Automatic Verification of Tasks Schedulers

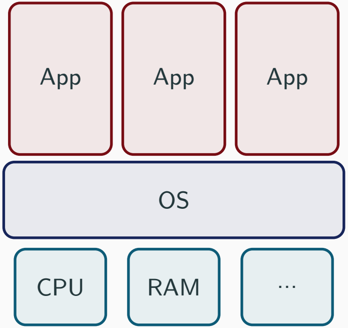
Ph.D. defense

Josselin Giet[1]

September 26, 2024

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

The importance of OSes

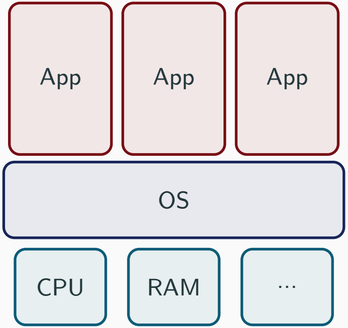


Operating systems fulfill two missions:

- Provide an execution environment for user applications
  abstracts the hardware (CPU, memory, device driver)
- Manages resources on the behalf of user applications
  Example of resources: memory usage, CPU time
  The OS decides which application can access which resource

A failure at the OS level may impact all applications.
In some cases, the whole computer is unusable
(*e.g.* CrowdStrike/Windows)

The importance of OSes



Operating systems fulfill two missions:

- Provide an execution environment for user applications
  abstracts the hardware (CPU, memory, device driver)
- Manages resources on the behalf of user applications
  Example of resources: memory usage, CPU time
  The OS decides which application can access which resource

A failure at the OS level may impact all applications.
In some cases, the whole computer is unusable
(*e.g.* CrowdStrike/Windows)

Question: How to gain higher trust in OSes?

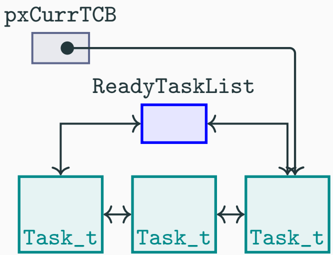
Our Case study: Scheduler of FreeRTOS

FreeRTOS is a small, free, mature, industrial, and highly customizable real-time OS.

Our Case study: Scheduler of FreeRTOS

FreeRTOS is a small, free, mature, industrial, and highly customizable real-time OS.

Tasks in the FreeRTOS kernel can be in two states:



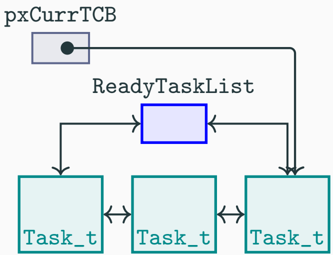
Tasks in the **ready** state:

- are stored in `pxReadyTasksList`,
- contain the **running** task pointed by `pxCurrentTCB`.

Our Case study: Scheduler of FreeRTOS

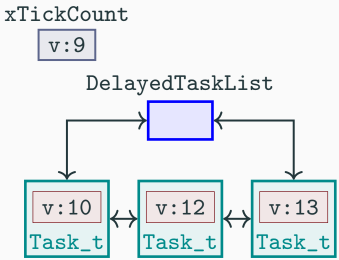
FreeRTOS is a small, free, mature, industrial, and highly customizable real-time OS.

Tasks in the FreeRTOS kernel can be in two states:



Tasks in the **ready** state:

- are stored in `pxReadyTasksList`,
- contain the **running** task pointed by `pxCurrentTCB`.



Tasks in the **delayed** state:

- are stored in `pxDelayedTaskList`,
- sorted according to the end of their delay,
- which are greater that the **tick** value, stored in `xTickCount`

What kind of properties do we attempt to prove?

1. Absence of Run-time error
   All pointer dereference are correct.

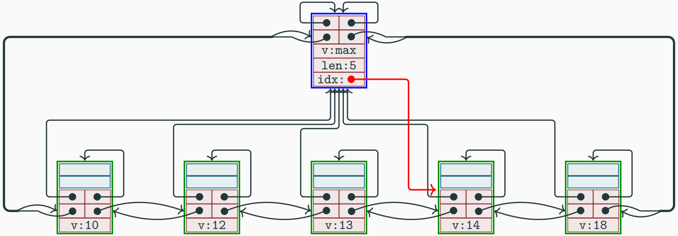
What kind of properties do we attempt to prove?

1. Absence of Run-time error

   All pointer dereference are correct.

2. Preservation of structural invariants

   Manipulation of the doubly-linked lists maintain the invariants.



What kind of properties do we attempt to prove?

1. Absence of Run-time error
   All pointer dereference are correct.
2. Preservation of structural invariants
   Manipulation of the doubly-linked lists maintain the invariants.
3. Preservation of functional invariants of the scheduler
   The list of delayed tasks is sorted.

What kind of properties do we attempt to prove?

1. Absence of Run-time error
   All pointer dereference are correct.
2. Preservation of structural invariants
   Manipulation of the doubly-linked lists maintain the invariants.
3. Preservation of functional invariants of the scheduler
   The list of delayed tasks is sorted.
4. Partial functional correctness
   No task shall stay in the delayed state, after its delay expires.

What kind of properties do we attempt to prove?

1. Absence of Run-time error
   All pointer dereference are correct.
2. Preservation of structural invariants
   Manipulation of the doubly-linked lists maintain the invariants.
3. Preservation of functional invariants of the scheduler
   The list of delayed tasks is sorted.
4. Partial functional correctness
   No task shall stay in the delayed state, after its delay expires.
5. Termination
   All systems call should eventually end.

What kind of properties do we attempt to prove?

1. Absence of Run-time error
   All pointer dereference are correct.
2. Preservation of structural invariants
   Manipulation of the doubly-linked lists maintain the invariants.
3. Preservation of functional invariants of the scheduler
   The list of delayed tasks is sorted.
4. Partial functional correctness
   No task shall stay in the delayed state, after its delay expires.
5. Termination
   All systems call should eventually end.
6. Concurrency related properties
   Interruptions must not cause race-conditions.

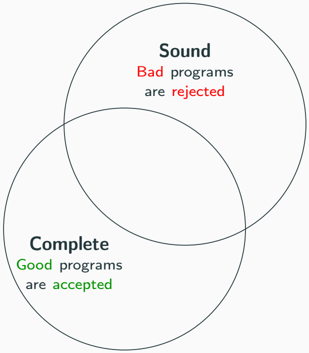
Methods to verify programs

We can classify verification methods:

Methods to verify programs

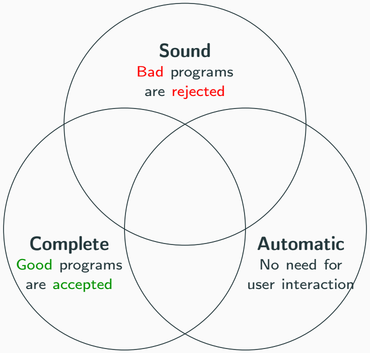
We can classify verification methods:

We can classify verification methods:



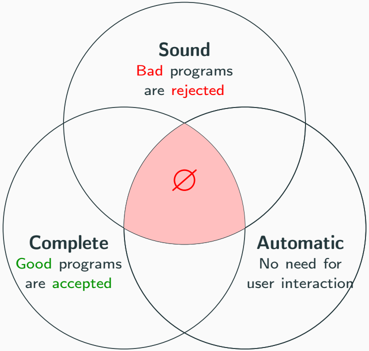
Methods to verify programs

We can classify verification methods:



Methods to verify programs

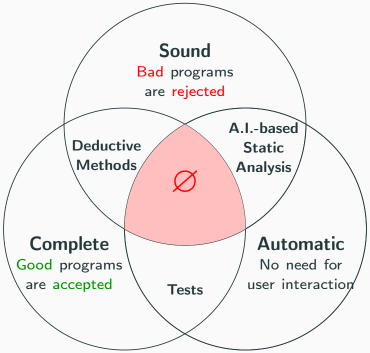
We can classify verification methods:



Theorem: Rice's theorem

Any *non-trivial semantic property* is non-decidable.

We can classify verification methods:



Theorem: Rice's theorem

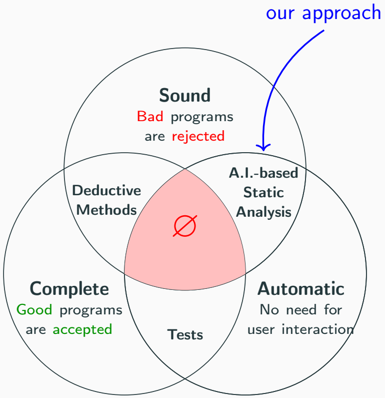
Any *non-trivial semantic property* is non-decidable.

Any sound verification method must be either:

- Limited to non-Turing complete programs
  bounded loops and memories
  Example: Serval
- Non-automatic (proof assistants/external solvers)
  Expensive proof burden
  Example: seL4
- Non-Complete
  Example: Static analysis by abstract interpretation
  limited expressiveness: absence of run-time error.
  Preservation of structural invariants

Methods to verify programs

We can classify verification methods:



Theorem: Rice's theorem

Any *non-trivial semantic property* is non-decidable.

Any sound verification method must be either:

- Limited to non-Turing complete programs

  bounded loops and memories

  Example: Serval

- Non-automatic (proof assistants/external solvers)

  Expensive proof burden

  Example: seL4

- Non-Complete

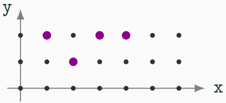
  Example: Static analysis by abstract interpretation

  limited expressiveness: absence of run-time error.

  Preservation of structural invariants

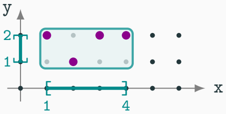
An **abstract domain** provides:

- An efficient representation of over-approximation of set of states $\wp\left(\mathbb{Z}^2\right)$



An **abstract domain** provides:

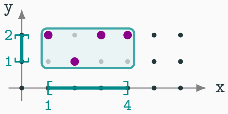
- An efficient representation of over-approximation of set of states $\wp\left(\mathbb{Z}^2\right) \xleftarrow{\gamma} (\mathbb{Z} \cup \{\pm\infty\})^4$



$$\gamma \begin{pmatrix} \mathrm{x} : (1, 4) \\ \mathrm{y} : (1, 2) \end{pmatrix} := \left\{ (x, y) \in \mathbb{Z}^2 \middle| \begin{array}{r} 1 \leqslant x \leqslant 4 \\ \wedge\, 1 \leqslant y \leqslant 2 \end{array} \right\}$$

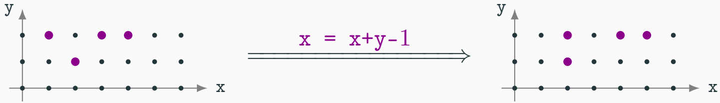
An **abstract domain** provides:

- An efficient representation of over-approximation of set of states $\wp\left(\mathbb{Z}^2\right) \xleftarrow{\gamma} (\mathbb{Z} \cup \{\pm\infty\})^4$



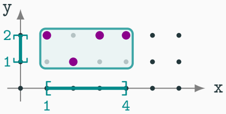
$$\gamma\begin{pmatrix} \mathrm{x} : (1, 4) \\ \mathrm{y} : (1, 2) \end{pmatrix} := \left\{ (x, y) \in \mathbb{Z}^2 \middle| \begin{array}{r} 1 \leqslant x \leqslant 4 \\ \wedge\, 1 \leqslant y \leqslant 2 \end{array} \right\}$$

- Operators that over-approximate the behaviors of the program



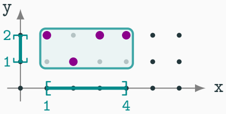
An **abstract domain** provides:

- An efficient representation of over-approximation of set of states $\wp\left(\mathbb{Z}^2\right) \xleftarrow{\gamma} \left(\mathbb{Z} \cup \{\pm\infty\}\right)^4$

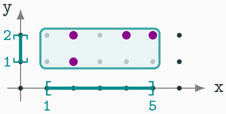


$$\gamma\begin{pmatrix} \mathtt{x} : (1,4) \\ \mathtt{y} : (1,2) \end{pmatrix} := \left\{ (x,y) \in \mathbb{Z}^2 \,\middle|\, \begin{array}{r} 1 \leqslant x \leqslant 4 \\ \wedge\, 1 \leqslant y \leqslant 2 \end{array} \right\}$$

- Operators that over-approximate the behaviors of the program

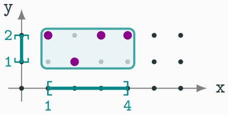


$$\xrightarrow[\textsf{assign}^\sharp_n(\mathtt{x},\ \mathtt{x+y-1})]{\mathtt{x\ =\ x+y-1}}$$



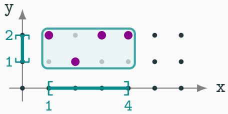
An **abstract domain** provides:

- An efficient representation of over-approximation of set of states $\wp\left(\mathbb{Z}^2\right) \xleftarrow{\gamma} \left(\mathbb{Z} \cup \{\pm\infty\}\right)^4$

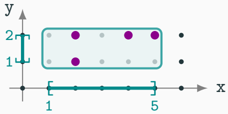


$$\gamma\begin{pmatrix}\mathtt{x} : (1,4) \\ \mathtt{y} : (1,2)\end{pmatrix} := \left\{ (x,y) \in \mathbb{Z}^2 \,\middle|\, \begin{array}{r} 1 \leqslant x \leqslant 4 \\ \wedge\, 1 \leqslant y \leqslant 2 \end{array} \right\}$$

- Operators that over-approximate the behaviors of the program



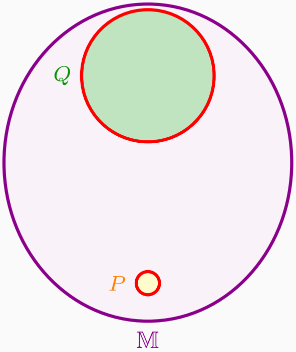
$$\xRightarrow[\textbf{assign}^\sharp_n(\mathtt{x},\ \mathtt{x+y-1})]{\mathtt{x\ =\ x+y-1}}$$



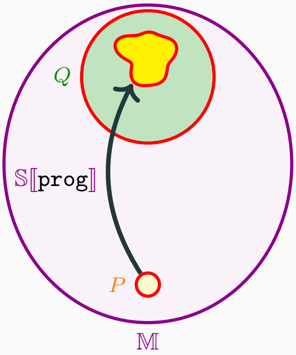
Using these operators, we define the **abstract semantics**:

$$\mathbb{S}[\![l = e]\!]^\sharp_n(\sigma^\sharp) := \textbf{assign}^\sharp_n(l, e, \sigma^\sharp)$$
$$\mathbb{S}[\![s; s']\!]^\sharp_n(\sigma^\sharp) := \left(\mathbb{S}[\![s']\!]^\sharp_n \circ \mathbb{S}[\![s]\!]^\sharp_n\right)(\sigma^\sharp)$$
$$\mathbb{S}[\![\mathtt{if}(b)\{s\}\mathtt{else}\{s'\}]\!]^\sharp_n(\sigma^\sharp) := \left(\mathbb{S}[\![s]\!]^\sharp_n \circ \textbf{guard}^\sharp_n(b, \sigma^\sharp)\right) \sqcup^\sharp_n \left(\mathbb{S}[\![s']\!]^\sharp_n \circ \textbf{guard}^\sharp_n(\neg b, \sigma^\sharp)\right)$$
$$\ldots$$

How to prove $\{P\}\texttt{prog}\{Q\}$ by abstract interpretation?

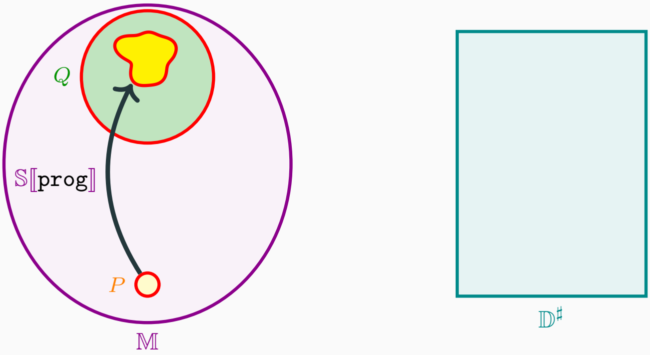


How to prove $\{P\}\texttt{prog}\{Q\}$ by abstract interpretation?



Problem: $\mathbb{S}[\![\texttt{prog}]\!]$ cannot be computed

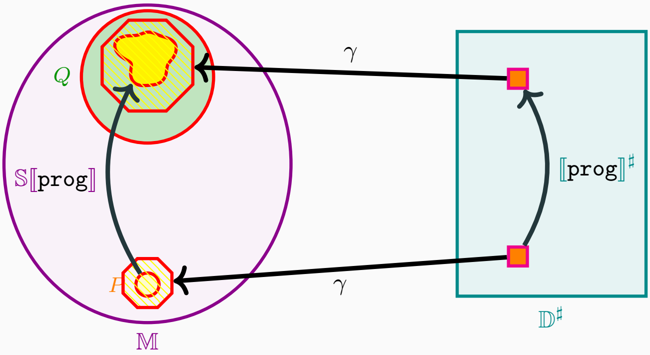
How to prove $\{P\}\texttt{prog}\{Q\}$ by abstract interpretation?



Problem: $\mathbb{S}[\![\texttt{prog}]\!]$ cannot be computed

Solution: Compute an over-approximation using an **abstract domain** $\mathbb{D}^\sharp$

How to prove $\{P\}\texttt{prog}\{Q\}$ by abstract interpretation?



Problem: $\mathbb{S}[\![\texttt{prog}]\!]$ cannot be computed

Solution: Compute an over-approximation using an **abstract domain** $\mathbb{D}^\sharp$

The program is proved by two inclusions:

Established by distinct arguments

1. $\mathbb{S}[\![\texttt{prog}]\!](P) \subseteq$ (octagon) **Correctness by construction**
2. (octagon) $\subseteq Q$ **Result** (checked by the analysis)

Comparison of existing static analyses

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

Analysis	pointer dereference	structural invariants	partial f^{al} correctness	
			SLL	others
[Emami *et al.*, PLDI, 94]	✓	✗	✗	✗
Pointer analysis				
[Sagiv *et al.*, TOPLAS, 99]	✓	✓	✗	✗
Shape analysis based on 3-value logic				
[Chang *et al.*, POPL, 08]	✓	✓	✗	✗
Shape analysis based on separation logic				
[Bouajjani *et al.*, CAV, 10]	✓	✓	✓	✗
Shape analysis based on k-limited graphs combined with decidable array logic				

Comparison of existing static analyses

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

Analysis	pointer dereference	structural invariants	partial f^al correctness	
			SLL	others
[Emami *et al.*, PLDI, 94] Pointer analysis	✓	✗	✗	✗
[Sagiv *et al.*, TOPLAS, 99] Shape analysis based on 3-value logic	✓	✓	✗	✗
[Chang *et al.*, POPL, 08] Shape analysis based on separation logic	✓	✓	✗	✗
[Bouajjani *et al.*, CAV, 10] Shape analysis based on k-limited graphs combined with decidable array logic	✓	✓	✓	✗

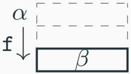
How to improve the expressiveness of static analysis to automatically prove partial functional correctness of task schedulers?

Separation logic-based shape analysis

[Chang *et al.* POPL, 08] uses a subset of separation logic to summarize memory states:

- **points-to predicate** denotes a single memory cell

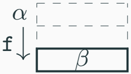
  Example: $\alpha.\mathtt{f} \mapsto \beta$ correspond to the memory containing one cell:



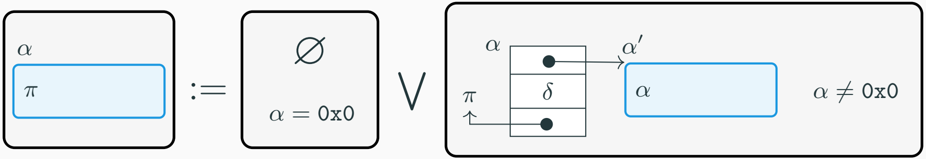
Separation logic-based shape analysis

[Chang *et al.* POPL, 08] uses a subset of separation logic to summarize memory states:

- **points-to predicate** denotes a single memory cell
  Example: $\alpha.\mathtt{f} \mapsto \beta$ correspond to the memory containing one cell:

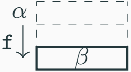


- **Inductive predicate** denotes an unbounded dynamic data structure
  Example: list(α, π) denotes a dll starting at address α where the previous node is at address π.

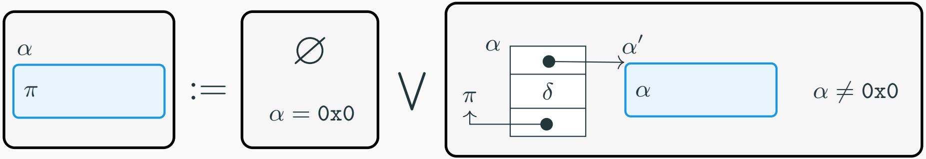


[Chang *et al.* POPL, 08] uses a subset of separation logic to summarize memory states:

- **points-to predicate** denotes a single memory cell
  Example: $\alpha.\mathtt{f} \mapsto \beta$ correspond to the memory containing one cell:



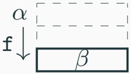
- **Inductive predicate** denotes an unbounded dynamic data structure
  Example: list(α, π) denotes a dll starting at address α where the previous node is at address π.



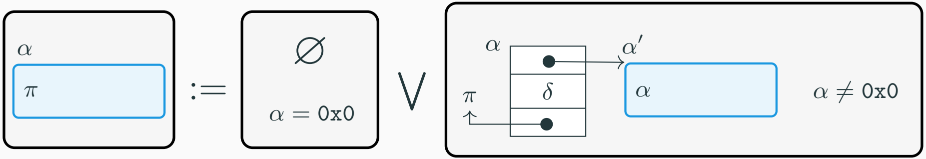
- **The separating conjunction** $*$ binds these predicates.
  $m_1^\sharp * m_2^\sharp$ means that the memories described by $m_1^\sharp$ and $m_2^\sharp$ are disjoint.

[Chang *et al.* POPL, 08] uses a subset of separation logic to summarize memory states:

- **points-to predicate** denotes a single memory cell
  Example: $\alpha.\mathtt{f} \mapsto \beta$ correspond to the memory containing one cell:



- **Inductive predicate** denotes an unbounded dynamic data structure
  Example: list(α, π) denotes a dll starting at address α where the previous node is at address π.

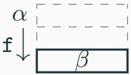


- **The separating conjunction** $*$ binds these predicates.
  $m_1^\sharp * m_2^\sharp$ means that the memories described by $m_1^\sharp$ and $m_2^\sharp$ are disjoint.

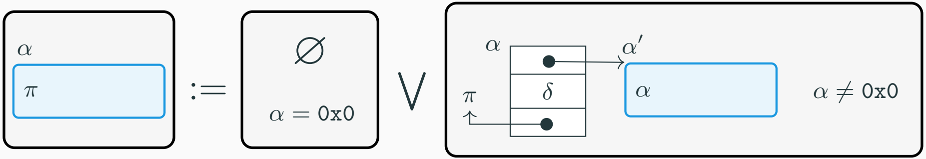
Separation logic-based shape analysis

[Chang *et al.* POPL, 08] uses a subset of separation logic to summarize memory states:

- **points-to predicate** denotes a single memory cell
  Example: $\alpha.\mathtt{f} \mapsto \beta$ correspond to the memory containing one cell:

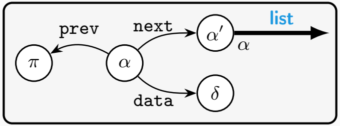


- **Inductive predicate** denotes an unbounded dynamic data structure
  Example: list(α, π) denotes a dll starting at address α where the previous node is at address π.

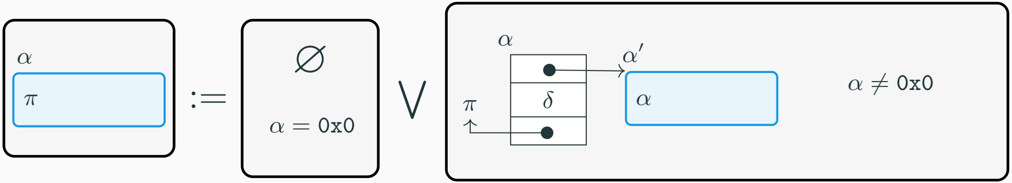


- **The separating conjunction** $*$ binds these predicates.
  $m_1^\sharp * m_2^\sharp$ means that the memories described by $m_1^\sharp$ and $m_2^\sharp$ are disjoint.

These predicates are manipulated using a **graph representation**

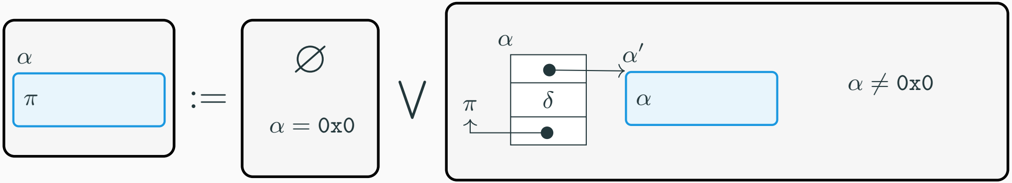


We have to improve the level of expressiveness



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

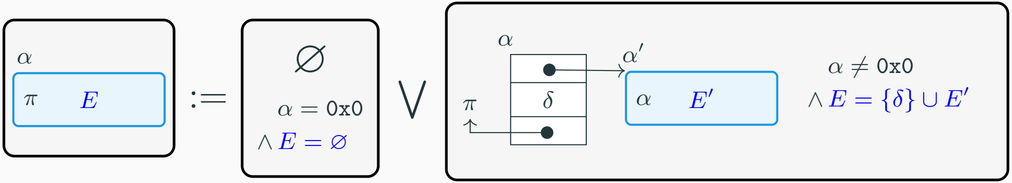
We have to improve the level of expressiveness



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

Problem: It is not enough for partial functional correctness: **list** forgets the content !

We have to improve the level of expressiveness



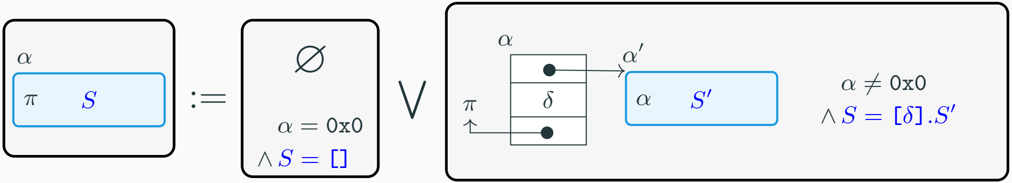
$\Longrightarrow$ This predicate is expressive enough to prove memory safety & structure preservation.

Problem: It is not enough for partial functional correctness: **list** 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 !

We have to improve the level of expressiveness



$\Longrightarrow$ This predicate is expressive enough to prove memory safety & structure preservation.
Problem: It is not enough for partial functional correctness: **list** 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 !

Our solution: Express constraints on the sequence of values stored in the list.
Add a **sequence parameter** to the inductive predicate: **list**(α, π, S).

Example: `res = xTaskIncrementTick()`

Pre

Ready

(R_v)

Delayed

$(D_v^{exp}.D_v^{nexp})$

$D_v^{exp}.D_v^{nexp}$ is sorted
$\wedge \max(D_v^{exp}) < \texttt{xTick} + 1 \leqslant \min(D_v^{nexp})$

Post

Ready

$(D_v^{exp}.R_v)$

Delayed

(D_v^{nexp})

$$\texttt{xTick} = \texttt{xTick@old} + 1$$
$$\wedge\, \texttt{res} = \begin{cases} 1 & \text{if } |D_v^{exp}| + |R_v| > 1 \\ 0 & \text{otherwise} \end{cases}$$

Example: `res = xTaskIncrementTick()`

Pre

Ready: (R_v)

Delayed: $(D_v^{exp}.D_v^{nexp})$

$D_v^{exp}.D_v^{nexp}$ is sorted
$\wedge \max(D_v^{exp}) < \texttt{xTick} + 1 \leqslant \min(D_v^{nexp})$

Post

Ready: $(D_v^{exp}.R_v)$

Delayed: (D_v^{nexp})

$\texttt{xTick} = \texttt{xTick@old} + 1$
$\wedge\, \texttt{res} = \begin{cases} 1 & \text{if } |D_v^{exp}| + |R_v| > 1 \\ 0 & \text{otherwise} \end{cases}$

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

Verification of an instance of FreeRTOS

Specification and analysis of real-time constraints

An abstract domain reasoning over sequences

Domain description

At a high level, an **abstract value** $\sigma_s^\sharp$ of the sequence abstract domain $\mathbb{D}_s^\sharp$ consists of:

$$\left(\begin{array}{l} S = S_1.[\delta] \\ \wedge\, S = S_2 \\ \wedge\, S = \textbf{sort}(S) \\ \wedge\, S_1 = \textbf{sort}(S_3) \end{array} \quad \begin{array}{l} \texttt{mset}_S = \{\!\{\, \delta \,\}\!\} \uplus \texttt{mset}_{S_1} \\ \wedge\, \texttt{mset}_S = \texttt{mset}_{S_2} \\ \wedge\, \texttt{mset}_{S_1} = \texttt{mset}_{S_3} \end{array} \quad \begin{array}{l} \texttt{min}_S \leqslant \delta \leqslant \texttt{max}_S \\ \wedge\, \texttt{max}_{S_1} \leqslant \delta \\ \quad \ldots \\ \wedge\, \texttt{len}_S = 1 + \texttt{len}_{S_1} \end{array}\right)$$

Domain description

At a high level, an **abstract value** $\sigma_s^\sharp$ of the sequence abstract domain $\mathbb{D}_s^\sharp$ consists of:

$$\left(\begin{array}{l} S = S_1 . [\delta] \\ \wedge\, S = S_2 \\ \wedge\, S = \textbf{sort}(S) \\ \wedge\, S_1 = \textbf{sort}(S_3) \end{array} \quad \begin{array}{l} \texttt{mset}_S = \{\!\{\, \delta \,\}\!\} \uplus \texttt{mset}_{S_1} \\ \wedge\, \texttt{mset}_S = \texttt{mset}_{S_2} \\ \wedge\, \texttt{mset}_{S_1} = \texttt{mset}_{S_3} \end{array} \quad \begin{array}{l} \texttt{min}_S \leqslant \delta \leqslant \texttt{max}_S \\ \wedge\, \texttt{max}_{S_1} \leqslant \delta \\ \quad \ldots \\ \wedge\, \texttt{len}_S = 1 + \texttt{len}_{S_1} \end{array}\right)$$

$\texttt{U-F} : S \sim S_2$
$\texttt{Sorted} : S, S_1, S_2$
$\ldots$

An element of a multiset domain $\mathbb{D}_{ms}^\sharp$

An element of a numerical domain $\mathbb{D}_n^\sharp$

Domain description

At a high level, an **abstract value** $\sigma_s^\sharp$ of the sequence abstract domain $\mathbb{D}_s^\sharp$ consists of:

$$\left(\begin{array}{l} S = S_1 . [\delta] \\ \wedge\, S = S_2 \\ \wedge\, S = \textbf{sort}(S) \\ \wedge\, S_1 = \textbf{sort}(S_3) \end{array} \quad \begin{array}{l} \texttt{mset}_S = \{\!\{ \delta \}\!\} \uplus \texttt{mset}_{S_1} \\ \wedge\, \texttt{mset}_S = \texttt{mset}_{S_2} \\ \wedge\, \texttt{mset}_{S_1} = \texttt{mset}_{S_3} \end{array} \quad \begin{array}{l} \texttt{min}_S \leqslant \delta \leqslant \texttt{max}_S \\ \wedge\, \texttt{max}_{S_1} \leqslant \delta \\ \quad \dots \\ \wedge\, \texttt{len}_S = 1 + \texttt{len}_{S_1} \end{array}\right)$$

$\texttt{U-F}: S \sim S_2$
$\texttt{Sorted}: S, S_1, S_2$
$\dots$

An element of a multiset domain $\mathbb{D}_{ms}^\sharp$

An element of a numerical domain $\mathbb{D}_n^\sharp$

The **concretization**, $\gamma_s(\sigma_s^\sharp)$ is the set of valuation functions that satisfy the constraints expressed by $\sigma_s^\sharp$:

Example: $\left\{ \begin{array}{c} \alpha \mapsto 2 \\ \delta \mapsto 8 \end{array} \right\} \left\{ \begin{array}{c} S, S_2 \mapsto 4;6;8 \\ S_1 \mapsto 4;6 \\ S_3 \mapsto 6;4 \end{array} \right\}$

Adding a new constraint

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

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

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

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

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

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

Adding a new constraint

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

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

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

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

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

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

1. add the new definition in the conjunction

Adding a new constraint

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

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

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

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

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

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

1. add the new definition in the conjunction
2. fold other definitions

Adding a new constraint

guard$^\sharp_s$: $\mathbb{D}^\sharp_s \to$ seq. constraint $\to \mathbb{D}^\sharp_s$

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

$$\land\, \mathtt{mset}_S = \{\!\{\, \alpha \,\}\!\} \uplus \mathtt{mset}_{S_1}$$

$$\land\, \mathtt{len}_S = 1 + \mathtt{len}_{S_1} + \mathtt{len}_{S_2}$$

$$\land\, \mathtt{min}_S \leqslant \alpha \leqslant \mathtt{max}_S$$
$$\land\, \mathtt{min}_S \leqslant \mathtt{min}_{S_1} \land \mathtt{max}_{S_1} \leqslant \mathtt{max}_S$$

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

1. add the new definition in the conjunction
2. fold other definitions
3. detect & remove cyclic constraints

Adding a new constraint

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

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

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

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

Adding a new constraint

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

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

$$\wedge \mathtt{mset}_S = \{\!\{ \alpha \}\!\} \uplus \mathtt{mset}_{S_1}$$
$$\wedge\ \mathtt{mset}_{S_r} = \{\!\{ \alpha \}\!\}$$

$$\wedge \mathtt{len}_S = 1 + \mathtt{len}_{S_1} + \mathtt{len}_{S_2}$$
$$\wedge\ \mathtt{len}_{S_r} = 1$$

$$\wedge \mathtt{min}_S \leqslant \alpha \leqslant \mathtt{max}_S$$
$$\wedge \mathtt{min}_S \leqslant \mathtt{min}_{S_1} \wedge \mathtt{max}_{S_1} \leqslant \mathtt{max}_S$$
$$\wedge\ \mathtt{min}_{S_r} = \alpha = \mathtt{max}_{S_r}$$

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

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

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

Adding a new constraint

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

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

$$\wedge \texttt{mset}_S = \{\!\{ \alpha \}\!\} \uplus \texttt{mset}_{S_1}$$
$$\wedge \texttt{mset}_{S_r} = \{\!\{ \alpha \}\!\}$$

$$\wedge \texttt{len}_S = 1 + \texttt{len}_{S_1} + \texttt{len}_{S_2}$$
$$\wedge \texttt{len}_{S_r} = 1$$

$$\wedge \texttt{min}_S \leqslant \alpha \leqslant \texttt{max}_S$$
$$\wedge \texttt{min}_S \leqslant \texttt{min}_{S_1} \wedge \texttt{max}_{S_1} \leqslant \texttt{max}_S$$
$$\wedge \texttt{min}_{S_r} = \alpha = \texttt{max}_{S_r}$$

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

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

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

Theorem: Soundness of $\textbf{guard}^\sharp_s$

$\textbf{guard}^\sharp_s(\sigma^\sharp_s, S = E)$ over-approximates all valuations summarized by $\sigma^\sharp_s$ satisfying $S = E$.

Abstract operators

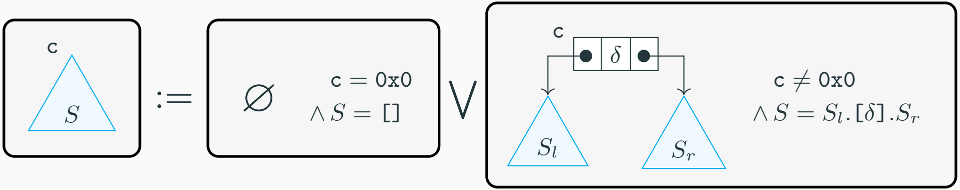
- $\textbf{sat}^\sharp_s : \mathbb{D}^\sharp_s \to \text{seq constraint} \to \{\textbf{true}, \textbf{false}\}$
  $\textbf{sat}^\sharp_s(\sigma^\sharp_s, S = E)$ conservatively checks if $\sigma^\sharp_s$ satisfies $S = E$.

- $\sqsubseteq^\sharp_s : \mathbb{D}^\sharp_s \to \mathbb{D}^\sharp_s \to \{\textbf{true}, \textbf{false}\}$
  Abstract inclusion checking, using $\textbf{sat}^\sharp_s$

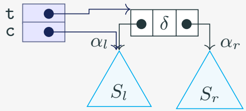
- $\sqcup^\sharp_s : \mathbb{D}^\sharp_s \to \mathbb{D}^\sharp_s \to \mathbb{D}^\sharp_s$
  That tries to infer common definitions from both inputs.
  Example $\begin{pmatrix} S = S_1.S_2 \\ \wedge\, S_3 = [] \end{pmatrix} \sqcup^\sharp_s \begin{pmatrix} S = S_2.S_3 \\ \wedge\, S_1 = [] \end{pmatrix}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

- $\nabla^\sharp_s : \mathbb{D}^\sharp_s \to \mathbb{D}^\sharp_s \to \mathbb{D}^\sharp_s$
  That selects the constraints in the left arguments verified in the right one.

Shape analysis with sequence predicates



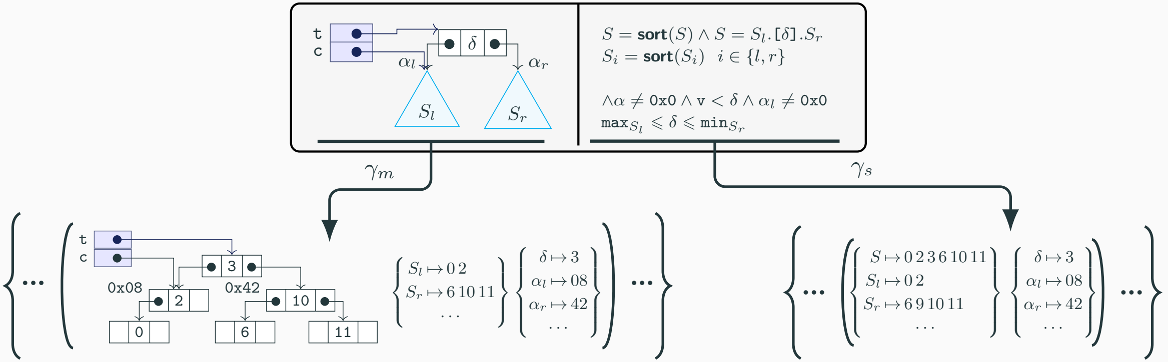
Elements of the combined domain and their concretization



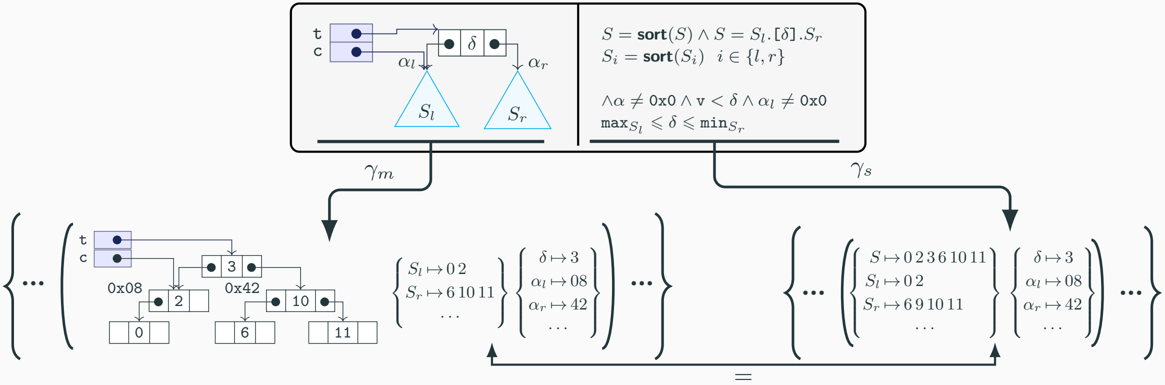
$S = \textbf{sort}(S) \wedge S = S_l.[\delta].S_r$
$S_i = \textbf{sort}(S_i) \quad i \in \{l, r\}$

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

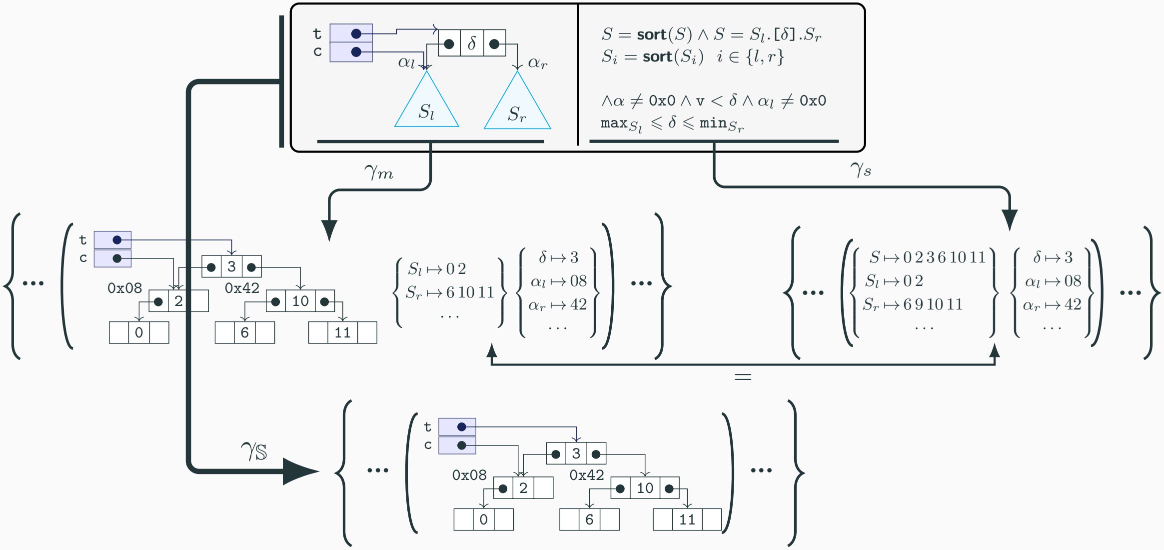
Elements of the combined domain and their concretization

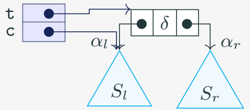


Elements of the combined domain and their concretization



Elements of the combined domain and their concretization





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

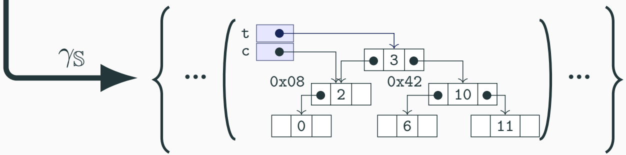
$\wedge \alpha \neq \mathtt{0x0} \wedge \mathtt{v} < \delta \wedge \alpha_l \neq \mathtt{0x0}$
$\mathtt{max}_{S_l} \leqslant \delta \leqslant \mathtt{min}_{S_r}$

Consequence

Symbolic variable are existentially quantified at the level of the abstract state

This also includes sequence variables S, S_l, S_r

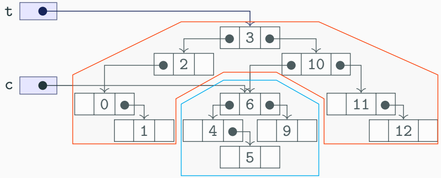
$$\gamma_{\mathbb{S}}(m^\sharp, \sigma_s) := \left\{ m \in \mathbb{M} \,\middle|\, \exists \nu, \begin{array}{l} (m, \nu) \in \gamma_m(m^\sharp) \\ \wedge\, \nu \in \gamma_s(\sigma_s) \end{array} \right\}$$

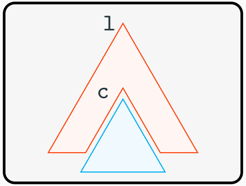


Integrating sequence parameters in the shape domain

The **tree**(c) predicate only synthesizes full binary trees.

To abstract partial trees, the shape domain uses a **segment predicate** **treeseg**(l, c).





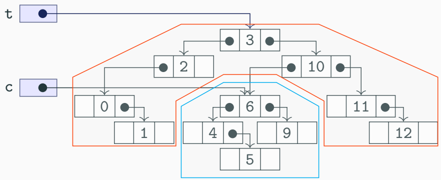
The shape domain automatically derives **treeseg** from **tree**.

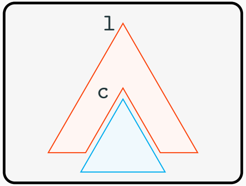
The analysis must keep tracks of the content stored in the segment

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

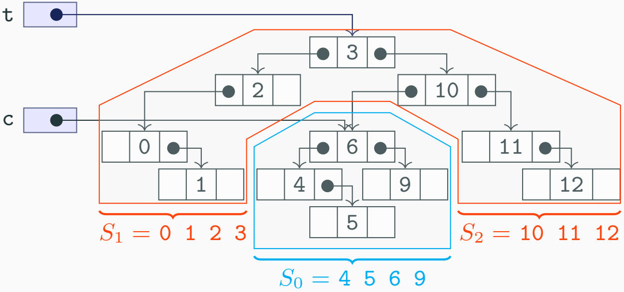
In order to reason precisely over inductive predicates, the shape analysis relies on:

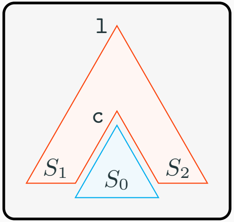
- **Unfold**: refines the memory by materializing synthesized memory.
- **Fold**: extrapolates the memory state to weaken it.

  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

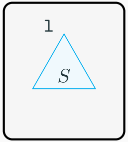
The analysis needs to recall the location of the missing sequence in **treeseg**.

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

One for each side of the missing sequence

Refining abstract memory state with unfolding

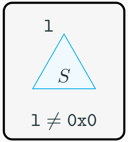
To analyze `if(l){v=l->data}` with initial state **tree**$(\texttt{l}, S)$:



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.



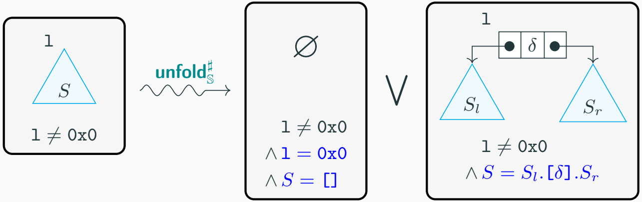
Refining abstract memory state with unfolding

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



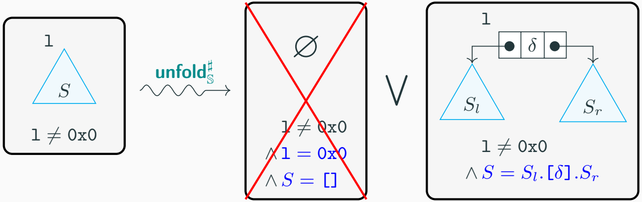
To analyze `if(l){v=l->data}` with initial state **tree**$(\texttt{l}, S)$:

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

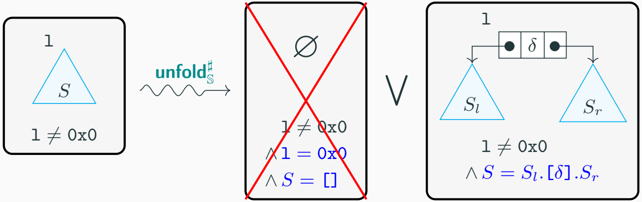
To analyze `if(l){v=l->data}` with initial state **tree**$(\texttt{l}, S)$:

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
   - **The non-empty case**: `l->data` corresponds to δ.



Refining abstract memory state with unfolding

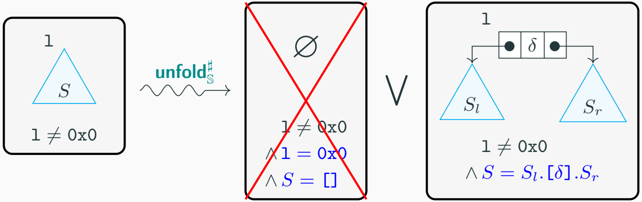
To analyze `if(l){v=l->data}` with initial state **tree**(`l`, S):

1. The numerical constraint $\texttt{l} \neq \texttt{0x0}$ is guarded in the numerical part of the sequence domain.
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**: `l->data` corresponds to δ.
3. The assignment $\texttt{v} \leftarrow \delta$ is performed.



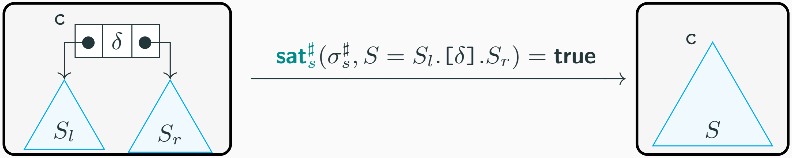
Theorem: Soundness of unfolding

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

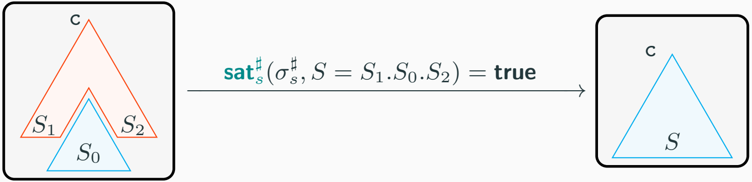
Fold generalizes the abstract state by rewriting parts of the memory into a predicate.

The analysis first checks that some constraints hold in the sequence domain.

Folding an inductive predicate



Folding segment and predicates



Theorem: Soundness of folding

The folded abstract state over-approximates the original one.

Lattice operators

- $\sqsubseteq^\sharp_{\mathbb{S}} : \mathbb{S}^\sharp \to \mathbb{S}^\sharp \to \{\textbf{true}, \textbf{false}\}$
  Abstract inclusion checking
- $\sqcup^\sharp_{\mathbb{S}} : \mathbb{S}^\sharp \to \mathbb{S}^\sharp \to \mathbb{S}^\sharp$
  Compute a common over-approximation of the inputs.
- $\nabla^\sharp_{\mathbb{S}} : \mathbb{S}^\sharp \to \mathbb{S}^\sharp \to \mathbb{S}^\sharp$
  Compute a common over-approximation of the inputs and ensures convergence ofiterations.

Lattice operators

- $\sqsubseteq^\sharp_{\mathbb{S}} : \mathbb{S}^\sharp \to \mathbb{S}^\sharp \to \{\mathbf{true}, \mathbf{false}\}$

  Abstract inclusion checking
- $\sqcup^\sharp_{\mathbb{S}} : \mathbb{S}^\sharp \to \mathbb{S}^\sharp \to \mathbb{S}^\sharp$

  Compute a common over-approximation of the inputs.
- $\nabla^\sharp_{\mathbb{S}} : \mathbb{S}^\sharp \to \mathbb{S}^\sharp \to \mathbb{S}^\sharp$

  Compute a common over-approximation of the inputs and ensures convergence ofiterations.

All these operators follow a three-step principle:

1. **Shape step**

   Shape parts are folded to establish the result

   Sequence constraints to verify are accumulated
2. **Instantiation step**

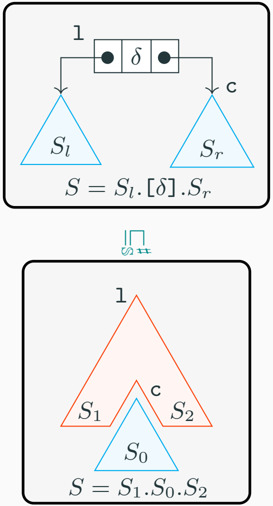
   Accumulated sequence constraints are used to enrich the sequence part of the input
3. **Sequence step** The result is computed in the sequence part

- $\sqsubseteq^\sharp_{\mathbb{S}} : \mathbb{S}^\sharp \to \mathbb{S}^\sharp \to \{\textbf{true}, \textbf{false}\}$
  Abstract inclusion checking
- $\sqcup^\sharp_{\mathbb{S}} : \mathbb{S}^\sharp \to \mathbb{S}^\sharp \to \mathbb{S}^\sharp$
  Compute a common over-approximation of the inputs.
- $\nabla^\sharp_{\mathbb{S}} : \mathbb{S}^\sharp \to \mathbb{S}^\sharp \to \mathbb{S}^\sharp$
  Compute a common over-approximation of the inputs and ensures convergence ofiterations.

All these operators follow a three-step principle:

1. **Shape step**
   Shape parts are folded to establish the result
   Sequence constraints to verify are accumulated
2. **Instantiation step**
   Accumulated sequence constraints are used to enrich the sequence part of the input
3. **Sequence step** The result is computed in the sequence part

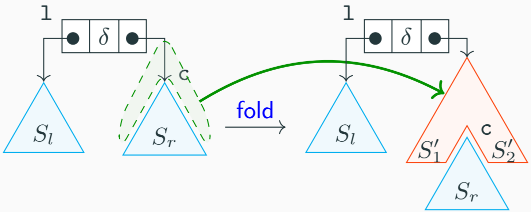
Example:



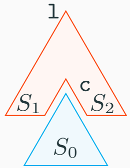
Inclusion test (shape step)

Accumulated constraints :

Inclusion test (shape step)



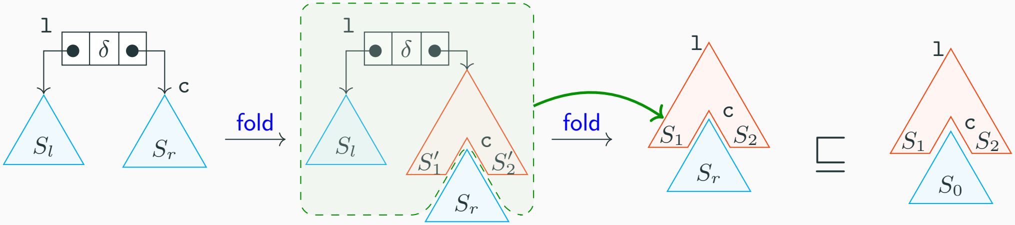
$\sqsubseteq$



$S'_1 = S'_2 = []$

Accumulated constraints :

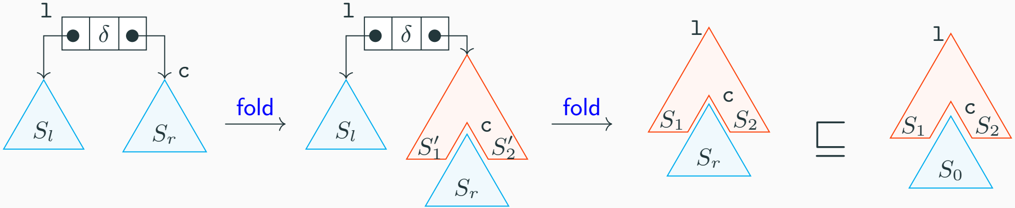
Inclusion test (shape step)



Accumulated constraints :

$$S'_1 = S'_2 = []$$
$$\wedge\ S_1 = S_l.[\delta].S'_1$$
$$\wedge\ S_2 = S'_2$$

Inclusion test (shape step)



Accumulated constraints :

$$
\begin{aligned}
& S'_1 = S'_2 = [] \\
\wedge\ & S_1 = S_l.[\delta].S'_1 \\
\wedge\ & S_2 = S'_2 \\
\wedge\ & S_0 = S_r
\end{aligned}
$$

Inclusion test (instantiation & sequence steps)

The inclusion between the shape parts hold if the accumulated constraints are valid:

$$S'_1 = S'_2 = [] \quad \wedge S_2 = S'_2$$
$$\wedge S_1 = S_l.[\delta].S'_1 \quad \wedge S_0 = S_r$$

Problem S_0, S_1, ... do not even appear in the left input.

It contains only S, S_l, and S_r

Inclusion test (instantiation & sequence steps)

The inclusion between the shape parts hold if the accumulated constraints are valid:

$$\begin{array}{ll} S'_1 = S'_2 = [] & \wedge S_2 = S'_2 \\ \wedge S_1 = S_l.[\delta].S'_1 & \wedge S_0 = S_r \end{array}$$

Problem S_0, S_1, ... do not even appear in the left input.

It contains only S, S_l, and S_r

Solution Use the accumulated constraints as definitions of unknown variables: **Instantiation**

This is sound since sequence variables are implicitly existentially quantified at the level of the abstract state.

Theorem: Soundness of instantiation

If S is not occurring in $s^\sharp$ nor in E, then $\gamma_{\mathbb{S}}(s^\sharp) \subseteq \gamma_{\mathbb{S}} \circ \mathbf{guard}^\sharp_s(s^\sharp, S = E)$

Inclusion test (instantiation & sequence steps)

The inclusion between the shape parts hold if the accumulated constraints are valid:

$$\begin{array}{ll} S_1' = S_2' = [] & \wedge S_2 = S_2' \\ \wedge S_1 = S_l.[\delta].S_1' & \wedge S_0 = S_r \end{array}$$

Problem S_0, S_1, ... do not even appear in the left input.

It contains only S, S_l, and S_r

Solution Use the accumulated constraints as definitions of unknown variables: **Instantiation**

This is sound since sequence variables are implicitly existentially quantified at the level of the abstract state.

Theorem: Soundness of instantiation

If S is not occurring in $s^\sharp$ nor in E, then $\gamma_{\mathbb{S}}(s^\sharp) \subseteq \gamma_{\mathbb{S}} \circ \mathbf{guard}_s^\sharp(s^\sharp, S = E)$

After the instantiation phase, the analysis performs the following inclusion test:

$$\begin{array}{l} S = S_1.S_r \\ \wedge S_1' = S_2' = S_2 = [] \\ \wedge S_1 = S_l.[\delta] \\ \wedge S_r = S_0 \end{array} \sqsubseteq_s^\sharp S = S_1.S_0.S_2$$

Inclusion test (instantiation & sequence steps)

The inclusion between the shape parts hold if the accumulated constraints are valid:

$$\begin{array}{ll} S'_1 = S'_2 = [] & \wedge\, S_2 = S'_2 \\ \wedge\, S_1 = S_l.[\delta].S'_1 & \wedge\, S_0 = S_r \end{array}$$

Problem S_0, S_1, ... do not even appear in the left input.

It contains only S, S_l, and S_r

Solution Use the accumulated constraints as definitions of unknown variables: **Instantiation**

This is sound since sequence variables are implicitly existentially quantified at the level of the abstract state.

Theorem: Soundness of instantiation

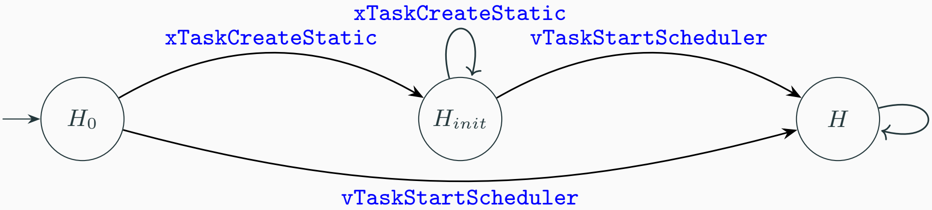
If S is not occurring in $s^\sharp$ nor in E, then $\gamma_{\mathbb{S}}(s^\sharp) \subseteq \gamma_{\mathbb{S}} \circ \mathbf{guard}^\sharp_s(s^\sharp, S = E)$

After the instantiation phase, the analysis performs the following inclusion test:

$$\begin{array}{l} S = S_1.S_r \\ \wedge\, S'_1 = S'_2 = S_2 = [] \\ \wedge\, S_1 = S_l.[\delta] \\ \wedge\, S_r = S_0 \end{array} \sqsubseteq^\sharp_s S = S_1.S_0.S_2 \qquad \Longrightarrow \text{The inclusion test returns } \textbf{true}$$

Verification of an instance of FreeRTOS

Specifications of the States of the scheduler



$H(\vec{p}) :=$

Ready: $(R_a,\ R_v)$

Delayed: (D_a, D_v)

$\texttt{pxCurrentTCB} \in \texttt{mset}_{R_a}$
$\wedge\ \texttt{xTick} < \texttt{min}_{D_v}$
$\wedge\ D_v$ is sorted
$\wedge\ \texttt{xNextUnblockTime} = \begin{cases} \texttt{MAX\_INT} & \text{if } D_v = \varepsilon \\ \min(D_v) & \text{otherwise} \end{cases}$
$\wedge\ \texttt{uxNumberOfTasks} = |R_a| + |D_a|$
$\wedge\ \texttt{uxSchedulerSuspended} \ldots$

H is fully described using 17 parameters: $R_a, R_v, \ldots, \texttt{xTick}, \ldots$

$$\{H(\vec{p}) \wedge \varphi_{pre}\}\ \mathtt{r} = \mathtt{f}(\mathtt{args})\ \{H(\vec{p}') \wedge \varphi_{post}\}$$

All pre- and post-conditions are written using $H(\vec{p})$

This ensures that `f` maintains the invariants of the scheduler

Cost of Specification :

- Simple functions (15/19)
  - 1∼2 goals per function A goal is similar to a *behavior* in ACSL
  - $<$ 10 lines per goal
- Complex functions (4/19) functions with loops: `xTaskIncrementTick` and callers
  - 4∼5 goals per function
  - 15∼40 lines per goals

$\Longrightarrow$ the full specification is done in ∼750 lines: specification/code ratio < 1.1

inductive predicates + scheduler invariants + functions pre- and post-conditions

Experimental results

Function name	Goal	LoS	Property verified	time (s) all	time (s) num	memory usage (MB)
`vTaskSwitchContext`	a	6	✓	0.63	0.00	28.11
	b	14	✓	0.76	0.08	29.37
	c	15	✓	0.79	0.09	29.73
	d	9	✓	0.72	0.05	29.28
`xTaskIncrementTick`	a	10	✓	0.82	0.03	29.55
	b	17	✓	0.75	0.06	30.10
	c	14	✓	0.73	0.04	30.42
	d	14	✓	0.74	0.04	30.06
	e	26	✓	36.48	33.39	68.10
	f	24	✓	21.80	19.69	42.35
`xTaskResumeAll`	a	36	✓	178.05	163.72	203.81
	b	34	✓	316.83	284.04	298.86
	c	9	✓	0.69	0.01	31.93
	d	25	✓	2.36	1.26	34.39
	e	26	✓	1.85	0.91	36.09
`xTaskCatchUpTicks`	a	26	✓	214.09	197.00	204.54
	b	28	✓	463.48	410.65	384.84
	c	17	✓	1.55	0.73	36.45
	d	18	✓	1.62	0.78	36.29
`vTaskDelay`	a	31	✓	14.51	12.94	35.48
	b	31	✓	21.31	19.44	37.96
	c	40	✓	759.65	694.22	661.01
	d	42	Safe Fc	823.71	762.47	612.33

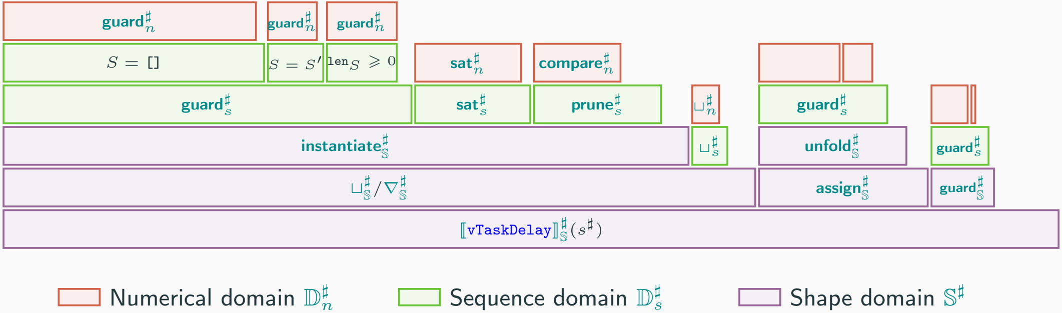
Safe:

- Memory safety
- Structural invariants

Fc:

- Functional invariants
- Partial functional correctness

Lessons learned: Cost of the analysis



- $\sim 70\%$ of time spent in $\sqcup^\sharp_{\mathbb{S}} / \nabla^\sharp_{\mathbb{S}}$.

  Curse of disjunctions introduced by unfolding predicates (up to 38 in `vTaskDelay`)
- Numerical domain operations have an **exponential cost**

  Light (in-)equalities domains do not reason on incremented values, we have to use polyhedra domain

Lessons learned: Impact on the analysis

- Modification of the sequence domain: only once

  for `xTaskIncrementTick`
- Efforts to improve the performance of the analysis:
  - Remove superfluous reduction operations
  - Try to use simple domains for (in-/dis-)equalities

    Does not work for functions incrementing values
  - Memorize calls to costly operators

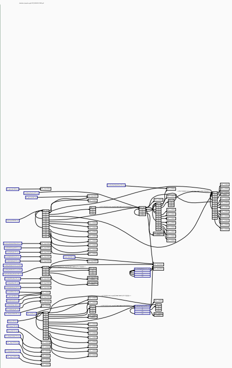
    Example Bound saturations of sequence variables, finding equal variables
- Help to the analysis
  - **Directive** for loop unroll, predicate unfolding, merging or introducing disjunctions
  - **Ghost code** to avoid aggressive predicates folding during widening

Lessons learned: Verification effort

- Overall **8 months**, distributed as follows:
  - $\sim 25\%$: writing/modifying the specification
  - $\sim 15\%$: Improving the analysis
  - $\sim 60\%$: Inspecting logs of analyzes

    Imprecision in the shape part is easily detected.
    Imprecision in the seq/num parts require more effort.
- Simple functions are easily proved
- Analysis of `xTaskIncrementTick` required 8 weeks

  Most of it was spent inspecting abstract states to localize the loss of precision
- `xTaskResumeAll` and `xTaskCatchUpTicks` were proved easily after
- `vTaskDelay` took 2~3 months.



Conclusion

Conclusion

How to improve the expressiveness of static analysis to automatically prove partial functional correctness of task schedulers?

Adding sequence parameters to inductive predicates

- Design of a novel sequence abstract domain

  It also provides insights over their length/bounds/content/sortedness

- Integration into a separation logic based shape analysis

  Using two sequence parameters for segments, Instantiation step for folding

Analysis of an instance of FreeRTOS

Specification of an instance

Verification of this instance using our analysis

Promising results!

The future:

- Analyzing other instances applying history of development
- Extending the analysis to support new features & prove new properties

Thank you for your attention !

Questions?

Sequence related stuff

Could we relax the sortedness checking?

Lemma

If $S = S_1 \dots S_n$, then

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

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

NO ! Because of the empty sequence case !

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

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:

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

$\nu_s \models S_i = \textsf{sort}(S_i), \quad \forall i$

$\nu_s \models \texttt{max}_{S_1} \leqslant \texttt{min}_{S_2}$

$\nu_s \models \texttt{max}_{S_2} \leqslant \texttt{min}_{S_3}$

But:

$\nu_s \not\models S = \textsf{sort}(S)$

Removing cyclic constraints

Assume the abstract state σ_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:

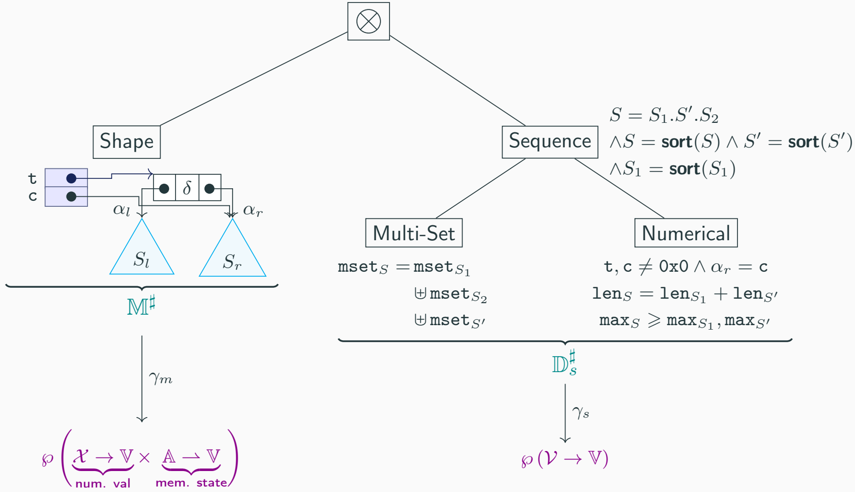
$$S = S_1.S_3.S.S_4.S_2$$

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 $\bot_s$.

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

Analysis structure

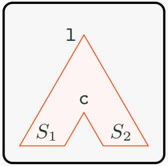


$$\mathbb{S}^\sharp =: \mathbb{M}^\sharp \times \mathbb{D}_s^\sharp \xrightarrow{\gamma_{\mathbb{S}}} \wp(\mathbb{A} \rightharpoonup \mathbb{V})$$

Construction of segments predicates

Segment tree predicate

To derive **treeseg**$(\texttt{l}, \texttt{c}, S_1 \boxdot S_2)$, denoting a partial tree between `c` and `l`:

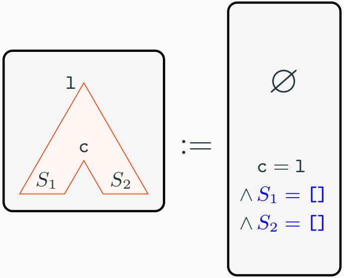


$:=$

Segment tree predicate

To derive **treeseg**$(\texttt{l}, \texttt{c}, S_1 \boxdot S_2)$, denoting a partial tree between c and l:

- We add the **empty segment case**
  - c and l are equal
  - There is no content: S_1 and S_2 are empty.



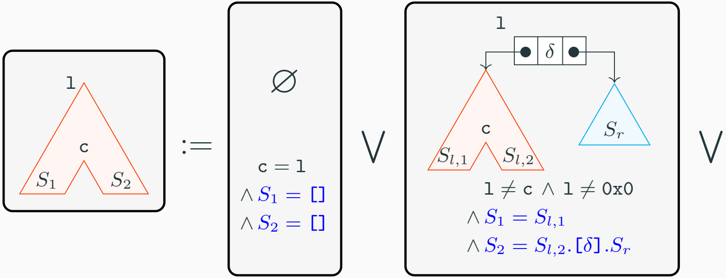
Segment tree predicate

To derive **treeseg**($\texttt{l}, \texttt{c}, S_1 \boxdot S_2$), denoting a partial tree between $\texttt{c}$ and $\texttt{l}$:

- We add the **empty segment case**
  - $\texttt{c}$ and $\texttt{l}$ are equal
  - There is no content: S_1 and S_2 are empty.
- The other cases must have at least one element inside: $\texttt{c}$ is in of the two subtrees.

The content constraints are synthesized by matching each side of the insertion point

e.g. in the left case: $S = S_l.[\delta].S_r \{S \leftarrow S_1 \boxdot S_2\} \{S_l \leftarrow S_{l,1} \boxdot S_{l,2}\}$
$\equiv S_1 \boxdot S_2 = S_{l,1} \boxdot S_{l,2}.[\delta].S_r \implies S_1 = S_{l,1}$, $S_2 = S_{l,2}.[\delta].S_r$



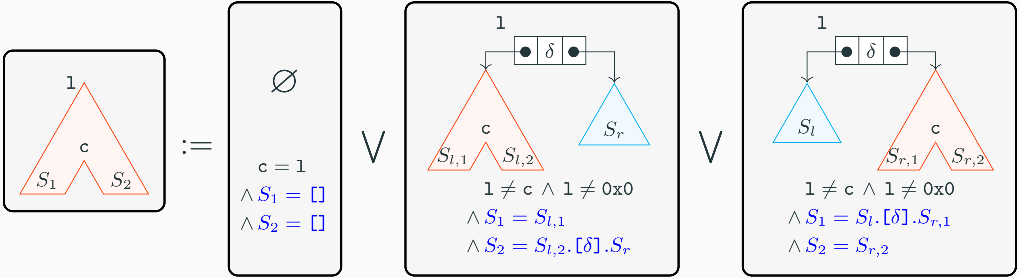
Segment tree predicate

To derive **treeseg**$(\texttt{l}, \texttt{c}, S_1 \boxdot S_2)$, denoting a partial tree between c and l:

- We add the **empty segment case**
  - c and l are equal
  - There is no content: S_1 and S_2 are empty.
- The other cases must have at least one element inside: c is in of the two subtrees.

  The content constraints are synthesized by matching each side of the insertion point

  e.g. in the left case: $S = S_l.[\delta].S_r \, \{S \leftarrow S_1 \boxdot S_2\} \, \{S_l \leftarrow S_{l,1} \boxdot S_{l,2}\} \equiv S_1 \boxdot S_2 = S_{l,1} \boxdot S_{l,2}.[\delta].S_r \implies \begin{array}{l} S_1 = S_{l,1} \\ S_2 = S_{l,2}.[\delta].S_r \end{array}$

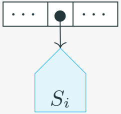


Hypothesis to derive segment from full predicate

Hypothesis

- The **only** constraint over sequence parameter is **concatenation based**. *i.e.* no **sort** predicate
- The argument of each recursive call occurs **exactly once** in the constraint.

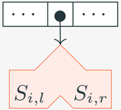
$:=$ $\quad\bigvee$



$$S = E.S_i.E'$$

$\bigvee$

$:=$ $\quad\bigvee$

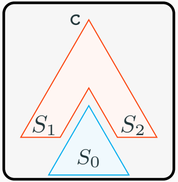


$$\begin{aligned} S_l \boxdot S_r &= E.S_i.E'\{S_i \leftarrow S_{i,l} \boxdot S_{i,r}\} \\ &= E.S_{i,l} \boxdot S_{i,r}.E' \\ &\Longrightarrow \begin{cases} S_l = E.S_{i,l} \\ S_r = S_{i,r}.E' \end{cases} \end{aligned}$$

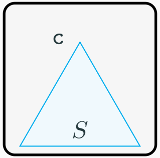
$\bigvee$

Concatenating inductive predicates

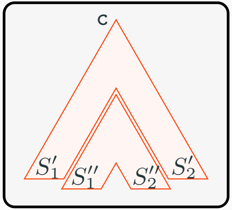
seg-full case



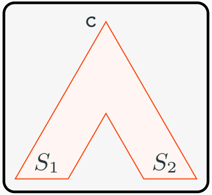
$$\mathsf{sat}^{\sharp}_{s}(\sigma^{\sharp}, S = S_1.S_0.S_2) = \textbf{true}$$



seg-seg case



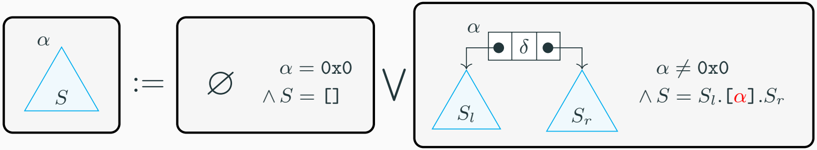
$$\mathsf{sat}^{\sharp}_{s}(\sigma^{\sharp}, S_1 = S'_1.S''_1) = \textbf{true}$$
$$\mathsf{sat}^{\sharp}_{s}(\sigma^{\sharp}, S_2 = S''_2.S'_2) = \textbf{true}$$



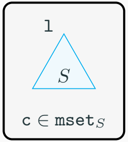
Other unfolding

Other examples of unfolding

Some unfolding leverages information from the sequence domain.
For instance, if S denotes the sequence of addresses of the nodes in the tree:

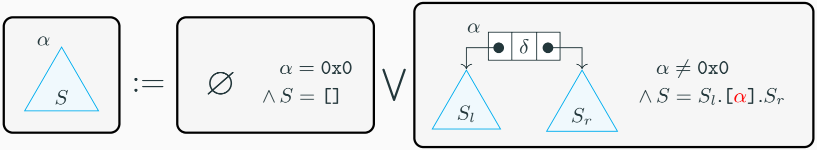


Let us analyze `v=c->data`, with initial state $(\textbf{tree}(l, S) \wedge \texttt{c} \in \texttt{mset}_S)$.

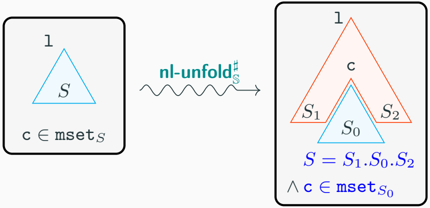


Other examples of unfolding

Some unfolding leverages information from the sequence domain.
For instance, if S denotes the sequence of addresses of the nodes in the tree:

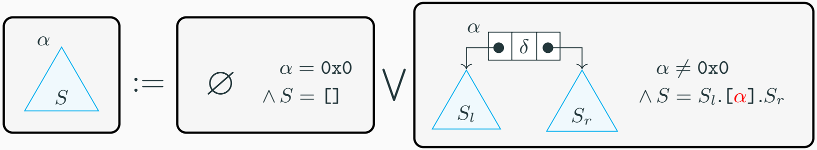


Let us analyze `v=c->data`, with initial state $(\textbf{tree}(l, S) \wedge \texttt{c} \in \texttt{mset}_S)$.

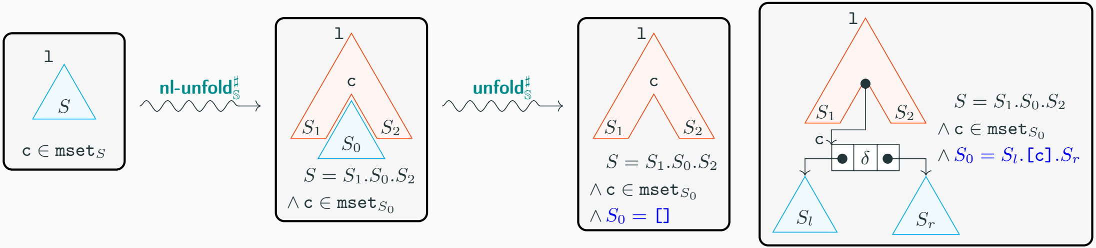


Other examples of unfolding

Some unfolding leverages information from the sequence domain.
For instance, if S denotes the sequence of addresses of the nodes in the tree:

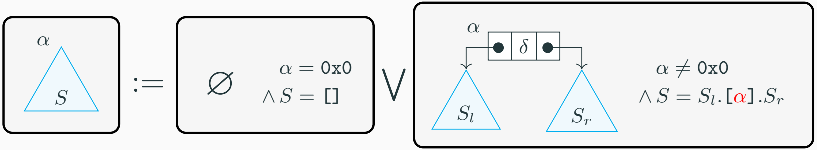


Let us analyze `v=c->data`, with initial state ($\mathbf{tree}(l, S) \wedge \texttt{c} \in \texttt{mset}_S$).

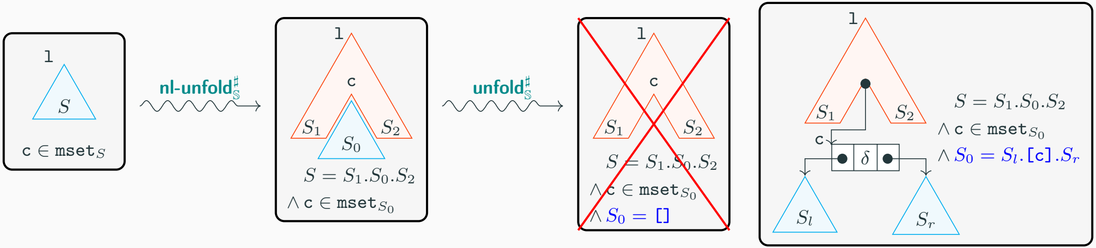


Other examples of unfolding

Some unfolding leverages information from the sequence domain.
For instance, if S denotes the sequence of addresses of the nodes in the tree:



Let us analyze `v=c->data`, with initial state ($\mathbf{tree}(l, S) \wedge$ `c` $\in$ `mset`$_S$).



Experiments

Experiment 1: Classical list & BST programs

Example	wo/ seq **Safe** verified	with seq parameters: time overhead	% num	**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 $\nabla_s^\sharp$ 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	✖	✖	✖
Extreme sentinel nodes			✖
Intrusive	✖	✖	
Pointer to header		✖	
Length in header		✖	✖
Sorted		✖	

	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✓		14/14✓

†: Cannot prove **Safe** for extraction at position.