Precise Reasoning for Programs Using Containers

Işıl Dillig Thomas Dillig Alex Aiken
Stanford University

Containers

Containers

General-purpose data structures for inserting, retrieving, removing, and iterating over elements

Containers

General-purpose data structures for inserting, retrieving, removing, and iterating over elements

- Examples: Array, vector, list, map, set, stack, queue, ...

Containers

Containers

General-purpose data structures for inserting, retrieving, removing, and iterating over elements

- Examples: Array, vector, list, map, set, stack, queue, . . .
- Widely used; provided by common programming languages or standard libraries

Containers

Containers

General-purpose data structures for inserting, retrieving, removing, and iterating over elements

- Examples: Array, vector, list, map, set, stack, queue, ...
- Widely used; provided by common programming languages or standard libraries

⇒ Associate arrays in scripting languages, data structures provided by C++ STL, etc.

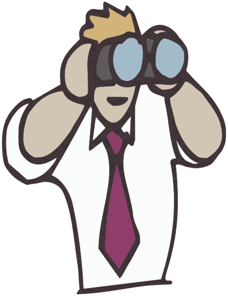
Containers

Precise static reasoning about containers crucial for successful verification

Observation #1

- Many different kinds of containers, varying in the convenience or efficiency of certain operations

Observation #1



- Many different kinds of containers, varying in the convenience or efficiency of certain operations
- But functionally, there are only two kinds.

Classification of Containers

Sequences (Arrays / Linked Lists) - ordered collections	
vector	a dynamic array, like C array (i.e., capable of random access) with the ability to resize itself automatically when inserting or erasing an object. Inserting and removing an element to/from back of the vector at the end takes amortized constant time. Inserting and erasing at the beginning or in the middle is linear in time. A specialization for type bool exists, which optimizes for space by storing bool values as bits.
list	a doubly-linked list; elements are not stored in contiguous memory. Opposite performance from a vector. Slow lookup and access (linear time), but once a position has been found, quick insertion and deletion (constant time).
deque (*double ended queue*)	a vector with insertion/erase at the beginning or end in amortized constant time, however lacking some guarantees on iterator validity after altering the deque.
Container adaptors	
queue	Provides FIFO queue interface in terms of `push`/`pop`/`front`/`back` operations. Any sequence supporting operations `front()`, `back()`, `push_back()`, and `pop_front()` can be used to instantiate queue (e.g. *list* and *deque*).
priority_queue	Provides priority queue interface in terms of `push`/`pop`/`top` operations (the element with the highest priority is on top). Any random-access sequence supporting operations `front()`, `push_back()`, and `pop_back()` can be used to instantiate priority_queue (e.g. *vector* and *deque*). Elements should additionally support comparison (to determine which element has a higher priority and should be popped first).
stack	Provides LIFO stack interface in terms of `push`/`pop`/`top` operations (the last-inserted element is on top). Any sequence supporting operations `back()`, `push_back()`, and `pop_back()` can be used to instantiate stack (e.g. *vector*, *list*, and *deque*).
Associative containers - unordered collections	
set	a mathematical set; inserting/erasing elements in a set does not invalidate iterators pointing in the set. Provides set operations union, intersection, difference, symmetric difference and test of inclusion. Type of data must implement comparison operator < or custom comparator function must be specified; such comparison operator or comparator function must guarantee strict weak ordering, otherwise behavior is undefined. Typically implemented using a self-balancing binary search tree.
multiset	same as a set, but allows duplicate elements.
map	an associative array; allows mapping from one data item (a key) to another (a value). Type of key must implement comparison operator < or custom comparator function must be specified; such comparison operator or comparator function must guarantee strict weak ordering, otherwise behavior is undefined. Typically implemented using a self-balancing binary search tree.
multimap	same as a map, but allows duplicate keys.
hash_set **hash_multiset** **hash_map** **hash_multimap**	similar to a set, multiset, map, or multimap, respectively, but implemented using a hash table; keys are not ordered, but a hash function must exist for the key type. These containers are not part of the C++ Standard Library, but are included in SGI's STL extensions, and are included in common libraries such as the GNU C++ Library in the `__gnu_cxx` namespace. These are scheduled to be added to the C++ standard as part of TR1, with the slightly different names of unordered_set, unordered_multiset, unordered_map and unordered_multimap.

⋮

1. Position-dependent Containers

Classification of Containers

Sequences (Arrays / Linked Lists) - ordered collections	
vector	a dynamic array, like C array (i.e., capable of random access) with the ability to resize itself automatically when inserting or erasing an object. Inserting and removing an element to/from back of the vector at the end takes amortized constant time. Inserting and erasing at the beginning or in the middle is linear in time. A specialization for type bool exists, which optimizes for space by storing bool values as bits.
list	a doubly-linked list; elements are not stored in contiguous memory. Opposite performance from a vector. Slow lookup and access (linear time), but once a position has been found, quick insertion and deletion (constant time).
deque (*double ended queue*)	a vector with insertion/erase at the beginning or end in amortized constant time, however lacking some guarantees on iterator validity after altering the deque.
Container adaptors	
queue	Provides FIFO queue interface in terms of `push/pop/front/back` operations. Any sequence supporting operations `front()`, `back()`, `push_back()`, and `pop_front()` can be used to instantiate queue (e.g. *list* and *deque*).
priority_queue	Provides priority queue interface in terms of `push/pop/top` operations (the element with the highest priority is on top). Any random-access sequence supporting operations `front()`, `push_back()`, and `pop_back()` can be used to instantiate priority_queue (e.g. *vector* and *deque*). Elements should additionally support comparison (to determine which element has a higher priority and should be popped first).
stack	Provides LIFO stack interface in terms of `push/pop/top` operations (the last-inserted element is on top). Any sequence supporting operations `back()`, `push_back()`, and `pop_back()` can be used to instantiate stack (e.g. *vector*, *list*, and *deque*).
Associative containers - unordered collections	
set	a mathematical set; inserting/erasing elements in a set does not invalidate iterators pointing in the set. Provides set operations union, intersection, difference, symmetric difference and test of inclusion. Type of data must implement comparison operator < or custom comparator function must be specified; such comparison operator or comparator function must guarantee strict weak ordering, otherwise behavior is undefined. Typically implemented using a self-balancing binary search tree.
multiset	same as a set, but allows duplicate elements.
map	an associative array; allows mapping from one data item (a key) to another (a value). Type of key must implement comparison operator < or custom comparator function must be specified; such comparison operator or comparator function must guarantee strict weak ordering, otherwise behavior is undefined. Typically implemented using a self-balancing binary search tree.
multimap	same as a map, but allows duplicate keys.
hash_set **hash_multiset** **hash_map** **hash_multimap**	similar to a set, multiset, map, or multimap, respectively, but implemented using a hash table; keys are not ordered, but a hash function must exist for the key type. These containers are not part of the C++ Standard Library, but are included in SGI's STL extensions, and are included in common libraries such as the GNU C++ Library in the `__gnu_cxx` namespace. These are scheduled to be added to the C++ standard as part of TR1, with the slightly different names of unordered_set, unordered_multiset, unordered_map and unordered_multimap.

⋮

1. Position-dependent Containers
   - Well-defined meaning of position

Classification of Containers

Sequences (Arrays / Linked Lists) - ordered collections	
vector	a dynamic array, like C array (i.e., capable of random access) with the ability to resize itself automatically when inserting or erasing an object. Inserting and removing an element to/from back of the vector at the end takes amortized constant time. Inserting and erasing at the beginning or in the middle is linear in time. A specialization for type bool exists, which optimizes for space by storing bool values as bits.
list	a doubly-linked list; elements are not stored in contiguous memory. Opposite performance from a vector. Slow lookup and access (linear time), but once a position has been found, quick insertion and deletion (constant time).
deque (*double ended queue*)	a vector with insertion/erase at the beginning or end in amortized constant time, however lacking some guarantees on iterator validity after altering the deque.
Container adaptors	
queue	Provides FIFO queue interface in terms of `push/pop/front/back` operations. Any sequence supporting operations `front()`, `back()`, `push_back()`, and `pop_front()` can be used to instantiate queue (e.g. *list* and *deque*).
priority_queue	Provides priority queue interface in terms of `push/pop/top` operations (the element with the highest priority is on top). Any random-access sequence supporting operations `front()`, `push_back()`, and `pop_back()` can be used to instantiate priority_queue (e.g. *vector* and *deque*). Elements should additionally support comparison (to determine which element has a higher priority and should be popped first).
stack	Provides LIFO stack interface in terms of `push/pop/top` operations (the last-inserted element is on top). Any sequence supporting operations `back()`, `push_back()`, and `pop_back()` can be used to instantiate stack (e.g. *vector*, *list*, and *deque*).
Associative containers - unordered collections	
set	a mathematical set; inserting/erasing elements in a set does not invalidate iterators pointing in the set. Provides set operations union, intersection, difference, symmetric difference and test of inclusion. Type of data must implement comparison operator < or custom comparator function must be specified; such comparison operator or comparator function must guarantee strict weak ordering, otherwise behavior is undefined. Typically implemented using a self-balancing binary search tree.
multiset	same as a set, but allows duplicate elements.
map	an associative array; allows mapping from one data item (a key) to another (a value). Type of key must implement comparison operator < or custom comparator function must be specified; such comparison operator or comparator function must guarantee strict weak ordering, otherwise behavior is undefined. Typically implemented using a self-balancing binary search tree.
multimap	same as a map, but allows duplicate keys.
hash_set **hash_multiset** **hash_map** **hash_multimap**	similar to a set, multiset, map, or multimap, respectively, but implemented using a hash table; keys are not ordered, but a hash function must exist for the key type. These containers are not part of the C++ Standard Library, but are included in SGI's STL extensions, and are included in common libraries such as the GNU C++ Library in the `__gnu_cxx` namespace. These are scheduled to be added to the C++ standard as part of TR1, with the slightly different names of unordered_set, unordered_multiset, unordered_map and unordered_multimap.

⋮

1. Position-dependent Containers
   - Well-defined meaning of position
   - Iteration in a pre-defined order

Classification of Containers

Sequences (Arrays / Linked Lists) - ordered collections	
vector	a dynamic array, like C array (i.e., capable of random access) with the ability to resize itself automatically when inserting or erasing an object. Inserting and removing an element to/from back of the vector at the end takes amortized constant time. Inserting and erasing at the beginning or in the middle is linear in time. A specialization for type bool exists, which optimizes for space by storing bool values as bits.
list	a doubly-linked list; elements are not stored in contiguous memory. Opposite performance from a vector. Slow lookup and access (linear time), but once a position has been found, quick insertion and deletion (constant time).
deque (*double ended queue*)	a vector with insertion/erase at the beginning or end in amortized constant time, however lacking some guarantees on iterator validity after altering the deque.
Container adaptors	
queue	Provides FIFO queue interface in terms of `push/pop/front/back` operations. Any sequence supporting operations `front()`, `back()`, `push_back()`, and `pop_front()` can be used to instantiate queue (e.g. *list* and *deque*).
priority_queue	Provides priority queue interface in terms of `push/pop/top` operations (the element with the highest priority is on top). Any random-access sequence supporting operations `front()`, `push_back()`, and `pop_back()` can be used to instantiate priority_queue (e.g. *vector* and *deque*). Elements should additionally support comparison (to determine which element has a higher priority and should be popped first).
stack	Provides LIFO stack interface in terms of `push/pop/top` operations (the last-inserted element is on top). Any sequence supporting operations `back()`, `push_back()`, and `pop_back()` can be used to instantiate stack (e.g. *vector*, *list*, and *deque*).
Associative containers - unordered collections	
set	a mathematical set; inserting/erasing elements in a set does not invalidate iterators pointing in the set. Provides set operations union, intersection, difference, symmetric difference and test of inclusion. Type of data must implement comparison operator < or custom comparator function must be specified; such comparison operator or comparator function must guarantee strict weak ordering, otherwise behavior is undefined. Typically implemented using a self-balancing binary search tree.
multiset	same as a set, but allows duplicate elements.
map	an associative array; allows mapping from one data item (a key) to another (a value). Type of key must implement comparison operator < or custom comparator function must be specified; such comparison operator or comparator function must guarantee strict weak ordering, otherwise behavior is undefined. Typically implemented using a self-balancing binary search tree.
multimap	same as a map, but allows duplicate keys.
hash_set hash_multiset hash_map hash_multimap	similar to a set, multiset, map, or multimap, respectively, but implemented using a hash table; keys are not ordered, but a hash function must exist for the key type. These containers are not part of the C++ Standard Library, but are included in SGI's STL extensions, and are included in common libraries such as the GNU C++ Library in the `__gnu_cxx` namespace. These are scheduled to be added to the C++ standard as part of TR1, with the slightly different names of unordered_set, unordered_multiset, unordered_map and unordered_multimap.

⋮

1. Position-dependent Containers
   - Well-defined meaning of position
   - Iteration in a pre-defined order
2. Value-dependent Containers

Classification of Containers

Sequences (Arrays / Linked Lists) - ordered collections	
vector	a dynamic array, like C array (i.e., capable of random access) with the ability to resize itself automatically when inserting or erasing an object. Inserting and removing an element to/from back of the vector at the end takes amortized constant time. Inserting and erasing at the beginning or in the middle is linear in time. A specialization for type bool exists, which optimizes for space by storing bool values as bits.
list	a doubly-linked list; elements are not stored in contiguous memory. Opposite performance from a vector. Slow lookup and access (linear time), but once a position has been found, quick insertion and deletion (constant time).
deque (*double ended queue*)	a vector with insertion/erase at the beginning or end in amortized constant time, however lacking some guarantees on iterator validity after altering the deque.
Container adaptors	
queue	Provides FIFO queue interface in terms of `push`/`pop`/`front`/`back` operations. Any sequence supporting operations `front()`, `back()`, `push_back()`, and `pop_front()` can be used to instantiate queue (e.g. *list* and *deque*).
priority_queue	Provides priority queue interface in terms of `push`/`pop`/`top` operations (the element with the highest priority is on top). Any random-access sequence supporting operations `front()`, `push_back()`, and `pop_back()` can be used to instantiate priority_queue (e.g. *vector* and *deque*). Elements should additionally support comparison (to determine which element has a higher priority and should be popped first).
stack	Provides LIFO stack interface in terms of `push`/`pop`/`top` operations (the last-inserted element is on top). Any sequence supporting operations `back()`, `push_back()`, and `pop_back()` can be used to instantiate stack (e.g. *vector*, *list*, and *deque*).
Associative containers - unordered collections	
set	a mathematical set; inserting/erasing elements in a set does not invalidate iterators pointing in the set. Provides set operations union, intersection, difference, symmetric difference and test of inclusion. Type of data must implement comparison operator < or custom comparator function must be specified; such comparison operator or comparator function must guarantee strict weak ordering, otherwise behavior is undefined. Typically implemented using a self-balancing binary search tree.
multiset	same as a set, but allows duplicate elements.
map	an associative array; allows mapping from one data item (a key) to another (a value). Type of key must implement comparison operator < or custom comparator function must be specified; such comparison operator or comparator function must guarantee strict weak ordering, otherwise behavior is undefined. Typically implemented using a self-balancing binary search tree.
multimap	same as a map, but allows duplicate keys.
hash_set **hash_multiset** **hash_map** **hash_multimap**	similar to a set, multiset, map, or multimap, respectively, but implemented using a hash table; keys are not ordered, but a hash function must exist for the key type. These containers are not part of the C++ Standard Library, but are included in SGI's STL extensions, and are included in common libraries such as the GNU C++ Library in the `__gnu_cxx` namespace. These are scheduled to be added to the C++ standard as part of TR1, with the slightly different names of unordered_set, unordered_multiset, unordered_map and unordered_multimap.

⋮

1. Position-dependent Containers
   - Well-defined meaning of position
   - Iteration in a pre-defined order
2. Value-dependent Containers
   - Keys of arbitrary type

Classification of Containers

Sequences (Arrays / Linked Lists) - ordered collections	
vector	a dynamic array, like C array (i.e., capable of random access) with the ability to resize itself automatically when inserting or erasing an object. Inserting and removing an element to/from back of the vector at the end takes amortized constant time. Inserting and erasing at the beginning or in the middle is linear in time. A specialization for type bool exists, which optimizes for space by storing bool values as bits.
list	a doubly-linked list; elements are not stored in contiguous memory. Opposite performance from a vector. Slow lookup and access (linear time), but once a position has been found, quick insertion and deletion (constant time).
deque (*double ended queue*)	a vector with insertion/erase at the beginning or end in amortized constant time, however lacking some guarantees on iterator validity after altering the deque.
Container adaptors	
queue	Provides FIFO queue interface in terms of `push/pop/front/back` operations. Any sequence supporting operations `front()`, `back()`, `push_back()`, and `pop_front()` can be used to instantiate queue (e.g. *list* and *deque*).
priority_queue	Provides priority queue interface in terms of `push/pop/top` operations (the element with the highest priority is on top). Any random-access sequence supporting operations `front()`, `push_back()`, and `pop_back()` can be used to instantiate priority_queue (e.g. *vector* and *deque*). Elements should additionally support comparison (to determine which element has a higher priority and should be popped first).
stack	Provides LIFO stack interface in terms of `push/pop/top` operations (the last-inserted element is on top). Any sequence supporting operations `back()`, `push_back()`, and `pop_back()` can be used to instantiate stack (e.g. *vector*, *list*, and *deque*).
Associative containers - unordered collections	
set	a mathematical set; inserting/erasing elements in a set does not invalidate iterators pointing in the set. Provides set operations union, intersection, difference, symmetric difference and test of inclusion. Type of data must implement comparison operator < or custom comparator function must be specified; such comparison operator or comparator function must guarantee strict weak ordering, otherwise behavior is undefined. Typically implemented using a self-balancing binary search tree.
multiset	same as a set, but allows duplicate elements.
map	an associative array; allows mapping from one data item (a key) to another (a value). Type of key must implement comparison operator < or custom comparator function must be specified; such comparison operator or comparator function must guarantee strict weak ordering, otherwise behavior is undefined. Typically implemented using a self-balancing binary search tree.
multimap	same as a map, but allows duplicate keys.
hash_set **hash_multiset** **hash_map** **hash_multimap**	similar to a set, multiset, map, or multimap, respectively, but implemented using a hash table; keys are not ordered, but a hash function must exist for the key type. These containers are not part of the C++ Standard Library, but are included in SGI's STL extensions, and are included in common libraries such as the GNU C++ Library in the `__gnu_cxx` namespace. These are scheduled to be added to the C++ standard as part of TR1, with the slightly different names of unordered_set, unordered_multiset, unordered_map and unordered_multimap.

⋮

1. Position-dependent Containers
   - Well-defined meaning of position
   - Iteration in a pre-defined order
2. Value-dependent Containers
   - Keys of arbitrary type
   - Iteration order may be undefined

Observation #2:

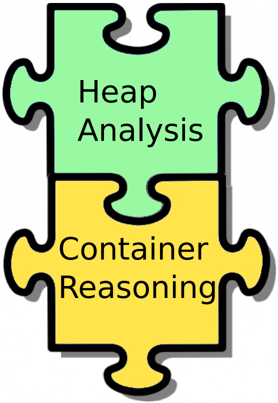
- Orders of magnitude more clients of containers than there are container implementations

Observation #2:

- Orders of magnitude more clients of containers than there are container implementations

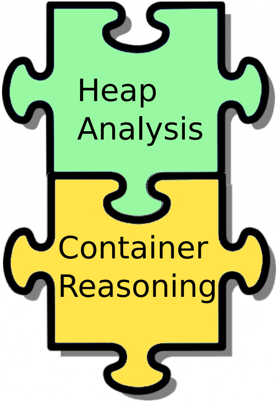
⇒ Need fully automatic, scalable techniques for reasoning about client-side use of container data structures

This Talk



Precise, fully-automatic technique that integrates container reasoning into heap analysis

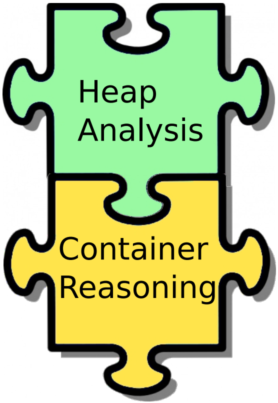
This Talk



Precise, fully-automatic technique that integrates container reasoning into heap analysis

1. tracks key-value correlations

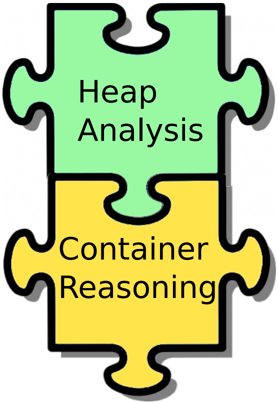
This Talk



Precise, fully-automatic technique that integrates container reasoning into heap analysis

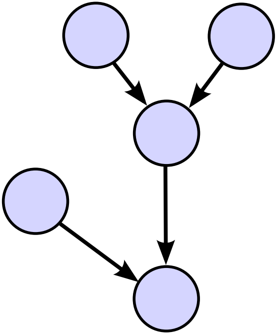
1. tracks key-value correlations
2. can model nested containers in a precise way

This Talk



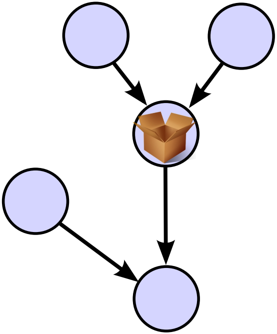
Precise, fully-automatic technique that integrates container reasoning into heap analysis

1. tracks key-value correlations
2. can model nested containers in a precise way
3. unifies heap and container analysis



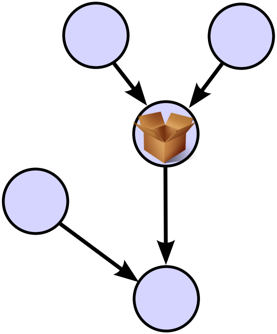
- To integrate containers into heap analysis, we model containers as abstract memory locations in the heap abstraction

Integrating Container Reasoning into Heap Analysis



- To integrate containers into heap analysis, we model containers as abstract memory locations in the heap abstraction

Integrating Container Reasoning into Heap Analysis



- To integrate containers into heap analysis, we model containers as abstract memory locations in the heap abstraction
- For precise, per-element reasoning, we model containers using indexed locations we introduced in ESOP'10 for reasoning about arrays

Indexed Locations

$\langle container \rangle_i$

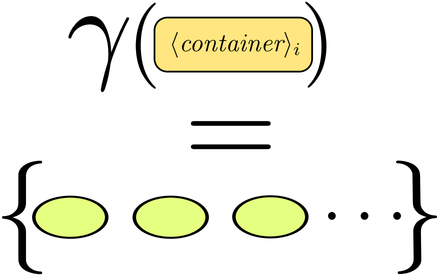
- Container represented using a single abstract location qualified by index variable

Indexed Locations

$\langle container \rangle_i$

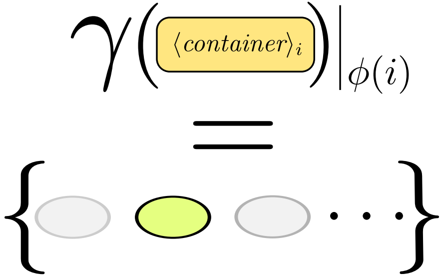
- Container represented using a single abstract location qualified by index variable
- Index variable ranges over possible elements of container

Indexed Locations



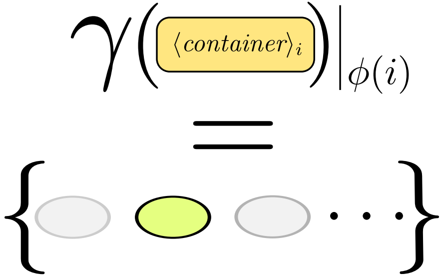
- Container represented using a single abstract location qualified by index variable
- Index variable ranges over possible elements of container

Indexed Locations



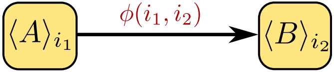
- Container represented using a single abstract location qualified by index variable
- Index variable ranges over possible elements of container

Indexed Locations



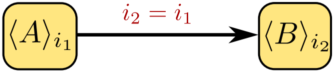
- Container represented using a single abstract location qualified by index variable
- Index variable ranges over possible elements of container
- Key advantage: Can refer to individual elements in container using only one abstract location

Symbolic Points-to Relations



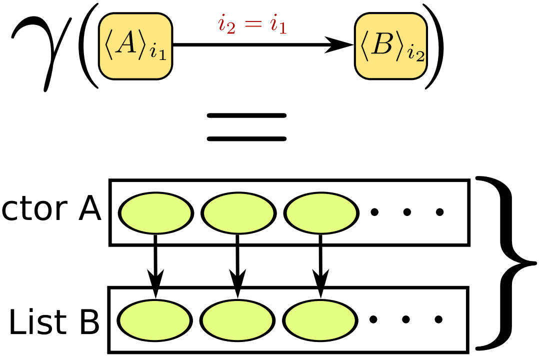
Points-to edges are qualified by constraints on index variables.

Symbolic Points-to Relations



Points-to edges are qualified by constraints on index variables.

Symbolic Points-to Relations



Points-to edges are qualified by constraints on index variables.

Modeling Value-Dependent Containers

Problem

- Natural representation for position-dependent containers

Modeling Value-Dependent Containers

Problem

- Natural representation for position-dependent containers
- But how do we represent points-to relations for value-dependent containers?

Problem

- Natural representation for position-dependent containers
- But how do we represent points-to relations for value-dependent containers?

Solution

Introduce a level of indirection mapping keys to abstract indices

Key-to-Index Mapping for Value-Dependent Containers

- For value-dependent containers, any such key-to-index mapping M must satisfy the axiom:

$$\forall k_1, k_2.\ M(k_1) = M(k_2) \Rightarrow k_1 = k_2$$

Key-to-Index Mapping for Value-Dependent Containers

- For value-dependent containers, any such key-to-index mapping M must satisfy the axiom:

$$\forall k_1, k_2.\ M(k_1) = M(k_2) \Rightarrow k_1 = k_2$$

- Otherwise, distinct keys may map to same index, overwriting each other's value

- For value-dependent containers, any such key-to-index mapping M must satisfy the axiom:

$$\forall k_1, k_2.\ M(k_1) = M(k_2) \Rightarrow k_1 = k_2$$

- Otherwise, distinct keys may map to same index, overwriting each other's value

- Thus, for soundness, M's inverse is a function

Is this Mapping a Function?

Is this Mapping a Function?

Two Alternatives

1. To model multimaps, multisets directly, allow same key can map to different abstract indices

Is this Mapping a Function?

Two Alternatives

1. To model multimaps, multisets directly, allow same key can map to different abstract indices $\Rightarrow M$ is not a function

Is this Mapping a Function?

Two Alternatives

1. To model multimaps, multisets directly, allow same key can map to different abstract indices $\Rightarrow M$ is not a function

2. Or model data structures that allow multiple values as nested data structures

Is this Mapping a Function?

Two Alternatives

1. To model multimaps, multisets directly, allow same key can map to different abstract indices
$\Rightarrow$ M is not a function

2. Or model data structures that allow multiple values as nested data structures
$\Rightarrow$ make M a function

Two Alternatives

1. To model multimaps, multisets directly, allow same key can map to different abstract indices
$\Rightarrow$ M is not a function

2. Or model data structures that allow multiple values as nested data structures
$\Rightarrow$ make M a function

Using Invertible, Uninterpreted Functions

$pos($$) = \chi$

$$\Leftrightarrow$$

$pos^{-1}(\chi) =$

Thus, map key to index in abstract location using invertible, uninterpreted function

Simple Example

- Consider map `scores` mapping student names (strings) to a vector of their grades.

Simple Example

- Consider map `scores` mapping student names (strings) to a vector of their grades.

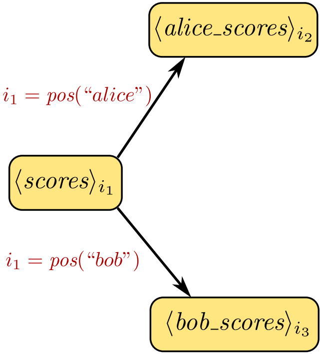
$\langle scores \rangle_{i_1}$

Simple Example

$\langle scores \rangle_{i_1}$

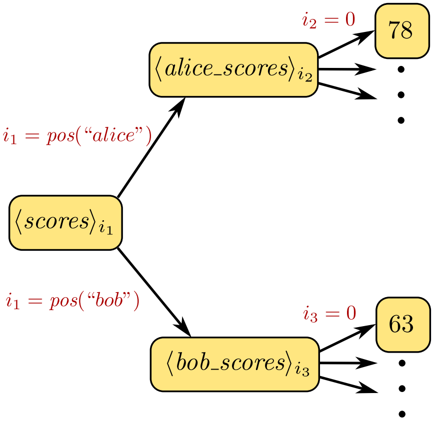
- Consider map `scores` mapping student names (strings) to a vector of their grades.
- Map initially contains scores associated with two students: Alice and Bob

Simple Example



- Consider map `scores` mapping student names (strings) to a vector of their grades.
- Map initially contains scores associated with two students: Alice and Bob

Simple Example

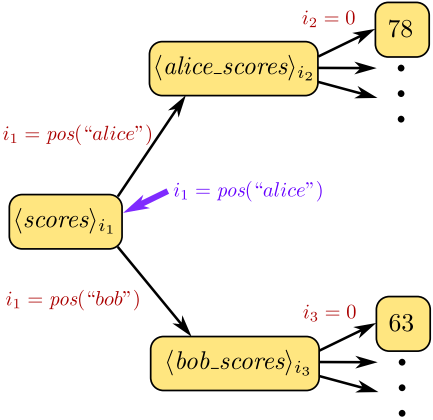


- Consider map `scores` mapping student names (strings) to a vector of their grades.
- Map initially contains scores associated with two students: Alice and Bob
- Alice's first score is 78; Bob's first score is 63

- We have seen how to represent containers

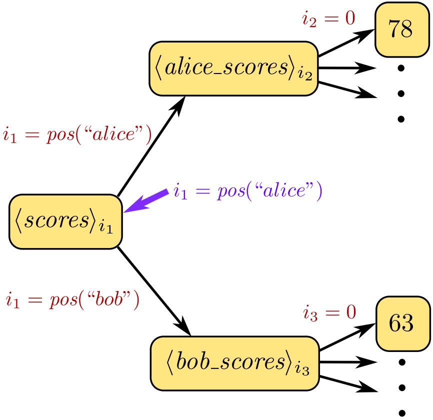
- We have seen how to represent containers
- But how do we statically analyze statements that manipulate them?

Simple Example: Reading from Containers



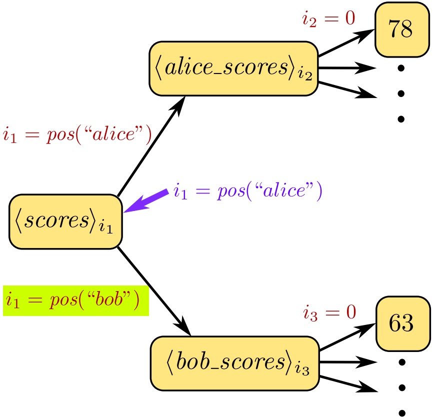
- What is the value of `scores["alice"][0]`?

Simple Example: Reading from Containers



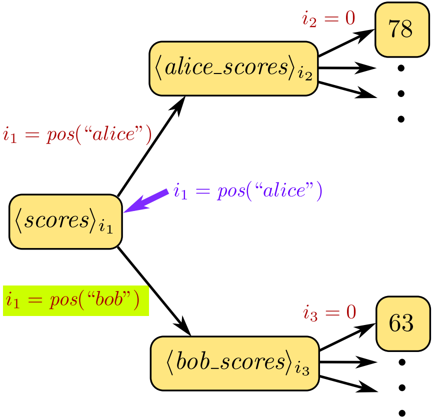
- What is the value of `scores["alice"][0]`?
- Determine where `scores` points to under $i_1 = pos(\text{"alice"})$

Simple Example: Reading from Containers

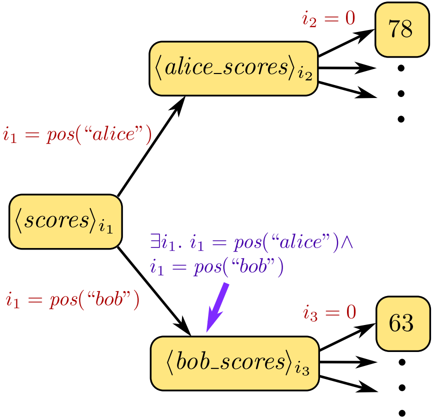


- What is the value of `scores["alice"][0]`?
- Determine where `scores` points to under $i_1 = pos(\text{“}alice\text{”})$
- $i_1 = pos(\text{“}bob\text{”})$

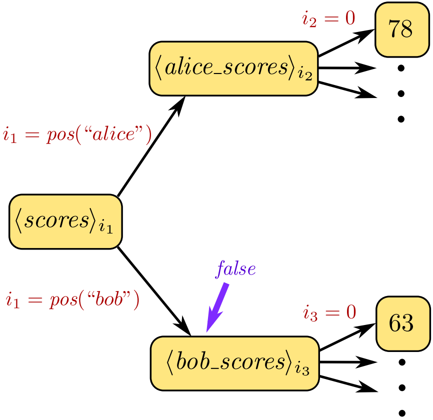
Simple Example: Reading from Containers



- What is the value of `scores["alice"][0]`?
- Determine where scores points to under $i_1 = pos(\text{"}alice\text{"})$
- $i_1 = pos(\text{"}bob\text{"}) \land i_1 = pos(\text{"}alice\text{"})$



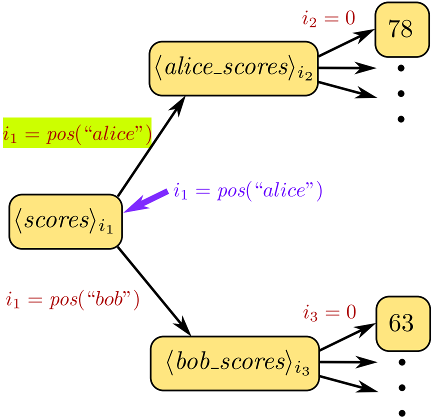
- What is the value of `scores["alice"][0]`?
- Determine where scores points to under $i_1 = pos(\text{“}alice\text{”})$
- $\exists i_1 . i_1 = pos(\text{“}bob\text{”}) \land i_1 = pos(\text{“}alice\text{”})$



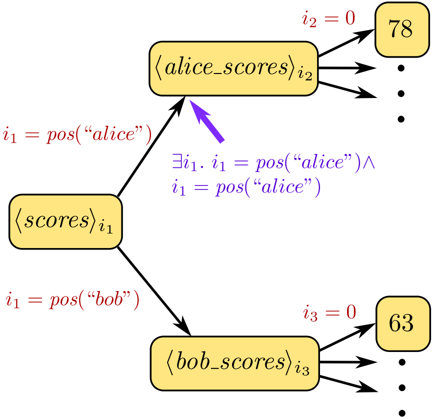
- What is the value of `scores["alice"][0]`?
- Determine where `scores` points to under $i_1 = pos(\text{"alice"})$
- $\exists i_1 . i_1 = pos(\text{"bob"}) \land i_1 = pos(\text{"alice"})$

$\Rightarrow$ UNSAT because pos is invertible

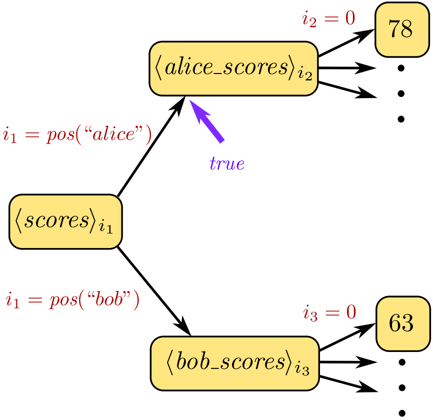
Simple Example: Reading from Containers



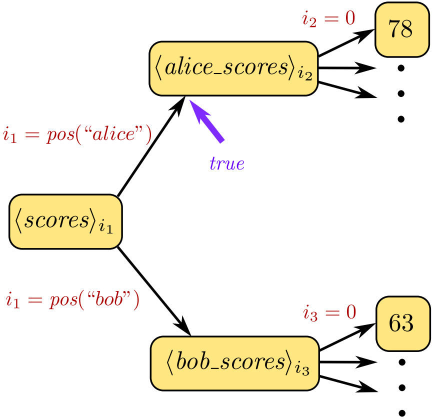
Simple Example: Reading from Containers



Simple Example: Reading from Containers

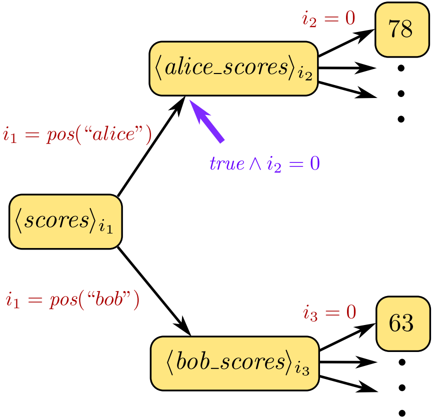


Simple Example: Reading from Containers



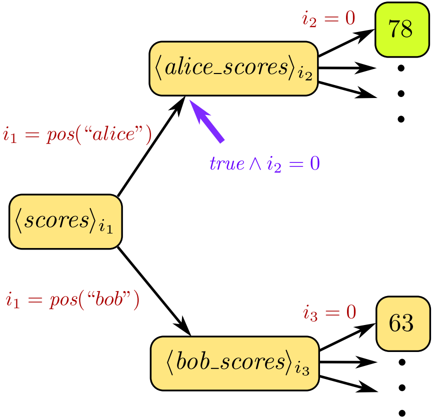
- Thus, entry for "alice" points to vector represented by $\langle alice\_scores \rangle_{i_2}$

Simple Example: Reading from Containers



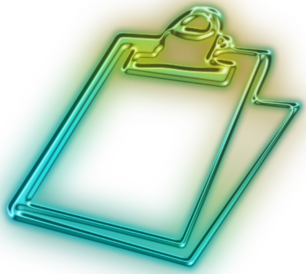
- Thus, entry for "alice" points to vector represented by $\langle alice\_scores \rangle_{i_2}$
- Finally, determine where $\langle alice\_scores \rangle_{i_2}$ points to under constraint $i_2 = 0$

Simple Example: Reading from Containers



- Thus, entry for "alice" points to vector represented by $\langle alice\_scores \rangle_{i_2}$
- Finally, determine where $\langle alice\_scores \rangle_{i_2}$ points to under constraint $i_2 = 0$

Summary: Reading from Containers



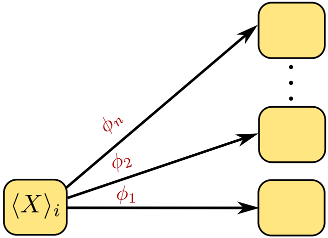
- Statically analyzing reads from containers requires checking for satisfiability and existential quantifier elimination

Summary: Reading from Containers

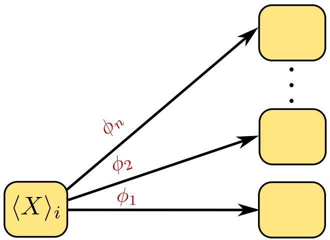
- Statically analyzing reads from containers requires checking for satisfiability and existential quantifier elimination
- Use of invertible functions for key-value mapping is crucial for precisely tracking key-value correlations

How do we analyze stores to containers?

Writing to Containers



Consider storing object Y for key k in container X:

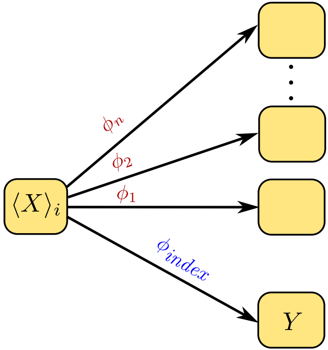


Consider storing object Y for key k in container X:

1. Compute

$$\phi_{index} : \begin{cases} i = k & \text{X position-dependent} \\ i = pos(k) & \text{X value-dependent} \end{cases}$$

Writing to Containers



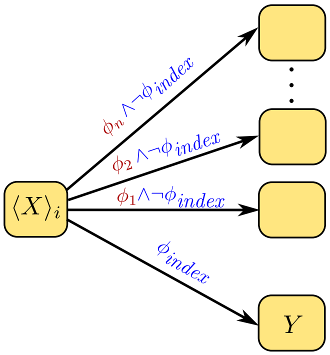
Consider storing object Y for key k in container X:

1. Compute

$$\phi_{index} : \begin{cases} i = k & \text{X position-dependent} \\ i = pos(k) & \text{X value-dependent} \end{cases}$$

2. Add edge to Y under ϕ_{index}

Writing to Containers



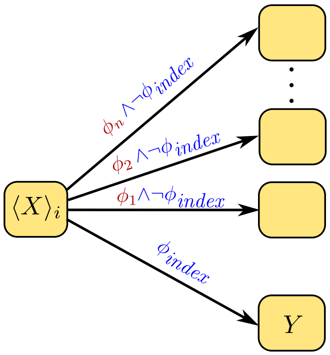
Consider storing object Y for key k in container X:

1. Compute

$$\phi_{index} : \begin{cases} i = k & \text{X position-dependent} \\ i = pos(k) & \text{X value-dependent} \end{cases}$$

2. Add edge to Y under ϕ_{index}
3. Preserve existing edges under $\neg\phi_{index}$

Writing to Containers



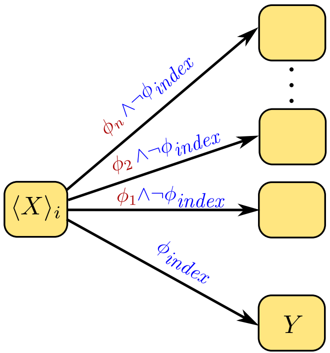
Consider storing object Y for key k in container X:

1. Compute

$$\phi_{index} : \begin{cases} i = k & \text{X position-dependent} \\ i = pos(k) & \text{X value-dependent} \end{cases}$$

2. Add edge to Y under ϕ_{index}
3. Preserve existing edges under $\neg\phi_{index}$

Need bracketing constraints $\langle \phi_{may}, \phi_{must} \rangle$ for sound negation



Consider storing object Y for key k in container X:

1. Compute

$$\phi_{index} : \begin{cases} i = k & \text{X position-dependent} \\ i = pos(k) & \text{X value-dependent} \end{cases}$$

2. Add edge to Y under ϕ_{index}
3. Preserve existing edges under $\neg\phi_{index}$

Need bracketing constraints $\langle \phi_{may}, \phi_{must} \rangle$ for sound negation
$\Rightarrow \neg\langle \phi_{may}, \phi_{must} \rangle = \langle \neg\phi_{must}, \neg\phi_{may} \rangle$

Allocations

- Nested containers usually involve dynamic memory allocation

Allocations

- Nested containers usually involve dynamic memory allocation

⇒ Precise reasoning about nested containers requires precise reasoning about memory allocations

- Nested containers usually involve dynamic memory allocation

⇒ Precise reasoning about nested containers requires precise reasoning about memory allocations

- Need to distinguish between allocations in different loop iterations or recursive calls

Allocations

Consider the following example

```
for(int i=0; i<N; i++)
  v.push_back(new map());
```

Allocations

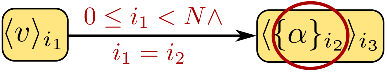
Consider the following example

```
for(int i=0; i<N; i++)
  v.push_back(new map());
```

Difficulty

Statically unknown number of allocations

Allocations



Consider the following example

```
for(int i=0; i<N; i++)
  v.push_back(new map());
```

Solution

Model allocation with indexed location

Allocations

Consider the following example

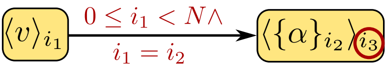
```
for(int i=0; i<N; i++)
  v.push_back(new map());
```

Solution

Model allocation with indexed location

- i_2 differentiates allocations from different loop iterations

Allocations



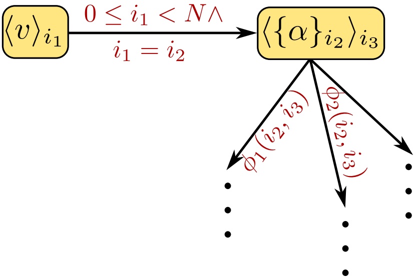
Consider the following example

```
for(int i=0; i<N; i++)
  v.push_back(new map());
```

Solution

Model allocation with indexed location

- i_2 differentiates allocations from different loop iterations
- i_3 differentiates indices in map



Consider the following example

```
for(int i=0; i<N; i++)
  v.push_back(new map());
```

Solution

Model allocation with indexed location

- i_2 differentiates allocations from different loop iterations
- i_3 differentiates indices in map
- Outgoing edges from $\langle \{\alpha\}_{i_2} \rangle_{i_3}$ qualify both i_2 and i_3

Implementation

- Implemented heap/container analysis in our Compass program analysis framework for C and C++ programs

Implementation

- Implemented heap/container analysis in our Compass program analysis framework for C and C++ programs
- Analysis requires solving constraints in combined theory of linear inequalities over integers and uninterpreted functions and quantifier elimination
  ⇒ used our Mistral SMT solver

Experiments

- Analyzed real open-source C++ applications using containers

Experiments

- Analyzed real open-source C++ applications using containers
  - LiteSQL, 16,030 LOC

Experiments

- Analyzed real open-source C++ applications using containers
  - LiteSQL, 16,030 LOC
  - Inkscape Widget Library, 37,211 LOC

Experiments

- Analyzed real open-source C++ applications using containers
  - LiteSQL, 16,030 LOC
  - Inkscape Widget Library, 37,211 LOC
  - DigiKam, 128,318 LOC

Ran our Compass verification tool

- Detect all possible segmentation faults or run-time exceptions caused by:
  - null dereference errors
  - accessing deleted memory
- Also checked memory leaks

First Experiment:

- Represent containers as bags of values

First Experiment

First Experiment:

- Represent containers as bags of values
- Existing tools that analyze programs of this size use this abstraction

First Experiment:

- Represent containers as bags of values
- Existing tools that analyze programs of this size use this abstraction
- To achieve this effect, we modeled containers using summary nodes

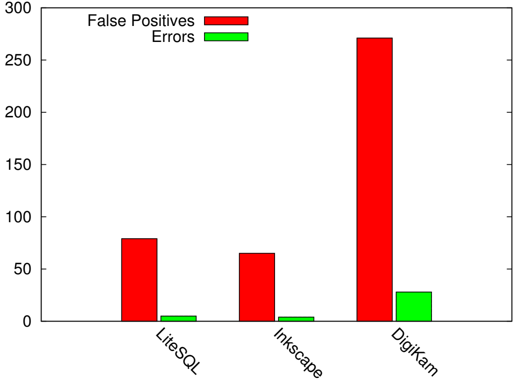
First Experiment

First Experiment:

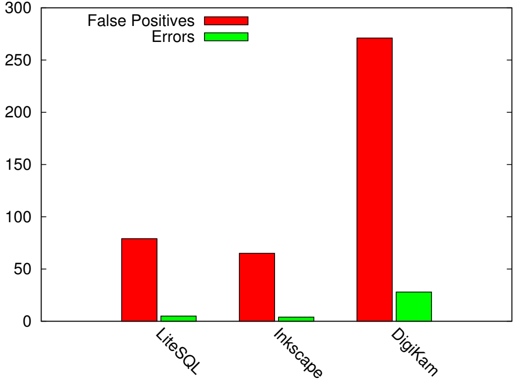
- Represent containers as bags of values
- Existing tools that analyze programs of this size use this abstraction
- To achieve this effect, we modeled containers using summary nodes

⇒ Cannot track index-to-value correlations, modification to one container element contaminates all others

Containers as Bags



Containers as Bags



Conclusion

Treating containers as bags leads to unacceptable number of false alarms.

Second Experiment

Second Experiment:

- Used the techniques described in this talk: indexed locations, symbolic points-to relations

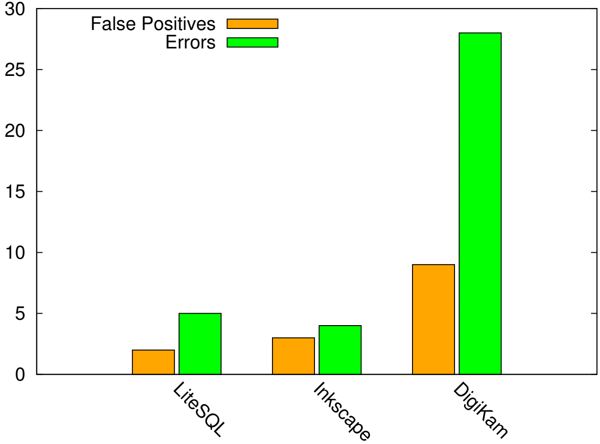
Second Experiment

Second Experiment:

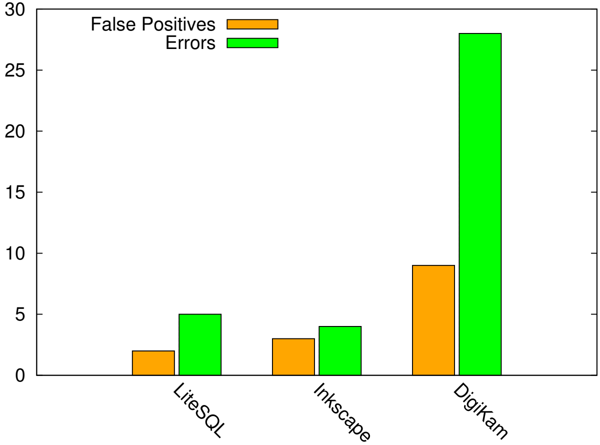
- Used the techniques described in this talk: indexed locations, symbolic points-to relations

$\Rightarrow$ Able to track key-value correlations; precise reasoning about heap objects stored in containers

Containers Modeled as Indexed Locations

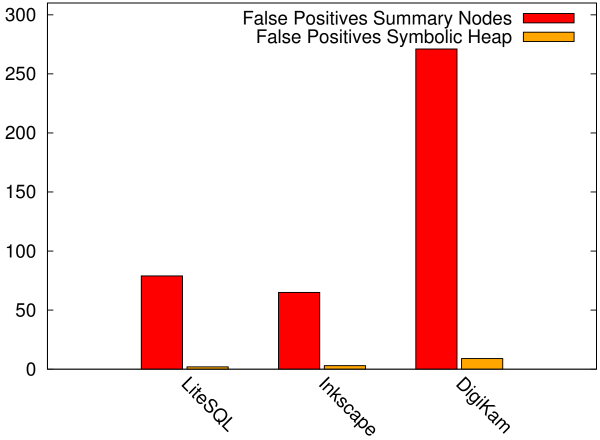


Containers Modeled as Indexed Locations

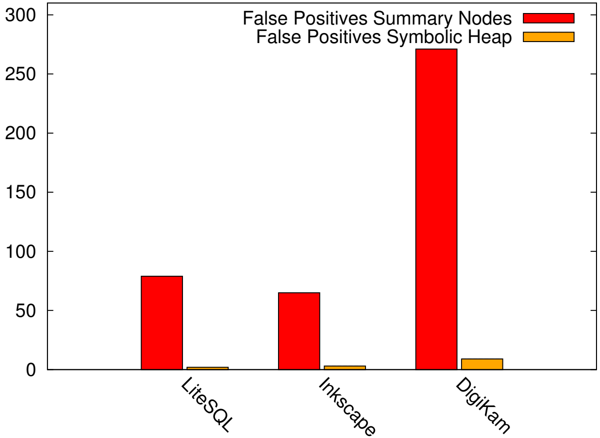


✓ Analysis reports very few false positives

Containers Modeled as Indexed Locations

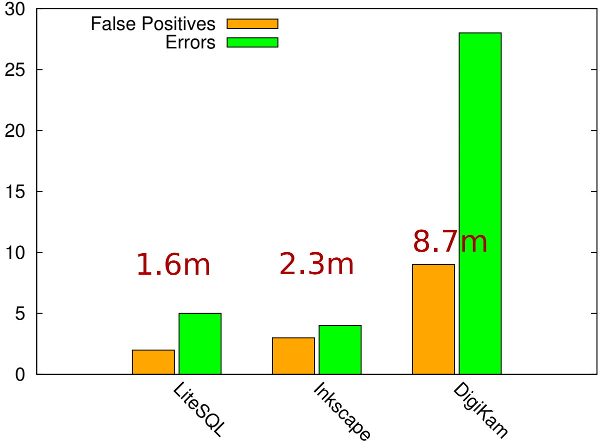


Containers Modeled as Indexed Locations



✓ More than an order of magnitude reduction compared to less precise analysis

Containers Modeled as Indexed Locations



✓ Cost of the analysis is tractable

Contributions

- A sound, precise, and automatic technique for client-side reasoning about contents of an important family of data structures

Contributions

- A sound, precise, and automatic technique for client-side reasoning about contents of an important family of data structures
- Precise reasoning for key-value correlations, nested data structures, and dynamic allocations

Contributions

- A sound, precise, and automatic technique for client-side reasoning about contents of an important family of data structures
- Precise reasoning for key-value correlations, nested data structures, and dynamic allocations
- First practical verification of container- and heap-manipulating programs

Related Work

Dillig, I., Dillig, T., Aiken, A.:
Fluid Updates: Beyond Strong vs. Weak Updates.
In: ESOP. (2010)

Lam, P., Kuncak, V., Rinard, M.:
Hob: A Tool for Verifying Data Structure Consistency.
In: CC. 237–241

Reps, T.W., Sagiv, S., Wilhelm, R.:
Static Program Analysis via 3-Valued Logic.
In: CAV. (2004) 15–30

Deutsch, A.:
Interprocedural May-Alias Analysis for Pointers: Beyond k-limiting.
In: PLDI. (1994) 230–241

Marron, M., Stefanovic, D., Hermenegildo, M., Kapur, D.:
Heap Analysis in the Presence of Collection Libraries.
In: PASTE. (2007)