A Product of Shape & Sequences abstractions

Static Analysis Symposium, Cascais, 2023

Josselin Giet¹, Félix Ridoux²,³, Xavier Rival¹
October, 22 2023

¹INRIA Paris/CNRS/École Normale Supérieure/PSL Research University, Paris, France
²IMDEA Software Institute, Madrid, Spain
³Univ Rennes, F-35000 Rennes, France
Introduction
What do we want to verify?

When we talk about automatic static analysis of program manipulating dynamic data-structure, there are several properties we are interested in.

```c
int insert(tree *t, int v) {
    tree *m = malloc(sizeof(tree));
    m->left = m->right = NULL;
    m->data = v;
    if (!t) {
        // Empty case
    } else {
        tree *c = t;
        while (v < c->data && c->left || v >= c->data && c->right)
            if (v <= c->data) {
                c = c->left;
            } else {
                c = c->right;
            }
        if (v <= c->data) {
            c->left = m;
        } else {
            c->right = m;
        }
        return t;
    }
}
```

1. **No ill-pointer (null, ...) dereference "c-"**
2. **Preservation of structural invariants**
   "If \(t\) is a well-formed binary tree then so is the returned value."
3. **Partial functional correctness**
   "If \(t\) is a well-formed BST, then the returned value \(r\) should be a well-formed BST containing the same elements as \(t\) plus value \(v\)."
### Comparison of existing static analysis over dynamic data structures

Various automatic static analysis over dynamic data-structures have been proposed:

<table>
<thead>
<tr>
<th>Analysis</th>
<th>pointer dereference</th>
<th>structural invariants</th>
<th>partial f^\text{al}\text{orrectness}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointer analysis</td>
<td>✓</td>
<td>✓</td>
<td>✗ ✗</td>
</tr>
<tr>
<td>Shape analysis based on...</td>
<td></td>
<td></td>
<td>SLL</td>
</tr>
<tr>
<td>...3-Value logic</td>
<td>✓</td>
<td>✓</td>
<td>✗ ✗</td>
</tr>
<tr>
<td>e.g. [Sagiv et al. TOPLAS, 99]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...Separation logic</td>
<td>✓</td>
<td>✓</td>
<td>✗ ✗</td>
</tr>
<tr>
<td>e.g. [Chang et al. POPL, 08]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...k-limited graphs</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>e.g. [Bouajjani et al, CAV, 10]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

None of these approaches could prove functional correctness of insertion into a binary search tree!
Comparison of existing static analysis over dynamic data structures

Various automatic static analysis over dynamic data-structures have been proposed:

<table>
<thead>
<tr>
<th>Analysis</th>
<th>pointer dereference</th>
<th>structural invariants</th>
<th>partial f\textsuperscript{a}l correctness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointer analysis</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Shape analysis based on...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...3-Value logic</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>e.g. [Sagiv et al. TOPLAS, 99]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...Separation logic</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>e.g. [Chang et al. POPL, 08]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...k-limited graphs</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>e.g. [Bouajjani et al, CAV, 10]</td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

None of these approaches could prove functional correctness of insertion into a binary search tree!

How to improve the expressiveness of automatic static analysis over dynamic data-structures to prove partial functional correctness?
Separation Logic based shape analysis

[Chang et al. POPL, 2008] introduces a shape analysis based on abstract interpretation.

It uses a subset of separation logic [Reynolds, LICS 02] as an abstract representation for memory states:

- Abstract memory regions are connected with the separating conjunction. It expresses that these regions are disjoint. This allows to reason locally.

- Inductive data-structures are synthesized by inductive predicates

**Example** The predicate `tree(c)`, denoting a binary tree:
Inductive predicates are not expressive enough

\[ c \triangleq 0x0 \land E = \emptyset \lor c \neq 0x0 \land E = \{ \delta \} \cup E_{l} \cup E_{r} \]

\[\Rightarrow\] This predicate is expressive enough to prove memory safety & structure preservation.
Inductive predicates are not expressive enough

\[ c := \emptyset \quad \text{or} \quad c \neq 0x0 \quad \text{or} \quad \varepsilon \]

\[ \implies \text{This predicate is expressive enough to prove memory safety & structure preservation.} \]

**Problem** This not enough for partial functional correctness: tree forgets the content!
Inductive predicates are not expressive enough

\[ c = \emptyset \land E = \emptyset \]
\[ c \neq \emptyset \land E = \{ \delta \} \cup E_l \cup E_r \]

\[ \implies \text{This predicate is expressive enough to prove memory safety & structure preservation.} \]

**Problem** This not enough for partial functional correctness: *tree* forgets the content!

[Li et al. SAS, 2015] added **set parameters** expressing the content of data-structures.

**Problem** Set parameters express no constraint in respect to order of appearance!
Sequence parameters

**Our solution**: Express constraints on the sequence of values stored in the tree. Add a *sequence parameter* to the inductive predicate: \( \text{tree}(c, S) \).

The specification of the (partial) functional correctness of \( \text{insert} \) can be expressed as:

\[
\left\{ \begin{align*}
\text{tree}(t, S) \\
S &= \text{sort}(S)
\end{align*} \right\} r = \text{insert}(t, v) \left\{ \begin{align*}
\text{tree}(r, S') \\
\text{where } S' &= \text{sort}(S'[v])
\end{align*} \right\}
\]
Sequence parameters

Our solution: Express constraints on the sequence of values stored in the tree. Add a sequence parameter to the inductive predicate: \texttt{tree}(c, S).

\[
\begin{align*}
S & := \\
\text{c = 0x0} & \land S = [] \\
c \neq 0x0 & \land S = S_l[\delta].S_r
\end{align*}
\]

The specification of the (partial) functional correctness of \texttt{insert} can be expressed as:

\[
\begin{align*}
\text{\texttt{tree}(t, S)} & \land S = \text{\texttt{sort}(S)} \Rightarrow r = \text{\texttt{insert}(t, v)} \\
\text{\texttt{tree}(r, S')} & \land \text{\texttt{sort}(S'[v])} \\
\end{align*}
\]

Requires to extend the shape analysis to derive precise sequence constraints. Requires an abstract domain to reason about (possibly) sorted sequences.
Contributions

An abstract domain reasoning over sequence constraints
To reason on content with order, length constraint, extremal elements, sortedness

A Reduced product between the sequence domain and an existing shape domain
To express constraints over the content of inductive data structures

Evaluation of the analysis in the MemCAD tool
To demonstrate the gain of the expressiveness, the versatility of the approach, and discuss its cost
Sequence domain
Domain description

We build a domain in order to abstract sets of functions from variables to values and sequences of values:

\[
\begin{align*}
\{ & \alpha \mapsto 2 \\ & \delta \mapsto 1 \} \quad \begin{cases} 
S \mapsto 4; 6; 1 \\
S_1 \mapsto 4; 6
\end{cases}
\end{align*}
\]

An abstract value \( \sigma^\#_S \) of the sequence abstract domain \( \mathbb{D}^\#_S \) consists of:

\[
\left( \land S_i = E_i, \sigma^\#_M, \sigma^\#_N \right)
\]

A conjunction of sequence definitions
\[
S = S_1.[\delta] \\
\land S = \text{sort}(S) \\
\land S_1 = \text{sort}(S_1)
\]

An element of a multiset domain \( \mathbb{D}^\#_M \)
\[
mset_S = \{ \delta \} \cup mset_{S_1}
\]

An element of a numerical domain \( \mathbb{D}^\#_N \)
\[
\min_S \leq \delta \leq \max_S \\
\land \max_{S_1} \leq \delta \\
\ldots \\
\land \text{len}_S = 1 + \text{len}_{S_1}
\]
Adding a new constraint

\[
guard_S : \mathbb{D}_S^\# \rightarrow \text{seq. constraint} \rightarrow \mathbb{D}_S^\#
\]

\[
S = S_1.\{\alpha\} \land S = \text{sort}(S) \\
\land S_1 = \text{sort}(S_1)
\]

To assume \(S_r = [\alpha]\), \(guard_S\) follows this algorithm:

\[
\land \text{mset}_S = \{\{\alpha\}\} \cup \text{mset}_{S_1}
\]

\[
\land \text{len}_S = 1 + \text{len}_{S_1} + \text{len}_{S_2}
\]

\[
\land \text{min}_S \leq \alpha \leq \text{max}_S \\
\land \text{min}_S \leq \text{min}_{S_1} \land \text{max}_{S_1} \leq \text{max}_S
\]
Adding a new constraint

\[
\text{guard}_S : \mathbb{D}_S^# \rightarrow \text{seq. constraint} \rightarrow \mathbb{D}_S^#
\]

\[
S = S_1.\lbrack \alpha \rbrack \land S = \text{sort}(S) \\
\land S_1 = \text{sort}(S_1) \\
\land S_r = \lbrack \alpha \rbrack \\
\land \text{mset}_S = \{\alpha\} \cup \text{mset}_{S_1} \\
\land \text{len}_S = 1 + \text{len}_{S_1} + \text{len}_{S_2}
\]

To assume \( S_r = \lbrack \alpha \rbrack \), \text{guard}_S follows this algorithm:

1. add the new definition in the conjunction

\[
\land \text{min}_S \leq \alpha \leq \text{max}_S \\
\land \text{min}_S \leq \text{min}_{S_1} \land \text{max}_{S_1} \leq \text{max}_S
\]
Adding a new constraint

\[ \text{guard}_S : \mathbb{D}_S^\# \rightarrow \text{seq. constraint} \rightarrow \mathbb{D}_S^\# \]

\[
\begin{align*}
S &= S_1.\{\alpha\} \land S = \text{sort}(S) \\
\land S_1 &= \text{sort}(S_1) \\
\land S_r &= \{\alpha\} \\
\land \text{mset}_S &= \{\alpha\} \cup \text{mset}_{S_1} \\
\land \text{mset}_{S_r} &= \{\alpha\} \\
\land \text{len}_S &= 1 + \text{len}_{S_1} + \text{len}_{S_2} \\
\land \text{len}_{S_r} &= 1 \\
\land \text{min}_S \leq \alpha \leq \text{max}_S \\
\land \text{min}_S \leq \text{min}_{S_1} \land \text{max}_{S_1} \leq \text{max}_S \\
\land \text{min}_{S_r} &= \alpha = \text{max}_{S_r}
\end{align*}
\]

To assume \( S_r = [\alpha] \), \( \text{guard}_S \) follows this algorithm:

1. add the new definition in the conjunction
2. add content/length/bounds constraints
Adding a new constraint

\[ \text{guard}_S : \mathbb{D}_S^\# \rightarrow \text{seq. constraint} \rightarrow \mathbb{D}_S^\# \]

\[
S = S_1.S_r \land S = \text{sort}(S) \\
\land S_1 = \text{sort}(S_1) \\
\land S_r = [\alpha] \\
\land \text{mset}_S = \{\alpha\} \cup \text{mset}_{S_1} \\
\land \text{mset}_{S_r} = \{\alpha\} \\
\land \text{len}_S = 1 + \text{len}_{S_1} + \text{len}_{S_2} \\
\land \text{len}_{S_r} = 1
\]

To assume \(S_r = [\alpha]\), \text{guard}_S \) follows this algorithm:

1. add the new definition in the conjunction
2. add content/length/bounds constraints
3. fold other definitions

\[
\land \text{min}_S \leq \alpha \leq \text{max}_S \\
\land \text{min}_S \leq \text{min}_{S_1} \land \text{max}_{S_1} \leq \text{max}_S \\
\land \text{min}_{S_r} = \alpha = \text{max}_{S_r}
\]
Adding a new constraint

\[ \text{guard}_S : \mathbb{D}_S^\# \rightarrow \text{seq. constraint} \rightarrow \mathbb{D}_S^\# \]

\[ S = S_1.S_r \land S = \text{sort}(S) \]
\[ \land S_1 = \text{sort}(S_1) \]
\[ \land S_r = [\alpha] \land S_r = \text{sort}(S_r) \]
\[ \land \text{mset}_S = \{ \alpha \} \cup \text{mset}_{S_1} \]
\[ \land \text{mset}_{S_r} = \{ \alpha \} \]
\[ \land \text{len}_S = 1 + \text{len}_{S_1} + \text{len}_{S_2} \]
\[ \land \text{len}_{S_r} = 1 \]
\[ \land \min_S \leq \alpha \leq \max_S \]
\[ \land \min_S \leq \min_{S_1} \land \max_{S_1} \leq \max_S \]
\[ \land \min_{S_r} = \alpha = \max_{S_r} \]

To assume \( S_r = [\alpha] \), \text{guard}_S \) follows this algorithm:

1. add the new definition in the conjunction
2. add content/length/bounds constraints
3. fold other definitions
4. Saturate constraints

\[ S = S_1....S_n \]
\[ \forall i, S_i = \text{sort}(S_i) \quad \forall i < j, \max_{S_i} \leq \min_{S_j} \]
\[ S = \text{sort}(S) \]
Adding a new constraint

**guard**$_S$: $D_S^\# \rightarrow$ seq. constraint $\rightarrow D_S^\#

\begin{align*}
    S &= S_1.S_r \land S = \text{sort}(S) \\
    \land S_1 &= \text{sort}(S_1) \\
    \land S_r &= [\alpha] \land S_r = \text{sort}(S_r) \\
    \land \text{mset}_S &= \{\alpha\} \cup \text{mset}_{S_1} \\
    \land \text{mset}_{S_r} &= \{\alpha\} \\
    \land \text{len}_S &= 1 + \text{len}_{S_1} + \text{len}_{S_2} \\
    \land \text{len}_{S_r} &= 1 \\
    \land \min_S &\leq \alpha \leq \max_S \\
    \land \min_S &\leq \min_{S_1} \land \max_{S_1} \leq \max_S \\
    \land \min_{S_r} &= \alpha = \max_{S_r}
\end{align*}

To assume $S_r = [\alpha]$, **guard**$_S$ follows this algorithm:

1. add the new definition in the conjunction
2. add content/length/bounds constraints
3. fold other definitions
4. Saturate constraints
   $$S = S_1....S_n$$
   $$\forall i, S_i = \text{sort}(S_i) \land \forall i < j, \max_{S_i} \leq \min_{S_j}$$
   $$S = \text{sort}(S)$$
5. detect & remove cyclic constraints
Adding a new constraint

\text{guard}_S : \mathcal{D}_S^\# \rightarrow \text{seq. constraint} \rightarrow \mathcal{D}_S^\#

\begin{align*}
S &= S_1.S_r \land S = \text{sort}(S) \\
\land S_1 &= \text{sort}(S_1) \\
\land S_r &= [\alpha] \land S_r = \text{sort}(S_r) \\
\land \text{mset}_S &= \{\alpha\} \cup \text{mset}_{S_1} \\
\land \text{mset}_{S_r} &= \{\alpha\} \\
\land \text{len}_S &= 1 + \text{len}_{S_1} + \text{len}_{S_2} \\
\land \text{len}_{S_r} &= 1 \\
\land \text{min}_S &\leq \alpha \leq \text{max}_S \\
\land \text{min}_S &\leq \text{min}_{S_1} \land \text{max}_{S_1} \leq \text{max}_S \\
\land \text{min}_{S_r} &= \alpha = \text{max}_{S_r}
\end{align*}

To assume \( S_r = [\alpha] \), \text{guard}_S follows this algorithm:

1. add the new definition in the conjunction
2. add content/length/bounds constraints
3. fold other definitions
4. Saturate constraints
5. detect & remove cyclic constraints

\begin{align*}
&\forall i, S_i = \text{sort}(S_i) \quad \forall i < j, \text{max}_{S_i} \leq \text{min}_{S_j} \\
&S = S_1....S_n \\
&S = \text{sort}(S)
\end{align*}

\textbf{Theorem: Soundness of} \text{guard}_S

\[ \gamma_s(\text{guard}_s(\sigma_s^\#, S = E)) \] contains all valuations in \( \gamma_s(\sigma_s^\#) \) satisfying \( S = E \).
Abstract lattice operators

- **verify**_

\[ \text{verify}_S : \mathbb{D}^\#_S \rightarrow \text{seq constraint} \rightarrow \{\text{true, false}\} \]

\[ \text{verify}_S(\sigma^\#_S, S = E) \] conservatively checks if \( \sigma^\#_S \) satisfies \( S = E \).

- \( \sqsubseteq_S : \mathbb{D}^\#_S \rightarrow \mathbb{D}^\#_S \rightarrow \{\text{true, false}\} \)

Abstract inclusion checking, using **verify**_

- \( \sqcup_S : \mathbb{D}^\#_S \rightarrow \mathbb{D}^\#_S \rightarrow \mathbb{D}^\#_S \)

That tries to infer common definitions from both inputs.

**Example**

\[ \left( \begin{array}{c}
S = S_1.S_2 \\
\land S_3 = []
\end{array} \right) \sqcup_S \left( \begin{array}{c}
S = S_2.S_3 \\
\land S_1 = []
\end{array} \right) = (S = S_1.S_2.S_3) \]

- \( \triangledown_S : \mathbb{D}^\#_S \rightarrow \mathbb{D}^\#_S \rightarrow \mathbb{D}^\#_S \)

That selects the constraints in the left arguments verified in the right one.
Shape analysis with sequence predicates
The tree(c) predicate only synthesizes full binary trees. To abstract partial trees, the shape domain uses a segment predicate treeseg(l, c).

The shape domain automatically derives treeseg from tree.

The analysis must keep tracks of the content stored in the segment.
Integrating sequence parameters in the shape domain

The \texttt{tree}(c) predicate only synthesizes full binary trees. To abstract partial trees, the shape domain uses a \textit{segment predicate} \texttt{treeseg}(l, c).

The shape domain automatically derives \texttt{treeseg} from \texttt{tree}. The analysis must keep tracks of the content stored in the segment.

In order to reason precisely over inductive predicates, the shape analysis relies on:

- \textbf{Unfold}: refines the memory by materializing synthesized memory.
- \textbf{Fold}: extrapolates the memory state to gain generality. Used to over-approximate two memory states

For each of these operations, the shape domain should \textit{derive the corresponding sequence constraints to assume or verify}.
Adding sequence parameters to segment predicates

The sequence stored in the tree is: 0 1 2 3 4 5 6 9 10 11 12

The analysis needs to recall the location of the missing sequence in treeseg.

⇒ the segment predicate has two sequence parameters: $S_1, S_2$

One for each side of the missing sequence
To analyze \( \texttt{if}(1)\{v = l->\text{data}\} \) with initial state \( \texttt{tree}(l, S) \).
Refining abstract memory state with unfolding

To analyze $\text{if}(l)\{v = l->\text{data}\}$ with initial state $\text{tree}(l, S)$

1. The numerical constraint $l \neq 0x0$ is guarded in the numerical part of the sequence domain.
Refining abstract memory state with unfolding

To analyze \( \text{if}(\text{l})\{v= \text{l->data}\} \) with initial state \( \text{tree}(\text{l}, S) \)

1. The numerical constraint \( \text{l} \neq 0\times0 \) is guarded in the numerical part of the sequence domain.
2. To materialize \( \text{l->data} \), the analysis **unfolds the predicate**

   The abstract memory is replaced by the definition: \( \delta, S_l, S_r \) are fresh variables
   The numerical and sequences constraints are guarded in the sequence domain

\[
\delta \neq 0\times0
\land \delta = 0\times0
\land S = []
\]

\[
S_l
\]

\[
S_r
\]
Refining abstract memory state with unfolding

To analyze $\text{if}(l)\{v = l->\text{data}\}$ with initial state $\text{tree}(l, S)$

1. The numerical constraint $l \neq 0x0$ is guarded in the numerical part of the sequence domain.

2. To materialize $l->\text{data}$, the analysis **unfolds the predicate**

   The abstract memory is replaced by the definition: $\delta, S_l, S_r$ are fresh variables

   The numerical and sequences constraints are guarded in the sequence domain

   - **The empty case:** Inconsistent with the $\text{if}$ assumption $\implies$ Discarded

\[1 \neq 0x0 \land l = 0x0 \land S = []\]
Refining abstract memory state with unfolding

To analyze $\text{if}(l)\{v = l->\text{data}\}$ with initial state $\text{tree}(l, S)$

1. The numerical constraint $l \neq 0x0$ is guarded in the numerical part of the sequence domain.
2. To materialize $l->\text{data}$, the analysis unfolds the predicate 

   The abstract memory is replaced by the definition: $\delta, S_l, S_r$ are fresh variables

   The numerical and sequences constraints are guarded in the sequence domain
   - The empty case: Inconsistent with the if assumption $\implies$ Discarded
   - The non-empty case: $c->\text{data}$ corresponds to $\delta$.

\[ l \neq 0x0 \land l = 0x0 \land S = [] \]
Refining abstract memory state with unfolding

To analyze $\text{if}(l)\{v = l->\text{data}\}$ with initial state $\text{tree}(l, S)$

1. The numerical constraint $l \neq 0x0$ is guarded in the numerical part of the sequence domain.
2. To materialize $l->\text{data}$, the analysis **unfolds the predicate**
   
   The abstract memory is replaced by the definition: $\delta, S_l, S_r$ are fresh variables
   
   The numerical and sequences constraints are guarded in the sequence domain
   
   • **The empty case**: Inconsistent with the $\text{if}$ assumption $\implies$ Discarded
   
   • **The non-empty case**: $c->\text{data}$ corresponds to $\delta$.

3. The assignment $v \leftarrow \delta$ is performed.

**Theorem: Soundness of unfolding**

The resulting disjunction of abstract states over approximates the original state.
Fold generalizes the abstract state by rewriting parts of the memory into a predicate. The analysis first checks that some constraints hold in the sequence domain.

**Folding an inductive predicate**

\[
\text{verify}_{S}(\sigma^{\#}_{S}, S = S_{l}.[\delta].S_{r}) = \text{true}
\]

**Folding segment and predicates**

\[
\text{verify}_{S}(\sigma^{\#}_{S}, S = S_{1}.S_{0}.S_{2}) = \text{true}
\]

**Theorem: Soundness of folding**

The folded abstract state over-approximates the original one.
Lattice operators

Inclusion checking
Folds the left input until both are syntactically equal.

Upper bound
Folds both inputs until they are syntactically equal.

Widening

With these operators, we design a sound and automatic static analysis by forward abstract interpretation. And the analysis checks that the final state satisfies to post-condition to prove partial functional correctness.
Proving the insertion into a BST

After two iterations, the analysis inferred the following loop invariant:

\[
S = S_1.S_0.S_2 \\
\land S = \text{sort}(S) \\
\land S_i = \text{sort}(S_i), \ i = 0, 1, 2 \\
\land l, c \neq 0x0 \\
\land \max_{S_1} \leq v < \min_{S_2}
\]

Finally, the analysis was able to prove that the final state satisfies the post condition:

\[
S_r = \text{sort}(S.[v])
\]
Experiments
Experimental Setup

The analysis described has been implemented in the MemCAD static analyzer available at gitlab.inria.fr/memcad/memcad

For each test, we specify:

- the full inductive predicates,
- the pre- and post-conditions,

Everything else (segment predicates/loop invariants) is inferred by the analysis.

(Q1) Is this analysis precise enough to prove memory safety (Safe) and functional properties (Fc)?

(Q2) How significant is the overhead of the combined analysis compared to the baseline?

(Q3) Can this analysis successfully verify real-world C libraries?
## Experiment 1: Classical list & BST programs

<table>
<thead>
<tr>
<th>Example</th>
<th>wo/ seq</th>
<th>with seq parameters</th>
<th>Safe verified</th>
<th>time</th>
<th>Fc</th>
<th>overhd. % num verified</th>
</tr>
</thead>
<tbody>
<tr>
<td>Singly linked list</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>concat</td>
<td>Safe</td>
<td></td>
<td>Safe</td>
<td>2.4x</td>
<td>21.7%</td>
<td>Fc</td>
</tr>
<tr>
<td>deep copy</td>
<td>Safe</td>
<td></td>
<td>Safe</td>
<td>1.7x</td>
<td>18.1%</td>
<td>Fc</td>
</tr>
<tr>
<td>length</td>
<td>Safe</td>
<td></td>
<td>Safe</td>
<td>4.7x</td>
<td>50.0%</td>
<td>Fc</td>
</tr>
<tr>
<td>insert at position</td>
<td>Safe</td>
<td></td>
<td>Safe</td>
<td>5.4x</td>
<td>60.2%</td>
<td>Fc</td>
</tr>
<tr>
<td>sorted insertion</td>
<td>Safe</td>
<td></td>
<td>Safe</td>
<td>6.1x</td>
<td>47.3%</td>
<td>Fc</td>
</tr>
<tr>
<td>minimum</td>
<td>Safe</td>
<td></td>
<td>Safe</td>
<td>7.8x</td>
<td>45.9%</td>
<td>Fc</td>
</tr>
<tr>
<td>insertion sort</td>
<td>Safe</td>
<td></td>
<td>Safe</td>
<td>29.0x</td>
<td>46.0%</td>
<td>Fc</td>
</tr>
<tr>
<td>bubble sort</td>
<td>Safe</td>
<td></td>
<td>Safe</td>
<td>19.1x</td>
<td>51.5%</td>
<td>Fc</td>
</tr>
<tr>
<td>merge sorted lists</td>
<td>Safe</td>
<td></td>
<td>Safe</td>
<td>9.6x</td>
<td>51.4%</td>
<td>Fc</td>
</tr>
<tr>
<td>Binary search trees</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Insertion</td>
<td>Safe</td>
<td></td>
<td>Safe</td>
<td>6.0x</td>
<td>38.6%</td>
<td>Fc</td>
</tr>
<tr>
<td>Delete max</td>
<td>Safe</td>
<td></td>
<td>Safe</td>
<td>6.2x</td>
<td>48.6%</td>
<td>Fc</td>
</tr>
<tr>
<td>Search (present)</td>
<td>Safe</td>
<td></td>
<td>Safe</td>
<td>4x</td>
<td>45.3%</td>
<td>Fc</td>
</tr>
<tr>
<td>BST to list</td>
<td>Safe</td>
<td></td>
<td>Safe</td>
<td>3.2x</td>
<td>38.2%</td>
<td>Fc</td>
</tr>
<tr>
<td>list to BST</td>
<td>Safe</td>
<td></td>
<td>Safe</td>
<td>11.9x</td>
<td>46.1%</td>
<td>Fc</td>
</tr>
</tbody>
</table>

### Expressiveness
- Prove Fc for complex programs including 3 sorting algorithms
- Sequences improve precision for Safe!

### Overhead
- High slowdown for complex programs
  - Up to 30x for insertion sort
- Most of it in the numerical domain
  - Quadratic cost of sortedness checking
  - Length constraints are expensive
- Sequence domain slows down convergence
  - Needs one more iteration for $\nabla_s$ to stabilize.
Experiment 2: Real-world libraries

We tested MemCAD on real-world list libraries implementing various features:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Linux</th>
<th>FreeRTOS</th>
<th>GDSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circular DLL with distinguished header</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Extreme sentinel nodes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Intrusive</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Pointer to header</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Length in header</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Sorted</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Feature</th>
<th>Linux</th>
<th>FreeRTOS</th>
<th>GDSL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>wo/ seq</td>
<td>w/ seq</td>
<td>wo/ seq</td>
</tr>
<tr>
<td>Safe</td>
<td>4/4✓</td>
<td>4/4✓</td>
<td>4/4✓</td>
</tr>
<tr>
<td>Fc</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

†: Cannot prove Safe for extraction at position.
‡: Cannot prove Fc for min/max extraction.
Conclusion
Conclusion

How to improve the expressiveness of static analysis over dynamic data-structures to prove partial functional correctness?

- Design of a novel sequence abstract domain
  It leverages existing numerical/set domains to express length/bounds/content constraints.

- Integration into a separation logic based shape analysis
  The reduced product derives corresponding sequence constraints for unfolding/weakening.

- Implementation in the MemCAD static analyzer
  Proves partial functional correctness for complex algorithms on SLL/BST and real world libraries.
Thank you!
Could we relax the sortedness checking?

**Lemma**

If \( S = S_1 \ldots S_n \), then

\[
S = \text{sort}(S) \iff \forall i, S_i = \text{sort}(S_i) \land \forall i < j, \text{max}_{S_i} \leq \text{min}_{S_j}
\]

**Question** The number of constraints in the right-hand side is quadratic! Could we relax it for \( j := i + 1 \)?

\[\Rightarrow \text{NO! Because of the empty sequence case!}\]

By consistency of the concretization: \( \nu_s(S) = \varepsilon \iff \begin{cases} \max_S = -\infty \\ \min_S = +\infty \end{cases} \)

Consider \( \nu_s = \begin{cases} S \mapsto 3 \ 1 \\ S_1 \mapsto 3 \\ S_2 \mapsto \varepsilon \\ S_3 \mapsto 1 \end{cases} \)

We have indeed:

\[
\begin{align*}
\nu_s \models S &= S_1 . S_2 . S_3 \\
\nu_s \models S_i &= \text{sort}(S_i), \ \forall i \\
\nu_s \models \max_{S_1} \leq \min_{S_2} \\
\nu_s \models \max_{S_2} \leq \min_{S_3}
\end{align*}
\]

But:

\( \nu_s \not\models S = \text{sort}(S) \)
Removing cyclic constraints

Assume the abstract state $\sigma^{\#}_s$ contains the following constraints:

\[
S = S_1.S'.S_2 \\
\land S' = S_3.S'' \\
\land S'' = S.S_4
\]

If we inline definitions over $S'$ and $S''$ into the definition of $S$ we obtain:

\[
\]

The constraints over $S, S', S''$ are replaced by

\[
\begin{cases} 
S_1 = S_2 = S_3 = S_4 = [] \\
S = S' = S''
\end{cases}
\]

If one constraint contains at least one atom $[\alpha]$, then the state is reduced to $\bot_s$.

$S = \text{sort}(S)$ does not count as a cyclic constraint as the implementation of the abstract domain does not represent it as such.
Concatenating inductive predicates

**seg-full case**

\[ \text{verify}_s(\sigma^#, S = S_1.S_0.S_2) = \text{true} \]

**seg-seg case**

\[ \text{verify}_s(\sigma^#, S_1 = S'_1.S''_1) = \text{true} \]
\[ \text{verify}_s(\sigma^#, S_2 = S''_2.S'_2) = \text{true} \]
Segment tree predicate

\[\emptyset \quad \text{c} = l \land S_1 = [] \land S_2 = []\]

\[\neg c \neq c \land l \neq 0x0 \land S_1 = S_{l,1} \land S_2 = S_{l,2} \cdot [\delta] \cdot S_r\]

\[\neg c \neq c \land l \neq 0x0 \land S_1 = S_{l,1} \cdot [\delta] \cdot S_{r,1} \land S_2 = S_{r,2}\]
Hypothesis to derive segment from full predicate

Hypothesis
- The constraint over sequence parameter is **only concatenation based**
- The argument of each recursive call occurs **exactly once** in the constraint

\[ S := W \ldots S_i \ldots S_j = E.S_i.E' \]

\[ S_l \sqcup S_r = E.S_i.E'\{S_i \leftarrow S_{i,l} \sqcup S_{i,r}\} \]

\[ = E.S_{i,l} \sqcup S_{i,r}.E' \]

\[ \Rightarrow \begin{cases} S_l = E.S_{i,l} \\ S_r = S_{i,r}.E' \end{cases} \]
Exemple: insertion in binary search tree

\[
\{ \text{tree}(t, S) \land S = \text{sort}(S) \}\]

- \textbf{if}( t == NULL ){ // Empty case }
- \textbf{else}{
  ptree c = t;
  \textbf{while}(v < c->d \&\& c->l \mid\mid \\
    v >= c->d \&\& c->r )
  \textbf{if}(v < c->d ){
    c = c->l
  } \textbf{else }{
    c = c->r;
  }
  \textbf{if}( v< c->d ){
    c->l = m;
  } \textbf{else}{
    c->r = m;
  }
  \textbf{return } t;
}

\{ \text{tree}(t, \text{sort}(S.[v])))\}
Exemple: insertion in binary search tree

\[
\begin{align*}
\{ \text{tree}(t, S) \} & \quad \land \quad S = \text{sort}(S) \\
\text{if}( t == \text{NULL} ) \{ \\
\quad \text{// Empty case} \\
\} \text{else} \{ \\
\quad \text{ptree } c = t; \\
\quad \text{while}( v < c->d \text{ \&\& } c->l \| \| \\
\quad \quad v >= c->d \text{ \&\& } c->r ) \\
\quad \text{if}( v < c->d ) \{ \\
\quad \quad \text{c = c}->l \\
\quad \} \text{else} \{ \\
\quad \quad \text{c = c}->r; \\
\quad \} \\
\quad \text{if}( v < c->d )\{ \\
\quad \quad c->l = m; \\
\quad \} \text{else} \{ \\
\quad \quad c->r = m; \\
\quad \} \\
\quad \text{return } t; \\
\} \\
\{ \text{tree}(t, \text{sort}(S.[v]))) \}
\end{align*}
\]

\[
S = \text{sort}(S) \\
\land \alpha \neq 0x0
\]
Exemple: insertion in binary search tree

```
{tree(t, S) ∧ S = sort(S)}

if (t == NULL) {
    // Empty case
} else {
    ptree c = t;
    while(v < c->d && c->l || v >= c->d && c->r)
        if (v < c->d) {
            c = c->l;
        } else {
            c = c->r;
        }
    if (v < c->d) {
        c->l = m;
    } else{
        c->r = m;
    }
    return t;
}

{tree(t, sort(S.[v]))}
```
Exemple: insertion in binary search tree

\[
\{ \text{tree}(t, S) \land S = \text{sort}(S) \}\]

if (t == NULL ){
  // Empty case
} else{
  ptree c = t;
  while( (v < c->d && c->l) || (v >= c->d && c->r ) )
    if(v < c->d) {
      c = c->l
    } else {
      c = c->r;
    }
  if( v < c->d ){c->l = m;} else{c->r = m;}
  return t;
}

\[
\{ \text{tree}(t, \text{sort}(S.\[v\])))\} \]
Exemple: insertion in binary search tree

\[
\{ \text{tree}(t, S) \land S = \text{sort}(S) \}\]

if( t== NULL ){
   // Empty case
}else{
   ptree c = t;
   while(v< c->d && c->l ||
      v>= c->d && c->r )
      if(v < c->d) {
         c = c->l
      }else {
         c = c->r;
      }
   if( v< c->d ){
      c->l = m;
   }else{
      c->r = m;
   }
   return t;
}

\{ \text{tree}(t, \text{sort}(S.[v])))\}
Exemple: insertion in binary search tree

\[
\{ \text{tree}(t, S) \land S = \text{sort}(S) \}\n\]

if( t == NULL ){
    // Empty case
}else{
    ptree c = t;
    while(v < c->d && c->l ||
        v >= c->d && c->r )
        if(v < c->d ) {
            c = c->l;
        }else {
            c = c->r;
        }
    if(v < c->d ){
        c->l = m;
    }else{
        c->r = m;
    }
    return t;
}

\[
\{ \text{tree}(t, \text{sort}(S.[v]))) \}\]
Exemple: insertion in binary search tree

\begin{align*}
\{ & \text{tree}(t, S) \land S = \text{sort}(S) \} \\
& \quad \text{if( } t == \text{ NULL } \{ \\
& \quad \quad \text{// Empty case} \\
& \quad \quad \} \text{else} \\
& \quad \quad \text{ptree } c = t; \\
& \quad \quad \text{while}(v < c->d && c->l || \\
& \quad \quad \quad v >= c->d && c->r ) \\
& \quad \quad \quad \text{if}(v < c->d) \{ \\
& \quad \quad \quad \quad c = c->l; \\
& \quad \quad \quad \} \text{else} \{ \\
& \quad \quad \quad \quad c = c->r; \\
& \quad \quad \} \quad \text{if}( v < c->d )\{ \\
& \quad \quad \quad c->l = m; \\
& \quad \quad \} \text{else} \{ \\
& \quad \quad \quad c->r = m; \\
& \quad \quad \} \quad \text{return } t; \\
& \quad \} \\
\{ & \text{tree}(t, \text{sort}(S.[v])) \}\end{align*}

\begin{align*}
S & = \text{sort}(S) \land S = S_l.[\delta].S_r \\
S_i & = \text{sort}(S_i) \quad i \in \{l, r\} \\
\land & \alpha \neq 0x0 \land v < \delta \land \alpha_l \neq 0x0 \\
\text{max}_{S_l} & \leq \delta \leq \text{min}_{S_r}
\end{align*}
Union (Shape part)
Union (Shape part)

\[ t \xrightarrow{\alpha} S \]

\[ t \xrightarrow{\alpha} \delta \xrightarrow{\alpha_r} S_r \]

\[ t \xrightarrow{\alpha} \delta \xrightarrow{\alpha_l} S_l \]

\[ t \xrightarrow{\alpha} \delta \xrightarrow{\alpha_l} S_l, S_r \]

\[ t \xrightarrow{\alpha} S_0 \xrightarrow{\alpha} S_l, S_r \]
Union (Shape part)

\[ \alpha \leftrightarrow \alpha_c \rightarrow \alpha_l, \alpha_r \]
\[ S \leftrightarrow S_0 \rightarrow S_l, S_r \]
\[ \alpha \leftrightarrow \alpha_t \rightarrow \alpha, \alpha \]
\[ [] \leftrightarrow S_1 \rightarrow ??, ?? \]
\[ [] \leftrightarrow S_2 \rightarrow ??, ?? \]
To verify:

\[ S_{l,1} = \begin{array} \delta \\ S_{l,2} = \begin{array} \delta \end{array} \end{array} \]

\[ \alpha_{l} = \alpha_{c} \quad S_{1} = S_{l,1} \quad S_{2} = S_{l,2} \]

\[ \delta.S_{r} \alpha_{t} \neq 0 \]

The constraints are simplified.

The numerical ones are checked with verify.

The sequence ones are used for definition of \( S_{1} \) and \( S_{2} \).
Weakening: inclusion test

To verify:
\[ S_{l,1} = [] \]
\[ S_{l,2} = [] \]
\[ \alpha_l = \alpha_c \]

The constraints are simplified.
\[ \alpha \neq 0 \]
\[ \alpha_l = \alpha_l \]
\[ S_1 = [] \]
\[ S_2 = [\delta].S_r \]

The numerical ones are checked with \texttt{verify}_S.
The sequence ones are used for definition of \( S_1 \) and \( S_2 \).
Weakening: inclusion test

To verify:
\[ S_{l,1} = [] \]
\[ S_{l,2} = [] \]
\[ \alpha_l = \alpha_c \]
\[ S_1 = S_{l,1} \]
\[ S_2 = S_{l,2} \cdot [\delta] \cdot S_r \]
\[ \alpha_t \neq 0 \]

The constraints are simplified.
\[ \alpha \neq 0 \]
\[ \alpha_l = \alpha_l \]
\[ S_1 = [] \]
\[ S_2 = [\delta] \cdot S_r \]

The numerical ones are checked with \texttt{verify}_s.
The sequence ones are used for definition of \( S_1 \) and \( S_2 \).
Union (Shape part)

\[ t \subseteq T \]

\[ \alpha \leftarrow \alpha_c \rightarrow \alpha_l, \alpha_r \]

\[ S \leftarrow S_0 \rightarrow S_l, S_r \]

\[ \alpha \leftarrow \alpha_t \rightarrow \alpha \]

\[ \emptyset \leftarrow S_1 \rightarrow \emptyset, S_l, [\delta] \]

\[ \emptyset \leftarrow S_2 \rightarrow [\delta].S_r, \emptyset \]
Union (Numerical part)

\[
\begin{align*}
\alpha & \leftarrow \alpha_c \mapsto \alpha_l, \alpha_r \\
S & \leftarrow S_0 \mapsto S_{l}, S_{r} \\
\alpha & \leftarrow \alpha_t \mapsto \alpha, \alpha \\
[] & \leftarrow S_1 \mapsto [], S_{l}.[\delta] \\
[] & \leftarrow S_2 \mapsto [\delta].S_{l}, []
\end{align*}
\]

Result:

\[
\begin{align*}
S &= \text{sort}(S) \\
\land \alpha & \neq 0x0
\end{align*}
\]

\[
\begin{align*}
S &= \text{sort}(S) \land S = S_{l}.[\delta].S_{r} \\
S_{i} &= \text{sort}(S_{i}) \quad i \in \{l, r\} \\
\land \alpha & \neq 0x0 \\
\land \alpha_{l} & \neq 0x0 \\
\land v & < \delta \\
\text{max}_{S_{l}} & \leq \delta \leq \text{min}_{S_{r}}
\end{align*}
\]
Union (Numerical part)

\[ \alpha \leftarrow \alpha_c \leftarrow \alpha_l, \alpha_r \]
\[ S \leftarrow S_0 \leftarrow S_l, S_r \]
\[ \alpha \leftarrow \alpha_t \leftarrow \alpha \]
\[ \emptyset \leftarrow S_1 \leftarrow \emptyset \]
\[ \emptyset \leftarrow S_2 \leftarrow [\delta].S_l, \emptyset \]

Result:

\[ S = \text{sort}(S) \]
\[ S_1 = S_2 = \emptyset \]
\[ \land \alpha = \alpha_c = \alpha_t \neq 0x0 \]

\[ S = \text{sort}(S) \land S = S_0.S_2 \]
\[ S_i = \text{sort}(S_i) \quad i \in \{l, r, 1, 2\} \]
\[ S_0 = S_l \land S_1 = \emptyset \land S_2 = [\delta].S_r \]
\[ \land \alpha = \alpha_t \neq 0x0 \]
\[ \land \alpha_l = \alpha_c \neq 0x0 \]
\[ \land \forall \leq \delta = \min S_2 \land \max S_1 = -\infty \]
\[ \max S_l \leq \delta \leq \min S_r \]

\[ S = \text{sort}(S) \land S = S_1.S_0 \]
\[ S_i = \text{sort}(S_i) \quad i \in \{l, r, 1, 2\} \]
\[ S_0 = S_r \land S_1 = S_l.[\delta] \land S_2 = \emptyset \]
\[ \land \alpha = \alpha_t \neq 0x0 \]
\[ \land \alpha_r = \alpha_c \neq 0x0 \]
\[ \land \forall \geq \delta = \max S_1 \land \min S_2 = +\infty \]
\[ \max S_l \leq \delta \leq \min S_r \]
Union (Numerical part)

\[
\begin{align*}
\alpha & \leftarrow \alpha_c \leftrightarrow \alpha_l, \alpha_r \\
S & \leftarrow S_0 \leftrightarrow S_l, S_r \\
\alpha & \leftarrow \alpha_t \leftrightarrow \alpha \\
[] & \leftarrow S_1 \leftrightarrow [] \\
[] & \leftarrow S_2 \leftrightarrow [\delta].S_l, []
\end{align*}
\]

Result:

\[S_i = \text{sort}(S_i) \quad i \in \{\_, 0, 1, 2\}\]

\[
\begin{align*}
S & = \text{sort}(S) \\
S_1 & = S_2 = [] \\
\wedge \alpha = \alpha_c = \alpha_t & \neq 0x0
\end{align*}
\]

\[
\begin{align*}
S & = \text{sort}(S) \wedge S = S_0.S_2 \\
S_i & = \text{sort}(S_i) \quad i \in \{l, r, 1, 2\} \\
S_0 & = S_l \wedge S_1 = [] \wedge S_2 = [\delta].S_r \\
\wedge \alpha = \alpha_t & \neq 0x0 \\
\wedge \alpha_l & = \alpha_c \neq 0x0 \\
\wedge v & < \delta = \min_{S_2} \wedge \max_{S_1} = -\infty \\
\max_{S_l} & \leq \delta \leq \min_{S_r}
\end{align*}
\]

\[
\begin{align*}
S & = \text{sort}(S) \wedge S = S_1.S_0 \\
S_i & = \text{sort}(S_i) \quad i \in \{l, r, 1, 2\} \\
S_0 & = S_r \wedge S_1 = S_l.[\delta] \wedge S_2 = [] \\
\wedge \alpha = \alpha_t & \neq 0x0 \\
\wedge \alpha_r = \alpha_c & \neq 0x0 \\
\wedge v & \geq \delta = \max_{S_1} \wedge \min_{S_2} = +\infty \\
\max_{S_l} & \leq \delta \leq \min_{S_r}
\end{align*}
\]
Union (Numerical part)

\[
\begin{align*}
\alpha & \leftarrow \alpha_c \leftarrow \alpha_l , \alpha_r \\
S & \leftarrow S_0 \leftarrow S_l , S_r \\
\alpha & \leftarrow \alpha_t \leftarrow \alpha \\
[] & \leftarrow S_1 \leftarrow [] , S_l . [\delta] \\
[] & \leftarrow S_2 \leftarrow [\delta] . S_l , []
\end{align*}
\]

Result:

\[
\begin{align*}
S_i & = \text{sort}(S_i) \quad i \in \{_, 0, 1, 2\} \\
S & = S_1 . S_0 . S_2
\end{align*}
\]

\[
\begin{align*}
S & = \text{sort}(S) \\
S_1 & = S_2 = [] \\
\land \alpha & = \alpha_c = \alpha_t \neq 0x0
\end{align*}
\]

\[
\begin{align*}
S & = \text{sort}(S) \land S = S_0 . S_2 \\
S_i & = \text{sort}(S_i) \quad i \in \{l, r, 1, 2\} \\
S_0 & = S_l \land S_1 = [] \land S_2 = [\delta] . S_r \\
\land \alpha & = \alpha_t \neq 0x0 \\
\land \alpha_l & = \alpha_c \neq 0x0 \\
\land v & < \delta = \min S_2 \land \max S_1 = -\infty \\
\max S_l & \leq \delta \leq \min S_r
\end{align*}
\]

\[
\begin{align*}
S & = \text{sort}(S) \land S = S_1 . S_0 \\
S_i & = \text{sort}(S_i) \quad i \in \{l, r, 1, 2\} \\
S_0 & = S_r \land S_1 = S_l . [\delta] \land S_2 = [] \\
\land \alpha & = \alpha_t \neq 0x0 \\
\land \alpha_r & = \alpha_c \neq 0x0 \\
\land v & \geq \delta = \max S_1 \land \min S_2 = +\infty \\
\max S_l & \leq \delta \leq \min S_r
\end{align*}
\]
Union (Numerical part)

\[ \alpha \leftarrow \alpha_c \rightarrow \alpha_l, \alpha_r \]
\[ S \leftarrow S_0 \rightarrow S_l, S_r \]
\[ \alpha \leftarrow \alpha_t \rightarrow \alpha \]
\[ [] \leftarrow S_1 \rightarrow [], S_l.\left[ \delta \right] \]
\[ [] \leftarrow S_2 \rightarrow [\delta].S_l, [] \]

Result:

\[ S_i = \text{sort}(S_i) \quad i \in \{\_0, 1, 2\} \]
\[ S = S_1.S_0.S_2 \]
\[ \alpha_c, \alpha_t \neq 0x0 \]

\[ S = \text{sort}(S) \land S = S_0.S_2 \]
\[ S_i = \text{sort}(S_i) \quad i \in \{l, r, 1, 2\} \]
\[ S_0 = S_l \land S_1 = [] \land S_2 = [\delta].S_r \]
\[ \land \alpha = \alpha_t \neq 0x0 \]
\[ \land \alpha_l = \alpha_c \neq 0x0 \]
\[ \land v < \delta = \min_{S_2} \land \max_{S_1} = -\infty \]
\[ \max_{S_l} \leq \delta \leq \min_{S_r} \]

\[ S = \text{sort}(S) \land S = S_1.S_0 \]
\[ S_i = \text{sort}(S_i) \quad i \in \{l, r, 1, 2\} \]
\[ S_0 = S_r \land S_1 = S_l.\left[ \delta \right] \land S_2 = [] \]
\[ \land \alpha = \alpha_t \neq 0x0 \]
\[ \land \alpha_r = \alpha_c \neq 0x0 \]
\[ \land v \geq \delta = \max_{S_1} \land \min_{S_2} = +\infty \]
\[ \max_{S_l} \leq \delta \leq \min_{S_r} \]
Union (Numerical part)

\[ \alpha \leftarrow \alpha_c \leftarrow \alpha_l , \alpha_r \]
\[ S \leftarrow S_0 \leftarrow S_l , S_r \]
\[ \alpha \leftarrow \alpha_t \leftarrow \alpha \]
\[ [] \leftarrow S_1 \leftarrow [] , S_l . [\delta] \]
\[ [] \leftarrow S_2 \leftarrow [\delta] . S_l , [] \]

Result:

\[ S_i = \text{sort}(S_i) \quad i \in \{ _, 0, 1, 2 \} \]
\[ S = S_1 . S_0 . S_2 \]
\[ \alpha_c , \alpha_t \neq 0x0 \]
\[ \max_{S_1} \leq v \leq \min_{S_2} \]

\[ S = \text{sort}(S) \]
\[ S_1 = S_2 = [] \]
\[ \land \alpha = \alpha_c = \alpha_t \neq 0x0 \]

\[ S = \text{sort}(S) \land S = S_0 . S_2 \]
\[ S_i = \text{sort}(S_i) \quad i \in \{ l, r, 1, 2 \} \]
\[ S_0 = S_l \land S_1 = [] \land S_2 = [\delta] . S_r \]
\[ \land \alpha = \alpha_t \neq 0x0 \]
\[ \land \alpha_l = \alpha_c \neq 0x0 \]
\[ \land v < \delta = \min_{S_2} \land \max_{S_1} = -\infty \]
\[ \max_{S_l} \leq \delta \leq \min_{S_r} \]

\[ S = \text{sort}(S) \land S = S_1 . S_0 \]
\[ S_i = \text{sort}(S_i) \quad i \in \{ l, r, 1, 2 \} \]
\[ S_0 = S_r \land S_1 = S_l . [\delta] \land S_2 = [] \]
\[ \land \alpha = \alpha_t \neq 0x0 \]
\[ \land \alpha_r = \alpha_c \neq 0x0 \]
\[ \land v \geq \delta = \max_{S_1} \land \min_{S_2} = +\infty \]
\[ \max_{S_l} \leq \delta \leq \min_{S_r} \]
Exemple: insertion in binary search tree

\[
\begin{align*}
\text{tree}(t, S) & \quad \land S = \text{sort}(S) \\
\text{if}(t == \text{NULL}) & \\
\quad \text{// Empty case} \\
\text{else} & \\
\quad \text{ptree } c = t; \\
\quad \text{while}(v < c->d \land c->l \lor v \geq c->d \land c->r) & \\
\quad \quad \text{if}(v < c->d) & \\
\quad \quad \quad c = c->l & \\
\quad \quad \text{else} & \\
\quad \quad \quad c = c->r; & \\
\quad \quad \text{if}(v < c->d) & \\
\quad \quad \quad c->l = m; & \\
\quad \quad \text{else} & \\
\quad \quad \quad c->r = m; & \\
\quad \text{return } t; & \\
\end{align*}
\]

\[
\{\text{tree}(t, \text{sort}(S[v]))\}\]
Exemple: insertion in binary search tree

\[
\{ \text{tree}(t, S) \} \quad \land \quad S = \text{sort}(S)
\]

if( t == NULL ){
    // Empty case
} else{
    ptree c = t;
    while( v < c -> d && c -> l || v >= c -> d && c -> r )
        if( v < c -> d ) {
            c = c -> l
        } else {
            c = c -> r;
        }
    if( v < c -> d ){
        c -> l = m;
    } else{
        c -> r = m;
    }
    return t;
}

\{ \text{tree}(t, \text{sort}(S[v])) \}
Exemple: insertion in binary search tree

\[
\begin{align*}
\text{tree(t, S)} & \quad \text{\{tree(t, S)\}} \\
\wedge S = \text{sort}(S) & \quad \text{\{tree(t, S)\}} \\
\text{if( t== NULL )} & \quad \text{if( t== NULL )} \\
& \quad \{ // Empty case \\
& \quad } \\
\text{else} & \quad \text{else} \\
& \quad \{ \\
& \quad \text{ptree c= t; } \\
& \quad \text{while(v< c->d && c->l || v>= c->d && c->r )} \\
& \quad \text{if(v < c->d) } \\
& \quad \text{c = c->l } \\
& \quad \text{else } \\
& \quad \text{c = c->r; } \\
& \quad \} \\
& \quad \text{if( v< c->d )} \\
& \quad \text{c->l = m; } \\
& \quad \text{else} \\
& \quad \text{c->r = m; } \\
& \quad \} \\
& \quad \} \\
& \quad \text{return t; } \\
& \quad } \\
{\text{tree(t, sort(S.[v]))}} \\
\end{align*}
\]
Exemple: insertion in binary search tree

\[
\begin{align*}
\text{tree}(t, S) & \\
\land S = \text{sort}(S)
\end{align*}
\]

\[
\begin{cases}
\text{if}(t == \text{NULL}) \{
    \text{// Empty case}
\} \text{else} \{
    \text{ptree } c = t;
    \text{while}(v < c->d && c->l \lor v >= c->d && c->r) \\
    \text{if}(v < c->d) \{
    c = c->l
    \} \text{else} \{
    c = c->r;
    \}
    \text{if}(v < c->d) \{
    c->l = m;
    \} \text{else} \{
    c->r = m;
    \}
\text{return } t;
\}
\end{cases}
\]

\[
\text{tree}(t, \text{sort}(S.[v]))
\]
Exemple: insertion in binary search tree

\[ S_i = \text{sort}(S_i) \quad i \in \{ , 1, 2 \} \]
\[ S = S_1.S_0.S_2 \]
\[ S_0 = S_l.[\delta].S_r \]
\[ \alpha_c, \alpha_t \neq 0x0 \]
\[ \max_{S_1} \leq v \leq \min_{S_2} \]
\[ v < \delta \]
Exemple: insertion in binary search tree

\[ S_i = \text{sort}(S_i) \quad i \in \{\_ , 1, 2 \} \]
\[ S = S_1.S_0.S_2 \]
\[ S_0 = S_l.[\delta].S_r \]
\[ \alpha_c, \alpha_t \neq 0x0 \]
\[ \max_{S_1} \leq v \leq \min_{S_2} \]
\[ v < \delta \]

\[ \Rightarrow \text{Invariant found after two iterations!} \]
Exemple: insertion in binary search tree

\[ S_i = \text{sort}(S_i) \quad i \in \{\_, 1, 2, l, r\} \]
\[ S = S_1.S_0.S_2 \]
\[ S_0 = S_l.[\delta].S_r \]
\[ \alpha_c, \alpha_t, \alpha_l \neq 0x0 \]
\[ \max_{S_1} \leq v \leq \min_{S_2} \]
\[ v < \delta \]

\[ S = S_1.S_0.S_2 \]
\[ S_0 = S_l.[\delta].S_r \]
\[ \alpha_c, \alpha_t \neq 0x0 \]
\[ \max_{S_1} \leq v \leq \min_{S_2} \]

→ Invariant found after two iterations!