

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 analysis

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 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 *
- **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

$e ::= n$	$(n \in \mathbb{N})$	constants	
	l	l-value	
	$e_0 + e_1$	binary operations	
	\dots		
$P ::= e_0 = e_1$	$ P' \vee P''$	$ P' \wedge P'' \dots$	pure predicates

Pure formulas semantics: $\gamma(P) \subseteq \mathbb{E} \times \mathbb{H}$

Separation logic: points-to predicates

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

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

Points-to predicate

- **Predicate:**

$$F ::= \dots \mid a \mapsto v \quad \text{where } a \text{ is an address and } v \text{ is a value}$$

- **Concretization:**

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

- **Example:**

$$F = \&x \mapsto 18 \quad \&x = 308 \quad \boxed{18}$$

- We also note $\perp \mapsto e$, as an l-value \perp denotes an address

Separation logic: separating conjunction

Merge of concrete heaps: let $h_0, h_1 \in (\mathbb{V}_{\text{addr}} \rightarrow \mathbb{V})$, such that $\text{dom}(h_0) \cap \text{dom}(h_1) = \emptyset$; then, we let $h_0 \circledast h_1$ be defined by:

$$\begin{aligned} h_0 \circledast h_1 : \quad & \text{dom}(h_0) \cup \text{dom}(h_1) & \longrightarrow & \mathbb{V} \\ & x \in \text{dom}(h_0) & \longmapsto & h_0(x) \\ & x \in \text{dom}(h_1) & \longmapsto & h_1(x) \end{aligned}$$

Separating conjunction

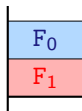
- Predicate:**

$$F ::= \dots \mid F_0 * F_1$$

- Concretization:**

$$\gamma(F_0 * F_1) = \{(e, h_0 \circledast h_1) \mid (e, h_0) \in \gamma(F_0) \wedge (e, h_1) \in \gamma(F_1)\}$$

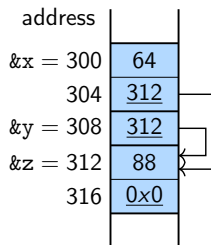
$F_0 * F_1$



An example

Concrete memory layout

(pointer values underlined)



$$\begin{array}{lcl}
 e : & x & \mapsto 300 \\
 & y & \mapsto 308 \\
 & z & \mapsto 312
 \end{array}$$

$$\begin{array}{lcl}
 h : & 300 & \mapsto 64 \\
 & 304 & \mapsto 312 \\
 & 308 & \mapsto 312 \\
 & 312 & \mapsto 88 \\
 & 316 & \mapsto 0
 \end{array}$$

A formula that abstracts away the addresses:

$$\&x \mapsto \langle 64, \&z \rangle * \&y \mapsto \&z * \&z \mapsto \langle 88, 0 \rangle$$

Separation logic: non separating conjunction

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

Non separating conjunction

- **Predicate:**

$$F ::= \dots \mid F_0 \wedge F_1$$

- **Concretization:**

$$\gamma(F_0 \wedge F_1) = \gamma(F_0) \cap \gamma(F_1)$$

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

- $\&a \mapsto \&b \wedge \&b \mapsto \&a$
- $\&a \mapsto \&b * \&b \mapsto \&a$

Separating conjunction vs non separating conjunction

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

$$\&a \mapsto \&b \wedge \&b \mapsto \&a$$

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

$$\&a \mapsto \&b * \&b \mapsto \&a$$

- 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:**

$$\frac{(e, h_0) \in \gamma(F_0) \quad (e, h_1) \in \gamma(F_1)}{(e, h_0 \otimes h_1) \in \gamma(F_0 * F_1)}$$

- **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:**

$$F ::= \dots \mid \text{emp}$$

- **Concretization:**

$$\gamma(\text{emp}) = \{(e, []) \mid e \in \mathbb{E}\} = \mathbb{E} \times \{[]\}$$

where $[]$ denotes the empty store

- emp is the **neutral element for** $*$
(monoid structure induced by $*$)
- by contrast the **neutral element for** \wedge is **TRUE**, with concretization:

$$\gamma(\text{TRUE}) = \mathbb{E} \times \mathbb{H}$$

Separation logic: other connectors

Disjunction:

- $F ::= \dots \mid F_0 \vee F_1$
- concretization:

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

Spatial implication (*aka, magic wand*):

- $F ::= \dots \mid F_0 \multimap F_1$
- concretization:

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

- 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

$$\begin{aligned}
 \gamma(\text{emp}) &= \mathbb{E} \times \{\emptyset\} \\
 \gamma(\text{TRUE}) &= \mathbb{E} \times \mathbb{H} \\
 \gamma(1 \mapsto v) &= \{(e, [\![1]\!](e, h) \mapsto v) \mid e \in \mathbb{E}\} \\
 \gamma(F_0 * F_1) &= \{(e, h_0 \circledast h_1) \mid (e, h_0) \in \gamma(F_0) \wedge (e, h_1) \in \gamma(F_1)\} \\
 \gamma(F_0 \wedge F_1) &= \gamma(F_0) \cap \gamma(F_1) \\
 \gamma(F_0 \vee F_1) &= \gamma(F_0) \cup \gamma(F_1) \\
 \gamma(F_0 \multimap F_1) &= \{(e, h) \mid \forall h_0 \in \mathbb{H}, (e, h_0) \in \gamma(F_0) \implies (e, h \circledast h_0) \in \gamma(F_1)\}
 \end{aligned}$$

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:

$$\forall s, s' \in \mathbb{S}, s \in \gamma(F) \wedge s' \in \llbracket p \rrbracket(s) \implies s' \in \gamma(F')$$

- Application: **formalize proofs of programs**

A few rules (straightforward proofs):

$$\frac{\overline{\{\&x \mapsto ?\}x := e\{\&x \mapsto e\}} \text{ mutation}}{\frac{F_0 \implies F'_0 \quad \{F'_0\}b\{F'_1\} \quad F'_1 \implies F_1}{\{F_0\}b\{F_1\}} \text{ consequence}}$$

$$\frac{x \text{ does not appear in } F}{\{\&x \mapsto ? * F\}x := e\{\&x \mapsto e * F\}} \text{ mutation-2}$$

(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

$$\frac{\{F_0\}b\{F_1\} \quad \text{freevar}(F) \cap \text{write}(b) = \emptyset}{\{F_0 * F\}b\{F_1 * F\}} \text{ frame}$$

- 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**:

```

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)

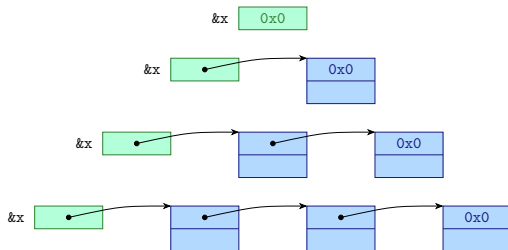
Summarization and inductive definitions

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:



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

Inductive definitions in separation logic

List definition

$$\alpha \cdot \text{list} := \alpha = 0 \wedge \text{emp} \vee \alpha \neq 0 \wedge \alpha \cdot \text{next} \mapsto \delta * \alpha \cdot \text{data} \mapsto \beta * \delta \cdot \text{list}$$

- Formula abstracting our set of structures:

$$\&x \mapsto \alpha * \alpha \cdot \text{list}$$

- **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 $\xrightarrow{\mathcal{U}}$
- A formula F with inductive predicates describes all stores described by all formulas F' such that $F \xrightarrow{\mathcal{U}} F'$

Example:

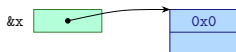
- Let us start with $x \mapsto \alpha_0 * \alpha_0 \cdot \text{list}$; we can unfold it as follows:

$$\&x \mapsto \alpha_0 * \alpha_0 \cdot \text{list}$$

$$\xrightarrow{\mathcal{U}} \quad \&x \mapsto \alpha_0 * \alpha_0 \cdot \text{next} \mapsto \alpha_1 * \alpha_0 \cdot \text{data} \mapsto \beta_1 * \alpha_1 \cdot \text{list}$$

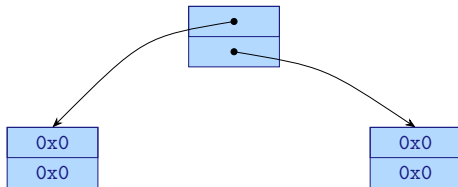
$$\xrightarrow{\mathcal{U}} \quad \&x \mapsto \alpha_0 * \alpha_0 \cdot \text{next} \mapsto \alpha_1 * \alpha_0 \cdot \text{data} \mapsto \beta_1 * \text{emp} \wedge \alpha_1 = \mathbf{0x0}$$

- We get the concrete state below:



Example: tree

- **Example:**



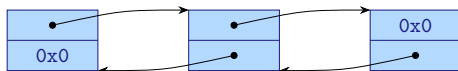
Inductive definition

- **Two recursive calls** instead of one:

$$\begin{aligned}
 \alpha \cdot \text{tree} & := & \alpha = 0 \wedge \text{emp} \\
 & \vee & \alpha \neq 0 \wedge \alpha \cdot \text{left} \mapsto \beta * \alpha \cdot \text{right} \mapsto \delta \\
 & & * \beta \cdot \text{tree} * \delta \cdot \text{tree}
 \end{aligned}$$

Example: doubly linked list

- **Example:**



Inductive definition

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

$$\begin{aligned}
 \alpha \cdot \text{dll}(\delta) & := && \alpha = 0 \wedge \text{emp} \\
 & \vee && \alpha \neq 0 \wedge \alpha \cdot \text{next} \mapsto \beta * \alpha \cdot \text{prev} \mapsto \delta \\
 & && * \beta \cdot \text{dll}(\alpha)
 \end{aligned}$$

Example: sortedness

- **Example:** sorted list



Inductive definition

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

$$\alpha \cdot \text{lsort}_{\text{aux}}(n) := \begin{array}{l} \alpha = 0 \wedge \text{emp} \\ \vee \alpha \neq 0 \wedge n \leq \beta \wedge \alpha \cdot \text{next} \mapsto \delta \\ * \alpha \cdot \text{data} \mapsto \beta * \delta \cdot \text{lsort}_{\text{aux}}(\beta) \end{array}$$

$$\alpha \cdot \text{lsort}() := \alpha \cdot \text{lsort}_{\text{aux}}(-\infty)$$

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

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 \wedge and $*$
- 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**

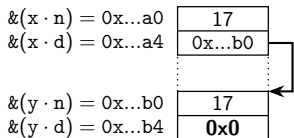
Analysis algorithms:

- inference of “optimal” invariants in SL, with numerical predicates obviously **not computable**

Basic abstraction: structures and their contents (1/2)

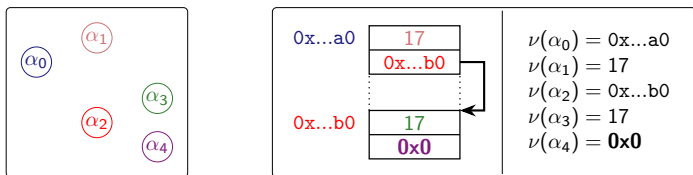
- Concrete memory states

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



Basic abstraction: structures and their contents (1/2)

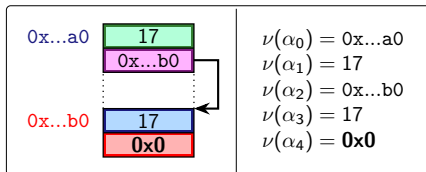
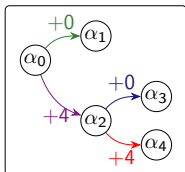
- Concrete memory states
- Abstraction of values into symbolic variables (nodes)



- ▶ characterized by valuation ν
- ▶ ν maps symbolic variables into concrete addresses

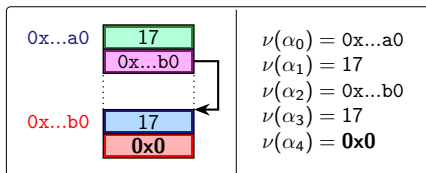
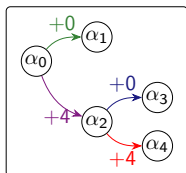
Basic abstraction: structures and their contents (1/2)

- Concrete memory states
- Abstraction of values into symbolic variables / nodes
- Abstraction of regions into points-to edges



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

$$\gamma_{\text{sh}}(G) = \{(h, \nu) \mid \dots\}$$

valuation ν plays an important role to combine abstraction...

Structure of shape graphs

Valuations bridge the gap between nodes and values

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

Symbolic variables

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

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 ν): $\text{Val} = \mathbb{V}^\sharp \longrightarrow \mathbb{V}$

Note that valuations treat **in the same way addresses** and **raw values**

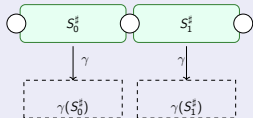
Structure of shape graphs

Distinct edges describe separate regions

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

Separating conjunction

$$\gamma_{\text{sh}}(S_0^\# * S_1^\#) = \{ (h_0 \otimes h_1, \nu) \mid (h_0, \nu) \in \gamma_{\text{sh}}(S_0^\#) \wedge (h_1, \nu) \in \gamma_{\text{sh}}(S_1^\#) \}$$



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

$$\gamma_{\text{sh}}(\text{emp}) = \{ (\emptyset, \nu) \mid \nu : \mathbb{V}^\# \rightarrow \mathbb{V} \}$$


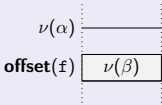
Abstraction of contiguous regions

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

Graph edge	Separation logic formula	Concrete view
	$\alpha \cdot \mathbf{f} \mapsto \beta$	

- Concretization:

$$\gamma_{\text{sh}}(\alpha \cdot \mathbf{f} \mapsto \beta) = \{([\nu(\alpha) + \mathbf{offset}(\mathbf{f}) \mapsto \nu(\beta)], \nu) \mid \nu : \{\alpha, \beta, \dots\} \rightarrow \mathbb{N}\}$$

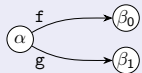
Abstraction of contiguous regions

Contiguous regions are described by adjacent points-to edges

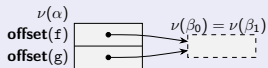
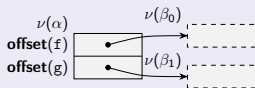
To describe **blocks** containing series of **cells** (e.g., in a **C structure**), shape graphs utilize several outgoing edges from the node representing the base address of the block

Field splitting model

- Separation impacts edges / fields, *not pointers*



- Shape graph accounts for both abstract states below:

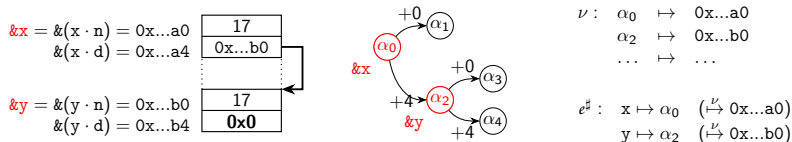


In other words, in a field splitting model, separation:

- asserts addresses are distinct
- says nothing about contents

Abstraction of the environment

Environments bind variables to their (concrete / abstract) address



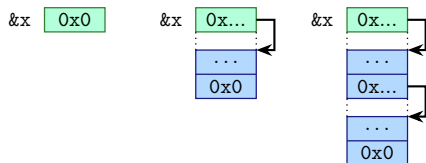
Abstract environments

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

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

Basic abstraction: summarization

Set of all lists of any length:



Well-founded list inductive def.

$$\alpha \cdot \text{list} :=$$

$$(\text{emp} \wedge \alpha = \mathbf{0x0})$$

$$\vee (\alpha \cdot d \mapsto \beta_0 * \alpha \cdot n \mapsto \beta_1$$

$$* \beta_1 \cdot \text{list} \wedge \alpha \neq \mathbf{0x0})$$

well-founded predicate

Inductive summary predicates

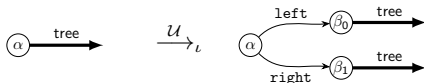
Concretization based on **unfolding** and **least-fixpoint**:

- $\xrightarrow{\mathcal{U}}$ replaces **an $\alpha \cdot \text{list}$ predicate** with **one of its premises**
- $\gamma(S^\#, F) = \bigcup \{ \gamma(S_u^\#, F_u) \mid (S^\#, F) \xrightarrow{\mathcal{U}} (S_u^\#, F_u) \}$

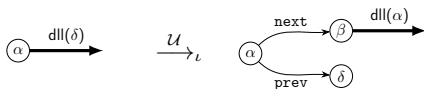
Inductive structures: a few instances

As before, **many interesting inductive predicates** encode nicely into graph inductive definitions:

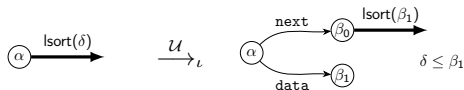
- **More complex shapes: trees**



- **Relations among pointers: doubly-linked lists**

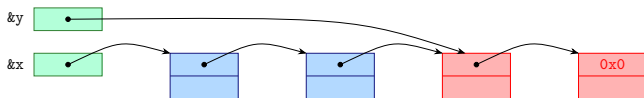


- **Relations between pointers and numerical: sorted lists**



Inductive segments

A frequent pattern:



A first attempt:

- x points to a list, so $\&x \mapsto \alpha * \alpha \cdot \text{list}$ holds
- y points to a list, so $\&y \mapsto \beta * \beta \cdot \text{list}$ holds

However, the following **does not hold**

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

Why ? **violation of separation!**

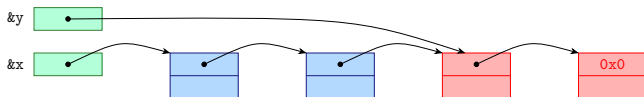
A second attempt:

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

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:

$$\begin{aligned} \alpha \cdot \text{list_endp}(\pi) &::= (\text{emp}, \alpha = \pi) \\ &| (\alpha \cdot \text{next} \mapsto \beta_0 * \alpha \cdot \text{data} \mapsto \beta_1 \\ &\quad * \beta_0 \cdot \text{list_endp}(\pi), \alpha \neq 0) \end{aligned}$$

This definition **straightforwardly derives** from list

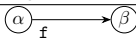
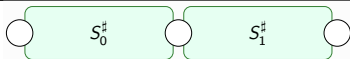
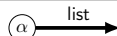
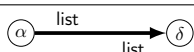
Thus, we make **segments** part of the **fundamental predicates of the domain**



Multi-segments: possible, but harder for analysis

Shape graphs and separation logic

Semantic preserving translation Π of graphs into separation logic formulas:

Graph $S^\# \in \mathbb{D}_{sh}^\#$	Translated formula $\Pi(S^\#)$
	$\alpha \cdot \mathbf{f} \mapsto \beta$
	$\Pi(S_0^\#) * \Pi(S_1^\#)$
	$\alpha \cdot \mathbf{list}$
	$\alpha \cdot \mathbf{list_endp}(\delta)$
other inductives and segments	similar

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

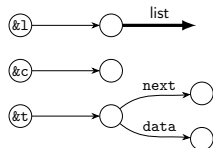
Static analysis overview

A list insertion function:

```

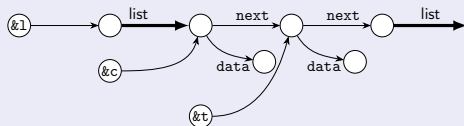
list * l assumed to point to a list
list * t assumed to point to a list element
list * c = l;
while(c != NULL && c -> next != NULL && (...)){
    c = c -> next;
}
t -> next = c -> next;
c -> next = t;
  
```

- 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** $\llbracket b \rrbracket : \mathbb{S} \rightarrow \mathcal{P}(\mathbb{S})$
- Abstract post-condition** $\llbracket b \rrbracket^\#(S)$, computed by the analysis:

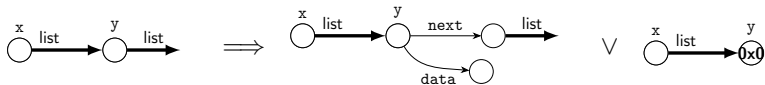
$$s \in \gamma(S) \implies \llbracket b \rrbracket(s) \subseteq \gamma(\llbracket b \rrbracket^\#(S))$$

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

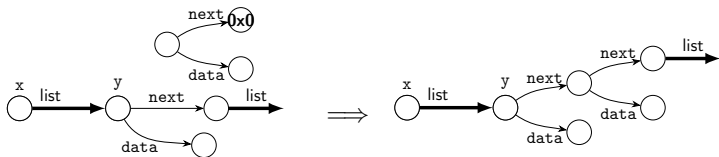
$$\begin{aligned}
 \llbracket b_0; b_1 \rrbracket^\#(S) &= \llbracket b_1 \rrbracket^\# \circ \llbracket b_0 \rrbracket^\#(S) \\
 \llbracket l = e \rrbracket^\#(S) &= \mathit{assign}(l, e, S) \\
 \llbracket l = \mathbf{malloc}(n) \rrbracket^\#(S) &= \mathit{alloc}(l, n, S) \\
 \llbracket \mathbf{free}(l) \rrbracket^\#(S) &= \mathit{free}(l, n, S) \\
 \llbracket \mathbf{if}(e) b_t \mathbf{else} b_f \rrbracket^\#(S) &= \begin{cases} \mathit{join}(\llbracket b_t \rrbracket^\#(\mathit{test}(e, S)), \\ \llbracket b_f \rrbracket^\#(\mathit{test}(e = \mathbf{false}, S))) \end{cases} \\
 \llbracket \mathbf{while}(e)b \rrbracket^\#(S) &= \mathit{test}(e = \mathbf{false}, \mathit{lfp}_S^\# F^\#) \\
 &\text{where, } F^\# : S_0 \mapsto \llbracket b \rrbracket^\#(\mathit{test}(e, S_0))
 \end{aligned}$$

The algorithms underlying the transfer functions

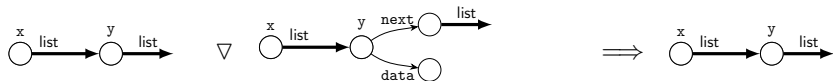
- **Unfolding:** cases analysis on summaries



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



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



Outline

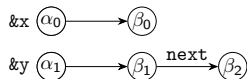
- 1 Introduction
- 2 Separation Logic
- 3 A shape abstract domain relying on separation
- 4 Standard static analysis algorithms**
 - Overview of the analysis
 - Post-conditions and unfolding
 - Folding: widening and inclusion checking
 - Abstract interpretation framework: assumptions and results
 - Comparing Separation Logic and Three-Valued logic abstractions
- 5 Combining shape and value abstractions
- 6 Conclusion

Analysis of an assignment in the graph domain

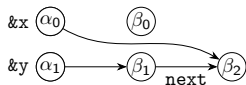
Steps for analyzing $x = y \rightarrow \text{next}$ (local reasoning)

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

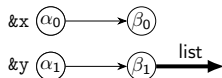
With pre-condition:



- Step 1 produces $\alpha_0 \mapsto \beta_0$
- Step 2 produces β_2
- End result:



With pre-condition:



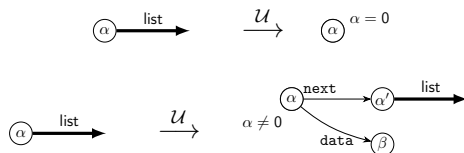
- Step 1 produces $\alpha_0 \mapsto \beta_0$
- **Step 2 fails**

- 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:



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

Soundness: by definition of the concretization of inductive structures

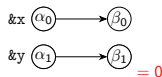
$$\gamma_{\text{sh}}(S^\#) \subseteq \bigcup \{ \gamma_{\text{sh}}(S_0^\#) \mid S^\# \xrightarrow{\mathcal{U}} S_0^\# \}$$

Analysis of an assignment, with unfolding

Principle

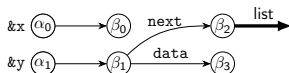
- We have $\gamma_{\text{sh}}(\alpha \cdot \iota) = \bigcup \{ \gamma_{\text{sh}}(S^\#) \mid \alpha \cdot \iota \xrightarrow{\mathcal{U}} S^\# \}$
- Replace $\alpha \cdot \iota$ with a finite number of disjuncts and continue

Disjunct 1:

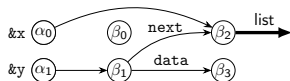


- Step 1 produces $\alpha_0 \mapsto \beta_0$
- **Step 2 fails: Null pointer !**
- In a **correct** program, would be ruled out by a **condition** $y \neq 0$ i.e., $\beta_1 \neq 0$ in $\mathbb{D}_{\text{num}}^\#$

Disjunct 2:



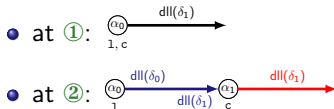
- Step 1 produces $\alpha_0 \mapsto \beta_0$
- Step 2 produces β_2
- **End result:**



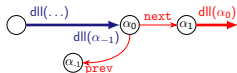
Unfolding and degenerated cases

```

assume(l points to a dll)
c = 1;
① while(c ≠ NULL && condition)
  c = c -> next;
② if(c ≠ 0 && c -> prev ≠ 0)
  c = c -> prev -> prev;
  
```

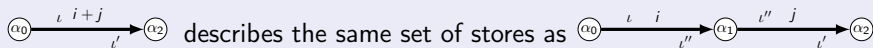


⇒ **non trivial unfolding**



- **Materialization of $c \rightarrow \text{prev}$:**

Segment splitting lemma: basis for segment unfolding



- **Materialization of $c \rightarrow \text{prev} \rightarrow \text{prev}$:**
- **Implementation issue:** discover **which inductive edge** to unfold **very hard !**

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

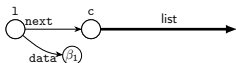
Back to the **list traversal** example:

First iterates in the loop:

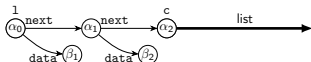
- at **iteration 0** (before entering the loop):



- at **iteration 1**:



- at **iteration 2**:

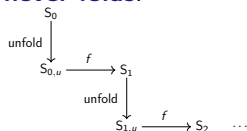


- How to guarantee **termination** of the analysis ?
- How to **introduce segment edges** / perform **abstraction** ?

```

assume(l points to a list)
c = l;
while(c ≠ NULL){
  c = c → next;
}
  
```

The analysis **unfolds**, but **never folds**:



Widening

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

Definition of a widening operator ∇

- **Over-approximates join:**

$$\begin{cases} \gamma(X^\#) \subseteq \gamma(X^\# \nabla Y^\#) \\ \gamma(Y^\#) \subseteq \gamma(X^\# \nabla Y^\#) \end{cases}$$

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

$$\begin{cases} Y_0^\# = X_0^\# \\ \forall n \in \mathbb{N}, Y_{n+1}^\# = Y_n^\# \nabla X_{n+1}^\# \end{cases}$$

Canonicalization

Upper closure operator

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

Canonicalization

- **Disjunctive completion:** $\mathbb{D}_\vee^\# =$ finite disjunctions over $\mathbb{D}^\#$
- **Canonicalization operator** ρ_\vee defined by $\rho_\vee : \mathbb{D}_\vee^\# \longrightarrow \mathbb{D}_{\text{can}\vee}^\#$ and $\rho_\vee(X^\#) = \{\rho(x^\#) \mid x^\# \in X^\#\}$ where ρ 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^\sharp **can be weakened into** S_1^\sharp if and only if

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

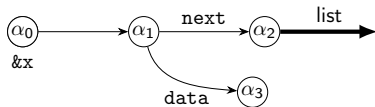
We then note $S_0^\sharp \preceq S_1^\sharp$

Applications:

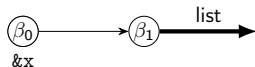
- **inclusion test** (comparison) inputs S_0^\sharp, S_1^\sharp ; if returns true $S_0^\sharp \preceq S_1^\sharp$
- **canonicalization** (unary weakening) inputs S_0^\sharp and returns $\rho(S_0^\sharp)$ such that $S_0^\sharp \preceq \rho(S_0^\sharp)$
- **widening / join** (binary weakening ensuring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

Weakening: example

We consider S_0^\sharp defined by:



and S_1^\sharp defined by:



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

$$\Psi : \begin{array}{l} \beta_0 \longmapsto \alpha_0 \\ \beta_1 \longmapsto \alpha_1 \end{array}$$

- 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:

Separating conjunction rule (\preceq_*)

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 $S_0^\# * S_1^\#$ into $S_{0,\text{weak}}^\# * S_{1,\text{weak}}^\#$

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 $S^\#$

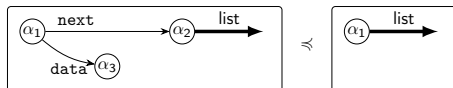
- the proof is trivial:

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

- 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

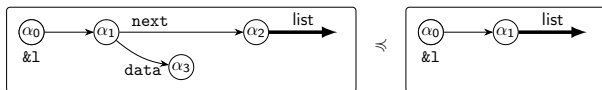
By **rule** (\preceq_u):



Additionally, by **rule** (\preceq_{id}):



Thus, by **rule** (\preceq_*):



Inclusion checking rules in the shape domain

Graphs to compare have distinct sets of nodes, thus inclusion check should carry out a **valuation transformer** $\Psi : \mathbb{V}^\sharp(S_1^\sharp) \longrightarrow \mathbb{V}^\sharp(S_0^\sharp)$ (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:**

$$\begin{aligned} \forall i, \Psi(\beta_i) = \alpha_i &\implies \alpha_0 \cdot f \mapsto \alpha_1 \sqsubseteq_{\Psi}^{\sharp} \beta_0 \cdot f \mapsto \beta_1 \\ \Psi(\beta) = \alpha &\implies \alpha \cdot \text{list} \sqsubseteq_{\Psi}^{\sharp} \beta \cdot \text{list} \\ \forall i, \Psi(\beta_i) = \alpha_i &\implies \alpha_0 \cdot \text{list_endp}(\alpha_1) \sqsubseteq_{\Psi}^{\sharp} \beta_0 \cdot \text{list_endp}(\beta_1) \end{aligned}$$

- **Rules on inductives:**

$$\begin{aligned} \forall i, \Psi(\beta_i) = \alpha_i &\implies \text{emp} \sqsubseteq_{\Psi}^{\sharp} \beta_0 \cdot \text{list_endp}(\beta_1) \\ S_0^\sharp \sqsubseteq_{\Psi}^{\sharp} S_1^\sharp \wedge \beta \cdot \iota \xrightarrow{\mathcal{U}} S_1^\sharp &\implies S_0^\sharp \sqsubseteq_{\Psi}^{\sharp} \beta \cdot \iota \\ \text{if } \beta_1 \text{ fresh, } \Psi' = \Psi[\beta_1 \mapsto \alpha_1] \text{ and } \Psi(\beta_0) = \alpha_0 \text{ then,} & \\ S_0^\sharp \sqsubseteq_{\Psi'}^{\sharp} \beta_1 \cdot \text{list} &\implies \alpha_0 \cdot \text{list_endp}(\alpha_1) * S_0^\sharp \sqsubseteq_{\Psi}^{\sharp} \beta_0 \cdot \iota \end{aligned}$$

Inclusion checking algorithm

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

- 1 start with Ψ defined by $\Psi(\beta) = \alpha$ if and only if there exists a variable x such that $e_0^\sharp(x) = \alpha \wedge e_1^\sharp(x) = \beta$
 - 2 iteratively **apply local rules**, and extend Ψ when needed
 - 3 return true when both shape graphs become empty
- the first step ensures both environments are consistent

This algorithm is sound:

Soundness

$$(e_0^\sharp, S_0^\sharp) \sqsubseteq_\Psi^\sharp (e_1^\sharp, S_1^\sharp) \implies \gamma(e_0^\sharp, S_0^\sharp) \subseteq \gamma(e_1^\sharp, S_1^\sharp)$$

Over-approximation of union

The principle of join and widening algorithm **is similar to that of \sqsubseteq^\sharp** :

- It can be computed **region by region**, as for weakening in general:
If $\forall i \in \{0, 1\}, \forall s \in \{\text{left}, \text{right}\}, S_{i,s}^\sharp \preceq S_s^\sharp$,



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

$$\Psi : \mathbb{V}^\sharp(S_{\text{left}}^\sharp) \times \mathbb{V}^\sharp(S_{\text{right}}^\sharp) \longrightarrow \mathbb{V}^\sharp(S^\sharp)$$

- 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 ∇
though the abstract union would be defined similarly in the shape domain

Several rules derive **from** (\preceq_{id}) :

- If $S_{\text{lft}}^{\#} = \alpha_0 \cdot \mathbf{f} \mapsto \alpha_1$
and $S_{\text{rgh}}^{\#} = \beta_0 \cdot \mathbf{f} \mapsto \beta_1$
and $\Psi(\alpha_0, \beta_0) = \delta_0$, $\Psi(\alpha_1, \beta_1) = \delta_1$, then:

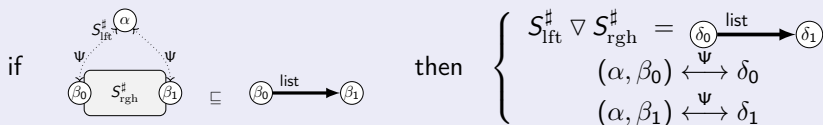
$$S_{\text{lft}}^{\#} \nabla S_{\text{rgh}}^{\#} = \delta_0 \cdot \mathbf{f} \mapsto \delta_1$$

- If $S_{\text{lft}}^{\#} = \alpha_0 \cdot \text{list}$
and $S_{\text{rgh}}^{\#} = \beta_0 \cdot \text{list}_1$
and $\Psi(\alpha_0, \beta_0) = \delta_0$, then:

$$S_{\text{lft}}^{\#} \nabla S_{\text{rgh}}^{\#} = \delta_0 \cdot \text{list}$$

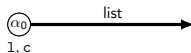
Over-approximation of union: segment introduction rule

Rule

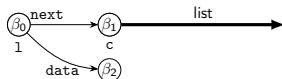


Application to list traversal, at the end of iteration 1:

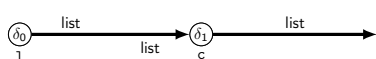
- before iteration 0:



- end of iteration 0:



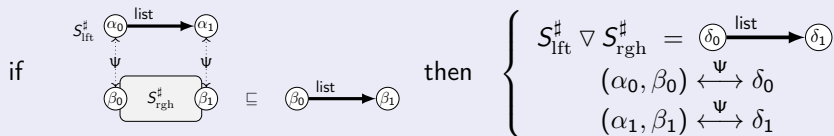
- join, before iteration 1:



$$\left\{ \begin{array}{l} \Psi(\alpha_0, \beta_0) = \delta_0 \\ \Psi(\alpha_0, \beta_1) = \delta_1 \end{array} \right.$$

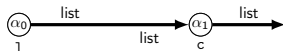
Over-approximation of union: segment extension rule

Rule

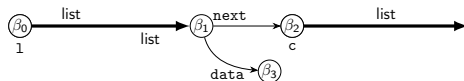


Application to list traversal, at the end of iteration 1:

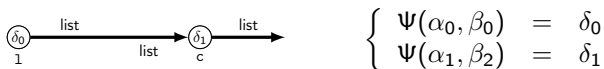
- previous invariant before iteration 1:



- end of iteration 1:



- join, before iteration 1:



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 $(e_0^\#, S_0^\#)$ and $(e_1^\#, S_1^\#)$

- 1 define Ψ, e by $\Psi(\alpha, \beta) = e(x) = \delta$ (where δ is a fresh node) if and only if $e_0^\#(x) = \alpha \wedge e_1^\#(x) = \beta$
- 2 iteratively **apply join local rules**, and extend Ψ 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

$$\gamma(e_0^\#, S_0^\#) \cup \gamma(e_1^\#, S_1^\#) \subseteq \gamma(e^\#, S^\#)$$

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 :

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

$F : \mathcal{P}(\mathbb{M}) \rightarrow \mathcal{P}(\mathbb{M})$ is the concrete semantic (“post”) function of b

thus, the concrete semantics writes down as $\llbracket b \rrbracket = \text{lfp}_\emptyset F$

- **Assumptions in the abstract level:**

$\mathbb{M}^\#$ set of abstract elements, no order a priori

$m^\# ::= (e^\#, S^\#)$

$\gamma_{\text{mem}} : \mathbb{M}^\# \rightarrow \mathcal{P}(\mathbb{M})$ concretization

$F^\# : \mathbb{M}^\# \rightarrow \mathbb{M}^\#$ sound abstract semantic function

i.e., such that $F \circ \gamma_{\text{mem}} \subseteq \gamma_{\text{mem}} \circ F^\#$

$\nabla : \mathbb{M}^\# \times \mathbb{M}^\# \rightarrow \mathbb{M}^\#$ widening operator, terminates, and such that

$\gamma_{\text{mem}}(m_0^\#) \cup \gamma_{\text{mem}}(m_1^\#) \subseteq \gamma_{\text{mem}}(m_0^\# \nabla m_1^\#)$

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:

$$\begin{cases} m_0^\# &= \perp \\ m_{n+1}^\# &= m_n^\# \nabla F^\#(m_n^\#) \end{cases}$$

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

$$\text{fp}_\emptyset F = \bigcup_k F^k(\emptyset) \subseteq \gamma_{\text{mem}}(m_N^\#)$$

Discussion on the abstract ordering

How about the abstract ordering ? We assumed *NONE* so far...

- **Logical ordering**, induced by concretization, used for **proofs**

$$m_0^\# \sqsubseteq m_1^\# \quad ::= \quad " \gamma_{\text{mem}}(m_0^\#) \subseteq \gamma_{\text{mem}}(m_1^\#) "$$

- **Approximation of the logical ordering**, **implemented** as a function $\text{is_le} : \mathbb{M}^\# \times \mathbb{M}^\# \rightarrow \{\text{true}, \top\}$, used to **test the convergence of abstract iterates**

$$\text{is_le}(m_0^\#, m_1^\#) = \text{true} \quad \implies \quad \gamma_{\text{mem}}(m_0^\#) \subseteq \gamma_{\text{mem}}(m_1^\#)$$

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

- Also, **computational ordering** would be used for **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)

$$\begin{array}{l}
 F ::= \text{emp} \\
 \quad | \text{TRUE} \\
 \quad | l \mapsto l \\
 \quad | F_0 * F_1 \\
 \quad | F_0 \wedge F_1 \\
 \quad | F_0 \text{---} * F_1
 \end{array}$$

Concretization:

$$\begin{aligned}
 \gamma(\text{emp}) &= \mathbb{E} \times \{\emptyset\} \\
 \gamma(\text{TRUE}) &= \mathbb{E} \times \mathbb{H} \\
 \gamma(l \mapsto v) &= \{(e, \llbracket l \rrbracket(e, h) \mapsto v) \mid e \in \mathbb{E}\} \\
 \gamma(F_0 * F_1) &= \{(e, h_0 \otimes h_1) \mid (e, h_0) \in \gamma(F_0) \wedge (e, h_1) \in \gamma(F_1)\} \\
 \gamma(F_0 \wedge F_1) &= \gamma(F_0) \cap \gamma(F_1) \\
 \gamma(F_0 \text{---} * F_1) &= \text{exercise}
 \end{aligned}$$

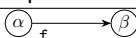
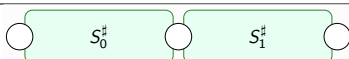
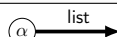
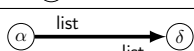
Program reasoning: **frame rule** and **strong updates**

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 Π of graphs into separation logic formulas:

Graph $S^\# \in \mathbb{D}_{sh}^\#$	Translated formula $\Pi(S^\#)$
	$\alpha \cdot f \mapsto \beta$
	$\Pi(S_0^\#) * \Pi(S_1^\#)$
	$\alpha \cdot list$
	$\alpha \cdot list_endp(\delta)$
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:

$$F_0 * F_1 * \dots * F_n$$

- 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:

$$p_0 \wedge p_1 \wedge \dots \wedge p_n$$

- 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

Large / unbounded numbers of concrete cells need to be abstracted

- **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

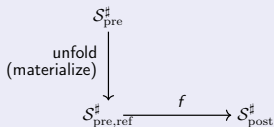
Concretize partially, update, abstract

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

Separation logic:

Local (partial) concretization

For **materialization**:

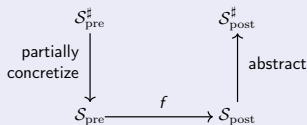


In both cases, two mechanisms are needed:

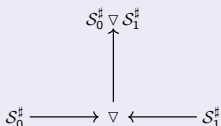
- 1 refine summaries
- 2 synthesize summaries

TVLA:

Focus, analyze, canonicalize



Global abstraction: widening



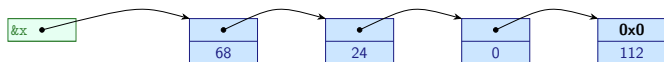
Outline

- 1 Introduction
- 2 Separation Logic
- 3 A shape abstract domain relying on separation
- 4 Standard static analysis algorithms
- 5 Combining shape and value abstractions**
 - Shape and value properties
 - Combined abstraction with cofibered abstract domain
 - Combined analysis algorithms
- 6 Conclusion

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:**



- **sorted list:**



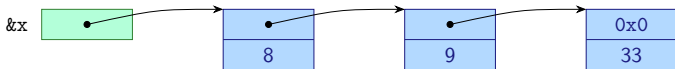
- **list with a length constraint**
- **tries:** binary trees with paths labelled with sequences of “0” and “1”
- **balanced trees:** red-black, AVL...

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



Inductive definition

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

$$\alpha \cdot \text{l-sort}_{\text{aux}}(n) := \begin{array}{l} \alpha = 0 \wedge \text{emp} \\ \vee \alpha \neq 0 \wedge n \leq \beta \wedge \alpha \cdot \text{next} \mapsto \delta \\ * \alpha \cdot \text{data} \mapsto \beta * \delta \cdot \text{l-sort}_{\text{aux}}(\beta) \end{array}$$

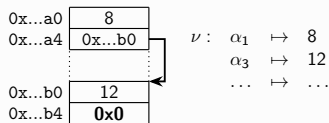
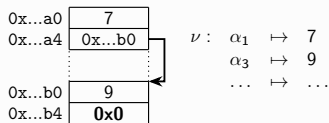
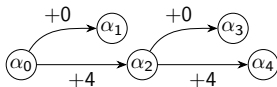
$$\alpha \cdot \text{l-sort}() := \alpha \cdot \text{l-sort}_{\text{aux}}(-\infty)$$

Adding value information (here, numeric)

Concrete numeric values appear in the valuation thus the abstracting contents boils down to abstracting ν !

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

Memory abstraction:



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** $\mathbb{D}_{\text{sh}}^{\#}$ of graphs
 abstract stores together with a **physical mapping** of nodes

$$\gamma_{\text{sh}} : \mathbb{D}_{\text{sh}}^{\#} \rightarrow \mathcal{P}((\mathbb{D}_{\text{sh}}^{\#} \rightarrow \mathbb{M}) \times (\mathbb{V}^{\#} \rightarrow \mathbb{V}))$$
- numerical abstract domain** $\mathbb{D}_{\text{num}}^{\#}$, abstracts physical mapping of nodes

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

Combined domain [CR]

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

$$\gamma(S^{\#}, N^{\#}) = \{(\ell, \nu) \in \mathbb{M} \mid \nu \in \gamma_{\text{num}}(N^{\#}) \wedge (\ell, \nu) \in \gamma_{\text{sh}}(S^{\#})\}$$

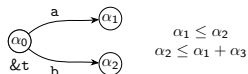
Can it be described as a reduced product ?

- 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:



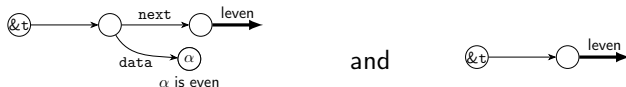
$$\alpha_1 \leq \alpha_2$$

$$\alpha_2 \leq \alpha_1 + \alpha_3$$

- this element exists in the reduced product domain (independent components)
- but, ... **what is α_3 ?**

Unclear comparison:

How can we compare the two elements below ?



and

- 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** $\mathbb{D}_{\text{sh}}^{\#}$
- For **each** graph $S^{\#} \in \mathbb{D}_{\text{sh}}^{\#}$, we have a **numerical lattice** $\mathbb{D}_{\text{num}\langle S^{\#} \rangle}^{\#}$
 - ▶ example: if graph $S^{\#}$ contains nodes $\alpha_0, \alpha_1, \alpha_2$, $\mathbb{D}_{\text{num}\langle S^{\#} \rangle}^{\#}$ should abstract $\{\alpha_0, \alpha_1, \alpha_2\} \rightarrow \mathbb{V}$
- **An abstract value is a pair** $(S^{\#}, N^{\#})$, such that $N^{\#} \in \mathbb{D}_{\text{num}\langle N^{\#} \rangle}^{\#}$

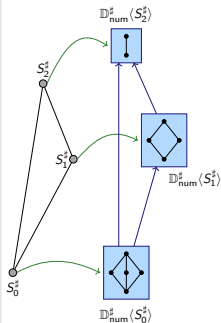
Outline

- 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 $(\mathbb{D}_{\text{sh}}^{\#}, \sqsubseteq^{\#})$, with concretization $\gamma_{\text{sh}} : \mathbb{D}_{\text{sh}}^{\#} \rightarrow \mathbb{D}$
- **Function:** $\phi : \mathbb{D}_{\text{sh}}^{\#} \rightarrow \mathcal{D}$, where each element of \mathcal{D} is an abstract domain instance $(\mathbb{D}_{\text{num}}^{\#}, \sqsubseteq_{\text{num}}^{\#})$, with a concretization $\gamma_{\text{num}} : \mathbb{D}_{\text{num}}^{\#} \rightarrow \mathbb{D}$ (**tied to a shape graph**)
- **Domain $\mathbb{D}^{\#}$:** set of **pairs $(S^{\#}, N^{\#})$ where $N^{\#} \in \phi(S^{\#})$**
- **Concretization:** $\gamma(S^{\#}, N^{\#}) = \gamma(S^{\#}) \cap \gamma(N^{\#})$
- **Lift functions:** $\forall S_0^{\#}, S_1^{\#} \in \mathbb{D}_{\text{sh}}^{\#}$, such that $S_0^{\#} \sqsubseteq^{\#} S_1^{\#}$, there exists a function $\Pi_{S_0^{\#}, S_1^{\#}} : \phi(S_0^{\#}) \rightarrow \phi(S_1^{\#})$, that is monotone for $\gamma_{S_0^{\#}}$ and $\gamma_{S_1^{\#}}$

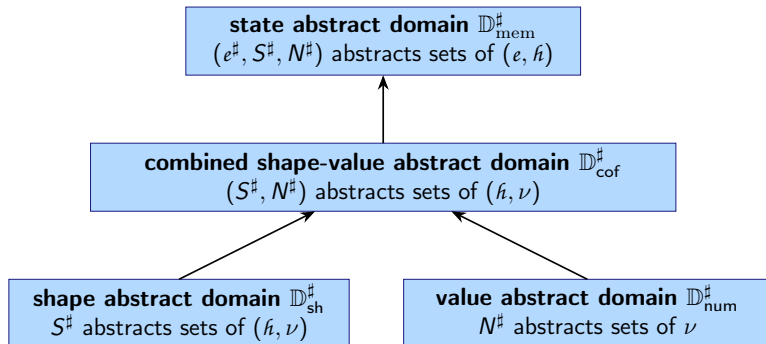


- General construction presented in **[AV]** (Arnaud Venet)
- Intuition: a **dependent domain product**

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 $\mathbb{D}_{sh}^\#$
- 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** $S^\# = S_0^\# \nabla S_1^\#$
- 2 Then **move to** $\phi(S^\#)$, by computing $N_{0c}^\# = \Pi_{S_0^\#, S^\#}(N_0^\#)$ and $N_{1c}^\# = \Pi_{S_1^\#, S^\#}(N_1^\#)$
- 3 Last **widen in** $\phi(S^\#)$: $N^\# = N_{0c}^\# \nabla_{S^\#} N_{1c}^\#$
- 4 Return $(S_0^\#, N_0^\#) \nabla (S_1^\#, N_1^\#) = (S^\#, N^\#)$

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

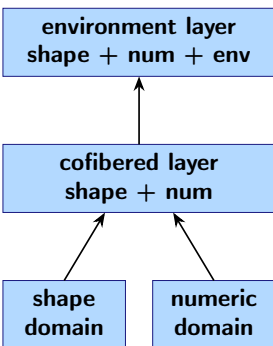
Typical process to compute abstract post-conditions

- 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



$$\&x \ (\alpha_0) \longrightarrow \alpha_1$$

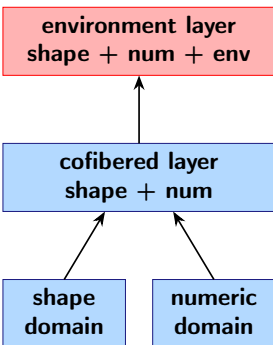
$$\&y \ (\alpha_2) \longrightarrow \alpha_3 \xrightarrow{\text{!pos}}$$

$$N^\# = \alpha_1 \geq 0 \wedge \alpha_3 \neq \mathbf{0x0}$$

$$y \rightarrow d = x + 1$$

Abstract post-condition ?

Analysis of an assignment in the combined domain



$$\&x \ (\alpha_0) \longrightarrow (\alpha_1)$$

$$\&y \ (\alpha_2) \longrightarrow (\alpha_3) \xrightarrow{\text{pos}}$$

$$N^\# = \alpha_1 \geq 0 \wedge \alpha_3 \neq \mathbf{0x0}$$

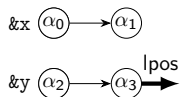
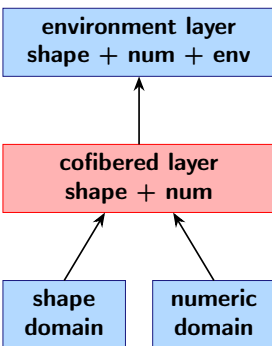
$$y \rightarrow d = x + 1 \quad \Rightarrow \quad (*\alpha_2) \cdot d = (*\alpha_0) + 1$$

Abstract post-condition ?

Stage 1: environment resolution

- replaces x with $*e^\#(x)$

Analysis of an assignment in the combined domain



$$N^\# = \alpha_1 \geq 0 \wedge \alpha_3 \neq \mathbf{0x0}$$

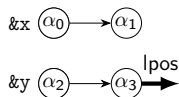
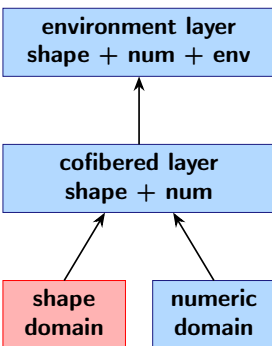
$$(*\alpha_2) \cdot \mathbf{d} = (*\alpha_0) + 1$$

Abstract post-condition ?

Stage 2: propagate into the shape + numerics domain

- only symbolic nodes appear

Analysis of an assignment in the combined domain



$$N^\# = \alpha_1 \geq 0 \wedge \alpha_3 \neq \mathbf{0x0}$$

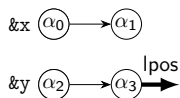
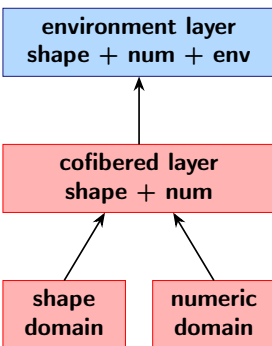
$$(*\alpha_2) \cdot d = (*\alpha_0) + 1$$

Abstract post-condition ?

Stage 3: resolve cells in the shape graph abstract domain

- $*\alpha_0$ evaluates to α_1 ; $*\alpha_2$ evaluates to α_3
- $(*\alpha_2) \cdot d$ **fails to evaluate: no points-to out of α_3**

Analysis of an assignment in the combined domain



$$N^\# = \alpha_1 \geq 0 \wedge \alpha_3 \neq \mathbf{0x0}$$

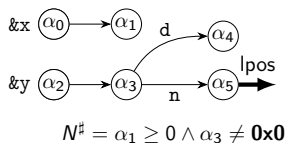
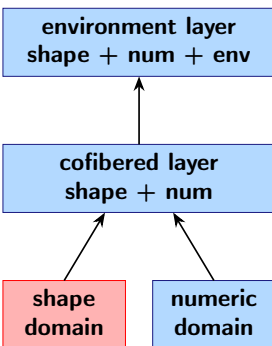
$$(*\alpha_2) \cdot \mathbf{d} = (*\alpha_0) + 1$$

Abstract post-condition ?

Stage 4 (a): unfolding triggered

- the analysis needs to locally materialize $\alpha_3 \cdot \text{lpos}$...
- thus, unfolding starts at symbolic variable α_3

Analysis of an assignment in the combined domain



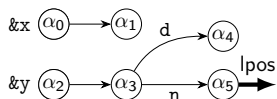
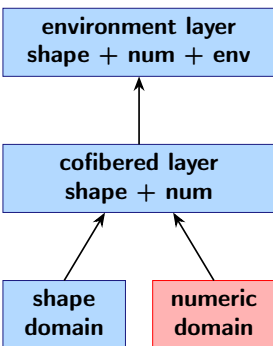
$$(*\alpha_2) \cdot d = (*\alpha_0) + 1$$

Abstract post-condition ?

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



$$N^\# = \alpha_1 \geq 0 \wedge \alpha_3 \neq \mathbf{0x0} \wedge \alpha_4 \geq 0$$

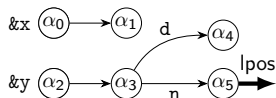
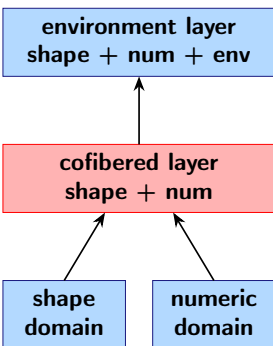
$$(*\alpha_2) \cdot d = (*\alpha_0) + 1$$

Abstract post-condition ?

Stage 4 (c): unfolding, numeric part

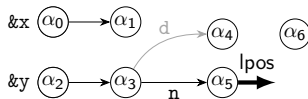
- numerical predicates taken into account
- l-value $\alpha_3 \cdot d$ **now evaluates into edge** $\alpha_3 \cdot d \mapsto \alpha_4$

Analysis of an assignment in the combined domain



$$N^\# = \alpha_1 \geq 0 \wedge \alpha_3 \neq \mathbf{0x0} \wedge \alpha_4 \geq 0$$

create node α_6

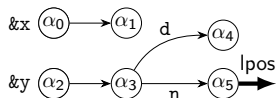
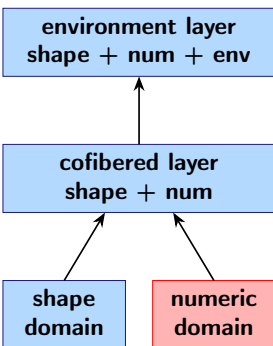


$$N^\# = \alpha_1 \geq 0 \wedge \alpha_3 \neq \mathbf{0x0} \wedge \alpha_4 \geq 0$$

Stage 5: create a new node

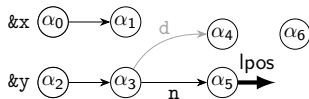
- new node α_6 denotes a new value
will store the new value

Analysis of an assignment in the combined domain



$$N^\# = \alpha_1 \geq 0 \wedge \alpha_3 \neq \mathbf{0x0} \wedge \alpha_4 \geq 0$$

$\alpha_6 \leftarrow \alpha_1 + 1$ in numerics

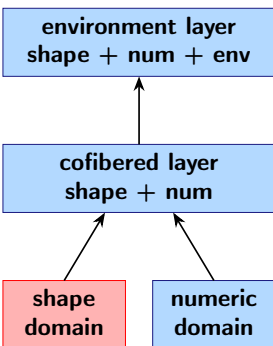


$$N^\# = \alpha_1 \geq 0 \wedge \alpha_3 \neq \mathbf{0x0} \wedge \alpha_4 \geq 0 \wedge \alpha_6 \geq 1$$

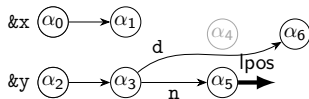
Stage 6: perform numeric assignment

- numeric assignment **completely ignores pointer structures** to the new node

Analysis of an assignment in the combined domain



mutate $(\alpha_3 \cdot d) \mapsto \alpha_4$ into α_6

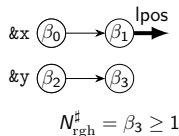
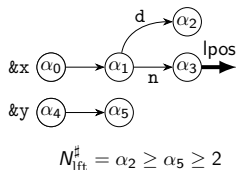
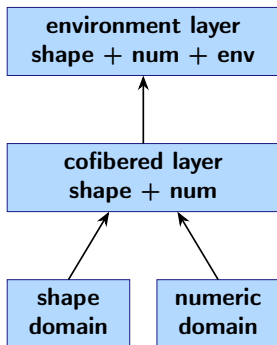


$$N^\# = \alpha_1 \geq 0 \wedge \alpha_3 \neq \mathbf{0x0} \wedge \alpha_4 \geq 0 \wedge \alpha_6 \geq 1$$

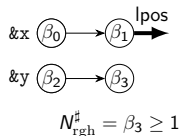
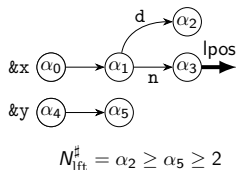
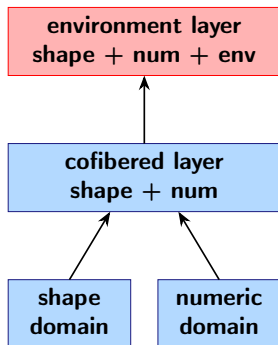
Stage 7: perform the update in the graph

- classic **strong update** in a pointer aware domain
- symbolic node α_4 becomes redundant and can be removed

Widening / join in the combined domain



Widening / join in the combined domain

 $\&x \delta_0$ $\&y \delta_1$

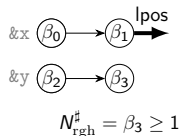
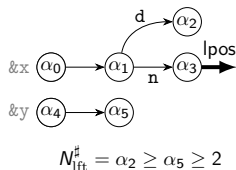
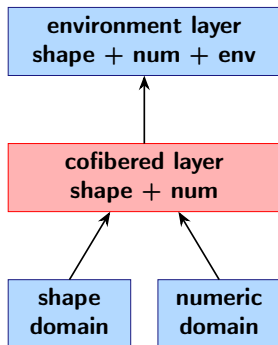
$$\Psi(\alpha_0, \beta_0) = \delta_0$$

$$\Psi(\alpha_4, \beta_2) = \delta_1$$

Stage 1: abstract environment

- compute new abstract environment and initial node relation
e.g., α_0, β_0 both denote $\&x$

Widening / join in the combined domain

 $\&x \ \delta_0$ $\&y \ \delta_1$

$$\Psi(\alpha_0, \beta_0) = \delta_0$$

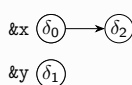
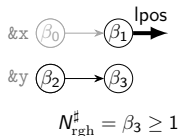
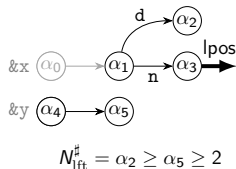
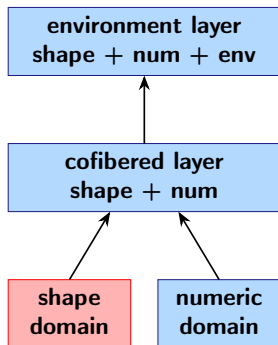
$$\Psi(\alpha_4, \beta_2) = \delta_1$$

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

Widening / join in the combined domain



$$\Psi(\alpha_0, \beta_0) = \delta_0$$

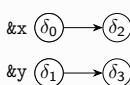
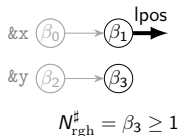
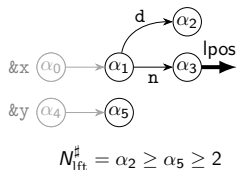
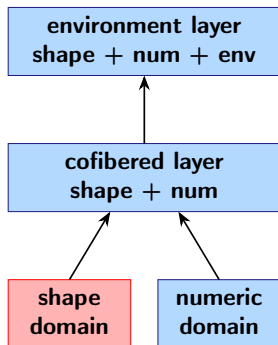
$$\Psi(\alpha_4, \beta_2) = \delta_1$$

$$\Psi(\alpha_1, \beta_1) = \delta_2$$

Stage 2: graph join

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

Widening / join in the combined domain

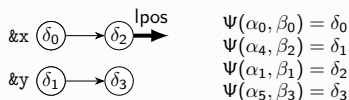
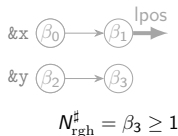
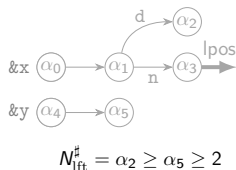
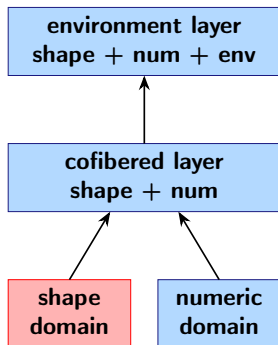


$$\begin{aligned} \Psi(\alpha_0, \beta_0) &= \delta_0 \\ \Psi(\alpha_4, \beta_2) &= \delta_1 \\ \Psi(\alpha_1, \beta_1) &= \delta_2 \\ \Psi(\alpha_5, \beta_3) &= \delta_3 \end{aligned}$$

Stage 2: graph join

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

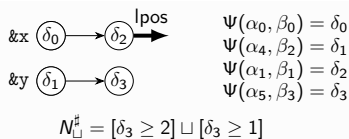
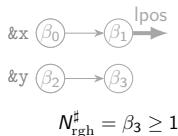
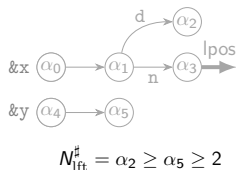
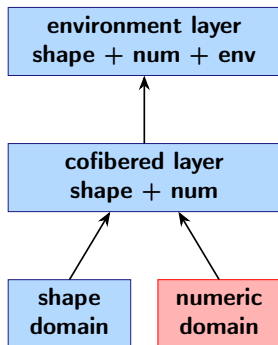
Widening / join in the combined domain



Stage 2: graph join

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

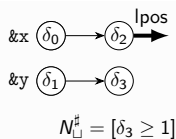
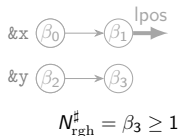
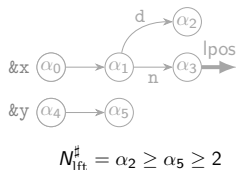
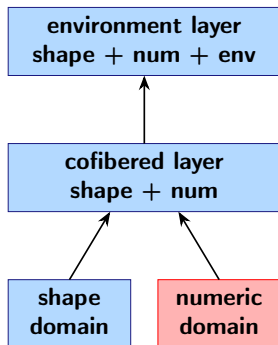
Widening / join in the combined domain



Stage 3: conversion function application in numerics

- remove nodes that were abstracted away
- rename other nodes

Widening / join in the combined domain



$$\begin{aligned} \Psi(\alpha_0, \beta_0) &= \delta_0 \\ \Psi(\alpha_4, \beta_2) &= \delta_1 \\ \Psi(\alpha_1, \beta_1) &= \delta_2 \\ \Psi(\alpha_5, \beta_3) &= \delta_3 \end{aligned}$$

Stage 4: join in the numeric domain

- apply \sqcup for regular join, ∇ 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.
In POPL'08, pages 247–260, 2008.

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:

Separation Logic: A Logic for Shared Mutable Data Structures.

John C. Reynolds.

In LICS'02, pages 55–74, 2002.

Formalizes the Frame rule, among others

Assignment: a simple analysis in Separation Logic (after TVLA)

l, k assumed to be disjoint lists

```
while( $l \neq 0$ ) {
```

```
     $t = l \rightarrow n$ ;
```

```
     $l \rightarrow n = k$ ;
```

```
     $k = l$ ;
```

```
     $l = t$ ;
```

```
}
```