Precise Reasoning for Programs Using Containers

Işıl Dillig  Thomas Dillig  Alex Aiken
Stanford University
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

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

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.
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

Position-dependent Containers

<table>
<thead>
<tr>
<th>Container Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>vector</strong></td>
<td>A dynamic array, like C array (i.e., capable of random access) with the ability to resize itself automatically when inserting or erasing an element. Inserting and removing an element from the front or 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.</td>
</tr>
<tr>
<td><strong>list</strong></td>
<td>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).</td>
</tr>
<tr>
<td><strong>deque (double-ended queue)</strong></td>
<td>A vector with insertion/erasure at the beginning or end in amortized constant time, however lacking some guarantee on iterator validity after altering the deque.</td>
</tr>
</tbody>
</table>

**Container adaptors**

<table>
<thead>
<tr>
<th>Container Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>queue</strong></td>
<td>Provides FIFO queue interface in terms of pushN/popN/frontN/back operations. Any sequence supporting operations front(), back(), push_back(), and pop_front() can be used to instantiate queue (e.g., list and deque).</td>
</tr>
<tr>
<td><strong>priority_queue</strong></td>
<td>Provides priority queue interface in terms of push/pop/replace 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).</td>
</tr>
<tr>
<td><strong>stack</strong></td>
<td>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).</td>
</tr>
</tbody>
</table>

**Associative containers - unordered collections**

<table>
<thead>
<tr>
<th>Container Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>set</strong></td>
<td>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 &lt; or custom comparator function must be specified; such comparison operator or comparator function must guarantee strict weak ordering. Otherwise, behavior is undefined. Typically implemented as a self-balancing binary search tree.</td>
</tr>
<tr>
<td><strong>multiset</strong></td>
<td>Same as a set, but allows duplicate elements.</td>
</tr>
<tr>
<td><strong>map</strong></td>
<td>An associative array; allows mapping from one data item (a key) to another (a value). Type of key must implement comparison operator &lt; or custom comparator function must be specified; such comparison operator or comparator function must guarantee strict weak ordering, otherwise, behavior is undefined. Typically implemented as a self-balancing binary search tree.</td>
</tr>
<tr>
<td><strong>multimap</strong></td>
<td>Same as a map, but allows duplicate keys.</td>
</tr>
<tr>
<td><strong>hash_set</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>hash_multiset</strong></td>
<td>Same as hash_set.</td>
</tr>
<tr>
<td><strong>hash_map</strong></td>
<td>Same as hash_set.</td>
</tr>
<tr>
<td><strong>hash_multimap</strong></td>
<td>Same as hash_set.</td>
</tr>
</tbody>
</table>
## Classification of Containers

### Position-dependent Containers
- **Well-defined meaning of position**

<table>
<thead>
<tr>
<th>Container</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>vector</strong></td>
<td>A dynamic array, like C array (i.e., capable of random access) with the ability to resize itself automatically when inserting or erasing an element. Inserting and removing an element from 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.</td>
</tr>
<tr>
<td><strong>list</strong></td>
<td>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).</td>
</tr>
<tr>
<td><strong>deque (double-ended queue)</strong></td>
<td>A vector with insertion/deletion at the beginning or end in amortized constant time, however lacking some guarantee on iterator validity after altering the deque.</td>
</tr>
<tr>
<td><strong>queue</strong></td>
<td>Provides FIFO queue interface in terms of <code>push()/pop()/front()/back</code> operations. Any sequence supporting operations <code>front()</code>, <code>back()</code>, <code>push_back()</code>, and <code>pop_front()</code> can be used to instantiate queue (e.g., list and deque).</td>
</tr>
<tr>
<td><strong>priority_queue</strong></td>
<td>Provides priority queue interface in terms of <code>push()/pop()/top()</code> operations (the element with the highest priority is on top). Any random-access sequence supporting operations <code>front()</code>, <code>push_back()</code>, and <code>pop_back()</code> can be used to instantiate <code>priority_queue</code> (e.g., vector and deque). Elements should additionally support comparison (to determine which element has a higher priority and should be popped first).</td>
</tr>
<tr>
<td><strong>stack</strong></td>
<td>Provides LIFO stack interface in terms of <code>push()/pop()/top()</code> operations (the last-inserted element is on top). Any sequence supporting operations <code>back()</code>, <code>push_back()</code>, and <code>pop_back()</code> can be used to instantiate stack (e.g., vector, list, and deque).</td>
</tr>
</tbody>
</table>

### Associative containers - unordered collections
- **set**: A mathematical set; inserting/deleting 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.
- **hash_set**, **hash_multiset**: Similar to 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 SGIs STL extensions, and are included in common libraries such as the GNU C++ Library in the `<unordered>` 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.
### Classification of Containers

<table>
<thead>
<tr>
<th>Sequence (Arrays / Linked Lists) - ordered collections</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>vector</strong></td>
</tr>
<tr>
<td><strong>list</strong></td>
</tr>
<tr>
<td><strong>deque (double-ended queue)</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Container adaptors</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>queue</strong></td>
</tr>
<tr>
<td><strong>priority_queue</strong></td>
</tr>
<tr>
<td><strong>stack</strong></td>
</tr>
</tbody>
</table>

### Position-dependent Containers
- Well-defined meaning of position
- Iteration in a pre-defined order
Classification of Containers

<table>
<thead>
<tr>
<th>Container Classifications</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position-dependent Containers</td>
<td>Well-defined meaning of position, Iteration in a pre-defined order</td>
</tr>
<tr>
<td>Value-dependent Containers</td>
<td></td>
</tr>
</tbody>
</table>

### Position-dependent Containers

1. **Well-defined meaning of position**
2. **Iteration in a pre-defined order**

### Value-dependent Containers

- Provides LIFO stack interface in terms of push/pop/top operations (the last-inserted element is on top).
- Any sequence supporting operations `push()`, `pop()`, and `top()` 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. Provided 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 `<unordered.container>` 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.
Classification of Containers

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

2. Value-dependent Containers
   - Keys of arbitrary type

Sequences (Arrays/Linked Lists) - ordered collections

- vector
  - a dynamic array, like C array (e.g., capable of random access) with the ability to resize itself automatically when inserting or erasing an element. Inserting and removing an element from 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/erasure at the beginning or end in amortized constant time, however lacking some guarantee on iterator validity after altering the deque.

Container adaptors

- queue
  - Provides FIFO queue interface in terms of push/pop/peek/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/peek operations (the element with the highest priority is on top).
  - Any random-access sequence supporting operations front(), push_heap(), and pop_heap() 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/peek 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 into 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
  - 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.
### Position-dependent Containers

- Well-defined meaning of position
- Iteration in a pre-defined order

### Value-dependent Containers

- Keys of arbitrary type
- Iteration order may be undefined
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
Precise, fully-automatic technique that integrates container reasoning into heap analysis.
Precise, fully-automatic technique that integrates container reasoning into heap analysis

- tracks key-value correlations
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
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.
To integrate containers into heap analysis, we model containers as abstract memory locations in the heap abstraction.
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

Container represented using a single abstract location qualified by index variable
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

\[ \gamma \left( \langle \text{container} \rangle_i \right) \mid \phi(i) \]

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

\[ \gamma((\text{container}_i) | \phi(i)) \]

- 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
Points-to edges are qualified by constraints on index variables.
Points-to edges are qualified by constraints on index variables.
Points-to edges are qualified by constraints on index variables.
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?
Modeling 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
For value-dependent containers, any such key-to-index mapping $M$ must satisfy the axiom:

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

Two Alternatives

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

2. Or model data structures that allow multiple values as nested data structures. \(M\) makes a function.
Two Alternatives

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

1. To model multimaps, multisets directly, allow same key can map to different abstract indices
   \[\implies M \text{ is not a function}\]
Two Alternatives

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

2. Or model data structures that allow multiple values as nested data structures
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
   \[ \Rightarrow M \text{ is not a function} \]

2. Or model data structures that allow multiple values as nested data structures
   \[ \Rightarrow \text{make } M \text{ a function} \]
\[ \text{pos}(\chi) = \chi \]

\[ \text{pos}^{-1}(\chi) = \chi \]

Thus, map key to index in abstract location using invertible, uninterpreted function.
Consider map \textit{scores} mapping student names (strings) to a vector of their grades.
Simple Example

Consider map $\textit{scores}$ mapping student names (strings) to a vector of their grades.
Consider map $\text{scores}$ mapping student names (strings) to a vector of their grades.

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

Map initially contains scores associated with two students: Alice and Bob
Consider map \texttt{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 \(\text{scores["alice"] [0]}\)?

\[
i_1 = \text{pos(“alice”)}
\]

\[
\langle \text{Alice} \rangle_{i_2} \rightarrow 78
\]

\[
i_2 = 0
\]

\[
i_1 = \text{pos(“alice”)}
\]

\[
\langle \text{scores} \rangle_{i_1}
\]

\[
i_1 = \text{pos(“bob”)}
\]

\[
\langle \text{Bob} \rangle_{i_3} \rightarrow 63
\]

\[
i_3 = 0
\]
Simple Example: Reading from Containers

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

Determine where `scores` points to under $i_1 = \text{pos}(\text{“alice”})$.

- $i_1 = \text{pos}(\text{“alice”})$
- $i_1 = \text{pos}(\text{“bob”})$
- $i_2 = 0$  
- $i_3 = 0$  
- $78$
- $63$
Simple Example: Reading from Containers

- What is the value of \( \text{scores["alice"]}[0] \)?

- Determine where \( \text{scores} \) points to under \( i_1 = \text{pos}(\text{"alice"}) \)

- \( i_1 = \text{pos}(\text{"bob"}) \)
Simple Example: Reading from Containers

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

Determine where `scores` points to under $i_1 = pos(\text{"alice"})$.

- $i_1 = pos(\text{"bob"}) \land i_1 = pos(\text{"alice"})$

$i_1$ is pos(“alice”)
Simple Example: Reading from Containers

What is the value of $\text{scores["alice"]}[0]$?

Determine where $\text{scores}$ points to under $i_1 = \text{pos(“alice”)}$

$\exists i. i_1 = \text{pos(“bob”) \land i_1 = \text{pos(“alice”)}}$

$I\¸sıl$ Dillig  Thomas Dillig  Alex Aiken
What is the value of \( \text{scores["alice"]}[0] \)?)

Determine where \( \text{scores} \) points to under \( i_1 = \text{pos(“alice”)} \)

\( \exists i_1. i_1 = \text{pos(“bob”)} \land i_1 = \text{pos(“alice”)} \)

\( \Rightarrow \) UNSAT because \( \text{pos} \) is invertible
Simple Example: Reading from Containers

Thus, entry for "alice" points to vector represented by \( \langle alice\_scores \rangle \).

Finally, determine where \( \langle alice\_scores \rangle \) points to under constraint \( i_2 = 0 \).

\[ i_1 = pos(“alice”) \]

\[ i_1 = pos(“bob”) \]

\[ i_3 = 0 \]

\( \langle scores \rangle \)

\( \langle alice\_scores \rangle \)

\( \langle bob\_scores \rangle \)

78

63
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 \( i_2 = 0 \).

\( i_1 = \text{pos}(“alice”) \)

\( \exists i_1. i_1 = \text{pos}(“alice”) \land i_1 = \text{pos}(“alice”) \)

\( i_1 = \text{pos}(“bob”) \)

\( i_3 = 0 \)

\( \langle bob\_scores \rangle_{i_3} \)

78

63
Thus, entry for "alice" points to \( \langle alice_{\text{scores}} \rangle_{i_2} \). Finally, determine where \( \langle alice_{\text{scores}} \rangle_{i_2} \) points to under constraint \( i_2^2 = 0 \).
Thus, entry for “alice” points to vector represented by \( \langle alice\_scores \rangle_{i_2} \).
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 $i_2 = 0$
Thus, entry for “alice” points to vector represented by \(\langle alice\_scores\rangle_i\). 

Finally, determine where \(\langle alice\_scores\rangle_i\) points to under constraint \(i_2 = 0\).
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?
Consider storing object $Y$ for key $k$ in container $X$:

1. Compute $\phi$ index:
   - $i = k \cdot X$ (position-dependent)
   - $i = \text{pos}(k) \cdot X$ (value-dependent)

2. Add edge to $Y$ under $\phi$ index

3. Preserve existing edges under $\neg \phi$ index

Need bracketing constraints $\langle \phi \text{ may}, \phi \text{ must} \rangle$ for sound negation:

$\neg \langle \phi \text{ may}, \phi \text{ must} \rangle = \langle \neg \phi \text{ must}, \neg \phi \text{ may} \rangle$
Consider storing object $Y$ for key $k$ in container $X$:

1. Compute

$$\phi_{\text{index}} : \begin{cases} 
  i = k & \text{X position-dependent} \\
  i = \text{pos}(k) & \text{X value-dependent}
\end{cases}$$
Consider storing object $Y$ for key $k$ in container $X$:

1. Compute

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

2. Add edge to $Y$ under $\phi_{\text{index}}$
Consider storing object $Y$ for key $k$ in container $X$:

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

2. **Add edge to** $Y$ under $\phi_{index}$

3. **Preserve existing edges under** $\neg \phi_{index}$
Writing to Containers

Consider storing object $Y$ for key $k$ in container $X$:

1. Compute
   \[ \phi \text{ index} : \begin{cases} 
   i = k & \text{X position-dependent} \\
   i = \text{pos}(k) & \text{X value-dependent}
   \end{cases} \]

2. Add edge to $Y$ under $\phi \text{ index}$

3. Preserve existing edges under $\neg \phi \text{ index}$

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

3. Preserve existing edges under $\neg \phi_{index}$

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

\[ \neg \langle \phi_{may}, \phi_{must} \rangle = \langle \neg \phi_{must}, \neg \phi_{may} \rangle \]
Nested containers usually involve dynamic memory allocation.
Nested containers usually involve dynamic memory allocation

⇒ Precise reasoning about nested containers requires precise reasoning about memory allocations
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
Consider the following example

```cpp
for(int i=0; i<N; i++)
    v.push_back(new map());
```
Consider the following example

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

**Difficulty**

Statically unknown number of allocations
Consider the following example:

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

Solution

Model allocation with indexed location
Consider the following example

```cpp
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
Consider the following example

```cpp
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

```cpp
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 $\langle\{\alpha\}_{i_2}i_3$ qualify both $i_2$ and $i_3$
Implemented heap/container analysis in our Compass program analysis framework for C and C++ programs.
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
\[ \Rightarrow \] 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:

- 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:

- 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

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

Işıl Dillig  Thomas Dillig  Alex Aiken
Conclusion

Treating containers as bags leads to an unacceptable number of false alarms.
Second Experiment:

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

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

⇒ Able to track key-value correlations; precise reasoning about heap objects stored in containers
Analysis reports very few false positives.
Analysis reports very few false positives
More than an order of magnitude reduction compared to less precise analysis.
More than an order of magnitude reduction compared to less precise analysis
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.

Precise reasoning for key-value correlations, nested data structures, and dynamic allocations.

First practical verification of container- and heap-manipulating programs.
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
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.  

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)