Shape analysis based on separation logic

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

Xavier Rival

INRIA

Dec, 17th, 2014
Overview of the lecture

How to reason about memory properties

Last lecture:
- analyses specific to several kinds of structures
- concrete and abstract memory models
- an introduction to shape analysis with TVLA

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

1. An introduction to separation logic
2. A shape abstract domain relying on separation
3. Combination with a numerical domain
4. Standard static analysis algorithms
5. Inference of inductive definitions / call-stack summarization
6. Conclusion
Our model

Environment + Heap

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

\[ E = X \rightarrow V_{\text{addr}} \]
\[ H = V_{\text{addr}} \rightarrow V \]

- $h$ is actually only a partial function
- **Memory states:**

\[ M = E \times H \]
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)

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

<table>
<thead>
<tr>
<th>e : x</th>
<th>300</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>308</td>
</tr>
<tr>
<td>z</td>
<td>312</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>h : 300</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>304</td>
<td>312</td>
</tr>
<tr>
<td>308</td>
<td>312</td>
</tr>
<tr>
<td>312</td>
<td>88</td>
</tr>
<tr>
<td>316</td>
<td>0</td>
</tr>
</tbody>
</table>
Example of a concrete memory state (variables + heap)

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

**Memory layout**

<table>
<thead>
<tr>
<th>address</th>
<th>&amp;x = 300</th>
<th>304</th>
<th>&amp;y = 308</th>
<th>308</th>
<th>&amp;z = 312</th>
<th>312</th>
<th>316</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>64</td>
<td>312</td>
<td>312</td>
<td>88</td>
<td>508</td>
<td>508</td>
<td>512</td>
</tr>
</tbody>
</table>

$e : \begin{align*} 
\text{x} & \mapsto 300 \\
\text{y} & \mapsto 308 \\
\text{z} & \mapsto 312 
\end{align*}$

$h : \begin{align*} 
300 & \mapsto 64 \\
304 & \mapsto 312 \\
308 & \mapsto 312 \\
312 & \mapsto 88 \\
316 & \mapsto 508 \\
508 & \mapsto 25 \\
512 & \mapsto 0 
\end{align*}$
A introduction to separation logic

Weak update problems

\[
x \in [-10, -5]; \ y \in [5, 10]
\]

```c
int * p;
if(?)
    p = &x;
else
    p = &y;
⋆p = 0;
```

- What is the final range for \(x\)?
- What is the final range for \(y\)?

- After the `if` statement, \(p\) may contain any address in \{&x, &y\}
- Thus, the assignment must consider all cases, in a conservative way
- Thus, \(x\) may receive a new value (0) or keep its old value
- Conclusion: \(x \in [-10, 0], \ y \in [0, 10]\)

Weak updates

Any imprecision in the analysis may lead to weak updates...
An introduction to separation logic

Separation logic principle: avoid weak updates

How to deal with weak updates?

Avoid them!

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

- Notion of property that holds over a memory region.
- Use a special separating conjunction operator $\ast$.
- Local reasoning:
  powerful principle, which allows to consider only part of the program memory.
- Separation logic has been used in many contexts, including manual verification, static analysis, etc...
An introduction to separation logic

Separation logic

- Logic made of a set of formulas
- inference rules...

Pure formulas

- Set of pure formulas, similar to first order logics

\[
e ::= \ n \quad (n \in \mathbb{N}) \\
| \quad l \quad \text{l-value} \\
| \quad e' + e'' \quad \text{binary} \\
| \quad ... \\
\]

\[
P ::= \ e = e' \mid P' \lor P'' \mid P' \land P'' \ldots \\
\]

- Denote numerical properties among the values

Heap formulas (syntax on the next slide)

- Set of formulas to describe concrete heaps
- Concretization relation of the form \((e, h) \in \gamma(F)\)
An introduction to separation logic

Heap formulas

Main connectors

Each formula describes a **heap region**

\[
F ::= \text{emp} \quad \text{empty region} \\
| \text{true} \quad \text{complete heap} \\
| l \mapsto v \quad \text{memory cell} \\
| F' \ast F'' \quad \text{separating conjunction} \\
| F' \land F'' \quad \text{classical conjunction} \\
| \ldots \quad \text{many other connectors (see biblio)}
\]

Denotations: the usual stuff...

- \( \gamma(\text{emp}) = \emptyset \); \( \gamma(\text{true}) = \mathbb{M} \)
- \( (e, h) \in \gamma(F' \land F'') \) if and only if \( (e, h) \in \gamma(F') \) and \( (e, h) \in \gamma(F'') \)

**Separating conjunction:** next slide...
### An introduction to separation logic

#### The separating conjunction

### Single cells

\[(e, h) \in \gamma(l \mapsto v) \text{ if and only if } h = \llbracket l \rrbracket(e, h) \mapsto v\]

#### Merge of concrete stores

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 \(h_0 \otimes h_1\) be defined by:

\[
\begin{align*}
  h_0 \otimes h_1 : \quad \text{dom}(h_0) \cup \text{dom}(h_1) & \rightarrow \mathbb{V} \\
  x \in \text{dom}(h_0) & \mapsto h_0(x) \\
  x \in \text{dom}(h_1) & \mapsto h_1(x)
\end{align*}
\]

### Concretization of separating conjunction

- Logical formulas denote sets of heaps; concretization \(\gamma\)
- **Binary logical connector on formulas** \(*\) defined by:

\[
\gamma(F_0 \ast F_1) = \{(e, h_0 \otimes h_1) | (e, h_0) \in \gamma(F_0) \land (e, h_1) \in \gamma(F_1)\}
\]
Separating conjunction vs non separating conjunction

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

\[
\text{if } a \mapsto \& b \land b \mapsto \& a \text{ then the same heap verifies } a \mapsto \& b \text{ and } b \mapsto \& a \text{ and there can be only one cell thus } a = b
\]

\[
\text{if } a \mapsto \& b \ast b \mapsto \& a \text{ then two separate sub-heaps respectively satisfy } a \mapsto \& b \text{ and } b \mapsto \& a \text{ thus } a \neq b
\]

- Separating conjunction and non-separating conjunction have **very different properties**
- Both express very different properties e.g., no ambiguity on weak / strong updates
An introduction to separation logic

An example

Concrete memory layout
(pointer values underlined)

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

A formula that abstracts away the addresses:

\[
\begin{align*}
  x & \mapsto \langle 64, \&z \rangle \\
  y & \mapsto \&z \\
  z & \mapsto \langle 88, 0 \rangle
\end{align*}
\]
Separating and non separating conjunction

- There are two conjunction operators $\land$ and $\ast$
- How to relate them?

**Separating conjunction vs normal conjunction**

\[
\begin{align*}
(e, h_0) & \in \gamma(F_0) & (e, h_1) & \in \gamma(F_1) & (e, h) & \in \gamma(F_0 \ast F_1) \\
(e, h_0 \otimes h_1) & \in \gamma(F_0 \ast F_1) & (e, h_0) & \in \gamma(F_0) & (e, h_1) & \in \gamma(F_1)
\end{align*}
\]

- Reminiscent of **Linear Logic** [Girard87]: resource aware / non resource aware conjunction operators
Syntax extension: quite a few additional constructions

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

We do not consider pointer arithmetics here
Programs with pointers: semantics

Case of l-values:

\[
\begin{align*}
[x](e, h) &= e(x) \\
[*e](e, h) &= \begin{cases} 
  h([e](e, h)) & \text{if } [e](e, h) \neq 0 \land [e](e, h) \in \text{Dom}(h) \\
  \Omega & \text{otherwise}
\end{cases} \\
[l \cdot f](e, heap) &= [l](e, h) + \text{offset}(f) \text{ (numeric offset)}
\end{align*}
\]

Case of expressions:

\[
\begin{align*}
[l](e, heap) &= h([l](e, heap)) \\
[l \& l](e, heap) &= [l](e, heap)
\end{align*}
\]

Case of statements:

- **memory allocation** \( x = \text{malloc}(c): (e, h) \rightarrow (e, h') \) where
  \( h' = h[e(x) \leftarrow k] \cup \left\{ k \mapsto v_k, k + 1 \mapsto v_{k+1}, \ldots, k + c - 1 \mapsto v_{k+c-1} \right\} \)
  and \( k, \ldots, k + c - 1 \) are fresh in \( h \)

- **memory deallocation** \( \text{free}(x): (e, h) \rightarrow (e, h') \) where \( k = e(x) \) and
  \( h = h' \cup \left\{ k \mapsto v_k, k + 1 \mapsto v_{k+1}, \ldots, k + c - 1 \mapsto v_{k+c-1} \right\} \)
Separating logic triple

Program proofs based on triples

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

**Hoare triples**

- **Application:** formalize proofs of programs

A few rules (straightforward proofs):

\[
\begin{align*}
F_0 & \implies F_0' & \{F_0'\} p \{F_1'\} & F_1' & \implies F_0' \\
\{F_0\} p \{F_1\} & & \text{consequence}
\end{align*}
\]

\[
\begin{align*}
\{x \mapsto ?\} x := e & \{x \mapsto e\} & \text{mutation} \\
\{x \mapsto ? * F\} x := e & \{x \mapsto e * F\} & \text{mutation} \quad 2
\end{align*}
\]

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

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

**Theorem: the frame rule**

\[
\frac{\{F_0\} s \{F_1\}}{\{F_0 \ast F\} s \{F_1 \ast F\}} \quad \text{frame}
\]

- Proof by induction on the rules
  (see biblio for a more complete set of rules)
- Rules are proved by case analysis on the program syntax

**We can reason locally about programs**
Application of the frame rule

Let us consider the program below:

```c
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
```

- Each step impacts a disjoint memory region
- This case is easy
  
  See biblio for more complex applications
  
  (verification of the Deutsch-Shorr-Waite algorithm)
An introduction to separation logic

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:

  \&x \quad \begin{array}{c}
  \text{0x0}
  \end{array}

  \begin{array}{c}
  \text{0x0}
  \end{array}

  \begin{array}{c}
  \text{0x0}
  \end{array}

  \begin{array}{c}
  \text{0x0}
  \end{array}

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

List definition

\[ \alpha \cdot \text{list} := \begin{align*}
\alpha &= 0 \land \text{emp} \\
\lor \quad \alpha &\neq 0 \land \alpha \cdot \text{next} \rightarrow \gamma \cdot \alpha \cdot \text{data} \rightarrow \beta \cdot \gamma \cdot \text{list}
\end{align*} \]

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

- **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 \( \mathcal{U} \rightarrow \)

- A formula \( F \) with inductive predicates describes all stores described by all formulas \( F' \) such that \( F \mathcal{U} \rightarrow F' \)

**Example:**

- Let us start with \( x \mapsto \alpha_0 \cdot \alpha_0 \cdot \text{list} \); we can unfold it as follows:
  \[
  \begin{align*}
  \&x &\mapsto \alpha_0 \cdot \alpha_0 \cdot \text{list} \\
  \mathcal{U} \rightarrow &\&x \mapsto \alpha_0 \cdot \alpha_0 \cdot \text{next} \mapsto \alpha_1 \cdot \alpha_0 \cdot \text{data} \mapsto \beta_1 \cdot \alpha_1 \cdot \text{list} \\
  \mathcal{U} \rightarrow &\&x \mapsto \alpha_0 \cdot \alpha_0 \cdot \text{next} \mapsto \alpha_1 \cdot \alpha_0 \cdot \text{data} \mapsto \beta_1 \cdot \text{emp} \land \alpha_1 = 0x0
  \end{align*}
  \]

- We get the concrete state below:
Example: tree

Example:

\[ \alpha \cdot \text{tree} := \begin{aligned} &\alpha = 0 \land \text{emp} \\ \lor &\alpha \neq 0 \land \alpha \cdot \text{left} \mapsto \beta \ast \alpha \cdot \text{right} \mapsto \gamma \\ \ast &\beta \cdot \text{tree} \ast \gamma \cdot \text{tree} \end{aligned} \]
Example: doubly linked list

Example:

![Doubly Linked List Diagram]

Inductive definition

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

\[
\alpha \cdot \text{dll}(p) := \begin{cases} \alpha = 0 \land \text{emp} \\ \lor \ (\alpha \neq 0 \land \alpha \cdot \text{next} \mapsto \gamma \ast \alpha \cdot \text{prev} \mapsto p \ast \gamma \cdot \text{dll}(\alpha) \end{cases}
\]
Example: sorted list

$\&x \rightarrow \begin{array}{c}
\begin{array}{c}
8 \\
9 \\
0x0 \\
33
\end{array}
\end{array}$

Inductive definition

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

\[
\alpha \cdot \text{lsort}_{\text{aux}}(n) := \begin{cases} 
\alpha = 0 \land \text{emp} \\
\lor \alpha \neq 0 \land \beta \leq n \land \alpha \cdot \text{next} \rightarrow \gamma \\
\ast \alpha \cdot \text{data} \rightarrow \beta \ast \gamma \cdot \text{lsort}_{\text{aux}}(\beta)
\end{cases}
\]

\[
\alpha \cdot \text{lsort()} := \alpha \cdot \text{lsort}_{\text{aux}}(-\infty)
\]
How to apply separation logic to static analysis and design abstract interpretation algorithms based on it?

In this lecture, we will:

- choose a **small but expressive set of separation logic formulas**
- define wide **families of abstract domains**
- study algorithms for **local concretization** (equivalent to focus) and **global abstraction** (widening...)
Outline

1. An introduction to separation logic
2. A shape abstract domain relying on separation
3. Combination with a numerical domain
4. Standard static analysis algorithms
5. Inference of inductive definitions / call-stack summarization
6. Conclusion
Choice of a set of formulas

Our set of predicates

- An abstract value is a **separating conjunction** of terms
- Each term describes
  - either a **contiguous region**
  - or a **summarized region**, described by an **inductive definition**

- Abstract elements have a straightforward interpretation as a **shape graph**
- Unless necessary, we omit environments (concretization into sets of heaps)
Abstraction into separating shape graphs

- **Memory splitting into regions**

- **Graph abstraction:** \[\{\text{values, addresses} \rightarrow \text{nodes}, \text{cells} \rightarrow \text{edges}\}\]

- **Region summarization:**

  - abstraction *parameterized* by a *set of inductive definitions*

- Defines a *concretization relation*

- Let us formalize this...
Contiguous regions

Shape graphs

- **Edges**: denote memory regions
- **Nodes**: denote values, i.e. addresses or cell contents

Points-to edge, denote **contiguous** memory regions

- **Separation logic formula**: $\alpha \cdot f \mapsto \beta$
- **Abstract and concrete** views:

  \[
  \begin{array}{ccc}
    \alpha & \xrightarrow{f} & \beta \\
    \nu(\alpha) & & \nu(\beta) \\
    \text{offset}(f) & & \nu(\beta)
  \end{array}
  \]

- **Concretization**:

  \[
  \gamma_S(\alpha \cdot f \mapsto \beta) = \\
  \{ ([\nu(\alpha) + \text{offset}(f) \mapsto \nu(\beta)], \nu) \mid \nu : \{\alpha, \beta, \ldots\} \rightarrow \mathbb{N} \}
  \]

  $\nu$: **bridge** between memory and values
A shape abstract domain relying on separation

Separation

- A graph = a set of edges
- Denotes the separating conjunction of the edges

Empty graph $\text{emp}$

$$\gamma_S(\text{emp}) = \{ (\emptyset, \nu) \mid \nu : \text{nodes} \rightarrow \mathbb{V} \}$$ i.e., empty store

Separating conjunction

$$\gamma_S(S_0^* \ast S_1^*) = \{ (h_0 \otimes h_1, \nu) \mid (h_0, \nu) \in \gamma_S(S_0^*) \land (h_1, \nu) \in \gamma_S(S_1^*) \}$$
Separation example

Field splitting model

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

accounts for both abstract states below:

In other words, separation
- asserts addresses are distinct
- says nothing about contents
A shape abstract domain relying on separation logic

**Inductive edges**

**List definition**

\[
\alpha \cdot \text{list} ::= (\text{emp}, \alpha = 0) \quad \mid \quad (\alpha \cdot \text{next} \mapsto \beta_0 \ast \alpha \cdot \text{data} \mapsto \beta_1 \ast \beta_0 \cdot \text{list}, \alpha \neq 0)
\]

where \text{emp} denotes the empty heap

**Concretization as a least fixpoint**

Given an inductive def \( \iota \)

\[
\gamma_S(\alpha \cdot \iota) = \bigcup \left\{ \gamma_S(F) \mid \alpha \cdot \iota \xrightarrow{\mathcal{U}} F \right\}
\]

- **Alternate approach:**
  - **index** inductive applications with **induction depth**
  - allows to reason on **length of structures**
Inductive structures IV: a few instances

- **More complex shapes:** trees

  ![Diagram of trees](image)

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

  ![Diagram of doubly-linked lists](image)

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

  ![Diagram of sorted lists](image)
Inductive segments

- **A frequent pattern:**

  \[ \Box y \quad \Box x \quad 0 \times 0 \]

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

  \[
  \alpha \cdot \text{list} \_\text{endp}(\pi) \quad ::= \quad (\text{emp}, \alpha = \pi) \\
  \quad \mid \quad (\alpha \cdot \text{next} \mapsto \beta_0 \ast \alpha \cdot \text{data} \mapsto \beta_1 \ast \beta_0 \cdot \text{list} \_\text{endp}(\pi), \alpha \neq 0)
  \]

- This definition would **derive from list**

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

- **Multi-segments:** possible, but harder for analysis
Outline

1. An introduction to separation logic
2. A shape abstract domain relying on separation
3. Combination with a numerical domain
4. Standard static analysis algorithms
5. Inference of inductive definitions / call-stack summarization
6. Conclusion
How to express both shape and numerical properties?

- **List of even elements:**

  ![List of even elements diagram]

- **Sorted list:**

  ![Sorted list diagram]

- **Many other examples:**
  - list of open filed descriptors
  - tries
  - balanced trees: red-black, AVL...

- **Note:** inductive definitions also talk about data
A first approach to domain combination

**Basis**

- Graphs form a **shape domain** $D^\#_S$
  abstract stores together with a **physical mapping** of nodes
  \[
  \gamma_S : D^\#_S \rightarrow \mathcal{P}((D^\#_S \rightarrow M) \times (\text{nodes} \rightarrow V))
  \]
- Numerical values are taken in a **numerical domain** $D^\#_{\text{num}}$
  abstracts physical mapping of nodes
  \[
  \gamma_{\text{num}} : D^\#_{\text{num}} \rightarrow \mathcal{P}((\text{nodes} \rightarrow V))
  \]

**Concretization of the combined domain [CR]**

\[
\gamma(S^\#, N^\#) = \{ \sigma \in M \mid \exists \nu \in \gamma_{\text{num}}(N^\#), (\sigma, \nu) \in \gamma_S(S^\#) \}
\]

- Quite similar to a **reduced product**
Combination by reduced product

Reduced product

- **Product abstraction:** $\mathbb{D}^\# = \mathbb{D}_0^\# \times \mathbb{D}_1^\#
\quad \gamma(x_0, x_1) = \gamma(x_0) \cap \gamma(x_1)$

- **Reduction:** $\mathbb{D}^\#_r$ is the quotient of $\mathbb{D}^\#$ by the equivalence relation $\equiv$ defined by $(x_0, x_1) \equiv (x'_0, x'_1) \iff \gamma(x_0, x_1) = \gamma(x'_0, x'_1)$

- Domain operations (join, transfer functions) are **pairwise** (are usually composed with reduction)

- Why not to use a product of the shape domain with a numerical domain?

How to compare / join the following two elements?

- $\alpha$ is even

and

- $\alpha$ is even
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

⇒ Thus the product is not symmetric

Intuitions

- Graphs form a shape domain $\mathbb{D}_S^\#$
- For each graph $S^\# \in \mathbb{D}_S^\#$, 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 $\{\alpha_0, \alpha_1, \alpha_2\} \rightarrow \forall$

- An abstract value is a pair $(S^\#, N^\#)$, such that $N^\# \in \mathbb{D}_\text{num}^\langle N^\# \rangle$
Cofibered domain

**Definition [AV]**

- **Basis:** abstract domain $\langle D^\#, \subseteq^\# \rangle$, with concretization $\gamma_0 : D^\#_0 \to D$
- **Function:** $\phi : D^\#_0 \to D_1$, where each element of $D_1$ is an abstract domain $\langle D^\#_1, \subseteq^\#_1 \rangle$, with a concretization $\gamma_{D^\#_1} : D^\#_1 \to D$
- **Lift functions:** $\forall x^\#, y^\# \in D^\#_0$, such that $x^\# \subseteq^\#_0 y^\#$, there exists a function $\Pi_{x^\#, y^\#} : \phi(x^\#) \to \phi(y^\#)$, that is monotone for $\gamma_{x^\#}$ and $\gamma_{y^\#}$
- **Domain:** $D^\#$ is the set of **pairs** $(x^\#_0, x^\#_1)$ where $x^\#_1 \in \phi(x^\#_0)$

- **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 \(x_0^\sharp \sqsubseteq_0 y_0^\sharp\), **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** \(z_0^\sharp = x_0^\sharp \nabla y_0^\sharp\)
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** \(\phi(z_0^\sharp)\): \(z_1^\sharp = x_2^\sharp \nabla z_0^\sharp y_2^\sharp\)

\((x_0^\sharp, x_1^\sharp) \nabla (y_0^\sharp, y_1^\sharp) = (z_0^\sharp, z_1^\sharp)\)
Combination with a numerical domain

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
Outline

1. An introduction to separation logic
2. A shape abstract domain relying on separation
3. Combination with a numerical domain
4. Standard static analysis algorithms
   - Overview of the analysis
   - Post-conditions and unfolding
   - Folding: widening and inclusion checking
5. Inference of inductive definitions / call-stack summarization
6. Conclusion
Standard static analysis algorithms

Overview of the analysis

Static analysis overview

A list insertion function:

```c
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:

Xavier Rival (INRIA)  Shape analysis based on separation logic  Dec, 17th, 2014
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

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

Abstract operators

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

**Denotational style abstract interpreter**

- **Concrete denotational semantics** $\llbracket p \rrbracket : s \mapsto \mathcal{P}(s')$
- **Abstract semantics** $\llbracket p \rrbracket^\#(S) = S'$, computed by the analysis:
  $$s \in \gamma(S) \implies \llbracket p \rrbracket(s) \subseteq \gamma(\llbracket p \rrbracket^\#(S))$$

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

$$\begin{align*}
\llbracket p_0; p_1 \rrbracket^\#(S) &= \llbracket p_1 \rrbracket^\# \circ \llbracket p_0 \rrbracket^\#(S) \\
\llbracket \ell = e \rrbracket^\#(S) &= \text{assign}(\ell, e, S) \\
\llbracket \ell = \text{malloc}(n) \rrbracket^\#(S) &= \text{alloc}(\ell, n, S) \\
\llbracket \text{free}(\ell) \rrbracket^\#(S) &= \text{free}(\ell, n, S) \\
\llbracket \text{if}(e) \ p_t \ \text{else} \ p_f \rrbracket^\#(S) &= \begin{cases} 
\text{join}(\llbracket p_t \rrbracket^\#(\text{guard}(e, S)), \\
\llbracket p_f \rrbracket^\#(\text{guard}(e = \text{false}, S))) 
\end{cases}
\end{align*}$$

$$\llbracket \text{while}(e)p \rrbracket^\#(S) = \text{guard}(e = \text{false}, \text{lfp}^\#_S F^\#)$$

where, $F^\# : S_0 \mapsto \llbracket p \rrbracket^\#(\text{guard}(e, S_0))$

...
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. An introduction to separation logic
2. A shape abstract domain relying on separation
3. Combination with a numerical domain
4. Standard static analysis algorithms
   - Overview of the analysis
   - Post-conditions and unfolding
   - Folding: widening and inclusion checking
5. Inference of inductive definitions / call-stack summarization
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 $\beta'$
3. Replace points-to edge $\alpha \mapsto \beta$ with points-to edge $\alpha \mapsto \beta'$

With pre-condition:

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

With pre-condition:

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

Unfolding principle

- **Case analysis**, based on the inductive definition
- Generates **symbolic disjunctions**
  analysis performed in a **disjunction domain**

- Example, for lists:

  \[
  \alpha \xrightarrow{\text{list}} \alpha \xrightarrow{\mathcal{U}} \alpha = 0
  \]

  \[
  \alpha \xrightarrow{\text{list}} \alpha \xrightarrow{\mathcal{U}} \alpha \neq 0 \quad \text{next} \quad \alpha' \xrightarrow{\text{data}} \beta
  \]

- **Numeric predicates**: approximated in the numerical domain

Soundness: by definition of the concretization of inductive structures

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

**Principle**

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

**Disjunct 1:**

- Step 1 produces $\alpha_0 \xrightarrow{\rightarrow} \beta_0$
- Step 2 fails: Null pointer dereference!

**Disjunct 2:**

- Step 1 produces $\alpha_0 \xrightarrow{\rightarrow} \beta_0$
- Step 2 produces $\beta_2$
- End result:
Unfolding and degenerated cases

\textbf{assume}(l points to a dll)
\begin{align*}
&c = l; \\
&\text{① while}(c \neq \text{NULL} \&\& \text{condition}) \\
&\hspace{1cm} c = c \to \text{next}; \\
&\text{② if}(c \neq 0 \&\& c \to \text{prev} \neq 0) \\
&\hspace{1cm} c = c \to \text{prev} \to \text{prev};
\end{align*}

\begin{itemize}
\item at ①: \[
\begin{array}{c}
α \quad \text{dll}(δ_1) \\
\text{l, c}
\end{array}
\]
\item at ②: \[
\begin{array}{c}
α_0 \quad \text{dll}(δ_0) \\
\text{l} \\
α_1 \quad \text{dll}(δ_1) \\
\text{c}
\end{array}
\]
\end{itemize}

\implies \text{non trivial unfolding}

\textbf{Materialization of } c \to \text{prev}:

\textbf{Segment splitting lemma: basis for segment unfolding}
\begin{align*}
\alpha &\quad i \quad i+j \quad i' \quad α' \\
\text{describes the same set of stores as} \quad & α \quad i \quad i'' \quad j \quad α'
\end{align*}

\textbf{Materialization of } c \to \text{prev} \to \text{prev}:

\textbf{Implementation issue}: discover \textbf{which inductive edge} to unfold \textbf{very hard!}
Analysis of an assignment in the combined domain

$$\text{shape} \ + \ \text{num} \ + \ \text{env}$$

**cobbled layer**

$$\text{shape} \ + \ \text{num}$$

**shape domain**

**numeric domain**

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

$$\& y \ (\alpha_2 \rightarrow \alpha_3)$$

$$lpos$$

$$N = \alpha_1 \geq 0 \land \alpha_3 \neq 0x0$$

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

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

Stage 1: environment resolution

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

Stage 2: propagate into the shape + numerics domain
- only symbolic nodes appear

Abstract post-condition?

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

\[
(\star \alpha_2) \cdot d = (\star \alpha_0) + 1
\]
Standard static analysis algorithms

Analysis of an assignment in the combined domain

environment layer
shape + num + env

cofibered layer
shape + num

shape domain
numeric domain

&x (α0 → α1)
&y (α2 → α3) → lpos

N = α₁ ≥ 0 ∧ α₃ ≠ 0x0

(⋆α₂) · d = (⋆α₀) + 1

Abstract post-condition?

Stage 3: resolve cells in the shape graph abstract domain

• ⋆α₀ evaluates to α₁; ⋆α₂ evaluates to α₃

• (⋆α₂) · d fails to evaluate: no points-to out of α₃
Analysis of an assignment in the combined domain

Stage 4: unfolding (several steps, skipped here)

- locally materialize $\alpha_3 \cdot \text{lpos}$; update keys / relations in the numerics
- 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

Stage 5: create a new node

- new node $\alpha_6$ denotes a new value
  will store the new value
Analysis of an assignment in the combined domain

Stage 6: perform numeric assignment

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

Stage 7: perform the update in the graph

- classic **strong update** in a pointer aware domain
- symbolic node $\alpha_4$ becomes redundant and can be removed
Outline

1. An introduction to separation logic
2. A shape abstract domain relying on separation
3. Combination with a numerical domain
4. Standard static analysis algorithms
   - Overview of the analysis
   - Post-conditions and unfolding
   - Folding: widening and inclusion checking
5. Inference of inductive definitions / call-stack summarization
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):
  
  ```
  assume(l points to a list)
  c = l;
  while(c ≠ NULL){
    c = c → next;
  }
  ```

  - at **iteration 1**:

  - at **iteration 2**:

- 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:**

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

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

\[
\begin{align*}
Y^\#_0 & = X^\#_0 \\
\forall n \in \mathbb{N}, \quad Y^\#_{n+1} & = Y^\#_n \triangledown X^\#_{n+1}
\end{align*}
\]
Canonicalization

Upper closure operator

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

Canonicalization

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

- Usually **more simple to compute**
- Canonicalization is used in many shape analysis tools: TVLA, most separation logic based analysis tools
- However **less powerful** than widening: does not exploit history of computation
Per region weakening

The weakening principles shown in the following apply both in canonicalization and widening approaches.

We can apply the **local reasoning principle** to weakening:
- inclusion test (comparison)
- canonicalization
- join / widening

**Application: inclusion test**
- Operator $\sqsubseteq^\#$ should satisfy $X^\# \sqsubseteq^\# Y^\# \implies \gamma(X^\#) \subseteq \gamma(Y^\#)$
- If $S_0^\# \sqsubseteq^\# S_{0,weak}^\#$ and $S_1^\# \sqsubseteq^\# S_{1,weak}^\#$
Inductive weakening

Weakening identity

- $X^\# \subseteq^* X^\# \ldots$
- Trivial, but useful, when a graph region appears in both widening arguments

Weakening unfolded region

- If $S_0^\# \xrightarrow{U} S_1^\#$, $\gamma_S(S_1^\#) \subseteq \gamma_S(S_0^\#)$
- Soundness follows the soundness of unfolding

Application to a simple example:
Comparison operator in the shape domain

Algorithm structure

- **Based on separation and local reasoning:**
  \[ \gamma_S(S_0^\#) \subseteq \gamma_S(S_1^\#) \implies \gamma_S(S_0^\# \ast S^\#) \subseteq \gamma_S(S_1^\# \ast S^\#) \]

- **Algorithm:**
  - applies local rules and “consumes” graph regions proved weaker
  - keeps discovering new rule applications

- **Structural rules** such as:
  - **segment splitting:**
    \[ S^\# \sqsubseteq^\# \alpha \overset{\ell}{\implies} S^\# \ast \beta \overset{\ell}{\implies} \alpha \sqsubseteq^\# \beta \overset{\ell}{\implies} \]
  - **inductive folding:**
    \[ \alpha \overset{\ell}{\implies} S^\# \sqsubseteq^# S_0^\# \overset{U}{\implies} S_0^\# \]
    \[ \implies S^\# \sqsubseteq^\# \alpha \overset{\ell}{\implies} \]

**Correctness:**

\[ S_0^\# \sqsubseteq^\# S_1^\# \implies \gamma_S(S_0^\#) \subseteq \gamma_S(S_1^\#) \]
Comparison operator in the combined domain

We need to tackle the fact nodes names may differ (cofibered domain)

\[ \alpha_2 \text{ is even} \]

Instrumented comparison in the shape domain

- Comparison \( S_0^\# \sqsubseteq S_1^\# \): rules should compute a physical mapping \( \Psi : \text{nodes}_1 \longrightarrow \text{nodes}_0 \)
- Soundness condition: \( (\sigma, \nu) \in \gamma_S(S_0^\#) \implies (\sigma, \nu \circ \Psi) \in \gamma_S(S_0^\#) \)

Comparison in the cofibered domain

- Lift function for numerical abstract values: \( \Pi_{S_0^\#, S_1^\#}(N_0^\#) = N_0^\# \circ \Psi \)
- Thus, we simply need to compare \( N_0^\# \circ \Psi \) and \( N_1^\# \)
Join operator

- Similar iterative scheme, based on local rules
- But needs to reason locally on **two graphs** in the same time: each rule maps two regions into a common over-approximation

Graph partitioning and mapping

- **Inputs:** $S^\#_0, S^\#_1$
- Performed by a function $\Psi : \text{nodes}_0 \times \text{nodes}_1 \rightarrow \text{nodes}_\sqcup$
- $\Psi$ is computed at the same time as the join

If $\forall i \in \{0, 1\}$, $\forall s \in \{\text{lft}, \text{rgh}\}$, $S^\#_{i,s} \sqsubseteq S^\#_s$, 

\[
\begin{array}{c}
\alpha_0 \quad S^\#_{0,\text{lft}} \quad \alpha_1 \quad S^\#_{1,\text{lft}} \quad \alpha_2 \\
\psi \quad \psi \quad \psi
\end{array}
\begin{array}{c}
\beta_0 \quad S^\#_{0,\text{rgh}} \quad \beta_1 \quad S^\#_{1,\text{rgh}} \quad \beta_2
\end{array}
\begin{array}{c}
\gamma_0 \quad S^\#_0 \quad \gamma_1 \quad S^\#_1 \quad \gamma_2
\end{array}
\]
Segment introduction

**Rule**

\[
\text{if } S_{\text{left}}^\# \psi S_{\text{right}}^\# \subseteq S_{\text{left}}^\# \text{ then } \left\{ \begin{array}{l}
S_{\text{left}}^\# \triangledown S_{\text{right}}^\# = \gamma_0 \\
(\alpha, \beta_0) \leftrightarrow \Psi \gamma_0 \\
(\alpha, \beta_1) \leftrightarrow \Psi \gamma_1
\end{array} \right.
\]

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

- **before iteration 0:**

- **end of iteration 0:**

- **join, before iteration 1:**

\[
\left\{ \begin{array}{l}
\Psi (\alpha_0, \beta_0) = \gamma_0 \\
\Psi (\alpha_0, \beta_1) = \gamma_1
\end{array} \right.
\]
Segment extension

**Rule**

If

\[
S^\#_{\text{left}} \sqsubseteq S^\#_{\text{right}}
\]

then

\[
\begin{cases}
S^\#_{\text{left}} \triangledown S^\#_{\text{right}} = \gamma_0 \\
(\alpha_0, \beta_0) \leftrightarrow \gamma_0 \\
(\alpha_1, \beta_1) \leftrightarrow \gamma_1
\end{cases}
\]

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

- **previous invariant before iteration 1:**

  \[
  \begin{array}{c}
  \alpha_0 \ \text{list} \\
  \beta_0 \ \text{list} \\
  \psi
  \end{array}
  \]

- **end of iteration 1:**

  \[
  \begin{array}{c}
  \beta_0 \ \text{list} \\
  \beta_1 \ \text{next} \\
  \beta_2 \ \text{list}
  \end{array}
  \]

- **join, before iteration 1:**

  \[
  \begin{array}{c}
  \gamma_0 \ \text{list} \\
  \psi(\alpha_0, \beta_0) = \gamma_0 \\
  \psi(\alpha_1, \beta_2) = \gamma_1
  \end{array}
  \]
Rewrite system properties

- Comparison, canonicalization and widening algorithms can be considered **rewriting systems over tuples of graphs**
- Each step applies a rule / computation step

**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
Properties

Inclusion checking is sound

If $S_0 \subseteq S_1$, then $\gamma(S_0) \subseteq \gamma(S_1)$

Canonicalization is sound

$\gamma(S) \subseteq \gamma(\rho_{can}(S))$

Widening is sound and terminating

$\gamma(S_0) \subseteq \gamma(S_0 \triangledown S_1)$
$\gamma(S_1) \subseteq \gamma(S_0 \triangledown S_1)$

$\triangledown$ ensures termination of abstract iterates

- **Soundness** of local reasoning and of local rules
- **Termination of widening**: $\triangledown$ can introduce only segments, and may not introduce infinitely many of them
Widening / join in the combined domain

\[ N = \alpha_2 \geq \alpha_5 \geq 2 \]

\[ N' = \beta_3 \geq 1 \]
Widening / join in the combined domain

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

\[ N = \alpha_2 \geq \alpha_5 \geq 2 \]

\[ N' = \beta_3 \geq 1 \]

\[ \delta_0 \equiv (\alpha_0, \beta_0) \]
\[ \delta_1 \equiv (\alpha_4, \beta_2) \]
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

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
Standard static analysis algorithms

Folding: widening and inclusion checking

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

Stage 4: join in the numeric domain
- apply △ for regular join, ▽ for a widening
Outline

1. An introduction to separation logic
2. A shape abstract domain relying on separation
3. Combination with a numerical domain
4. Standard static analysis algorithms
5. Inference of inductive definitions / call-stack summarization
6. Conclusion
Interprocedural analysis

- Analysis of programs that consist in several functions (or procedures)
- Difficulty: how to cope with multiple (possibly recursive) calls

Relational approach
- analyze each function once
- compute function summaries abstraction of input-output relations
- analysis of a function call: instantiate the function summary (hard)

Inlining approach
- inline functions at function calls
- just an extension of intraprocedural analysis

In this section, we study the inlining approach for recursion
- Side result: a widening for inductive definitions
**Approaches to interprocedural analysis**

<table>
<thead>
<tr>
<th>“relational” approach</th>
<th>“inlining” approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>analyze each definition</td>
<td>analyze each call</td>
</tr>
<tr>
<td>abstracts $\mathcal{P}(\tilde{S} \rightarrow \tilde{S})$</td>
<td>abstracts $\mathcal{P}(S)$</td>
</tr>
</tbody>
</table>

- **Relational** approach:
  - + modularity
  - + reuse of invariants
  - - deals with state relations
  - - complex higher order iteration strategy

- **Inlining** approach:
  - - not modular
  - - re-analysis in $\neq$ contexts
  - + deals with states
  - + straightforward iteration strategy

**Challenge:**
- Relational: frame problem
- Inlining: unbounded calls
Inference of inductive definitions / call-stack summarization

Challenges in interprocedural analysis

```c
void main()
{
    dll * l; // assume l points to a sll
    l = fix(l, NULL);
}
dll * fix(dll * c, dll * p)
{
    dll * ret;
    if(c != NULL){
        c->prev = p;
        if(1) c->next = fix(c->next, c);
        if(check(c->data)){
            ret = c->next;
            remove(c);
        } else ret = c;
    }
    return ret;
}
```

- **Heap** is unbounded, needs abstraction (shape analysis)
- But **stack** may also grow unbounded, needs abstraction
- Complex relations between both **stack** and **heap**
Calling contexts as shape graphs

- **Concrete assembly call stack** modelled in a **separating shape graph** together with the **heap**
  - one node per activation record address
Calling contexts as shape graphs

- Concrete assembly call stack modelled in a separating shape graph together with the heap
  - one node per activation record address
Calling contexts as shape graphs

- **Concrete assembly call stack** modelled in a *separating shape graph* together with the heap
  - one node per activation record address
  - explicit edges for frame pointers
Concrete assembly call stack modelled in a separating shape graph together with the heap
- one node per activation record address
- explicit edges for frame pointers
- local variables turn into activation record fields
Concrete assembly call stack modelled in a separating shape graph together with the heap

- one node per activation record address
- explicit edges for frame pointers
- local variables turn into activation record fields
Concrete assembly call stack modelled in a separating shape graph together with the heap

- one node per activation record address
- explicit edges for frame pointers
- local variables turn into activation record fields
Inference of a call-stack inductive structure

- **Second and third iterates:** a repeating pattern

- Computing an inductive rule for summarization: subtraction
Inference of a call-stack inductive structure

- **Second and third iterates:** a repeating pattern

- Computing an inductive rule for summarization: subtraction
  - subtract **top-most activation record**
Inference of inductive definitions / call-stack summarization

Inference of a call-stack inductive structure

- **Second and third iterates**: a repeating pattern

- Computing an inductive rule for summarization: subtraction
  - subtract *top-most activation record*
  - subtract *common stack region*
Inference of a call-stack inductive structure

- **Second and third iterates:** a repeating pattern

- Computing an inductive rule for summarization: subtraction
  - subtract top-most activation record
  - subtract common stack region
  - gather relations with next activation records: additional parameters
  - collect numerical constraints
Inference of a call-stack inductive structure

- **Second and third iterates**: a repeating pattern

- Computing an inductive rule for summarization: subtraction

Inferred inductive rule

\[
\text{stk}(\beta_1, \beta_2) \xrightarrow{\text{fix}::\text{ctx}} \text{stk}(\beta_0, \beta_1)
\]
Inference of a call-stack summary: widening iterates

- **Fixpoint at function entry:**

  - **first iterate:**
  - **second iterate:**
  - **widened iterate:**

- **Fixpoint reached**

- **Fixpoint upon function return:**
  - function return involves **unfolding** of stack summaries
  - **simpler widening sequence:** no rule to infer
Inference of inductive definitions / call-stack summarization

Widening over inductive definitions

Domain structure

An abstract value should comprise:

- a set $S$ of unfolding rules for the stack inductive
- a shape graph $G$
- a numeric abstract value $N$

Shape graph $G$ is defined in a lattice specified by $S$, thus, this is an instance of the cofibered abstraction

- **Lift functions** are trivial:
  - adding rules simply weakens shape graphs
  - i.e., no need to change them syntactically, their concretization just gets weaker

- **Termination** in the lattice of rules:
  - limiting of the number of rules that can be generated to some given bound
Outline

1. An introduction to separation logic
2. A shape abstract domain relying on separation
3. Combination with a numerical domain
4. Standard static analysis algorithms
5. Inference of inductive definitions / call-stack summarization
6. Conclusion
Abstraction choices

Many families of symbolic abstractions including TVLA and separation logic based approaches

- Variants: region logic, ownership, fractional permissions

Common ingredients

- **Splitting** of the heap in regions
  - **TVLA**: covering, via embedding
  - **Separation logic**: partitioning, enforced at the concrete level
- Use of **induction** in order to summarize large regions
- More **limited pointer analyses**: no inductives, no summarization...
Algorithms

Rather different process, compared to numerical domains

From abstract to concrete (locally)
- **Unfold** abstract properties in a local manner
- Allows **quasi-exact** analysis of usual operations (assignment, condition test...)

From concrete to abstract (globally)
- Guarantees **termination**
- Allows to **infer** pieces of code build complex structures
- **Intuition:**
  - static analysis involves **post-fixpoint** computations (over program traces)
  - widening produces a **fixpoint** over memory cells
Open problems

Many opportunities for research:

- Improving **expressiveness**
  - e.g., sharing in data-structures
    - new abstractions
    - combining several abstractions into more powerful ones

- Improving **scalability**
  - shape analyses remain expensive analyses, with few “cheap” and useful abstractions
  - cut down the cost of complex algorithms
  - isolate smaller families of predicates

- **Applications**, beyond software safety:
  - security
  - verification of functional properties
Internships

Several topics possible, soon to be announced on the lecture webpage:

**Internal reduction operator**
- inductive definitions are very expressive thus tricky to reason about
- design of an internal reduction operator on abstract elements with inductive definitions

**Modular inter-procedural analysis**
- a relation between pre-conditions and post-conditions can be formalized in a single shape graph
- design of an abstract domain for relations between states
Bibliography