking

- **Pro** "Push-button" when it works and somewhat intuitive.
- **Con** Computationally expensive, can be difficult to set up.

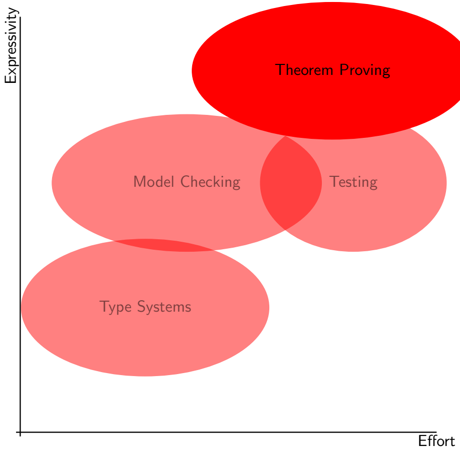
Techniques for Reasoning About Code



Theorem Proving

- **Pro** Expressive and provably correct.
- **Con** Proving can be tedious, often requires an “expert”.

Techniques for Reasoning About Code



Theorem Proving

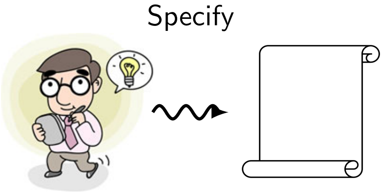
- **Pro** Expressive and provably correct.
- **Con** Proving can be tedious, often requires an "expert".
- This is the focus of the talk.

The Myth of "Correctness"

- "Correct" is dependent on what the system should do.

The Myth of "Correctness"

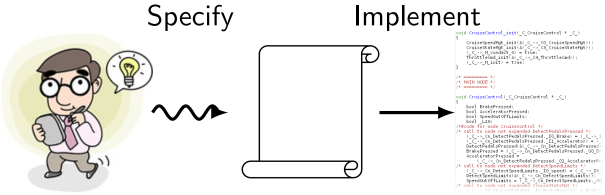
- "Correct" is dependent on what the system should do.



- Errors can enter at the specification level.
  - Specification shouldn't talk about complex implementation details.
  - Should be easier to write and reason about.

The Myth of “Correctness”

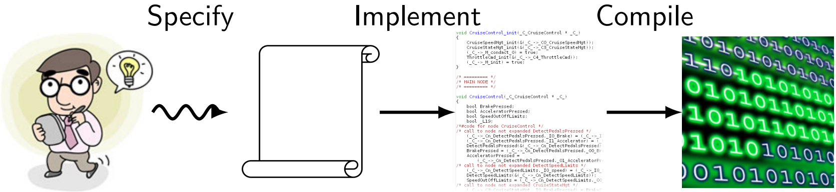
- “Correct” is dependent on what the system should do.



- Errors can enter at the specification level.
  - Specification shouldn’t talk about complex implementation details.
  - Should be easier to write and reason about.
- We can verify an implementation with respect to a specification.

The Myth of "Correctness"

- "Correct" is dependent on what the system should do.



- Errors can enter at the specification level.
  - Specification shouldn't talk about complex implementation details.
  - Should be easier to write and reason about.
- We can verify an implementation with respect to a specification.
- Compile the implementation in a certified way.

Outline

Building on Types

- How do you figure out what a function does?

```
int largest(int cnt, int* ary) {
  ... /** implementation **/ ...
}
```

Building on Types

- How do you figure out what a function does?

```
int largest(int cnt, int* ary) {
  ... /** implementation **/ ...
}
```

Building on Types

- How do you figure out what a function does?

```
/** largest(cnt, ary)
 **   returns the largest element in the first
 **   cnt elements of ary
 ** Requires:
 ** = 1 <= cnt <= length of ary
 **/
int largest(int cnt, int* ary) {
  ... /** implementation **/ ...
}
```

A Little More Expressive

- Annotation languages like *PREfix/PREfast* allow specifying properties like array bounds information.

```
/** largest(cnt, ary)
 **   returns the largest element in the first
 **   cnt elements of ary
 ** Assumes:
 ** = 1 <= cnt <= length of ary
 **/
int largest(int cnt, __in_ecount(cnt) int* ary) {
  ... /** implementation **/ ...
}
```

Specifications as Dependent Types

- Still aren't specifying everything...
  - Input: Empty arrays.
  - Output: The result is really the largest element.

```
largest(int cnt, int[cnt] ary, (0 < cnt) _pf) :
  {x : int | maximal x ary}
{ ... /* implementation */ ... }
```

- Types *depend* on run-time values.
  - Length of `ary` is `cnt`.
- Require proofs of preconditions & return proofs of correctness.
  - Proof that $0 <$ `cnt`.
  - Returns pair of the result and a proof that the result is correct.

A Monkey-Wrench: Effects

- The previous code was basically functional.
- Most programs use imperative state and effects.

```
void sortInPlace(int cnt, int[] ary) {
  ... /** Implementation **/ ...
}
```

- We need to state that the contents of ary changes.

A Standard Approach

- Can reason about effectful code using Hoare Logic.

$$\{P\}\, c\, \{r \Rightarrow Q\}$$

  - P is the precondition.
  - c is the command to execute.
  - r is a binder for the return value.
  - Q is the postcondition which depends on r.
- When the state of the program is described by P, c can be run and, if c terminates with return value r, the state of the program will be described by Q.

Describing the World

Example Program

```
{ p₁ ↦ 1 ∧ p₂ ↦ 1 }
*p₁ = 3
{_ ⇒ p₁ ↦ 3 ∧ p₂ ↦ 1 }
```

- Can we prove this?

Describing the World

Example Program

```
{ p₁ ↦ 1 ∧ p₂ ↦ 1 }
*p₁ = 3
{_ ⇒ p₁ ↦ 3 ∧ p₂ ↦ ??? }
```

- Can we prove this? No.
  - What if p_1 is an alias of p_2?

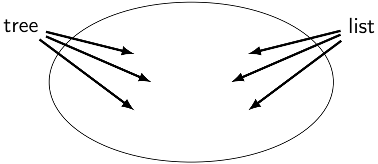
Describing the World

Example Program

$\{\ p_1 \mapsto 1 \wedge p_2 \mapsto 1 \wedge [p_1 \neq p_2]\ \}$
$*p_1 = 3$
$\{\_ \Rightarrow p_1 \mapsto 3 \wedge p_2 \mapsto 1\ \}$

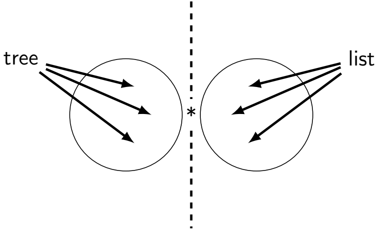
- Can we prove this? No.
  - What if p_1 is an alias of p_2?
  - We need a side condition for every pair of pointers.
  - Can't encode abstraction easily.

Separation Logic (Reynolds '02)



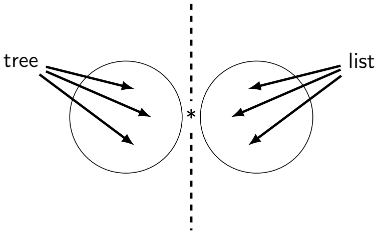
- We want to be able to reason about two structures independently.

Separation Logic (Reynolds '02)



- We want to be able to reason about two structures independently.
- Encode the disjointness condition in the $*$.
  - Easy to write the common case when pointers don't alias.

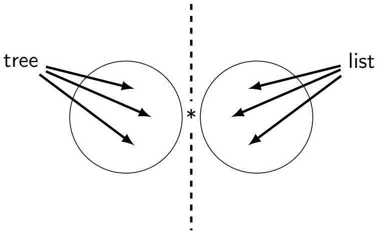
Separation Logic (Reynolds '02)



- We want to be able to reason about two structures independently.
- Encode the disjointness condition in the $*$.
  - Easy to write the common case when pointers don't alias.
- Called the "Frame Rule"
  - Allows us to temporarily "forget" about the list, reason about the tree, and then remember the list.

$$\frac{}{\{tree * list\}c\{r \Rightarrow tree' * list\}}\text{ Frame}$$

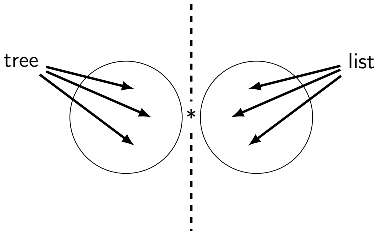
Separation Logic (Reynolds '02)



- We want to be able to reason about two structures independently.
- Encode the disjointness condition in the $*$.
    - Easy to write the common case when pointers don't alias.
- Called the "Frame Rule"
    - Allows us to temporarily "forget" about the list, reason about the tree, and then remember the list.

$$\frac{\{tree\}c\{r \Rightarrow tree'\}}{\{tree * list\}c\{r \Rightarrow tree' * list\}}\ \text{Frame}$$

Separation Logic (Reynolds '02)



- We want to be able to reason about two structures independently.
- Encode the disjointness condition in the $*$.
  - Easy to write the common case when pointers don't alias.
- Called the "Frame Rule"
  - Allows us to temporarily "forget" about the list, reason about the tree, and then remember the list.

$$\frac{\{P\}c\{r \Rightarrow Q\ r\}}{\{P * R\}c\{r \Rightarrow Q\ r * R\}}\ \text{Frame}$$

Hoare Type Theory: Specifying Effects in Types

- Embed Hoare logic into the *types* of terms.
- Hoare triples are represented by the `Cmd` type.

$$\{P\}\, c\, \{r \Rightarrow Q\} \quad \equiv \quad \mathtt{c} : \mathtt{Cmd}\ \mathtt{P}\ (\mathtt{r} \Rightarrow \mathtt{Q})$$

Pointer Operations

```
Write p v : Cmd (∃ w, p ↦ w)
                (_ ⇒ p ↦ v)
```

sortInPlace in HTT

```
sortInPlace(int cnt, int[cnt] ary, #list int# m)
  : Cmd (array ary m)
        (_ ⇒ array ary (sort m))
{  ... /** implementation **/ ... }
```

- m is computationally irrelevant, i.e. compile-time only.
  - Used only to simplify reasoning.
- `array a l` is an abstraction predicate that states the contents of the array (a) are the same as the contents of the list (l).

Outline

Example: C-style Linked Lists

```
interface IntList {
  Integer get(int index);
  void insert(int index, int value);
  ...
}
```

- Specifies an abstract type `IntList` with two methods.

Example: C-style Linked Lists

```
interface IntList {
  Integer get(int index);
  void insert(int index, int value);
  ...
}
```

- Specifies an abstract type `IntList` with two methods.
- To reason about correctness, we need specifications.
  1. How do we describe the value of the list?

Example: C-style Linked Lists

```
interface IntList {
  Integer get(int index);
  void insert(int index, int value);
  ...
}
```

- Specifies an abstract type `IntList` with two methods.
- To reason about correctness, we need specifications.
  1. How do we describe the value of the list?
     Relate to an irrelevant list

Example: C-style Linked Lists

```
interface IntList {
  Integer get(int index);
  void insert(int index, int value);
  ...
}
```

- Specifies an abstract type `IntList` with two methods.
- To reason about correctness, we need specifications.
  1. How do we describe the value of the list?
     Relate to an irrelevant list
  2. How do we describe the heap that contains a particular list?

Example: C-style Linked Lists

```
interface IntList {
  Integer get(int index);
  void insert(int index, int value);
  ...
}
```

- Specifies an abstract type `IntList` with two methods.
- To reason about correctness, we need specifications.
  1. How do we describe the value of the list?
     Relate to an irrelevant list
  2. How do we describe the heap that contains a particular list?
     Specify a representation predicate

Example: C-style Linked Lists

```
interface IntList {
  Integer get(int index);
  void insert(int index, int value);
  ...
}
```

- Specifies an abstract type `IntList` with two methods.
- To reason about correctness, we need specifications.
  1. How do we describe the value of the list?
     Relate to an irrelevant list
  2. How do we describe the heap that contains a particular list?
     Specify a representation predicate
  3. What does each function do? What does "correct" mean?

Example: C-style Linked Lists

```
interface IntList {
  Integer get(int index);
  void insert(int index, int value);
  ...
}
```

- Specifies an abstract type `IntList` with two methods.
- To reason about correctness, we need specifications.
  1. How do we describe the value of the list?
     Relate to an irrelevant list
  2. How do we describe the heap that contains a particular list?
     Specify a representation predicate
  3. What does each function do? What does “correct” mean?
     Give a Hoare-logic specification

An Elaborated Interface

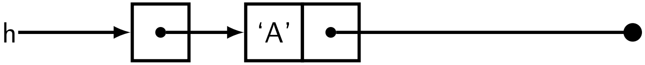
```
Interface IntList H {
  llist : H → list int → hprop ;

  get (H h, int index, #list int# m)
    : Cmd (llist h m)
          (r ⇒ llist h m * [r = nth m index]) ;
  insert (H h, int index, int val, #list int# m)
    : Cmd (llist h m)
          (_ ⇒ llist h (spec_insert m index val)) ;

  /** ... **/
}
```

- `H` is the type of handles to lists.
- `llist` is the representation predicate.

Implementation: The Representation Predicate



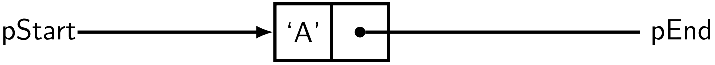
- Describe the heap computationally using a functional model.

Implementation: The Representation Predicate

- Describe the heap computationally using a functional model.

```
llseg pStart pEnd  nil    ⟺ [pStart = pEnd]
```

Implementation: The Representation Predicate



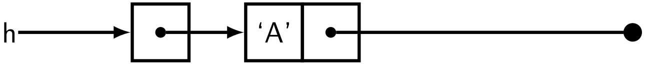
- Describe the heap computationally using a functional model.

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

llseg pStart pEnd   nil       ⟺ [pStart = pEnd]

llseg (Ptr p) pEnd (a :: b) ⟺ ∃ nx : optr,
                          p ↦ mkNode a nx * llseg nx pEnd b
```

Implementation: The Representation Predicate



- Describe the heap computationally using a functional model.

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

llseg pStart pEnd   nil       ⟺ [pStart = pEnd]

llseg (Ptr p) pEnd (a :: b) ⟺ ∃ nx : optr,
                          p ↦ mkNode a nx * llseg nx pEnd b

tlst        ≡ ptr
llist h m ⟺ ∃ st : optr, h ↦ st * llseg st Null m
```

Outline

Outline

An Interface for Iterators

- Iterators and collections go hand-in-hand.

```
Interface ListIterable titr {
  iter : titr → list int → nat → hprop ;
```

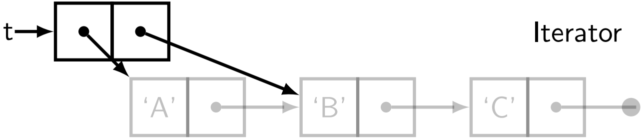
An Interface for Iterators

- Iterators and collections go hand-in-hand.

```
Interface ListIterable titr {
  iter : titr → list int → nat → hprop ;
  next (titr t, #list int# m, #nat# index)
    : Cmd (iter t m idx)
          (res ⇒ iter t m (nextIndex index (length m)) *
                     [res = nth m index])
}
```

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

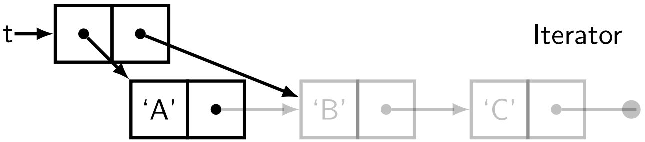
Implementing Iterators over Lists




```
titr ≡ ptr

iter (t : titr) (ls : list int) (idx : nat) ⟺
  ∃ st : optr, ∃ cur : optr, t ↦ (st, cur)
```

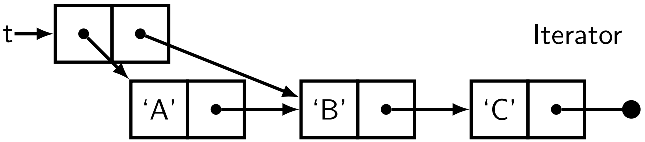
Implementing Iterators over Lists




```
titr ≡ ptr

iter (t : titr) (ls : list int) (idx : nat) ⟺
  ∃ st : optr, ∃ cur : optr, t ↦ (st, cur) *
  llseg st cur (firstn idx ls)
```

Implementing Iterators over Lists

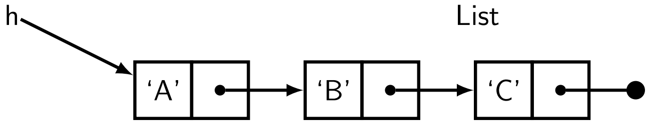



```
titr ≡ ptr

iter (t : titr) (ls : list int) (idx : nat) ⟺
  ∃ st : optr, ∃ cur : optr, t ↦ (st, cur) *
  llseg st cur (firstn idx ls) *
  llseg cur Null (skipn idx ls)
```

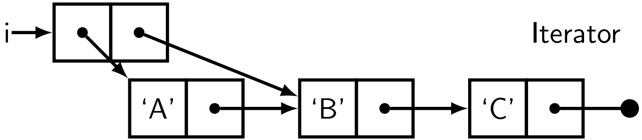
The Sharing Problem

- Requires access to the same memory as the underlying list.
  - Creating an iterator transfers ownership of memory from the list to iterator.
  - Can't have multiple iterators.



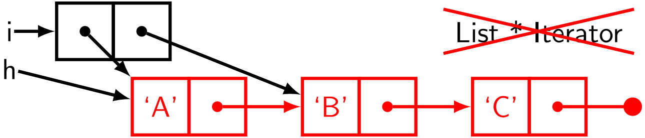
The Sharing Problem

- Requires access to the same memory as the underlying list.
  - Creating an iterator transfers ownership of memory from the list to iterator.
  - Can't have multiple iterators.



The Sharing Problem

- Requires access to the same memory as the underlying list.
  - Creating an iterator transfers ownership of memory from the list to iterator.
  - Can't have multiple iterators.



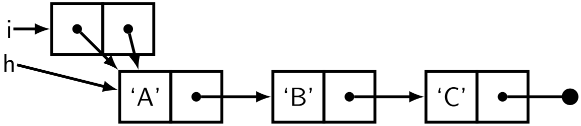
The Sharing Problem: Specifications

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

```
zip(titr i1, titr i2, #list int# l1, #list int# l2) :
  Cmd (iter i1 l1 0 * iter i2 l2 0 * [length l1 = length l2])
      (res ⇒
         iter i1 l1 (length l1) * iter i2 l2 (length l2) *
         llist res (fzip l1 l2))
```

A Real Sharing Problem

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

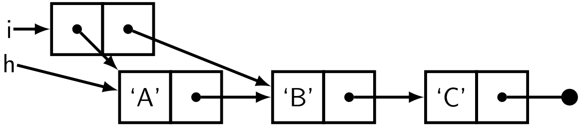


- Consider the following program:

```
Iterator<Integer> itr = lst.iterator();
```

A Real Sharing Problem

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

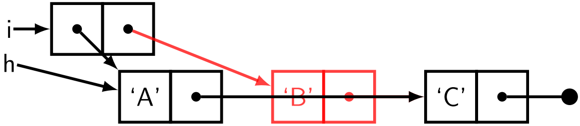


- Consider the following program:

```
Iterator<Integer> itr = lst.iterator();
itr.next();
```

A Real Sharing Problem

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

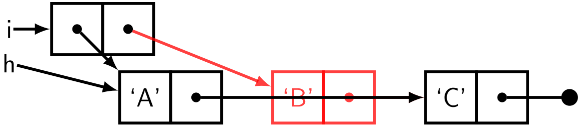


- Consider the following program:

```
Iterator<Integer> itr = lst.iterator();
itr.next();
lst.remove(1);
```

A Real Sharing Problem

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

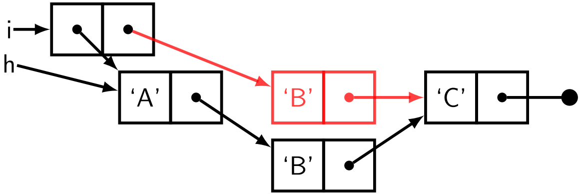


- Consider the following program:

```
Iterator<Integer> itr = lst.iterator();
itr.next();
lst.remove(1);
itr.next();
```

A Real Sharing Problem

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



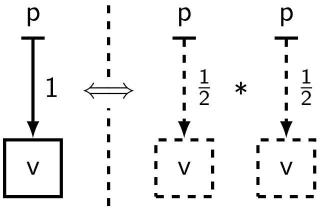
- Consider the following program:

```
Iterator<Integer> itr = lst.iterator();
itr.next();
lst.remove(1);
lst.insert(1,'B');
itr.next();
```

- Source of Java's `ConcurrentModificationException`.

Sharing with Fractional Permissions (Boyland '03)

- Parametrize points-to by a fractional ownership.
  - $\mathtt{p} \overset{q}{\mapsto} \mathtt{v}$, 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.



A Fractional Iterator

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

```
(** Representation predicate **)
liter (owner : tlst) (q : Fp)
    (t : titr) (ls : list int) (idx : nat) ⟺
  ∃ st : optr, ∃ cur : optr,
  owner ↦q st * t ↦1 cur *
  llseg st cur (firstn idx ls) q *
  llseg cur Null (skipn idx ls) q
```

- q is the fraction of the list that is owned.
- Allows multiple iterators over the same list.

Exposing Fractions

- Need to prove that lists can be split…
  - `q |#| q'` states that q and q' are compatible, i.e. sum to less than or equal to 1.

Lemma llist_perm_split : $\forall$ q q' t ls,
  q |#| q' $\rightarrow$
  llist $q+q'$ t ls $\Longrightarrow$ llist q t ls * llist q' t ls

Exposing Fractions

- Need to prove that lists can be split...
  - `q |#| q'` states that q and q' are compatible, i.e. sum to less than or equal to 1.

```
Lemma llist_perm_split : ∀ q q' t ls,
  q |#| q' →
  llist q+q' t ls ⟹ llist q t ls * llist q' t ls
```

- ...and joined together.

```
Lemma llist_perm_join : ∀ q q' t ls,
  q |#| q' →
  llist q t ls * llist q' t ls ⟹ llist q+q' t ls
```

Recap: Fractional Iterators

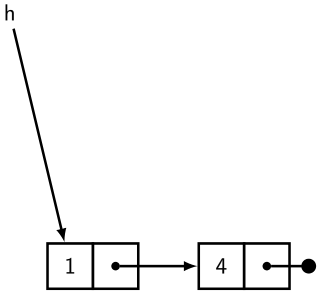
- **Original Problem** Couldn't have multiple views of the same list.
  - Either a list or an iterator, not both.
  - Only 1 iterator at a time.
- **Solution** Fractional permissions allow sharing.
  - Lift fractional permissions to the level of abstract data types.
  - Only slight modifications to incorporate fractions.
  - Prove two simple lemmas about splitting and joining.
  - Able to pass-out read-only permissions, finer granularity permissions.

Outline

1. Techniques for Gaining Confidence
2. Software Verification with Types
   - Modularity and Abstraction
3. My Work: Addressing Sharing and Aliasing
   - Sharing: Iterators
   - Aliasing: B+ Trees
4. Conclusions

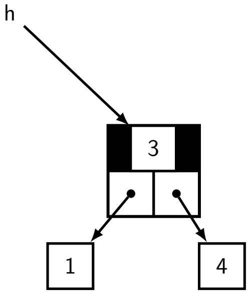
Specifications with Aliasing

- Aliasing presents a unique problem for separation logic.
  - Lists are easy...



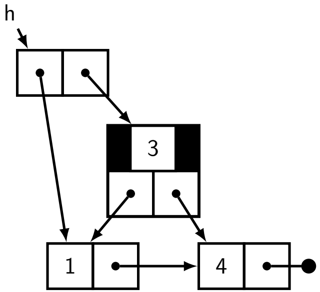
Specifications with Aliasing

- Aliasing presents a unique problem for separation logic.
  - Lists are easy...
  - Trees are easy...



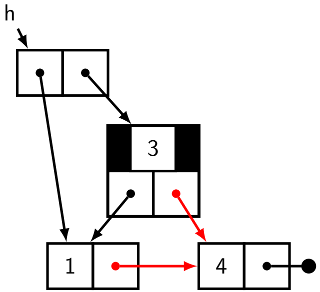
Specifications with Aliasing

- Aliasing presents a unique problem for separation logic.
  - Lists are easy...
  - Trees are easy...
  - Trees with lists are not easy ...

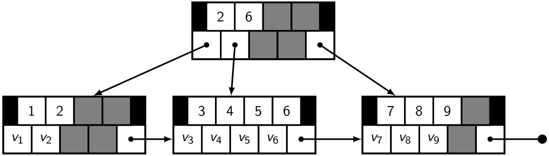


Specifications with Aliasing

- Aliasing presents a unique problem for separation logic.
  - Lists are easy...
  - Trees are easy...
  - Trees with lists are not easy because of aliasing...



B+ Trees



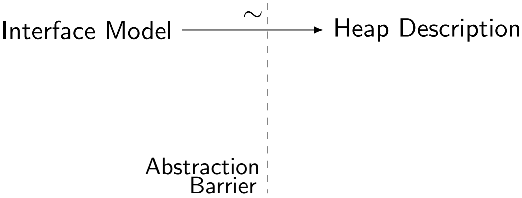
- B+ trees are *n*-ary trees where the leaves are connected by a linked list.
  - Support fast lookup and in-order iteration.
  - Commonly used for database indices. (Malecha '10)
- Previous formalizations exist, but neither is mechanically verified:
  - Classical conjunction, $(\texttt{list} * \texttt{any})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

Difficulties of the Invariant

- Have to encode pointer aliasing explicitly.

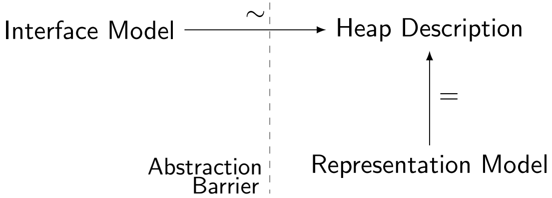
Difficulties of the Invariant

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



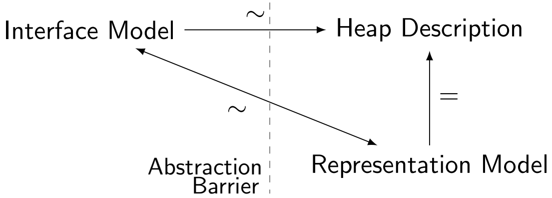
Difficulties of the Invariant

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



Difficulties of the Invariant

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



- Other properties that we won't focus on.
  - Enforce the tree balancedness.
  - Enforce the ordering of keys.
  - Invariants on the size of branches and leaves.

Defining a Representation Model

- A standard, functionaly *n*-ary tree is enough for the trunk.

```
tree = Branch (list tree) | Leaf (list value)
```

Defining a Representation Model

- A standard, functionaly *n*-ary tree is enough for the trunk.

```
tree = Branch (list tree) | Leaf (list value)
```

- This stores the structure, but the aliasing is still difficult.
  - We need to give equations on pointers, in the representation.

Defining a Representation Model

- A standard, functionaly *n*-ary tree is enough for the trunk.

```
tree = Branch (list tree) | Leaf (list value)
```

- This stores the structure, but the aliasing is still difficult.
  - We need to give equations on pointers, in the representation.
- **Solution**: Elaborate the functional tree with the pointers.

```
ptree = Branch ptr * (list tree)
      | Leaf ptr * (list value)
```

- Enforce that the pointer stored in each node is the pointer that points to the node.
  - Quantifies all pointers simultaneously.
  - Makes it easy to state aliasing constraints.

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.

```
rep (p : BptMap) (m : Model) ⟺
  ∃ pRoot : ptr, ∃ tr : ptree,
  p ↦ (pRoot, #tr#) *
  repTree pRoot Null 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`).

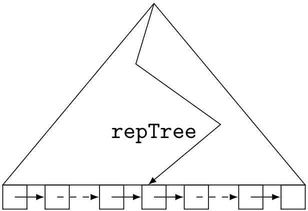
```
rep (p : BptMap) (m : Model) ⟺
  ∃ pRoot : ptr, ∃ tr : ptree,
  p ↦ (pRoot, #tr#) *
  repTree pRoot Null tr *
  [m = as_map tr]
```

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`.

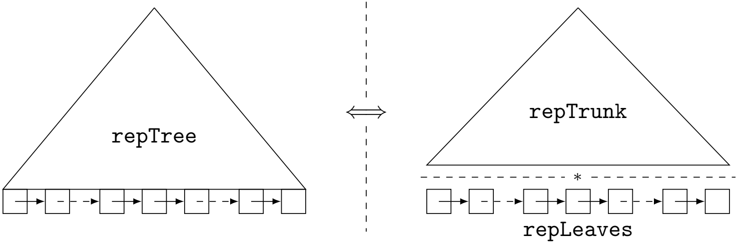
```
rep (p : BptMap) (m : Model) ⟺
  ∃ pRoot : ptr, ∃ tr : ptree,
  p ↦ (pRoot, #tr#) *
  repTree pRoot Null tr *
  [m = as_map tr] *
  [inv tr MinK MaxK]
```

Implementation: `insert` and `lookup`



- Most operations act on the tree.
  - Efficient lookup ($O(\lg n)$).
  - Efficient insert ($O(\lg n)$).
- Implementation follows recursive structure of the tree
  - Simple recursion invariant.
  - Relatively simple to verify.
  - The complexities come from the width of the branches.

Implementation: Iteration



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

```
Lemma repTree_repTrunkLeaves : ∀ (h : nat)
    (p : ptr) (last : optr) (m : ptree h),
  repTree p last m
  ⟺
  repTrunk p last m *
  repLeaves (Ptr (firstPtr m)) (leaves m) last.
```

Recap: B+ trees

- **Original Problem** Aliasing at the leaves and relational heap predicate makes describing the heap difficult.
  - Existing approaches seem cumbersome to verify.
- **Solution** Factor out the relation by quantifying an irrelevant model.
  - Including the pointers in the model makes them easy to access.
  - Simple, computational heap predicate.
  - Support multiple views by proving an equivalence of formulae.
  - Avoid unnecessary guessing during proof search.
  - Use irrelevance to avoid run-time overhead.

Outline

The Take-away

- Higher-order abstraction simplifies specifications and proofs.
- 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.
  - Win for automation.
- Automation pays off when reasoning about separation logic.

Outlook

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.

Code Slides

Implementation: insert

```
get (H h, int index, #list int# m)
  : Cmd (llist h m)
        (r ⇒ llist h m * [r = nth m index])
{
  let hd := *h in
  // Extract the index element from the list from hd to Null
  while (hd != Null) {

    let nde := *hd in
    if (index == 0) return (Some nde.val);
    hd := nde.next;
    index--;
  }
  return None;
}
```

Implementation: insert

```
get (H h, int index, #list int# m)
  : Cmd (llist h m)
        (r ⇒ llist h m * [r = nth m index])
{
  let hd := *h in
  // Extract the index element from the list from hd to Null
  while (hd != Null) {
    // Need to specify the loop invariant
    let nde := *hd in
    if (index == 0) return (Some nde.val);
    hd := nde.next;
    index--;
  }
  return None;
}
```

Implementation: insert

```
get (H h, int index, #list int# m)
  : Cmd (llist h m)
      (r ⇒ llist h m * [r = nth m index])
{
  let hd := *h in
  Fix3 (fun hd j m ⇒ llseg hd m Null)
       (fun hd j m (r : option int) ⇒ llseg hd m Null *
          [r = nth m j])
       (fun self hd j m ⇒
          IfNull hd Then
            {{ Return None }}
          Else
            let nde := *hd in
            IfZero j Then
              {{ Return (Some (val nde)) }}
            Else
              {{ self (next nde) j (tail m) <@> _ }})
      hd i m <@> _
}
```