Shape analysis abstractions

MPRI — Cours 2.6 “Interprétation abstraite : application à la vérification et à l’analyse statique”

Xavier Rival

INRIA

Jan, 17th, 2022
Shape analyses aim at discovering structural invariants of programs that manipulate complex unbounded data-structures.

**Applications:**
- establish **memory safety**
- verify the preservation of **structural properties**
  e.g., list, doubly-linked lists, trees, ...
- reason about programs that manipulate **unbounded** memory states

**Previous course:** TVLA, *i.e.*,
- **logical predicates** which evaluate in three valued logic
- **shape graphs**, described by the predicates
Another family of shape analyses

Today: systematically avoid weak updates

- separation logic, a logic to describe properties of memory states
- abstract domain
- static analysis algorithms
- combination with numerical domains
- widening operators...
Outline

1 Introduction

2 Separation Logic

3 A shape abstract domain relying on separation

4 Standard static analysis algorithms

5 Combining shape and value abstractions

6 Conclusion
Separation logic principle: avoid weak updates

How to deal with weak updates?

Avoid them!

- Always materialize exactly the cell that needs to be modified.
- Can be very costly to achieve, and not always feasible.

- Notion of property that holds over a memory region: special separating conjunction operator $\ast$

- Local reasoning:
  - Powerful principle, which allows to consider only part of the memory.

- Separation logic has been used in many contexts, including manual verification, static analysis, etc...
Separation logic

Two kinds of formulas:
- **pure formulas** behave like formulas in first-order logic, i.e., are not attached to a memory region
- **spatial formulas** describe properties attached to a memory region

**Pure formulas** denote value properties

\[
e ::= n \quad (n \in \mathbb{N}) \quad \text{constants} \\
| 1 \quad \text{l-value} \\
| e_0 + e_1 \quad \text{binary operations} \\
| \ldots
\]

\[
P ::= e_0 = e_1 \mid P' \lor P'' \mid P' \land P'' \ldots \quad \text{pure predicates}
\]

**Pure formulas semantics:** \( \gamma(P) \subseteq E \times H \)
Separation logic: points-to predicates

The next slides introduce the main separation logic formulas $F ::= \ldots$

We start with the most basic predicate, that describes a single cell:

Points-to predicate

- **Predicate:**
  
  $$F ::= \ldots \mid a \mapsto v$$
  
  where $a$ is an address and $v$ is a value

- **Concretization:**

  $$(e, h) \in \gamma(a \mapsto v) \quad \text{if and only if} \quad h = \llbracket a \rrbracket (e, h) \mapsto v$$

- **Example:**

  $$F = \& x \mapsto 18 \quad \& x = 308$$

  We also note $l \mapsto e$, as an l-value $l$ denotes an address
Separation logic: separating conjunction

**Merge of concrete heaps:** let \( h_0, h_1 \in (\forall \text{addr} \rightarrow \forall) \), such that \( \text{dom}(h_0) \cap \text{dom}(h_1) = \emptyset \); then, we let \( h_0 \otimes h_1 \) be defined by:

\[
\begin{align*}
\text{dom}(h_0) \cup \text{dom}(h_1) &\rightarrow \forall \\
\times \in \text{dom}(h_0) &\mapsto h_0(\times) \\
\times \in \text{dom}(h_1) &\mapsto h_1(\times)
\end{align*}
\]

**Separating conjunction**

- **Predicate:**
  \[
  F ::= \ldots \mid F_0 \ast F_1
  \]

- **Concretization:**
  \[
  \gamma(F_0 \ast F_1) = \{(e, h_0 \otimes h_1) \mid (e, h_0) \in \gamma(F_0) \land (e, h_1) \in \gamma(F_1)\}
  \]
An example

**Concrete memory layout**
(pointer values underlined)

<table>
<thead>
<tr>
<th>address</th>
<th>&amp;x = 300</th>
<th>&amp;y = 308</th>
<th>&amp;z = 312</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td>64</td>
<td>312</td>
<td>88</td>
</tr>
<tr>
<td>304</td>
<td>312</td>
<td></td>
<td></td>
</tr>
<tr>
<td>316</td>
<td>0x0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A formula that abstracts away the addresses:

\[
&x \mapsto \langle 64, &z \rangle \ast &y \mapsto &z \ast &z \mapsto \langle 88, 0 \rangle
\]
Separation Logic

Separation logic: non separating conjunction

We can also add the **conventional conjunction operator**, with its **usual concretization**:

### Non separating conjunction

- **Predicate:**
  \[ F ::= \ldots | F_0 \land F_1 \]

- **Concretization:**
  \[ \gamma(F_0 \land F_1) = \gamma(F_0) \cap \gamma(F_1) \]

**Exercise:** describe and compare the concretizations of

- \&a \mapsto \&b \land \&b \mapsto \&a
- \&a \mapsto \&b \ast \&b \mapsto \&a
Separating conjunction vs non separating conjunction

- **Classical conjunction**: properties for the same memory region
- **Separating conjunction**: properties for disjoint memory regions

\[ \&a \mapsto \&b \land \&b \mapsto \&a \]

- the same heap verifies \( \&a \mapsto \&b \) and \( \&b \mapsto \&a \)
- there can be only one cell
- thus \( a = b \)

\[ \&a \mapsto \&b \not\! \& \&b \mapsto \&a \]

- two separate sub-heaps respectively satisfy \( \&a \mapsto \&b \) and \( \&b \mapsto \&a \)
- thus \( a \neq b \)

Separating conjunction and non-separating conjunction have very different properties

Both express very different properties e.g., no ambiguity on weak / strong updates
Separating and non separating conjunction

Logic rules of the two conjunction operators of SL:

- **Separating conjunction:**
  \[
  \frac{(e, h_0) \in \gamma(F_0) \quad (e, h_1) \in \gamma(F_1)}{(e, h_0 \otimes h_1) \in \gamma(F_0 \ast F_1)}
  \]

- **Non separating conjunction:**
  \[
  \frac{(e, h) \in \gamma(F_0) \quad (e, h) \in \gamma(F_1)}{(e, h) \in \gamma(F_0 \wedge F_1)}
  \]

Reminiscent of Linear Logic [Girard87]: resource aware / non resource aware conjunction operators
Separation logic: empty store

Empty store

- **Predicate:**
  \[ F ::= \ldots \mid \text{emp} \]

- **Concretization:**
  \[ \gamma(\text{emp}) = \{(e, []) \mid e \in E\} = E \times \{[]\} \]

  where [] denotes the empty store

- \text{emp} is the **neutral element for} \ast \)**
  (monoid structure induced by \ast)

- by contrast the **neutral element for} \wedge \)** is TRUE, with concretization:
  \[ \gamma(\text{TRUE}) = E \times \text{H} \]
Separation logic: other connectors

Disjunction:

- $F ::= \ldots | F_0 \lor F_1$

concretization:

$$\gamma(F_0 \lor F_1) = \gamma(F_0) \cup \gamma(F_1)$$

Spatial implication (aka, magic wand):

- $F ::= \ldots | F_0 \rightarrow F_1$

concretization:

$$\gamma(F_0 \rightarrow F_1) = \{ (e, h) \mid \forall h_0 \in \mathbb{H}, (e, h_0) \in \gamma(F_0) \implies (e, h \circledast h_0) \in \gamma(F_1) \}$$

- very powerful connector to describe **structure segments**, used in complex SL proofs
Separation logic

Summary of the main separation logic constructions seen so far:

Separation logic main connectors

\[ \gamma(\text{emp}) = E \times \{[]\} \]
\[ \gamma(\text{TRUE}) = E \times H \]
\[ \gamma(l \mapsto v) = \{(e, [l](e, h) \mapsto v)) | e \in E\} \]
\[ \gamma(F_0 \ast F_1) = \{(e, h_0 \otimes h_1) | (e, h_0) \in \gamma(F_0) \land (e, h_1) \in \gamma(F_1)\} \]
\[ \gamma(F_0 \land F_1) = \gamma(F_0) \cap \gamma(F_1) \]
\[ \gamma(F_0 \lor F_1) = \gamma(F_0) \cup \gamma(F_1) \]
\[ \gamma(F_0 \rightarrow* F_1) = \{(e, h) | \forall h_0 \in H, (e, h_0) \in \gamma(F_0) \implies (e, h \otimes h_0) \in \gamma(F_1)\} \]

Concretization of pure formulas is standard

How does this help for program reasoning?
Separation logic triple

Program proofs based on Hoare triples

- **Notation:** \( \{F\} p \{F'\} \) if and only if:
  \[
  \forall s, s' \in S, \ s \in \gamma(F) \land s' \in \llbracket p \rrbracket(s) \implies s' \in \gamma(F')
  \]

- **Application:** formalize proofs of programs

A few rules (straightforward proofs):

\[
\begin{align*}
\{\&x \mapsto ?\}x := e\{\&x \mapsto e\} & \quad \text{mutation} \\
F_0 \implies F'_0 & \\
\{F'_0\}b\{F'_1\} & \\
F'_1 \implies F_1 & \quad \text{consequence} \\
\{F_0\}b\{F_1\} & \\
x \text{ does not appear in } F & \\
\{\&x \mapsto ? \ast F\}x := e\{\&x \mapsto e \ast F\} & \quad \text{mutation-2}
\end{align*}
\]

(we assume that e does not allocate memory)
The frame rule

What about the resemblance between rules “mutation” and “mutation-2”? 

**Theorem: the frame rule**

\[
\begin{align*}
\{F_0\} b \{F_1\} & \quad \text{freevar}(F) \cap \text{write}(b) = \emptyset \\
\{F_0 \ast F\} b \{F_1 \ast F\} & \quad \text{frame}
\end{align*}
\]

- Proof by induction on the logical rules on program statements, *i.e.*, essentially a large case analysis
  (see biblio for a more complete set of rules)
- Rules are proved by case analysis on the program syntax

**The frame rule allows to reason locally about programs**
Application of the frame rule

**A program with intermittent invariants**, derived using the frame rule, since each step impacts a disjoint region:

```plaintext
int i;
int * x;
int * y;
{&i ↦ ? * &x ↦ ? * &y ↦ ?}
x = &i;
{&i ↦ ? * &x ↦ &i * &y ↦ ?}
y = &i;
{&i ↦ ? * &x ↦ &i * &y ↦ &i}
*x = 42;
{&i ↦ 42 * &x ↦ &i * &y ↦ &i}
```

Many other program proofs done using separation logic
e.g., verification of the Deutsch-Shorr-Waite algorithm (biblio)
What do we still miss?

So far, formulas denote **fixed sets of cells**
Thus, no summarization of unbounded regions...

- **Example** all lists pointed to by `x`, such as:

  ![Example Diagram]

  - How to precisely abstract these stores with **a single formula**
    *i.e.*, no infinite disjunction?
Inductive definitions in separation logic

List definition

\[ \alpha \cdot \text{list} := \alpha = 0 \land \text{emp} \lor \alpha \neq 0 \land \alpha \cdot \text{next} \mapsto \delta \ast \alpha \cdot \text{data} \mapsto \beta \ast \delta \cdot \text{list} \]

- Formula abstracting our set of structures:
  \[ \&x \mapsto \alpha \ast \alpha \cdot \text{list} \]

- **Summarization:**
  this formula is finite and describe infinitely many heaps

- **Concretization:** next slide...

Practical implementation in verification/analysis tools

- **Verification:** hand-written definitions
- **Analysis:** either built-in or user-supplied, or partly inferred
Concretization by unfolding

Intuitive semantics of inductive predicates

- Inductive predicates can be **unfolded**, by **unrolling their definitions**
  - Syntactic unfolding is noted $U \rightarrow$
  - A formula $F$ with inductive predicates describes all stores described by all formulas $F'$ such that $F \xrightarrow{U} F'$

**Example:**

- Let us start with $x \mapsto \alpha_0 \cdot \alpha_0 \cdot \text{list}$; we can unfold it as follows:
  
  $\&x \mapsto \alpha_0 \cdot \alpha_0 \cdot \text{list}$
  
  $\xrightarrow{U}$
  
  $\&x \mapsto \alpha_0 \cdot \alpha_0 \cdot \text{next} \mapsto \alpha_1 \cdot \alpha_0 \cdot \text{data} \mapsto \beta_1 \cdot \alpha_1 \cdot \text{list}$
  
  $\xrightarrow{U}$
  
  $\&x \mapsto \alpha_0 \cdot \alpha_0 \cdot \text{next} \mapsto \alpha_1 \cdot \alpha_0 \cdot \text{data} \mapsto \beta_1 \cdot \text{emp} \land \alpha_1 = 0x0$

- We get the concrete state below:
Example: tree

- Example:

\[
\alpha \cdot \text{tree} := \begin{align*}
\alpha &= 0 \land \text{emp} \\
\lor \alpha &\neq 0 \land \alpha \cdot \text{left} \rightarrow \beta \ast \alpha \cdot \text{right} \rightarrow \delta \\
\ast \beta \cdot \text{tree} \ast \delta \cdot \text{tree}
\end{align*}
\]
Example: doubly linked list

Inductive definition

- We need to propagate the `prev` pointer as an additional parameter:

\[
\alpha \cdot \text{dll}(\delta) \quad := \\
\begin{align*}
\alpha &= 0 \land \text{emp} \\
\lor \quad \alpha &\neq 0 \land \alpha \cdot \text{next} \mapsto \beta \land \alpha \cdot \text{prev} \mapsto \delta \\
&& \star \beta \cdot \text{dll}(\alpha)
\end{align*}
\]
Example: sortedness

- **Example:** sorted list

---

Inductive definition

- Each element should be greater than the previous one
- The first element simply needs be greater than $-\infty$...
- We need to propagate the lower bound, using a scalar parameter

\[
\alpha \cdot \text{lsort}_{\text{aux}}(n) := \begin{cases} 
\alpha = 0 & \land \text{emp} \\
\alpha \neq 0 & \land n \leq \beta & \land \alpha \cdot \text{next} \mapsto \delta \\
* \alpha \cdot \text{data} \mapsto \beta & * \delta \cdot \text{lsort}_{\text{aux}}(\beta)
\end{cases}
\]

\[
\alpha \cdot \text{lsort}() := \alpha \cdot \text{lsort}_{\text{aux}}(-\infty)
\]
Design of an abstract domain

A lot of things are missing to turn SL into an abstract domain

Set of logical predicates:
- separation logic formulas are very expressive
  e.g., arbitrary alternations of $\land$ and $\ast$
- such expressiveness is not necessarily required in static analysis

Representation:
- unstructured formulas can be represented as ASTs,
  but this representation is not easy to manipulate efficiently
- intuition over memory states typically involves graphs

Analysis algorithms:
- inference of “optimal” invariants in SL, with numerical predicates obviously not computable
Basic abstraction: structures and their contents (1/2)

- **Concrete memory states**
  - very low level description
    numeric offsets / field names
  - pointers, numeric values:
    raw sequences of bits

\[
\begin{align*}
&(x \cdot n) = 0x\ldots a_0 \\
&(x \cdot d) = 0x\ldots a_4 \\
&(y \cdot n) = 0x\ldots b_0 \\
&(y \cdot d) = 0x\ldots b_4
\end{align*}
\]
Basic abstraction: structures and their contents (1/2)

- **Concrete memory states**

- **Abstraction of values into symbolic variables** (nodes)

- characterized by valuation $\nu$

- $\nu$ maps **symbolic variables** into **concrete addresses**

\[ \begin{align*}
\nu(\alpha_0) &= 0x...a0 \\
\nu(\alpha_1) &= 17 \\
\nu(\alpha_2) &= 0x...b0 \\
\nu(\alpha_3) &= 17 \\
\nu(\alpha_4) &= 0x0
\end{align*} \]
Basic abstraction: structures and their contents (1/2)

- **Concrete memory states**

- **Abstraction of values into symbolic variables / nodes**

- **Abstraction of regions into points-to edges**

\[ \nu(\alpha_0) = 0x...a0 \]
\[ \nu(\alpha_1) = 17 \]
\[ \nu(\alpha_2) = 0x...b0 \]
\[ \nu(\alpha_3) = 17 \]
\[ \nu(\alpha_4) = 0x0 \]
Basic abstraction: structures and their contents (1/2)

- **Concrete memory states**

- **Abstraction of values into symbolic variables / nodes**

- **Abstraction of regions into points-to edges**

- **Shape graph concretization**

  \[ \gamma_{sh}(G) = \{(h, \nu) \mid \ldots\} \]

  valuation \( \nu \) plays an important role to combine abstraction...
Valuations bridge the gap between nodes and values

**Symbolic variables / nodes** and intuitively abstract concrete values:

**Symbolic variables**

We let $\mathbb{V}^\#$ denote a countable set of **symbolic variables**; we usually let them be denoted by Greek letters in the following: $\mathbb{V}^\# = \{\alpha, \beta, \delta, \ldots\}$

When concretizing a shape graph, we need to **characterize how the concrete instance evaluates each symbolic variable**, which is the purpose of the **valuation functions**:

**Valuations**

A **valuation** is a function from **symbolic variables** into **concrete values** (and is often denoted by $\nu$): $\text{Val} = \mathbb{V}^\# \rightarrow \mathbb{V}$

Note that valuations treat **in the same way addresses** and **raw values**
A shape abstract domain relying on separation

Structure of shape graphs

Distinct edges describe separate regions

In particular, if we **split** a graph into **two parts**:

**Separating conjunction**

\[
\gamma_{sh}(S_0^\# \ast S_1^\#) = \{(h_0 \otimes h_1, \nu) \mid (h_0, \nu) \in \gamma_{sh}(S_0^\#) \land (h_1, \nu) \in \gamma_{sh}(S_1^\#)\}
\]

Similarly, when considering the **empty set of edges**, we get the empty heap (where \(V^\#\) is the set of nodes):

\[
\gamma_{sh}(\text{emp}) = \{(\emptyset, \nu) \mid \nu : V^\# \to V\}
\]
A shape abstract domain relying on separation

Abstraction of contiguous regions

A single points-to edge represents one heap cell

A points-to edge encodes basic points to predicate in separation logic:

Points-to edges

- **Syntax**

<table>
<thead>
<tr>
<th>Graph edge</th>
<th>Separation logic formula</th>
<th>Concrete view</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha \overset{f}{\rightarrow} \beta$</td>
<td>$\alpha \cdot f \leftrightarrow \beta$</td>
<td>$\nu(\alpha)$</td>
</tr>
<tr>
<td>$\text{offset}(f)$</td>
<td>$\nu(\beta)$</td>
<td></td>
</tr>
</tbody>
</table>

- **Concretization:**

$$\gamma_{sh}(\alpha \cdot f \leftrightarrow \beta) = \{([\nu(\alpha) + \text{offset}(f) \leftrightarrow \nu(\beta)], \nu) | \nu : \{\alpha, \beta, \ldots\} \rightarrow \mathbb{N}\}$$
Abstraction of contiguous regions

Contiguous regions are described by adjacent points-to edges

To describe **blocks** containing series of **cells** (e.g., in a **C structure**), shape graphs utilize several outgoing edges from the node representing the base address of the block.

**Field splitting model**

- Separation impacts edges / fields, *not pointers*

![Shape graph diagram](image)

- Shape graph accounts for both abstract states below:

  $$\nu(\alpha) \text{ offset}(f) \rightarrow \nu(\beta_0)$$
  $$\nu(\alpha) \text{ offset}(g) \rightarrow \nu(\beta_1)$$
  $$\nu(\beta_0) = \nu(\beta_1)$$

In other words, in a field splitting model, separation:

- asserts addresses are distinct
- says nothing about contents
Abstract environments

- An abstract environment is a function $e^\#$ from variables to symbolic nodes.
- The concretization extends as follows:

$$\gamma_{\text{mem}}(e^\#, S^\#) = \{(e, h, \nu) \mid (h, \nu) \in \gamma_{\text{sh}}(S^\#) \land e = \nu \circ e^\#\}$$

Environments bind variables to their (concrete / abstract) address

$$\begin{align*}
&x = \&x \cdot n = 0x...a0 \\
&\text{and } (x \cdot d) = 0x...a4 \\
\hline
&x = \&x \cdot n = 0x...b0 \\
&\text{and } (x \cdot d) = 0x...b4 \\
\hline
&y = \&y \cdot n = 0x...b0 \\
&\text{and } (y \cdot d) = 0x...b4 \\
\hline
\end{align*}$$
Basic abstraction: summarization

Set of all lists of any length:

Well-founded list inductive def.

\[ \alpha \cdot \text{list} := \]
\[ (\text{emp} \land \alpha = 0x0) \]
\[ \lor (\alpha \cdot d \mapsto \beta_0 \ast \alpha \cdot n \mapsto \beta_1 \ast \beta_1 \cdot \text{list} \land \alpha \neq 0x0) \]

Well-founded predicate

Inductive summary predicates

Concretization based on unfolding and least-fixpoint:

- \( \mathcal{U} \) replaces an \( \alpha \cdot \text{list} \) predicate with one of its premises

\[ \gamma(S^\#, F) = \bigcup \{ \gamma(S^\#_u, F_u) \mid (S^\#, F) \xrightarrow{\mathcal{U}} (S^\#_u, F_u) \} \]
As before, many interesting inductive predicates encode nicely into graph inductive definitions:

- **More complex shapes: trees**
  - ![Tree Diagram]

- **Relations among pointers: doubly-linked lists**
  - ![Doubly-Linked List Diagram]

- **Relations between pointers and numerical: sorted lists**
  - ![Sorted List Diagram]
Inductive segments

A frequent pattern:

A first attempt:
- $x$ points to a list, so $\&x \mapsto \alpha \ast \alpha \cdot \text{list}$ holds
- $y$ points to a list, so $\&y \mapsto \beta \ast \beta \cdot \text{list}$ holds

However, the following does not hold

$$\&x \mapsto \alpha \ast \alpha \cdot \text{list} \ast \&y \mapsto \beta \ast \beta \cdot \text{list}$$

Why? violation of separation!

A second attempt:

$$(\&x \mapsto \alpha \ast \alpha \cdot \text{list} \ast \text{TRUE}) \land (\&y \mapsto \beta \ast \beta \cdot \text{list} \ast \text{TRUE})$$

Why is it still not all that good? relation lost!
Inductive segments

A frequent pattern:

Could be **expressed directly** as an inductive with a parameter:

\[
\alpha \cdot \text{list\_endp}(\pi) \; ::= \; (\text{emp}, \alpha = \pi) \\
| \; (\alpha \cdot \text{next} \mapsto \beta_0 \ast \alpha \cdot \text{data} \mapsto \beta_1 \\
\ast \beta_0 \ast \text{list\_endp}(\pi), \alpha \neq 0)
\]

This definition **straightforwardly derives** from list

Thus, we make **segments** part of the **fundamental predicates of the domain**

**Multi-segments:** possible, but harder for analysis
### Shape graphs and separation logic

**Semantic preserving translation** $\Pi$ of graphs into separation logic formulas:

<table>
<thead>
<tr>
<th>Graph $S^# \in D^#_{sh}$</th>
<th>Translated formula $\Pi(S^#)$</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="Diagram" alt="Graph" /></td>
<td>$\alpha \cdot f \mapsto \beta$</td>
</tr>
<tr>
<td>$S_0^#$</td>
<td>$\Pi(S_0^#) \ast \Pi(S_1^#)$</td>
</tr>
<tr>
<td>$S_1^#$</td>
<td>$\alpha \cdot \text{list}$</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>$\alpha \cdot \text{list_endp}(\delta)$</td>
</tr>
<tr>
<td>other inductives and segments</td>
<td>similar</td>
</tr>
</tbody>
</table>

Note that:
- **shape graphs can be encoded into separation logic formula**
- **the opposite is usually not true**

**Value information:**
- discussed in the next course
- intuitively, assume we maintain numerical information next to shape graphs
Outline

1. Introduction

2. Separation Logic

3. A shape abstract domain relying on separation

4. Standard static analysis algorithms
   - Overview of the analysis
   - Post-conditions and unfolding
   - Folding: widening and inclusion checking
   - Abstract interpretation framework: assumptions and results
   - Comparing Separation Logic and Three-Valued logic abstractions

5. Combining shape and value abstractions

6. Conclusion
Static analysis overview

A list insertion function:

- list ** l assumed to point to a list
- list ** t assumed to point to a list element
- list ** c = l;
- while (c != NULL && c -> next != NULL && (...)){
  - c = c -> next;
}
- t -> next = c -> next;
- c -> next = t;

- list inductive structure def.
- Abstract precondition:

Result of the (interprocedural) analysis

- Over-approximations of reachable concrete states
  - e.g., after the insertion:
Transfer functions

Abstract interpreter design

- **Follows the semantics** of the language under consideration
- The abstract domain should provide **sound transfer functions**

Transfer functions:

- **Assignment**: $x \rightarrow f = y \rightarrow g$ or $x \rightarrow f = e_{\text{arith}}$
- **Test**: analysis of conditions (if, while)
- Variable **creation and removal**
- **Memory management**: `malloc`, `free`

Abstract operators:

- **Join** and **widening**: over-approximation
- **Inclusion checking**: check stabilization of abstract iterates

Should be **sound** *i.e.*, not forget any concrete behavior
Abstract operations

Denotational style abstract interpreter

- Concrete **denotational semantics** \( [b] : S \rightarrow \mathcal{P}(S) \)
- **Abstract post-condition** \( [b]^\#(S) \), computed by the analysis:
  \[
s \in \gamma(S) \implies [b](s) \subseteq \gamma([b]^\#(S))\]

Analysis by induction on the syntax using **domain operators**

\[
\begin{align*}
[b_0; b_1]^\#(S) & = \ [b_1]^\# \circ [b_0]^\#(S) \\
[1 = e]^\#(S) & = \ assign(1, e, S) \\
[1 = \text{malloc}(n)]^\#(S) & = \ alloc(1, n, S) \\
[\text{free}(1)]^\#(S) & = \ free(1, n, S) \\
[\text{if}(e) \ b_t \ \text{else} \ b_f]^\#(S) & = \ \{ \ join([b_t]^\#(\text{test}(e, S))), \\
& \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \四是
The algorithms underlying the transfer functions

- **Unfolding**: cases analysis on summaries
  
  \[ \begin{align*}
  x \text{ list} & \rightarrow y \text{ list} \\
  \implies & \\
  x \text{ list} & \rightarrow y \text{ next} \text{ list} \\
  \lor & \\
  x \text{ list} & \rightarrow y \text{ 0x0} 
  \end{align*} \]

- **Abstract postconditions**, on “exact” regions, e.g. insertion

- **Widening**: builds summaries and ensures termination

\[ \begin{align*}
  x \text{ list} & \rightarrow y \text{ list} \\
  \triangledown & \\
  \implies & \\
  x \text{ list} & \rightarrow y \text{ next} \text{ list} \\
  \lor & \\
  x \text{ list} & \rightarrow y \text{ list} 
  \end{align*} \]
Outline

1 Introduction

2 Separation Logic

3 A shape abstract domain relying on separation

4 Standard static analysis algorithms
   - Overview of the analysis
   - Post-conditions and unfolding
   - Folding: widening and inclusion checking
   - Abstract interpretation framework: assumptions and results
   - Comparing Separation Logic and Three-Valued logic abstractions

5 Combining shape and value abstractions

6 Conclusion
Standard static analysis algorithms
Post-conditions and unfolding

Analysis of an assignment in the graph domain

Steps for analyzing $x = y \rightarrow \text{next}$ (local reasoning)

1. Evaluate **l-value** $x$ into **points-to edge** $\alpha \mapsto \beta$
2. Evaluate **r-value** $y \rightarrow \text{next}$ into **node** $\beta'$
3. Replace points-to edge $\alpha \mapsto \beta$ with **points-to edge** $\alpha \mapsto \beta'$

With pre-condition:

- Step 1 produces $\alpha_0 \mapsto \beta_0$
- Step 2 produces $\beta_2$
- End result:

With pre-condition:

- Step 1 produces $\alpha_0 \mapsto \beta_0$
- Step 2 fails
- Abstract state **too abstract**
- We need to **refine it**
Unfolding as a local case analysis

Unfolding principle

- **Case analysis**, based on the inductive definition
- Generates *symbolic disjunctions* (analysis performed in a **disjunction domain**, e.g., trace partitioning)

- Example, for lists:

  ![Unfolding example for lists](image)

  - **Numeric predicates**: next course on shape + value abstraction

Soundness: by definition of the concretization of inductive structures

\[ \gamma_{sh}(S^\#) \subseteq \bigcup \{ \gamma_{sh}(S_0^\#) \mid S^\# \xrightarrow{U} S_0^\# \} \]
Analysis of an assignment, with unfolding

**Principle**

- We have \( \gamma_{sh}(\alpha \cdot \iota) = \bigcup \{ \gamma_{sh}(S^\#) \mid \alpha \cdot \iota \xrightarrow{\mathcal{U}} S^\# \} \)
- Replace \( \alpha \cdot \iota \) with a finite number of disjuncts and continue

**Disjunct 1:**

- Step 1 produces \( \alpha_0 \mapsto \beta_0 \)
- **Step 2 fails:** Null pointer!
- In a **correct** program, would be ruled out by a condition \( y \neq 0 \) i.e., \( \beta_1 \neq 0 \) in \( \mathbb{D}_\text{num}^\# \)

**Disjunct 2:**

- Step 1 produces \( \alpha_0 \mapsto \beta_0 \)
- Step 2 produces \( \beta_2 \)
- **End result:**
Standard static analysis algorithms

Post-conditions and unfolding

Unfolding and degenerated cases

**assume** (l points to a dll)

\[
c = l;
\]

1. **while** (c \(!=\) NULL && condition)
   \[
c = c \rightarrow \text{next};
   \]

2. **if** (c \(!=\) 0 && c \(\rightarrow\) prev \(!=\) 0)
   \[
c = c \rightarrow \text{prev} \rightarrow \text{prev};
   \]

\[
\begin{align*}
\text{at } 1: & \quad \alpha_0 \quad \text{dll}(\delta_1) \\
\text{at } 2: & \quad \alpha_0 \quad \text{dll}(\delta_0) \quad \alpha_1 \quad \text{dll}(\delta_1) \quad \text{dll}(\alpha_0) \\
\Rightarrow & \quad \text{non trivial unfolding}
\end{align*}
\]

**Materialization of** \(c \rightarrow \text{prev}:\)

Segment splitting lemma: basis for segment unfolding

\[
\alpha_0 \quad l \quad i+j \quad \alpha_2
\]

\[
\text{describes the same set of stores as}
\alpha_0 \quad l \quad i \quad \alpha_1 \quad l'' \quad j \quad \alpha_2
\]

**Materialization of** \(c \rightarrow \text{prev} \rightarrow \text{prev}:\)

**Implementation issue:** discover **which inductive edge** to unfold very hard!

Xavier Rival (INRIA)

Shape analysis abstractions

Jan, 17th, 2022

47 / 94
Outline

1 Introduction

2 Separation Logic

3 A shape abstract domain relying on separation

4 Standard static analysis algorithms
   - Overview of the analysis
   - Post-conditions and unfolding
   - Folding: widening and inclusion checking
   - Abstract interpretation framework: assumptions and results
   - Comparing Separation Logic and Three-Valued logic abstractions

5 Combining shape and value abstractions

6 Conclusion
Need for a folding operation

Back to the list traversal example:

First iterates in the loop:
- at iteration 0 (before entering the loop):
  \[ l, c \]

- at iteration 1:

- at iteration 2:

The analysis unfolds, but never folds:

How to guarantee termination of the analysis?
How to introduce segment edges / perform abstraction?
Widening

- The lattice of shape abstract values has **infinite height**
- Thus iteration sequences **may not terminate**

**Definition of a widening operator \( \nabla \)**

- **Over-approximates join:**

\[
\begin{align*}
\gamma(X^\#) \subseteq & \quad \gamma(X^\# \nabla Y^\#) \\
\gamma(Y^\#) \subseteq & \quad \gamma(X^\# \nabla Y^\#)
\end{align*}
\]

- **Enforces termination:** for all sequence \( (X_n^\#)_{n \in \mathbb{N}} \), the sequence \( (Y_n^\#)_{n \in \mathbb{N}} \) defined below is ultimately stationary

\[
\begin{align*}
\forall n \in \mathbb{N}, \quad Y_{n+1}^\# &= Y_n^\# \nabla X_{n+1}^\#
\end{align*}
\]
Canonicalization

Upper closure operator

\( \rho : \mathbb{D}^\# \rightarrow \mathbb{D}^\#_{\text{can}} \subseteq \mathbb{D}^\# \) is an upper closure operator (uco) iff it is monotone, extensive and idempotent.

Canonicalization

- **Disjunctive completion**: \( \mathbb{D}^\#_{\lor} = \) finite disjunctions over \( \mathbb{D}^\# \)
- **Canonicalization operator** \( \rho_{\lor} \) defined by \( \rho_{\lor} : \mathbb{D}^\#_{\lor} \rightarrow \mathbb{D}^\#_{\text{can}_\lor} \) and
  \[ \rho_{\lor}(X^\#) = \{ \rho(x^\#) \mid x^\# \in X^\# \} \]
  where \( \rho \) is an uco and \( \mathbb{D}^\#_{\text{can}} \) is finite.

- Canonicalization is used in **many shape analysis tools**
- **Easier to compute** but **less powerful** than widening: does not exploit history of computation
Weakening: definition

To design inclusion test, join and widening algorithms, we first study a more general notion of weakening:

Weakening

We say that $S_0^\#$ can be weakened into $S_1^\#$ if and only if

$$\forall (h, \nu) \in \gamma_{sh}(S_0^\#), \exists \nu' \in \text{Val}, (h, \nu') \in \gamma_{sh}(S_1^\#)$$

We then note $S_0^\# \preceq S_1^\#$

Applications:

- **inclusion test** (comparison) inputs $S_0^\#, S_1^\#$; if returns true $S_0^\# \preceq S_1^\#$
- **canonicalization** (unary weakening) inputs $S_0^\#$ and returns $\rho(S_0^\#)$ such that $S_0^\# \preceq \rho(S_0^\#)$
- **widening / join** (binary weakening ensuring termination or not) inputs $S_0^\#, S_1^\#$ and returns $S_{up}^\#$ such that $S_i^\# \preceq S_{up}^\#$
Weakening: example

We consider $S_0^\#$ defined by:

\[
\begin{align*}
\alpha_0 & \xrightarrow{\& x} \alpha_1 \\
& \xrightarrow{\text{next}} \alpha_2 \\
& \xrightarrow{\text{data}} \alpha_3
\end{align*}
\]

and $S_1^\#$ defined by:

\[
\begin{align*}
\beta_0 & \xrightarrow{\& x} \beta_1 \\
& \xrightarrow{\text{list}} \beta
\end{align*}
\]

Then, we have the weakening $S_0^\# \preceq S_1^\#$ up-to a renaming in $S_1^\#$:

\[
\Psi : \begin{align*}
\beta_0 & \mapsto \alpha_0 \\
\beta_1 & \mapsto \alpha_1
\end{align*}
\]

- weakening up-to renaming is generally required as graphs do not have the same name space
- formalized a bit later...
Local weakening: separating conjunction rule

We can apply the local reasoning principle to weakening

If \( S_0^\# \preceq S_0^{\#, \text{weak}} \) and \( S_1^\# \preceq S_1^{\#, \text{weak}} \) then:

\[
\begin{array}{c}
\alpha_0 \quad S_0^\# \quad \alpha_1 \\
\alpha_1 \quad S_1^\# \quad \alpha_2
\end{array} \preceq
\begin{array}{c}
\alpha_0 \quad S_{0, \text{weak}}^\# \\
\alpha_1 \quad S_{1, \text{weak}}^\# \\
\alpha_2
\end{array}
\]

Separating conjunction rule (\( \preceq \ast \))

Let us assume that

- \( S_0^\# \) and \( S_1^\# \) have distinct set of source nodes
- we can weaken \( S_0^\# \) into \( S_{0, \text{weak}}^\# \)
- we can weaken \( S_1^\# \) into \( S_{1, \text{weak}}^\# \)

then:

we can weaken \( S_0^\# \ast S_1^\# \) into \( S_{0, \text{weak}}^\# \ast S_{1, \text{weak}}^\# \)
Local weakening: unfolding rule, identity rule

Weakening unfolded region ($\preceq_U$)

Let us assume that $S_0^\# \xrightarrow{U} S_1^\#$. Then, by definition of the concretization of unfolding

we can weaken $S_1^\#$ into $S_0^\#$

- the proof follows from the definition of unfolding
- it can be applied locally, on graph regions that differ due to unfolding of inductive definitions

Identity weakening ($\preceq_{Id}$)

we can weaken $S^\#$ into $S^\#$

- the proof is trivial:

$$\gamma_{sh}(S^\#) \subseteq \gamma_{sh}(S^\#)$$

- on itself, this principle is not very useful, but it can be applied locally, and combined with ($\preceq_U$) on graph regions that are not equal
Local weakening: example

By \textbf{rule \((\preceq_{\mathcal{U}})\):}

Additionally, by \textbf{rule \((\preceq_{\text{Id}})\):}

Thus, by \textbf{rule \((\preceq_{\ast})\):}
Inclusion checking rules in the shape domain

Graphs to compare have distinct sets of nodes, thus inclusion check should carry out a **valuation transformer** $\Psi : \forall^\#(S_1^\#) \longrightarrow \forall^\#(S_0^\#)$ (important when dealing also with content values)

Using (and extending) the weakening principles, we obtain the following rules (considering only inductive definition list, though these rules would extend to other definitions straightforwardly):

- **Identity rules:**
  \[
  \forall i, \quad \Psi(\beta_i) = \alpha_i \quad \Longrightarrow \quad \alpha_0 \cdot f \mapsto \alpha_1 \quad \sqsubseteq^\#_\Psi \quad \beta_0 \cdot f \mapsto \beta_1
  \]
  \[
  \Psi(\beta) = \alpha \quad \Longrightarrow \quad \alpha \cdot \text{list} \quad \sqsubseteq^\#_\Psi \quad \beta \cdot \text{list}
  \]
  \[
  \forall i, \quad \Psi(\beta_i) = \alpha_i \quad \Longrightarrow \quad \alpha_0 \cdot \text{list$_{\text{endp}}$(\alpha_1)} \quad \sqsubseteq^\#_\Psi \quad \beta_0 \cdot \text{list$_{\text{endp}}$(\beta_1)}
  \]

- **Rules on inductives:**
  \[
  \forall i, \quad \Psi(\beta_i) = \alpha \quad \Longrightarrow \quad \text{emp} \quad \sqsubseteq^\#_\Psi \quad \beta_0 \cdot \text{list$_{\text{endp}}$(\beta_1)}
  \]
  \[
  S_0^\# \sqsubseteq^\#_\Psi S_1^\# \land \beta \cdot \iota \xrightarrow{U} S_1^\# \quad \Longrightarrow \quad S_0^\# \sqsubseteq^\#_\Psi \beta \cdot \iota
  \]
  if $\beta_1$ fresh, $\Psi' = \Psi[\beta_1 \mapsto \alpha_1]$ and $\Psi(\beta_0) = \alpha_0$ then,
  \[
  S_0^\# \sqsubseteq^\#_\Psi \beta_1 \cdot \text{list} \quad \Longrightarrow \quad \alpha_0 \cdot \text{list$_{\text{endp}}$(\alpha_1)} \ast S_0^\# \quad \sqsubseteq^\#_\Psi \quad \beta_0 \cdot \iota$
Inclusion checking algorithm

Comparison of \((e_0^\#, S_0^\#)\) and \((e_1^\#, S_1^\#)\)

1. Start with \(\Psi\) defined by \(\Psi(\beta) = \alpha\) if and only if there exists a variable \(x\) such that \(e_0^\#(x) = \alpha \land e_1^\#(x) = \beta\).

2. Iteratively apply local rules, and extend \(\Psi\) when needed.

3. Return true when both shape graphs become empty.

- The first step ensures both environments are consistent.

This algorithm is sound:

**Soundness**

\[
(e_0^\#, S_0^\#) \sqsubseteq^\# \Psi (e_1^\#, S_1^\#) \implies \gamma(e_0^\#, S_0^\#) \subseteq \gamma(e_1^\#, S_1^\#)
\]
Over-approximation of union

The principle of join and widening algorithm is similar to that of $\sqsubseteq^\#$:

- It can be computed region by region, as for weakening in general:
  If $\forall i \in \{0, 1\}$, $\forall s \in \{lft, rgh\}$, $S_{i,s}^\# \sqsubseteq S_s^\#$,

\[
\begin{align*}
S_{0,lft}^\# & \to S_{1,lft}^\# \\
S_{0,rgh}^\# & \to S_{1,rgh}^\#
\end{align*}
\]

The partitioning of inputs / different nodes sets requires a node correspondence function

\[
\psi : \forall^\#(S_{lft}^\#) \times \forall^\#(S_{rgh}^\#) \rightarrow \forall^\#(S^\#)
\]

- The computation of the shape join progresses by the application of local join rules, that produce a new (output) shape graph, that weakens both inputs
Over-approximation of union: syntactic identity rules

In the next few slides, we focus on $\nabla$
though the abstract union would be defined similarly in the shape domain

Several rules derive from $(\preceq_{Id})$:

- If $S^\#_{\text{lt}} = \alpha_0 \cdot f \mapsto \alpha_1$
  and $S^\#_{\text{rg}} = \beta_0 \cdot f \mapsto \beta_1$
  and $\Psi(\alpha_0, \beta_0) = \delta_0$, $\Psi(\alpha_1, \beta_1) = \delta_1$, then:

  $$S^\#_{\text{lt}} \nabla S^\#_{\text{rg}} = \delta_0 \cdot f \mapsto \delta_1$$

- If $S^\#_{\text{lt}} = \alpha_0 \cdot \text{list}$
  and $S^\#_{\text{rg}} = \beta_0 \cdot \text{list}_1$
  and $\Psi(\alpha_0, \beta_0) = \delta_0$, then:

  $$S^\#_{\text{lt}} \nabla S^\#_{\text{rg}} = \delta_0 \cdot \text{list}$$
Over-approximation of union: segment introduction rule

Rule

\[
\begin{align*}
\text{if } & \quad \Psi \subseteq S_{\text{lt}}^\# S_{\text{rg}}^\# \quad \text{then} \\
& \quad S_{\text{lt}}^\# \vee S_{\text{rg}}^\# = \delta_0 \text{ list } \\
& \quad (\alpha, \beta_0) \leftrightarrow \delta_0 \\
& \quad (\alpha, \beta_1) \leftrightarrow \delta_1
\end{align*}
\]

Application to list traversal, at the end of iteration 1:

- before iteration 0:
  \[
  \begin{array}{c}
  \alpha_0 \\
  1, c
  \end{array}
  \]

- end of iteration 0:
  \[
  \begin{array}{c}
  \beta_0 \\
  1, c \\
  \text{data} \\
  \beta_2
  \end{array}
  \]

- join, before iteration 1:
  \[
  \begin{array}{c}
  \delta_0 \\
  1, c
  \end{array}
  \]
  \[
  \begin{array}{c}
  \delta_1 \\
  1, c
  \end{array}
  \]
  \[
  \begin{array}{c}
  \Psi(\alpha_0, \beta_0) = \delta_0 \\
  \Psi(\alpha_0, \beta_1) = \delta_1
  \end{array}
  \]
Over-approximation of union: segment extension rule

Rule

\[
\text{if } S^\#_{\text{lf}} \subseteq S^\#_{\text{rgh}} \text{ then } \begin{cases} 
S^\#_{\text{lf}} \nabla S^\#_{\text{rgh}} = (\alpha_0, \beta_0) \leftrightarrow \delta_0 \\
(\alpha_1, \beta_1) \leftrightarrow \delta_1 
\end{cases}
\]

Application to list traversal, at the end of iteration 1:

- previous invariant before iteration 1:

- end of iteration 1:

- join, before iteration 1:

\[
\begin{align*}
\Psi(\alpha_0, \beta_0) &= \delta_0 \\
\Psi(\alpha_1, \beta_2) &= \delta_1 
\end{align*}
\]
Over-approximation of union: rewrite system properties

- Comparison, canonicalization and widening algorithms can be considered **rewriting systems over tuples of graphs**
- **Success configuration**: weakening applies on all components, i.e., the inputs are fully “consumed” in the weakening process
- **Failure configuration**: some components **cannot be weakened** i.e., the algorithm should return the conservative answer (i.e., \(\top\))

**Termination**
- The systems are **terminating**
- This ensures comparison, canonicalization, widening are **computable**

**Non confluence!**
- The results depends on the order of application of the rules
- Implementation requires the choice of an **adequate strategy**
Over-approximation of union in the combined domain

Widening of \((e^\#, S^\#_0)\) and \((e^\#, S^\#_1)\)

1. define \(\Psi, e\) by \(\Psi(\alpha, \beta) = e(x) = \delta\) (where \(\delta\) is a fresh node) if and only if \(e^\#_0(x) = \alpha \land e^\#_1(x) = \beta\)
2. iteratively apply **join local rules**, and extend \(\Psi\) when new relations are inferred (for instance for points-to edges)
3. return the result obtained when all regions of both inputs are approximated in the output graph

This algorithm is sound:

**Soundness**

\[ \gamma(e^\#, S^\#_0) \cup \gamma(e^\#, S^\#_1) \subseteq \gamma(e^\#, S^\#) \]

Widening also enforces **termination** (it only introduces segments, and the growth induced by the introduction of segments is bounded)
Outline

1. Introduction

2. Separation Logic

3. A shape abstract domain relying on separation

4. Standard static analysis algorithms
   - Overview of the analysis
   - Post-conditions and unfolding
   - Folding: widening and inclusion checking
   - Abstract interpretation framework: assumptions and results
   - Comparing Separation Logic and Three-Valued logic abstractions

5. Combining shape and value abstractions

6. Conclusion
Assumptions

What assumptions do we make? How do we prove soundness of the analysis of a loop?

- **Assumptions in the concrete level**, and for block $b$:

  \[(\mathcal{P}(M), \subseteq)\] is a complete lattice, hence a CPO

  \[F : \mathcal{P}(M) \rightarrow \mathcal{P}(M)\] is the concrete semantic ("post") function of $b$

  thus, the concrete semantics writes down as $[b] = \text{lfp}_0 F$

- **Assumptions in the abstract level**:

  - $M^\#$ set of abstract elements, no order a priori
    \[m^\# ::= (e^\#, S^\#)\]
  - $\gamma_{\text{mem}} : M^\# \rightarrow \mathcal{P}(M)$ concretization
  - $F^\# : M^\# \rightarrow M^\#$ sound abstract semantic function
    \[i.e., \text{such that } F \circ \gamma_{\text{mem}} \subseteq \gamma_{\text{mem}} \circ F^\#\]
  - $\triangledown : M^\# \times M^\# \rightarrow M^\#$ widening operator, terminates, and such that
    \[\gamma_{\text{mem}}(m_0^\#) \cup \gamma_{\text{mem}}(m_1^\#) \subseteq \gamma_{\text{mem}}(m_0^\# \triangledown m_1^\#)\]
Computing a loop abstract post-condition

**Loop abstract semantics**

The abstract semantics of loop `while(rand()){b}` is calculated as the limit of the sequence of abstract iterates below:

\[
\begin{align*}
  m_0^\# &= \bot \\
  m_{n+1}^\# &= m_n^\# \triangledown F^\#(m_n^\#)
\end{align*}
\]

**Soundness proof:**

- by induction over \( n \), \( \bigcup_{k \leq n} F^k(\emptyset) \subseteq \gamma_{\text{mem}}(m_n^\#) \)
- by the property of widening, the abstract sequence converges at a rank \( N \):
  \( \forall k \geq N, \ m_k^\# = m_N^\# \), thus

\[
\text{lfp}_\emptyset F = \bigcup_k F^k(\emptyset) \subseteq \gamma_{\text{mem}}(m_N^\#)
\]
Discussion on the abstract ordering

How about the abstract ordering? We assumed \textit{NONE} so far...

- **Logical ordering**, induced by concretization, used for \textit{proofs}
  \[
m_0 \not\subseteq m_1 \iff \gamma_{\text{mem}}(m_0) \not\subseteq \gamma_{\text{mem}}(m_1)
\]

- **Approximation of the logical ordering**, implemented as a function
  \texttt{is\_le : } M^\# \times M^\# \rightarrow \{\text{true, } \top\},
  used to test the convergence of abstract iterates
  \[
  \text{is\_le}(m_0, m_1) = \text{true} \implies \gamma_{\text{mem}}(m_0) \subseteq \gamma_{\text{mem}}(m_1)
  \]

Abstract semantics is not assumed (and is actually most likely \textit{NOT}) monotone with respect to either of these orders...

- Also, \textit{computational ordering} would be used for \textit{proving widening termination}
Outline

1 Introduction

2 Separation Logic

3 A shape abstract domain relying on separation

4 Standard static analysis algorithms
   - Overview of the analysis
   - Post-conditions and unfolding
   - Folding: widening and inclusion checking
   - Abstract interpretation framework: assumptions and results
   - Comparing Separation Logic and Three-Valued logic abstractions

5 Combining shape and value abstractions

6 Conclusion
Separation logic

Separation logic formulas (main connectors only)

\[
F ::= \text{emp} \\
| \text{TRUE} \\
| l \mapsto l \\
| F_0 \ast F_1 \\
| F_0 \land F_1 \\
| F_0 \rightarrow F_1
\]

Concretization:

\[
\begin{align*}
\gamma(\text{emp}) &= E \times \{[]\} \\
\gamma(\text{TRUE}) &= E \times \mathbb{H} \\
\gamma(l \mapsto v) &= \{(e, [[1][e, h] \mapsto v]) \mid e \in E\} \\
\gamma(F_0 \ast F_1) &= \{(e, h_0 \oplus h_1) \mid (e, h_0) \in \gamma(F_0) \land (e, h_1) \in \gamma(F_1)\} \\
\gamma(F_0 \land F_1) &= \gamma(F_0) \cap \gamma(F_1) \\
\gamma(F_0 \rightarrow F_1) &= \text{exercise}
\end{align*}
\]

Program reasoning: frame rule and strong updates
# Shape graphs and separation logic

**Shape graphs**: provide an efficient data-structure to describe a **subset** of separation logic predicates, and do static analysis with them.

**Important addition**: **inductive predicates**.

**Semantic preserving translation** $\Pi$ of graphs into separation logic formulas:

<table>
<thead>
<tr>
<th>Graph $S^\sharp \in \mathbb{D}^\sharp_{sh}$</th>
<th>Translated formula $\Pi(S^\sharp)$</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Graph" /> $S_0^\sharp$ <img src="image" alt="Graph" /> $S_1^\sharp$</td>
<td>$\Pi(S_0^\sharp) \ast \Pi(S_1^\sharp)$</td>
</tr>
<tr>
<td><img src="image" alt="Graph" /> $\alpha \cdot \mathsf{list}$</td>
<td>$\alpha \cdot \mathsf{list}$</td>
</tr>
<tr>
<td><img src="image" alt="Graph" /> $\alpha \cdot \mathsf{list} \mathsf{endp}(\delta)$</td>
<td></td>
</tr>
</tbody>
</table>

| other inductives and segments | similar |

Note that:

- shape graphs can be encoded into separation logic formula
- the opposite is usually not true
Comparing the structure of abstract formulae

**Separation logic:**

\[ F_0 \ast F_1 \ast \ldots \ast F_n \]

- first the heap is partitioned
- each region is described separately
- some of the \( F_i \) components may be summary predicates, describing unbounded regions
- reachability is implicit
- allows local reasoning

**Three valued logic:**

\[ p_0 \land p_1 \land \ldots \land p_n \]

- first a conjunction of properties
- each predicate \( p_i \) may talk about any heap region
- no direct heap partitioning
- reachability can be expressed (natively)
- no local reasoning

Two very different sets of predicates

- one allows local reasoning, the other not
- the other way for reachability predicates
Summarization: one abstract cell, many concrete cells

- **Dynamic structures** (lists, trees) have an unknown and unbounded number of cells, hence require summarization
- We also needed summaries to deal with **arrays**

### Summary

A **summary predicate** allows to describe an **unbounded number** of memory locations using a fixed, finite set of predicates

**Principles underlying summarization:**

- **in separation logic:**
  - using inductive definitions for lists, trees...
  - unbounded size of the summarized region is hidden in the **recursion**
- **in three-valued logic:**
  - summary nodes + high level predicates (such as reachability)
  - one summary node **carries the properties** of an unbounded number of cells
Concretize partially, update, abstract

For precise analysis, summaries need to be (temporarily) refined

Separation logic:

Local (partial) concretization

For materialization:

\[ S_{\text{pre}}^\# \xrightarrow{\text{unfold}} S_{\text{pre,ref}}^\# \xrightarrow{f} S_{\text{post}}^\# \]

In both cases, two mechanisms are needed:

1. **refine** summaries
2. **synthesize** summaries

TVLA:

Focus, analyze, canonicalize

Global abstraction: widening

\[ S_0^\# \xrightarrow{\nabla} S_1^\# \]

\[ S_0^\# \xrightarrow{\nabla} S_1^\# \]

\[ S_{\text{pre}}^\# \xrightarrow{\text{abstract}} S_{\text{post}}^\# \]

\[ S_{\text{pre}}^\# \xrightarrow{f} S_{\text{post}}^\# \]
1 Introduction

2 Separation Logic

3 A shape abstract domain relying on separation

4 Standard static analysis algorithms

5 Combining shape and value abstractions
   - Shape and value properties
   - Combined abstraction with cofibered abstract domain
   - Combined analysis algorithms

6 Conclusion
Combining shape and value abstractions

Shape and value properties

Common data-structures require to reason both about shape and data:

- **hybrid stores**: data stored next to inductive structures
- **list of even elements**:

  ```
  \&x
  68
  24
  0
  112
  0x0
  ```

- **sorted list**:

  ```
  \&x
  8
  9
  0x0
  33
  0
  ```

- **list with a length constraint**
- **tries**: binary trees with paths labelled with sequences of “0” and “1”
- **balanced trees**: red-black, AVL...

This part of the course:

- **how to express both shape and numerical properties**?
- **how to extend shape analysis algorithms**
Description of a sorted list

- **Example:** sorted list

\[8 \rightarrow 9 \rightarrow 0x0\]

Inductive definition

- Each element should be greater than the previous one
- The first element simply needs be greater than \(-\infty\)...
- We need to propagate the lower bound, using a scalar parameter

\[
\alpha \cdot \text{lsort}_{\text{aux}}(n) := \begin{cases} 
\alpha = 0 \land \text{emp} \\
\alpha \neq 0 \land n \leq \beta \land \alpha \cdot \text{next} \mapsto \delta \\
\star \alpha \cdot \text{data} \mapsto \beta \star \delta \cdot \text{lsort}_{\text{aux}}(\beta)
\end{cases} 
\]

\[
\alpha \cdot \text{lsort}() := \alpha \cdot \text{lsort}_{\text{aux}}(-\infty)
\]
Adding value information (here, numeric)

Concrete numeric values appear in the valuation thus the abstracting contents boils down to abstracting $\nu$!

**Example:** all lists of length 2, sorted in the increasing order of data fields

**Memory abstraction:**

![Diagram](image)

**Abstraction of valuations:** $\nu(\alpha_1) < \nu(\alpha_3)$, can be described by the constraint $\alpha_1 < \alpha_3$
A first step towards a combined domain

Domains and their concretization:
- **shape abstract domain** $\mathbb{D}_{sh}^\#$ of graphs
  abstract stores together with a **physical mapping** of nodes
  
  \[
  \gamma_{sh} : \mathbb{D}_{sh}^\# \rightarrow \mathcal{P}((\mathbb{D}_{sh}^\# \rightarrow \mathbb{M}) \times (\mathbb{V}_{sh}^\# \rightarrow \mathbb{V}))
  \]

- **numerical abstract domain** $\mathbb{D}_{num}^\#$, abstracts physical mapping of nodes

  \[
  \gamma_{num} : \mathbb{D}_{num}^\# \rightarrow \mathcal{P}(\mathbb{V}_{num}^\# \rightarrow \mathbb{V})
  \]

Combined domain [CR]

- **Set of abstract values**: $\mathbb{D}^\# = \mathbb{D}_{sh}^\# \times \mathbb{D}_{num}^\#

- **Concretization**:

  \[
  \gamma(S^\#, N^\#) = \{(h, \nu) \in \mathbb{M} \mid \nu \in \gamma_{num}(N^\#) \land (h, \nu) \in \gamma_{sh}(S^\#)\}
  \]

Can it be described as a reduced product?

- **product abstraction**: $\mathbb{D}^\# = \mathbb{D}_0^\# \times \mathbb{D}_1^\#$ (componentwise ordering)

- **concretization**: $\gamma(x_0, x_1) = \gamma(x_0) \cap \gamma(x_1)$

- **reduction**: $\mathbb{D}_r^\#$ is the quotient of $\mathbb{D}^\#$ by the equivalence relation $\equiv$ defined by

  \[
  (x_0, x_1) \equiv (x'_0, x'_1) \iff \gamma(x_0, x_1) = \gamma(x'_0, x'_1)
  \]
Formalizing the product domain

The use of a simple reduced product raises several issues

Elements without a clear meaning:

- this element exists in the reduced product domain (independent components)
- but, ... what is $\alpha_3$?

Unclear comparison:
How can we compare the two elements below?

- in the reduced product domain, they are not comparable: nodes do not match, so componentwise comparison does not make sense
- when concretizing them, there is clear inclusion
Towards a more adapted combination operator

Reason why the reduced product construction does not work well:
- the set of nodes / symbolic variables is not fixed
- the set of dimensions in the numerical domain depends on the shape abstraction

⇒ thus the product is not symmetric
however, the reduced product construction is symmetric

Intuitions
- Graphs form a shape domain $D_{\text{sh}}^\#$
- For each graph $S^\# \in D_{\text{sh}}^\#$, we have a numerical lattice $D_{\text{num}}^\langle S^\# \rangle$
  - example: if graph $S^\#$ contains nodes $\alpha_0, \alpha_1, \alpha_2$, $D_{\text{num}}^\langle S^\# \rangle$ should abstract $\{\alpha_0, \alpha_1, \alpha_2\} \rightarrow \forall$
- An abstract value is a pair $(S^\#, N^\#)$, such that $N^\# \in D_{\text{num}}^\langle N^\# \rangle$
Outline

1. Introduction

2. Separation Logic

3. A shape abstract domain relying on separation

4. Standard static analysis algorithms

5. **Combining shape and value abstractions**
   - Shape and value properties
   - Combined abstraction with cofibered abstract domain
   - Combined analysis algorithms

6. Conclusion
Cofibered domain

**Definition, for shape + num**

- **Basis:** abstract domain \((D^\#_{sh}, \sqsubseteq^\#)\), with concretization
  \(\gamma_{sh} : D^\#_{sh} \rightarrow D\)

- **Function:** \(\phi : D^\#_{sh} \rightarrow D\), where each element of \(D\) is an
  abstract domain instance \((D^\#_{num}, \sqsubseteq^\#_{num})\), with a
  concretization \(\gamma_{num} : D^\#_{num} \rightarrow D\) (tied to a shape
  graph)

- **Domain** \(D^\#\): set of *pairs* \((S^\#, N^\#)\) where \(N^\# \in \phi(S^\#)\)

- **Concretization:** \(\gamma(S^\#, N^\#) = \gamma(S^\#) \cap \gamma(N^\#)\)

- **Lift functions:** \(\forall S^\#_0, S^\#_1 \in D^\#_{sh}, \text{ such that } S^\#_0 \sqsubseteq^\# S^\#_1\), there
  exists a function \(\prod_{S^\#_0, S^\#_1} : \phi(S^\#_0) \rightarrow \phi(S^\#_1)\), that is
  monotone for \(\gamma_{S^\#_0}\) and \(\gamma_{S^\#_1}\)

- General construction presented in [AV](Arnaud Venet)

- Intuition: a dependent domain product
Overall abstract domain structure

Implementation exploiting the modular structure

- Each layer accounts for one aspect of the concrete states
- Each layer boils down to a module or functor in ML

How about operations, transfer functions? Also to be modularly defined
Domain operations

The cofibered structure allows to define standard domain operations:

- **ift functions** allow to **switch domain when needed**
- computations first done in the basis, then in the numerical domains, after lifting, when needed

### Comparison of \((S_0^\#, N_0^\#)\) and \((S_1^\#, N_1^\#)\)

1. First, compare \(S_0^\#\) and \(S_1^\#\) in \(D_{sh}^\#\)
2. If \(S_0^\# \sqsubseteq S_1^\#\), compare \(\Pi_{S_0^\#, S_1^\#}(N_0^\#)\) and \(N_1^\#\)

### Widening of \((S_0^\#, N_0^\#)\) and \((S_1^\#, N_1^\#)\)

1. First, compute the **widening in the basis** \(S^\# = S_0^\# \lor S_1^\#\)
2. Then move to \(\phi(S^\#)\), by computing \(N_{0c}^\# = \Pi_{S_0^\#, S_1^\#}(N_0^\#)\) and \(N_{1c}^\# = \Pi_{S_1^\#, S_1^\#}(N_1^\#)\)
3. Last widen in \(\phi(S^\#)\): \(N^\# = N_{0c}^\# \lor S_1^\# N_{1c}^\#\)
4. Return \((S_0^\#, N_0^\#) \lor (S_1^\#, A^\#) = (S^\#, N^\#)\)
Outline

1 Introduction

2 Separation Logic

3 A shape abstract domain relying on separation

4 Standard static analysis algorithms

5 Combining shape and value abstractions
   - Shape and value properties
   - Combined abstraction with cofibered abstract domain
   - Combined analysis algorithms

6 Conclusion
Domain operations and transfer functions

Abstract assignments, condition tests:
- need to modify both the shape abstraction and the value abstraction
- both modification are interdependent

Typical process to compute abstract post-conditions
1. compute the post in the shape abstract domain and update the basis
2. update the value abstraction (numerics) to model dimensions additions and removals
3. compute the post in the value abstract domain

Proofs of soundness of transfer functions rely on:
- the soundness of the lift functions
- the soundness of both domain transfer functions
Analysis of an assignment in the combined domain

\[ N^\# = \alpha_1 \geq 0 \land \alpha_3 \neq 0x0 \]

\[ y \rightarrow d = x + 1 \]

Abstract post-condition?
Analysis of an assignment in the combined domain

Stage 1: environment resolution

- replaces $x$ with $\star e^\#(x)$
Analysis of an assignment in the combined domain

\[ \begin{align*}
&\text{cofibered layer} \\
&\text{shape} + \text{num} \\
&\text{environment layer} \\
&\text{shape} + \text{num} + \text{env}
\end{align*} \]

\[ \begin{align*}
&\alpha_0 \rightarrow \alpha_1 \\
&\alpha_2 \rightarrow \alpha_3 \\
N^\# &= \alpha_1 \geq 0 \land \alpha_3 \neq 0 \times 0 \\
(\ast\alpha_2) \cdot d &= (\ast\alpha_0) + 1
\end{align*} \]

Abstract post-condition?

Stage 2: propagate into the shape + numerics domain

- only symbolic nodes appear
Analysis of an assignment in the combined domain

Stage 3: resolve cells in the shape graph abstract domain

- \( \star \alpha_0 \) evaluates to \( \alpha_1 \); \( \star \alpha_2 \) evaluates to \( \alpha_3 \)
- \((\star \alpha_2) \cdot d\) fails to evaluate: no points-to out of \( \alpha_3 \)

Abstract post-condition?

\[ N^\# = \alpha_1 \geq 0 \land \alpha_3 \neq 0\times0 \]

\[ (\star \alpha_2) \cdot d = (\star \alpha_0) + 1 \]
Analysis of an assignment in the combined domain

Stage 4 (a): unfolding triggered

- the analysis needs to locally materialize $\alpha_3 \cdot \text{lpos}$...
- thus, unfolding starts at symbolic variable $\alpha_3$
Combining shape and value abstractions

Combined analysis algorithms

Analysis of an assignment in the combined domain

Stage 4 (b): unfolding, shape part

- unfolding of the memory predicate part
- numerical predicates still need be taken into account
Analysis of an assignment in the combined domain

Stage 4 (c): unfolding, numeric part

- Numerical predicates taken into account
- L-value $\alpha_3 \cdot d$ now evaluates into edge $\alpha_3 \cdot d \mapsto \alpha_4$

Abstract post-condition?

$N^# = \alpha_1 \geq 0 \land \alpha_3 \neq 0 \land \alpha_4 \geq 0$

$(\ast \alpha_2) \cdot d = (\ast \alpha_0) + 1$
Analysis of an assignment in the combined domain

### Combined analysis algorithms

Combining shape and value abstractions

**Environment layer**

- Shape + num + env

**Cofibered layer**

- Shape + num

**Shape domain**

**Numeric domain**

---

**Stage 5: create a new node**

- New node $\alpha_6$ denotes a new value
- Will store the new value

Mathematical notation:

$$N^\# = \alpha_1 \geq 0 \land \alpha_3 \neq 0 \land \alpha_4 \geq 0$$

Diagram:

```
& x \alpha_0 \rightarrow \alpha_1 \rightarrow \alpha_4
& y \alpha_2 \rightarrow \alpha_3 \rightarrow \alpha_5
```

```
N^\# = \alpha_1 \geq 0 \land \alpha_3 \neq 0 \land \alpha_4 \geq 0
```
Analysis of an assignment in the combined domain

Stage 6: perform numeric assignment

- numeric assignment completely ignores pointer structures to the new node
Analysis of an assignment in the combined domain

Stage 7: perform the update in the graph

- classic **strong update** in a pointer aware domain
- symbolic node $\alpha_4$ becomes redundant and can be removed
Widening / join in the combined domain

\[ N_{\text{lft}} = \alpha_2 \geq \alpha_5 \geq 2 \]

\[ N_{\text{rght}} = \beta_3 \geq 1 \]
Widening / join in the combined domain

Stage 1: abstract environment
- compute new abstract environment and initial node relation
  e.g., $\alpha_0, \beta_0$ both denote $\&x$

\[ N_{\text{lift}}^\# = \alpha_2 \geq \alpha_5 \geq 2 \]
\[ N_{\text{rgh}}^\# = \beta_3 \geq 1 \]

\[ \Psi(\alpha_0, \beta_0) = \delta_0 \]
\[ \Psi(\alpha_4, \beta_2) = \delta_1 \]
Widening / join in the combined domain

Stage 2: join in the “cofibered” layer

Operations to perform:
1. Compute the join in the graph
2. Convert value abstractions, and join the resulting lattice
Combining shape and value abstractions

Widening / join in the combined domain

- environment layer: shape + num + env
- cofibered layer: shape + num
- shape domain
- numeric domain

\[
N'_{\text{ltf}} = \alpha_2 \geq \alpha_5 \geq 2
\]

\[
N'_{\text{rgt}} = \beta_3 \geq 1
\]

Stage 2: graph join
- apply local join rules
  - ex: points-to matching, weakening to inductive...
- incremental algorithm
Combining shape and value abstractions

Combined analysis algorithms

Widening / join in the combined domain

Stage 2: graph join

- apply local join rules
  - ex: points-to matching, weakening to inductive...
- incremental algorithm
Combining shape and value abstractions

Combined analysis algorithms

Widening / join in the combined domain

Stage 2: graph join

- apply local join rules
  - ex: points-to matching, weakening to inductive...
- incremental algorithm

Stage 2: graph join

- apply local join rules
  - ex: points-to matching, weakening to inductive...
- incremental algorithm

\[ N_{\text{lift}}^\# = \alpha_2 \geq \alpha_5 \geq 2 \]

\[ N_{\text{rgh}}^\# = \beta_3 \geq 1 \]
Widening / join in the combined domain

Stage 3: conversion function application in numerics
- remove nodes that were abstracted away
- rename other nodes
Widening / join in the combined domain

Stage 4: join in the numeric domain

- apply \( \boxplus \) for regular join, \( \triangledown \) for a widening
Outline

1 Introduction
2 Separation Logic
3 A shape abstract domain relying on separation
4 Standard static analysis algorithms
5 Combining shape and value abstractions
6 Conclusion
Updates and summarization

Weak updates cause significant precision loss...
Separation logic makes updates strong

Separation logic

Separating conjunction combines properties on disjoint stores

- Fundamental idea: *forces to identify what is modified*
- Before an **update** (or a **read**) takes place, memory cells need to be **materialized**
- **Local reasoning**: properties on unmodified cells pertain

Summaries

**Inductive predicates** describe unbounded memory regions

- Last lecture: **array segments** and **transitive closure** (TVLA)
Bibliography

- **[JR]**: *Separation Logic: A Logic for Shared Mutable Data Structures.*
  John C. Reynolds.
  In LICS’02, pages 55–74, 2002.

- **[DHY]**: *A Local Shape Analysis Based on Separation Logic.*
  Dino Distefano, Peter W. O’Hearn and Hongseok Yang.
  In TACAS’06, pages 287–302.

- **[CR]**: *Relational inductive shape analysis.*
  Bor-Yuh Evan Chang and Xavier Rival.
Assignment and paper reading

The Frame rule:

- formalize the Hoare logic rules for a language with pointer assignments and condition tests
- prove the Frame rule by induction over the syntax of programs

Reading:


Formalizes the Frame rule, among others
Assignment: a simple analysis in Separation Logic (after TVLA)

1, k assumed to be disjoint lists

\textbf{while}(1 \neq 0)\{

\hspace{1cm} t = l \rightarrow n;

\hspace{1cm} l \rightarrow n = k;

\hspace{1cm} k = l;

\hspace{1cm} l = t;

\}

\