A Product of Shape & Sequences abstractions

Static Analysis Symposium, Cascais, 2023

Josselin Giet[1], Félix Ridoux[2,3], Xavier Rival[1]

October, 22 2023

[1]INRIA Paris/CNRS/École Normale Supérieure/PSL Research University, Paris, France

[2]IMDEA Software Institute, Madrid, Spain

[3]Univ 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
tree *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:

Analysis	pointer dereference	structural invariants	partial f^{al} correctness	
			SLL	tree
Pointer analysis	✔	✗	✗	✗
Shape analysis based on...				
...3-Value logic *e.g.* [Sagiv et al. TOPLAS, 99]	✔	✔	✗	✗
...Separation logic *e.g.* [Chang et al. POPL, 08]	✔	✔	✗	✗
...k-limited graphs *e.g.* [Bouajjani et al, CAV, 10]	✔	✔	✔	✗

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:

Analysis	pointer dereference	structural invariants	partial f^al correctness	
			SLL	tree
Pointer analysis	✔	✗	✗	✗
Shape analysis based on...				
... 3-Value logic *e.g.* [Sagiv et al. TOPLAS, 99]	✔	✔	✗	✗
... Separation logic *e.g.* [Chang et al. POPL, 08]	✔	✔	✗	✗
... k-limited graphs *e.g.* [Bouajjani et al, CAV, 10]	✔	✔	✔	✗

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

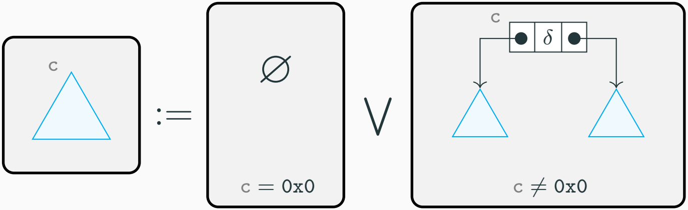
How to improve the expressiveness of automatic static analysis over dynamic data-structures to prove partial functional correctness?

Separation Logic based shape analysis

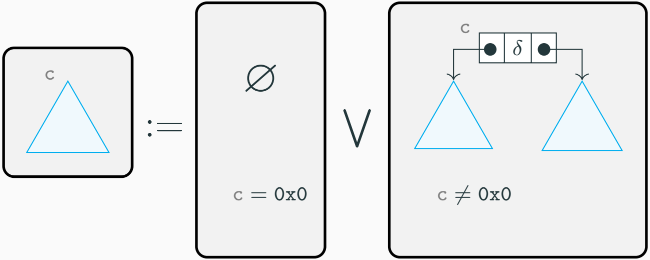
[Chang et al. POPL, 2008] introduces a shape analysis based on abstract interpretation.

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:

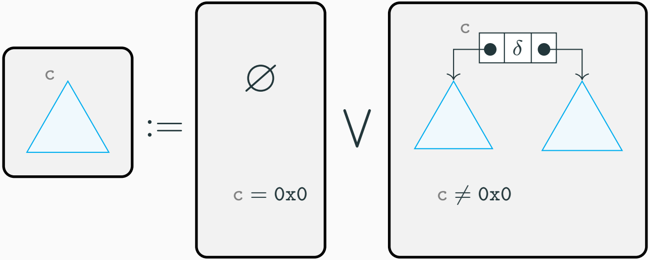


Inductive predicates are not expressive enough



$\Longrightarrow$ This predicate is expressive enough to prove memory safety & structure preservation.

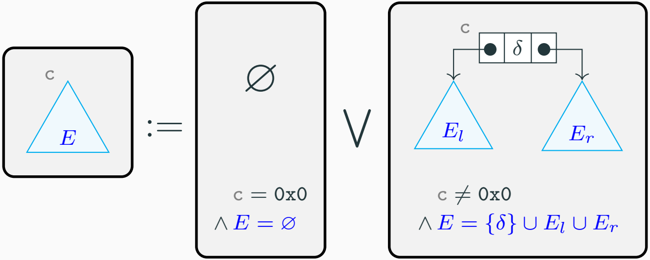
Inductive predicates are not expressive enough



$\Longrightarrow$ 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



$\Longrightarrow$ This predicate is expressive enough to prove memory safety & structure preservation.

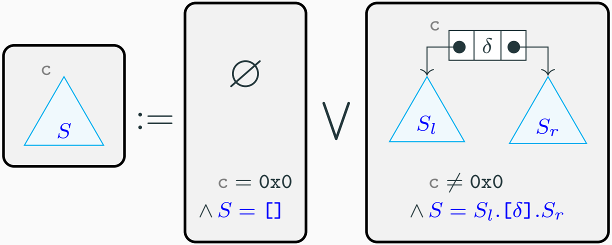
Problem This not enough for partial functional correctness: **tree** forgets the content !

[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: **tree**$(\mathtt{c}, S)$.

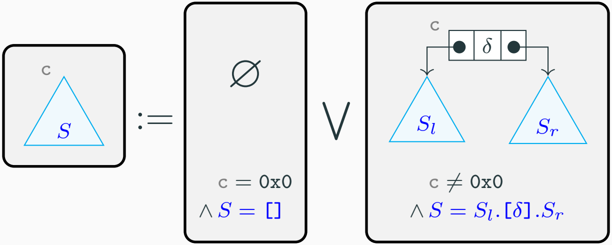


The specification of the (partial) functional correctness of `insert` can be expressed as:

$$\left\{\begin{array}{l}\textbf{tree}(\mathtt{t}, S)\\ S = \textbf{sort}(S)\end{array}\right\}\ \mathtt{r} = \mathtt{insert}(\mathtt{t}, \mathtt{v})\ \left\{\begin{array}{l}\textbf{tree}(\mathtt{r}, S')\\ \text{ where } S' = \textbf{sort}(S.[\mathtt{v}])\end{array}\right\}$$

Sequence parameters

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



The specification of the (partial) functional correctness of `insert` can be expressed as:

$$\left\{\begin{array}{l}\textbf{tree}(\mathtt{t}, S)\\ S = \textbf{sort}(S)\end{array}\right\}\ \mathtt{r} = \mathtt{insert}(\mathtt{t}, \mathtt{v})\ \left\{\begin{array}{l}\textbf{tree}(\mathtt{r}, S')\\ \text{where } S' = \textbf{sort}(S.[\mathtt{v}])\end{array}\right\}$$

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:

$$\left\{ \begin{array}{c} \alpha \mapsto 2 \\ \delta \mapsto 1 \end{array} \right\} \left\{ \begin{array}{c} S \mapsto 4; 6; 1 \\ S_1 \mapsto 4; 6 \end{array} \right\}$$

An **abstract value** $\sigma_{\mathcal{S}}^{\#}$ of the sequence abstract domain $\mathbb{D}_{\mathcal{S}}^{\#}$ consists of:

$$\left(\wedge S_i = E_i,\ \sigma_{\mathcal{M}}^{\#},\ \sigma_{\mathcal{N}}^{\#} \right)$$

A conjuction of sequence definitions

$$S = S_1 . [\delta]$$
$$\wedge S = \mathbf{sort}(S)$$
$$\wedge S_1 = \mathbf{sort}(S_1)$$

An element of a multiset domain $\mathbb{D}_{\mathcal{M}}^{\#}$

$$\mathbf{mset}_S = \{\!\{\delta\}\!\} \uplus \mathbf{mset}_{S_1}$$

An element of a numerical domain $\mathbb{D}_{\mathcal{N}}^{\#}$

$$\mathbf{min}_S \leqslant \delta \leqslant \mathbf{max}_S$$
$$\wedge\ \mathbf{max}_{S_1} \leqslant \delta$$
$$\dots$$
$$\wedge\ \mathbf{len}_S = 1 + \mathbf{len}_{S_1}$$

Adding a new constraint

$\textbf{guard}_s : \mathbb{D}_s^\# \to$ seq. constraint $\to \mathbb{D}_s^\#$

$$S = S_1.[\alpha] \wedge S = \textbf{sort}(S)$$
$$\wedge\, S_1 = \textbf{sort}(S_1)$$

$$\wedge\, \textbf{mset}_S = \{\!\{\alpha\}\!\} \uplus \textbf{mset}_{S_1}$$

$$\wedge\, \textbf{len}_S = 1 + \textbf{len}_{S_1} + \textbf{len}_{S_2}$$

$$\wedge\, \textbf{min}_S \leqslant \alpha \leqslant \textbf{max}_S$$
$$\wedge\, \textbf{min}_S \leqslant \textbf{min}_{S_1} \wedge \textbf{max}_{S_1} \leqslant \textbf{max}_S$$

To assume $S_r = [\alpha]$, $\textbf{guard}_s$ follows this algorithm:

Adding a new constraint

$\mathbf{guard}_s : \mathbb{D}_s^\# \to$ seq. constraint $\to \mathbb{D}_s^\#$

$$
\begin{aligned}
& S = S_1.[\alpha] \wedge S = \mathbf{sort}(S) \\
\wedge\ & S_1 = \mathbf{sort}(S_1) \\
\wedge\ & S_r = [\alpha] \\
\wedge\ & \mathbf{mset}_S = \{\!\{\alpha\}\!\} \uplus \mathbf{mset}_{S_1} \\
\wedge\ & \mathbf{len}_S = 1 + \mathbf{len}_{S_1} + \mathbf{len}_{S_2} \\
\wedge\ & \mathbf{min}_S \leqslant \alpha \leqslant \mathbf{max}_S \\
\wedge\ & \mathbf{min}_S \leqslant \mathbf{min}_{S_1} \wedge \mathbf{max}_{S_1} \leqslant \mathbf{max}_S
\end{aligned}
$$

To assume $S_r = [\alpha]$, $\mathbf{guard}_s$ follows this algorithm:

1. add the new definition in the conjunction

Adding a new constraint

$\mathbf{guard}_s : \mathbb{D}_s^\# \to \text{seq. constraint} \to \mathbb{D}_s^\#$

$$
\begin{aligned}
& S = S_1.[\alpha] \wedge S = \mathbf{sort}(S) \\
& \wedge S_1 = \mathbf{sort}(S_1) \\
& \wedge\ S_r = [\alpha] \\
& \wedge \mathbf{mset}_S = \{\!\{\alpha\}\!\} \uplus \mathbf{mset}_{S_1} \\
& \wedge\ \mathbf{mset}_{S_r} = \{\!\{\alpha\}\!\} \\
& \wedge \mathbf{len}_S = 1 + \mathbf{len}_{S_1} + \mathbf{len}_{S_2} \\
& \wedge\ \mathbf{len}_{S_r} = 1 \\
& \wedge \mathbf{min}_S \leqslant \alpha \leqslant \mathbf{max}_S \\
& \wedge \mathbf{min}_S \leqslant \mathbf{min}_{S_1} \wedge \mathbf{max}_{S_1} \leqslant \mathbf{max}_S \\
& \wedge\ \mathbf{min}_{S_r} = \alpha = \mathbf{max}_{S_r}
\end{aligned}
$$

To assume $S_r = [\alpha]$, $\mathbf{guard}_s$ follows this algorithm:

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

Adding a new constraint

$\textbf{guard}_s : \mathbb{D}_s^\# \rightarrow \text{seq. constraint} \rightarrow \mathbb{D}_s^\#$

$$
\begin{aligned}
& S = S_1.S_r \wedge S = \textbf{sort}(S) \\
\wedge\ & S_1 = \textbf{sort}(S_1) \\
\wedge\ & S_r = [\alpha] \\
\\
\wedge\ & \textbf{mset}_S = \{\!\{\alpha\}\!\} \uplus \textbf{mset}_{S_1} \\
\wedge\ & \textbf{mset}_{S_r} = \{\!\{\alpha\}\!\} \\
\\
\wedge\ & \textbf{len}_S = 1 + \textbf{len}_{S_1} + \textbf{len}_{S_2} \\
\wedge\ & \textbf{len}_{S_r} = 1 \\
\\
\wedge\ & \textbf{min}_S \leqslant \alpha \leqslant \textbf{max}_S \\
\wedge\ & \textbf{min}_S \leqslant \textbf{min}_{S_1} \wedge \textbf{max}_{S_1} \leqslant \textbf{max}_S \\
\wedge\ & \textbf{min}_{S_r} = \alpha = \textbf{max}_{S_r}
\end{aligned}
$$

To assume $S_r = [\alpha]$, $\textbf{guard}_s$ follows this algorithm:

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

Adding a new constraint

$\textbf{guard}_s : \mathbb{D}_s^\# \to \text{seq. constraint} \to \mathbb{D}_s^\#$

$$\begin{aligned}
& S = S_1.S_r \wedge S = \textbf{sort}(S) \\
\wedge\ & S_1 = \textbf{sort}(S_1) \\
\wedge\ & S_r = [\alpha] \wedge S_r = \textbf{sort}(S_r) \\
\\
\wedge\ & \textbf{mset}_S = \{\!\{\alpha\}\!\} \uplus \textbf{mset}_{S_1} \\
\wedge\ & \textbf{mset}_{S_r} = \{\!\{\alpha\}\!\} \\
\\
\wedge\ & \textbf{len}_S = 1 + \textbf{len}_{S_1} + \textbf{len}_{S_2} \\
\wedge\ & \textbf{len}_{S_r} = 1 \\
\\
\wedge\ & \textbf{min}_S \leqslant \alpha \leqslant \textbf{max}_S \\
\wedge\ & \textbf{min}_S \leqslant \textbf{min}_{S_1} \wedge \textbf{max}_{S_1} \leqslant \textbf{max}_S \\
\wedge\ & \textbf{min}_{S_r} = \alpha = \textbf{max}_{S_r}
\end{aligned}$$

To assume $S_r = [\alpha]$, $\textbf{guard}_s$ follows this algorithm:

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

$$\frac{S = S_1.\dots.S_n \qquad \forall i, S_i = \textbf{sort}(S_i) \quad \forall i < j, \textbf{max}_{S_i} \leqslant \textbf{min}_{S_j}}{S = \textbf{sort}(S)}$$

Adding a new constraint

$\mathbf{guard}_s : \mathbb{D}_s^\# \to$ seq. constraint $\to \mathbb{D}_s^\#$

$$
\begin{aligned}
& S = S_1.S_r \wedge S = \mathbf{sort}(S) \\
& \wedge S_1 = \mathbf{sort}(S_1) \\
& \wedge S_r = [\alpha] \wedge S_r = \mathbf{sort}(S_r) \\
& \wedge \mathbf{mset}_S = \{\!\{\alpha\}\!\} \uplus \mathbf{mset}_{S_1} \\
& \wedge \mathbf{mset}_{S_r} = \{\!\{\alpha\}\!\} \\
& \wedge \mathbf{len}_S = 1 + \mathbf{len}_{S_1} + \mathbf{len}_{S_2} \\
& \wedge \mathbf{len}_{S_r} = 1 \\
& \wedge \mathbf{min}_S \leqslant \alpha \leqslant \mathbf{max}_S \\
& \wedge \mathbf{min}_S \leqslant \mathbf{min}_{S_1} \wedge \mathbf{max}_{S_1} \leqslant \mathbf{max}_S \\
& \wedge \mathbf{min}_{S_r} = \alpha = \mathbf{max}_{S_r}
\end{aligned}
$$

To assume $S_r = [\alpha]$, $\mathbf{guard}_s$ follows this algorithm:

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

$$
\frac{S = S_1 \dots . S_n \qquad \forall i, S_i = \mathbf{sort}(S_i) \quad \forall i < j, \mathbf{max}_{S_i} \leqslant \mathbf{min}_{S_j}}{S = \mathbf{sort}(S)}
$$

5. detect & remove cyclic constraints

Adding a new constraint

$\mathbf{guard}_s : \mathbb{D}_s^\# \rightarrow$ seq. constraint $\rightarrow \mathbb{D}_s^\#$

$$
\begin{aligned}
& S = S_1.S_r \wedge S = \mathbf{sort}(S) \\
\wedge\; & S_1 = \mathbf{sort}(S_1) \\
\wedge\; & S_r = [\alpha] \wedge S_r = \mathbf{sort}(S_r) \\
\\
\wedge\; & \mathbf{mset}_S = \{\!\{\alpha\}\!\} \uplus \mathbf{mset}_{S_1} \\
\wedge\; & \mathbf{mset}_{S_r} = \{\!\{\alpha\}\!\} \\
\\
\wedge\; & \mathbf{len}_S = 1 + \mathbf{len}_{S_1} + \mathbf{len}_{S_2} \\
\wedge\; & \mathbf{len}_{S_r} = 1 \\
\\
\wedge\; & \mathbf{min}_S \leqslant \alpha \leqslant \mathbf{max}_S \\
\wedge\; & \mathbf{min}_S \leqslant \mathbf{min}_{S_1} \wedge \mathbf{max}_{S_1} \leqslant \mathbf{max}_S \\
\wedge\; & \mathbf{min}_{S_r} = \alpha = \mathbf{max}_{S_r}
\end{aligned}
$$

To assume $S_r = [\alpha]$, $\mathbf{guard}_s$ follows this algorithm:

1. add the new definition in the conjunction
2. add content/length/bounds constraints
3. fold other definitions
4. Saturate constraints
$$\frac{S = S_1 \dots . S_n \qquad \forall i, S_i = \mathbf{sort}(S_i) \quad \forall i < j, \mathbf{max}_{S_i} \leqslant \mathbf{min}_{S_j}}{S = \mathbf{sort}(S)}$$
5. detect & remove cyclic constraints

Theorem: Soundness of $\mathbf{guard}_s$

$\gamma_s(\mathbf{guard}_s(\sigma_s^\#, S = E))$ contains all valuations in $\gamma_s(\sigma_s^\#)$ satisfying $S = E$.

Abstract lattice operators

- $\textbf{verify}_{\mathcal{S}} : \mathbb{D}^{\#}_{\mathcal{S}} \to \text{seq constraint} \to \{\textbf{true}, \textbf{false}\}$
  $\textbf{verify}_{\mathcal{S}}(\sigma^{\#}_{\mathcal{S}}, S = E)$ conservatively checks if $\sigma^{\#}_{\mathcal{S}}$ satisfies $S = E$.

- $\sqsubseteq_{\mathcal{S}}: \mathbb{D}^{\#}_{\mathcal{S}} \to \mathbb{D}^{\#}_{\mathcal{S}} \to \{\textbf{true}, \textbf{false}\}$
  Abstract inclusion checking, using **verify**$_{\mathcal{S}}$

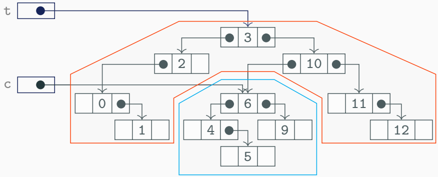
- $\sqcup_{\mathcal{S}} : \mathbb{D}^{\#}_{\mathcal{S}} \to \mathbb{D}^{\#}_{\mathcal{S}} \to \mathbb{D}^{\#}_{\mathcal{S}}$
  That tries to infer common definitions from both inputs.
  Example $\begin{pmatrix} S = S_1.S_2 \\ \wedge\, S_3 = [] \end{pmatrix} \sqcup_{\mathcal{S}} \begin{pmatrix} S = S_2.S_3 \\ \wedge\, S_1 = [] \end{pmatrix} = (S = S_1.S_2.S_3)$

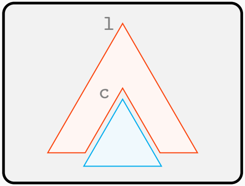
- $\nabla_{\mathcal{S}} : \mathbb{D}^{\#}_{\mathcal{S}} \to \mathbb{D}^{\#}_{\mathcal{S}} \to \mathbb{D}^{\#}_{\mathcal{S}}$
  That selects the constraints in the left arguments verified in the right one.

Shape analysis with sequence predicates

Integrating sequence parameters in the shape domain

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

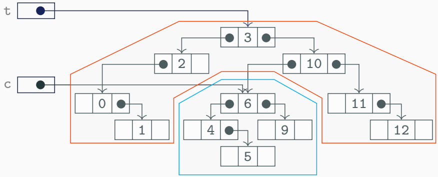


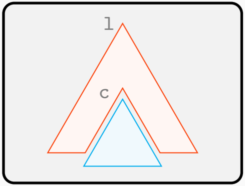


The shape domain automatically derives **treeseg** from **tree**.
The analysis must keep tracks of the content stored in the segment

Integrating sequence parameters in the shape domain

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





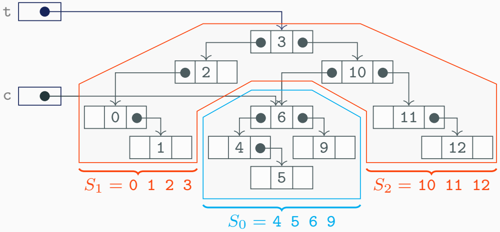
The shape domain automatically derives **treeseg** from **tree**.
The analysis must keep tracks of the content stored in the segment

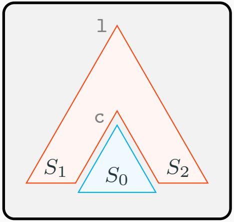
In order to reason precisely over inductive predicates, the shape analysis relies on:

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

For each of these operations, the shape domain should **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

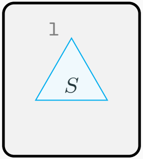
The analysis needs to recall the location of the missing sequence in **treeseg**.

$\Longrightarrow$ the segment predicate has **two sequence parameters**: S_1, S_2

One for each side of the missing sequence

Refining abstract memory state with unfolding

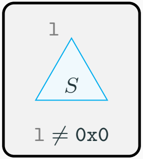
To analyze `if(l){v= l->data}` with initial state **tree**$(\mathtt{l}, S)$



Refining abstract memory state with unfolding

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

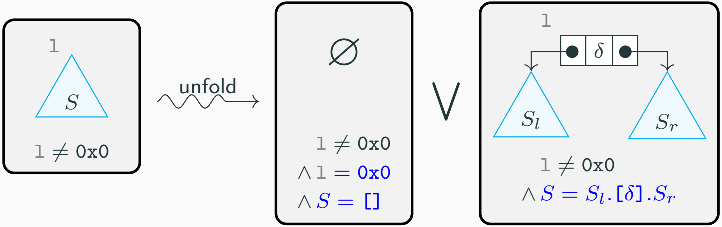
1. The numerical constraint $\texttt{l} \neq \texttt{0x0}$ is guarded in the numerical part of the sequence domain.



Refining abstract memory state with unfolding

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

1. The numerical constraint $\mathtt{l} \neq \mathtt{0x0}$ is guarded in the numerical part of the sequence domain.
2. To materialize `l->data`, the analysis **unfolds the predicate**
   The abstract memory is replaced by the definition: δ, S_l, S_r are **fresh variables**
   The numerical and sequences constraints are guarded in the sequence domain



Refining abstract memory state with unfolding

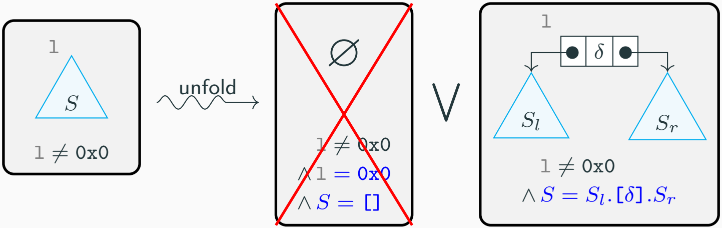
To analyze `if(l){v= l->data}` with initial state **tree**$(\texttt{l}, S)$

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

   The abstract memory is replaced by the definition: δ, 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 $\Longrightarrow$ Discarded



Refining abstract memory state with unfolding

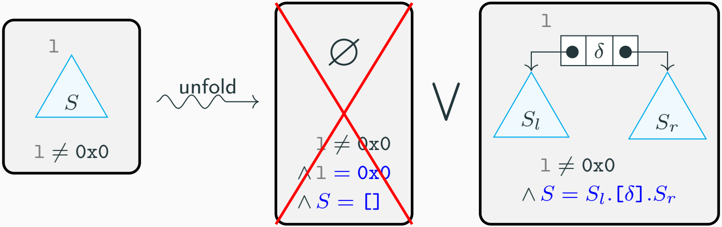
To analyze `if(l){v= l->data}` with initial state **tree**(l, S)

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

   The abstract memory is replaced by the definition: δ, S_l, S_r are **fresh variables**

   The numerical and sequences constraints are guarded in the sequence domain

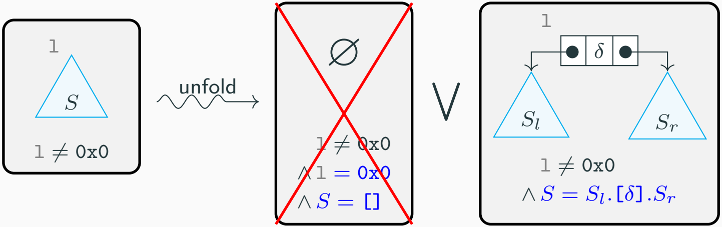
   - **The empty case**: Inconsistent with the `if` assumption $\Longrightarrow$ Discarded
   - **The non-empty case**: `c->data` corresponds to δ.



Refining abstract memory state with unfolding

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

1. The numerical constraint $\mathtt{l} \neq \mathtt{0x0}$ is guarded in the numerical part of the sequence domain.
2. To materialize `l->data`, the analysis **unfolds the predicate**
   The abstract memory is replaced by the definition: δ, 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->data` corresponds to δ.
3. The assignment $\mathtt{v} \leftarrow \delta$ is performed.



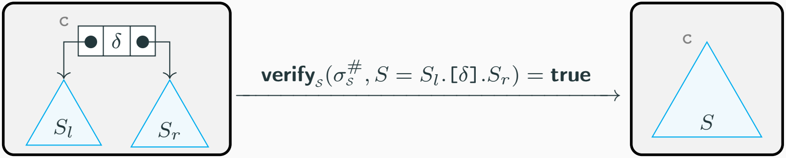
Theorem: Soundness of unfolding

The resulting disjunction of abstract states over approximates the original state.

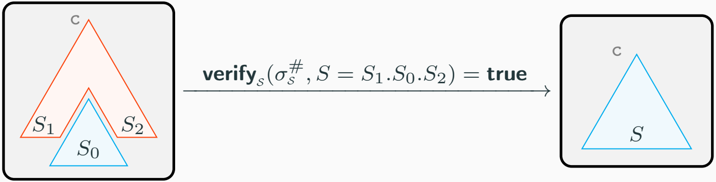
Folding the abstract state

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



Folding segment and predicates



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

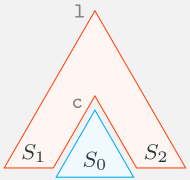
Folds both inputs until they are syntactically equal.

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.

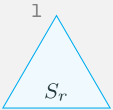
Proving the insertion into a BST

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



$$S = S_1.S_0.S_2$$
$$\wedge\, S = \textbf{sort}(S)$$
$$\wedge\, S_i = \textbf{sort}(S_i),\ \ i = 0, 1, 2$$
$$\wedge\, \mathtt{l}, \mathtt{c} \neq \mathtt{0x0}$$
$$\wedge\, \textbf{max}_{S_1} \leqslant \mathtt{v} < \textbf{min}_{S_2}$$

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



$$S_r = \textbf{sort}(S.[\mathtt{v}])$$

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

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

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 ∇_S to stabilize.

Experiment 2: Real-world libraries

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

	Linux	FreeRTOS	GDSL
Circular DLL with distinguished header	Yes	Yes	Yes
Extreme sentinel nodes	No	No	Yes
Intrusive	Yes	Yes	No
Pointer to header	No	Yes	No
Length in header	No	Yes	Yes
Sorted	No	Yes	No

	Linux		FreeRTOS		GDSL	
	wo/ seq	w/ seq	wo/ seq	w/ seq	wo/ seq	w/ seq
Safe	4/4✔	4/4✔	4/4✔	4/4✔	13✔ 1✗(†)	14/14✔
Fc		4/4✔		4/4✔		12✔ 2✗(‡)

†: 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 $S = S_1 \dots S_n$, then

$$S = \textbf{sort}(S) \Leftrightarrow \forall i, S_i = \textbf{sort}(S_i) \wedge \forall i < j, \textbf{max}_{S_i} \leqslant \textbf{min}_{S_j}$$

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

$\Longrightarrow$ NO ! Because of the empty sequence case !

By consistency of the concretization: $\nu_S(S) = \varepsilon \Longrightarrow \begin{cases} \textbf{max}_S = -\infty \\ \textbf{min}_S = +\infty \end{cases}$

Consider $\nu_S = \left\{ \begin{array}{r} S \mapsto 3\,1 \\ S_1 \mapsto 3 \\ S_2 \mapsto \varepsilon \\ S_3 \mapsto 1 \end{array} \right\}$

We have indeed:

$\nu_S \models S = S_1.S_2.S_3$

$\nu_S \models S_i = \textbf{sort}(S_i), \quad \forall i$

$\nu_S \models \textbf{max}_{S_1} \leqslant \textbf{min}_{S_2}$

$\nu_S \models \textbf{max}_{S_2} \leqslant \textbf{min}_{S_3}$

But:

$\nu_S \not\models S = \textbf{sort}(S)$

Removing cyclic constraints

Assume the abstract state $\sigma_{\mathcal{S}}^{\#}$ contains the following constraints:

$$\begin{aligned} & S = S_1.S'.S_2 \\ \wedge\, & S' = S_3.S'' \\ \wedge\, & S'' = S.S_4 \end{aligned}$$

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

$$S = S_1.S_3.S.S_4.S_2$$

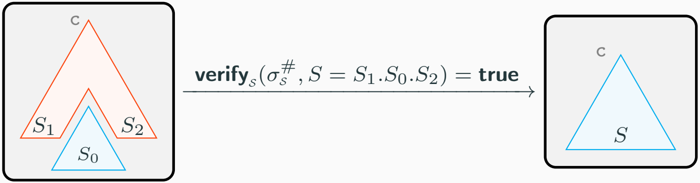
The constraints over S, S', S'' are replaced by $\begin{cases} S_1 = S_2 = S_3 = S_4 = [] \\ S = S' = S'' \end{cases}$

If one constraint contains at least one atom $[\alpha]$, then the state is reduced to $\bot_{\mathcal{S}}$.

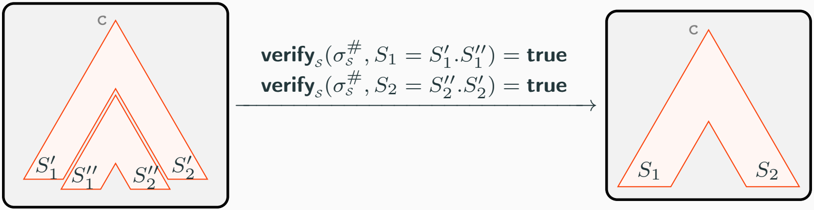
$S = \textbf{sort}(S)$ does not count as a cyclic constraint as the implementation of the abstract domain does not represent it as such.

Concatenating inductive predicates

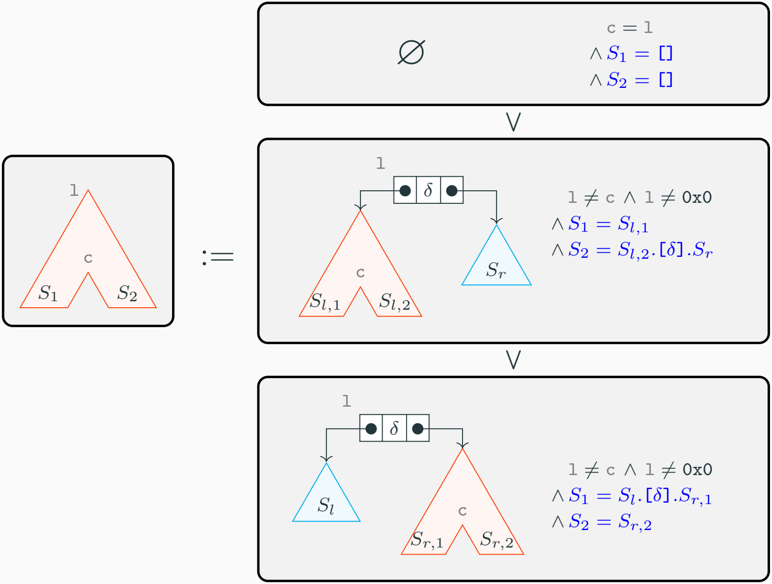
seg-full case



seg-seg case



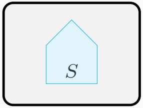
Segment tree predicate



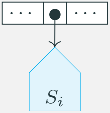
Hypothesis to derive segment from full predicate

Hypothesis

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

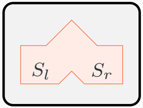


:= $\bigvee$

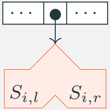


$S = E.S_i.E'$

$\bigvee$



:= $\bigvee$



$$S_l \boxdot S_r = E.S_i.E'\{S_i \leftarrow S_{i,l} \boxdot S_{i,r}\}$$
$$= E.S_{i,l} \boxdot S_{i,r}.E'$$
$$\Longrightarrow \begin{cases} S_l = E.S_{i,l} \\ S_r = S_{i,r}.E' \end{cases}$$

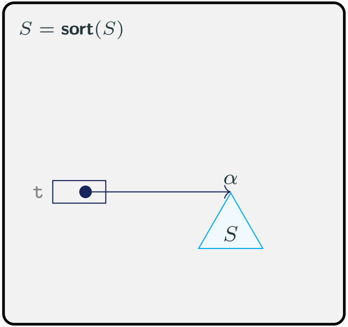
$\bigvee$

Exemple: insertion in binary search tree

$\{\textbf{tree}(\texttt{t}, S) \quad \wedge \quad S = \textbf{sort}(S)\}$

```
●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;
}
```

$\{\textbf{tree}(\texttt{t}, \textbf{sort}(S.[\texttt{v}]))\}$



Exemple: insertion in binary search tree

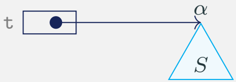
$\{\textbf{tree}(\texttt{t}, S) \wedge S = \textbf{sort}(S)\}$

```
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;
}
```

$\{\textbf{tree}(\texttt{t}, \textbf{sort}(S.[\texttt{v}]))\}$

$S = \textbf{sort}(S)$

$\wedge\ \alpha \neq \texttt{0x0}$



Exemple: insertion in binary search tree

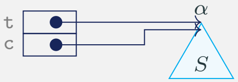
$\{\textbf{tree}(\mathtt{t}, S) \quad \wedge \quad S = \textbf{sort}(S)\}$

```
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;
}
```

$\{\textbf{tree}(\mathtt{t}, \textbf{sort}(S.[\mathtt{v}]))\}$

$S = \textbf{sort}(S)$

$\wedge\ \alpha \neq \mathtt{0x0}$



Exemple: insertion in binary search tree

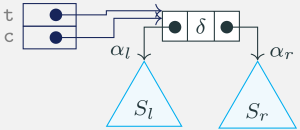
$\{\textbf{tree}(\texttt{t}, S) \quad \wedge \quad S = \textbf{sort}(S)\}$

```
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;
}
```

$\{\textbf{tree}(\texttt{t}, \textbf{sort}(S.[\texttt{v}]))\}$

$S = \textbf{sort}(S) \wedge S = S_l.[\delta].S_r$
$S_i = \textbf{sort}(S_i) \quad i \in \{l, r\}$

$\wedge\ \alpha \neq \texttt{0x0}$
$\textbf{max}_{S_l} \leqslant \delta \leqslant \textbf{min}_{S_r}$



Exemple: insertion in binary search tree

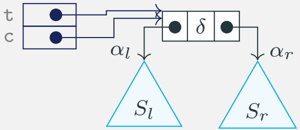
$\{\textbf{tree}(\mathtt{t}, S) \quad \wedge \quad S \quad = \textbf{sort}(S)\}$

```
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;
}
```

$\{\textbf{tree}(\mathtt{t}, \textbf{sort}(S.[\mathtt{v}]))\}$

$S = \textbf{sort}(S) \wedge S = S_l.[\delta].S_r$
$S_i = \textbf{sort}(S_i) \quad i \in \{l, r\}$

$\wedge\ \alpha \neq \mathtt{0x0} \wedge \mathtt{v} < \delta \wedge \alpha_l \neq \mathtt{0x0}$
$\textbf{max}_{S_l} \leqslant \delta \leqslant \textbf{min}_{S_r}$



Exemple: insertion in binary search tree

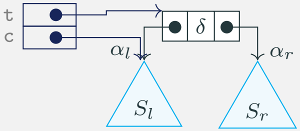
$\{\textbf{tree}(\mathtt{t}, S) \quad \wedge \quad S \quad = \textbf{sort}(S)\}$

```
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;
}
```

$\{\textbf{tree}(\mathtt{t}, \textbf{sort}(S.[\mathtt{v}]))\}$

$S = \textbf{sort}(S) \wedge S = S_l.[\delta].S_r$
$S_i = \textbf{sort}(S_i) \quad i \in \{l, r\}$

$\wedge\ \alpha \neq 0x0 \wedge \mathtt{v} < \delta \wedge \alpha_l \neq 0x0$
$\textbf{max}_{S_l} \leqslant \delta \leqslant \textbf{min}_{S_r}$



Exemple: insertion in binary search tree

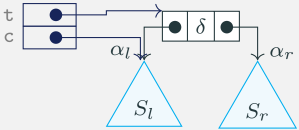
$\{\mathbf{tree}(\mathtt{t}, S) \quad \wedge \quad S = \mathbf{sort}(S)\}$

```
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;
}
```

$\{\mathbf{tree}(\mathtt{t}, \mathbf{sort}(S.[\mathtt{v}]))\}$

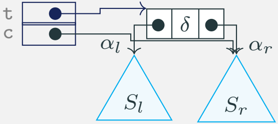
$S = \mathbf{sort}(S) \wedge S = S_l.[\delta].S_r$
$S_i = \mathbf{sort}(S_i) \quad i \in \{l, r\}$

$\wedge\ \alpha \neq \mathtt{0x0} \wedge \mathtt{v} < \delta \wedge \alpha_l \neq \mathtt{0x0}$
$\mathbf{max}_{S_l} \leqslant \delta \leqslant \mathbf{min}_{S_r}$

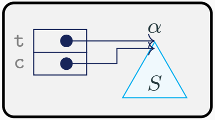


$S = \mathbf{sort}(S) \wedge S = S_l.[\delta].S_r$
$S_i = \mathbf{sort}(S_i) \quad i \in \{l, r\}$

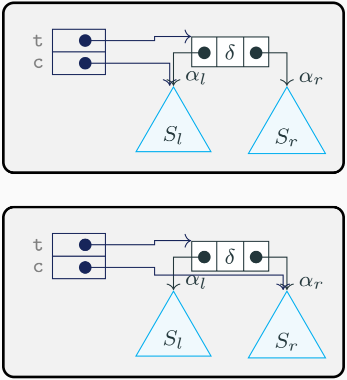
$\wedge \alpha \neq \mathtt{0x0} \wedge \mathtt{v} \geqslant \delta \wedge \alpha_r \neq \mathtt{0x0}$
$\mathbf{max}_{S_l} \leqslant \delta \leqslant \mathbf{min}_{S_r}$

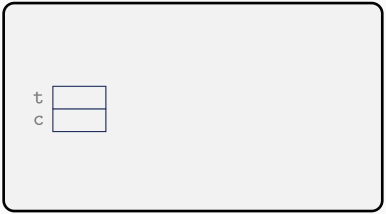


Union (Shape part)

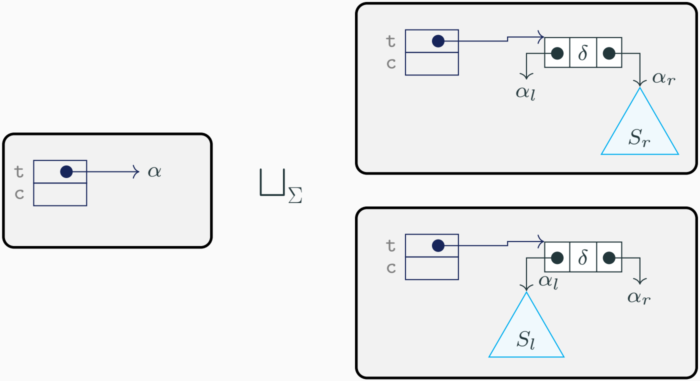


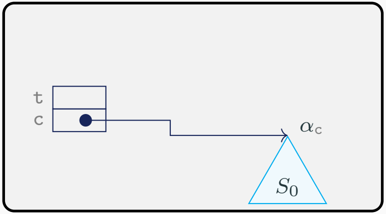
$\bigsqcup_{\Sigma}$





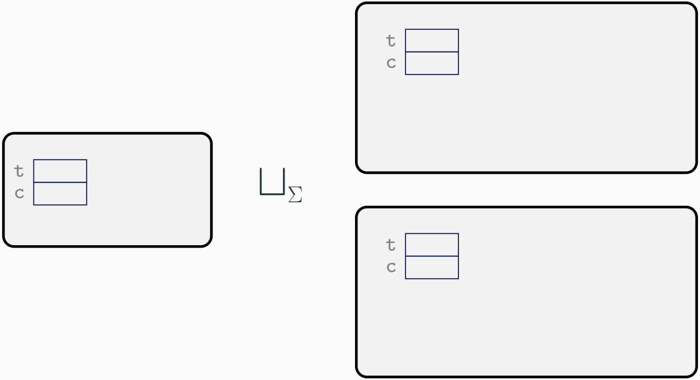
Union (Shape part)

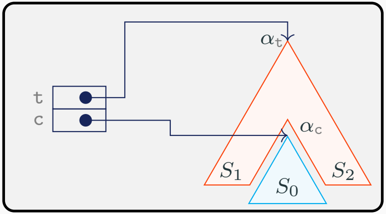




$$\alpha \leftarrow\!\shortmid\, \alpha_{\mathsf{c}} \mapsto \alpha_l\;, \alpha_r$$
$$S \leftarrow\!\shortmid\, S_0 \mapsto S_l\;, S_r$$

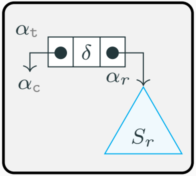
Union (Shape part)





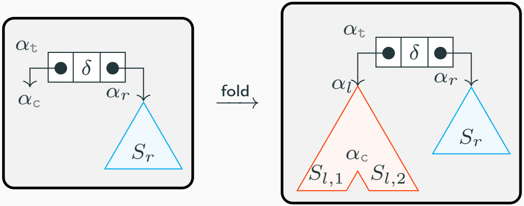
$$\alpha \hookleftarrow \alpha_{c} \mapsto \alpha_l \,, \alpha_r$$
$$S \hookleftarrow S_0 \mapsto S_l \,, S_r$$
$$\alpha \hookleftarrow \alpha_{t} \mapsto \alpha \,, \alpha$$
$$[] \hookleftarrow S_1 \mapsto ?? \,, ??$$
$$[] \hookleftarrow S_2 \mapsto ?? \,, ??$$

Weakening: inclusion test



To verify:

Weakening: inclusion test



To verify:

$S_{l,1} = []$

$S_{l,2} = []$

$\alpha_l = \alpha_c$

The constraints are simplified.

$\alpha \neq 0$

$\alpha_l = \alpha_l$

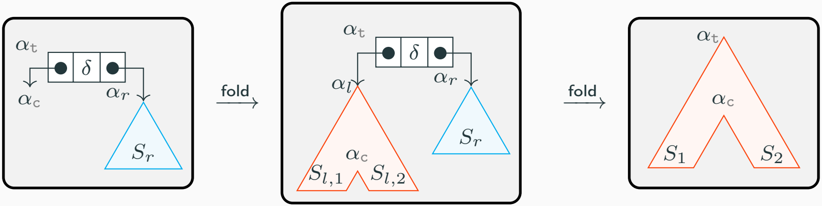
$S_1 = []$

$S_2 = [\delta].S_r$

The numerical ones are checked with **verify**$_s$

The sequence ones are used for definition of S_1 and S_2

Weakening: inclusion test



To verify:

$S_{l,1} = []$

$S_{l,2} = []$

$\alpha_l = \alpha_c$

$S_1 = S_{l,1}$

$S_2 = S_{l,2}.[\delta].S_r$

$\alpha_t \neq 0x0$

The constraints are simplified.

$\alpha \neq 0$

$\alpha_l = \alpha_l$

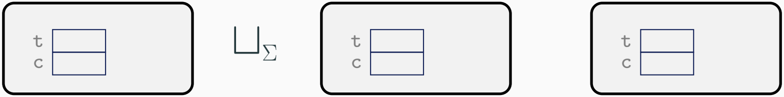
$S_1 = []$

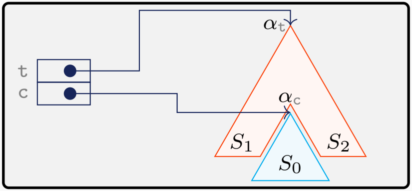
$S_2 = [\delta].S_r$

The numerical ones are checked with **verify**$_s$

The sequence ones are used for definition of S_1 and S_2

Union (Shape part)





$$\alpha \leftarrow\!\dashv \alpha_{\mathtt{c}} \mapsto \alpha_l \quad , \alpha_r$$
$$S \leftarrow\!\dashv S_0 \mapsto S_l \quad , S_r$$
$$\alpha \leftarrow\!\dashv \alpha_{\mathtt{t}} \mapsto \alpha \quad , \alpha$$
$$[] \leftarrow\!\dashv S_1 \mapsto [] \quad , S_l.[\delta]$$
$$[] \leftarrow\!\dashv S_2 \mapsto [\delta].S_r \, , []$$

Union (Numerical part)

$\alpha \leftarrow\!\shortmid\, \alpha_c \mapsto \alpha_l \quad , \alpha_r$

$S \leftarrow\!\shortmid\, S_0 \mapsto S_l \quad , S_r$

$\alpha \leftarrow\!\shortmid\, \alpha_t \mapsto \alpha \quad , \alpha$

$[] \leftarrow\!\shortmid\, S_1 \mapsto [] \quad , S_l.[\delta]$

$[] \leftarrow\!\shortmid\, S_2 \mapsto [\delta].S_l \, , []$

Result:

$S = \textbf{sort}(S)$

$\wedge \alpha \neq 0x0$

$\bigsqcup_\Sigma$

$S = \textbf{sort}(S) \wedge S = S_l.[\delta].S_r$

$S_i = \textbf{sort}(S_i) \quad i \in \{l, r\}$

$\wedge \alpha \neq 0x0$

$\wedge \alpha_l \neq 0x0$

$\wedge v < \delta$

$\textbf{max}_{S_l} \leqslant \delta \leqslant \textbf{min}_{S_r}$

$S = \textbf{sort}(S) \wedge S = S_l.[\delta].S_r$

$S_i = \textbf{sort}(S_i) \quad i \in \{l, r\}$

$\wedge \alpha \neq 0x0$

$\wedge \alpha_r \neq 0x0$

$\wedge v \geqslant \delta$

$\textbf{max}_{S_l} \leqslant \delta \leqslant \textbf{min}_{S_r}$

Union (Numerical part)

$$\begin{array}{ll} \alpha \leftarrow\!\!\dashv \alpha_c \mapsto \alpha_l & , \alpha_r \\ S \leftarrow\!\!\dashv S_0 \mapsto S_l & , S_r \\ \alpha \leftarrow\!\!\dashv \alpha_t \mapsto \alpha & , \alpha \\ [] \leftarrow\!\!\dashv S_1 \mapsto [] & , S_l.[\delta] \\ [] \leftarrow\!\!\dashv S_2 \mapsto [\delta].S_l & , [] \end{array}$$

Result:

$$\begin{array}{l} S = \textbf{sort}(S) \\ S_1 = S_2 = [] \\ \wedge \alpha = \alpha_c = \alpha_t \neq \texttt{0x0} \end{array}$$

$$\bigsqcup_\Sigma$$

$$\begin{array}{l} S = \textbf{sort}(S) \wedge S = S_0.S_2 \\ S_i = \textbf{sort}(S_i) \quad i \in \{l, r, 1, 2\} \\ S_0 = S_l \wedge S_1 = [] \wedge S_2 = [\delta].S_r \\ \wedge \alpha = \alpha_t \neq \texttt{0x0} \\ \wedge \alpha_l = \alpha_c \neq \texttt{0x0} \\ \wedge \texttt{v} < \delta = \textbf{min}_{S_2} \wedge \textbf{max}_{S_1} = -\infty \\ \textbf{max}_{S_l} \leqslant \delta \leqslant \textbf{min}_{S_r} \end{array}$$

$$\begin{array}{l} S = \textbf{sort}(S) \wedge S = S_1.S_0 \\ S_i = \textbf{sort}(S_i) \quad i \in \{l, r, 1, 2\} \\ S_0 = S_r \wedge S_1 = S_l.[\delta] \wedge S_2 = [] \\ \wedge \alpha = \alpha_t \neq \texttt{0x0} \\ \wedge \alpha_r = \alpha_c \neq \texttt{0x0} \\ \wedge \texttt{v} \geqslant \delta = \textbf{max}_{S_1} \wedge \textbf{min}_{S_2} = +\infty \\ \textbf{max}_{S_l} \leqslant \delta \leqslant \textbf{min}_{S_r} \end{array}$$

Union (Numerical part)

$$\begin{array}{ll} \alpha \leftarrow\!\!\dashv \alpha_c \mapsto \alpha_l & , \alpha_r \\ S \leftarrow\!\!\dashv S_0 \mapsto S_l & , S_r \\ \alpha \leftarrow\!\!\dashv \alpha_t \mapsto \alpha & , \alpha \\ [] \leftarrow\!\!\dashv S_1 \mapsto [] & , S_l.[\delta] \\ [] \leftarrow\!\!\dashv S_2 \mapsto [\delta].S_l & , [] \end{array}$$

Result:

$$S_i = \textbf{sort}(S_i) \quad i \in \{\_, 0, 1, 2\}$$

$$\begin{array}{l} S = \textbf{sort}(S) \\ S_1 = S_2 = [] \\ \wedge \alpha = \alpha_c = \alpha_t \neq \texttt{0x0} \end{array} \quad \bigsqcup_{\Sigma}$$

$$\begin{array}{l} S = \textbf{sort}(S) \wedge S = S_0.S_2 \\ S_i = \textbf{sort}(S_i) \quad i \in \{l, r, 1, 2\} \\ S_0 = S_l \wedge S_1 = [] \wedge S_2 = [\delta].S_r \\ \wedge \alpha = \alpha_t \neq \texttt{0x0} \\ \wedge \alpha_l = \alpha_c \neq \texttt{0x0} \\ \wedge \mathtt{v} < \delta = \textbf{min}_{S_2} \wedge \textbf{max}_{S_1} = -\infty \\ \textbf{max}_{S_l} \leqslant \delta \leqslant \textbf{min}_{S_r} \end{array}$$

$$\begin{array}{l} S = \textbf{sort}(S) \wedge S = S_1.S_0 \\ S_i = \textbf{sort}(S_i) \quad i \in \{l, r, 1, 2\} \\ S_0 = S_r \wedge S_1 = S_l.[\delta] \wedge S_2 = [] \\ \wedge \alpha = \alpha_t \neq \texttt{0x0} \\ \wedge \alpha_r = \alpha_c \neq \texttt{0x0} \\ \wedge \mathtt{v} \geqslant \delta = \textbf{max}_{S_1} \wedge \textbf{min}_{S_2} = +\infty \\ \textbf{max}_{S_l} \leqslant \delta \leqslant \textbf{min}_{S_r} \end{array}$$

Union (Numerical part)

$$\begin{aligned}
\alpha &\leftarrow\!\!\dashv \alpha_c \mapsto \alpha_l &&, \alpha_r \\
S &\leftarrow\!\!\dashv S_0 \mapsto S_l &&, S_r \\
\alpha &\leftarrow\!\!\dashv \alpha_t \mapsto \alpha &&, \alpha \\
[] &\leftarrow\!\!\dashv S_1 \mapsto [] &&, S_l.[\delta] \\
[] &\leftarrow\!\!\dashv S_2 \mapsto [\delta].S_l &&, []
\end{aligned}$$

Result:

$S_i = \textbf{sort}(S_i) \quad i \in \{\_, 0, 1, 2\}$
$S = S_1.S_0.S_2$

$S = \textbf{sort}(S)$
$S_1 = S_2 = []$
$\wedge \alpha = \alpha_c = \alpha_t \neq \texttt{0x0}$

$\bigsqcup_\Sigma$

$S = \textbf{sort}(S) \wedge S = S_0.S_2$
$S_i = \textbf{sort}(S_i) \quad i \in \{l, r, 1, 2\}$
$S_0 = S_l \wedge S_1 = [] \wedge S_2 = [\delta].S_r$
$\wedge \alpha = \alpha_t \neq \texttt{0x0}$
$\wedge \alpha_l = \alpha_c \neq \texttt{0x0}$
$\wedge \texttt{v} < \delta = \textbf{min}_{S_2} \wedge \textbf{max}_{S_1} = -\infty$
$\textbf{max}_{S_l} \leqslant \delta \leqslant \textbf{min}_{S_r}$

$S = \textbf{sort}(S) \wedge S = S_1.S_0$
$S_i = \textbf{sort}(S_i) \quad i \in \{l, r, 1, 2\}$
$S_0 = S_r \wedge S_1 = S_l.[\delta] \wedge S_2 = []$
$\wedge \alpha = \alpha_t \neq \texttt{0x0}$
$\wedge \alpha_r = \alpha_c \neq \texttt{0x0}$
$\wedge \texttt{v} \geqslant \delta = \textbf{max}_{S_1} \wedge \textbf{min}_{S_2} = +\infty$
$\textbf{max}_{S_l} \leqslant \delta \leqslant \textbf{min}_{S_r}$

Union (Numerical part)

$$
\begin{array}{ll}
\alpha \hookleftarrow \alpha_c \mapsto \alpha_l & , \alpha_r \\
S \hookleftarrow S_0 \mapsto S_l & , S_r \\
\alpha \hookleftarrow \alpha_t \mapsto \alpha & , \alpha \\
[] \hookleftarrow S_1 \mapsto [] & , S_l.[\delta] \\
[] \hookleftarrow S_2 \mapsto [\delta].S_l & , []
\end{array}
$$

Result:

$S_i = \textbf{sort}(S_i) \quad i \in \{\_, 0, 1, 2\}$
$S = S_1.S_0.S_2$
$\alpha_c, \alpha_t \neq \mathtt{0x0}$

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

$\bigsqcup_\Sigma$

$S = \textbf{sort}(S) \land S = S_0.S_2$
$S_i = \textbf{sort}(S_i) \quad i \in \{l, r, 1, 2\}$
$S_0 = S_l \land S_1 = [] \land S_2 = [\delta].S_r$
$\land \alpha = \alpha_t \neq \mathtt{0x0}$
$\land \alpha_l = \alpha_c \neq \mathtt{0x0}$
$\land \mathtt{v} < \delta = \textbf{min}_{S_2} \land \textbf{max}_{S_1} = -\infty$
$\textbf{max}_{S_l} \leqslant \delta \leqslant \textbf{min}_{S_r}$

$S = \textbf{sort}(S) \land S = S_1.S_0$
$S_i = \textbf{sort}(S_i) \quad i \in \{l, r, 1, 2\}$
$S_0 = S_r \land S_1 = S_l.[\delta] \land S_2 = []$
$\land \alpha = \alpha_t \neq \mathtt{0x0}$
$\land \alpha_r = \alpha_c \neq \mathtt{0x0}$
$\land \mathtt{v} \geqslant \delta = \textbf{max}_{S_1} \land \textbf{min}_{S_2} = +\infty$
$\textbf{max}_{S_l} \leqslant \delta \leqslant \textbf{min}_{S_r}$

Union (Numerical part)

$$\alpha \leftarrow\!\!\dashv \alpha_c \mapsto \alpha_l \quad , \alpha_r$$
$$S \leftarrow\!\!\dashv S_0 \mapsto S_l \quad , S_r$$
$$\alpha \leftarrow\!\!\dashv \alpha_t \mapsto \alpha \quad , \alpha$$
$$[] \leftarrow\!\!\dashv S_1 \mapsto [] \quad , S_l.[\delta]$$
$$[] \leftarrow\!\!\dashv S_2 \mapsto [\delta].S_l \, , []$$

Result:

$S_i = \textbf{sort}(S_i) \quad i \in \{\_, 0, 1, 2\}$
$S = S_1.S_0.S_2$
$\alpha_c, \alpha_t \neq 0x0$
$\textbf{max}_{S_1} \leqslant v \leqslant \textbf{min}_{S_2}$

$S = \textbf{sort}(S)$
$S_1 = S_2 = []$
$\wedge \alpha = \alpha_c = \alpha_t \neq 0x0$

$\bigsqcup_{\Sigma}$

$S = \textbf{sort}(S) \wedge S = S_0.S_2$
$S_i = \textbf{sort}(S_i) \quad i \in \{l, r, 1, 2\}$
$S_0 = S_l \wedge S_1 = [] \wedge S_2 = [\delta].S_r$
$\wedge \alpha = \alpha_t \neq 0x0$
$\wedge \alpha_l = \alpha_c \neq 0x0$
$\wedge v < \delta = \textbf{min}_{S_2} \wedge \textbf{max}_{S_1} = -\infty$
$\textbf{max}_{S_l} \leqslant \delta \leqslant \textbf{min}_{S_r}$

$S = \textbf{sort}(S) \wedge S = S_1.S_0$
$S_i = \textbf{sort}(S_i) \quad i \in \{l, r, 1, 2\}$
$S_0 = S_r \wedge S_1 = S_l.[\delta] \wedge S_2 = []$
$\wedge \alpha = \alpha_t \neq 0x0$
$\wedge \alpha_r = \alpha_c \neq 0x0$
$\wedge v \geqslant \delta = \textbf{max}_{S_1} \wedge \textbf{min}_{S_2} = +\infty$
$\textbf{max}_{S_l} \leqslant \delta \leqslant \textbf{min}_{S_r}$

Exemple: insertion in binary search tree

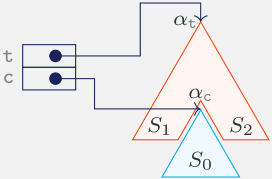
$$\left\{\begin{array}{l}\textsf{tree}(\mathtt{t}, S) \\ \wedge\, S = \textsf{sort}(S)\end{array}\right\}$$

```
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;
}
```

$$\{\textsf{tree}(\mathtt{t}, \textsf{sort}(S.[\mathtt{v}]))\}$$

$S_i = \textsf{sort}(S_i) \quad i \in \{\_, 1, 2 \qquad \}$
$S = S_1.S_0.S_2$

$\alpha_{\mathtt{c}}, \alpha_{\mathtt{t}} \qquad \neq \mathsf{0x0}$
$\textsf{max}_{S_1} \leqslant \mathtt{v} \leqslant \textsf{min}_{S_2}$

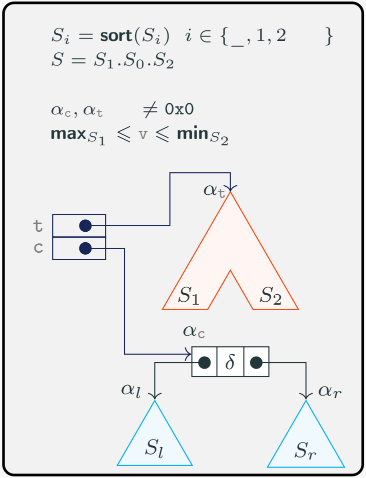


Exemple: insertion in binary search tree

$$\left\{\begin{array}{l}\textbf{tree}(\mathtt{t}, S)\\ \wedge\, S = \textbf{sort}(S)\end{array}\right\}$$

```
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;
}
```

$$\{\textbf{tree}(\mathtt{t}, \textbf{sort}(S.[\mathtt{v}]))\}$$



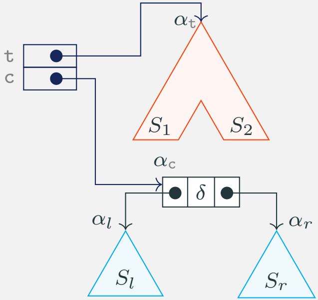
Exemple: insertion in binary search tree

$$\left\{\begin{array}{l}\textbf{tree}(\mathtt{t}, S) \\ \wedge S = \textbf{sort}(S)\end{array}\right\}$$

```
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;
}
```

$$\{\textbf{tree}(\mathtt{t}, \textbf{sort}(S.[\mathtt{v}]))\}$$

$S_i = \textbf{sort}(S_i) \quad i \in \{\_, 1, 2, l, r\}$
$S = S_1.S_0.S_2$
$S_0 = S_l.[\delta].S_r$
$\alpha_{\mathtt{c}}, \alpha_{\mathtt{t}}, \alpha_l \neq \mathtt{0x0}$
$\textbf{max}_{S_1} \leqslant \mathtt{v} \leqslant \textbf{min}_{S_2}$
$\mathtt{v} < \delta$



Exemple: insertion in binary search tree

$$\left\{\begin{array}{l}\textbf{tree}(\mathtt{t}, S)\\ \wedge S = \textbf{sort}(S)\end{array}\right\}$$

```
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;
}
```

$\{\textbf{tree}(\mathtt{t}, \textbf{sort}(S.[\mathtt{v}]))\}$

$S_i = \textbf{sort}(S_i) \quad i \in \{\_, 1, 2, l, r\}$

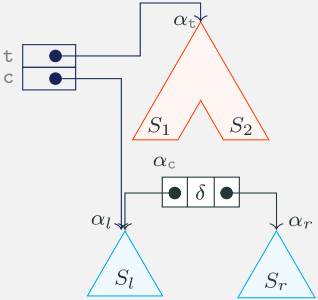
$S = S_1.S_0.S_2$

$S_0 = S_l.[\delta].S_r$

$\alpha_{\mathtt{c}}, \alpha_{\mathtt{t}}, \alpha_l \neq \mathtt{0x0}$

$\textbf{max}_{S_1} \leqslant \mathtt{v} \leqslant \textbf{min}_{S_2}$

$\mathtt{v} < \delta$



Exemple: insertion in binary search tree

$S_i = \mathbf{sort}(S_i) \quad i \in \{\_, 1, 2 \quad\}$

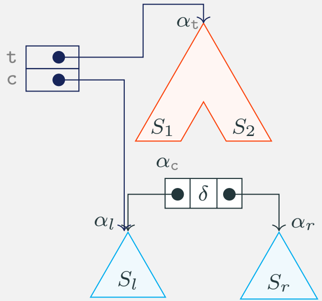
$S = S_1 . S_0 . S_2$

$S_0 = S_l . [\delta] . S_r$

$\alpha_{\mathtt{c}}, \alpha_{\mathtt{t}} \quad \neq \mathtt{0x0}$

$\mathbf{max}_{S_1} \leqslant \mathtt{v} \leqslant \mathbf{min}_{S_2}$

$\mathtt{v} < \delta$



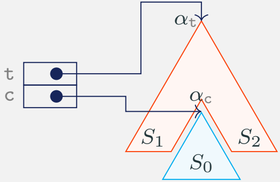
$\sqsubseteq_\Sigma$

$S_i = \mathbf{sort}(S_i) \quad i \in \{\_, 1, 2\}$

$S = S_1 . S_0 . S_2$

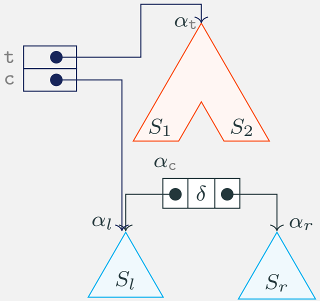
$\alpha_{\mathtt{c}}, \alpha_{\mathtt{t}} \neq \mathtt{0x0}$

$\mathbf{max}_{S_1} \leqslant \mathtt{v} \leqslant \mathbf{min}_{S_2}$



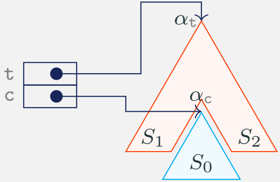
Exemple: insertion in binary search tree

$S_i = \textbf{sort}(S_i) \quad i \in \{\_, 1, 2 \}$
$S = S_1.S_0.S_2$
$S_0 = S_l.[\delta].S_r$
$\alpha_c, \alpha_t \neq 0x0$
$\textbf{max}_{S_1} \leqslant v \leqslant \textbf{min}_{S_2}$
$v < \delta$



$\sqsubseteq_\Sigma$

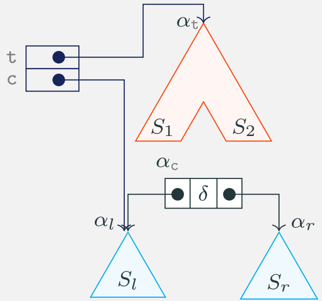
$S_i = \textbf{sort}(S_i) \quad i \in \{\_, 1, 2\}$
$S = S_1.S_0.S_2$
$\alpha_c, \alpha_t \neq 0x0$
$\textbf{max}_{S_1} \leqslant v \leqslant \textbf{min}_{S_2}$



$\Longrightarrow$ Invariant found after two iterations !

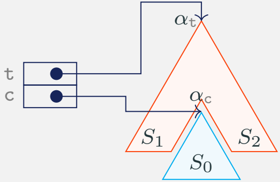
Exemple: insertion in binary search tree

$S_i = \textbf{sort}(S_i) \quad i \in \{\_, 1, 2, l, r\}$
$S = S_1.S_0.S_2$
$S_0 = S_l.[\delta].S_r$
$\alpha_c, \alpha_t, \alpha_l \neq \texttt{0x0}$
$\textbf{max}_{S_1} \leqslant \texttt{v} \leqslant \textbf{min}_{S_2}$
$\texttt{v} < \delta$



$\sqsubseteq_\Sigma$

$S_i = \textbf{sort}(S_i) \quad i \in \{\_, 1, 2\}$
$S = S_1.S_0.S_2$
$\alpha_c, \alpha_t \neq \texttt{0x0}$
$\textbf{max}_{S_1} \leqslant \texttt{v} \leqslant \textbf{min}_{S_2}$



$\Longrightarrow$ Invariant found after two iterations !