Symbolic Heap Abstraction with Demand-Driven Axiomatization of Memory Invariants

Isil Dillig  Thomas Dillig  Alex Aiken

Stanford University
Goal of heap analysis: Statistically describe all possible points-to relations in the heap for any execution of the program.
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.
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.
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.
Consider the code snippet:

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

y = x;
assert(*x == *y);
```
Consider the code snippet:

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

y = x;
assert(*x == *y);
```
Consider the code snippet:

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

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

- Non-relational:

  Does not encode $x$ and $y$ must point to same location
Consider the code snippet:

```c
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
Consider the code snippet:

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

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

Perform case split on possible heaps.
Consider the code snippet:

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

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

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

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

- Each abstract location points to exactly one target location per heap
- \( \Rightarrow \) precise relational reasoning
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
- \( \Rightarrow \) very expensive and unscalable
Advantages:
- Each abstract location points to exactly one target location per heap
- \( \Rightarrow \) precise relational reasoning

Disadvantages:
- Generates exponential number of heaps
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
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
- \( \Rightarrow \) Very expensive and unscalable
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
- **Uniqueness**: Every memory location has at most one value

⇒ Heap splitting is one way of enforcing these 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.
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
To encode that \( x \) cannot point to \( a \) and \( b \) at the same time, we can use two constraints \( \phi \) and \( \neg\phi \).
To encode that \( x \) cannot point to \( a \) and \( b \) at the same time, we can use two constraints \( \phi \) and \( \neg \phi \).
To encode that $x$ cannot point to $a$ and $b$ at the same time, we can use two constraints $\phi$ and $\neg \phi \Rightarrow \text{Uniqueness}$.
To encode that \( x \) cannot point to \( a \) and \( b \) at the same time, we can use two constraints \( \phi \) and \( \neg \phi \Rightarrow \text{Uniqueness} \). Also encodes that \( x \) must point to either \( a \) or \( b \).
To encode that $x$ cannot point to $a$ and $b$ at the same time, we can use two constraints $\phi$ and $\neg \phi \Rightarrow$ Uniqueness.

Also encodes that $x$ must point to either $a$ or $b \Rightarrow$ Existence.
if(*)
    *x = a;
else
    *x = b;

y = x;
assert(*x == *y);
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 $\phi \land \neg \phi$
  $\Rightarrow$ Can prove the assertion!
Easy to enforce these invariants when each abstract location corresponds to one concrete location.
Easy to enforce these invariants when each abstract location corresponds to one concrete location.

But what about abstract locations that represent multiple concrete 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]);
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 \textit{summary node}. 
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 $b$. 
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]);
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
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!
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.
Use the **symbolic heap** from our previous work that allows distinguishing individual elements in a summary location.
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
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.
Abstract locations that represent more than one concrete location are qualified by **index variables**.
Abstract locations that represent more than one concrete location are qualified by *index variables*. 

```java
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.

```java
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]);
```
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.
```cpp
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.
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
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.
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.

```c
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]);
```
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 $\Delta$ enforcing these invariants.

2. Preserves all the partial information encoded in the original symbolic heap.
   - Restore existing information by adding quantified axioms relating $\Delta$ to the original 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

Isil Dillig  Thomas Dillig  Alex Aiken
Symbolic Heap Abstraction with Demand-Driven Axiomatization
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 $\Delta$ enforcing these invariants.
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 $\Delta$ enforcing these invariants.

2. Preserves all the partial information encoded in the original symbolic heap
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 $\Delta$ enforcing these invariants.

2. Preserves all the partial information encoded in the original symbolic heap
   - Restore existing information by adding quantified axioms relating $\Delta$ to the original constraints
Consider any location $A$ for which invariants are violated.
Consider any location $A$ for which invariants are violated.

Replace constraint on $i$'th edge from $A$ with constraint $\Delta_i$ enforcing memory invariants on each concrete element in $A$. 

$\Delta_i$: Each concrete element $\rightarrow$ one abstract target $\Theta_i$: In this abstract target, select one concrete element.
Consider any location $A$ for which invariants are violated.

Replace constraint on $i$'th edge from $A$ with constraint $\Delta_i$ enforcing memory invariants on each concrete element in $A$.

These $\Delta_i$'s are of the form $\Gamma_i \land \Theta_i$
Consider any location $A$ for which invariants are violated.

Replace constraint on $i$’th edge from $A$ with constraint $\Delta_i$ enforcing memory invariants on each concrete element in $A$.

These $\Delta_i$’s are of the form $\Gamma_i \land \Theta_i$.
Consider any location $A$ for which invariants are violated.

Replace constraint on $i$'th edge from $A$ with constraint $\Delta_i$ enforcing memory invariants on each concrete element in $A$.

These $\Delta_i$'s are of the form $\Gamma_i \land \Theta_i$. 
Consider any location $A$ for which invariants are violated.

Replace constraint on $i$’th edge from $A$ with constraint $\Delta_i$ enforcing memory invariants on each concrete element in $A$.

These $\Delta_i$’s are of the form $\Gamma_i \land \Theta_i$. 

![Diagram showing a central location connected to multiple abstract targets](image)
Consider any location $A$ for which invariants are violated.

Replace constraint on $i$’th edge from $A$ with constraint $\Delta_i$ enforcing memory invariants on each concrete element in $A$.

These $\Delta_i$’s are of the form $\Gamma_i \land \Theta_i$.
Consider any location $A$ for which invariants are violated.

Replace constraint on $i$’th edge from $A$ with constraint $\Delta_i$ enforcing memory invariants on each concrete element in $A$.

These $\Delta_i$’s are of the form $\Gamma_i \land \Theta_i$

- $\Gamma$: Each concrete element $\rightarrow$ one abstract target
Consider any location $A$ for which invariants are violated.

Replace constraint on $i$’th edge from $A$ with constraint $\Delta_i$ enforcing memory invariants on each concrete element in $A$.

These $\Delta_i$’s are of the form $\Gamma_i \land \Theta_i$

$\Gamma$: Each concrete element $\rightarrow$ one abstract target
Consider any location \( A \) for which invariants are violated.

Replace constraint on \( i \)'th edge from \( A \) with constraint \( \Delta_i \) enforcing memory invariants on each concrete element in \( A \).

These \( \Delta_i \)'s are of the form \( \Gamma_i \land \Theta_i \)

\[ \Gamma: \text{Each concrete element} \rightarrow \text{one abstract target} \]
Consider any location $A$ for which invariants are violated.

Replace constraint on $i$’th edge from $A$ with constraint $\Delta_i$ enforcing memory invariants on each concrete element in $A$.

These $\Delta_i$’s are of the form $\Gamma_i \land \Theta_i$

- $\Gamma$: Each concrete element $\rightarrow$ one abstract target
- $\Theta$: In this abstract target, select one concrete element.
For any assignment $v$ to $i$:

- $\Gamma_j(v) \land \Gamma_m(v)$ 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 the same target.
Want to ensure \( i \)'th element of \( A \) points to exactly one \( B_j \).
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.
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

For any assignment \( v \) to \( i \):

\[ \Gamma_j(v) \land \Gamma_m(v) \text{ is UNSAT.} \]

\[ \bigvee_j \Gamma_j(v) \text{ is VALID.} \]

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.
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.

For any assignment $v$ to $i$:
- $\Gamma_j(v) \land \Gamma_m(v)$ is UNSAT.
- $\lor_j \Gamma_j(v)$ is VALID.
For any assignment $v$ to $i$:
- $\Gamma_j(v) \land \Gamma_m(v)$ 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.
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.
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]);
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]);
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]);
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!
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
  \( \delta(k) \leq 0 \land \delta(k) \geq 1 \Rightarrow \text{UNSAT} \)
Why do we need $\Theta$?
Why do we need $\Theta$?

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

y = x;
// 0 \leq k < size
assert(x[k] == y[k]);
Why do we need $\Theta$?

```java
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 $\Theta$?

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
Want the heap abstraction to encode that \( i \)’th element of \( A \) must point to exactly one element in \( B \).

Since \( \tau \) is a function, each element in \( A \) is mapped to exactly one element in \( B \).

Since \( \tau \) is uninterpreted, each element in \( A \) is mapped to an unknown element in \( B \).
Want the heap abstraction to encode that $i$'th element of $A$ must point to exactly one element in $B$. 
Want the heap abstraction to encode that $i'$th element of $A$ must point to exactly one element in $B$. 

$\langle A \rangle_i \xrightarrow{i' = \tau(i)} \langle B \rangle_{i'}$
Want the heap abstraction to encode that $i$’th element of $A$ must point to exactly one element in $B$.

Since $\tau$ is a function, each element in $A$ is mapped to exactly one element in $B$. 
Want the heap abstraction to encode that \( i \)'th element of \( A \) must point to exactly one element in \( B \).

Since \( \tau \) is a function, each element in \( A \) is mapped to exactly one element in \( B \).

Since \( \tau \) is uninterpreted, each element in \( A \) is mapped to an unknown element in \( B \).
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]);
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]);
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.
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.
So far, we have enforced the memory invariants; but we did not preserve all the information in the original symbolic heap.
So far, we have enforced the memory invariants; but we did not preserve all the information in the original symbolic heap.

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

- But using the modified heap, we can no longer prove this.
Preserving Existing Information

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

\[
\forall i. \quad \Gamma \Rightarrow \phi_{\text{may}} \\
\forall i. \quad \phi_{\text{must}} \Rightarrow \Gamma
\]

Can prove everything provable under original symbolic heap abstraction and much more because we have relational reasoning. This does not hold without enforcing memory invariants!

Isil Dillig  Thomas Dillig  Alex Aiken

Symbolic Heap Abstraction with Demand-Driven Axiomatization
Solution:

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

\[
\forall i. \quad \Gamma \Rightarrow \phi_{\text{may}} \\
\forall i. \quad \phi_{\text{must}} \Rightarrow \Gamma
\]

- Can prove everything provable under original symbolic heap.
Preserving Existing Information

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

$$\forall i. \Gamma \Rightarrow \phi_{\text{may}}$$
$$\forall i. \phi_{\text{must}} \Rightarrow \Gamma$$

• 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:

$$\forall i. \; \Gamma \Rightarrow \phi_{may}$$
$$\forall i. \; \phi_{must} \Rightarrow \Gamma$$

- Can prove **everything** 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 \( \langle \phi_{\text{may}}, \phi_{\text{must}} \rangle \), then introduce axioms of the form:

\[
\forall i. \quad \Gamma \Rightarrow \phi_{\text{may}} \\
\forall i. \quad \phi_{\text{must}} \Rightarrow \Gamma
\]

- Can prove \textbf{everything} provable under \textbf{original symbolic heap}
  - And \textbf{much more} because we have \textbf{relational reasoning}

- Set of provable assertions is now \textbf{monotonic} with respect to the precision of the original heap abstraction
  - This does not hold without enforcing memory invariants!
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

<table>
<thead>
<tr>
<th></th>
<th>Relational</th>
<th>Non-relational</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (s)</td>
<td>261</td>
<td>788</td>
</tr>
<tr>
<td>Max memory used (MB)</td>
<td>208</td>
<td>763</td>
</tr>
<tr>
<td># reported buffer errors</td>
<td>2</td>
<td>77</td>
</tr>
<tr>
<td># reported null errors</td>
<td>3</td>
<td>53</td>
</tr>
<tr>
<td># reported cast errors</td>
<td>0</td>
<td>28</td>
</tr>
<tr>
<td>Total # of errors</td>
<td>5</td>
<td>158</td>
</tr>
<tr>
<td>Total # of false positives</td>
<td>1</td>
<td>154</td>
</tr>
</tbody>
</table>
Compared relational symbolic heap with basic non-relational symbolic heap for verifying memory safety in OpenSSH.

<table>
<thead>
<tr>
<th></th>
<th>Relational</th>
<th>Non-relational</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (s)</td>
<td>261</td>
<td>788</td>
</tr>
<tr>
<td>Max memory used (MB)</td>
<td>208</td>
<td>763</td>
</tr>
<tr>
<td># reported buffer errors</td>
<td>2</td>
<td>77</td>
</tr>
<tr>
<td># reported null errors</td>
<td>3</td>
<td>53</td>
</tr>
<tr>
<td># reported cast errors</td>
<td>0</td>
<td>28</td>
</tr>
<tr>
<td>Total # of errors</td>
<td>5</td>
<td>158</td>
</tr>
<tr>
<td>Total # of false positives</td>
<td>1</td>
<td>154</td>
</tr>
</tbody>
</table>
Results on OpenSSH

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

- 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

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

- Relational technique is very **precise**.
## Results on OpenSSH

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

- Relational technique is very precise.
- Technique **without** memory invariants reports many false positives.
Results on OpenSSH

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

- **Relational technique is very precise.**
- **Technique without memory invariants reports many false positives.**
Results on OpenSSH

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

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

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

- 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.

Isil Dillig  Thomas Dillig  Alex Aiken

Symbolic Heap Abstraction with Demand-Driven Axiomatization
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.  

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)