Verification with Sharing and Aliasing

Gregory Malecha

Harvard University SEAS

March 5, 2010
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
Basic trade-off between the amount of effort required and the expressivity of the properties.
Goal

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

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

Techniques for Gaining Confidence

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.
Theorem Proving

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

Techniques for Reasoning About Code
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.

“Correctness” 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.

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

**Implement**  
- We can verify an implementation with respect to a specification.

Gregory Malecha (Harvard University SEAS)  
Verification with Sharing and Aliasing  
March 5, 2010
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

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
How do you figure out what a function does?

```c
int largest(int cnt, int* ary) {
    ... /**< implementation */ ... 
}
```
How do you figure out what a function does?

```c
int largest(int cnt, int* ary) {
    ... /**< implementation */ ... 
}
```
Building on Types

How do you figure out what a function does?

```c
/** 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 */ ... 
}
```
Annotation languages like `PREfix/PREfast` allow specifying properties like array bounds information.

```c
/** 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.

```c
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

\{ \ p_1 \mapsto 1 \land p_2 \mapsto 1 \ \}\n*p_1 = 3
\{ \_ \Rightarrow p_1 \mapsto 3 \land p_2 \mapsto 1 \ \}\n
- Can we prove this?
Describing the World

Example Program

\{ \texttt{p}_1 \mapsto 1 \land \texttt{p}_2 \mapsto 1 \}\}
\* \texttt{p}_1 = 3
\{ _\Rightarrow \texttt{p}_1 \mapsto 3 \land \texttt{p}_2 \mapsto ??? \}\}

- Can we prove this? No.
  - What if \texttt{p}_1 is an alias of \texttt{p}_2?
Describing the World

Example Program

\{
  p_1 \mapsto 1 \land p_2 \mapsto 1 \land [p_1 \neq p_2]
\}

*p_1 = 3

{ _ \Rightarrow p_1 \mapsto 3 \land 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.
We want to be able to reason about two structures independently.
We want to be able to reason about two structures independently.

Encode the disjointness condition in the $\ast$.

- 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 $\ast$.
  - 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.

$$\{\text{tree} \ast \text{list}\} c\{r \Rightarrow \text{tree'} \ast \text{list}\}$$  Frame
We want to be able to reason about two structures independently.
Encode the disjointness condition in the $\ast$.
- 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.

\[
\{
\text{tree}\} \text{c}\{r \Rightarrow \text{tree}'\} \\
\{\text{tree} \ast \text{list}\} \text{c}\{r \Rightarrow \text{tree}' \ast \text{list}\}
\]

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

- Encode the disjointness condition in the $\ast$.
  - 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.

\[
\begin{align*}
\{P\} c\{r \Rightarrow Q r\} \\
\{P \ast R\} c\{r \Rightarrow Q r \ast R\}
\end{align*}
\]

Frame
Embed Hoare logic into the *types* of terms.
Hoare triples are represented by the \( \text{Cmd} \) type.

\[
\{ P \} \; c \; \{ r \Rightarrow Q \} \equiv c : \text{Cmd} \; P \; (r \Rightarrow Q)
\]

**Pointer Operations**

\[
\text{Write } p \; v : \text{Cmd} \; (\exists \; w, \; p \leftrightarrow w) \\
(\_ \Rightarrow p \leftrightarrow v)
\]
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

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
Example: C-style Linked Lists

```java
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

```java
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.
  - How do we describe the value of the list?
Example: C-style Linked Lists

```java
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.
  - How do we describe the value of the list?
    - Relate to an irrelevant list
Example: C-style Linked Lists

```java
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

```java
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.
  - How do we describe the value of the list?
    - Relate to an irrelevant list
  - How do we describe the heap that contains a particular list?
    - Specify a representation predicate
Example: C-style Linked Lists

```java
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

```java
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 \to list int \to hprop ;

    get (H h, int index, #list int# m)
        : Cmd (llist h m)
            (r \Rightarrow llist h m \ast [r = nth m index]) ;

    insert (H h, int index, int val, #list int# m)
        : Cmd (llist h m)
            (_ \Rightarrow 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.

\[ llseg \ pStart \ pEnd \ \text{nil} \iff [pStart = pEnd] \]
Describe the heap computationally using a functional model.

Record l1Node := mkNode { val : int ; next : optr }

l1seg pStart pEnd nil ⇐⇒ [pStart = pEnd]

l1seg (Ptr p) pEnd (a :: b) ⇐⇒ ∃ nx : optr,
    p ↦ mkNode a nx * l1seg nx pEnd b
Describe the heap computationally using a functional model.

\[ h \Rightarrow \text{`A'} \Rightarrow \bullet \]

- \text{Record} \ l\text{lnode} := \text{mkNode} \{ \text{val} : \text{int} ; \text{next} : \text{optr} \}

\[ \text{l\text{llseg}} \ p\text{Start} \ p\text{End} \ \text{nil} \iff [p\text{Start} = p\text{End}] \]

\[ \text{l\text{llseg}} \ (\text{Ptr} \ p) \ p\text{End} \ (a :: b) \iff \exists \ nx : \text{optr}, \]
\[ p \mapsto \text{mkNode} \ a \ nx \ast \text{l\text{llseg}} \ nx \ p\text{End} \ b \]

\[ \text{tl\text{llst}} \equiv \text{ptr} \]

\[ \text{ll\text{list}} \ h \ m \iff \exists \ st : \text{optr}, \ h \mapsto st \ast \text{l\text{llseg}} \ st \ \text{Null} \ m \]
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
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.
- \( \text{titr} \) is the type of the iterator handle.
- Representation predicate (\( \text{iter} \)) and \( \text{next} \) command.
Implementing Iterators over Lists

\[ t \xrightarrow{\text{iterator}} \left[ \text{‘A’} \rightarrow \text{‘B’} \rightarrow \text{‘C’} \rightarrow \text{Null} \right] \]

\begin{align*}
titr & \equiv \text{ptr} \\
\text{iter} (t : \text{titr}) (ls : \text{list int}) (idx : \text{nat}) & \iff \\
\exists st : \text{optr}, \exists cur : \text{optr}, t & \mapsto (st, cur)
\end{align*}
Implementing Iterators over Lists

\[
titr \equiv ptr
\]

\[
\text{iter} (t : titr) (ls : list int) (idx : nat) \iff \exists st : optr, \exists cur : optr, t \mapsto (st, cur) \quad \text{llseg st cur (firstn idx ls)}
\]
Implementing Iterators over Lists

\[ \text{titr} \equiv \text{ptr} \]

\[
\text{iter} \ (t : \text{titr}) \ (ls : \text{list int}) \ (idx : \text{nat}) \iff \\
\exists \ st : \text{optr}, \ \exists \ cur : \text{optr}, \ t \mapsto (st, \ cur) \ *
\text{llseg} \ st \ cur \ (\text{firstn} \ idx \ ls) \ *
\text{llseg} \ cur \ \text{Null} \ (\text{skipn} \ idx \ ls)
\]
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.

![Diagram showing sharing problem with iterators and list]

Gregory Malecha (Harvard University SEAS)  Verification with Sharing and Aliasing  March 5, 2010  26 / 47
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:

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

![Diagram showing a list with pointers to 'A', 'B', and 'C', with iterators i and h.]
A Real Sharing Problem

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

Consider the following program:

```java
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:

```java
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:

```java
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:

```java
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.
  - \( p \xrightarrow{q} v \), \( q \) 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.

\[
p \quad \xrightarrow{1} \quad v \quad \xleftarrow{\frac{1}{2}} \quad p
\]

\[
p \quad \xrightarrow{\frac{1}{2}} \quad * \quad \xleftarrow{\frac{1}{2}} \quad p
\]
A Fractional Iterator

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

\[
(** \text{Representation predicate} **) \\
liter \ (\text{owner : tlst}) \ (q : Fp) \\
\quad (t : titr) \ (ls : \text{list int}) \ (idx : \text{nat}) \iff \\
\exists \ st : \text{optr}, \exists \ cur : \text{optr}, \\
\quad \text{owner} \ q \mapsto st \ast t \overset{1}{\mapsto} cur \ast \\
\quad \text{llseg} \ st \ cur \ (\text{firstn} \ idx \ ls) \ q \ast \\
\quad \text{llseg} \ cur \ \text{Null} \ (\text{skipn} \ idx \ ls) \ q
\]

- \(q\) 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...
  - \( q \mid\#\mid q' \) states that \( q \) and \( q' \) are compatible, i.e. sum to less than or equal to 1.

**Lemma** llist_perm_split : \( \forall q \ q' \ t \ ls, \)
\[
q \mid\#\mid q' \rightarrow \\
lлист \ q + q' \ t \ ls \implies \ lлист \ q \ t \ ls \ast \ lлист \ q' \ t \ ls
\]
Exposing Fractions

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

**Lemma llist_perm_split** : \( \forall \ q \ q' \ t \ ls, \ q \mid\#\mid q' \rightarrow \ llist \ q + q' \ t \ ls \Rightarrow llist \ q \ t \ ls \ast llist \ q' \ t \ ls \)

- ...and joined together.

**Lemma llist_perm_join** : \( \forall \ q \ q' \ t \ ls, \ q \mid\#\mid q' \rightarrow \ llist \ q \ t \ ls \ast llist \ q' \ t \ ls \Rightarrow llist \ q + q' \ t \ ls \)
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...

```
  h
 /   \
3----5----7
 /     /     /
1-----h-----4
```
Specifications with Aliasing

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

![Diagram of a tree with nodes labeled 1, 3, and 4 connected by arrows]
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, \((\text{list } \ast \text{any}) \land \text{(tree)}\). (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.

![Diagram]

- Interface Model \(\sim\) Heap Description
- Abstraction Barrier
- Representation Model
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.

\[
\text{tree} = \text{Branch (list tree)} \mid \text{Leaf (list value)}
\]
Defining a Representation Model

- A standard, functionally $n$-ary tree is enough for the trunk.
  \[
  \text{tree} = \text{Branch (list tree)} \mid \text{Leaf (list value)}
  \]

- 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.
  
  \[
  \text{tree} = \text{Branch (list tree)} \mid \text{Leaf (list value)}
  \]

- 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.
  
  \[
  \text{ptree} = \text{Branch ptr \ast (list tree)} \\
  \mid \text{Leaf ptr \ast (list value)}
  \]

- 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 \((\text{repTree})\) very computational.

\[
\text{rep} \ (p : \text{BptMap}) \ (m : \text{Model}) \iff \\
\exists \ p\text{Root} : \text{ptr}, \ \exists \ tr : \text{ptree}, \\
\ p \mapsto (p\text{Root}, \#tr\#) * \\
\text{repTree} \ p\text{Root} \ \text{Null} \ tr
\]
### 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 \((\text{repTree})\) very computational.
  - Connect the logical model \((m)\) to the physical model \((tr)\).

\[
\text{rep} \ (p : \text{BptMap}) \ (m : \text{Model}) \iff \\
\exists \ p\text{Root} : \text{ptr}, \exists \ tr : \text{ptree}, \\
p \mapsto (p\text{Root}, \#tr\#) * \\
\text{repTree} \ p\text{Root} \ Null \ tr * \\
[m = as\_map \ tr]
\]
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 \((\text{repTree})\) very computational.
  - Connect the logical model \((m)\) to the physical model \((tr)\).
  - Consolidate pure facts about the model in \(\text{inv}\).

\[
\text{rep}\ (p : \text{BptMap})\ (m : \text{Model}) \iff \exists p\text{Root} : \text{ptr}, \exists tr : \text{ptree}, \\
p \mapsto (p\text{Root}, #tr#) * \text{repTree\ pRoot\ Null\ tr\ *} \\
[m = \text{as\_map\ tr}] * \\
[\text{inv\ tr\ MinK\ MaxK}] 
\]
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** \( \text{repTree}_\ast \text{repTrunkLeaves} : \forall (h : \text{nat}) \)  
\( (p : \text{ptr}) (\text{last} : \text{optr}) (m : \text{ptree } h), \)  
\( \text{repTree } p \text{ last } m \)  
\( \iff \)  
\( \text{repTrunk } p \text{ last } m \ast \)  
\( \text{repLeaves} (\text{Ptr } (\text{firstPtr } m)) (\text{leaves } m) \) \( \text{last} \).
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

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
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 reifying an irrelevant model.
  - Win for automation.
Automation pays off when reasoning about separation logic.
Conclusions

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`

```plaintext
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

```ocaml
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 <@> _
}
```

Gregory Malecha (Harvard University SEAS)  Verification with Sharing and Aliasing  March 5, 2010  47 / 47