Manuel Di Agostino
manuel.diagostino@studenti.unipr.it

Università di Parma

December 10, 2025

A generic framework for heap and value analyses of object-oriented programming languages

Based on: Ferrara, P. (2016). Theoretical Computer Science, 631, 43-72.

Summary

Summary

The problem

We would like to say something about the following:

```
int absSum(ListInt l) {
  int sum = 0;
  ListInt it = l;

  while(it != null) {
    if (it.f < 0)
      it.f = -it.f;

    sum += it.f
    it = it.next;
  }
  return sum;
}
```

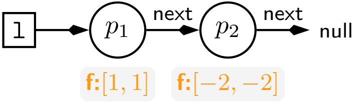
Listing: *Sum of absolute values of a list of integers.*

The problem

```
int absSum(ListInt l) {
  int sum = 0;
  ListInt it = l;

  while(it != null) {
    if (it.f < 0)
      it.f = -it.f;

    sum += it.f
    it = it.next;
  }
  return sum;
}
```

Two clients, different needs:

- **Client 1:** `absSum([1, -2])`
  - Fixed list, 2 nodes at labels p_1, p_2

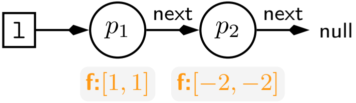


The problem

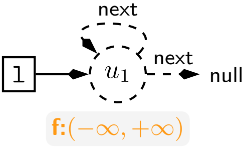
```
int absSum(ListInt l) {
  int sum = 0;
  ListInt it = l;

  while(it != null) {
    if (it.f < 0)
      it.f = -it.f;

    sum += it.f
    it = it.next;
  }
  return sum;
}
```

Two clients, different needs:

- **Client 1:** `absSum([1, -2])`
  - Fixed list, 2 nodes at labels p_1, p_2



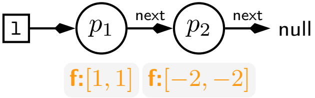
- **Client 2:** `absSum(`$[l_i \in \mathbb{Z}]$`)`, $1 \leq i \leq n$
  - Dynamic list, n unknown
  - Summary node u_1 abstracts all nodes



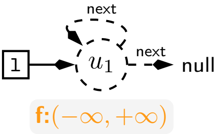
What we have

The Allocation-Site method

Client 1 (`fixed list`)



Client 2 (`dynamic list`)

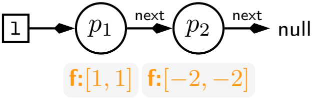


Property	Client 1 Result	Client 2 Result
P1: No NullPointer (Heap Structure)	✔ **Safe** Heap structure is precise	✔ **Safe** Summary handle nulls

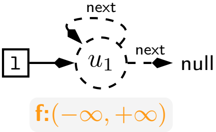
What we have

The Allocation-Site method

Client 1 (`fixed list`)



Client 2 (`dynamic list`)

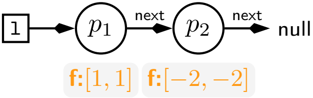


Property	Client 1 Result	Client 2 Result
P1: No NullPointer (Heap Structure)	✔ **Safe** Heap structure is precise	✔ **Safe** Summary handle nulls
P2: Return ≥ 0 (Sign Analysis)	✔ **Verified** Strong updates on p_1, p_2	✘ **False Alarm** Weak update on u_1

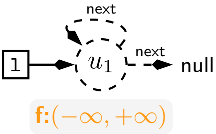
What we have

The Allocation-Site method

Client 1 (`fixed list`)



Client 2 (`dynamic list`)



Property	Client 1 Result	Client 2 Result
P1: No NullPointer (Heap Structure)	✔ **Safe** Heap structure is precise	✔ **Safe** Summary handle nulls
P2: Return ≥ 0 (Sign Analysis)	✔ **Verified** Strong updates on p_1, p_2	✘ **False Alarm** Weak update on u_1
P3: Elem ≥ 0 (Relational)	✔ **Verified** Relations preserved	✘ **False Alarm** Relations lost on u_1

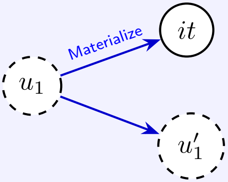
Materialization: The First Step

The Shape Analysis

When iterating the loop, the Heap Analysis **must materialize** the summary node u_1 to distinguish the current element from the rest:

- A concrete node it (the current element).
- A remaining summary node u_1' (the rest of the list).

Heap Analysis



NO AUTOMATIC COMMUNICATION

Value Analysis

Pre-state:
$u_1.f \in (-\infty, +\infty)$

Post-state:
$it.f \in ?$
$u_1'.f \in ?$

The Consequence: Information Loss

What we miss: When u_1 splits into it and u_1', where should the information $u_1.f \in (-\infty, +\infty)$ go?

Before		**After Materialization**
$u_1.f \in (-\infty, +\infty)$	$\xrightarrow{\text{materialize}}$	$\begin{cases} it.f \in (-\infty, +\infty) \\ u_1'.f \in (-\infty, +\infty) \end{cases}$

Remark (Consequence: False Alarms)

Without guidance from the Heap Analysis, the Value Domain conservatively assumes **top** for both $it.f$ and $u_1'.f$.
This forces analysis of the branch ‘`if (it.f < 0)`’, causing false alarms on properties P2 and P3.

The Solution: Substitutions

Key Idea: The Heap Analysis communicates how identifiers are transformed through a **substitution** message:

$$\{it.f, u'_1.f\} \mapsto \{u_1.f\}$$

What the framework does:

1. Heap Analysis materializes $\{it, u'_1\} \mapsto \{u_1\}$
2. Produces substitution: $\{it.f, u'_1.f\} \mapsto \{u_1.f\}$
3. Value Domain receives substitution
4. Value Domain **automatically updates** its state:
   - $it.f \in (-\infty, +\infty)$
   - $u'_1.f \in (-\infty, +\infty)$

The Solution: Substitutions

Key Idea: The Heap Analysis communicates how identifiers are transformed through a **substitution** message:

$$\{it.f, u'_1.f\} \mapsto \{u_1.f\}$$

Remark

- Inside the branch `if (it.f < 0)`, the analysis knows $it.f$ is negative.
- Since it is materialized, `it.f = -it.f` performs a **strong update**.
- The value is *overwritten* to $(0, +\infty)$, so we sum only **non-negative values**!

Summary

Summary

The Concrete Domain I

In a concrete execution, a standard object-oriented state holds two distinct types of data:

- **Values** (Val): Primitive data (integers, booleans, etc.)
- **References** (Ref): Pointers to objects or `null`

Information

Common in statically typed object-oriented programming languages like Java and C#.
It does not apply to imperative programming languages like C where references are threated as values (e.g., with pointer arithmetic).

The Concrete Domain II

Scope	Toy Language Definition (Syntax)	Target Domain Component
Reference	$\texttt{rexp} ::= x \mid x.f \mid \texttt{new}\ C$ $\texttt{rcond} ::= x \diamond \texttt{null} \mid x \diamond y$	$\Sigma_{\mathsf{Ref}} = \mathsf{Env}_{\mathsf{Ref}} \times \mathsf{Store}_{\mathsf{Ref}}$ (Heap Analysis, e.g., TVLA)
Value	$\texttt{vexp} ::= x \mid x.f \mid v_1 \diamond v_2$ $\texttt{vcond} ::= v_1 \diamond v_2$	$\Sigma_{\mathsf{Val}} = \mathsf{Env}_{\mathsf{Val}} \times \mathsf{Store}_{\mathsf{Val}}$ (Value Analysis, e.g., Intervals)
Statements	$\texttt{st} ::= x = \texttt{rexp} \mid x = \texttt{vexp} \mid \ldots$	Interaction via **Substitutions**

Table: Expressions, conditions and statements.

The Concrete Domain III

Definition (Concrete State)

The state Σ is composed of an environment and a store where types are mixed:

- **Environment**: Maps local variables to references or values.

$$\mathsf{Env} : \mathsf{Var} \rightarrow (\mathsf{Ref} \cup \mathsf{Val})$$

- **Store**: Maps heap locations (reference + field) to references or values.

$$\mathsf{Store} : (\mathsf{Ref} \times \mathsf{Fld}) \rightarrow (\mathsf{Ref} \cup \mathsf{Val})$$

The resulting state is then the cartesian product:

$$\Sigma = \mathsf{Env} \times \mathsf{Store}$$

The Split Domain I

To enable generic analysis combination, we formally split the concrete domain into **two disjoint components**.

Reference State (Σ_{Ref})

Tracks only topology and pointers.

- $\text{Env}_{\text{Ref}} : \text{Var} \to \text{Ref}$
- $\text{Store}_{\text{Ref}} : (\text{Ref} \times \text{Fld}) \to \text{Ref}$

Value State (Σ_{Val})

Tracks only primitive values.

- $\text{Env}_{\text{Val}} : \text{Var} \to \text{Val}$
- $\text{Store}_{\text{Val}} : (\text{Ref} \times \text{Fld}) \to \text{Val}$

The Split Domain II

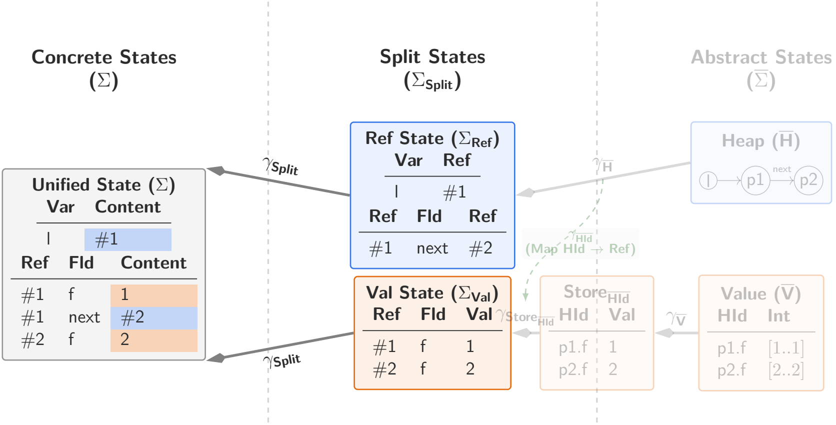
Definition (Split State Σ_{Split})

The resulting state is the cartesian product of the two components:

$$\Sigma_{\text{Split}} = \Sigma_{\text{Ref}} \times \Sigma_{\text{Val}}$$

- This domain is isomorphic to Σ but structurally separated.
- We can regard Σ_{Split} as a superstructure that is more convenient to abstract.

The Abstraction Hierarchy



Handling Mixed Expressions: Function $\mathbb{R}$ I

In the split domain, value expressions often depend on the heap (e.g., accessing `x.f`).

The Challenge

The Value Analysis (Σ_{Val}) needs to evaluate `x.f`, but it is "blind" to pointers: it does not know which reference `x` targets .

The Solution: Preprocessing Function $\mathbb{R}$

We introduce $\mathbb{R}$ to replace local variables in field accesses with their **concrete references** from the Reference State (Σ_{Ref}).

Transformation Rules:

$$\mathbb{R}[\![x, (e_{\mathsf{Ref}}, s_{\mathsf{Ref}})]\!] = x$$

$$\mathbb{R}[\![x.f, (e_{\mathsf{Ref}}, s_{\mathsf{Ref}})]\!] = \langle \mathbf{e}_{\mathsf{Ref}}(\mathbf{x}) \rangle .f$$

Handling Mixed Expressions: Function ℝ II

Integration in Semantics (Field Assignment): To evaluate $x.f = \texttt{vexp}$, we apply ℝ to resolve pointers on **both sides**, then delegate to Value semantics:

$$\frac{\langle \overbrace{\mathbb{R}[\![x.f, \sigma_{\mathsf{Ref}}]\!]}^{\text{Target Loc}} = \overbrace{\mathbb{R}[\![\texttt{vexp}, \sigma_{\mathsf{Ref}}]\!]}^{\text{Resolved Expr}}, \sigma_{\mathsf{Val}} \rangle \rightarrow_{\mathsf{Val}} \sigma'_{\mathsf{Val}}}{\langle x.f = \texttt{vexp}, (\sigma_{\mathsf{Ref}}, \sigma_{\mathsf{Val}}) \rangle \rightarrow_{\mathsf{Split}} (\sigma_{\mathsf{Ref}}, \sigma'_{\mathsf{Val}})}$$

Soundness: Lattice and Concretization

Lattice Structure

The Concrete Domain forms a **complete lattice** structure induced by standard set operations. The order is defined by set inclusion on the powerset of states:

$$\langle \wp(\Sigma), \subseteq \rangle$$

Like the concrete one, the lattice structure of the Split Domain is given by set of elements:

$$\langle \wp(\Sigma_{\mathsf{Split}}), \subseteq \rangle$$

Definition (Concretization γ_{Split})

Defines how split states map back to concrete states via **pointwise set union**:

$$\gamma_{\mathsf{Split}}(T) = \{(e_v \cup e_h, s_v \cup s_h) \mid ((e_h, s_h), (e_v, s_v)) \in T\}$$

Information

Crucial Assumption: Since the language distinguishes between value and reference expressions, the domains of e_v/e_h and s_v/s_h **do not overlap**.

Lemma (Galois Connection)

Since γ_{Split} is a complete $\cap$-morphism (based on set operators), it induces a valid Galois Connection:

$$\langle \wp(\Sigma), \subseteq \rangle \xleftarrow[\alpha_{Split}]{\gamma_{Split}} \langle \wp(\Sigma_{Split}), \subseteq \rangle$$

where $\alpha_{Split} = \lambda X. \cap \{Y : X \subseteq \gamma_{Split}(Y)\}$.

Proof.

α_{Split} is well-defined since γ_{Split} is a complete $\cap$-morphism since it is based on set operators. The fact that it forms a Galois connection follows immediately from the definition of α_{Split} [CC77]. □

Summary

The Generic Abstract Domain

The framework is designed to be **parametric**. We assume the existence of two abstract domains interacting to approximate the Split State.

1. Heap Analysis ($\overline{\mathsf{H}}$)

Approximates the reference state Σ_{Ref}.

- Lattice: $\langle \overline{\mathsf{H}}, \sqsubseteq_{\overline{\mathsf{H}}}, \sqcup_{\overline{\mathsf{H}}}, \sqcap_{\overline{\mathsf{H}}} \rangle$
- Widening: $\nabla_{\overline{\mathsf{H}}}$

2. Value Analysis ($\overline{\mathsf{V}}$)

Approximates the value state Σ_{Val}.

- Lattice: $\langle \overline{\mathsf{V}}, \sqsubseteq_{\overline{\mathsf{V}}}, \sqcup_{\overline{\mathsf{V}}}, \sqcap_{\overline{\mathsf{V}}} \rangle$
- Widening: $\nabla_{\overline{\mathsf{V}}}$

Definition (Abstract State $\overline{\Sigma}$)

The combined abstract state is the Cartesian product:

$$\overline{\Sigma} = \overline{\mathsf{H}} \times \overline{\mathsf{V}}$$

equipped with pointwise lattice operators.

Connecting Heap and Value: Heap Identifiers

The two domains need a common language to communicate.

- **Heap Domain ($\overline{\mathbf{H}}$)**: Knows about pointers, structure, and reachability. It abstracts memory addresses ($\mathbb{A}$).
- **Value Domain ($\overline{\mathbf{V}}$)**: Knows about numerical properties (intervals, polyhedra), but operates on a set of *variables*, not addresses.

Connecting Heap and Value: Heap Identifiers

The two domains need a common language to communicate.

- **Heap Domain ($\overline{\text{H}}$)**: Knows about pointers, structure, and reachability. It abstracts memory addresses ($\mathbb{A}$).
- **Value Domain ($\overline{\text{V}}$)**: Knows about numerical properties (intervals, polyhedra), but operates on a set of *variables*, not addresses.

The Solution: Abstract Heap Identifiers

The Heap Domain exports a set of symbolic names, called **Heap Identifiers**.

$$\overline{\text{HId}} = \{\text{id}_1, \text{id}_2, \ldots, \text{id}_n\}$$

- To $\overline{\text{H}}$, they represent abstract nodes (or regions of memory).
- To $\overline{\text{V}}$, they are treated simply as **variables**.

Value Analysis Concretization

Since the Value Analysis operates on identifiers, its concretization produces stores with abstract heap identifiers instead of concrete locations.

Abstract Store on Identifiers

$$\mathsf{Store}_{\overline{\mathsf{HId}}} : \overline{\mathsf{HId}} \to \wp(\mathsf{Val})$$

Codomain is a set of values because a summary node maps to many concrete references with different values.

Definition ($\gamma_{\overline{\mathbf{V}}}$)

The concretization of an abstract value state $\bar{v} \in \overline{\mathsf{V}}$ produces:

1. Environments in $\mathsf{Env}_{\mathsf{Val}}$
2. Stores in $\mathsf{Store}_{\overline{\mathsf{HId}}}$

Missing link: We have values for identifiers, but we need to know *where* these identifiers are in memory.

Heap Analysis Concretization

The Heap Analysis tracks the shape and symbolically represents nodes using $\overline{\mathsf{HId}}$. To concretize, it must map these symbols to concrete locations.

The Mapping Function $\gamma_{\overline{\mathsf{HId}}}$

The heap concretization provides a function relating identifiers to concrete locations ($\mathsf{Ref} \times \mathsf{Fld}$):

$$\gamma_{\overline{\mathsf{HId}}} : \overline{\mathsf{HId}} \to \wp(\mathsf{Ref} \times \mathsf{Fld})$$

Definition ($\gamma_{\overline{\mathsf{H}}}$)

Formally, $\gamma_{\overline{\mathsf{H}}} : \overline{\mathsf{H}} \to \wp(\Sigma_{\mathsf{Ref}} \times (\overline{\mathsf{HId}} \to \wp(\mathsf{Ref} \times \mathsf{Fld})))$
It returns a set of pairs containing:

- A concrete state σ_{H}.
- A concretization of heap identifiers $\gamma_{\overline{\mathsf{HId}}}$ to compute $\gamma_{\overline{\mathsf{V}}}$.

The Final Combined Concretization $\gamma_{\overline{\Sigma}}$

Finally, we define the concretization of the abstract state $(\bar{h}, \bar{v})$ into the Split domain.

Definition ($\gamma_{\overline{\Sigma}}$)

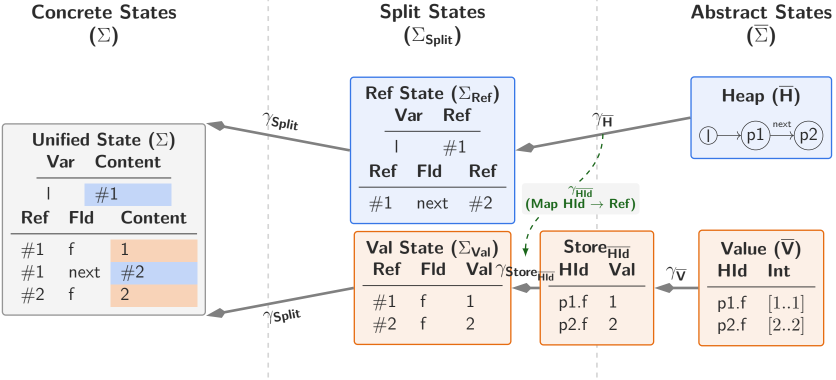
Formally, $\gamma_{\overline{\Sigma}}\big((\bar{h}, \bar{v})\big) : \overline{\Sigma} \to \wp(\Sigma_{\mathsf{Split}})$

$$\gamma_{\overline{\Sigma}}\big((\bar{h}, \bar{v})\big) = \{(\sigma_{\mathsf{H}}, (e_v, s_v)) \mid \text{conditions hold}\}$$

where:

- $(e_v, s_{\overline{\mathsf{HId}}}) \in \gamma_{\overline{\mathsf{V}}}(\bar{v})$
  (Value Analysis provides $\mathsf{Env}_{\mathsf{Val}}$ and $\mathsf{Store}_{\overline{\mathsf{HId}}}$)
- $(\sigma_{\mathsf{H}}, \gamma_{\overline{\mathsf{HId}}}) \in \gamma_{\overline{\mathsf{H}}}(\bar{h})$
  (Heap Analysis provides a state in Σ_{Ref} and ID-map)
- $s_v \in \gamma_{\mathsf{Store}_{\overline{\mathsf{HId}}}}(s_{\overline{\mathsf{HId}}}, \gamma_{\overline{\mathsf{HId}}})$
  (Combined to get concrete $\mathsf{Store}_{\mathsf{Val}}$)

The Abstraction Hierarchy



Soundness conditions

Remark (Requirements for Soundness (Conditions C1–C5))

The validity of the following Lemma relies on strict conditions on $\gamma_{\overline{\mathsf{HId}}}$:

- **Structural Consistency (C1, C3, C5):**
  - All heap identifiers must be concretizable (C1).
  - Identifiers must represent *disjoint* memory portions to allow independent updates (C3).
  - Non-summary nodes must map to exactly one reference (C5).
- **Mathematical Properties (C2, C4):** These ensure $\gamma_{\overline{\Sigma}}$ is **meet-preserving**. The heap identifiers' concretization of the intersection of heaps is the pointwise intersection of the heap identifiers' concretization of all the intersected states.

Lemma (Galois Connection)

Since $\gamma_{\overline{\Sigma}}$ is a complete $\cap$-morphism (based on set operators), it induces a valid Galois Connection:

$$\langle \wp(\Sigma_{\mathit{Split}}), \subseteq \rangle \underset{\alpha_{\overline{\Sigma}}}{\overset{\gamma_{\overline{\Sigma}}}{\leftrightarrows}} \langle \overline{\Sigma}, \sqsubseteq \rangle$$

where $\alpha_{\overline{\Sigma}} = \lambda X. \cap \{Y : X \subseteq \gamma_{\overline{\Sigma}}(Y)\}$.

Abstract Preprocessing: Function $\overline{\mathbb{R}}$ I

In the abstract domain, the Value Analysis ($\overline{\mathsf{V}}$) tracks information on **Heap Identifiers** (abstract nodes), not on raw pointer paths.

The Abstract Solution

We define the abstract preprocessing function $\overline{\mathbb{R}}$ to translate field accesses into the corresponding **set of Heap Identifiers** provided by the Heap Analysis ($\overline{\mathsf{H}}$).
Unlike the concrete case, this translation may return **multiple** results (e.g., if a variable points to multiple abstract nodes).

Abstract Transformation Rules :

$$\overline{\mathbb{R}}[\![x, \overline{h}]\!] = \{x\}$$
$$\overline{\mathbb{R}}[\![x.f, \overline{h}]\!] = I \quad \text{where } \langle x.f, \overline{h}\rangle \rightarrow_{\overline{\mathsf{H}}} I$$

Note: I is the set of heap identifiers (e.g., $\{id_1, id_2\}$) retrieved by the Heap Analysis.

Abstract Preprocessing: Function $\overline{\mathbb{R}}$ II

Semantics Integration (Updates)

Once $\overline{\mathbb{R}}$ resolves the field access, the Value Analysis performs the assignment. Formally, we compute the join of all possible outcomes for each identifier i returned by $\overline{\mathbb{R}}$:

$$\overline{v}_{new} = \bigsqcup \{\overline{v}' \mid i \in \overline{\mathbb{R}}[\![x.f, \overline{h}]\!], \langle i = \ldots, \overline{v} \rangle \rightarrow_{\overline{V}} \overline{v}'\}$$

The final state depends on the precision of the Heap Analysis:

- **Strong Update:** If i is unique and definite ($\neg$isSum(i)), the old value is replaced:

$$\overline{v}_{post} = \overline{v}_{new}$$

- **Weak Update:** If i is a summary node or multiple identifiers exist, we must join with the previous state to preserve soundness:

$$\overline{v}_{post} = \overline{v} \sqcup_{\overline{V}} \overline{v}_{new}$$

Summary

Instantiation: Heap Analyses - PA

The framework is validated by plugging in two different heap abstractions:

1. **Pointer Analysis (PA)**
   A flow-sensitive analysis based on *Allocation Site Abstraction* [And94; MSV10].
   - **Heap Identifiers:** Defined by the allocation site label and the field name:
     $$\overline{\mathsf{HId}}_{\mathsf{PA}} = \mathsf{Lab} \times \mathsf{Field}$$
   - **Substitutions:** Since abstract locations are statically determined, no dynamic splitting occurs. Therefore, substitutions are always empty ($\emptyset$).

Instantiation: Heap Analysis - TVAL+

2. **TVAL+ (Shape Analysis)**
   Based on the 3-valued logic engine TVLA [SRW02].
   - **Identity:** Introduces *Name Predicates* to track nodes across transformations.
   - **Normalization:** A `merge/split` procedure ensures states are normalized.
   - **Substitutions:** Generated dynamically to inform the value analysis about node materialization and summarization.

Instantiation: Value Analysis

The generic framework allows standard numerical domains to track information over heap contents.

3. **Numerical Domains**
   Standard domains (e.g., Intervals, Octagons) are plugged into the value component.
   - **Transparency:** Heap Identifiers ($\overline{\mathsf{HId}}$) are treated transparently, exactly as local variables.
   - **Handling Summaries:** To preserve soundness, assignments to summary nodes ($\overline{\mathsf{isSum}}(id) = \mathsf{true}$) are handled via **weak updates** (join of old and new values).

Conclusion on Instantiation: The framework successfully bridges static approaches (PA) and dynamic approaches (TVLA) with numerical reasoning without losing soundness.

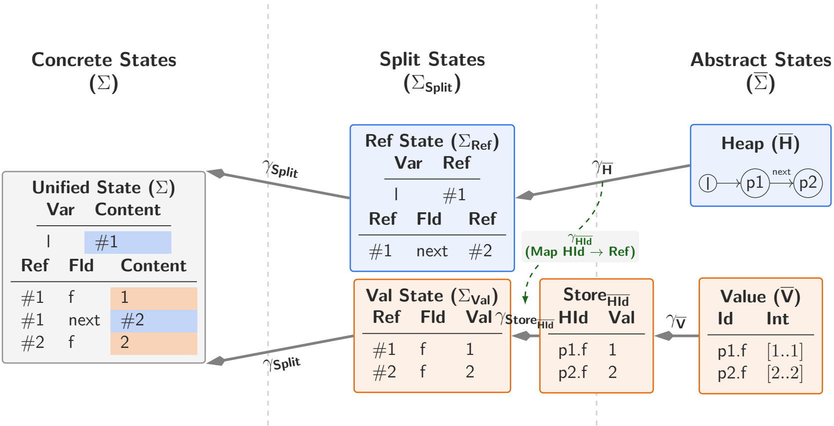
Summary

Conclusion and Contributions

The paper presented a formal approach to the static analysis of object-oriented languages.

Key Contributions:

- **Generic Framework:** A unified formalization to automatically combine arbitrary Heap and Value analyses.
- **Handling Dynamics:** The *first* generic framework capable of supporting **materialization** and **summarization** of heap identifiers.
- **Soundness:** Proved soundness of the combination, relying on standard Abstract Interpretation operators and a specific interface (Substitutions).
- **Versatility:** Successfully instantiated with:
    - Standard Pointer Analysis (Static).
    - TVAL+ / TVLA (Dynamic Shape Analysis).
    - Numerical Domains (Value Analysis).



Thank you for the attention!

References I

[And94] Lars Ole Andersen. "Program analysis and specialization for the C programming language". PhD thesis. DIKU, University of Copenhagen, 1994.

[CC77] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Proceedings of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '77. ACM, 1977, pp. 238–252.

[MSV10] Matthew Might, Yannis Smaragdakis, and David Van Horn. "Resolving and exploiting the k-CFA paradox". In: *Proceedings of PLDI '10*. ACM. 2010.

[SRW02] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. "Parametric shape analysis via 3-valued logic". In: *ACM Trans. Program. Lang. Syst.* 24.3 (2002), pp. 217–298.