Symbolic Heap Abstraction with Demand-Driven Axiomatization of Memory Invariants

Isil Dillig Thomas Dillig Alex Aiken

Stanford University

Relational vs. Non-Relational Heap analysis

- Goal of heap analysis: Statically describe all possible points-to relations in the heap for any execution of the program.

Relational vs. Non-Relational Heap analysis

- Goal of heap analysis: Statically describe all possible points-to relations in the heap for any execution of the program.
- Heap analyses can be characterized as *relational* or *non-relational*:

Relational vs. Non-Relational Heap analysis

- Goal of heap analysis: Statically describe all possible points-to relations in the heap for any execution of the program.
- Heap analyses can be characterized as *relational* or *non-relational*:
  - A relational analysis tracks correlations between points-to targets of two memory locations

Relational vs. Non-Relational Heap analysis

- Goal of heap analysis: Statically describe all possible points-to relations in the heap for any execution of the program.
- Heap analyses can be characterized as *relational* or *non-relational*:
  - A relational analysis tracks correlations between points-to targets of two memory locations
  - A non-relational heap analysis does not.

Relational vs. Non-Relational Heap analysis

- Goal of heap analysis: Statically describe all possible points-to relations in the heap for any execution of the program.
- Heap analyses can be characterized as *relational* or *non-relational*:
  - A relational analysis tracks correlations between points-to targets of two memory locations
  - A non-relational heap analysis does not.
- Relational heap analyses are more precise, but also more expensive.

An Example

- Consider the code snippet:

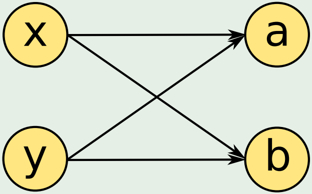
```
if(*)
 *x = a;
else
 *x = b;

y = x;
assert(*x == *y);
```

An Example

- Consider the code snippet:

```
if(*)
 *x = a;
else
 *x = b;

y = x;
assert(*x == *y);
```

Non-relational:

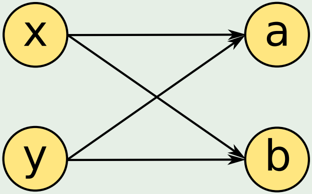


An Example

- Consider the code snippet:

```
if(*)
 *x = a;
else
 *x = b;

y = x;
assert(*x == *y);
```

Non-relational:



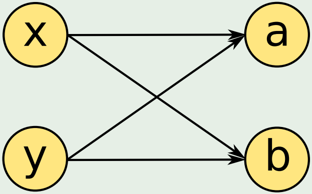
- Does not encode x and y must point to same location

An Example

- Consider the code snippet:

```
if(*)
 *x = a;
else
 *x = b;

y = x;
assert(*x == *y);
```

Non-relational:



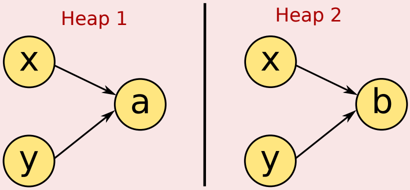
- Does not encode x and y must point to same location
- Cannot prove the assertion

An Example

- Consider the code snippet:

```
if(*)
 *x = a;
else
 *x = b;

y = x;
assert(*x == *y);
```

Relational:



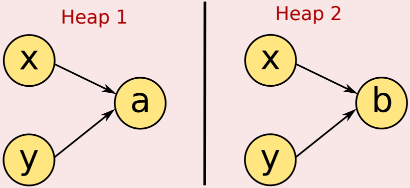
- Perform case split on possible heaps.

An Example

- Consider the code snippet:

```
if(*)
 *x = a;
else
 *x = b;

y = x;
assert(*x == *y);
```

Relational:



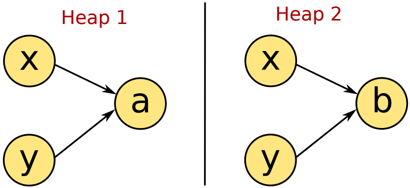
- Perform case split on possible heaps.
- Can prove assertion because in both heaps `x` and `y` point to same location.

Relational Analysis via Heap Splitting

- Advantages:

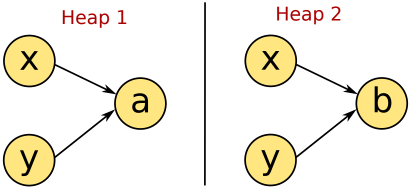
Relational Analysis via Heap Splitting

- Advantages:
  - Each abstract location points to exactly one target location per heap



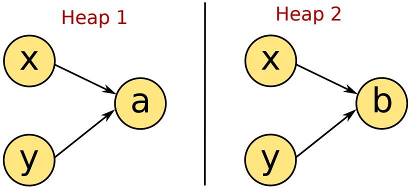
Relational Analysis via Heap Splitting

- Advantages:
  - Each abstract location points to exactly one target location per heap
  - ⇒ precise relational reasoning



Relational Analysis via Heap Splitting

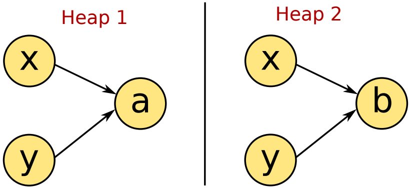
- Advantages:
  - Each abstract location points to exactly one target location per heap
  - ⇒ precise relational reasoning



- Disadvantages:

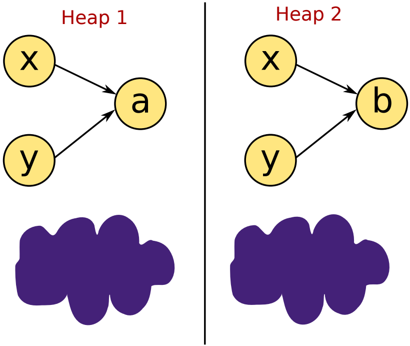
Relational Analysis via Heap Splitting

- Advantages:
  - Each abstract location points to exactly one target location per heap
  - $\Rightarrow$ precise relational reasoning
- Disadvantages:
  - Generates exponential number of heaps



Relational Analysis via Heap Splitting

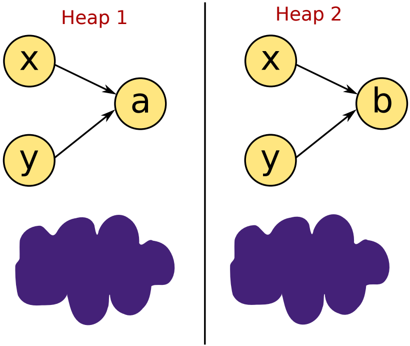
- Advantages:
  - Each abstract location points to exactly one target location per heap
  - $\Rightarrow$ precise relational reasoning
- Disadvantages:
  - Generates exponential number of heaps
  - Duplicates shared portion of the heaps



Relational Analysis via Heap Splitting

- Advantages:
  - Each abstract location points to exactly one target location per heap
  - ⇒ precise relational reasoning

- Disadvantages:
  - Generates exponential number of heaps
  - Duplicates shared portion of the heaps
  - ⇒ Very expensive and unscalable



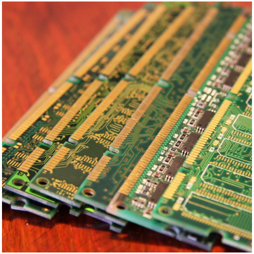
Relational Analysis via Heap Splitting

- Advantages:
  - Each abstract location points to exactly one target location per heap
  - ⇒ precise relational reasoning
- Disadvantages:
  - Generates exponential number of heaps
  - Duplicates shared portion of the heaps
  - ⇒ Very expensive and unscalable

This talk:

Scalable and precise relational heap analysis *without* performing explicit case splits on the heap

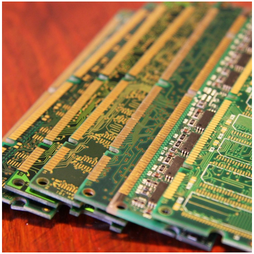
Memory Invariants



Insight:

We can achieve relational reasoning by enforcing two important memory invariants that real computer memories satisfy:

Memory Invariants

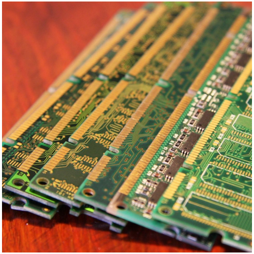


Insight:

We can achieve relational reasoning by enforcing two important memory invariants that real computer memories satisfy:

- Existence: Every memory location has at least one value

Memory Invariants

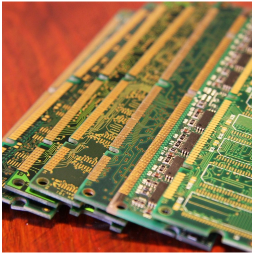


Insight:

We can achieve relational reasoning by enforcing two important memory invariants that real computer memories satisfy:

- Existence: Every memory location has at least one value
- Uniqueness: Every memory location has at most one value

Memory Invariants



Insight:

We can achieve relational reasoning by enforcing two important memory invariants that real computer memories satisfy:

- Existence: Every memory location has at least one value
- Uniqueness: Every memory location has at most one value

⇒ Heap splitting is one way of enforcing these invariants.

Enforcing Memory Invariants

Idea

Enforce memory invariants symbolically using constraints on a single heap abstraction.

Enforcing Memory Invariants

Idea

Enforce memory invariants symbolically using constraints on a single heap abstraction.

- No explicit case splits on the heap, but solver may internally need to perform case analysis

Enforcing Memory Invariants

Idea

Enforce memory invariants symbolically using constraints on a single heap abstraction.

- No explicit case splits on the heap, but solver may internally need to perform case analysis
- Still advantageous because:

Enforcing Memory Invariants

Idea

Enforce memory invariants symbolically using constraints on a single heap abstraction.

- No explicit case splits on the heap, but solver may internally need to perform case analysis
- Still advantageous because:
  - Solver can often prove a constraint SAT or UNSAT without considering all cases: eager vs. lazy

Enforcing Memory Invariants

Idea

Enforce memory invariants symbolically using constraints on a single heap abstraction.

- No explicit case splits on the heap, but solver may internally need to perform case analysis
- Still advantageous because:
  - Solver can often prove a constraint SAT or UNSAT without considering all cases: eager vs. lazy
  - Don't duplicate shared portions of the heap

Enforcing Memory Invariants

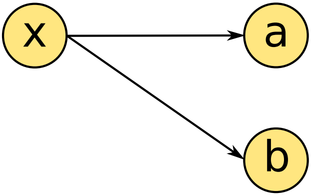
Idea

Enforce memory invariants symbolically using constraints on a single heap abstraction.

- No explicit case splits on the heap, but solver may internally need to perform case analysis
- Still advantageous because:
  - Solver can often prove a constraint SAT or UNSAT without considering all cases: eager vs. lazy
  - Don't duplicate shared portions of the heap
  - No heuristics for merging "similar" heaps

Enforcing Memory Invariants

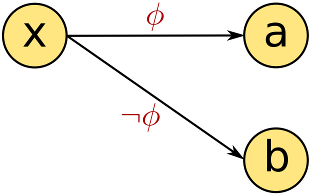
```
if(*)
 *x = a;
else
 *x = b;

y = x;
assert(*x == *y);
```



- To encode that x cannot point to a and b at the same time, we can use two constraints ϕ and $\neg\phi$

Enforcing Memory Invariants

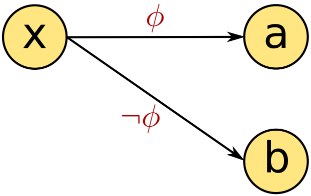
```
if(*)
 *x = a;
else
 *x = b;

y = x;
assert(*x == *y);
```



- To encode that x cannot point to a and b at the same time, we can use two constraints ϕ and $\neg\phi$

Enforcing Memory Invariants

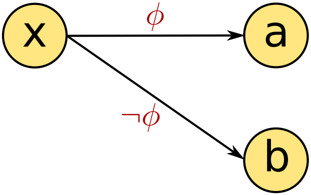
```
if(*)
 *x = a;
else
 *x = b;

y = x;
assert(*x == *y);
```



- To encode that x cannot point to a and b at the same time, we can use two constraints ϕ and $\neg\phi$ $\Rightarrow$ Uniqueness

Enforcing Memory Invariants

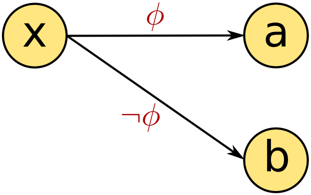
```
if(*)
 *x = a;
else
 *x = b;

y = x;
assert(*x == *y);
```



- To encode that x cannot point to a and b at the same time, we can use two constraints ϕ and $\neg\phi$ $\Rightarrow$ Uniqueness
- Also encodes that x must point to either a or b

Enforcing Memory Invariants

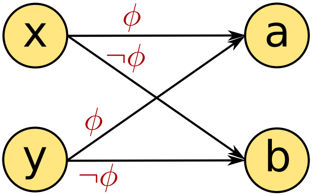
```
if(*)
 *x = a;
else
 *x = b;

y = x;
assert(*x == *y);
```



- To encode that x cannot point to a and b at the same time, we can use two constraints ϕ and $\neg\phi$ $\Rightarrow$ Uniqueness
- Also encodes that x must point to either a or b $\Rightarrow$ Existence

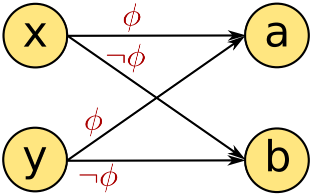
Enforcing Memory Invariants

```
if(*)
 *x = a;
else
 *x = b;

y = x;
assert(*x == *y);
```



Enforcing Memory Invariants

```
if(*)
 *x = a;
else
 *x = b;

y = x;
assert(*x == *y);
```



Correlation between x and y preserved

- x and y point to different locations under $\phi \wedge \neg\phi$

  $\Rightarrow$ Can prove the assertion!

Memory Invariants on Unbounded Locations

- Easy to enforce these invariants when each abstract location corresponds to one concrete location.

Memory Invariants on Unbounded Locations

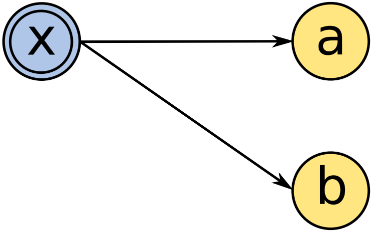
- Easy to enforce these invariants when each abstract location corresponds to one concrete location.
- But what about abstract locations that represent multiple concrete locations?

Memory Invariants on Summary Locations

```
for(int i=0; i<size; i++)
{
  if(*) x[i] = a;
  else x[i] = b;
}

y = x;
// 0 <= k < size
assert(x[k] == y[k]);
```

Memory Invariants on Summary Locations

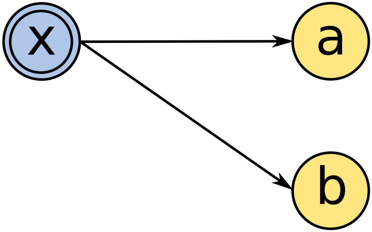
```
for(int i=0; i<size; i++)
{
  if(*) x[i] = a;
  else x[i] = b;
}

y = x;
// 0 <= k < size
assert(x[k] == y[k]);
```



- Most techniques represent the array with a summary node.

Memory Invariants on Summary Locations

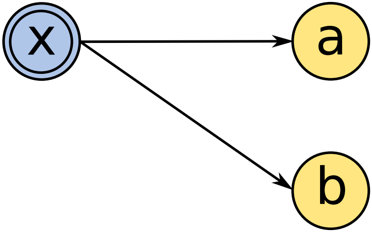
```
for(int i=0; i<size; i++)
{
  if(*) x[i] = a;
  else x[i] = b;
}

y = x;
// 0 <= k < size
assert(x[k] == y[k]);
```



- Most techniques represent the array with a summary node.
- Graph encodes that any element in x may point to either a or b.

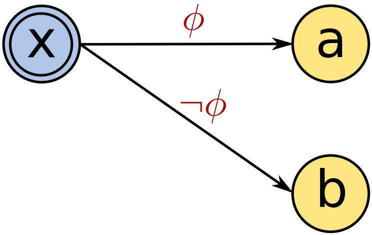
Memory Invariants on Summary Locations

```
for(int i=0; i<size; i++)
{
  if(*) x[i] = a;
  else x[i] = b;
}

y = x;
// 0 <= k < size
assert(x[k] == y[k]);
```



Memory Invariants on Summary Locations

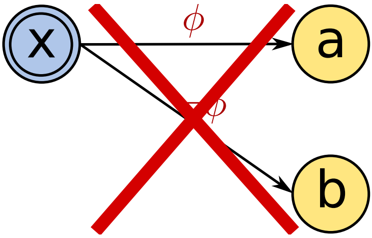
```
for(int i=0; i<size; i++)
{
  if(*) x[i] = a;
  else x[i] = b;
}

y = x;
// 0 <= k < size
assert(x[k] == y[k]);
```



- Encodes that an element of x cannot point to both a and b

Memory Invariants on Summary Locations

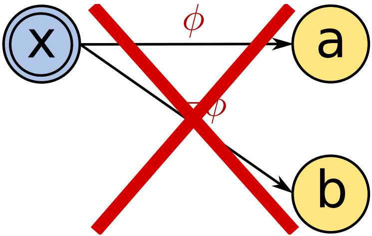
```
for(int i=0; i<size; i++)
{
  if(*) x[i] = a;
  else x[i] = b;
}

y = x;
// 0 <= k < size
assert(x[k] == y[k]);
```



- Encodes that an element of x cannot point to both a and b
- . . . but erroneously encodes x[1] and x[2] must have same value!

Memory Invariants on Summary Locations

```
for(int i=0; i<size; i++)
{
  if(*) x[i] = a;
  else x[i] = b;
}

y = x;
// 0 <= k < size
assert(x[k] == y[k]);
```



Conclusion

- To enforce memory invariants symbolically, we need a way to refer to individual elements in summary locations.

Symbolic Heap Abstraction

- Use the symbolic heap from our previous work that allows distinguishing individual elements in a summary location.

Symbolic Heap Abstraction

- Use the symbolic heap from our previous work that allows distinguishing individual elements in a summary location.
    - This basic symbolic heap does not enforce memory invariants

Symbolic Heap Abstraction

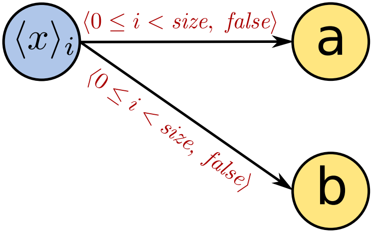
- Use the symbolic heap from our previous work that allows distinguishing individual elements in a summary location.
  - This basic symbolic heap does not enforce memory invariants
- Describe new technique to enforce memory invariants on the symbolic heap without explicit case splits

Symbolic Heap

- Abstract locations that represent more than one concrete location are qualified by index variables.

Symbolic Heap

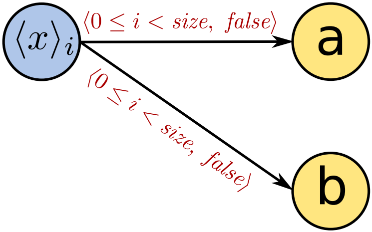
```
for(int i=0; i<size; i++)
{
  if(*) x[i] = a;
  else x[i] = b;
}

y = x;
// 0 <= k < size
assert(x[k] == y[k]);
```



- Abstract locations that represent more than one concrete location are qualified by index variables.

Symbolic Heap

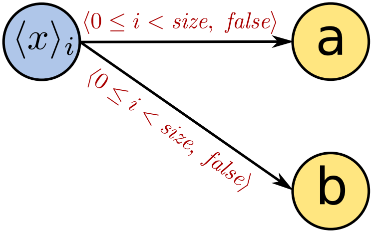
```
for(int i=0; i<size; i++)
{
  if(*) x[i] = a;
  else x[i] = b;
}

y = x;
// 0 <= k < size
assert(x[k] == y[k]);
```



- Abstract locations that represent more than one concrete location are qualified by index variables.
  - Index variables allow us to refer to individual elements inside the abstract location

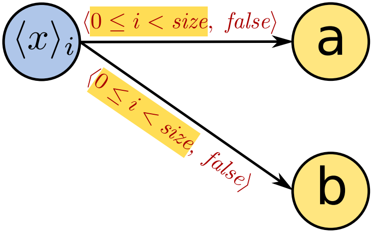
Symbolic Heap

```
for(int i=0; i<size; i++)
{
  if(*) x[i] = a;
  else x[i] = b;
}

y = x;
// 0 <= k < size
assert(x[k] == y[k]);
```



- Bracketing constraints on points-to edges qualify which elements in the source location may and must point to which elements in the target location.

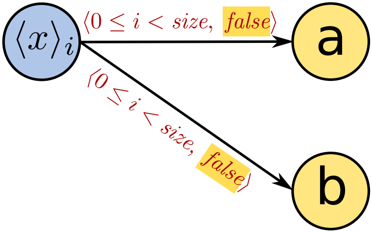
```
for(int i=0; i<size; i++)
{
  if(*) x[i] = a;
  else x[i] = b;
}

y = x;
// 0 <= k < size
assert(x[k] == y[k]);
```



- Bracketing constraints on points-to edges qualify which elements in the source location may and must point to which elements in the target location.

Symbolic Heap

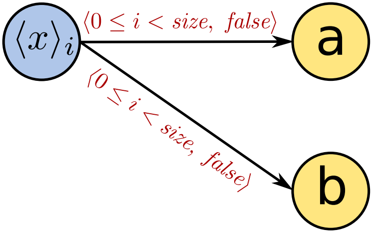
```
for(int i=0; i<size; i++)
{
  if(*) x[i] = a;
  else x[i] = b;
}

y = x;
// 0 <= k < size
assert(x[k] == y[k]);
```



- Bracketing constraints on points-to edges qualify which elements in the source location may and must point to which elements in the target location.

Symbolic Heap

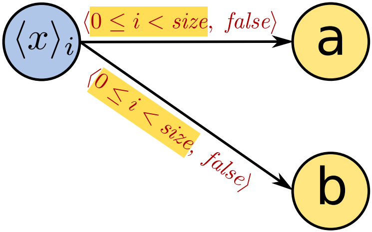
```
for(int i=0; i<size; i++)
{
  if(*) x[i] = a;
  else x[i] = b;
}

y = x;
// 0 <= k < size
assert(x[k] == y[k]);
```



This heap does not enforce memory invariants

Symbolic Heap

```
for(int i=0; i<size; i++)
{
  if(*) x[i] = a;
  else x[i] = b;
}

y = x;
// 0 <= k < size
assert(x[k] == y[k]);
```

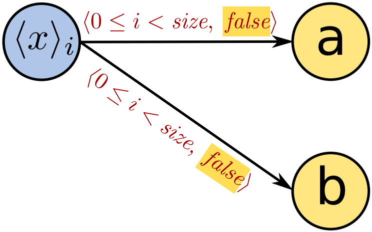


This heap does not enforce memory invariants

- Uniqueness violated because conjunction of may conditions is not unsatisfiable.

Symbolic Heap

```
for(int i=0; i<size; i++)
{
  if(*) x[i] = a;
  else x[i] = b;
}

y = x;
// 0 <= k < size
assert(x[k] == y[k]);
```



This heap does not enforce memory invariants

- Uniqueness violated because conjunction of may conditions is not unsatisfiable.
- Existence violated because disjunction of must conditions is not valid.

Making the Symbolic Heap Relational

Goal:

Modify the basic symbolic heap such that:

Making the Symbolic Heap Relational

Goal:

Modify the basic symbolic heap such that:

1. Enforces the existence and uniqueness of memory contents
   - Symbolically using constraints

Making the Symbolic Heap Relational

Goal:

Modify the basic symbolic heap such that:

1. Enforces the existence and uniqueness of memory contents
   - Symbolically using constraints
   - Replace original constraints with new constraints Δ enforcing these invariants.

Making the Symbolic Heap Relational

Goal:

Modify the basic symbolic heap such that:

1. Enforces the existence and uniqueness of memory contents
   - Symbolically using constraints
   - Replace original constraints with new constraints Δ enforcing these invariants.
2. Preserves all the partial information encoded in the original symbolic heap

Making the Symbolic Heap Relational

Goal:

Modify the basic symbolic heap such that:

1. Enforces the existence and uniqueness of memory contents
   - Symbolically using constraints
   - Replace original constraints with new constraints Δ enforcing these invariants.
2. Preserves all the partial information encoded in the original symbolic heap
   - Restore existing information by adding quantified axioms relating Δ to the original constraints

Enforcing Existence and Uniqueness on the Symbolic Heap

- Consider any location A for which invariants are violated.

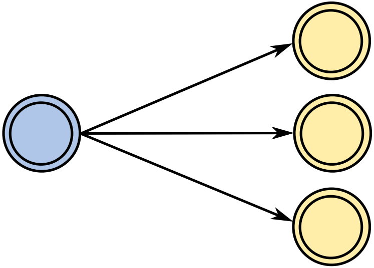
Enforcing Existence and Uniqueness on the Symbolic Heap

- Consider any location A for which invariants are violated.
- Replace constraint on i'th edge from A with constraint Δ_i enforcing memory invariants on each concrete element in A.

Enforcing Existence and Uniqueness on the Symbolic Heap

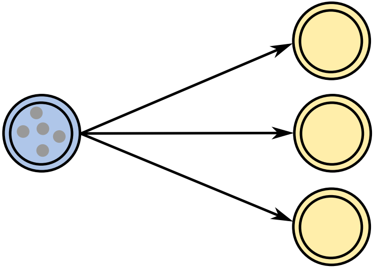
- Consider any location A for which invariants are violated.
- Replace constraint on i'th edge from A with constraint Δ_i enforcing memory invariants on each concrete element in A.
- These Δ_i's are of the form $\Gamma_i \wedge \Theta_i$

- Consider any location A for which invariants are violated.
- Replace constraint on i'th edge from A with constraint Δ_i enforcing memory invariants on each concrete element in A.
- These Δ_i's are of the form $\Gamma_i \wedge \Theta_i$



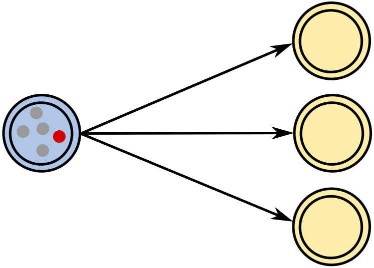
Enforcing Existence and Uniqueness on the Symbolic Heap

- Consider any location A for which invariants are violated.
- Replace constraint on i'th edge from A with constraint Δ_i enforcing memory invariants on each concrete element in A.
- These Δ_i's are of the form $\Gamma_i \wedge \Theta_i$



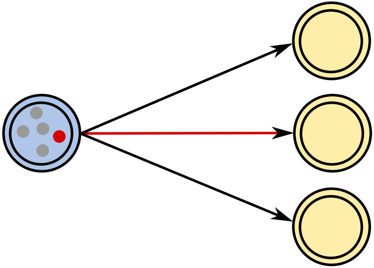
Enforcing Existence and Uniqueness on the Symbolic Heap

- Consider any location A for which invariants are violated.
- Replace constraint on i'th edge from A with constraint Δ_i enforcing memory invariants on each concrete element in A.
- These Δ_i's are of the form $\Gamma_i \wedge \Theta_i$

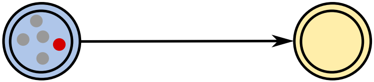


Enforcing Existence and Uniqueness on the Symbolic Heap

- Consider any location A for which invariants are violated.
- Replace constraint on i'th edge from A with constraint Δ_i enforcing memory invariants on each concrete element in A.
- These Δ_i's are of the form $\Gamma_i \wedge \Theta_i$



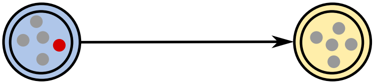
- Consider any location A for which invariants are violated.
- Replace constraint on i'th edge from A with constraint Δ_i enforcing memory invariants on each concrete element in A.
- These Δ_i's are of the form $\Gamma_i \wedge \Theta_i$



- Γ: Each concrete element $\rightarrow$ one abstract target

Enforcing Existence and Uniqueness on the Symbolic Heap

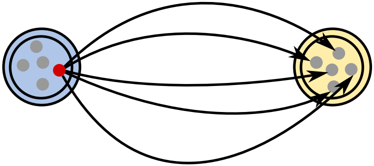
- Consider any location A for which invariants are violated.
- Replace constraint on i'th edge from A with constraint Δ_i enforcing memory invariants on each concrete element in A.
- These Δ_i's are of the form $\Gamma_i \wedge \Theta_i$



- Γ: Each concrete element $\rightarrow$ one abstract target

Enforcing Existence and Uniqueness on the Symbolic Heap

- Consider any location A for which invariants are violated.
- Replace constraint on i'th edge from A with constraint Δ_i enforcing memory invariants on each concrete element in A.
- These Δ_i's are of the form $\Gamma_i \wedge \Theta_i$



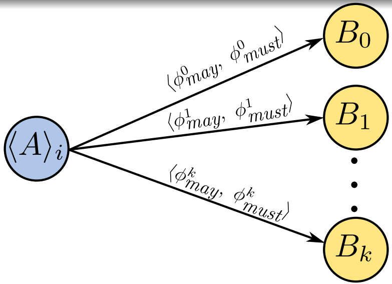
- Γ: Each concrete element $\rightarrow$ one abstract target

Enforcing Existence and Uniqueness on the Symbolic Heap

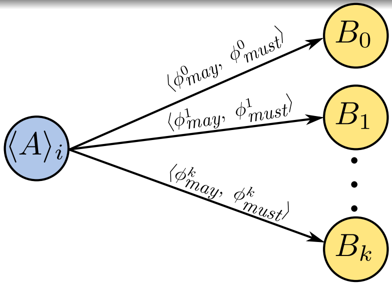
- Consider any location A for which invariants are violated.
- Replace constraint on i'th edge from A with constraint Δ_i enforcing memory invariants on each concrete element in A.
- These Δ_i's are of the form $\Gamma_i \wedge \Theta_i$

- Γ: Each concrete element $\rightarrow$ one abstract target
- Θ: In this abstract target, select one concrete element.

Constructing Γ's

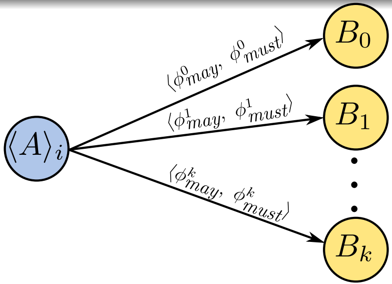


Constructing Γ's

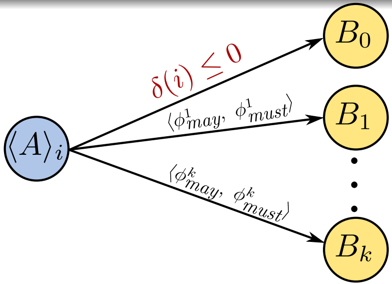


- Want to ensure i'th element of A points to exactly one B_j.

Constructing Γ's

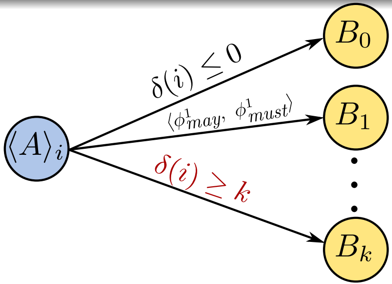


- Want to ensure i'th element of A points to exactly one B_j.
- Introduce an uninterpreted function $\delta(i)$ that selects an edge for the i'th element.



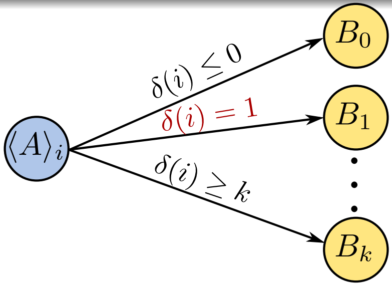
- Want to ensure i'th element of A points to exactly one B_j.
- Introduce an uninterpreted function $\delta(i)$ that selects an edge for the i'th element.

Constructing Γ's



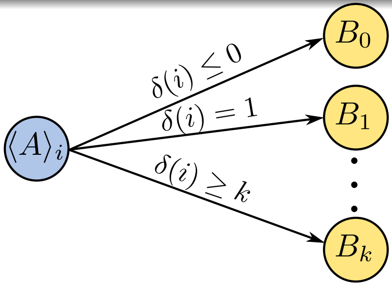
- Want to ensure i'th element of A points to exactly one B_j.
- Introduce an uninterpreted function $\delta(i)$ that selects an edge for the i'th element.

Constructing Γ's



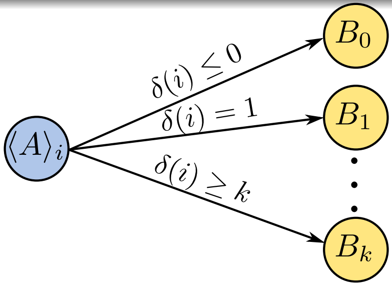
- Want to ensure i'th element of A points to exactly one B_j.
- Introduce an uninterpreted function $\delta(i)$ that selects an edge for the i'th element.

Constructing Γ's



- Want to ensure i'th element of A points to exactly one B_j.
- Introduce an uninterpreted function $\delta(i)$ that selects an edge for the i'th element.

Constructing Γ's

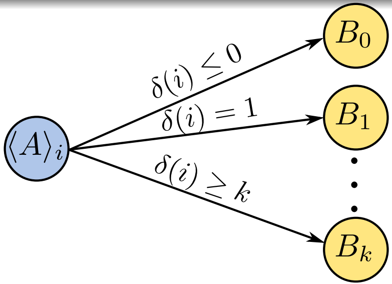


For any assignment v to i:

- $\Gamma_j(v) \wedge \Gamma_m(v)$ is UNSAT.
- $\bigvee_j \Gamma_j(v)$ is VALID.

- Want to ensure i'th element of A points to exactly one B_j.
- Introduce an uninterpreted function $\delta(i)$ that selects an edge for the i'th element.

Constructing Γ's



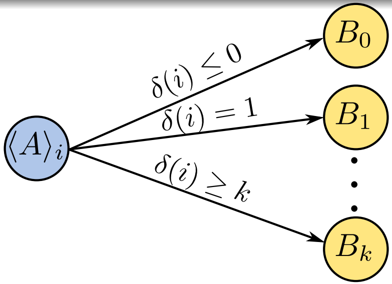
For any assignment v to i:

- $\Gamma_j(v) \wedge \Gamma_m(v)$ is UNSAT.
- $\bigvee_j \Gamma_j(v)$ is VALID.

- Want to ensure i'th element of A points to exactly one B_j.
- Introduce an uninterpreted function $\delta(i)$ that selects an edge for the i'th element.

- $\Rightarrow$ Each concrete element in A has exactly one abstract target.

Constructing Γ's



For any assignment v to i:

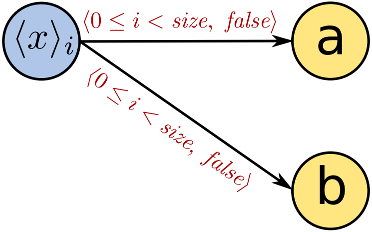
- $\Gamma_j(v) \wedge \Gamma_m(v)$ is UNSAT.
- $\bigvee_j \Gamma_j(v)$ is VALID.

- Want to ensure i'th element of A points to exactly one B_j.
- Introduce an uninterpreted function $\delta(i)$ that selects an edge for the i'th element.

- $\Rightarrow$ Each concrete element in A has exactly one abstract target.
- Correctly allows different indices to point to same target.

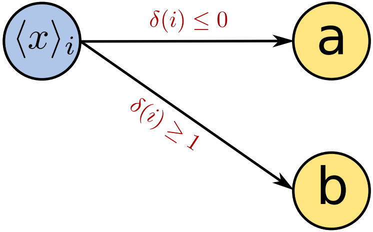
Example

```
for(int i=0; i<size; i++)
{
  if(*) x[i] = a;
  else x[i] = b;
}

y = x;
// 0 <= k < size
assert(x[k] == y[k]);
```



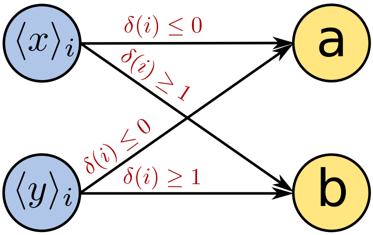
Example

```
for(int i=0; i<size; i++)
{
  if(*) x[i] = a;
  else x[i] = b;
}

y = x;
// 0 <= k < size
assert(x[k] == y[k]);
```



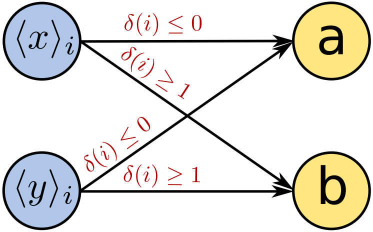
Example

```
for(int i=0; i<size; i++)
{
  if(*) x[i] = a;
  else x[i] = b;
}

y = x;
// 0 <= k < size
assert(x[k] == y[k]);
```



Example

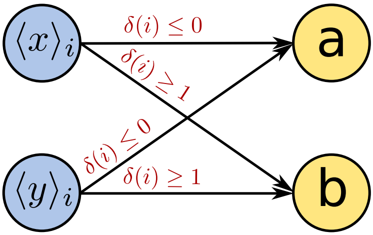
```
for(int i=0; i<size; i++)
{
  if(*) x[i] = a;
  else x[i] = b;
}

y = x;
// 0 <= k < size
assert(x[k] == y[k]);
```



- We can now prove the assertion!

Example

```
for(int i=0; i<size; i++)
{
  if(*) x[i] = a;
  else x[i] = b;
}

y = x;
// 0 <= k < size
assert(x[k] == y[k]);
```



- We can now prove the assertion!
  - Because x[k] and y[k] point to different locations under $\delta(k) \leq 0 \wedge \delta(k) \geq 1 \Rightarrow$ UNSAT

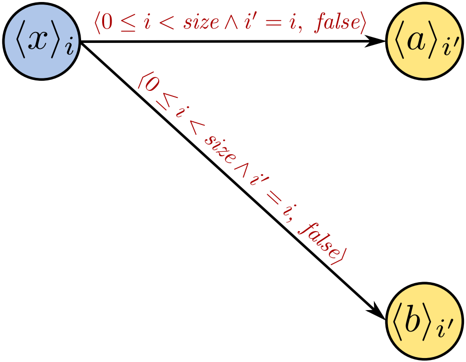
Why do we need Θ?

Why do we need Θ?

```
for(int i=0; i<size; i++)
{
  if(*) x[i] = a[i];
  else x[i] = b[i];
}

y = x;
// 0 <= k < size
assert(x[k] == y[k]);
```

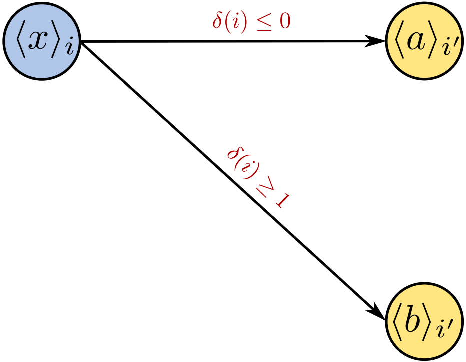
Why do we need Θ?

```
for(int i=0; i<size; i++)
{
  if(*) x[i] = a[i];
  else x[i] = b[i];
}

y = x;
// 0 <= k < size
assert(x[k] == y[k]);
```



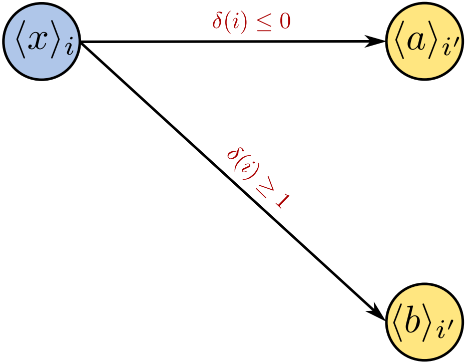
Why do we need Θ?

```
for(int i=0; i<size; i++)
{
  if(*) x[i] = a[i];
  else x[i] = b[i];
}

y = x;
// 0 <= k < size
assert(x[k] == y[k]);
```



Why do we need Θ?

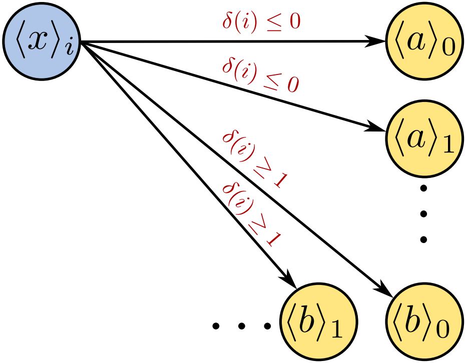
```
for(int i=0; i<size; i++)
{
  if(*) x[i] = a[i];
  else x[i] = b[i];
}

y = x;
// 0 <= k < size
assert(x[k] == y[k]);
```



- Encodes x[i] cannot point to a and b at the same time.

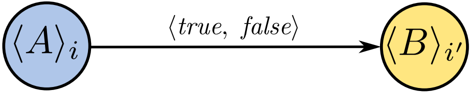
Why do we need Θ?

```
for(int i=0; i<size; i++)
{
  if(*) x[i] = a[i];
  else x[i] = b[i];
}

y = x;
// 0 <= k < size
assert(x[k] == y[k]);
```

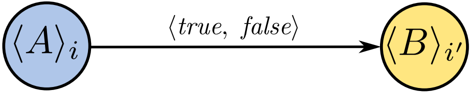


- Encodes x[i] cannot point to a and b at the same time.
- But x[i] can still point to two different elements in a

Constructing Θ

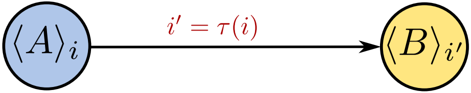


Constructing Θ



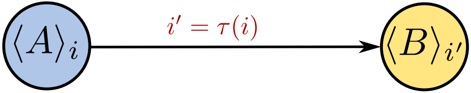
- Want the heap abstraction to encode that i'th element of A must point to exactly one element in B.

Constructing Θ



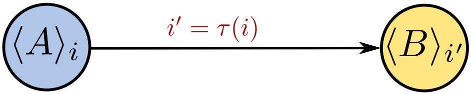
- Want the heap abstraction to encode that i'th element of A must point to exactly one element in B.

Constructing Θ



- Want the heap abstraction to encode that i'th element of A must point to exactly one element in B.
- Since τ is a function, each element in A is mapped to exactly one element in B.

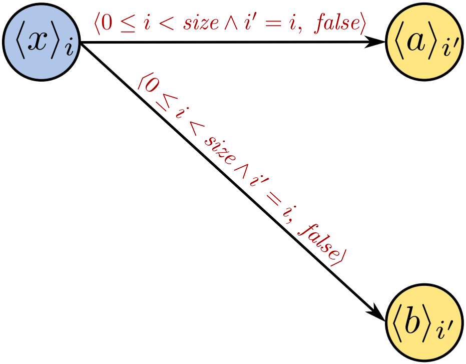
Constructing Θ



- Want the heap abstraction to encode that i'th element of A must point to exactly one element in B.
- Since τ is a function, each element in A is mapped to exactly one element in B.
- Since τ is uninterpreted, each element in A is mapped to an unknown element in B.

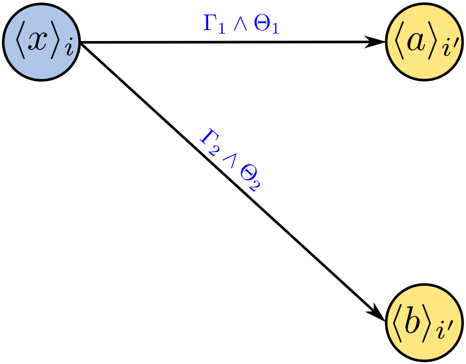
Example

```
for(int i=0; i<size; i++)
{
  if(*) x[i] = a[i];
  else x[i] = b[i];
}

y = x;
// 0 <= k < size
assert(x[k] == y[k]);
```



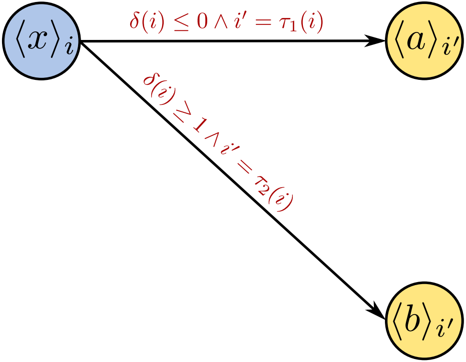
Example

```
for(int i=0; i<size; i++)
{
  if(*) x[i] = a[i];
  else x[i] = b[i];
}

y = x;
// 0 <= k < size
assert(x[k] == y[k]);
```



Example

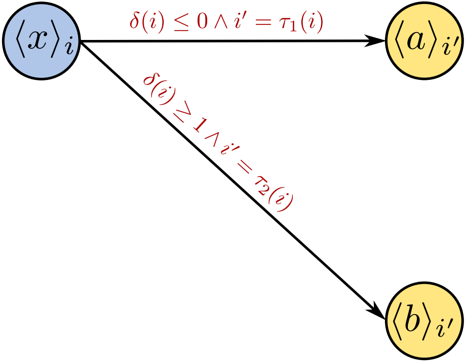
```
for(int i=0; i<size; i++)
{
  if(*) x[i] = a[i];
  else x[i] = b[i];
}

y = x;
// 0 <= k < size
assert(x[k] == y[k]);
```



- Now encodes that each element in x points to exactly one concrete element in a or b.

Example

```
for(int i=0; i<size; i++)
{
  if(*) x[i] = a[i];
  else x[i] = b[i];
}

y = x;
// 0 <= k < size
assert(x[k] == y[k]);
```



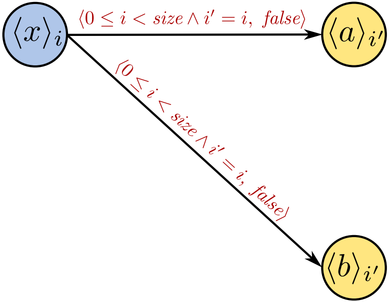
- Now encodes that each element in x points to exactly one concrete element in a or b.
- Can now prove assertion.

Preserving Existing Information

- So far, we have enforced the memory invariants; but we did not preserve all the information in the original symbolic heap.

Preserving Existing Information

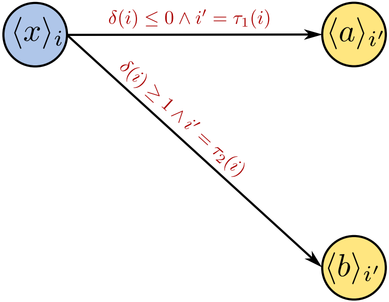
- So far, we have enforced the memory invariants; but we did not preserve all the information in the original symbolic heap.



- Using original heap, can prove `x[2]` cannot point to `a[4]`.

Preserving Existing Information

- So far, we have enforced the memory invariants; but we did not preserve all the information in the original symbolic heap.



- Using original heap, can prove `x[2]` cannot point to `a[4]`.
- But using the modified heap, we can no longer prove this.

Preserving Existing Information

Solution:

If edge in original heap is qualified by $\langle \phi_{may}, \phi_{must} \rangle$, then introduce axioms of the form:

$$\begin{array}{cccc} \forall i. & \Gamma & \Rightarrow & \phi_{may} \\ \forall i. & \phi_{must} & \Rightarrow & \Gamma \end{array}$$

Preserving Existing Information

Solution:

If edge in original heap is qualified by $\langle \phi_{may}, \phi_{must} \rangle$, then introduce axioms of the form:

$$\begin{array}{cccc} \forall i. & \Gamma & \Rightarrow & \phi_{may} \\ \forall i. & \phi_{must} & \Rightarrow & \Gamma \end{array}$$

- Can prove everthing provable under original symbolic heap

Preserving Existing Information

Solution:

If edge in original heap is qualified by $\langle \phi_{may}, \phi_{must} \rangle$, then introduce axioms of the form:

$$\begin{array}{cccc} \forall i. & \Gamma & \Rightarrow & \phi_{may} \\ \forall i. & \phi_{must} & \Rightarrow & \Gamma \end{array}$$

- Can prove everthing provable under original symbolic heap
  - And much more because we have relational reasoning

Preserving Existing Information

Solution:

If edge in original heap is qualified by $\langle \phi_{may}, \phi_{must} \rangle$, then introduce axioms of the form:

$$\begin{array}{llll} \forall i. & \Gamma & \Rightarrow & \phi_{may} \\ \forall i. & \phi_{must} & \Rightarrow & \Gamma \end{array}$$

- Can prove everthing provable under original symbolic heap
  - And much more because we have relational reasoning
- Set of provable assertions is now monotonic with respect to the precision of the original heap abstraction

Preserving Existing Information

Solution:

If edge in original heap is qualified by $\langle \phi_{may}, \phi_{must} \rangle$, then introduce axioms of the form:

$$\begin{array}{llll} \forall i. & \Gamma & \Rightarrow & \phi_{may} \\ \forall i. & \phi_{must} & \Rightarrow & \Gamma \end{array}$$

- Can prove everthing provable under original symbolic heap
  - And much more because we have relational reasoning
- Set of provable assertions is now monotonic with respect to the precision of the original heap abstraction
  - This does not hold without enforcing memory invariants!

Experiments

- We implemented this technique as part of our Compass program analysis system

Experiments

- We implemented this technique as part of our Compass program analysis system
- Verified memory safety properties (absence of buffer overruns, null dereferences, and casting errors) in a number of Unix Coreutils applications and on OpenSSH.

Results on OpenSSH

	Relational	Non-relational
Time (s)	261	788
Max memory used (MB)	208	763
# reported buffer errors	2	77
# reported null errors	3	53
# reported cast errors	0	28
Total # of errors	5	158
Total # of false positives	1	154

Results on OpenSSH

	Relational	Non-relational
Time (s)	261	788
Max memory used (MB)	208	763
# reported buffer errors	2	77
# reported null errors	3	53
# reported cast errors	0	28
Total # of errors	5	158
Total # of false positives	1	154

- Compared relational symbolic heap with basic non-relational symbolic heap for verifying memory safety in OpenSSH.

Results on OpenSSH

	Relational	Non-relational
Time (s)	261	788
Max memory used (MB)	208	763
# reported buffer errors	2	77
# reported null errors	3	53
# reported cast errors	0	28
Total # of errors	5	158
Total # of false positives	1	154

- Compared relational symbolic heap with basic non-relational symbolic heap for verifying memory safety in OpenSSH.
- Relational analysis symbolically enforces memory invariants.

Results on OpenSSH

	Relational	Non-relational
Time (s)	261	788
Max memory used (MB)	208	763
# reported buffer errors	2	77
# reported null errors	3	53
# reported cast errors	0	28
Total # of errors	5	158
Total # of false positives	1	154

- Relational technique is very precise.

Results on OpenSSH

	Relational	Non-relational
Time (s)	261	788
Max memory used (MB)	208	763
# reported buffer errors	2	77
# reported null errors	3	53
# reported cast errors	0	28
Total # of errors	5	158
Total # of false positives	1	154

- Relational technique is very precise.
- Technique without memory invariants reports many false positives.

Results on OpenSSH

	Relational	Non-relational
Time (s)	261	788
Max memory used (MB)	208	763
# reported buffer errors	2	77
# reported null errors	3	53
# reported cast errors	0	28
Total # of errors	5	158
Total # of false positives	1	154

- Relational technique is very precise.
- Technique without memory invariants reports many false positives.

Results on OpenSSH

	Relational	Non-relational
Time (s)	261	788
Max memory used (MB)	208	763
# reported buffer errors	2	77
# reported null errors	3	53
# reported cast errors	0	28
Total # of errors	5	158
Total # of false positives	1	154

- Relational technique is very precise.
- Technique without memory invariants reports many false positives.
- Surprisingly, more precise is also more efficient.

Results on OpenSSH

	Relational	Non-relational
Time (s)	261	788
Max memory used (MB)	208	763
# reported buffer errors	2	77
# reported null errors	3	53
# reported cast errors	0	28
Total # of errors	5	158
Total # of false positives	1	154

- Relational technique is very precise.
- Technique without memory invariants reports many false positives.
- Surprisingly, more precise is also more efficient.
    - Memory invariant alone is sufficient to discharge many facts.

Thank You!

Dillig, I., Dillig, T., Aiken, A.:
Fluid updates: Beyond strong vs. weak updates.
In: ESOP (2010) 246–266

Reps, T.W., Sagiv, S., Wilhelm, R.:
Static program analysis via 3-valued logic.
In: CAV (2004) 15–30

Gopan, D., Reps, T., Sagiv, M.:
A framework for numeric analysis of array operations.
In: POPL (2005) 338–350

Bogudlov, I., Lev-Ami, T., Reps, T., Sagiv, M.:
Revamping TVLA: Making parametric shape analysis competitive.
Lecture Notes in Computer Science **4590** (2007) 221

Manevich, R.:
Partially Disjunctive Shape Analysis.
PhD thesis, Tel Aviv University (2009)