Shape analysis based on separation logic

MPRI — Cours 2.6 "Interprétation abstraite : application à la vérification et à l'analyse statique"

Xavier Rival

INRIA

Oct, 25th, 2017

Overview of the lecture

How to reason about memory properties

Last lecture:

- concrete and abstract memory models
- abstractions for pointers and arrays
- issues specific to the precise analysis of updates
- an introduction to shape analysis with TVLA

Today: systematically avoid weak updates

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

Weak update problems

```
    x ∈ [−10, −5]; y ∈ [5, 10]
1 : int * p;
2 : if(?){
3 :     p = &x;
4 : } else {
5 :     p = &y;
6 : }
7 : *p = 0;
8 : ...
```

	&x	&y	&p
1	$[-10,-5]$	$[5,10]$	$\top$
2	$[-10,-5]$	$[5,10]$	$\top$
3	$[-10,-5]$	$[5,10]$	$\top$
4	$[-10,-5]$	$[5,10]$	$\{\&x\}$
5	$[-10,-5]$	$[5,10]$	$\top$
6	$[-10,-5]$	$[5,10]$	$\{\&y\}$
7	$[-10,-5]$	$[5,10]$	$\{\&x, \&y\}$
8	$[-10,0]$	$[0,10]$	$\{\&x, \&y\}$

- What is the final range for x ?
- What is the final range for y ?

Abstract locations: $\{\&x, \&y, \&p\}$

Imprecise results

- The abstract information about both x and y are weakened
- The fact that $x \neq y$ is lost

Outline

Our model

Not all memory cell corresponds to a variable

- a variable may correspond to **several cells** (structures...)
- **dynamically allocated cells** correspond to no variable at all...

Environment + Heap

- **Addresses** are values: $\mathbb{V}_{\mathrm{addr}} \subseteq \mathbb{V}$
- **Environments** $e \in \mathbb{E}$ map variables into their addresses
- **Heaps** ($h \in \mathbb{H}$) map addresses into values

$$\begin{array}{rcl} \mathbb{E} & = & \mathbb{X} \rightarrow \mathbb{V}_{\mathrm{addr}} \\ \mathbb{H} & = & \mathbb{V}_{\mathrm{addr}} \rightarrow \mathbb{V} \end{array}$$

  h is actually only a partial function
- **Memory states** (or **memories**): $\mathbb{M} = \mathbb{E} \times \mathbb{H}$

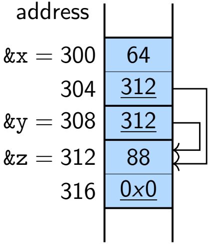
Avoid confusion between heap (function from addresses to values) and dynamic allocation space (often referred to as "heap")

Example of a concrete memory state (variables)

- x and z are two list elements containing values 64 and 88, and where the former points to the latter
- y stores a pointer to z

Memory layout

(pointer values underlined)

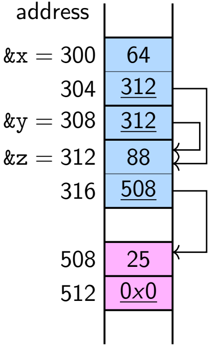


$$
\begin{array}{rlcl}
e: & \mathtt{x} & \mapsto & 300 \\
 & \mathtt{y} & \mapsto & 308 \\
 & \mathtt{z} & \mapsto & 312 \\
 & & & \\
h: & 300 & \mapsto & 64 \\
 & 304 & \mapsto & 312 \\
 & 308 & \mapsto & 312 \\
 & 312 & \mapsto & 88 \\
 & 316 & \mapsto & 0
\end{array}
$$

Example of a concrete memory state (variables + dyn. cell)

- same configuration
- + z points to a heap allocated list element (in purple)

Memory layout



e :	x	$\mapsto$	300
	y	$\mapsto$	308
	z	$\mapsto$	312
h :	300	$\mapsto$	64
	304	$\mapsto$	312
	308	$\mapsto$	312
	312	$\mapsto$	88
	316	$\mapsto$	508
	508	$\mapsto$	25
	512	$\mapsto$	0

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

Several 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

$$
\begin{array}{rcll}
\mathtt{e} & ::= & n \qquad (n \in \mathbb{N}) & \text{constants} \\
& | & \mathtt{l} & \text{l-value} \\
& | & \mathtt{e}_0 + \mathtt{e}_1 & \text{binary operations} \\
& | & \ldots & \\
\mathtt{P} & ::= & \mathtt{e}_0 = \mathtt{e}_1 \mid \mathrm{P}' \vee \mathrm{P}'' \mid \mathrm{P}' \wedge \mathrm{P}'' \ldots & \text{pure predicates}
\end{array}
$$

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

Separation logic: points-to predicates

The next slides introduce the main **separation logic formulas** $\mathtt{F} ::= \ldots$

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

Points-to predicate

- **Predicate:**

$$\mathtt{F} ::= \ldots \mid a \mapsto \mathtt{v} \qquad \text{where } a \text{ is an address and } \mathtt{v} \text{ is a value}$$

- **Concretization:**

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

- **Example:**

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

- We also note $\mathtt{l} \mapsto \mathtt{e}$, as an l-value $\mathtt{l}$ denotes an address

Separation logic: separating conjunction

Merge of concrete heaps: let $\hbar_0, \hbar_1 \in (\mathbb{V}_{\mathrm{addr}} \to \mathbb{V})$, such that $\mathbf{dom}(\hbar_0) \cap \mathbf{dom}(\hbar_1) = \emptyset$; then, we let $\hbar_0 \circledast \hbar_1$ be defined by:

$$\begin{array}{rlcl} \hbar_0 \circledast \hbar_1 : & \mathbf{dom}(\hbar_0) \cup \mathbf{dom}(\hbar_1) & \longrightarrow & \mathbb{V} \\ & x \in \mathbf{dom}(\hbar_0) & \longmapsto & \hbar_0(x) \\ & x \in \mathbf{dom}(\hbar_1) & \longmapsto & \hbar_1(x) \end{array}$$

Separating conjunction

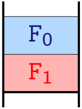
- **Predicate:**

$$\mathtt{F} ::= \ldots \mid \mathtt{F}_0 * \mathtt{F}_1$$

- **Concretization:**

$$\gamma(\mathtt{F}_0 * \mathtt{F}_1) = \{(e, \hbar_0 \circledast \hbar_1) \mid (e, \hbar_0) \in \gamma(\mathtt{F}_0) \wedge (e, \hbar_1) \in \gamma(\mathtt{F}_1)\}$$

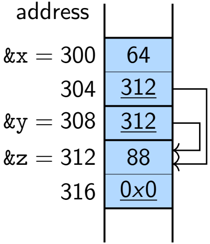
$\mathtt{F}_0 * \mathtt{F}_1$



An example

Concrete memory layout

(pointer values underlined)



$$\begin{array}{rlcl} e: & \mathtt{x} & \mapsto & 300 \\ & \mathtt{y} & \mapsto & 308 \\ & \mathtt{z} & \mapsto & 312 \\ \\ h: & 300 & \mapsto & 64 \\ & 304 & \mapsto & 312 \\ & 308 & \mapsto & 312 \\ & 312 & \mapsto & 88 \\ & 316 & \mapsto & 0 \end{array}$$

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:**

$$\mathtt{F} ::= \ldots \mid \mathtt{F}_0 \wedge \mathtt{F}_1$$

- **Concretization:**

$$\gamma(\mathtt{F}_0 \wedge \mathtt{F}_1) = \gamma(\mathtt{F}_0) \cap \gamma(\mathtt{F}_1)$$

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

- $\&\mathtt{a} \mapsto \&\mathtt{b} \wedge \&\mathtt{b} \mapsto \&\mathtt{a}$
- $\&\mathtt{a} \mapsto \&\mathtt{b} * \&\mathtt{b} \mapsto \&\mathtt{a}$

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:**

$$\frac{(e, h_0) \in \gamma(\mathtt{F}_0) \qquad (e, h_1) \in \gamma(\mathtt{F}_1)}{(e, h_0 \circledast h_1) \in \gamma(\mathtt{F}_0 * \mathtt{F}_1)}$$

- **Non separating conjunction:**

$$\frac{(e, h) \in \gamma(\mathtt{F}_0) \qquad (e, h) \in \gamma(\mathtt{F}_1)}{(e, h) \in \gamma(\mathtt{F}_0 \wedge \mathtt{F}_1)}$$

Reminiscent of Linear Logic [Girard87]:
resource aware / non resource aware conjunction operators

Separation logic: empty store

Empty store

- **Predicate:**

$$\mathtt{F} ::= \ldots \mid \textbf{emp}$$

- **Concretization:**

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

where [] denotes the empty store

- **emp** is the **neutral element for** $*$
- by contrast the **neutral element for** $\wedge$ is $\mathtt{TRUE}$, with concretization:

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

Separation logic: other connectors

Disjunction:

- $\mathtt{F} ::= \ldots \mid \mathtt{F}_0 \vee \mathtt{F}_1$
- concretization:

$$\gamma(\mathtt{F}_0 \vee \mathtt{F}_1) = \gamma(\mathtt{F}_0) \cup \gamma(\mathtt{F}_1)$$

Spatial implication (*aka*, magic wand):

- $\mathtt{F} ::= \ldots \mid \mathtt{F}_0 \mathrel{-\!\!*} \mathtt{F}_1$
- concretization:

$$\gamma(\mathtt{F}_0 \mathrel{-\!\!*} \mathtt{F}_1) = \\ \{(e, h) \mid \forall h_0 \in \mathbb{H},\ (e, h_0) \in \gamma(\mathtt{F}_0) \Longrightarrow (e, h \circledast h_0) \in \gamma(\mathtt{F}_1)\}$$

- 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

$$\begin{array}{rcl}
\gamma(\mathbf{emp}) & = & \mathbb{E} \times \{[]\} \\
\gamma(\mathtt{TRUE}) & = & \mathbb{E} \times \mathbb{H} \\
\gamma(\mathtt{l} \mapsto \mathtt{v}) & = & \{(e, [[\![\mathtt{l}]\!](e, h) \mapsto \mathtt{v}]) \mid e \in \mathbb{E}\} \\
\gamma(\mathtt{F}_0 * \mathtt{F}_1) & = & \{(e, h_0 \circledast h_1) \mid (e, h_0) \in \gamma(\mathtt{F}_0) \wedge (e, h_1) \in \gamma(\mathtt{F}_1)\} \\
\gamma(\mathtt{F}_0 \wedge \mathtt{F}_1) & = & \gamma(\mathtt{F}_0) \cap \gamma(\mathtt{F}_1)
\end{array}$$

Concretization of pure formulas is standard

How does this help for program reasoning ?

Programs with pointers: syntax

Syntax extension: quite a few additional constructions

$$
\begin{array}{rcll}
\mathtt{l} & ::= & \textbf{l-values} & \\
& | & \mathtt{x} & (\mathtt{x} \in \mathbb{X}) \\
& | & \ldots & \\
& | & *\mathtt{e} & \text{pointer dereference} \\
& | & \mathtt{l} \cdot \mathtt{f} & \text{field read} \\
\mathtt{e} & ::= & \textbf{expressions} & \\
& | & \mathtt{l} & \\
& | & \ldots & \\
& | & \&\mathtt{l} & \text{"address of" operator} \\
\mathtt{s} & ::= & \textbf{statements} & \\
& | & \ldots & \\
& | & \mathtt{x} = \textbf{malloc}(c) & \text{allocation of } c \text{ bytes} \\
& | & \textbf{free}(\mathtt{x}) & \text{deallocation of the block pointed to by } \mathtt{x}
\end{array}
$$

We do not consider **pointer arithmetics here**

Programs with pointers: semantics

Case of l-values:

$$
\begin{array}{rcl}
[\![\mathtt{x}]\!](e, h) & = & e(\mathtt{x}) \\
[\![*\mathtt{e}]\!](e, h) & = & \left\{ \begin{array}{ll} h([\![\mathtt{e}]\!](e, h)) & \text{if } [\![\mathtt{e}]\!](e, h) \neq 0 \wedge [\![\mathtt{e}]\!](e, h) \in \mathbf{Dom}(h) \\ \Omega & \text{otherwise} \end{array} \right. \\
[\![\mathtt{l} \cdot \mathtt{f}]\!](e, h) & = & [\![\mathtt{l}]\!](e, h) + \mathbf{offset}(\mathtt{f}) \text{ (numeric offset)}
\end{array}
$$

Case of expressions:

$$
[\![\mathtt{l}]\!](e, h) = h([\![\mathtt{l}]\!](e, h)) \qquad [\![\&\mathtt{l}]\!](e, h) = [\![\mathtt{l}]\!](e, h)
$$

Case of statements:

- **memory allocation** $\mathtt{x} = \mathbf{malloc}(c)$: $(e, h) \rightarrow (e, h')$ where $h' = h[e(\mathtt{x}) \leftarrow k] \uplus \{k \mapsto v_k, k+1 \mapsto v_{k+1}, \ldots, k+c-1 \mapsto v_{k+c-1}\}$ and $k, \ldots, k+c-1$ are fresh in h
- **memory deallocation** $\mathbf{free}(\mathtt{x})$: $(e, h) \rightarrow (e, h')$ where $k = e(\mathtt{x})$ and $h = h' \uplus \{k \mapsto v_k, k+1 \mapsto v_{k+1}, \ldots, k+c-1 \mapsto v_{k+c-1}\}$

Separation logic triple

Program proofs based on Hoare triples

- **Notation**: $\{\mathtt{F}\}p\{\mathtt{F}'\}$ if and only if:

$$\forall s, s' \in \mathbb{S},\ s \in \gamma(\mathtt{F}) \wedge s' \in [\![p]\!](s) \Longrightarrow s' \in \gamma(\mathtt{F}')$$

- Application: **formalize proofs of programs**

A few rules (straightforward proofs):

$$\frac{\mathtt{F}_0 \Longrightarrow \mathtt{F}_0' \qquad \{\mathtt{F}_0'\}\mathtt{b}\{\mathtt{F}_1'\} \qquad \mathtt{F}_1' \Longrightarrow \mathtt{F}_1}{\{\mathtt{F}_0\}\mathtt{b}\{\mathtt{F}_1\}}\ \text{consequence}$$

$$\frac{}{\{\&\mathtt{x} \mapsto ?\}\mathtt{x} := \mathtt{e}\{\&\mathtt{x} \mapsto \mathtt{e}\}}\ \text{mutation}$$

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

(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

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

- 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**:

$$
\begin{array}{l}
\quad \textbf{int}\ \mathtt{i};\\
\quad \textbf{int} * \mathtt{x};\\
\quad \textbf{int} * \mathtt{y};\\
\{\&\mathtt{i} \mapsto ? * \&\mathtt{x} \mapsto ? * \&\mathtt{y} \mapsto ?\}\\
\quad \mathtt{x} = \&\mathtt{i};\\
\{\&\mathtt{i} \mapsto ? * \&\mathtt{x} \mapsto \&\mathtt{i} * \&\mathtt{y} \mapsto ?\}\\
\quad \mathtt{y} = \&\mathtt{i};\\
\{\&\mathtt{i} \mapsto ? * \&\mathtt{x} \mapsto \&\mathtt{i} * \&\mathtt{y} \mapsto \&\mathtt{i}\}\\
\quad * \mathtt{x} = 42;\\
\{\&\mathtt{i} \mapsto 42 * \&\mathtt{x} \mapsto \&\mathtt{i} * \&\mathtt{y} \mapsto \&\mathtt{i}\}
\end{array}
$$

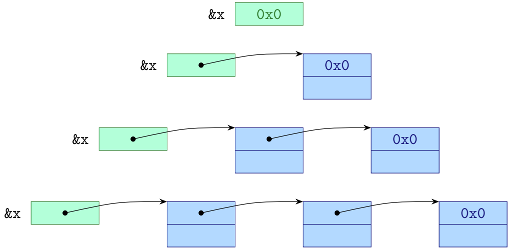
Many other program proofs done using separation logic
e.g., verification of the Deutsch-Shorr-Waite algorithm (biblio)

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

$$\begin{array}{rcl} \alpha \cdot \mathbf{list} & := & \alpha = 0 \wedge \mathbf{emp} \\ & \vee & \alpha \neq 0 \wedge \alpha \cdot \mathtt{next} \mapsto \delta * \alpha \cdot \mathtt{data} \mapsto \beta * \delta \cdot \mathbf{list} \end{array}$$

- Formula abstracting our set of structures:

$$\&\mathtt{x} \mapsto \alpha * \alpha \cdot \mathbf{list}$$

- **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

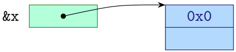
- Inductive predicates can be **unfolded**, by **unrolling their definitions**
  Syntactic unfolding is noted $\xrightarrow{\mathcal{U}}$
- A formula $\mathtt{F}$ with inductive predicates describes all stores described by all formulas $\mathtt{F}'$ such that $\mathtt{F} \xrightarrow{\mathcal{U}} \mathtt{F}'$

Example:

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

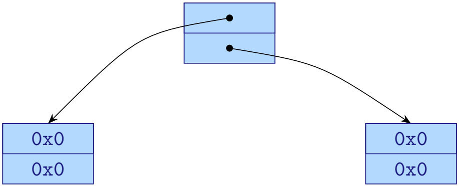
$$
\begin{array}{ll}
\multicolumn{2}{l}{\&\mathtt{x} \mapsto \alpha_0 * \alpha_0 \cdot \mathbf{list}} \\
\xrightarrow{\mathcal{U}} & \&\mathtt{x} \mapsto \alpha_0 * \alpha_0 \cdot \mathtt{next} \mapsto \alpha_1 * \alpha_0 \cdot \mathtt{data} \mapsto \beta_1 * \alpha_1 \cdot \mathbf{list} \\
\xrightarrow{\mathcal{U}} & \&\mathtt{x} \mapsto \alpha_0 * \alpha_0 \cdot \mathtt{next} \mapsto \alpha_1 * \alpha_0 \cdot \mathtt{data} \mapsto \beta_1 * \mathbf{emp} \wedge \alpha_1 = \mathbf{0x0}
\end{array}
$$

- We get the concrete state below:



Example: tree

- **Example:**



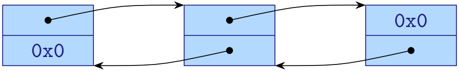
Inductive definition

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

$$\begin{array}{rcll} \alpha \cdot \mathbf{tree} & := & & \alpha = 0 \wedge \mathbf{emp} \\ & & \vee & \alpha \neq 0 \wedge \alpha \cdot \mathtt{left} \mapsto \beta * \alpha \cdot \mathtt{right} \mapsto \delta \\ & & & * \beta \cdot \mathbf{tree} * \delta \cdot \mathbf{tree} \end{array}$$

Example: doubly linked list

- **Example:**



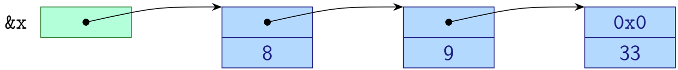
Inductive definition

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

$$
\begin{array}{rcll}
\alpha \cdot \mathbf{dll}(\delta) & := & & \alpha = 0 \wedge \mathbf{emp} \\
& & \vee & \alpha \neq 0 \wedge \alpha \cdot \mathtt{next} \mapsto \beta * \alpha \cdot \mathtt{prev} \mapsto \delta \\
& & & * \, \beta \cdot \mathbf{dll}(\alpha)
\end{array}
$$

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

$$\begin{array}{rcl} \alpha \cdot \mathbf{lsort}_{\mathrm{aux}}(n) & := & \alpha = 0 \wedge \mathbf{emp} \\ & \vee & \alpha \neq 0 \wedge n \leq \beta \wedge \alpha \cdot \mathtt{next} \mapsto \delta \\ & & * \, \alpha \cdot \mathtt{data} \mapsto \beta * \delta \cdot \mathbf{lsort}_{\mathrm{aux}}(\beta) \\ & & \\ \alpha \cdot \mathbf{lsort}() & := & \alpha \cdot \mathbf{lsort}_{\mathrm{aux}}(-\infty) \end{array}$$

A new overview of the remaining part of the lecture

How to apply separation logic to static analysis and design abstract interpretation algorithms based on it ?

In remainder of this lecture, we will:

- choose a **small but expressive set of separation logic formulas**
- combine it **with a numerical abstract domain**
- study algorithms for **local concretization** (equivalent to focus) and **global abstraction** (widening...)

Outline

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:

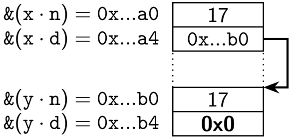
- unstructured formulas can be represented as **ASTs**,
  but this representation is **not easy to manipulate efficiently**
- intuition over memory states typically involves **graphs**

Analysis algorithms:

- inference of "optimal" invariants in SL obviously **not computable**

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

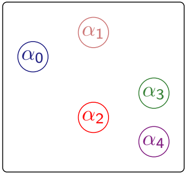
- **Concrete memory states**
  - very **low level** description
    numeric offsets / field names
  - pointers, numeric values:
    raw sequences of bits

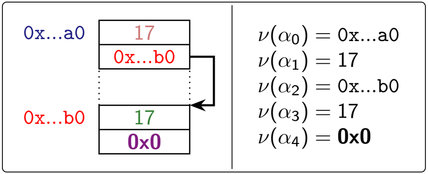


Basic abstraction: structures and their contents (1/2)

- **Concrete memory states**

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

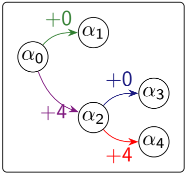


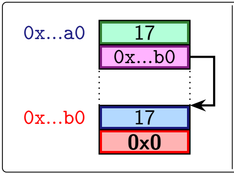


  - characterized by **valuation** ν
  - ν maps **symbolic variables** into **concrete addresses**

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**

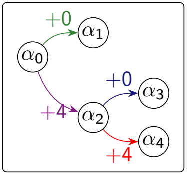


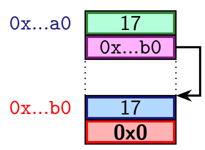


$$
\begin{aligned}
\nu(\alpha_0) &= \texttt{0x...a0}\\
\nu(\alpha_1) &= 17\\
\nu(\alpha_2) &= \texttt{0x...b0}\\
\nu(\alpha_3) &= 17\\
\nu(\alpha_4) &= \mathbf{0x0}
\end{aligned}
$$

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**





$$\begin{aligned}\nu(\alpha_0) &= \texttt{0x...a0}\\ \nu(\alpha_1) &= 17\\ \nu(\alpha_2) &= \texttt{0x...b0}\\ \nu(\alpha_3) &= 17\\ \nu(\alpha_4) &= \mathbf{0x0}\end{aligned}$$

- **Shape graph concretization**

$$\gamma_{\mathsf{sh}}(G) = \{(\hbar, \nu) \mid \ldots\}$$

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: $\mathbb{V}^\sharp = \{\alpha, \beta, \delta, \ldots\}$

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

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

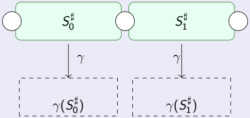
Structure of shape graphs

Distinct edges describe separate regions

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

Separating conjunction

$$\gamma_{\mathbf{sh}}(S_0^\sharp * S_1^\sharp) \quad = \quad \{(h_0 \circledast h_1, \nu) \mid (h_0, \nu) \in \gamma_{\mathbf{sh}}(S_0^\sharp) \wedge (h_1, \nu) \in \gamma_{\mathbf{sh}}(S_1^\sharp)\}$$



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

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

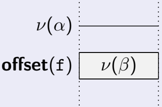
Abstraction of contiguous regions

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
$\alpha \xrightarrow{\mathtt{f}} \beta$	$\alpha \cdot \mathtt{f} \mapsto \beta$	

- **Concretization**:

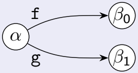
$$\gamma_{\mathbf{sh}}(\alpha \cdot \mathtt{f} \mapsto \beta) = \{([\nu(\alpha) + \mathbf{offset}(\mathtt{f}) \mapsto \nu(\beta)], \nu) \mid \nu : \{\alpha, \beta, \ldots\} \to \mathbb{N}\}$$

Abstraction of contiguous regions

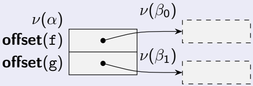
Contiguous regions are described by adjacent points-to edges

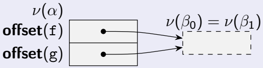
To describe **blocks** containing series of **cells** (*e.g.*, in a **C structure**), shape graphs utilize several outgoing edges from the node representing the base address of the block

Field splitting model

- Separation impacts edges / fields, *not pointers*
- Shape graph 

accounts for both abstract states below:



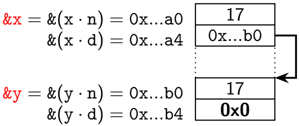


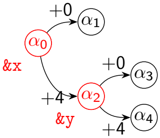
In other words, in a field splitting model, separation:

- asserts addresses are distinct
- says nothing about contents

Abstraction of the environment

Environments bind variables to their (concrete / abstract) address





$$\nu : \quad \begin{array}{lcl} \alpha_0 & \mapsto & \texttt{0x...a0} \\ \alpha_2 & \mapsto & \texttt{0x...b0} \\ \ldots & \mapsto & \ldots \end{array}$$

$$e^\sharp : \quad \begin{array}{ll} \texttt{x} \mapsto \alpha_0 & (\overset{\nu}{\mapsto} \texttt{0x...a0}) \\ \texttt{y} \mapsto \alpha_2 & (\overset{\nu}{\mapsto} \texttt{0x...b0}) \end{array}$$

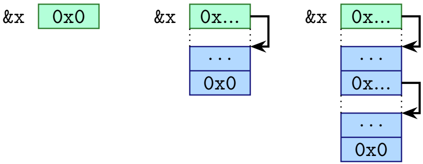
Abstract environments

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

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

Basic abstraction: summarization

Set of all lists of any length:

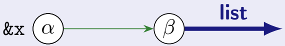


Well-founded list inductive def.

$$\begin{array}{rl} \alpha \cdot \textbf{list} := & \\ & (\textbf{emp} \wedge \alpha = \textbf{0x0}) \\ \vee & (\alpha \cdot \mathtt{d} \mapsto \beta_0 * \alpha \cdot \mathtt{n} \mapsto \beta_1 \\ & \quad * \beta_1 \cdot \textbf{list} \wedge \alpha \neq \textbf{0x0}) \end{array}$$

well-founded predicate

Inductive summary predicates



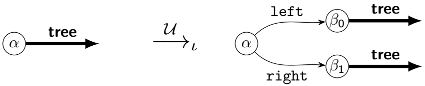
Concretization based on **unfolding** and **least-fixpoint**:

- $\xrightarrow{\mathcal{U}}$ replaces **an $\alpha \cdot$ list predicate** with **one of its premises**
- $\gamma(S^\sharp, \mathtt{F}) = \bigcup \{\gamma(S^\sharp_u, \mathtt{F}_u) \mid (S^\sharp, \mathtt{F}) \xrightarrow{\mathcal{U}} (S^\sharp_u, \mathtt{F}_u)\}$

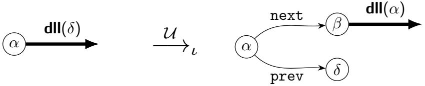
Inductive structures: a few instances

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

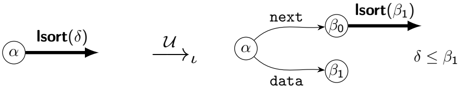
- **More complex shapes**: **trees**



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

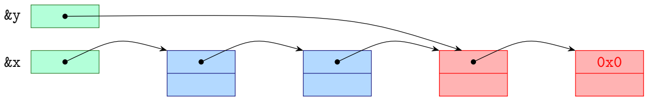


- **Relations between pointers and numerical**: **sorted lists**



Inductive segments

A frequent pattern:



A first attempt:

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

However, the following **does not hold**

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

Why ? **violation of separation!**

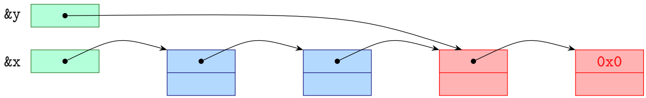
A second attempt:

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

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

Inductive segments

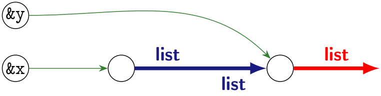
A frequent pattern:



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

$$\begin{array}{rcl} \alpha \cdot \textbf{list\_endp}(\pi) & ::= & (\textbf{emp}, \alpha = \pi) \\ & | & (\alpha \cdot \mathtt{next} \mapsto \beta_0 * \alpha \cdot \mathtt{data} \mapsto \beta_1 \\ & & \quad * \beta_0 \cdot \textbf{list\_endp}(\pi), \alpha \neq 0) \end{array}$$

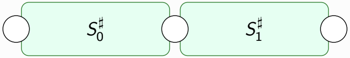
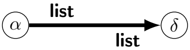
This definition **straightforwardly derives** from **list**
Thus, we make **segments** part of the **fundamental predicates of the domain**



Multi-segments: possible, but harder for analysis

Shape graphs and separation logic

Semantic preserving translation Π of graphs into separation logic formulas:

Graph $S^\sharp \in \mathbb{D}^\sharp_{\mathbf{sh}}$	Translated formula $\Pi(S^\sharp)$
	$\alpha \cdot \mathtt{f} \mapsto \beta$
	$\Pi(S^\sharp_0) * \Pi(S^\sharp_1)$
	$\alpha \cdot \textbf{list}$
	$\alpha \cdot \textbf{list\_endp}(\delta)$
other inductives and segments	similar

Note that:

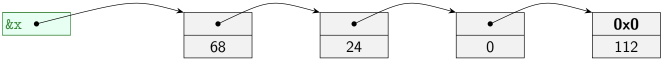
- **shape graphs can be encoded into separation logic formula**
- **the opposite is usually not true**

Outline

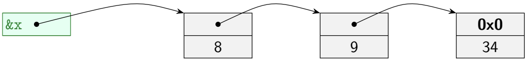
Example

How to express both shape and numerical properties ?

- **Hybrid stores: data stored next to structures**
- **List of even elements:**



- **Sorted list:**

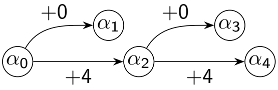


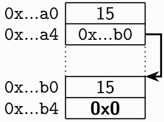
- **Many other examples:**
  - **list of open filed descriptors**
  - **tries**
  - **balanced trees**: red-black, AVL...
- Note: **inductive definitions also talk about data**

Adding value information (here, numeric)

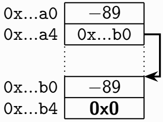
Concrete numeric values appear in the valuation thus the abstracting contents boils down to abstracting ν !

Example: all lists of length 2, with equal data fields **Memory abstraction**:





$$\nu : \begin{array}{ccc} \alpha_1 & \mapsto & 15 \\ \alpha_3 & \mapsto & 15 \\ \ldots & \mapsto & \ldots \end{array}$$



$$\nu : \begin{array}{ccc} \alpha_1 & \mapsto & -89 \\ \alpha_3 & \mapsto & -89 \\ \ldots & \mapsto & \ldots \end{array}$$

Abstraction of valuations: $\nu(\alpha_1) = \nu(\alpha_3)$, **(constraint $\alpha_1 = \alpha_3$)**

A first approach to domain combination

Assumptions:

- Graphs form a **shape domain** $\mathbb{D}^\sharp_{\mathbf{sh}}$
  abstract stores together with a **physical mapping** of nodes
  $$\gamma_{\mathbf{sh}} : \mathbb{D}^\sharp_{\mathbf{sh}} \rightarrow \mathcal{P}((\mathbb{D}^\sharp_{\mathbf{sh}} \rightarrow \mathbb{M}) \times (\mathbb{V}^\sharp \rightarrow \mathbb{V}))$$
- Numerical values are taken in a **numerical domain** $\mathbb{D}^\sharp_{\mathbf{num}}$
  abstracts physical mapping of nodes
  $$\gamma_{\mathbf{num}} : \mathbb{D}^\sharp_{\mathbf{num}} \rightarrow \mathcal{P}((\mathbb{V}^\sharp \rightarrow \mathbb{V}))$$

Combined domain [CR]

- **Set of abstract values:** $\mathbb{D}^\sharp = \mathbb{D}^\sharp_{\mathbf{sh}} \times \mathbb{D}^\sharp_{\mathbf{num}}$
- **Concretization:**

$$\gamma(S^\sharp, N^\sharp) = \{(\hbar, \nu) \in \mathbb{M} \mid \nu \in \gamma_{\mathbf{num}}(N^\sharp) \wedge (\hbar, \nu) \in \gamma_{\mathbf{sh}}(S^\sharp)\}$$

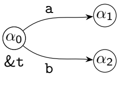
Formalizing the product domain

Can it be described as a reduced product ?

- **Product abstraction:** $\mathbb{D}^\sharp = \mathbb{D}^\sharp_0 \times \mathbb{D}^\sharp_1$
- **Concretization:** $\gamma(x_0, x_1) = \gamma(x_0) \cap \gamma(x_1)$
- **Reduction:** $\mathbb{D}^\sharp_r$ is the quotient of $\mathbb{D}^\sharp$ by the equivalence relation $\equiv$ defined by $(x_0, x_1) \equiv (x_0', x_1') \iff \gamma(x_0, x_1) = \gamma(x_0', x_1')$
- **Abstract order:** pairwise on reduced elements

Several issues:

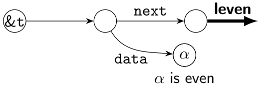
Shape + octagons:



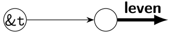
$\alpha_1 \leq \alpha_2$
$\alpha_2 \leq \alpha_1 + \alpha_3$

... what is α_3 ?

How to compare the two elements below ?



and



Towards a more adapted combination operator

Why does this fail here ?

- The set of nodes / symbolic variables **is not fixed**
- Variables represented in the numerical domain depend on the shape abstraction

$\Rightarrow$ Thus the product is **not** symmetric

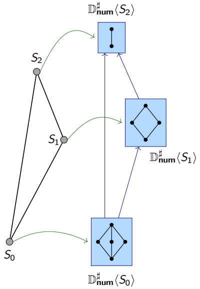
Intuitions

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

Cofibered domain

Definition [AV]

- **Basis:** abstract domain $(\mathbb{D}_0^\sharp, \sqsubseteq^\sharp{}_0)$, with concretization $\gamma_0 : \mathbb{D}_0^\sharp \to \mathbb{D}$
- **Function:** $\phi : \mathbb{D}_0^\sharp \to \mathcal{D}_1$, where each element of $\mathcal{D}_1$ is an abstract domain $(\mathbb{D}_1^\sharp, \sqsubseteq^\sharp{}_1)$, with a concretization $\gamma_{\mathbb{D}_1^\sharp} : \mathbb{D}_1^\sharp \to \mathbb{D}$
- **Domain:** $\mathbb{D}^\sharp$ is the set of **pairs** $(x_0^\sharp, x_1^\sharp)$ **where** $x_1^\sharp \in \phi(x_0^\sharp)$
- **Lift functions:** $\forall x^\sharp, y^\sharp \in \mathbb{D}_0^\sharp$, such that $x^\sharp \sqsubseteq^\sharp{}_0 y^\sharp$, there exists a function $\Pi_{x^\sharp, y^\sharp} : \phi(x^\sharp) \to \phi(y^\sharp)$, that is monotone for $\gamma_{x^\sharp}$ and $\gamma_{y^\sharp}$



- **Generic product**, where the second lattice depends on the first
- Provides a generic scheme for **widening**, **comparison**

Domain operations

- **Lift functions** allow to **switch domain when needed**

Comparison of $(x_0^\sharp, x_1^\sharp)$ and $(y_0^\sharp, y_1^\sharp)$

1. First, **compare** $x_0^\sharp$ **and** $y_0^\sharp$ in $\mathbb{D}_0^\sharp$
2. If $x_0^\sharp \sqsubseteq^\sharp{}_0\, y_0^\sharp$, **compare** $\Pi_{x_0^\sharp, y_0^\sharp}(x_1^\sharp)$ **and** $y_1^\sharp$

Widening of $(x_0^\sharp, x_1^\sharp)$ and $(y_0^\sharp, y_1^\sharp)$

1. First, compute the **widening in the basis** $z_0^\sharp = x_0^\sharp \triangledown y_0^\sharp$
2. Then **move to** $\phi(z_0^\sharp)$, by computing $x_2^\sharp = \Pi_{x_0^\sharp, z_0^\sharp}(x_1^\sharp)$ and $y_2^\sharp = \Pi_{y_0^\sharp, z_0^\sharp}(y_1^\sharp)$
3. Last **widen in** $\phi(z_0^\sharp)$: $z_1^\sharp = x_2^\sharp \triangledown_{z_0^\sharp} y_2^\sharp$

$$(x_0^\sharp, x_1^\sharp) \triangledown (y_0^\sharp, y_1^\sharp) = (z_0^\sharp, z_1^\sharp)$$

Domain operations

Transfer functions, *e.g.*, assignment

- Require memory location be **materialized** in the graph
  - *i.e.*, the graph may have to be modified
  - the numerical component should be updated with lift functions
- Require **update** in the graph and the numerical domain
  - *i.e.*, the numerical component should be kept coherent with the graph

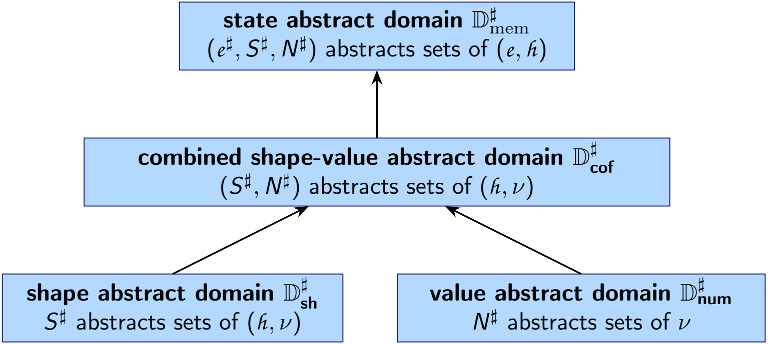
Proofs of soundness of transfer functions rely on:

- the soundness of the lift functions
- the soundness of both domain transfer functions

Overall abstract domain structure

Modular structure

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



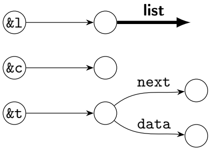
Outline

Static analysis overview

A list insertion function:

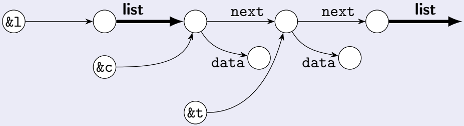
```
list ⋆ l assumed to point to a list
list ⋆ t assumed to point to a list element
list ⋆ c = l;
while(c != NULL && c -> next != NULL && (...)){
    c = c -> next;
}
t -> next = c -> next;
c -> next = t;
```

- **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:** $\mathrm{x} \rightarrow f = \mathrm{y} \rightarrow g$ or $\mathrm{x} \rightarrow f = e_{\mathrm{arith}}$
- **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** $[\![\mathrm{b}]\!] : \mathbb{S} \longrightarrow \mathcal{P}(\mathbb{S})$
- **Abstract post-condition** $[\![\mathrm{b}]\!]^\sharp(\mathbf{S})$, computed by the analysis:

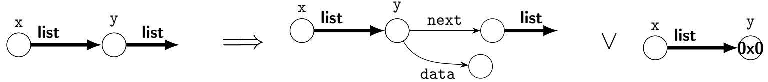
$$s \in \gamma(\mathbf{S}) \Longrightarrow [\![\mathrm{b}]\!](s) \subseteq \gamma([\![\mathrm{b}]\!]^\sharp(\mathbf{S}))$$

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

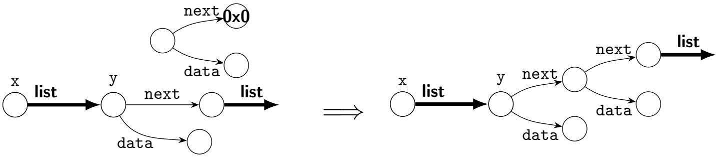
$$\begin{array}{rcl}
[\![\mathrm{b}_0; \mathrm{b}_1]\!]^\sharp(\mathbf{S}) & = & [\![\mathrm{b}_1]\!]^\sharp \circ [\![\mathrm{b}_0]\!]^\sharp(\mathbf{S}) \\
[\![\mathtt{l} = \mathrm{e}]\!]^\sharp(\mathbf{S}) & = & \mathit{assign}(\mathtt{l}, \mathrm{e}, \mathbf{S}) \\
[\![\mathtt{l} = \mathbf{malloc}(n)]\!]^\sharp(\mathbf{S}) & = & \mathit{alloc}(\mathtt{l}, n, \mathbf{S}) \\
[\![\mathbf{free}(\mathtt{l})]\!]^\sharp(\mathbf{S}) & = & \mathit{free}(\mathtt{l}, n, \mathbf{S}) \\
[\![\mathbf{if}(\mathrm{e})\ \mathrm{b}_\mathrm{t}\ \mathbf{else}\ \mathrm{b}_\mathrm{f}]\!]^\sharp(\mathbf{S}) & = & \left\{ \begin{array}{l} \mathit{join}([\![\mathrm{b}_\mathrm{t}]\!]^\sharp(\mathit{test}(\mathrm{e}, \mathbf{S})), \\ \qquad [\![\mathrm{b}_\mathrm{f}]\!]^\sharp(\mathit{test}(\mathrm{e} = \mathbf{false}, \mathbf{S}))) \end{array} \right. \\
[\![\mathbf{while}(\mathrm{e})\mathrm{b}]\!]^\sharp(\mathbf{S}) & = & \mathit{test}(\mathrm{e} = \mathbf{false}, \mathbf{lfp}^\sharp{}_\mathbf{S} F^\sharp) \\
\multicolumn{3}{c}{\text{where, } F^\sharp : \mathbf{S}_0 \mapsto [\![\mathrm{b}]\!]^\sharp(\mathit{test}(\mathrm{e}, \mathbf{S}_0))}
\end{array}$$

The algorithms underlying the transfer functions

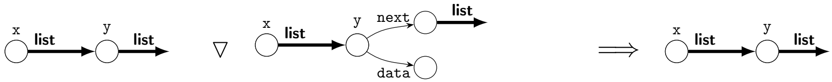
- **Unfolding**: **cases analysis on summaries**



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



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



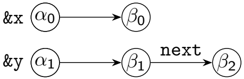
Outline

Analysis of an assignment in the graph domain

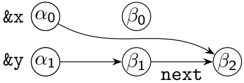
Steps for analyzing $\texttt{x} = \texttt{y -> next}$ (local reasoning)

1. Evaluate **l-value** `x` into **points-to edge** $\alpha \mapsto \beta$
2. Evaluate **r-value** `y -> next` into **node** β'
3. Replace points-to edge $\alpha \mapsto \beta$ with **points-to edge** $\alpha \mapsto \beta'$

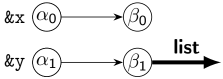
With pre-condition:



- 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**

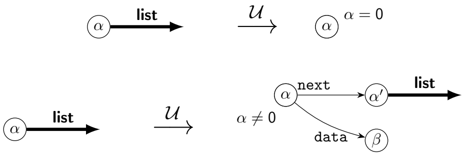
- Abstract state **too abstract**
- We need to **refine it**

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: approximated in the numerical domain**

Soundness: by definition of the concretization of inductive structures

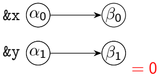
$$\gamma_{\mathsf{sh}}(S^\sharp) \subseteq \bigcup \{\gamma_{\mathsf{sh}}(S_0^\sharp) \mid S^\sharp \xrightarrow{\mathcal{U}} S_0^\sharp\}$$

Analysis of an assignment, with unfolding

Principle

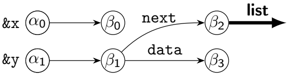
- We have $\gamma_{\mathbf{sh}}(\alpha \cdot \iota) = \bigcup\{\gamma_{\mathbf{sh}}(S^\sharp) \mid \alpha \cdot \iota \xrightarrow{\mathcal{U}} S^\sharp\}$
- Replace $\alpha \cdot \iota$ with a finite number of disjuncts and continue

Disjunct 1:

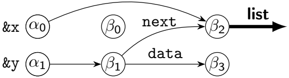


- Step 1 produces $\alpha_0 \mapsto \beta_0$
- **Step 2 fails: Null pointer !**
- In a **correct** program, would be ruled out by a **condition** $\mathtt{y} \neq 0$ *i.e.*, $\beta_1 \neq 0$ in $\mathbb{D}^\sharp_{\mathbf{num}}$

Disjunct 2:



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



Unfold, compute abstract post, and...

Evaluation of a transfer functions (assignment, test...)

1. **evaluate** all **expressions and l-values** that are required
   unfold inductive definitions if needed
2. **compute the effect** of the concrete operation on **fully materialized graph chunks**

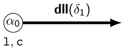
Comparison with the previous lecture:

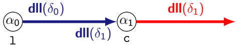
Using separation logic shape graphs	In TVLA
$\begin{tikzcd} \mathcal{S}^\sharp_{\mathrm{pre}} \arrow[d, "\text{unfold (materialize)}"'] & \\ \mathcal{S}^\sharp_{\mathrm{pre,ref}} \arrow[r, "f"] & \mathcal{S}^\sharp_{\mathrm{post}} \end{tikzcd}$	$\begin{tikzcd} \mathcal{S}^\sharp_{\mathrm{pre}} \arrow[d, "\text{partially concretize}"'] & \mathcal{S}^\sharp_{\mathrm{post}} \\ \mathcal{S}_{\mathrm{pre}} \arrow[r, "f"] & \mathcal{S}_{\mathrm{post}} \arrow[u, "\text{abstract}"'] \end{tikzcd}$

When does the abstraction takes place ? More on this a bit later

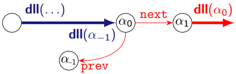
Unfolding and degenerated cases

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

- at ①: 

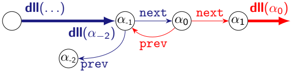
- at ②: 

$\Rightarrow$ **non trivial unfolding**

- **Materialization of** `c -> prev`: 

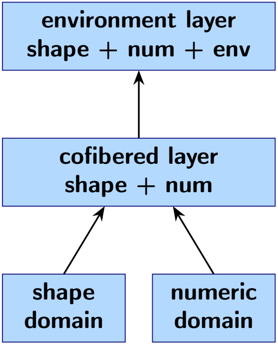
Segment splitting lemma: basis for segment unfolding

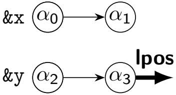
$\alpha_0 \xrightarrow[\iota']{\iota \quad i+j} \alpha_2$ describes the same set of stores as $\alpha_0 \xrightarrow[\iota'']{\iota \quad i} \alpha_1 \xrightarrow[\iota']{\iota'' \quad j} \alpha_2$

- **Materialization of** `c -> prev -> prev`: 

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

Analysis of an assignment in the combined domain



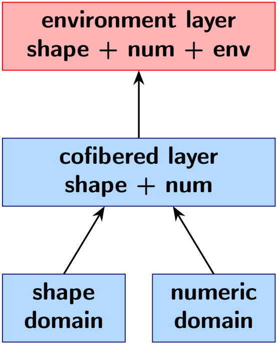


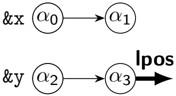
$$N^\sharp = \alpha_1 \geq 0 \wedge \alpha_3 \neq \mathbf{0x0}$$

```
y -> d = x + 1
```

Abstract post-condition ?

Analysis of an assignment in the combined domain





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

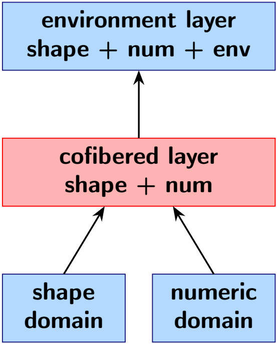
$$\texttt{y -> d} = \texttt{x} + 1 \quad \Rightarrow \quad (*\alpha_2) \cdot \texttt{d} = (*\alpha_0) + 1$$

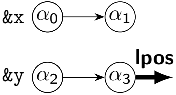
Abstract post-condition ?

Stage 1: environment resolution

- replaces $\texttt{x}$ with $*e^\sharp(\texttt{x})$

Analysis of an assignment in the combined domain





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

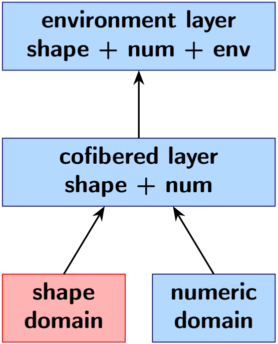
$$(*\alpha_2) \cdot \mathtt{d} = (*\alpha_0) + 1$$

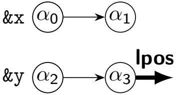
Abstract post-condition ?

Stage 2: propagate into the shape + numerics domain

- only symbolic nodes appear

Analysis of an assignment in the combined domain





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

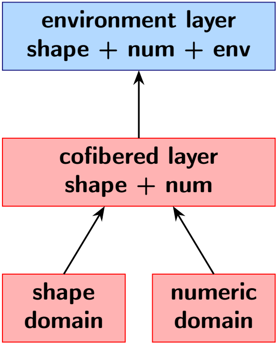
$$(*\alpha_2) \cdot \mathtt{d} = (*\alpha_0) + 1$$

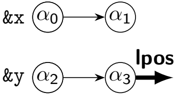
Abstract post-condition ?

Stage 3: resolve cells in the shape graph abstract domain

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

Analysis of an assignment in the combined domain





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

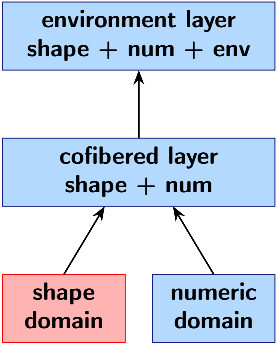
$$(*\alpha_2) \cdot \mathtt{d} = (*\alpha_0) + 1$$

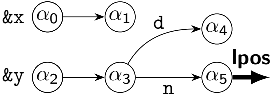
Abstract post-condition ?

Stage 4 (a): unfolding triggered

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

Analysis of an assignment in the combined domain





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

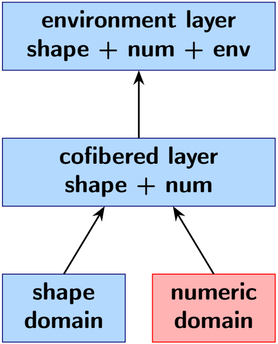
$$(*\alpha_2) \cdot \mathtt{d} = (*\alpha_0) + 1$$

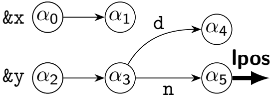
Abstract post-condition ?

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





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

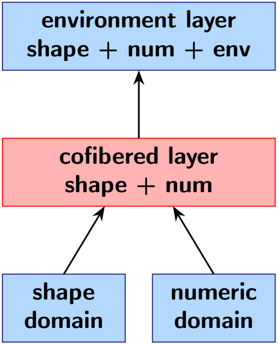
$$(*\alpha_2) \cdot \mathtt{d} = (*\alpha_0) + 1$$

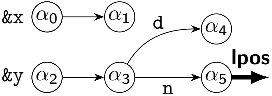
Abstract post-condition ?

Stage 4 (c): unfolding, numeric part

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

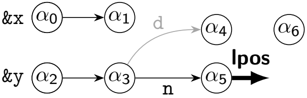
Analysis of an assignment in the combined domain





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

create node α_6

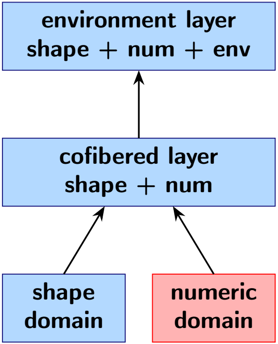


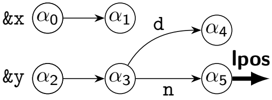
$$N^\sharp = \alpha_1 \geq 0 \wedge \alpha_3 \neq \mathbf{0x0} \wedge \alpha_4 \geq 0$$

Stage 5: create a new node

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

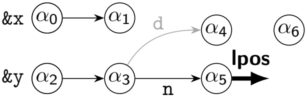
Analysis of an assignment in the combined domain





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

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

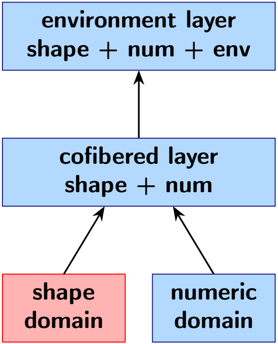


$N^\sharp = \alpha_1 \geq 0 \wedge \alpha_3 \neq \mathbf{0x0} \wedge \alpha_4 \geq 0 \wedge \alpha_6 \geq 1$

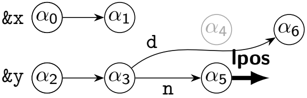
Stage 6: perform numeric assignment

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

Analysis of an assignment in the combined domain



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



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

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

Outline

Need for a folding operation

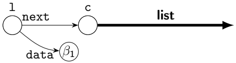
Back to the **list traversal** example:

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

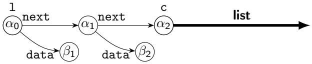
First iterates in the loop:

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

- at **iteration 1**:



- at **iteration 2**:



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

$$\begin{tikzcd} \mathbf{S}_0 \arrow[d, "\text{unfold}"'] & & \\ \mathbf{S}_{0,u} \arrow[r, "f"] & \mathbf{S}_1 \arrow[d, "\text{unfold}"'] & \\ & \mathbf{S}_{1,u} \arrow[r, "f"] & \mathbf{S}_2 \quad \cdots \end{tikzcd}$$

- 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 $\triangledown$

- **Over-approximates join:**

$$\begin{cases} \gamma(X^\sharp) & \subseteq & \gamma(X^\sharp \triangledown Y^\sharp) \\ \gamma(Y^\sharp) & \subseteq & \gamma(X^\sharp \triangledown Y^\sharp) \end{cases}$$

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

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

Canonicalization

Upper closure operator

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

Canonicalization

- **Disjunctive completion:** $\mathbb{D}^\sharp_\vee$ = finite disjunctions over $\mathbb{D}^\sharp$
- **Canonicalization operator** $\rho_\vee$ defined by $\rho_\vee : \mathbb{D}^\sharp_\vee \longrightarrow \mathbb{D}^\sharp_{\mathbf{can}\vee}$ and $\rho_\vee(X^\sharp) = \{\rho(x^\sharp) \mid x^\sharp \in X^\sharp\}$ where ρ is an uco and $\mathbb{D}^\sharp_{\mathbf{can}}$ has finite height

- Canonicalization is used in **many shape analysis tools**: **TVLA** (truth blurring), most **separation logic** based 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

$$\forall(\hbar, \nu) \in \gamma_{\mathsf{sh}}(S_0^\sharp),\ \exists \nu' \in \mathbf{Val},\ (\hbar, \nu') \in \gamma_{\mathsf{sh}}(S_1^\sharp)$$

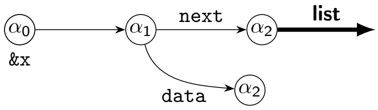
We then note $S_0^\sharp \preccurlyeq S_1^\sharp$

Applications:

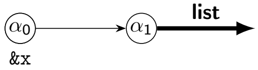
- **inclusion test** (comparison) inputs $S_0^\sharp, S_1^\sharp$; if returns true $S_0^\sharp \preccurlyeq S_1^\sharp$
- **canonicalization** (unary weakening) inputs $S_0^\sharp$ and returns $\rho(S_0^\sharp)$ such that $S_0^\sharp \preccurlyeq \rho(S_0^\sharp)$
- **widening** / **join** (binary weakening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

Weakening: example

We consider $S_0^\sharp$ defined by:



and $S_1^\sharp$ defined by:



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

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

- 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 $S^\sharp_0 \preccurlyeq S^\sharp_{0,\mathbf{weak}}$ and $S^\sharp_1 \preccurlyeq S^\sharp_{1,\mathbf{weak}}$ then:

Separating conjunction rule ($\preccurlyeq_*$)

Let us assume that

- $S^\sharp_0$ and $S^\sharp_1$ have distinct set of **source nodes**
- we can weaken $S^\sharp_0$ into $S^\sharp_{0,\mathbf{weak}}$
- we can weaken $S^\sharp_1$ into $S^\sharp_{1,\mathbf{weak}}$

then: we can weaken $S^\sharp_0 * S^\sharp_1$ into $S^\sharp_{0,\mathbf{weak}} * S^\sharp_{1,\mathbf{weak}}$

Local weakening: unfolding rule

Weakening unfolded region ($\preccurlyeq_{\mathcal{U}}$)

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

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

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

Local weakening: identity rule

Identity weakening ($\preccurlyeq_{\mathbf{Id}}$)

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

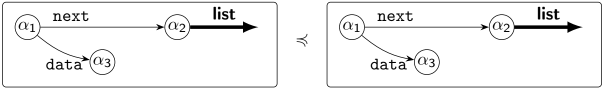
- the proof is trivial:

$$\gamma_{\mathbf{sh}}(S^\sharp) \subseteq \gamma_{\mathbf{sh}}(S^\sharp)$$

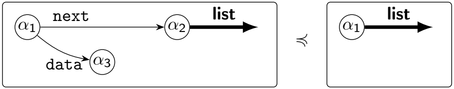
- on itself, this principle is not very useful, but it can be applied locally, and combined with ($\preccurlyeq_{\mathcal{U}}$) on graph regions that are not equal

Local weakening: example

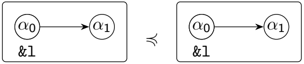
By **rule** ($\preccurlyeq_{\mathbf{Id}}$):



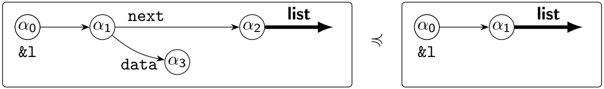
Thus, by **rule** ($\preccurlyeq_{\mathcal{U}}$):



Additionally, by **rule** ($\preccurlyeq_{\mathbf{Id}}$):



Thus, by **rule** ($\preccurlyeq_{*}$):



Inclusion checking rules in the shape domain

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

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:**

$$\begin{array}{rcrcl}
\forall i,\ \Psi(\beta_i) = \alpha_i & \Longrightarrow & \alpha_0 \cdot \mathtt{f} \mapsto \alpha_1 & \sqsubseteq^\sharp_\Psi & \beta_0 \cdot \mathtt{f} \mapsto \beta_1 \\
\Psi(\beta) = \alpha & \Longrightarrow & \alpha \cdot \mathbf{list} & \sqsubseteq^\sharp_\Psi & \beta \cdot \mathbf{list} \\
\forall i,\ \Psi(\beta_i) = \alpha_i & \Longrightarrow & \alpha_0 \cdot \mathbf{list_endp}(\alpha_1) & \sqsubseteq^\sharp_\Psi & \beta_0 \cdot \mathbf{list_endp}(\beta_1)
\end{array}$$

- **Rules on inductives:**

$$\begin{array}{rcrcl}
\forall i,\ \Psi(\beta_i) = \alpha & \Longrightarrow & \mathbf{emp} & \sqsubseteq^\sharp_\Psi & \beta_0 \cdot \mathbf{list_endp}(\beta_1) \\
S_0^\sharp \sqsubseteq^\sharp_\Psi S_1^\sharp \wedge \beta \cdot \iota \xrightarrow{\mathcal{U}} S_1^\sharp & \Longrightarrow & S_0^\sharp & \sqsubseteq^\sharp_\Psi & \beta \cdot \iota
\end{array}$$

if β_1 fresh , $\Psi' = \Psi[\beta_1 \mapsto \alpha_1]$ and $\Psi(\beta_0) = \alpha_0$ then,

$$S_0^\sharp \sqsubseteq^\sharp_{\Psi'} \beta_1 \cdot \mathbf{list} \quad \Longrightarrow \quad \alpha_0 \cdot \mathbf{list\_endp}(\alpha_1) * S_0^\sharp \quad \sqsubseteq^\sharp_\Psi \quad \beta_0 \cdot \iota$$

Inclusion checking algorithm

Comparison of $(e_0^\sharp, S_0^\sharp, N_0^\sharp)$ and $(e_1^\sharp, S_1^\sharp, N_1^\sharp)$

1. start with Ψ defined by $\Psi(\beta) = \alpha$ if and only if there exists a variable $\mathtt{x}$ such that $e_0^\sharp(\mathtt{x}) = \alpha \wedge e_1^\sharp(\mathtt{x}) = \beta$
2. iteratively **apply local rules**, and extend Ψ when needed
3. if the algorithm establishes $S_0^\sharp \preccurlyeq S_1^\sharp$, **compare** $N_0^\sharp \circ \Psi$ and $N_1^\sharp$ **in** $\mathbb{D}^\sharp_{\mathbf{num}}$

- the first step ensures both environments are consistent
- in the last step, composing with Ψ ensures we are comparing consistent numerical values (note that $N_0^\sharp$ and $N_1^\sharp$ may have distinct sets of nodes)

This algorithm is sound:

Soundness

$$(e_0^\sharp, S_0^\sharp, N_0^\sharp) \sqsubseteq^\sharp (e_1^\sharp, S_1^\sharp, N_1^\sharp) \Longrightarrow \gamma(e_0^\sharp, S_0^\sharp, N_0^\sharp) \subseteq \gamma(e_1^\sharp, S_1^\sharp, N_1^\sharp)$$

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 $\forall i \in \{0, 1\},\ \forall s \in \{\mathrm{lft}, \mathrm{rgh}\},\ S^\sharp_{i,s} \preccurlyeq S^\sharp_s$,

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

$$\Psi : \mathbb{V}^\sharp(S^\sharp_{\mathrm{lft}}) \times \mathbb{V}^\sharp(S^\sharp_{\mathrm{rgh}}) \longrightarrow \mathbb{V}^\sharp(S^\sharp)$$

- 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** $(\preccurlyeq_{\mathbf{Id}})$:

- If $S^{\sharp}_{\mathrm{lft}} = \alpha_0 \cdot \mathtt{f} \mapsto \alpha_1$
  and $S^{\sharp}_{\mathrm{lft}} = \beta_0 \cdot \mathtt{f} \mapsto \beta_1$
  and $\Psi(\alpha_0, \beta_0) = \delta_0$, $\Psi(\alpha_1, \beta_1) = \delta_1$, then:

$$S^{\sharp}_{\mathrm{lft}} \nabla S^{\sharp}_{\mathrm{rgh}} = \delta_0 \cdot \mathtt{f} \mapsto \delta_1$$

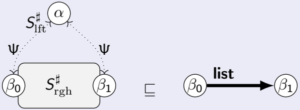
- If $S^{\sharp}_{\mathrm{lft}} = \alpha_0 \cdot \mathbf{list}$
  and $S^{\sharp}_{\mathrm{lft}} = \beta_0 \cdot \mathbf{list}_1$
  and $\Psi(\alpha_0, \beta_0) = \delta_0$, then:

$$S^{\sharp}_{\mathrm{lft}} \nabla S^{\sharp}_{\mathrm{rgh}} = \delta_0 \cdot \mathbf{list}$$

Over-approximation of union: segment introduction rule

Rule

if

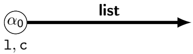


then

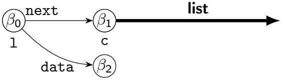
$$\begin{cases} S^\sharp_{\mathrm{lft}} \triangledown S^\sharp_{\mathrm{rgh}} = \delta_0 \xrightarrow{\textbf{list}} \delta_1 \\ (\alpha, \beta_0) \overset{\Psi}{\longleftrightarrow} \delta_0 \\ (\alpha, \beta_1) \overset{\Psi}{\longleftrightarrow} \delta_1 \end{cases}$$

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

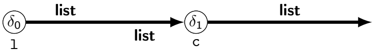
- **before iteration 0:**



- **end of iteration 0:**



- **join, before iteration 1:**

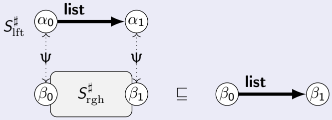


$$\begin{cases} \Psi(\alpha_0, \beta_0) = \delta_0 \\ \Psi(\alpha_0, \beta_1) = \delta_1 \end{cases}$$

Over-approximation of union: segment extension rule

Rule

if



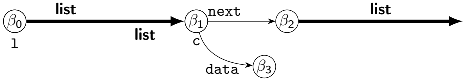
then

$$\left\{ \begin{array}{l} S^\sharp_{\mathrm{lft}} \triangledown S^\sharp_{\mathrm{rgh}} = \delta_0 \xrightarrow{\text{list}} \delta_1 \\ (\alpha_0, \beta_0) \stackrel{\Psi}{\longleftrightarrow} \delta_0 \\ (\alpha_1, \beta_1) \stackrel{\Psi}{\longleftrightarrow} \delta_1 \end{array} \right.$$

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

- **previous invariant before iteration 1:**

- **end of iteration 1:**



- **join, before iteration 1:**

$$\left\{ \begin{array}{lcl} \Psi(\alpha_0, \beta_0) & = & \delta_0 \\ \Psi(\alpha_1, \beta_2) & = & \delta_1 \end{array} \right.$$

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 $(e_0^\sharp, S_0^\sharp, N_0^\sharp)$ and $(e_1^\sharp, S_1^\sharp, N_1^\sharp)$

1. define Ψ, e by $\Psi(\alpha, \beta) = e(\mathrm{x}) = \delta$ (where δ is a fresh node) if and only if $e_0^\sharp(\mathrm{x}) = \alpha \wedge e_1^\sharp(\mathrm{x}) = \beta$
2. iteratively **apply join local rules**, and extend Ψ when new relations are inferred (for instance for points-to edges)
3. if the algorithm computes $S_0^\sharp \triangledown S_1^\sharp = S^\sharp$, compute the widening in the numeric domain: $N^\sharp = N_0^\sharp \circ \Psi_{\mathrm{lft}} \triangledown N_1^\sharp \circ \Psi_{\mathrm{rgh}}$

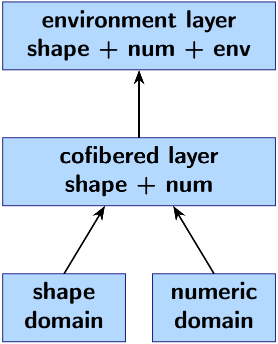
This algorithm is sound:

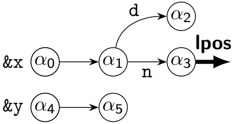
Soundness

$$\gamma(e_0^\sharp, S_0^\sharp, N_0^\sharp) \cup \gamma(e_1^\sharp, S_1^\sharp, N_1^\sharp) \subseteq \gamma(e^\sharp, S^\sharp, N^\sharp)$$

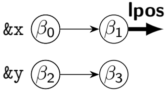
Widening also enforces **termination** (it only introduces segments, and the growth induced by the introduction of segments is bounded)

Widening / join in the combined domain



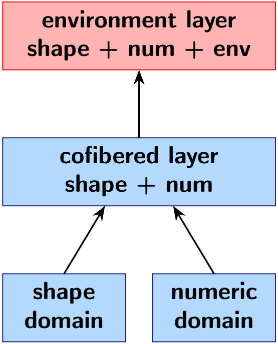


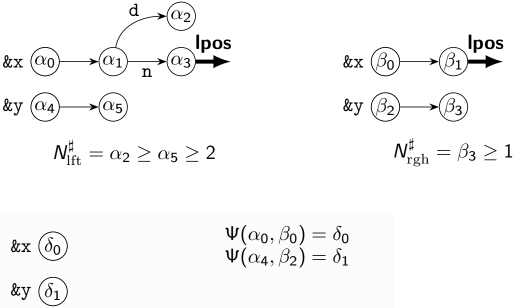
$$N^\sharp_{\mathrm{lft}} = \alpha_2 \geq \alpha_5 \geq 2$$



$$N^\sharp_{\mathrm{rgh}} = \beta_3 \geq 1$$

Widening / join in the combined domain





$N^\sharp_{\mathrm{lft}} = \alpha_2 \geq \alpha_5 \geq 2$

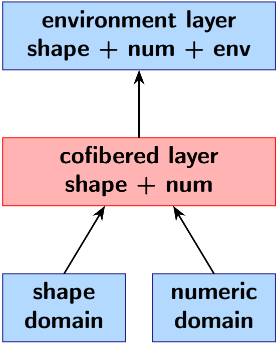
$N^\sharp_{\mathrm{rgh}} = \beta_3 \geq 1$

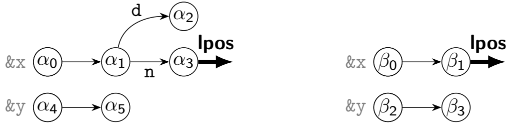
$\Psi(\alpha_0, \beta_0) = \delta_0$
$\Psi(\alpha_4, \beta_2) = \delta_1$

Stage 1: abstract environment

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

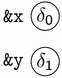
Widening / join in the combined domain





$N^\sharp_{\mathrm{lft}} = \alpha_2 \geq \alpha_5 \geq 2$

$N^\sharp_{\mathrm{rgh}} = \beta_3 \geq 1$



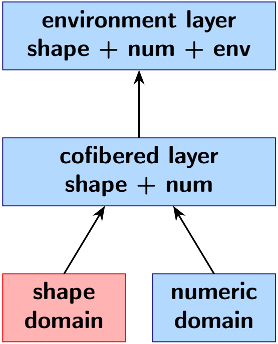
$\Psi(\alpha_0, \beta_0) = \delta_0$
$\Psi(\alpha_4, \beta_2) = \delta_1$

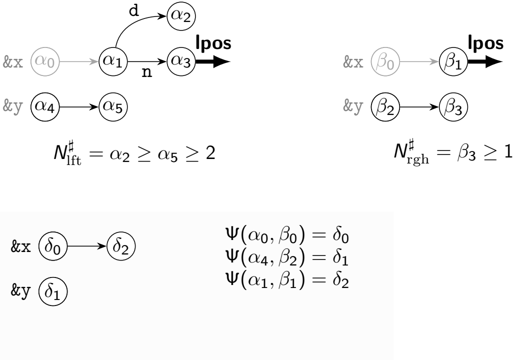
Stage 2: join in the "cofibered" layer

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





$N^\sharp_{\mathrm{lft}} = \alpha_2 \geq \alpha_5 \geq 2$

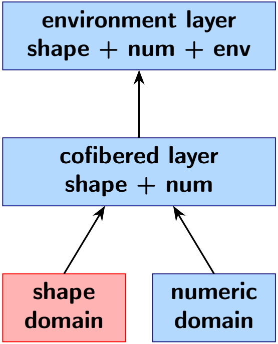
$N^\sharp_{\mathrm{rgh}} = \beta_3 \geq 1$

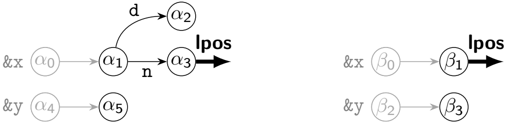
$$\Psi(\alpha_0, \beta_0) = \delta_0$$
$$\Psi(\alpha_4, \beta_2) = \delta_1$$
$$\Psi(\alpha_1, \beta_1) = \delta_2$$

Stage 2: graph join

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

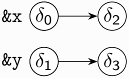
Widening / join in the combined domain





$N^\sharp_{\mathrm{lft}} = \alpha_2 \geq \alpha_5 \geq 2$

$N^\sharp_{\mathrm{rgh}} = \beta_3 \geq 1$

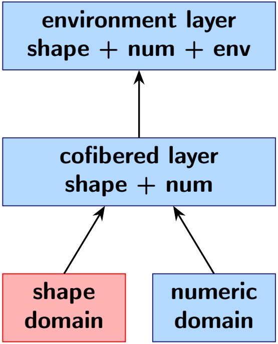


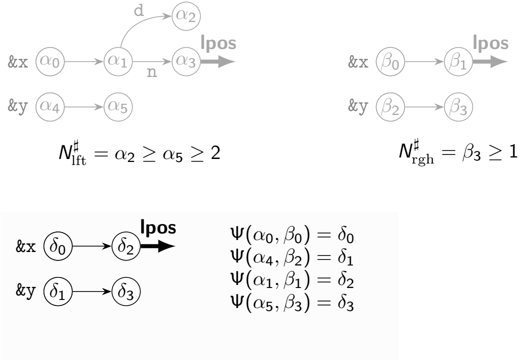
$$\begin{aligned}
\Psi(\alpha_0, \beta_0) &= \delta_0 \\
\Psi(\alpha_4, \beta_2) &= \delta_1 \\
\Psi(\alpha_1, \beta_1) &= \delta_2 \\
\Psi(\alpha_5, \beta_3) &= \delta_3
\end{aligned}$$

Stage 2: graph join

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

Widening / join in the combined domain

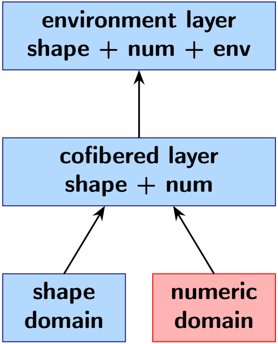


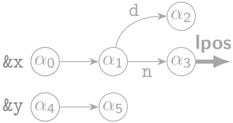


Stage 2: graph join

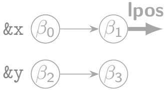
- apply local join rules
  ex: **points-to matching**, **weakening to inductive**...
- incremental algorithm

Widening / join in the combined domain

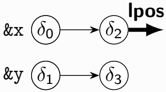




$N^{\sharp}_{\mathrm{lft}} = \alpha_2 \geq \alpha_5 \geq 2$



$N^{\sharp}_{\mathrm{rgh}} = \beta_3 \geq 1$



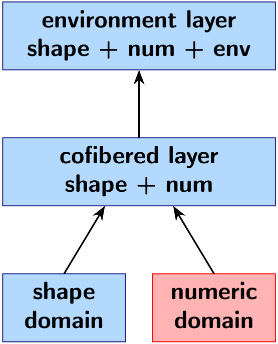
$$\begin{aligned}
\Psi(\alpha_0, \beta_0) &= \delta_0 \\
\Psi(\alpha_4, \beta_2) &= \delta_1 \\
\Psi(\alpha_1, \beta_1) &= \delta_2 \\
\Psi(\alpha_5, \beta_3) &= \delta_3
\end{aligned}$$

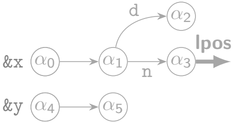
$N^{\sharp}_{\sqcup} = [\delta_3 \geq 2] \sqcup [\delta_3 \geq 1]$

Stage 3: conversion function application in numerics

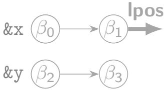
- remove nodes that were abstracted away
- rename other nodes

Widening / join in the combined domain

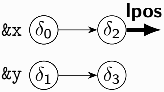




$N^\sharp_{\mathrm{lft}} = \alpha_2 \geq \alpha_5 \geq 2$



$N^\sharp_{\mathrm{rgh}} = \beta_3 \geq 1$



$$\Psi(\alpha_0, \beta_0) = \delta_0$$
$$\Psi(\alpha_4, \beta_2) = \delta_1$$
$$\Psi(\alpha_1, \beta_1) = \delta_2$$
$$\Psi(\alpha_5, \beta_3) = \delta_3$$

$N^\sharp_\sqcup = [\delta_3 \geq 1]$

Stage 4: join in the numeric domain

- apply $\sqcup$ for regular join, $\triangledown$ for a widening

Outline

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:

$$
\begin{array}{rl}
(\mathcal{P}(\mathbb{M}), \subseteq) & \text{is a complete lattice, hence a CPO} \\
F : \mathcal{P}(\mathbb{M}) \to \mathcal{P}(\mathbb{M}) & \text{is the concrete semantic ("post") function of b}
\end{array}
$$

  thus, the concrete semantics writes down as $[\![\mathtt{b}]\!] = \mathbf{lfp}_{\emptyset} F$

- **Assumptions in the abstract level:**

$$
\begin{array}{rl}
\mathbb{M}^\sharp & \text{set of abstract elements, no order a priori} \\
\gamma_{\mathrm{mem}} : \mathbb{M}^\sharp \to \mathcal{P}(\mathbb{M}) & \text{concretization} \\
F^\sharp : \mathbb{M}^\sharp \to \mathbb{M}^\sharp & \text{sound abstract semantic function} \\
 & \textit{i.e.}\text{, such that } F \circ \gamma_{\mathrm{mem}} \subseteq \gamma_{\mathrm{mem}} \circ F^\sharp \\
\triangledown : \mathbb{M}^\sharp \times \mathbb{M}^\sharp \to \mathbb{M}^\sharp & \text{widening operator, terminates, and such that} \\
 & \gamma_{\mathrm{mem}}(m_0^\sharp) \cup \gamma_{\mathrm{mem}}(m_1^\sharp) \subseteq \gamma_{\mathrm{mem}}(m_0^\sharp \triangledown m_1^\sharp)
\end{array}
$$

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:

$$\left\{ \begin{array}{rcl} m_0^\sharp & = & \bot \\ m_{n+1}^\sharp & = & m_n^\sharp \triangledown F^\sharp(m_n^\sharp) \end{array} \right.$$

Soundness proof:

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

$$\mathbf{lfp}_\emptyset F = \bigcup_k F^k(\emptyset) \subseteq \gamma_{\mathrm{mem}}(m_N^\sharp)$$

Discussion on the abstract ordering

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

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

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

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

$$\textbf{is\_le}(m_0^\sharp, m_1^\sharp) = \textbf{true} \quad \Longrightarrow \quad \gamma_{\text{mem}}(m_0^\sharp) \subseteq \gamma_{\text{mem}}(m_1^\sharp)$$

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

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)

Partial concretization, Global abstraction

Separation and summaries should be maintained by the analysis

Last lecture:

$$\begin{tikzcd} \mathcal{S}^\sharp_{\mathrm{pre}} \arrow[d, "\text{partially concretize}"'] & \mathcal{S}^\sharp_{\mathrm{post}} \\ \mathcal{S}_{\mathrm{pre}} \arrow[r, "f"] & \mathcal{S}_{\mathrm{post}} \arrow[u, "\text{abstract}"'] \end{tikzcd}$$

Today, two separate processes:

Local (partial) concretization

For **materialization**:

$$\begin{tikzcd} \mathcal{S}^\sharp_{\mathrm{pre}} \arrow[d, "\text{unfold (materialize)}"'] & \\ \mathcal{S}^\sharp_{\mathrm{pre,ref}} \arrow[r, "f"] & \mathcal{S}^\sharp_{\mathrm{post}} \end{tikzcd}$$

Global abstraction

Widening on loop heads:

$$\begin{tikzcd} & \mathcal{S}^\sharp_0 \triangledown \mathcal{S}^\sharp_1 & \\ \mathcal{S}^\sharp_0 \arrow[r] & \triangledown \arrow[u] & \mathcal{S}^\sharp_1 \arrow[l] \end{tikzcd}$$

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 et Hongseok Yang.** In TACAS'06, pages 287–302.
- **[CR]**: **Relational inductive shape analysis.** **Bor-Yuh Evan Chang et Xavier Rival.** In POPL'08, pages 247–260, 2008.
- **[AV]**: **Abstract Cofibered Domains: Application to the Alias Analysis of Untyped Programs.** **Arnaud Venet.** In SAS'96, pages 366–382.

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

Outline

Internship on memory abstraction

Reduced product of TVLA and separation logic abstract domains:

- **reduced product** allows to express conjunctive properties
  often used in numeric abstract domains, but not for heap abstraction
- **TVLA** (previous course) uses low level local predicates
- **separation logic** is based on region predicates
- how to **combine** them ? what information would we gain ?

Summarization based on universal quantification:

- memory abstractions use **summarization**
  for arrays, arrays segments, linked structures...
- **another form of summarization** based on an **unbounded set** E

$$\circledast\{P(x) \mid x \in E\}$$

  definition of fold / unfold, analysis operations...
- analysis of new kinds of structures, *e.g.*, **union finds**

Internship on Automated Verification of Fault-Tolerant Distributed Systems

Supervised by Cezara Drăgoi, **PhD funding available**

Fault tolerance is achieved using replication: the application/data is copied on different processess; at the core of replication is the consensus that ensures that all replicas are identical!

Theoretical and practical challenges

Class of programs:

- **Implementation of consensus:** Zab, Viewstamped, Multi-Paxos, PBFT (Practical Byzantine Faut Tolerant consensus)
- **Consensus protocols but also other weaker forms of agreement**, *e.g.*, lattice agreement, blockchain, where replicas are different but there is a notion of convergence

Verification challenges:

- **Abstract domains for consensus**: logics that capture the specification and the transition relation of the protocol/implementation with good algorithmic properties
- **Development of modular verification techniques** for safety and liveness, *e.g.*, Hoare like style reasoning