Memory abstraction 1

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

Xavier Rival

INRIA, ENS, CNRS

Oct, 16th. 2018
Overview of the lecture

So far, we have shown **numerical abstract domains**
- non relational: intervals, congruences...
- relational: polyhedra, octagons, ellipsoids...

- How to deal with non purely numerical states ?
- How to reason about complex data-structures ?

⇒ a very broad topic, and two lectures:

This lecture
- overview memory models and memory properties
- abstraction of pointer structures and separation logic based shape analysis

Next lecture: arrays, shape/numerical abstraction, composition of shape abstractions
Outline

1. Memory models
   - Towards memory properties
   - Formalizing concrete memory states
   - Treatment of errors
   - Language semantics

2. Pointer Abstractions

3. Separation Logic

4. A shape abstract domain relying on separation

5. Standard static analysis algorithms

6. Conclusion

7. Internships
Assumptions for the two lectures

Imperative programs viewed as *transition systems*:

- set of **control states**: $L$ (program points)
- set of **variables**: $X$ (all assumed globals)
- set of **values**: $V$ (so far: $V$ consists of integers (or floats) only)
- set of **memory states**: $M$ (so far: $M = X \rightarrow V$)
- **error state**: $\Omega$

**states**: $S$

$$S = L \times M$$
$$S_\Omega = S \cup \{\Omega\}$$

**transition relation**:

$$(\rightarrow) \subseteq S \times S_\Omega$$

**Abstraction** of sets of states

- **abstract domain** $\mathbb{D}^\#$
- **concretization** $\gamma : (\mathbb{D}^\#, \sqsubseteq^\#) \rightarrow (\mathcal{P}(S), \subseteq)$
Assumptions: syntax of programs

We start from the same language syntax and will extend l-values:

\[
\begin{align*}
1 & ::= \text{l-values} \\
        & | x \quad (x \in X) \\
        & | \ldots \text{ we will add other kinds of l-values} \\
        & | \ldots \text{ pointers, array dereference...} \\

\text{e} & ::= \text{expressions} \\
        & | c \quad (c \in V) \\
        & | l \quad (lvalue) \\
        & | e \oplus e \quad (\text{arith operation, comparison}) \\

\text{s} & ::= \text{statements} \\
        & | l = e \quad (\text{assignment}) \\
        & | s; \ldots s; \quad (\text{sequence}) \\
        & | \text{if}(e)\{s\} \quad (\text{condition}) \\
        & | \text{while}(e)\{s\} \quad (\text{loop})
\end{align*}
\]
Assumptions: semantics of programs

We assume **classical definitions for**:

- **l-values**: \([1] : M \rightarrow X\)
- **expressions**: \([e] : M \rightarrow V\)
- **programs and statements**:
  - we assume a label **before each statement**
  - each statement defines a **set of transition** \((\rightarrow)\)

In this course, we rely on the usual **reachable states semantics**

**reachable states semantics**

The reachable states are computed as \([S]_R = \text{lfp}F\) where

\[
F : \mathcal{P}(S) \quad \longrightarrow \quad \mathcal{P}(S) \\
X \quad \longmapsto \quad S_I \cup \{s \in S \mid \exists s' \in X, s' \rightarrow s\}
\]

and \(S_I\) denotes the set of initial states.
Assumptions: general form of the abstraction

We assume an abstraction for sets of memory states:

- memory abstract domain $D_{\text{mem}}$
- concretization function $\gamma_{\text{mem}} : D_{\text{mem}} \rightarrow \mathcal{P}(M)$

Reachable states abstraction

We construct $D^\# = L \rightarrow D_{\text{mem}}$ and:

$$\gamma : D^\# \rightarrow \mathcal{P}(S)$$

$$X^\# \mapsto \{(l, m) \in S \mid m \in \gamma_{\text{mem}}(X^\#(l))\}$$

The whole question is how do we choose $D_{\text{mem}}, \gamma_{\text{mem}}$...

- previous lectures:
  - $X$ is fixed and finite and, $V$ is scalars (integers or floats), thus, $M \equiv V^n$
- today:
  - we will extend the language thus, also need to extend $D_{\text{mem}}, \gamma_{\text{mem}}$
Abstraction of purely numeric memory states

Purely numeric case

- $\mathbb{V}$ is a set of values of the same kind
- e.g., integers ($\mathbb{Z}$), machine integers ($\mathbb{Z} \cap [-2^{63}, 2^{63} - 1]$)...
- If the set of variables is fixed, we can use any abstraction for $\mathbb{V}^N$

Example: $N = 2$, $\mathbb{X} = \{x, y\}$

- concrete set
- interval domain
- octagon domain
- polyedra domain
Heterogeneous memory states

In real life languages, there are many kinds of values:

- **scalars** (integers of various sizes, boolean, floating-point values)...
- **pointers, arrays**...

Heterogeneous memory states and non relational abstraction

- **types** $t_0, t_1, \ldots$ and **values** $\mathbb{V} = \mathbb{V}_{t_0} \cup \mathbb{V}_{t_1} \cup \ldots$
- finitely many **variables**; each has a **fixed type**: $\mathbb{X} = \mathbb{X}_{t_0} \cup \mathbb{X}_{t_1} \cup \ldots$
- **memory states**: $\mathbb{M} = \mathbb{X}_{t_0} \rightarrow \mathbb{V}_{t_0} \times \mathbb{X}_{t_1} \rightarrow \mathbb{V}_{t_1} \ldots$

**Principle:** compose abstractions for sets of memory states of each type

Non relational abstraction of heterogeneous memory states

- $\mathbb{M} \equiv \mathbb{M}_0 \times \mathbb{M}_1 \times \ldots$ where $\mathbb{M}_i = \mathbb{X}_i \rightarrow \mathbb{V}_i$
- **Concretization function** (case with two types)

\[
\gamma_{nr} : \mathcal{P}(\mathbb{M}_0) \times \mathcal{P}(\mathbb{M}_1) \rightarrow \mathcal{P}(\mathbb{M}) \\
(m_0^\#, m_1^\#) \mapsto \{(m_0, m_1) | \forall i, m_i \in \gamma_i(m_i^\#)\}
\]
Memory structures

Common structures (non exhaustive list)

- **Structures, records, tuples**: sequences of cells accessed with fields
- **Arrays**: similar to structures; indexes are integers in \([0, n - 1]\)
- **Pointers**: numerical values corresponding to the address of a memory cell
- **Strings and buffers**: blocks with a sequence of elements and a terminating element (e.g., 0x0)
- **Closures** (functional languages): pointer to function code and (partial) list of arguments

To describe memories, the definition \(M = X \rightarrow \mathbb{V}\) is too restrictive

Generally, non relational, heterogeneous abstraction cannot handle many such structures all at once: relations are needed!
Specific properties to verify

Memory safety

Absence of memory errors (crashes, or undefined behaviors)

Pointer errors:
- Dereference of a null pointer / of an invalid pointer

Access errors:
- Out of bounds array access, buffer overruns (often used for attacks)

Invariance properties

Data should not become corrupted (values or structures...)

Examples:
- Preservation of structures, e.g., lists should remain connected
- Preservation of invariants, e.g., of balanced trees
Properties to verify: examples

A program closing a list of file descriptors

```c
// l points to a list
c = l;
while (c != NULL){
    close(c -> FD);
    c = c -> next;
}
```

Correctness properties

1. memory safety
2. l is supposed to store all file descriptors at all times
   will its structure be preserved? yes, no breakage of a next link
3. closure of all the descriptors

Examples of structure preservation properties

- Algorithms manipulating trees, lists...
- Libraries of algorithms on balanced trees
- Not guaranteed by the language!
  e.g., the balancing of Maps in the OCaml standard library was incorrect for years (performance bug)
A more realistic model

No one-to-one relation between memory cells and program variables
- a variable may indirectly reference several cells (structures...)
- dynamically allocated cells correspond to no variable at all...

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

$$\mathbb{E} = X \rightarrow \mathbb{V}_{\text{addr}}$$
$$\mathbb{H} = \mathbb{V}_{\text{addr}} \rightarrow \mathbb{V}$$

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

Note: Avoid confusion between heap (function from addresses to values) and dynamic allocation space (often referred to as “heap”)

Xavier Rival (INRIA, ENS, CNRS)
Example of a concrete memory state (variables)

Example setup:
- \( x \) 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)

\[
\begin{array}{c|c|c|c}
\text{address} & \&x = 300 & 64 & 304 \\
& 300 & 312 & \&y = 308 & 312 \\
& 304 & 308 & \&z = 312 & 88 \\
& 312 & 312 & & 0x0 \\
\end{array}
\]

\[
e : \quad x \mapsto 300 \\
y \mapsto 308 \\
z \mapsto 312
\]

\[
h : \quad 300 \mapsto 64 \\
304 \mapsto 312 \\
308 \mapsto 312 \\
312 \mapsto 88 \\
316 \mapsto 0
\]
Example of a concrete memory state (variables + dyn. cell)

**Example setup:**
- same configuration
- $z$ points to a dynamically allocated list element (in purple)

**Memory layout**

<table>
<thead>
<tr>
<th>address</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;x = 300</td>
<td>64</td>
<td>312</td>
<td>312</td>
</tr>
<tr>
<td>304</td>
<td>312</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;y = 308</td>
<td>312</td>
<td></td>
<td></td>
</tr>
<tr>
<td>308</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;z = 312</td>
<td>88</td>
<td>508</td>
<td>88</td>
</tr>
<tr>
<td>312</td>
<td>508</td>
<td></td>
<td>508</td>
</tr>
<tr>
<td></td>
<td></td>
<td>316</td>
<td>512</td>
</tr>
<tr>
<td>508</td>
<td>25</td>
<td></td>
<td>508</td>
</tr>
<tr>
<td></td>
<td></td>
<td>512</td>
<td>0</td>
</tr>
</tbody>
</table>

$e: x \mapsto 300$
$e: y \mapsto 308$
$e: z \mapsto 312$

$h: 300 \mapsto 64$
$h: 304 \mapsto 312$
$h: 308 \mapsto 312$
$h: 312 \mapsto 88$
$h: 316 \mapsto 508$
$h: 508 \mapsto 25$
$h: 512 \mapsto 0$
Extending the semantic domains

Some slight modifications to the semantics of the initial language:

- **Addresses are values:** \( \mathbb{V}_{addr} \subseteq \mathbb{V} \)

- **L-values evaluate into addresses:** \( [1] : \mathbb{M} \rightarrow \mathbb{V}_{addr} \)

\[
[\mathbf{x}](e, h) = e(x)
\]

- **Semantics of expressions** \([e] : \mathbb{M} \rightarrow \mathbb{V} \), mostly unchanged

\[
[1](e, h) = h([1](e, h))
\]

- **Semantics of assignment** \( l_0 : l := e; l_1 : \ldots : \)

\[
(l_0, e, h_0) \rightarrow (l_1, e, h_1)
\]

where

\[
h_1 = h_0[1](e, h_0) \leftarrow [e](e, h_0)
\]
Realistic definitions of memory states

Our model is still not very accurate for most languages

- Memory cells do not all have the same size
- **Memory management algorithms** usually do not treat cells one by one, e.g., `malloc` returns a pointer to a **block**, applying `free` to that pointer will dispose the whole block

Other refined models

- **Partition of the memory** in **blocks** with a **base address** and a size
- **Partition of blocks** into **cells** with a size
- Description of **fields** with an offset
- Description of **pointer values** with a **base address** and an offset...

For a very formal description of such concrete memory states: see **CompCert** project source files (Coq formalization)
Language semantics: program crash

In an abnormal situation, we assume that the program will crash

- advantage: very clear semantics
- disadvantage (for the compiler designer): dynamic checks are required

Error state

- $\Omega$ denotes an error configuration
- $\Omega$ is a blocking: $(\rightarrow) \subseteq S \times (\{\Omega\} \cup S)$

OCaml:

- out-of-bound array access:
  
  ```
  Exception: Invalid_argument "index out of bounds".
  ```
- no notion of a null pointer

Java:

- exception in case of out-of-bound array access, null dereference:
  ```
  java.lang.ArrayIndexOutOfBoundsException
  ```
Language semantics: undefined behaviors

**Alternate choice:** leave the behavior of the program **unspecified** when an abnormal situation is encountered

- advantage: **easy implementation** (often architecture driven)
- disadvantage: **unintuitive semantics**, errors hard to reproduce different compilers may make different choices... or in fact, make no choice at all (= let the program evaluate even when performing invalid actions)

Modeling of undefined behavior

- Very hard to capture what a program operation may modify
- Abnormal situation at \((l_0, m_0)\) such that \(\forall m_1 \in \mathbb{M}, (l_0, m_0) \rightarrow (l_1, m_1)\)

**In C:**
array out-of-bound accesses and dangling pointer dereferences lead to undefined behavior (and potentially, memory corruption) whereas a null-pointer dereference always result into a crash
Composite objects

How are contiguous blocks of information organized?

Java objects, OCaml struct types
- sets of fields
- each field has a type
- **no assumption** on physical storage, **no pointer arithmetics**

C composite structures and unions
- **physical mapping** defined by the norm
- each field has a specified **size** and a specified **alignment**
- **union types / casts:**
  implementations may allow several views
Many languages provide pointers or references and allow to manipulate addresses, but with different levels of expressiveness.

What kind of objects can be referred to by a pointer?

Pointers only to records / structures / objects

- **Java**: only pointers to objects
- **OCaml**: only pointers to records, structures...

Pointers to fields

- **C**: pointers to any valid cell...

```c
struct {int a; int b} x;
int * y = &(x.b);
```
What kind of operations can be performed on a pointer?

Classical pointer operations

- **Pointer dereference**: 
  \( *p \) returns the contents of the cell of address \( p \)
- **“Address of”** operator: \( &x \) returns the address of variable \( x \)
- Can be analyzed with a **rather coarse pointer model**
  e.g., symbolic base + symbolic field

Arithmetics on pointers, requiring a more precise model

- **Addition of a numeric constant**: 
  \( p + n \): address contained in \( p + n \) times the size of the type of \( p \)
  Interaction with pointer casts...
- **Pointer subtraction**: returns a numeric offset
Manual memory management

Allocation of unbounded memory space
- How are new memory blocks **created** by the program?
- How do old memory blocks get **freed**?

OCaml memory management
- **implicit allocation**: when declaring a new object
- **garbage collection**: purely automatic process, that frees unreachable blocks

C memory management
- **manual allocation**: `malloc` operation returns a pointer to a new block
- **manual de-allocation**: `free` operation (block base address)

Manual memory management is not safe:
- **memory leaks**: growing unreachable memory region; memory exhaustion
- **dangling pointers** if freeing a block that is still referred to
Summary on the memory model

Language dependent items

- **Clear error cases** or **undefined behaviors**
  for analysis, a semantics with clear error cases is preferable

- **Composite objects**: structure fully exposed or not

- **Pointers to objct fields**: allowed or not

- **Pointer arithmetic**: allowed or not
  *i.e.*, are pointer values symbolic values or numeric values

- **Memory management**: automatic or manual

In this course, we start with a simple model, and study specific features one by one and in isolation from the others
Rest of the course

Structures for which we introduce abstractions:

- **arrays** (in a few weeks)
- **pointers** and **dynamically allocated pointer structures** (today)

Abstract operations:

- post-condition for the **reading** of a cell defined by an l-value
  - e.g., \( x = a[i] \) or \( x = *p \)
- post-condition for the **writing of a heap cell**
  - e.g., \( a[i] = p \) or \( p -> f = x \)
- **abstract join**, that approximates unions of concrete states
Outline

1 Memory models
2 Pointer Abstractions
3 Separation Logic
4 A shape abstract domain relying on separation
5 Standard static analysis algorithms
6 Conclusion
7 Internships
We do not consider **pointer arithmetics** here.
Programs with pointers: semantics

Case of l-values:

\[
\llbracket x \rrbracket (e, h) = e(x) \\
\llbracket *e \rrbracket (e, h) = \begin{cases} 
    h(\llbracket e \rrbracket (e, h)) & \text{if } \llbracket e \rrbracket (e, h) \neq 0 \land \llbracket e \rrbracket (e, h) \in \text{Dom}(h) \\
    \Omega & \text{otherwise}
\end{cases}
\]

\[
\llbracket 1 \cdot f \rrbracket (e, h) = \llbracket 1 \rrbracket (e, h) + \text{offset}(f) \quad \text{(numeric offset)}
\]

Case of expressions:

\[
\llbracket 1 \rrbracket (e, h) = h(\llbracket 1 \rrbracket (e, h)) \quad \text{(evaluates into the contents)}
\]

\[
\llbracket &1 \rrbracket (e, h) = \llbracket 1 \rrbracket (e, h) \quad \text{(evaluates into the address)}
\]

Case of statements:

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

- **memory deallocation** $\text{free}(x)$: $(e, h) \rightarrow (e, h')$ where $k = e(x)$ and
  \[
  h = h' \uplus \{ k \mapsto v_k, k + 1 \mapsto v_{k+1}, \ldots, k + c - 1 \mapsto v_{k+c-1} \}$
We rely on the non relational abstraction of heterogeneous states that was introduced earlier, with a few changes:

- we let $\mathbb{V} = \mathbb{V}_{\text{addr}} \cup \mathbb{V}_{\text{int}}$ and $X = X_{\text{addr}} \cup X_{\text{int}}$
- concrete memory cells now include structure fields, and fields of dynamically allocated regions
- abstract cells $C^\#$ finitely summarize concrete cells
- we apply a non relational abstraction:

Non relational pointer abstraction

- Set of pointer abstract values $D^\#_{\text{ptr}}$
- Concretization $\gamma_{\text{ptr}} : D^\#_{\text{ptr}} \rightarrow \mathcal{P}(\mathbb{V}_{\text{addr}})$ into pointer sets

We will see several instances of this kind of abstraction
Pointer non relational abstraction: null pointers

The dereference of a null pointer will cause a crash

To establish safety: compute which pointers may be null

Null pointer analysis

Abstract domain for addresses:

\[ \gamma_{\text{ptr}}(\bot) = \emptyset \]
\[ \gamma_{\text{ptr}}(\top) = V_{\text{addr}} \]
\[ \gamma_{\text{ptr}}(\neq \text{NULL}) = V_{\text{addr}} \setminus \{0\} \]

- we may also use a lattice with a fourth element = NULL
  - exercise: what do we gain using this lattice?
- very lightweight, can typically resolve rather trivial cases
- useful for C, but also for Java
Pointer non relational abstraction: dangling pointers

The dereference of a null pointer will cause a crash

To establish **safety**: compute **which pointers may be dangling**

**Null pointer analysis**

**Abstract domain for addresses:**

- $\gamma_{\text{ptr}}(\bot) = \emptyset$
- $\gamma_{\text{ptr}}(\top) = \mathbb{V}_{\text{addr}} \times \mathbb{H}$
- $\gamma_{\text{ptr}}(\text{Not dangling}) = \{(v, h) \mid h \in \mathbb{H} \land v \in \text{Dom}(h)\}$

- **very lightweight**, can typically resolve rather trivial cases
- useful for C, useless for Java (initialization requirement + GC)
Determine where a pointer may store a reference to

1: int x, y;
2: int * p;
3: y = 9;
4: p = &x;
5: *p = 0;

- what is the final value for x? 0, since it is modified at line 5...
- what is the final value for y? 9, since it is not modified at line 5...

Basic pointer abstraction

- We assume a set of abstract memory locations $A^\#$ is fixed:

$$A^\# = \{&x, &y, \ldots, &t, a_0, a_1, \ldots, a_N\}$$

- Concrete addresses are abstracted into $A^\#$ by $\phi_A : A \rightarrow A^\# \cup \{\top\}$

- A pointer value is abstracted by the abstraction of the addresses it may point to, i.e., $D^\#_{ptr} = \mathcal{P}(A^\#)$

and $\gamma_{ptr}(a^\#) = \{a \in A | \phi_A(a) = a^\#\}$

- example: p may point to $\{&x\}$
Points-to sets computation example

Example code:

```c
int x, y;
int * p;
y = 9;
p = &x;
*p = 0;
...
```

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

Analysis results:

<table>
<thead>
<tr>
<th></th>
<th>&amp;x</th>
<th>&amp;y</th>
<th>&amp;p</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>[9, 9]</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>[9, 9]</td>
<td>{&amp;x}</td>
</tr>
<tr>
<td>6</td>
<td>[0, 0]</td>
<td>[9, 9]</td>
<td>{&amp;x}</td>
</tr>
</tbody>
</table>
Points-to sets computation and imprecision

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

- 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
Weak-updates

We can formalize this imprecision a bit more:

Weak updates

- The modified concrete cell cannot be mapped into a well identified abstract cell
- The resulting abstract information is obtained by joining the new value and the old information

Effect in pointer analysis, in the case of an assignment:

- if the points-to set contains exactly one element, the analysis can perform a strong update as in the first example: \( p \mapsto \{ \&x \} \)
- if the points-to set may contain more than one element, the analysis needs to perform a weak-update as in the second example: \( p \mapsto \{ \&x, \&y \} \)
Pointer aliasing based on equivalence on access paths

Aliasing relation

Given $m = (e, h)$, pointers $p$ and $q$ are aliases iff $h(e(p)) = h(e(q))$

Abstraction to infer pointer aliasing properties

- An access path describes a sequence of dereferences to resolve an l-value (i.e., an address); e.g.:
  \[ a ::= x \mid a \cdot f \mid *a \]

- An abstraction for aliasing is an over-approximation for equivalence relations over access paths

Examples of aliasing abstractions:

- set abstractions: map from access paths to their equivalence class
  (ex: $\{\{p_0, p_1, &x\}, \{p_2, p_3\}, \ldots\}$)

- numerical relations, to describe aliasing among paths of the form $x(-n)^k$
  (ex: $\{x(-n)^k, \&(x(-n)^{k+1}) \mid k \in \mathbb{N}\}$)
Limitation of basic pointer analyses seen so far

**Weak updates:**
- **imprecision in updates** that spread out as soon as points-to set contain several elements
- impact **client analyses** severely (e.g., low precision on numerical)

**Unsatisfactory abstraction of unbounded memory:**
- common assumption that $\mathbb{C}^\#$ be finite
- programs using **dynamic allocations** often perform **unbounded** numbers of **malloc** calls (e.g., allocation of a list)

**Unable to express well structural invariants:**
- for instance, that a structure should be a **list**, a **tree**...
- **very indirect** abstraction in numeric / path equivalence abstraction

**A common solution:**
**shape abstraction**
Outline

1. Memory models
2. Pointer Abstractions
3. Separation Logic
4. A shape abstract domain relying on separation
5. Standard static analysis algorithms
6. Conclusion
7. Internships
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 $\ast$

- **Local reasoning**: powerful principle, which allows to consider only part of the memory

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

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

Pure formulas denote value properties

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

\[
\begin{align*}
  P & ::= e_0 = e_1 \mid P \lor P'' \mid P' \land P'' \ldots & \text{pure predicates}
\end{align*}
\]

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

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

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

**Points-to predicate**

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

- **Concretization:**
  
  \[(e, h) \in \gamma(1 \mapsto v) \iff h = \llbracket 1 \rrbracket(e, h) \mapsto v\]

- **Example:**
  
  \[ F = \&x \mapsto 18 \quad \&x = 308 \quad 18 \]

- We also note $1 \mapsto e$, as an l-value $1$ denotes an address
Separation logic: separating conjunction

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

$$h_0 \diamondsuit h_1 : \quad \text{dom}(h_0) \cup \text{dom}(h_1) \quad \longrightarrow \quad \mathbb{V}$$

$$x \in \text{dom}(h_0) \quad \longmapsto \quad h_0(x)$$

$$x \in \text{dom}(h_1) \quad \longmapsto \quad h_1(x)$$

---

**Separating conjunction**

- **Predicate:**

  $$F ::= \ldots \mid F_0 \ast F_1$$

- **Concretization:**

  $$\gamma(F_0 \ast F_1) = \{(e, h_0 \diamondsuit h_1) \mid (e, h_0) \in \gamma(F_0) \land (e, h_1) \in \gamma(F_1)\}$$
An example

Concrete memory layout
(pointer values underlined)

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

A formula that abstracts away the addresses:

\[
&x \mapsto \langle 64, \&z \rangle \ast \&y \mapsto \&z \ast \&z \mapsto \langle 88, 0 \rangle
\]
Separation logic: non separating conjunction

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

**Non separating conjunction**

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

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

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

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

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

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

\[
\&a \leftrightarrow \&b \ast \&b \leftrightarrow \&a
\]
- two separate sub-heaps respectively satisfy \&a \leftrightarrow \&b and \&b \leftrightarrow \&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:**

\[
(e, h_0) \in \gamma(F_0) \quad (e, h_1) \in \gamma(F_1) \\
(e, h_0 \otimes h_1) \in \gamma(F_0 \ast F_1)
\]

- **Non separating conjunction:**

\[
(e, h) \in \gamma(F_0) \quad (e, h) \in \gamma(F_1) \\
(e, h) \in \gamma(F_0 \land F_1)
\]

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

Empty store

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

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

  where [] denotes the empty store

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

**Disjunction:**

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

\[ \gamma(F_0 \lor F_1) = \gamma(F_0) \cup \gamma(F_1) \]

**Spatial implication (aka, magic wand):**

- $F ::= \ldots | F_0 \rightarrow^* F_1$
- concretization:

\[ \gamma(F_0 \rightarrow^* F_1) = \{ (e, h) \mid \forall h_0 \in \mathbb{H}, (e, h_0) \in \gamma(F_0) \implies (e, h \otimes h_0) \in \gamma(F_1) \} \]

- very powerful connector to describe **structure segments**, used in complex SL proofs
Summary of the main separation logic constructions seen so far:

**Separation logic main connectors**

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

Concretization of pure formulas is standard

How does this help for program reasoning?
Separation logic triple

Program proofs based on Hoare triples

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

- **Application**: formalize proofs of programs

A few rules (straightforward proofs):

\[
\frac{F_0 \implies F_0'}{F_0 \implies F_0'} \quad \frac{\{F_0\} b \{F_1\}}{F_0' \implies F_1'} \quad \frac{F_1' \implies F_1}{\{F_0\} b \{F_1\}} \quad \text{(consequence)}
\]

\[
\frac{\{\&x \mapsto ?\} x := e \{\&x \mapsto e\}}{x \text{ does not appear in } F} \quad \frac{\{\&x \mapsto ? \ast F\} x := e \{\&x \mapsto e \ast F\}}{x \text{ does not appear in } F} \quad \text{(mutation)}
\]

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

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

**Theorem: the frame rule**

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

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

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

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

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

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

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

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

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

List definition

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

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

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

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

Practical implementation in verification/analysis tools

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

Intuitive semantics of inductive predicates

- Inductive predicates can be **unfolded**, by **unrolling their definitions**
  
  Syntactic unfolding is noted $\mathcal{U}$

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

**Example:**

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

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

- We get the concrete state below:
Example: tree

Inductive definition

Two recursive calls instead of one:

\[
\alpha \cdot \text{tree} \; := \; \begin{cases} 
\alpha = 0 \land \text{emp} \\
\lor \; \alpha \neq 0 \land \alpha \cdot \text{left} \leftrightarrow \beta \ast \alpha \cdot \text{right} \leftrightarrow \delta \\
\ast \; \beta \cdot \text{tree} \ast \delta \cdot \text{tree}
\end{cases}
\]
Example: doubly linked list

Inductive definition

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

\[
\alpha \cdot \text{dll}(\delta) := \begin{cases} 
\alpha = 0 \land \text{emp} \\
\lor \alpha \neq 0 \land \alpha \cdot \text{next} \leftrightarrow \beta \land \alpha \cdot \text{prev} \leftrightarrow \delta \land *\beta \cdot \text{dll}(\alpha)
\end{cases}
\]
Example: sortedness

Example: sorted list

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

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

\[
\begin{align*}
\alpha \cdot \text{lsort}() & := \\
& \quad \alpha \cdot \text{lsort}_{\text{aux}}(-\infty)
\end{align*}
\]
Outline

1. Memory models
2. Pointer Abstractions
3. Separation Logic
4. A shape abstract domain relying on separation
5. Standard static analysis algorithms
6. Conclusion
7. Internships
A shape abstract domain relying on separation

Design of an abstract domain

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

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

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

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

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

\[
\begin{align*}
&(x \cdot n) = 0x...a0 & 17 \\
&(x \cdot d) = 0x...a4 & 0x...b0 \\
&(y \cdot n) = 0x...b0 & 17 \\
&(y \cdot d) = 0x...b4 & 0x0
\end{align*}
\]
Basic abstraction: structures and their contents (1/2)

- **Concrete memory states**

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

  ▶ characterized by valuation $\nu$
  ▶ $\nu$ maps **symbolic variables** into **concrete addresses**
Basic abstraction: structures and their contents (1/2)

- Concrete memory states

- Abstraction of values into symbolic variables / nodes

- Abstraction of regions into points-to edges

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

- **Concrete memory states**

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

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

- **Shape graph concretization**

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

  Valuation \(\nu\) plays an important role to combine abstraction...
Structure of shape graphs

Valuations bridge the gap between nodes and values

Symbolic variables / nodes and intuitively abstract concrete values:

Symbolic variables

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

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

Valuations

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

Note that valuations treat **in the same way addresses** and **raw values**
Structure of shape graphs

Distinct edges describe separate regions

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

**Separating conjunction**

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

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

\[
\gamma_{\text{sh}}(\text{emp}) = \{(\emptyset, \nu) \mid \nu : \mathcal{V}^\# \rightarrow \mathcal{V}\}
\]
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**

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

  **Concretization:**

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

Contiguous regions are described by adjacent points-to edges

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

Field splitting model

- Separation impacts edges / fields, not pointers

\[
\begin{align*}
\nu(\alpha) & \xrightarrow{\text{offset}(f)} \nu(\beta_0) \\
\nu(\alpha) & \xrightarrow{\text{offset}(g)} \nu(\beta_1)
\end{align*}
\]

Shape graph accounts for both abstract states below:

\[
\begin{align*}
\nu(\beta_0) & = \nu(\beta_1)
\end{align*}
\]

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

\[ \&x = \& (x \cdot n) = 0x...a0 \]
\[ \& (x \cdot d) = 0x...a4 \]

\[ \&y = \& (y \cdot n) = 0x...b0 \]
\[ \& (y \cdot d) = 0x...b4 \]

Abstract environments

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

\[
\gamma_{\text{mem}}(e^\#, S^\#) = \{ (e, h, \nu) \mid (h, \nu) \in \gamma_{\text{sh}}(S^\#) \land e = \nu \circ e^\# \}
\]
A shape abstract domain relying on separation

Basic abstraction: summarization

Set of all lists of any length:

\&x 0x0 \&x 0x... \&x 0x...

\ldots \ldots \ldots

0x0 0x0 0x0

Well-founded list inductive def.

\[ \alpha \cdot \text{list} := \]

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

well-founded predicate

Inductive summary predicates

\&x \alpha \beta \text{list}

Concretization based on unfolding and least-fixpoint:

\[ \U \text{ replaces an } \alpha \cdot \text{list predicate with one of its premises} \]

\[ \gamma(S^#, F) = \bigcup \{ \gamma(S'^#, F_u) \mid (S^#, F) \xrightarrow{\U} (S'^#, F_u) \} \]
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
A shape abstract domain relying on separation

Inductive segments

A frequent pattern:

\[ \&x \rightarrow \alpha \cdot \alpha \cdot \text{list} \]
\[ \&y \rightarrow \beta \cdot \beta \cdot \text{list} \]

A first attempt:

- \( x \) points to a list, so \( \&x \rightarrow \alpha \cdot \alpha \cdot \text{list} \) holds
- \( y \) points to a list, so \( \&y \rightarrow \beta \cdot \beta \cdot \text{list} \) holds

However, the following does not hold

\[ \&x \rightarrow \alpha \cdot \alpha \cdot \text{list} \ast \&y \rightarrow \beta \cdot \beta \cdot \text{list} \]

Why? violation of separation!

A second attempt:

\[ (\&x \rightarrow \alpha \cdot \alpha \cdot \text{list} \ast \text{TRUE}) \land (\&y \rightarrow \beta \cdot \beta \cdot \text{list} \ast \text{TRUE}) \]

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

A frequent pattern:

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

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

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

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

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

<table>
<thead>
<tr>
<th>Graph $S^# \in D^#_{sh}$</th>
<th>Translated formula $\Pi(S^#)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha \xrightarrow{f} \beta$</td>
<td>$\alpha \cdot f \leftrightarrow \beta$</td>
</tr>
<tr>
<td>$S^#_0$</td>
<td>$\Pi(S^#_0) \ast \Pi(S^#_1)$</td>
</tr>
<tr>
<td>$\alpha \xrightarrow{\text{list}} \delta$</td>
<td>$\alpha \cdot \text{list}_\text{endp}(\delta)$</td>
</tr>
<tr>
<td>other inductives and segments</td>
<td>similar</td>
</tr>
</tbody>
</table>

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

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

1. Memory models
2. Pointer Abstractions
3. Separation Logic
4. A shape abstract domain relying on separation
5. **Standard static analysis algorithms**
   - Overview of the analysis
   - Post-conditions and unfolding
   - Folding: widening and inclusion checking
   - Abstract interpretation framework: assumptions and results
6. Conclusion
7. Internships
A list insertion function:

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

- list inductive structure def.
- Abstract precondition:

Result of the (interprocedural) analysis

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

Abstract interpreter design

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

Transfer functions:

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

Abstract operators:

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

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

Denotational style abstract interpreter

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

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

- $[[b_0; b_1]]^\#(S) = [[b_1]]^\# \circ [[b_0]]^\#(S)$
- $[[1 = e]]^\#(S) = \text{assign}(1, e, S)$
- $[[1 = \text{malloc}(n)]]^\#(S) = \text{alloc}(1, n, S)$
- $[[\text{free}(1)]]^\#(S) = \text{free}(1, n, S)$
- $[[\text{if}(e) b_t \text{ else } b_f]]^\#(S) = \left\{ \begin{array}{l} \text{join}([[b_t]]^\#(\text{test}(e, S))), \\ [[b_f]]^\#(\text{test}(e = \text{false}, S))) \end{array} \right.$
- $[[\text{while}(e)b]]^\#(S) = \text{test}(e = \text{false}, \text{lfp}_S F^\#)$
  
  where, $F^\# : S_0 \mapsto [[b]]^\#(\text{test}(e, S_0))$
The algorithms underlying the transfer functions

- **Unfolding**: cases analysis on summaries

  \[ \begin{array}{c}
  x \text{ list} \rightarrow \text{y list} \\
  \Rightarrow \\
  x \text{ list} \rightarrow \text{y next list}
  \end{array} \]

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

  \[ \begin{array}{c}
  x \text{ list} \rightarrow \text{y next list} \\
  \Rightarrow \\
  x \text{ list} \rightarrow \text{y next list}
  \end{array} \]

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

  \[ \begin{array}{c}
  x \text{ list} \rightarrow \text{y list} \\
  \nabla \\
  \Rightarrow \\
  x \text{ list} \rightarrow \text{y next list}
  \end{array} \]
Outline

1 Memory models
2 Pointer Abstractions
3 Separation Logic
4 A shape abstract domain relying on separation
5 Standard static analysis algorithms
   - Overview of the analysis
   - Post-conditions and unfolding
   - Folding: widening and inclusion checking
   - Abstract interpretation framework: assumptions and results
6 Conclusion
7 Internships
Analysis of an assignment in the graph domain

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

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

With pre-condition:

\[
\begin{align*}
\&x \alpha_0 & \rightarrow \beta_0 \\
\&y \alpha_1 & \rightarrow \beta_1 \rightarrow \beta_2
\end{align*}
\]

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

\[
\begin{align*}
\&x \alpha_0 & \rightarrow \beta_0 \\
\&y \alpha_1 & \rightarrow \beta_1 \rightarrow \beta_2
\end{align*}
\]

With pre-condition:

\[
\begin{align*}
\&x \alpha_0 & \rightarrow \beta_0 \\
\&y \alpha_1 & \rightarrow \beta_1
\end{align*}
\]

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

Unfolding principle

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

Example, for lists:

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

Soundness: by definition of the concretization of inductive structures

\[ \gamma_{sh}(S^\#) \subseteq \bigcup \{ \gamma_{sh}(S_{0}^\#) \mid S^\# \xrightarrow{U} S_{0}^\# \} \]
Analysis of an assignment, with unfolding

**Principle**

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

**Disjunct 1:**

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

**Disjunct 2:**

- Step 1 produces $\alpha_0 \mapsto \beta_0$
- Step 2 produces $\beta_2$
- **End result:**
Unfolding and degenerated cases

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

\( c = l; \)

1. while (\( c \neq \text{NULL} \&\& \text{condition} \))
   \( c = c \rightarrow \text{next}; \)
2. if (\( c \neq 0 \&\& c \rightarrow \text{prev} \neq 0 \))
   \( c = c \rightarrow \text{prev} \rightarrow \text{prev}; \)

\[ \Rightarrow \text{non trivial unfolding} \]

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

Segment splitting lemma: basis for segment unfolding

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

\[ \alpha_0 \xrightarrow{i} i'' \xrightarrow{j} \alpha_2 \]

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

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

1. Memory models
2. Pointer Abstractions
3. Separation Logic
4. A shape abstract domain relying on separation
5. Standard static analysis algorithms
   - Overview of the analysis
   - Post-conditions and unfolding
   - Folding: widening and inclusion checking
   - Abstract interpretation framework: assumptions and results
6. Conclusion
7. Internships
Need for a folding operation

Back to the **list traversal** example:

**First iterates** in the loop:
- at **iteration 0** (before entering the loop):
  
  \[
  \begin{align*}
  l, c & \rightarrow \alpha_0 \rightarrow \text{list} \\
  \end{align*}
  \]

- at **iteration 1**:
  
  \[
  \begin{align*}
  l_{\text{next}}, c & \rightarrow \text{list} \\
  \end{align*}
  \]

- at **iteration 2**:
  
  \[
  \begin{align*}
  l_{\text{next}}, c & \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \text{list} \\
  \end{align*}
  \]

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

\[
\begin{align*}
S_0 & \xrightarrow{\text{unfold}} S_{0,u} \\
S_{0,u} & \xrightarrow{f} S_1 \\
S_1 & \xrightarrow{\text{unfold}} S_{1,u} \\
S_{1,u} & \xrightarrow{f} S_2 \\
\end{align*}
\]

- 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 ▽**

- **Over-approximates join:**
  \[
  \begin{align*}
  \gamma(X^\#) & \subseteq \gamma(X^\# \downarrow Y^\#) \\
  \gamma(Y^\#) & \subseteq \gamma(X^\# \downarrow Y^\#)
  \end{align*}
  \]

- **Enforces termination:** for all sequence \((X_n^\#)_{n \in \mathbb{N}}\), the sequence \((Y_n^\#)_{n \in \mathbb{N}}\) defined below is ultimately stationary
  \[
  \begin{align*}
  Y_0^\# & = X_0^\# \\
  \forall n \in \mathbb{N}, \quad Y_{n+1}^\# & = Y_n^\# \downarrow X^\#_{n+1}
  \end{align*}
  \]
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 \( D^\#_{\text{can} \lor} \) has finite height

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

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

Weakening

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

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

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

Applications:

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

We consider $S_0^\#$ defined by:

$$
\begin{array}{c}
\alpha_0 \\
& x \\
\end{array} \rightarrow
\begin{array}{c}
\alpha_1 \\
\text{next} \\
\text{data} \\
\end{array} \rightarrow
\begin{array}{c}
\alpha_2 \\
\text{list} \\
\end{array}
$$

and $S_1^\#$ defined by:

$$
\begin{array}{c}
\alpha_0 \\
& x \\
\end{array} \rightarrow
\begin{array}{c}
\alpha_1 \\
\text{list} \\
\end{array}
$$

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

$$
\Psi : \begin{array}{c}
\beta_0 \\
\beta_1 \\
\end{array} \mapsto
\begin{array}{c}
\alpha_0 \\
\alpha_1 \\
\end{array}
$$

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

We can apply the local reasoning principle to weakening

If $S^\#_0 \preceq S^\#_{0, \text{weak}}$ and $S^\#_1 \preceq S^\#_{1, \text{weak}}$ then:

Separating conjunction rule ($\preceq^*$)

Let us assume that
- $S^\#_0$ and $S^\#_1$ have distinct set of source nodes
- we can weaken $S^\#_0$ into $S^\#_{0, \text{weak}}$
- we can weaken $S^\#_1$ into $S^\#_{1, \text{weak}}$

then:

we can weaken $S^\#_0 \ast S^\#_1$ into $S^\#_{0, \text{weak}} \ast S^\#_{1, \text{weak}}$
Local weakening: unfolding rule, identity rule

Weakening unfolded region ($\preceq_U$)

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

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

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

Identity weakening ($\preceq_{\text{Id}}$)

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

- the proof is trivial:

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

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

By rule $(\preceq_{\text{Id}})$:

Thus, by rule $(\preceq_{\text{U}})$:

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

Thus, by rule $(\preceq_{\text{*}})$:
Inclusion checking rules in the shape domain

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

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

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

- **Rules on inductives:**
  \[
  \forall i, \; \Psi(\beta_i) = \alpha \implies \text{emp} ^\# \psi \beta_0 \cdot \text{list}_\text{endp}(\beta_1)
  \]
  \[
  S_0 ^\# \sqsubseteq_\psi S_1^\# \land \beta \cdot \iota \xrightarrow{u} S_1^\# \implies S_0^\# 
  \]
  If \( \beta_1 \) fresh, \( \Psi' = \Psi[\beta_1 \mapsto \alpha_1] \) and \( \Psi(\beta_0) = \alpha_0 \) then,
  \[
  S_0^\# \sqsubseteq_\psi \beta_1 \cdot \text{list} \implies \alpha_0 \cdot \text{list}_\text{endp}(\alpha_1) \ast S_0^\# ^\# \psi \beta_0 \cdot \iota
  \]
Inclusion checking algorithm

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

1. start with \(\Psi\) defined by \(\Psi(\beta) = \alpha\) if and only if there exists a variable \(x\) such that \(e_0^\#(x) = \alpha \land e_1^\#(x) = \beta\)
2. iteratively apply local rules, and extend \(\Psi\) when needed
3. return \textbf{true} when both shape graphs become empty

- the first step ensures both environments are consistent

This algorithm is sound:

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

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

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

\[
\begin{align*}
S_{0,lft}^\# & \quad S_{1,lft}^\# \\
\beta_0 & \quad \psi & \quad \psi & \quad \psi \\
S_{0,rgh}^\# & \quad S_{1,rgh}^\#
\end{align*}
\]

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

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

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

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

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

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

  $$S_{\text{left}}^\# \triangleleft S_{\text{right}}^\# = \delta_0 \cdot f \mapsto \delta_1$$

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

  $$S_{\text{left}}^\# \triangleleft S_{\text{right}}^\# = \delta_0 \cdot \text{list}$$
Over-approximation of union: segment introduction rule

\[
\begin{align*}
\text{if } & \quad \Psi(S_{\text{lt}}^\#) \subseteq S_{\text{rgt}}^\# \quad \text{then} \\
& \quad S_{\text{lt}}^\# \nabla S_{\text{rgt}}^\# = \delta_0 \quad \delta_1
\end{align*}
\]

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

- **before iteration 0:**
  \[
  \begin{array}{c}
  \alpha_0 \\
  \text{list} \\
  \end{array}
  \begin{array}{c}
  \text{l, c} \\
  \end{array}
  \]

- **end of iteration 0:**
  \[
  \begin{array}{c}
  \beta_0 \\
  \text{next} \\
  \end{array}
  \begin{array}{c}
  \beta_1 \\
  \text{list} \\
  \end{array}
  \begin{array}{c}
  \text{l, c} \\
  \text{data} \\
  \beta_2 \\
  \end{array}
  \]

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

**Rule**

\[
\text{if } \quad S^\#_{\text{left}}(\beta_0) \subseteq S^\#_{\text{right}}(\alpha_0, \beta_0) \quad \text{then } \quad S^\#_{\text{left}} \vee S^\#_{\text{right}} = \delta_0 \ \text{list} \text{ } \delta_1 \\
\begin{align*}
(\alpha_0, \beta_0) & \leftrightarrow \delta_0 \\
(\alpha_1, \beta_1) & \leftrightarrow \delta_1
\end{align*}
\]

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

- previous invariant before iteration 1:

  \[
  \Psi(\alpha_0, \beta_0) = \delta_0 \\
  \Psi(\alpha_1, \beta_2) = \delta_1
  \]

- end of iteration 1:

- join, before iteration 1:
Over-approximation of union: rewrite system properties

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

**Termination**

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

**Non confluence!**

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

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

1. define \(\Psi, e\) by \(\Psi(\alpha, \beta) = e(x) = \delta\) (where \(\delta\) is a fresh node) if and only if \(e^\#_0(x) = \alpha \land e^\#_1(x) = \beta\)

2. iteratively apply join local rules, and extend \(\Psi\) when new relations are inferred (for instance for points-to edges)

3. return the result obtained when all regions of both inputs are approximated in the output graph

This algorithm is sound:

Soundness

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

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

1 Memory models
2 Pointer Abstractions
3 Separation Logic
4 A shape abstract domain relying on separation
5 Standard static analysis algorithms
   - Overview of the analysis
   - Post-conditions and unfolding
   - Folding: widening and inclusion checking
   - Abstract interpretation framework: assumptions and results
6 Conclusion
7 Internships
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 \( b \):

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

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

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

• **Assumptions in the abstract level**:

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

\[
m^\# := (e^\#, S^\#)
\]

\[
\gamma_{\text{mem}} : \mathbb{M}^\# \to \mathcal{P}(\mathbb{M}) \text{ concretization}
\]

\[
F^\# : \mathbb{M}^\# \to \mathbb{M}^\# \text{ sound abstract semantic function}
\]

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

\[
\triangledown : \mathbb{M}^\# \times \mathbb{M}^\# \to \mathbb{M}^\# \text{ widening operator, terminates, and such that}
\]

\[
\gamma_{\text{mem}}(m_0^\#) \cup \gamma_{\text{mem}}(m_1^\#) \subseteq \gamma_{\text{mem}}(m_0^\# \triangledown m_1^\#)
\]
Computing a loop abstract post-condition

Loop abstract semantics

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

$$\left\{ \begin{array}{l}
m_0^\# = \bot \\
m_{n+1}^\# = m_n^\# \nabla F^\#(m_n^\#) 
\end{array} \right.$$  

Soundness proof:

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

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

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

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

  \[
  m_0^\sharp \sqsubseteq m_1^\sharp \iff \gamma_{\text{mem}}(m_0^\sharp) \subseteq \gamma_{\text{mem}}(m_1^\sharp)
  \]

- **Approximation of the logical ordering**, implemented as a function \texttt{is\_le}: \( \mathbb{M}^\sharp \times \mathbb{M}^\sharp \rightarrow \{\text{true}, \top\} \), used to test the convergence of abstract iterates

  \[
  \text{is\_le}(m_0^\sharp, m_1^\sharp) = \text{true} \implies \gamma_{\text{mem}}(m_0^\sharp) \subseteq \gamma_{\text{mem}}(m_1^\sharp)
  \]

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

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

1. Memory models
2. Pointer Abstractions
3. Separation Logic
4. A shape abstract domain relying on separation
5. Standard static analysis algorithms
6. Conclusion
7. Internships
Updates and summarization

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

Separation logic

Separating conjunction combines properties on disjoint stores

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

Summaries

**Inductive predicates** describe unbounded memory regions

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


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

1 Memory models
2 Pointer Abstractions
3 Separation Logic
4 A shape abstract domain relying on separation
5 Standard static analysis algorithms
6 Conclusion
7 Internships
Internship on memory abstraction

**Summarization based on universal quantification:**

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

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

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

**Proof of correctness properties of a process scheduler for Linux**

(PhD funding available):

- schedulers rely on very complex structures **overlayed lists / balanced trees**
- relations between memory properties and the **high level scheduling policy**
- part of a project for the **generation of verified schedulers**