Memory abstraction 1

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

Xavier Rival

INRIA, ENS, CNRS

Dec, 10th. 2014
Towards memory properties

Overview of the lecture

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

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

⇒ a very broad topic, and two lectures:

This lecture:
- overview most common problems
- discuss arrays, strings
- introduction to shape analysis

Next lecture: deeper study of a family of shape analyses
Assumptions

Programs can be viewed as **transition systems**:

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

\[
S = L \times M \\
S_{\Omega} = S \uplus \{\Omega\}
\]

- a program is described by a **transition relation**:

\[
\rightarrow \subseteq S \times S_{\Omega}
\]

**Abstraction**: described by a domain \( D^{\#} \) and a concretization:

\[
\gamma : (D^{\#}, \sqsubseteq^{\#}) \longrightarrow (P(S), \subseteq)
\]
Towards memory properties

Programs: syntax

We start with a **minimal language**, to be extended with arrays, strings, pointers...

### A minimal imperative language

```plaintext
l ::= l-values
    | x (x ∈ X)

e ::= expressions
    | c (c ∈ V)
    | l (lvalue)
    | e ⊕ e (arithoperation, comparison)

s ::= statements
    | l = e (assignment)
    | s;...;s; (sequence)
    | if(e){s} (condition)
    | while(e){s} (loop)
```
Towards memory properties

Programs: semantics

We assume **classical definitions for**:  

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

We rely on the usual:

**Reachable states semantics**

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

\[
F : \mathcal{P}(S) \rightarrow \mathcal{P}(S) \\
X \mapsto \mathcal{P}(I) \cup \{ s \in S \mid \exists s' \in X, \ s' \rightarrow s \} 
\]
Towards memory properties

Programs: semantics abstraction

We assume a **memory abstraction**:
- 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:

$$
\begin{align*}
\gamma : & \quad D^\# \quad \longrightarrow \quad \mathcal{P}(S) \\
X^\# & \quad \longmapsto \quad \{ (l, m) \in S \mid m \in \gamma_{\text{mem}}(X^\#(l)) \}
\end{align*}
$$

The whole question is how do we choose $D^\#_{\text{mem}}, \gamma_{\text{mem}}$...

- previous lectures: **$X$ is fixed and finite** and, usually, **$V$ is integers**
- thus, $M \equiv V^n$
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**...

| types: | \( t_0, t_1, \ldots \) |
| 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: | \( \mathbb{M} = \mathbb{X}_{t_0} \rightarrow \mathbb{V}_{t_0} \times \mathbb{X}_{t_1} \rightarrow \mathbb{V}_{t_1} \ldots \) |

At a later point, we will add **pointers**:
- \( t_0 \) denotes pointers, \( \mathbb{V} = \ldots \cup \mathbb{V}_{\text{addr}} \)
- For a moment, we let \( t_0 \) be integers, and \( t_1 \) be booleans
Heterogeneous memory states: non relational abstraction

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

**Non relational abstraction of heterogeneous memory states**

- \( M \equiv M_{t_0} \times M_{t_1} \times \ldots \) where \( M_{t_i} = X_{t_i} \to V_{t_i} \)
- **Concretization function** (case with two types)
  \( \gamma_{nr} : \mathcal{P}(M_{t_0}) \times \mathcal{P}(M_{t_1}) \to \mathcal{P}(M) \)
  \( (m_0^\#, m_1^\#) \mapsto \{(m_{t_0}, m_{t_1}) | \forall i, m_{t_i} \in \gamma_i(m_i^\#)\} \)

**Example:** \( V = V_{\text{int}} \uplus V_{\text{bool}} \), thus, \( M = M_{\text{int}} \times M_{\text{bool}} \)

**Abstraction of** \( \mathcal{P}(X_{\text{int}} \to V_{\text{int}}) \):
- intervals
- polyhedra...

**Abstraction of** \( \mathcal{P}(X_{\text{bool}} \to V_{\text{bool}}) \):
- lattice of boolean constants
- relational abstraction with BDDs

How about a relational analysis?
Memory structures

- The definition $\mathbb{M} = X \rightarrow V$ is too restrictive.
- It ignores many ways of organizing data in the memory states.

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:**
  numeric values corresponding to the address of a memory cell
- **Strings and buffers:**
  blocks with a sequence of elements and a terminating element (e.g., *null character*)
- **Closures** (functional languages):
  pointer to function code and (partial) list of arguments
Towards memory properties

Specific properties to verify

Memory safety

Absence of memory errors (crashes, or undefined behaviors)

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

Access errors:
- Access to an array out of its bounds
- Buffer overrun (very commonly used for attacks)

Invariance properties

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

Xavier Rival (INRIA, ENS, CNRS)

Memory abstraction

Dec, 10th. 2014
Properties to verify: examples

A program closing a list of file descriptors

```c
// l points to a list
int l;

while (l != NULL) {
    close(l->fd);
    l = l->next;
}
```

Correctness properties

- memory safety
- l is supposed to store all file descriptors at all times
  Will its structure be preserved? Yes, no breakage of a next link
  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., balancing of Maps was wrong in the OCaml standard library...
Issues to consider in this lecture

- Propose a **concrete model**: expressive, intuitive...
- Abstract the **layout of memory states**
  i.e., what is the structure of the data
- Abstract the **contents of data structures**
- Express **relations** among various elements
  e.g., structural properties and properties of the contents of the structures
- Design **abstract interpretation algorithms**
  - transfer functions
  - widening
Outline

1. Towards memory properties

2. Memory models
   - Formalizing concrete memory states
   - Treatment of errors
   - Language semantics

3. Abstraction of arrays

4. Abstraction of strings and buffers

5. Basic pointer analyses

6. Three valued logic heap abstraction

7. Conclusion
A more realistic model

Not all memory cell corresponds to a variable
- a variable may correspond to several cells
- heap 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

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

$h$ is actually only a partial function

- **Memory states**: $\mathbb{M} = \mathbb{E} \times \mathbb{H}$
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></td>
</tr>
<tr>
<td></td>
<td>316</td>
<td>0x0</td>
</tr>
</tbody>
</table>

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

\[ m : \]
- 300 \mapsto 64
- 304 \mapsto 312
- 308 \mapsto 312
- 312 \mapsto 88
- 316 \mapsto 0
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>64</th>
<th>312</th>
<th>312</th>
<th>312</th>
<th>508</th>
<th>25</th>
<th>0x0</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;x = 300</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>304</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;y = 308</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>308</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;z = 312</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>316</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

$$m : \begin{align*}
300 & \mapsto 64 \\
304 & \mapsto 312 \\
308 & \mapsto 312 \\
312 & \mapsto 88 \\
316 & \mapsto 508 \\
508 & \mapsto 25 \\
512 & \mapsto 0 \\
\end{align*}$$
Extending the language syntax

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

\[
\begin{align*}
\text{l} & ::= \text{l-values} \\
& \quad | \quad \text{x} \quad (x \in X) \\
& \quad | \quad \ldots \quad \text{other kinds of l-values} \\
& \quad | \quad \text{pointers, array dereference...} \\
\text{e} & ::= \text{expressions} \\
& \quad | \quad \text{c} \quad (c \in \mathbb{V}) \\
& \quad | \quad \text{l} \quad (lvalue) \\
& \quad | \quad \text{e} \oplus \text{e} \quad (\text{arithoperation, comparison}) \\
\text{s} & ::= \text{statements} \\
& \quad | \quad \text{l} = \text{e} \quad \text{(assignment)} \\
& \quad | \quad \text{s; \ldots s;} \quad \text{(sequence)} \\
& \quad | \quad \text{if(e)}\{\text{s}\} \quad \text{(condition)} \\
& \quad | \quad \text{while(e)}\{\text{s}\} \quad \text{(loop)}
\end{align*}
\]
Extending the language semantics

Some slight modifications to the semantics of the initial language:

- **Values are addresses:** \( \mathbb{V}_{\text{addr}} \subseteq \mathbb{V} \)
- **L-values evaluate into addresses:** \([1] : \mathbb{M} \rightarrow \mathbb{V}_{\text{addr}}\)
  \[
  \llbrack x \rrbrack(e, h) = e(x)
  \]
- **Semantics of expressions** \([e] : \mathbb{M} \rightarrow \mathbb{V}_{\text{addr}}, \) mostly unchanged
  \[
  \llbrack 1 \rrbrack(e, h) = m(\llbrack 1 \rrbrack(e, h))
  \]
- **Semantics of assignment** \(l_0 : l := e; l_1 : \ldots:\)
  \[
  (l_0, e, h_0) \rightarrow (l_1, e, h_1)
  \]
  where
  \[
  h_1 = h_0[\llbrack l \rrbrack(e, h_0) \leftarrow \llbrack e \rrbrack(e, h_0)]
  \]
Extensions of the symbolic model

Our model is still not quite realistic

- 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

- Division of the memory in blocks with a base address and a size
- Division 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 concrete memory states: see CompCert project source files (Coq formalization)
Language semantics: program crash

- In an abnormal situation, **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:**
- out-of-bound array access: exception
  `java.lang.ArrayIndexOutOfBoundsException`
Language semantics: undefined behaviors

- The behavior of the program is **not specified** when an abnormal situation is encountered
- Advantage: easy implementation (often architecture driven)
- Disadvantage: unintuitive semantics, errors hard to reproduce

Modeling of undefined behavior

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

- **In C:**
  Array out-of-bound accesses and dangling pointer dereferences 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 its 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
Our purpose is not to select a language for programming
It is to remark salient language features, and their impact on abstractions

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**: \( \star p \) returns the contents of the cell pointed to by \( p \)
- **“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
String operations

- Many **data-structures** can be handled in very different ways depending on the languages
- **Strings** are just one example

**OCaml strings**

- **Abstract type**: representation not part of the language definition
- **Type safe** implementation
  - no buffer overrun
  - exception for out of bound accesses
  - i.e., like arrays
- Most operations **generate new string structures**

**C strings**

- A **string** is an **array of characters** (**char** array) with a terminal zero character
- **Direct access** to string elements (array dereference)
- String copy operation `strcpy(s, "foo_bar")`:
  - copies "foo_bar" into `s`
  - **undefined behavior** if length of `s` < 7
Manual memory management

Allocation of unbounded memory space
- How are new memory blocks made available to 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

List of choices:

- **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 object fields**: allowed or not

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

- **Memory management**: automatic or manual

We will generally assume a simple model,
unless considering specific features
Outline

1. Towards memory properties

2. Memory models

3. Abstraction of arrays
   - A micro language for manipulating arrays
   - Verifying safety of array operations
   - Abstraction of array contents
   - Abstraction of array properties

4. Abstraction of strings and buffers

5. Basic pointer analyses

6. Three valued logic heap abstraction
Programs: extension with arrays

Extension of the syntax:

\[
1 ::= \begin{align*}
& l\text{-values} \\
& \cdots \text{ previous constructions} \\
& x[e] \text{ cell of array } x \\
& \cdots ::= \cdots \text{ the rest is unchanged}
\end{align*}
\]

Extension of the semantics:

- if \( x \) is an array variable, and corresponds to an array of length \( N \), we have \( N \) cells corresponding to it, with addresses

\[
\{ e(x) + 0, e(x) + s, \ldots, e(x) + (N - 1)s \}
\]

where \( s \) is the size of an array cell (e.g., 8 bytes for a 64-bit int)

- evaluation of an array cell read:

\[
[x[e]](e, h) = \begin{cases} 
 e(x) + is & \text{if } \llbracket e \rrbracket(e, h) = i \in [0, N - 1] \\
\Omega & \text{otherwise}
\end{cases}
\]
Example

```c
// a is an integer array of length n
bool s;
do{
s = false;
    for(int i = 0; i < n - 1; i = i + 1){
        if(a[i] < a[i + 1]){
            swap(a[i] < a[i + 1]);
            s = true;
        }
    }
} while(s);
```

Properties to verify by static analysis

1. **Safety property**: the program will not crash (no index out of bound)
2. **Contents property**: if the values in the array are in \([0, 100]\) before, they are also in that range after
3. **Global array property**: at the end, the array is sorted
Outline

1. Towards memory properties

2. Memory models

3. Abstraction of arrays
   - A micro language for manipulating arrays
   - Verifying safety of array operations
   - Abstraction of array contents
   - Abstraction of array properties

4. Abstraction of strings and buffers

5. Basic pointer analyses

6. Three valued logic heap abstraction
Expressing correctness of array operations

**Goal of the analysis: establish safety**

Prove the absence of runtime error due to array reads / writes
i.e., that no $\Omega$ will ever arise

**Safety verification:**

- At label $\ell$, the analysis computes a local abstraction of the set of reachable memory states $\Phi^\#(\ell)$
- If a statement at label $\ell$ performs array read or write operation $x[e]$, where $x$ is an array of length $n$, the analysis simply needs to establish

$$\forall m \in \gamma_{\text{mem}}(\Phi^\#(\ell)), \ [e](m) \in [0, n - 1]$$

- In many cases, this can be done with an interval abstraction
  ... but not always (Exercise: when would it not be enough ?)

For now, we ignore the contents of the array
(Exercise: when does this fail ?)
Verifying correctness of array operations

Case where intervals are enough:

```c
// x array of length 40
int i = 0;
while (i < 40) {
    printf("%d; ", x[i]);
    i = i + 1;
}
```

- interval analysis establishes that $i \in [0; 39]$ at the loop head
- this allows the verification of the code

Case where intervals cannot represent precise enough invariants:

```c
// x array of length 40
int i, j;
if (0 \leq i && i < j)
    if (j < 41)
        printf("%d; ", x[i]);
```

- in the concrete, $i \in [0, 39]$ at the array access point
- to establish this in the abstract, after the first test, relation $i < j$ need be represented
- e.g., octagon abstract domain
Outline

1. Towards memory properties
2. Memory models
3. Abstraction of arrays
   - A micro language for manipulating arrays
   - Verifying safety of array operations
   - Abstraction of array contents
   - Abstraction of array properties
4. Abstraction of strings and buffers
5. Basic pointer analyses
6. Three valued logic heap abstraction
Elementwise abstraction

**Goal of the analysis: abstract contents**

Inferring invariants about the contents of the array

- e.g., that the values in the array are in a given range
- e.g., in order to verify the safety of $x[y[i + j] + k]$ or $y = n/x[i]$

**Assumption:**

- One array $t$, of known, fixed length $n$ (element size $s$)
- Scalar variables $x_0, x_1, \ldots, x_{m-1}$

**Concrete memory cell addresses:**

$$\mathbb{V}_{\text{addr}} = \{&x_0, \ldots, &x_{m-1}\} \cup \{&\tilde{t}, &\tilde{t} + 1 \cdot s, \ldots, &\tilde{t} + (n - 1) \cdot s\}$$

**Elementwise abstraction**

- Each concrete cell is mapped into one abstract cell
- $\mathbb{D}^#$ should simply be an abstraction of $\mathcal{P}(\mathbb{V}^{m+n})$
Array smashing abstraction: abstraction into one cell

The elementwise abstraction is **too costly**:
- high number of abstract cells if the arrays are big
- will not work if the size of arrays is not known statically

Alternative: use fewer abstract cells, e.g., a single cell

**Assumption**: \( m \) scalar variables, \( \bar{\tilde{t}} \) array of length \( n \)

**Array smashing**

- All cells of the array are mapped into **one abstract cell** \( \bar{\tilde{t}} \)
- **Abstract cells**: \( \mathbb{C}^\# = \{ &x_0, \ldots, &x_{m-1} \} \cup \{ &\bar{\tilde{t}} \} \)
- \( \mathbb{D}^\# \) should simply be an abstraction of \( \mathcal{P}(\mathbb{V}^{m+1}) \)

This also works if the size of the array is not known statically:

```
int n = ...;
int t[n];
```

The contents of \( t \) is represented using one abstract cell whatever the value of \( n \)
Array smashing abstraction

**Definition**

- **Abstract domain** $\mathcal{P}(\mathbb{C}^\# \rightarrow \mathcal{P}(\mathbb{V}))$
- **Abstraction function:**

$$\alpha_{\text{smash}}(H) = \left\{ \begin{array}{c}
  \&x_i \mapsto \{ h(x_i) \} \\
  \&t \mapsto \{ h(\&t + 0), \ldots, h(\&t + n - 1) \}
\end{array} \right| h \in H$$

**Example:**

- No variable, array of length 2
- **Set of concrete states:**

$$\left\{ \begin{array}{c}
  t[0] \mapsto 0 \\
  t[1] \mapsto 10
\end{array} \right\}, \left\{ \begin{array}{c}
  t[0] \mapsto 2 \\
  t[1] \mapsto 11
\end{array} \right\}, \left\{ \begin{array}{c}
  t[0] \mapsto 1 \\
  t[1] \mapsto 12
\end{array} \right\}$$

- **Abstract state**, using interval abstraction: $\&t \mapsto [0, 12]$
Weak updates: an imprecision in the analysis

Assumptions:

- **Smashing abstraction**, with the **interval abstract domain**
- Array $t$ is supposed of **known length** $n \geq 2$
- We consider statement $\ell_0 : t[i] = 0; \ell_1$
- Given $m_0^\#$, using intervals to describe a set of states at $\ell$, we wish to compute an over-approximation $m_1^\#$ of
  $$\{m_1 \mid \exists m_0 \in \gamma_{\text{mem}}(m_0^\#), (\ell_0, m_0) \rightarrow (\ell_1, m_1)\}$$
- **Abstract pre-condition**: $m_0^\#(&i) = [0, 0], m_0^\#(&\bar{t}) = [a, b]$

Post-condition:

- in the **concrete** level:
  $$\begin{cases} &t + 0 \rightarrow 0 \text{ (cell just modified)} \\
                             &t + 1 \rightarrow v \text{ where } v \in [a, b] \text{ (cell not modified)}
\end{cases}$$
- in the **abstract** level, we **only lose precision**:
  $$\&\bar{t} \rightarrow [0, 0] \sqcup [a, b] = [\min(a, 0), \max(b, 0)]$$
Weak updates

Summary:

- $i$ was known very precisely
- $\&\tilde{t}$ stands for several concrete cells
- The assignment will modify only one cell
  the others will keep their old value
- The abstraction cannot distinguish unmodified values from the modified cell
- As a consequence, the range for $\&\tilde{t}$ may only grow

Weak updates

- It would only be worse if the value of $i$ was not known precisely
- This is a significant loss in precision
- This is a limitation of all smashing analyses
Weak updates and strong updates

Definitions

- **Strong update**: modified abstract cell fully materialized, and old value fully discarded
- **Weak update**: modified abstract cell not fully materialized, and new value “joined” with old values

In the case of $t[i] := e$, weak updates may arise in the following cases:

- using a **smashing abstraction**: $\bar{t}$ denotes several concrete cells; only one gets modified, so we must keep old values
- using a **pointwise abstraction**, if $m^h_0(i) = [i, i']$ where $i < i'$:
  - one cell in $\{\&t + i \cdot s, \ldots, \&t + i' \cdot s\}$ gets modified
  - the other cells in that set remain the same
  - so we must also keep old values
Weak updates and strong updates: example

// x uninitialized array of length n
int i = 0;
while(i < n){
    x[i] = 0;
    i = i + 1;
}

Pointwise abstraction:
- initially $\forall i, m^\#(&t + i \cdot s) = \top$
- if loop unrolled completely, at the end, $\forall i, m^\#(&t + i \cdot s) = [0, 0]$
- weak updates, if the loop is not unrolled; then, at the end
  $\forall i, m^\#(&t + i \cdot s) = \top$

Smashing abstraction:
- initially $m^\#(\overline{t}) = \top$
- weak updates at each step (whatever the unrolling that is performed); at the end: $m^\#(\overline{t}) = \top$

- Weak updates may cause a serious loss of precision
- Workaround ahead: more complex array abstractions may help
Other forms of array smashing

- **Smashing does not have to affect the whole array**
- Efficient smashing strategies can be found

**Segment smashing:**
- Abstraction of the array cells into \( \{ \bar{t}_0, \ldots, \bar{t}_{k-1} \} \) where \( \bar{t}_i \) corresponds to **a segment of the array**
- Useful when sub-segments have interesting properties
- **Issue:** determine the segment by analysis

**Modulo smashing:**
- Abstraction of the array cells into \( \{ \bar{t}_0, \ldots, \bar{t}_{k-1} \} \) where \( \bar{t}_i \) corresponds to **a repeating set of offsets** \( \{ \&\bar{t} + k \cdot i \cdot s \mid k \cdot i < n \} \)
- Useful for arrays of structures
- **Issue:** determine \( k \) by analysis
Outline

1. Towards memory properties

2. Memory models

3. Abstraction of arrays
   - A micro language for manipulating arrays
   - Verifying safety of array operations
   - Abstraction of array contents
   - Abstraction of array properties

4. Abstraction of strings and buffers

5. Basic pointer analyses

6. Three valued logic heap abstraction
Example array properties

Goal of the analysis: precisely abstract contents

Discover non trivial properties of **array regions**
- Initialization to a constant (e.g., 0)
- Sortedness

An array initialization loop:

```c
// t integer array of length n
int i = 0;
while(i < n){
    t[i] = 0;
    i = i + 1;
}
```

Sketch of a hand proof:
- At iteration \( i \), \( i = i \) and the segment \( t[0], \ldots t[i-1] \) is initialized
- At the loop exit, \( i = n \) and the whole array is initialized

We need to express properties on segments; otherwise the proof cannot be completed
Array segment properties

An array initialization loop:

```c
// t integer array of length n
int i = 0;
while (i < n) {
    t[i] = 0;
    i = i + 1;
}
```

Concrete state after 6 iterations:

```
i = 6
```
```
t = [0, 0, 0, 0, 0, 0, ?, ?, ?, ?]
```

Corresponding abstract state:

```
i ∈ [1, 10]
```
```
t = zero^t(0, i - 1) ⊤
```
Array segment predicates

**Definition**

An array segment predicate is an abstract predicate that describes the contents of a contiguous series of cells in the array, such as:

- **Initialization**: $zero_t(i, j)$ iff $t$ initialized to 0 between $i$ and $j$
- **Sortedness**: $sort_t(i, j)$ iff $t$ sorted between $i$ and $j$

**Examples:**

- array satisfying $zero_t(2, 6)$:
  
  \[
  \begin{array}{cccccccccc}
  i &=& 6 \\
  t &|& 8 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 10 & 3
  \end{array}
  \]

- array satisfying $sort_t(1, 4)$ and $sort_t(6, 8)$:
  
  \[
  \begin{array}{cccccccccc}
  i &=& 6 \\
  t &|& 8 & 2 & 5 & 6 & 8 & 11 & 1 & 2 & 3 & 2
  \end{array}
  \]
Composing sortedness predicates

As part of the proof, predicates need be composed

\[
\begin{align*}
\text{zero}_t(i, j) \land \text{zero}_\bar{t}(j + 1, k) & \Rightarrow \text{zero}_t(i, k) \\
\text{zero}_t(i, j) \land t[j + 1] = 0 & \Rightarrow \text{zero}_t(i, j + 1) \\
\text{sort}_t(i, j) \land \text{sort}_\bar{t}(j + 1, k) & \not\Rightarrow \text{sort}_t(i, k) \\
t[j] \leq t[j + 1] \land \text{sort}_t(i, j) \land \text{sort}_\bar{t}(j + 1, k) & \Rightarrow \text{sort}_t(i, k)
\end{align*}
\]

- **counter example** for the third line: for $[0; 3; 9; 2; 4; 8]$, we have:

\[
\text{sort}_t(0, 2) \land \text{sort}_t(3, 5) \quad \text{but not} \quad \text{sort}_t(0, 5)
\]

Another sortedness predicate: \(\text{sort}_t(i, j, \text{min}, \text{max})\)

\[
B \leq C \land \text{sort}_t(i, j, A, B) \land \text{sort}_\bar{t}(j + 1, k, C, D) \Rightarrow \text{sort}_t(i, k, A, D)
\]
Analysis operators (for predicate zero)

Assignment transfer function:

1. Identify segments that may be modified
2. If a single segment is impacted, split it
3. Do a strong update

\[
\begin{align*}
\text{zero}_t(0, n) \land 0 \leq i < n \quad &\xrightarrow{t[i]=?} \quad \text{zero}_t(0, i - 1) \land \text{zero}_t(i + 1, n) \land 0 \leq i < n \\
\top \land 0 \leq i < n \quad &\xrightarrow{t[i]=0} \quad \text{zero}_t(i, i) \land 0 \leq i < n
\end{align*}
\]

Abstract join operator: generalizes bounds

\[
(\top \land i = 0 < n) \sqcup^\# (\text{zero}_t(0, 0) \land i = 1 < n) = (\text{zero}_t(0, i - 1) \land 0 \leq i < n)
\]
Array analysis: example

// t integer array of length \( n > 0 \)

```c
int i = 0;
while (i < n) {
    t[i] = 0;
    i = i + 1;
}
```

Xavier Rival (INRIA, ENS, CNRS)
Array analysis: example

// t integer array of length n > 0

int i = 0;

while (i < n) {
    t[i] = 0;
    i = i + 1;
}

Xavier Rival (INRIA, ENS, CNRS)
Array analysis: example

// t integer array of length $n > 0$

```
int i = 0;
```

```
t t i ⊤
```

```
while(i < n){
```

```
t t i [0, 0]
```

```
t[i] = 0;
```

```
t t i ⊤
```

```
i = i + 1;
```

```
t t i ⊤
```

```
}
```

```
t t i ⊤
```
Array analysis: example

// t integer array of length n > 0

```java
int i = 0;
while (i < n) {
    t[i] = 0;
    i = i + 1;
}
```

Xavier Rival (INRIA, ENS, CNRS)
Array analysis: example

// t integer array of length $n > 0$

```java
int i = 0;
while (i < n) {
    t[i] = 0;
    i = i + 1;
}
```

![Diagram of array analysis example]
Array analysis: example

// t integer array of length $n > 0$

```plaintext
int i = 0;
while (i < n) {
    t[i] = 0;
    i = i + 1;
}
```

Xavier Rival (INRIA, ENS, CNRS)
Array analysis: example

```
// t integer array of length n > 0

int i = 0;

while(i < n){
    t[i] = 0;
    i = i + 1;
}
```
Array analysis: example

// t integer array of length $n > 0$

```c
int i = 0;
while (i < n) {
    t[0] = 0;
    i = i + 1;
}
```

Xavier Rival (INRIA, ENS, CNRS)
Array analysis: example

// t integer array of length $n > 0$

```
t int i = 0;

while (i < n) {
  t[i] = 0;
  i = i + 1;
}
```

Xavier Rival (INRIA, ENS, CNRS)
Array analysis: example

// t integer array of length n > 0

t | T | i | T

int i = 0;

t | zero_t(0,i-1) | T | i | [0,n]

while(i < n){

  t | zero_t(0,i-1) | T | i | [0,1]

  t[i] = 0;

  t | zero_t(0,i) | T | i | [0,1]

  i = i + 1;

  t | zero_t(0,i-1) | T | i | [1,2]

}
Array analysis: example

// t integer array of length $n > 0$

```c
int i = 0;
while (i < n) {
    t[i] = 0;
    i = i + 1;
}
```

Xavier Rival (INRIA, ENS, CNRS)
Array analysis: example

// t integer array of length $n > 0$

```
int i = 0;
while (i < n) {
    t[i] = 0;
    i = i + 1;
}
```
## Array analysis: example

// t integer array of length \( n > 0 \)

```
int i = 0;
while (i < n) {
    t[i] = 0;
    i = i + 1;
}
```

---

Xavier Rival (INRIA, ENS, CNRS)
// t integer array of length $n > 0$

```c
int i = 0;
while (i < n) {
    t[i] = 0;
    i = i + 1;
}
```

Xavier Rival (INRIA, ENS, CNRS)
Partitioning of arrays

Array partitions

A partition of an array $t$ of length $n$ is a sequence $\mathcal{P} = \{e_0, \ldots, e_k\}$ of symbolic expressions where

- $e_i$ denotes the lower (resp., upper) bound of element $i$ (resp. $i-1$) of the partition
- $e_0$ should be equal to 0 (and $e_k$ to $n$)

Example:

- set of four concrete states:
  \[
  \begin{align*}
  i &= 1 & [0, 4, 1, 2, 3, 5] \\
  i &= 2 & [0, 1, 5, 2, 3, 4] \\
  i &= 3 & [2, 2, 4, 5, 1, 8] \\
  i &= 5 & [0, 2, 4, 6, 7, 9]
  \end{align*}
  \]

- partition: $\{0, i+1, 6\}$

- note that the array is always
  ▶ sorted between 0 and $i$
  ▶ sorted between $i+1$ and 5
Abstraction based on array partitions

Segment and array abstraction

An array segmentation is given by a partition $\mathcal{P} = \{e_0, \ldots, e_k\}$ and a set of abstract properties $\{P_0, \ldots, P_{k-1}\}$. Its concretization is the set of memory states $m = (e, h)$ such that

$$\forall i, \ [t[v], t[v + 1], \ldots, t[w - 1]] \text{ satisfies } P_i, \text{ where } \begin{cases} v &= [e_i](m) \\ w &= [e_{i+1}](m) \end{cases}$$

- **Partitions can be:**
  - **static**, i.e., pre-computed by another analysis [HP’08]
  - **dynamic**, i.e., computed as part of the analysis [CCL’11]
    (more complex abstract domain structure with partitions and predicates)

- **Example:** array initialization
Outline

1. Towards memory properties
2. Memory models
3. Abstraction of arrays
4. Abstraction of strings and buffers
   - A micro-language with strings
   - Abstraction
5. Basic pointer analyses
6. Three valued logic heap abstraction
7. Conclusion
Strings in programming languages

- In **high-level programming languages**:
  - **high-level** API, like OCaml String module or Java String classes
  - a set of **exceptions** in case of an invalid operation
  - **no security** risk in case of a crash

- In **C**:
  - arrays of characters
  - integration in other structures with **no protection**
  - **direct access**, with **no protection**

We focus on the case of languages with strings à la C
Abstraction of strings and buffers

Programs: syntax and semantics

We extend our simple language with strings...

Encoding of strings in C

- **Strings** are represented by **character arrays**, with a **terminating 0**
- Only characters to the first zero are meaningful
- Example of a **string buffer** of length 10 containing string “hello”
  ```
  'h' 'e' 'l' 'l' 'o' '/0' 'b' '/0' 'a' 'x'
  ```

Thus, **the language is essentially the same as for arrays**:
- data-types remain the same; we include a **char** type;
- expressions and l-values remain the same too
- we consider a set of **string operations** (typically, library functions)
Programs: string operations

String operations

- `strcpy(char * d, char * s)`: copies `s` into `d`, including terminating 0, provided there is enough space (unspecified otherwise)
- `strncpy(char * d, char * s, int n)`: copies exactly `n` characters at most, from `s` into `d`
- `printf`: interprets “%s” as a string placeholder; displays up to the terminating 0 (unspecified if there is none)

```c
char q[2];
char s[2];
char t[4];
strcpy(t, "bon");
strncpy(s, t, 2);
strcpy(q, s);
printf("nres: %s/n", q);
```

Result?

- not fully defined
- depends on the order of memory blocks in memory...
Goal of static analysis

Prove the absence of runtime errors in string buffer operations

Such errors could:

- cause **abrupt crashes** (segmentation fault) or undefined behaviors
- make **exploits** possible (e.g., by overwriting other program data)

We remark that:

- the **positions of “zero” characters** matters
- the **value of the other characters** usually does not matter
  **exception**: cases where the program decides what to do depending on non zero characters, and where that impacts the error behavior of the program
Numeric abstraction of strings

**String characters abstractions**

We consider the character abstraction below:

\[
\begin{align*}
\phi : \emptyset & \mapsto \emptyset \\
\phi : c & \mapsto '?' \\
\phi : c_0 \cdots c_{n-1} & \mapsto \phi(c_0) \cdots \phi(c_{n-1})
\end{align*}
\]

\[\alpha_{\text{string}} : S \mapsto \{\phi(s) \mid s \in S\}\]

- \(\alpha_{\text{string}}\) abstracts unneeded characters information

**Numerical abstraction**

We consider memory states that comprise only one string buffer \(t\). We can abstract each such state using two numbers

- \(t_n\): size of buffer \(t\)
- \(t_z\): position of the first 0 in \(t\) if any (otherwise, we let \(t_z = t_n\))
Abstraction of string buffers

We consider a program with integer variables $X_{\text{int}} = \{x, y, \ldots\}$ and string buffer variables $X_{\text{buf}} = \{t, u, \ldots\}$

Abstract domain

- We let $X' = X_{\text{int}} \cup \{t_n, t_z, u_n, u_z, \ldots\}$
- Each memory state $m$ gets abstracted into a state $m' = \text{abs}(m)$ over $X'$
- Given an abstract domain $(D_{\text{num}}^{\#}, \subseteq_{\text{num}})$ of $\mathcal{P}(X' \rightarrow \mathbb{Z})$, we can build an abstraction of $(\mathcal{P}(M), \subseteq)$:

$$
\gamma_{\text{buf}} : D_{\text{num}}^{\#} \rightarrow \mathcal{P}(M) \\
X^{\#} \mapsto \{ m \in M \mid \text{abs}(m) \in \gamma_{\text{num}}(X^{\#}) \}
$$

Typical choice: polyhedra
Example

Example: abstraction of `h e l l o /0 b /0 a x`
into \( t_n = 10, t_z = 5 \)

Practical implementation:
- either as a classical static analysis
- or using a transformation into an integer program

Code transformation approach:

```c
char q[2];
char s[2];
char t[4];
strcpy(t, "bon");
strncpy(s, t, 2);
strcpy(q, s);
printf("nres: %s/n", q);
```

\[
\begin{align*}
q_n &= 2; \\
s_n &= 2; \\
t_n &= 2; \\
t_z &= 3; \\
\text{if}(t_z < 2) \{ s_z = t_z; \} \;
\text{else if}(s_z < t_n) \{ s_z = s_n \} \\
\text{assert}(s_z < q_n); q_z = s_z; \\
\text{assert}(q_z < q_n);
\end{align*}
\]
Outline

1. Towards memory properties
2. Memory models
3. Abstraction of arrays
4. Abstraction of strings and buffers
5. Basic pointer analyses
   - A micro-language with pointers
6. Three valued logic heap abstraction
7. Conclusion
Programs with pointers: syntax

Syntax extension: quite a few additional constructions

\[
\begin{align*}
  l & ::= \text{l-values} \\
  & \quad | \ x \ (x \in X) \\
  & \quad | \ \ldots \\
  & \quad | \ *e \quad \text{pointer dereference} \\
  & \quad | \ l \cdot f \quad \text{field read} \\
  e & ::= \text{expressions} \\
  & \quad | \ 1 \\
  & \quad | \ \ldots \\
  & \quad | \ &l \quad \text{"address of" operator} \\
  s & ::= \text{statements} \\
  & \quad | \ \ldots \\
  & \quad | \ x = \text{malloc}(c) \quad \text{allocation of } c \text{ bytes} \\
  & \quad | \ \text{free}(x) \quad \text{deallocation of the block pointed to by } x
\end{align*}
\]

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

Case of l-values:

\[ [x](e, h) = e(x) \]
\[ *[e](e, h) = \begin{cases} h([e](e, h)) & \text{if } [e](e, h) \neq 0 \land [e](e, h) \in \text{Dom}(h) \\ \Omega & \text{otherwise} \end{cases} \]
\[ [1 \cdot f](e, \text{heap}) = [1](e, h) + \text{offset}(f) \text{ (numeric offset)} \]

Case of expressions:

\[ [1](e, \text{heap}) = h([1](e, \text{heap})) \]
\[ &[1](e, \text{heap}) = [1](e, h) \]

Case of statements:

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

- **memory deallocation** \( \text{free}(x) \): \( (e, h) \rightarrow (e, h') \) where \( k = e(x) \) and
  \[ h = h' \cup \{k \mapsto v_k, k+1 \mapsto v_{k+1}, \ldots, k+c-1 \mapsto v_{k+c-1}\} \]
Pointer non relational abstraction: null pointers

The dereference of a null pointer will cause programs to crash

We go back to the non relational abstraction of heterogeneous states

- \( V = V_{\text{addr}} \cup V_{\text{int}} \), \( X = X_{\text{addr}} \cup X_{\text{int}} \)
- we apply a non relational abstraction to pointer variables, based on \( D^\#_{\text{addr}} \) and \( \gamma_{\text{addr}} : D^\#_{\text{addr}} \rightarrow P(V_{\text{addr}}) \)

Null pointer analysis

Abstract lattice for addresses:

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

- 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 programs to crash.

This requires a similar abstraction:

Null pointer analysis

Abstract lattice for addresses:

- $\gamma_{addr}(\perp) = \emptyset$
- $\gamma_{addr}(\top) = \bigvee_{addr} \times H$
- $\gamma_{addr}(\text{Not dangling}) = \{(v, h) \mid h \in H, v \in \text{Dom}(h)\}$

- very lightweight, can typically resolve rather trivial cases
- useful for C
- in Java, superseded by the requirement that any variable be initialized
Basic pointer analyses

A micro-language with pointers

Pointer non relational abstraction: pointer aliasing

Determine where a pointer may store a reference to

Very useful to support client analyses:

```plaintext
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 `x`?
  0, since it is not modified at line 5...

Basic pointer abstraction

- We assume a set of abstract memory locations \( A^\# \) is fixed:
  \[
  A^\# = \{&x, &y, \ldots, &t, a_0, a_1, \ldots, a_N\}
  \]

- All concrete addresses are abstracted into \( A^\# \)

- A pointer value is abstracted by the abstraction of the addresses it may point to (example, for `p`: \{&x\})
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 operations to compute 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: \( \{\{x(-n)^k, &(x(-n)^{k+1}) \mid k \in \mathbb{N}\} \)})
Weak update problems

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

\[
\text{int} \ \star p;
\]

\[
\text{if(?)}
\]

\[
p = \&x;
\]

\[
\text{else}
\]

\[
p = \&y;
\]

\[
\star p = 0;
\]

• What is the final range for \(x\)?
• What is the final range for \(y\)?
Weak update problems

```
x ∈ [−10, −5]; y ∈ [5, 10]
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 ∈ [−10, 0], y ∈ [0, 10]

Weak updates

Any imprecision in the analysis may lead to weak updates...
Limitation of basic pointer analyses

- **Weak updates:**
  imprecisions for pointer values quickly spread out

- Many programs with pointers address *unbounded memory*
  e.g., to create lists, trees and other dynamically allocated structures
  most pointer analyses do not deal with this well...

- Pointer analyses do not nicely capture *structural invariants*
  e.g., lists, trees, but also nested structures
Outline

1. Towards memory properties
2. Memory models
3. Abstraction of arrays
4. Abstraction of strings and buffers
5. Basic pointer analyses
6. Three valued logic heap abstraction
   - Basic principles
   - Building an abstract domain
   - Weakening abstract elements
   - Computation of transfer functions
An abstract representation of memory states: shape graphs

Goal of the static analysis

Discover complex invariants of programs that manipulate unbounded heap

Observation: representation of memory states by shape graphs

- **Nodes** (aka, atoms) denote **memory locations**
- **Edges** denote **properties**, such as:
  - “field f of location u points to v”
  - “variable x is stored at location u”

Two alias pointers:

\[
\begin{align*}
  y & \rightarrow u_1 \\
  x & \rightarrow u_0 \\
\end{align*}
\]

A list of length 2:

\[
\begin{align*}
  x & \rightarrow u_0 \quad \rightarrow u_1 \quad \rightarrow u_2 \\
\end{align*}
\]

⇒ We need to over-approximate sets of shape graphs
## Shape graphs and their representation

### Description with predicates

- **Boolean encoding**: nodes are atoms $u_0, u_1, \ldots$
- **Predicates over atoms**:
  - $x(u)$: variable $x$ contains the address of $u$
  - $n(u, v)$: field of $u$ points to $v$
- **Truth values**: traditionally noted 0 and 1 in the TVLA literature

### Two alias pointers:

![Diagram](image)

<table>
<thead>
<tr>
<th></th>
<th>$x$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u_0$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$u_1$</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$u_2$</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>$u_0$</th>
<th>$u_1$</th>
<th>$u_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u_0$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$u_1$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$u_2$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### A list of length 2:

![Diagram](image)

<table>
<thead>
<tr>
<th></th>
<th>$x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u_0$</td>
<td>1</td>
</tr>
<tr>
<td>$u_1$</td>
<td>0</td>
</tr>
<tr>
<td>$u_2$</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>$u_0$</th>
<th>$u_1$</th>
<th>$u_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u_0$</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$u_1$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$u_2$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Unknown value: three valued logic

How to abstract away some information? i.e., to abstract several graphs into one?

**Example**: pointer variable $p$ alias with $x$ or $y$

A boolean lattice

- Use **predicate tables**
- Add a $T$ boolean value;
  (denoted to by $\frac{1}{2}$ in TVLA papers)

- Graph representation: dotted edges
- **Abstract graph**:

- Memory abstraction

---

Xavier Rival (INRIA, ENS, CNRS)
Summary nodes

We cannot talk about unbounded memory states with finitely many nodes.

Lists of lengths 1, 2, 3:

\[
\begin{align*}
&x \rightarrow u_0^n \rightarrow u_1 \\
&x \rightarrow u_0^n \rightarrow u_1^n \rightarrow u_2 \\
&x \rightarrow u_0^n \rightarrow u_1^n \rightarrow u_2^n \rightarrow u_3
\end{align*}
\]

We would like to summarize the lists.

An idea:

- Choose a node to represent several concrete nodes.
- Similar to smashing.

Definition: summary node

A **summary node** is an atom that may denote several concrete atoms.

- Edges to \( u_1 \) are dotted.
A few interesting predicates

We have already seen:

- \( x(u) \): variable \( x \) contains the address of \( u \)
- \( n(u, v) \): field of \( u \) points to \( v \)
- \( \text{sum}(u) \): whether \( u \) is a summary node (convention: either 0 or \( \frac{1}{2} \))

The properties of lists are not well-captured in

\[
\begin{align*}
\exists v_0, v_1, & \quad v_0 \neq v_1 \\
\wedge & \quad n(v_0, u) \\
\wedge & \quad n(v_1, u)
\end{align*}
\]

“Is shared”

\( \text{sh}(u) \) ssi:

- \( u = v \lor \exists u_0, \ n(u, u_0) \land r(u_0, v) \)

Predicates defined by transitive closure

- **Reachability**: \( r(u, v) \) ssi

- **Acyclicity**: \( \overline{\text{acy}}(v) \)
  
  similar, with a negation
Outline

1. Towards memory properties
2. Memory models
3. Abstraction of arrays
4. Abstraction of strings and buffers
5. Basic pointer analyses
6. Three valued logic heap abstraction
   - Basic principles
   - Building an abstract domain
   - Weakening abstract elements
   - Computation of transfer functions
Three structures

Definition: 3-structures

A 3-structure is a tuple \((U, P, \phi)\):

- a set \(U = \{u_0, u_1, \ldots, p_m\}\) of \textit{atoms}
- a set \(P = \{p_0, p_1, \ldots, p_n\}\) of \textit{predicates}
  (we write \(k_i\) for the arity of predicate \(p_i\))
- a \textbf{truth table} \(\phi\) such that \(\phi(p_i, u_{l_1}, \ldots, u_{l_{k_i}})\) denotes the truth value of \(p_i\) for \(u_{l_1}, \ldots, u_{l_{k_i}}\)

\(\text{note: truth values are elements of the lattice } \{0, \frac{1}{2}, 1\}\)

\[
\begin{array}{c|c|c}
\text{ } & \text{x} & \text{sum} \\
\hline
u_0 & 1 & 0 \\
u_1 & 0 & \frac{1}{2} \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{ } & \text{n} & \text{ } \\
\hline
u_0 & u_0 & u_1 \\
u_0 & 0 & 1 \\
u_1 & 0 & 0 \\
\end{array}
\]
Embedding

- How to compare two 3-structures?
- How to describe the concretization of 3-structures?

The embedding principle

Let $S_0 = (U_0, P, \phi_0)$ and $S_1 = (U_1, P, \phi_1)$ be two three structures, with the same sets of predicates.

Let $f : U_0 \rightarrow U_1$, surjective.

We say that $f$ embeds $S_0$ into $S_1$ iff

$$
\text{for all predicate } p \in P \text{ or arity } k, \\
\text{for all } u_{l_1}, \ldots, u_{l_k_i} \in U_0, \\
\phi_0(u_{l_1}, \ldots, u_{l_k_i}) \sqsubseteq \phi_0(f(u_{l_1}), \ldots, f(u_{l_k_i}))
$$

Then, we write $S_0 \sqsubseteq^f S_1$

Note: we use the order $\sqsubseteq$ of the lattice $\{0, \frac{1}{2}, 1\}$
Embedding examples

where \( f : u_0 \mapsto u_0; u_1 \mapsto u_1; u_2 \mapsto u_1 \)

where \( f : u_0 \mapsto u_0; u_1 \mapsto u_1; u_2 \mapsto u_1; u_3 \mapsto u_1 \)

where \( f : u_0 \mapsto u_0; u_1 \mapsto u_1; u_2 \mapsto u_1 \)

Note on the last example

- Reachability would be necessary to constrain it be a list
- Alternatively: cells should not be shared
Two structures and concretization

Concrete states correspond to 2-structures

- **2-structure**: a 3-structure $(\mathcal{U}, \mathcal{P}, \phi)$ is a 2-structure, if and only if $\phi$ always returns in $\{0, 1\}$
- A **2-structure** corresponds to a set of concrete memory states (environment, heap):
  - we simply need to take into account all mappings of addresses into the memory
  - we let $\text{stores}(S)$ denote the stores corresponding to 2-structure $S$
  - more on this in the next lecture; here we keep it informal

Concretization

$$\gamma(S) = \bigcup \{\text{stores}(S') \mid S' \text{ 2-structure s.t. } \exists f, S' \sqsubseteq^f S\}$$
Concretization examples

- **Without reachability:**

  \[
  x \rightarrow u_0 \xrightarrow{n} u_1 \xrightarrow{n} u_2
  \]

  where \( f : u_0 \mapsto u_0; u_1 \mapsto u_1; u_2 \mapsto u_1 \)

- **With reachability:**

  \[
  x \rightarrow u_0^n \xrightarrow{n} u_1^n \xrightarrow{n} u_2^n
  \]

  \( r(u_0, u_1) \)

  where \( f : u_0 \mapsto u_0; u_1 \mapsto u_1; u_2 \mapsto u_1 \)
Principle for the design of sound transfer functions

How to carry out static analysis using 3-structures?

Embedding theorem

- Let $S_0 = (U_0, P, \phi_0)$ and $S_1 = (U_1, P, \phi_1)$ be two three structures, with the same sets of predicates.
- Let $f : U_0 \rightarrow U_1$, such that $S_0 \sqsubseteq^f S_1$.
- Let $\Psi$ be a logical formula, with variables in $X$ and $g : X \rightarrow U_0$ be an assignment for the variables of $\Psi$.

Then, $\llbracket \Psi|_g \rrbracket(S_0) \sqsubseteq \llbracket \Psi|_{f \circ g} \rrbracket(S_1)$.
Principle for the design of sound transfer functions

Transfer functions for static analysis

- Semantics of concrete statements encoded into boolean formulas
- **Example**: assignment $y := x$
  - let $y'$ denote the *new* value of $y$
  - add the constraint $y'(u) = x(u)$
  - rename $y'$ into $y$

**Full examples** of transfer functions computation in a few slides...

- Evaluation in the abstract is sound (embedding theorem)

**Advantages:**

- abstract transfer functions derive directly from the concrete transfer functions
  - intuition: $\alpha \circ f \circ \gamma$...
- the same solution works for **weakest pre-conditions**
Outline

1 Towards memory properties
2 Memory models
3 Abstraction of arrays
4 Abstraction of strings and buffers
5 Basic pointer analyses
6 Three valued logic heap abstraction
   - Basic principles
   - Building an abstract domain
   - Weakening abstract elements
   - Computation of transfer functions
A powerset abstraction

- Do 3-structures allow for a sufficient level of precision?
- How to over-approximate a set of two-structures?

```c
int * x; int * y; ...
int * p = NULL;
if(...){
    p = x;
} else{
    p = y;
}
printf("%d", *p);
*p = ...;
```

After the if statement:

```
y → u_1

y → u_1
```

```
p
x → u_0
```

abstracting here would be imprecise

Powerset abstraction

- Shape analyzers usually rely on a powerset abstract domain, i.e., TVLA manipulates finite disjunctions of 3-structures
- How to ensure disjunctions will not grow infinite?
Canonical abstraction

Canonicalization principle

Let $\mathcal{L}$ be a lattice, $\mathcal{L}' \subseteq \mathcal{L}$ be a finite sub-lattice and $\text{can} : \mathcal{L} \to \mathcal{L}'$:

- $\text{can}$ called a **canonicalization** if it is an upper closure operator
- then, $\text{can}$ extends into a canonicalization operator of $\mathcal{P}(\mathcal{L})$, into $\mathcal{P}(\mathcal{L}')$:
  
  $$\text{can}(\mathcal{E}) = \{\text{can}(x) \mid x \in \mathcal{E}\}$$

To make the powerset domain work, we simply need a $\text{can}$ over 3-structures

A canonicalization over 3-structures

- We assume there are $n$ variables $x_1, \ldots, x_n$
  
  **Thus the number of unary predicates is finite**

- **Sub-lattice**: structures with atoms **distinguished by the values of the unary predicates** (or *abstraction predicates*) $x_1, \ldots, x_n$

We may choose another set of predicates for the sub-lattice representation
Canonical abstraction

1. Identify nodes that have different abstraction predicates
2. When several nodes have the same abstraction predicate introduce a summary node
3. Compute new predicate values by doing a join over truth values

Elements not merged:
- \( y \rightarrow u_1 \)
- \( p \rightarrow u_0 \)
- \( x \rightarrow u_0 \)

Elements merged:
- Lists of lengths 1, 2, 3:
- Abstract into:

\[
\begin{align*}
  x & \rightarrow u_0^n \rightarrow u_1 \\
  x & \rightarrow u_0^n \rightarrow u_1^n \rightarrow u_2 \\
  x & \rightarrow u_0^n \rightarrow u_1^n \rightarrow u_2^n \rightarrow u_3
\end{align*}
\]

\[
\begin{align*}
  x & \rightarrow u_0^n \rightarrow u_1 \\
  u_0^n & \rightarrow u_1^n \\
  x & \rightarrow r(x)
\end{align*}
\]
Outline

1. Towards memory properties
2. Memory models
3. Abstraction of arrays
4. Abstraction of strings and buffers
5. Basic pointer analyses
6. Three valued logic heap abstraction
   - Basic principles
   - Building an abstract domain
   - Weakening abstract elements
   - Computation of transfer functions
Assignment: a simple case

Statement $l_0 : y = y \rightarrow n; l_1 : \ldots$  

Pre-condition $S$ \[ x, y \xrightarrow{u_0} u_1 \xrightarrow{n} u_2 \]

Transfer function

- Should yield an over-approximation of $\{ m_1 \in \mathbb{M} \mid (l_0, m_0) \rightarrow (l_1, m_1) \}$
- We let "pruned predicates" denote predicates after evaluation of the assignment, to evaluate them in the same structure

Then:

\[
\begin{align*}
x'(u) &= x(u) \\
y'(u) &= \exists v, \ y(v) \land n(v, u) \\
n'(u, v) &= n(u, v)
\end{align*}
\]

Result:

This was exactly what we expected
Assignment: a more involved case

Statement $l_0 : y = y \rightarrow n; l_1 : \ldots$

Pre-condition $S$

- Let us try to resolve the update in the same way as before:
  
  $$
  \begin{align*}
  x'(u) &= x(u) \\
  y'(u) &= \exists v, \ y(v) \land n(v, u) \\
  n'(u, v) &= n(u, v)
  \end{align*}
  $$

- We cannot resolve $y'$:
  
  $$
  \begin{cases}
  y'(u_0) = 0 \\
  y'(u_1) = \frac{1}{2}
  \end{cases}
  $$

  Imprecision: after the statement, $y$ may point to anywhere in the list, save for the first element...

- The assignment transfer function cannot be computed immediately
- We need to refine the 3-structure first
Focus

Focusing on a formula

We assume a 3-structure $S$ and a boolean formula $f$ are given, we call a focusing $S$ on $f$ the generation of a set $\hat{S}$ such that:

- $f$ evaluates to 0 or 1 on all elements of $\hat{S}$
- precision was gained: $\forall S' \in \hat{S}, S' \subseteq S$
- soundness is preserved: $\gamma(S) = \bigcup\{\gamma(S') \mid S' \in \hat{S}\}$

- Focusing algorithms are complex and tricky (see biblio)
- Involves splitting of summary nodes, solving of boolean constraints

Example: focusing on $y'(u) = \exists v, y(v) \land n(v, u)$

We obtain (we show $y$ and $y'$):

![Diagram of focusing on a formula]
Focus and coerce

Some of the 3-structures generated by focus are not precise

\[ u_0, u_1 \]
\[ x, y \quad r(x) \]

\[ u_1 \] is reachable from \( x \), but there is no sequence of \( n \) fields: this structure has empty concretization

\[ u_0 \] has an \( n \)-field to \( u_1 \) so \( u_1 \) denotes a unique atom and cannot be a summary node

Coerce operation

It enforces logical constraints among predicates and discards 3-structures with an empty concretization

Result:
Focus, transfer, abstract...

**Computation of a transfer function**

We consider a transfer function encoded into boolean formula $f$

![Diagram showing the computation of a transfer function]

**Soundness proof** steps:

1. sound encoding of the semantics of program statements into formulas typically, no loss of precision at this stage
2. focusing should yield an over-approximation of its input
3. canonicalization over-approximates graph (truth blurring weakening)

**A common picture in shape analysis**
Outline

1. Towards memory properties
2. Memory models
3. Abstraction of arrays
4. Abstraction of strings and buffers
5. Basic pointer analyses
6. Three valued logic heap abstraction
7. Conclusion
Conclusion

Concrete semantics:

- Splitting environment and heap
- Taking into account of the representation of data

Many families of domain specific abstractions:

- Based on numerical methods
  path based pointer analyses, array segment analyses, string analyses
- Symbolic abstractions based on pointer sets, structural predicates
- Locally concretize / globally abstract pattern (TVLA, arrays...)
  More on this during the next lecture...
Bibliography


- [CSSV’03]: CSSV: towards a realistic tool for statically detecting all buffer overflows in C. Nurit Dor, Michael Rodeh, Shmuel Sagiv. In PLDI’03, pages 155-167.