

# Memory abstraction 1

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

Xavier Rival

INRIA, ENS, CNRS

Oct, 19th. 2016

# Outline

- 1 Memory models
  - Towards memory properties
  - Formalizing concrete memory states
  - Treatment of errors
  - Language semantics
- 2 Abstraction of arrays
- 3 Basic pointer analyses
- 4 Three valued logic heap abstraction
- 5 Conclusion

# 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 memory models** and **memory properties**
- abstraction of **arrays**
- abstraction of **pointer structures** / **shape analysis**

**Next lecture:** abstractions based on **separation logic**

# Assumptions

Imperative programs viewed as **transition systems**:

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

$$\begin{aligned}\mathbb{S} &= \mathbb{L} \times \mathbb{M} \\ \mathbb{S}_\Omega &= \mathbb{S} \uplus \{\Omega\}\end{aligned}$$

- **transition relation**:

$$(\rightarrow) \subseteq \mathbb{S} \times \mathbb{S}_\Omega$$

**Abstraction** of sets of states described by domain  $\mathbb{D}^\#$  and concretization

$$\gamma : (\mathbb{D}^\#, \sqsubseteq^\#) \longrightarrow (\mathcal{P}(\mathbb{S}), \subseteq)$$

# Programs: syntax

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

<b>l</b>	::=	<b>l-valules</b>	
		<b>x</b>	( $x \in \mathbb{X}$ )
		<b>...</b>	<b>we will add other kinds of l-values pointers, array dereference...</b>
<b>e</b>	::=	<b>expressions</b>	
		<b>c</b>	( $c \in \mathbb{V}$ )
		<b>l</b>	( <i>lvalue</i> )
		<b><math>e \oplus e</math></b>	( <i>arithoperation, comparison</i> )
<b>s</b>	::=	<b>statements</b>	
		<b><math>l = e</math></b>	(assignment)
		<b><math>s; \dots s;</math></b>	(sequence)
		<b><math>\text{if}(e)\{s\}</math></b>	(condition)
		<b><math>\text{while}(e)\{s\}</math></b>	(loop)

# Programs: semantics

We assume **classical definitions for**:

- **l-values**:  $\llbracket l \rrbracket : \mathbb{M} \rightarrow \mathbb{X}$
- **expressions**:  $\llbracket e \rrbracket : \mathbb{M} \rightarrow \mathbb{V}$
- **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  $\llbracket S \rrbracket_{\mathcal{R}} = \mathbf{lfp} F$  where

$$\begin{array}{lcl}
 F : \mathcal{P}(\mathbb{S}) & \longrightarrow & \mathcal{P}(\mathbb{S}) \\
 X & \longmapsto & \mathbb{S}_{\mathcal{I}} \cup \{s \in \mathbb{S} \mid \exists s' \in X, s' \rightarrow s\}
 \end{array}$$

# Programs: semantics abstraction

We assume a **memory abstraction**:

- memory abstract domain  $\mathbb{D}_{\text{mem}}^\#$
- concretization function  $\gamma_{\text{mem}} : \mathbb{D}_{\text{mem}}^\# \rightarrow \mathcal{P}(\mathbb{M})$

## Reachable states abstraction

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

$$\begin{aligned} \gamma : \mathbb{D}^\# &\longrightarrow \mathcal{P}(\mathbb{S}) \\ \mathbb{X}^\# &\longmapsto \{(\ell, m) \in \mathbb{S} \mid m \in \gamma_{\text{mem}}(\mathbb{X}^\#(\ell))\} \end{aligned}$$

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

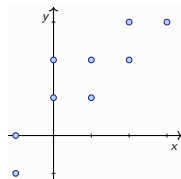
- previous lectures:  $\mathbb{X}$  is fixed and finite and,  $\mathbb{V}$  is scalars (integers or floats), thus,  $\mathbb{M} \equiv \mathbb{V}^n$
- today, we will extend the language and the abstractions

# Abstraction of purely numeric memory states

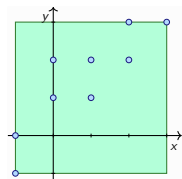
## Purely numeric case

- $\mathbb{V}$  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  $\mathbb{V}^N$**

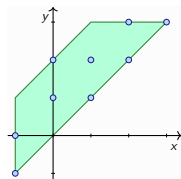
**Example:**  $N = 2$ ,  $\mathbb{X} = \{x, y\}$



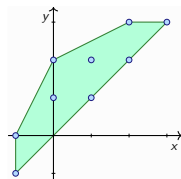
concrete set



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

- **types:**  $t_0, t_1, \dots$
- **values:**  $\mathbb{V} = \mathbb{V}_{t_0} \uplus \mathbb{V}_{t_1} \uplus \dots$
- finitely many **variables**; each has a **fixed type**:  $\mathbb{X} = \mathbb{X}_{t_0} \uplus \mathbb{X}_{t_1} \uplus \dots$
- **memory states:**

$$\mathbb{M} = \mathbb{X}_{t_0} \rightarrow \mathbb{V}_{t_0} \times \mathbb{X}_{t_1} \rightarrow \mathbb{V}_{t_1} \dots$$

- At a later point, we will add **pointers**:  $t_0$  **denotes pointers**,  $\mathbb{V} = \dots \uplus \mathbb{V}_{\text{addr}}$
- For the 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

- $\mathbb{M} \equiv \mathbb{M}_0 \times \mathbb{M}_1 \times \dots$  where  $\mathbb{M}_i = \mathbb{X}_i \rightarrow \mathbb{V}_i$

- **Concretization function** (case with two types)

$$\begin{aligned} \gamma_{nr} : \mathcal{P}(\mathbb{M}_0) \times \mathcal{P}(\mathbb{M}_1) &\longrightarrow \mathcal{P}(\mathbb{M}) \\ (m_0^\#, m_1^\#) &\longmapsto \{(m_0, m_1) \mid \forall i, m_i \in m_i^\#\} \end{aligned}$$

**Example:**  $\mathbb{V} = \mathbb{V}_{\text{int}} \uplus \mathbb{V}_{\text{bool}}$ , thus,  $\mathbb{M} = \mathbb{M}_{\text{int}} \times \mathbb{M}_{\text{bool}}$

**Abstraction of  $\mathcal{P}(\mathbb{X}_{\text{int}} \rightarrow \mathbb{V}_{\text{int}})$ :**

- intervals
- polyhedra...

**Abstraction of  $\mathcal{P}(\mathbb{X}_{\text{bool}} \rightarrow \mathbb{V}_{\text{bool}})$ :**

- lattice of boolean constants
- partitioning abstraction

# Memory structures

- To describe memories, the definition  $\mathbb{M} = \mathbb{X} \rightarrow \mathbb{V}$  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)

# 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...)**

- **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

```
//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., balancing of Maps was wrong in the OCaml standard library...

# A more realistic model

**Not all memory cell** corresponds to a variable

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

## Environment + Heap

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

$$\begin{aligned}\mathbb{E} &= \mathbb{X} \rightarrow \mathbb{V}_{\text{addr}} \\ \mathbb{H} &= \mathbb{V}_{\text{addr}} \rightarrow \mathbb{V}\end{aligned}$$

$h$  is actually only a partial function

- **Memory states** (or **memories**):  $\mathbb{M} = \mathbb{E} \times \mathbb{H}$

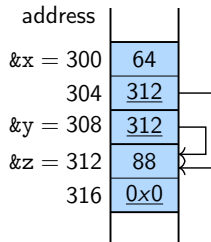
**Avoid confusion between heap (function from addresses to values) and dynamic allocation space (often referred to as “heap”)**

# Example of a concrete memory state (variables)

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

## Memory layout

(pointer values underlined)



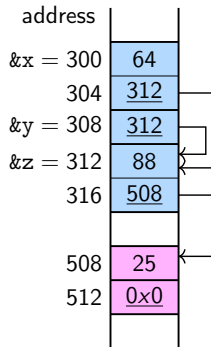
$$\begin{array}{l}
 e : \quad x \quad \mapsto \quad 300 \\
 \quad \quad y \quad \mapsto \quad 308 \\
 \quad \quad z \quad \mapsto \quad 312
 \end{array}$$

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

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

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

## Memory layout



```

e :  x   ↦ 300
     y   ↦ 308
     z   ↦ 312
  
```

```

hi : 300 ↦ 64
      304 ↦ 312
      308 ↦ 312
      312 ↦ 88
      316 ↦ 508
      508 ↦ 25
      512 ↦ 0
  
```



# Extending the semantic domains

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:**  $\llbracket \mathbf{l} \rrbracket : \mathbb{M} \rightarrow \mathbb{V}_{\text{addr}}$

$$\llbracket \mathbf{x} \rrbracket(e, h) = e(\mathbf{x})$$

- **Semantics of expressions**  $\llbracket \mathbf{e} \rrbracket : \mathbb{M} \rightarrow \mathbb{V}$ , mostly unchanged

$$\llbracket \mathbf{l} \rrbracket(e, h) = h(\llbracket \mathbf{l} \rrbracket(e, h))$$

- **Semantics of assignment**  $l_0 : \mathbf{l} := \mathbf{e}; l_1 : \dots :$

$$(l_0, e, h_0) \longrightarrow (l_1, e, h_1)$$

where

$$h_1 = h_0[\llbracket \mathbf{l} \rrbracket(e, h_0) \leftarrow \llbracket \mathbf{e} \rrbracket(e, h_0)]$$

# 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 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**:  $(\rightarrow) \subseteq \mathbb{S} \times (\{\Omega\} \uplus \mathbb{S})$

## 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

- 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
- 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  $(\ell_0, m_0)$  such that  $\forall m_1 \in \mathbb{M}, (\ell_0, m_0) \rightarrow (\ell_1, m_1)$
- **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

# Pointers and records / structures / objects

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...  

```
struct {int a; int b} x;  
int * y = &(x · b);
```

# Pointer arithmetics

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

# String operations

- Many **data-structures** can be handled in very different ways depending on the language
- **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 *`)** 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 **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

## Choices to fix a memory model

- **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

In this course, we start with a simple model, and add specific features one by one (arrays, pointers) in order to study corresponding abstractions

# Outline

- 1 Memory models
- 2 **Abstraction of arrays**
  - A micro language for manipulating arrays
  - Verifying safety of array operations
  - Abstraction of array contents
  - Abstraction of array properties
- 3 Basic pointer analyses
- 4 Three valued logic heap abstraction
- 5 Conclusion

# Programs: extension with arrays

## Extension of the syntax:

$l$	::=	<b>l-valules</b>	
		...	previous constructions
		$x[e]$	cell of array $x$
...	::=	...	the rest is unchanged

## Extension of the states:

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

$$\{e(x) + 0, e(x) + s, \dots, e(x) + (N - 1) \cdot s\}$$

where  $s$  is the **size of a base type value** (8 bytes for a 64-bit int)

## Extension of the semantics, case of an **array cell read**:

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

# Example

```

// 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, 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 Memory models
- 2 **Abstraction of arrays**
  - A micro language for manipulating arrays
  - Verifying safety of array operations
  - Abstraction of array contents
  - Abstraction of array properties
- 3 Basic pointer analyses
- 4 Three valued logic heap abstraction
- 5 Conclusion

# 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**  $\Phi^\sharp(\ell)$
- If a statement at label  $\ell$  performs array read or write operation  $\mathbf{x}[e]$ , where  $\mathbf{x}$  is an array of length  $n$ , the analysis simply needs to establish
 
$$\forall m \in \gamma_{\text{mem}}(\Phi^\sharp(\ell)), \llbracket e \rrbracket(m) \in [0, n - 1]$$
- 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 array contents (**exercise**: when does this fail ?)

# Verifying correctness of array operations

## Case where intervals are enough:

```
//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:

```
//x array of length 40
int i, j;
if(0 ≤ 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 Memory models
- 2 **Abstraction of arrays**
  - A micro language for manipulating arrays
  - Verifying safety of array operations
  - Abstraction of array contents
  - Abstraction of array properties
- 3 Basic pointer analyses
- 4 Three valued logic heap abstraction
- 5 Conclusion

# 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  $x_0, x_1, \dots, x_{m-1}$

## Elementwise abstraction

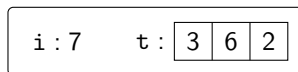
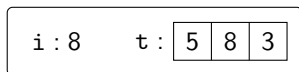
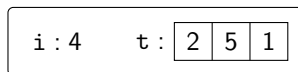
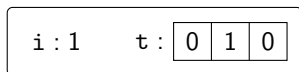
- **Each** concrete cell is **mapped into one abstract cell**
- $\mathbb{D}^\sharp$  should simply be an **abstraction of  $\mathcal{P}(\mathbb{V}^{m+n})$**  (relational or not)

## Abstract and concrete memory cell addresses:

$$\mathbb{C}^\sharp = \mathbb{V}_{\text{addr}} = \{\&x_0, \dots, \&x_{m-1}\} \cup \{\&\bar{t}, \&\bar{t} + 1 \cdot s, \dots, \&\bar{t} + (n - 1) \cdot s\}$$

# Elementwise abstraction example

We consider the following **set of concrete states**:



The **elementwise abstraction** produces the following vectors:

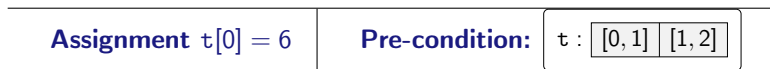
$$\begin{array}{cc} (1, 0, 1, 0) & (4, 2, 5, 1) \\ (8, 5, 8, 3) & (7, 3, 6, 2) \end{array}$$

After applying the **interval abstraction**, we get:

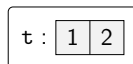
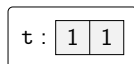
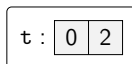
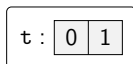
$$([1, 8], [0, 5], [1, 8], [0, 3])$$

This is **precise** but **costly** if arrays are big; also we need to know statically the length of arrays...

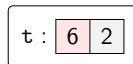
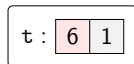
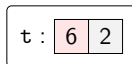
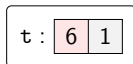
## Post-condition for an assignment: example 1



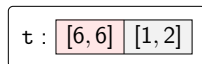
- concrete pre-condition:**



- effect** of the assignment **in the concrete** and post-condition:



Thus, we obtain the **abstract post-condition**:



This analysis step is **precise**, but what if the index is not known so precisely ?

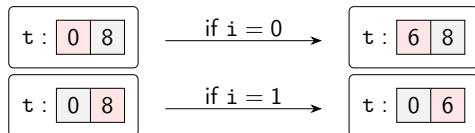
## Post-condition for an assignment: example 2

Assignment  $t[i] = 6$ Pre-condition:  $i \in [0, 1] \wedge$  $t : \begin{array}{|c|c|} \hline [0, 0] & [8, 8] \\ \hline \end{array}$ 

- concrete pre-condition:

 $t : \begin{array}{|c|c|} \hline 0 & 8 \\ \hline \end{array}$ 

- effect of the assignment **in the concrete** and post-condition:

Thus, we obtain the **abstract post-condition**: $t : \begin{array}{|c|c|} \hline [0, 6] & [6, 8] \\ \hline \end{array}$ 

This analysis step looks quite coarse, but it is actually fine here:  
each cell may get the new value or keep the old one...

# Two kinds of abstract updates

## Strong updates

- **One modified concrete cell abstracted by one, precisely known abstract cell**
- The effect of the update **is computed precisely** by the analysis

Strong updates are the **most favorable case**, as new information is computed precisely, and known information is not lost (example 1)

## 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**

In the example we have just seen, the weak update loses no information...

# 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, one array  $\bar{t}$  of length  $n$

## Array smashing

- All cells of the array are mapped into **one abstract cell**  $\bar{t}$
- **Concrete cells**:  

$$\mathbb{V}_{\text{addr}} = \{\&x_0, \dots, \&x_{m-1}\} \cup \{\&\bar{t}, \&\bar{t} + 1 \cdot s, \dots, \&\bar{t} + (n-1) \cdot s\}$$
- **Abstract cells**:  $\mathbb{C}^\# = \{\&x_0, \dots, \&x_{m-1}\} \cup \{\&\bar{t}\}$
- $\mathbb{D}^\#$  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];
```

# Array smashing abstraction

## Definition

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

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

**Example**, with no variable and an array of length 2:

- **Set of concrete states**  $H$ :

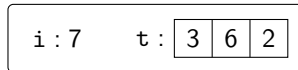
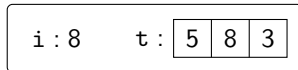
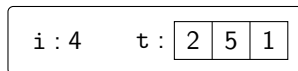
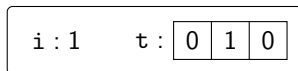
$$\left\{ \begin{array}{l} t[0] \mapsto 0 \\ t[1] \mapsto 10 \end{array} \right\}, \quad \left\{ \begin{array}{l} t[0] \mapsto 2 \\ t[1] \mapsto 11 \end{array} \right\}, \quad \left\{ \begin{array}{l} t[0] \mapsto 1 \\ t[1] \mapsto 12 \end{array} \right\}$$

- **Smashing abstraction** produces  $\{\{0, 10\}, \{2, 11\}, \{1, 12\}\}$
- After **non relational abstraction**, we obtain  $\&\bar{t} \mapsto \{0, 1, 2, 10, 11, 12\}$



# Array smashing abstraction example

We consider the following **set of concrete states**:



The **smashing abstraction** produces the following vectors:

$$\begin{array}{ll} (\{1\}, \{0, 1, 0\}) & (\{4\}, \{2, 5, 1\}) \\ (\{8\}, \{5, 8, 3\}) & (\{7\}, \{3, 6, 2\}) \end{array}$$

After **non relational abstraction**:

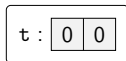
$$\begin{array}{ll} \&i & \longmapsto & \{1, 4, 8, 7\} \\ \&\bar{t} & \longmapsto & \{0, 1, 2, 3, 5, 6, 8\} \end{array}$$

After applying the **interval abstraction**, we get:  $([1, 8], [0, 8])$

## Post-condition for an assignment: example

<b>Assignment</b> $t[0] = 6$	<b>Pre-condition:</b> $t : \forall i, t[i] : [0, 0]$
------------------------------	--

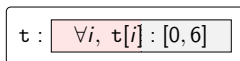
- **concrete pre-condition:**



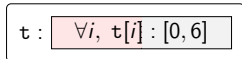
- **effect** of the assignment **in the concrete** and post-condition:



Thus, we obtain the **abstract post-condition:**

**Consequence:**

the analysis of  $t[0] = 6; t[1] = 6$ ; will also produce



**This is a another case of weak-update, resulting in significant precision loss**

# Weak-updates

## 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**

To summarize:

abstraction	$t[0] = \dots$	$t[[a, b]] = \dots$
element-wise	strong update	weak update
smashing	weak update	weak update

- relatively to the abstraction, a weak update may be precise (as in the examples)
- however, successions of weak updates will prevent from inferring invariants such as correctness of initialization

## Weak updates and strong updates: example

```

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

```

## Elementwise abstraction:

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

## Smashing abstraction:

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

- 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  $\{\bar{t}_0, \dots, \bar{t}_{k-1}\}$  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  $\{\bar{t}_0, \dots, \bar{t}_{k-1}\}$  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 Memory models
- 2 **Abstraction of arrays**
  - A micro language for manipulating arrays
  - Verifying safety of array operations
  - Abstraction of array contents
  - Abstraction of array properties
- 3 Basic pointer analyses
- 4 Three valued logic heap abstraction
- 5 Conclusion

# 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**

### Array initialization loop

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

### Hand proof sketch:

- **At iteration  $k$** ,  $i = k$  and the segment  $t[0], \dots, t[k - 1]$  is initialized
- **At the loop exit**,  $i = n$  and the whole array is initialized

**To complete the proof, we need to express properties on segments**

# Array segment properties

## Array initialization loop

```
// 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  $\in [1, 10]$

t	$\text{zero}_t(0, i - 1)$	$\top$
---	---------------------------	--------



# 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**:  $\text{zero}_t(i, j)$  iff  $t$  initialized to 0 between  $i$  and  $j$
- **Sortedness**:  $\text{sort}_t(i, j)$  iff  $t$  sorted between  $i$  and  $j$

## Examples:

- array satisfying  $\text{zero}_t(2, 6)$ :

$$i = 6$$

t	8	2	0	0	0	0	0	0	10	3
---	---	---	---	---	---	---	---	---	----	---

- array satisfying  $\text{sort}_t(1, 3)$  and  $\text{sort}_t(6, 8)$ :

$$i = 6$$

t	8	2	5	6	8	11	1	2	3	2
---	---	---	---	---	---	----	---	---	---	---

# Composing sortedness predicates

As part of the proof, predicates need be composed

$$\begin{aligned}
 \mathbf{zero}_t(i, j) \wedge \mathbf{zero}_{\bar{t}}(j + 1, k) &\Rightarrow \mathbf{zero}_t(i, k) \\
 t[j] = 0 &\Rightarrow \mathbf{zero}_t(j, j) \\
 \mathbf{zero}_t(i, j) \wedge t[j + 1] = 0 &\Rightarrow \mathbf{zero}_t(i, j + 1) \\
 \mathbf{sort}_t(i, j) \wedge \mathbf{sort}_{\bar{t}}(j + 1, k) &\not\Rightarrow \mathbf{sort}_t(i, k) \\
 t[j] \leq t[j + 1] \wedge \mathbf{sort}_t(i, j) \wedge \mathbf{sort}_{\bar{t}}(j + 1, k) &\Rightarrow \mathbf{sort}_t(i, k)
 \end{aligned}$$

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

$$\mathbf{sort}_t(0, 2) \wedge \mathbf{sort}_t(3, 5) \quad \text{but not} \quad \mathbf{sort}_t(0, 5)$$

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

$$B \leq C \wedge \mathbf{sort}_t(i, j, A, B) \wedge \mathbf{sort}_{\bar{t}}(j + 1, k, C, D) \Rightarrow \mathbf{sort}_t(i, k, A, D)$$

# 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

For instance, for an array of length  $n$ :

$$\begin{aligned} \mathbf{zero}_t(0, n-1) \wedge 0 \leq i < n &\xrightarrow{t[i]=?} \mathbf{zero}_t(0, i-1) \wedge \mathbf{zero}_t(i+1, n-1) \\ \top \wedge 0 \leq i < n &\xrightarrow{t[i]=0} \mathbf{zero}_t(i, i) \end{aligned}$$

## Abstract join operator: generalizes bounds

$$\begin{aligned} (\top \wedge i = 0 < n) \sqcup^\# (\mathbf{zero}_t(0, 0) \wedge i = 1 < n) \\ = (\mathbf{zero}_t(0, i-1) \wedge 0 \leq i < n) \end{aligned}$$

- this union introduces an empty initialized segment in the left hand side

## Array analysis: example

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

```
  t  i T
```

The diagram shows a horizontal rectangle representing an array. The letter 't' is to its left and 'i T' is to its right. Inside the rectangle, the letter 'T' is centered.

```
int i = 0;
```

```
  t  i T
```

The diagram shows a horizontal rectangle representing an array. The letter 't' is to its left and 'i T' is to its right. Inside the rectangle, the letter 'T' is centered.

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

```
  t  i T
```

The diagram shows a horizontal rectangle representing an array. The letter 't' is to its left and 'i T' is to its right. Inside the rectangle, the letter 'T' is centered.

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

```
  t  i T
```

The diagram shows a horizontal rectangle representing an array. The letter 't' is to its left and 'i T' is to its right. Inside the rectangle, the letter 'T' is centered.

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

```
  t  i T
```

The diagram shows a horizontal rectangle representing an array. The letter 't' is to its left and 'i T' is to its right. Inside the rectangle, the letter 'T' is centered.

```
}
```

```
  t  i T
```

The diagram shows a horizontal rectangle representing an array. The letter 't' is to its left and 'i T' is to its right. Inside the rectangle, the letter 'T' is centered.

## Array analysis: example

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

t	T	i	T
---	---	---	---

```
int i = 0;
```

t	T	i	[0, 0]
---	---	---	--------

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

t	T	i	T
---	---	---	---

```
  t[i] = 0;
```

t	T	i	T
---	---	---	---

```
  i = i + 1;
```

t	T	i	T
---	---	---	---

```
}
```

t	T	i	T
---	---	---	---

## Array analysis: example

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

t	T	i	T
---	---	---	---

```
int i = 0;
```

t	T	i	[0, 0]
---	---	---	--------

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

t	T	i	[0, 0]
---	---	---	--------

```
  t[i] = 0;
```

t	T	i	T
---	---	---	---

```
  i = i + 1;
```

t	T	i	T
---	---	---	---

```
}
```

t	T	i	T
---	---	---	---

## Array analysis: example

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

t	$\top$	i	$\top$
---	--------	---	--------

```
int i = 0;
```

t	$\top$	i	[0, 0]
---	--------	---	--------

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

t	$\top$	i	[0, 0]
---	--------	---	--------

```
  t[i] = 0;
```

t	$\text{zero}_{\bar{\epsilon}}(0, 0)$	i	[0, 0]
---	--------------------------------------	---	--------

```
  i = i + 1;
```

t	$\top$	i	$\top$
---	--------	---	--------

```
}
```

t	$\top$	i	$\top$
---	--------	---	--------

## Array analysis: example

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

t	T	i	T
---	---	---	---

```
int i = 0;
```

t	T	i	[0, 0]
---	---	---	--------

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

t	T	i	[0, 0]
---	---	---	--------

```
  t[i] = 0;
```

t	zero <sub>T</sub> (0, 0)	T	i	[0, 0]
---	--------------------------	---	---	--------

```
  i = i + 1;
```

t	zero <sub>T</sub> (0, 0)	T	i	[1, 1]
---	--------------------------	---	---	--------

```
}
```

t	T	i	T
---	---	---	---



## Array analysis: example

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

t	T	i	T
---	---	---	---

```
int i = 0;
```

t	zero <sub>t</sub> (0, i - 1)	T	i	[0, 1]
---	------------------------------	---	---	--------

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

t	T	i	[0, 0]
---	---	---	--------

```
  t[i] = 0;
```

t	zero <sub>t</sub> (0, 0)	T	i	[0, 0]
---	--------------------------	---	---	--------

```
  i = i + 1;
```

t	zero <sub>t</sub> (0, 0)	T	i	[1, 1]
---	--------------------------	---	---	--------

```
}
```

t	T	i	T
---	---	---	---

## Array analysis: example

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

t	T	i	T
---	---	---	---

```
int i = 0;
```

t	zero <sub>T</sub> (0, i - 1)	T	i	[0, 1]
---	------------------------------	---	---	--------

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

t	zero <sub>T</sub> (0, i - 1)	T	i	[0, 1]
---	------------------------------	---	---	--------

```
  t[i] = 0;
```

t	zero <sub>T</sub> (0, 0)	T	i	[0, 0]
---	--------------------------	---	---	--------

```
  i = i + 1;
```

t	zero <sub>T</sub> (0, 0)	T	i	[1, 1]
---	--------------------------	---	---	--------

```
}
```

t	T	i	T
---	---	---	---

## Array analysis: example

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

t	T	i	T
---	---	---	---

```
int i = 0;
```

t	$\text{zero}_{\bar{t}}(0, i - 1)$	T	i	[0, 1]
---	-----------------------------------	---	---	--------

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

t	$\text{zero}_{\bar{t}}(0, i - 1)$	T	i	[0, 1]
---	-----------------------------------	---	---	--------

```
  t[i] = 0;
```

t	$\text{zero}_{\bar{t}}(0, i)$	T	i	[0, 1]
---	-------------------------------	---	---	--------

```
  i = i + 1;
```

t	$\text{zero}_{\bar{t}}(0, 0)$	T	i	[1, 1]
---	-------------------------------	---	---	--------

```
}
```

t	T	i	T
---	---	---	---

## Array analysis: example

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

t	T	i	T
---	---	---	---

```
int i = 0;
```

t	zero <sub>T</sub> (0, i - 1)	T	i	[0, 1]
---	------------------------------	---	---	--------

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

t	zero <sub>T</sub> (0, i - 1)	T	i	[0, 1]
---	------------------------------	---	---	--------

```
  t[i] = 0;
```

t	zero <sub>T</sub> (0, i)	T	i	[0, 1]
---	--------------------------	---	---	--------

```
  i = i + 1;
```

t	zero <sub>T</sub> (0, i - 1)	T	i	[1, 2]
---	------------------------------	---	---	--------

```
}
```

t	T	i	T
---	---	---	---

## Array analysis: example

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

t	T	i	T
---	---	---	---

```
int i = 0;
```

t	zero <sub>t</sub> (0, i - 1)	T	i	[0, n]
---	------------------------------	---	---	--------

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

t	zero <sub>t</sub> (0, i - 1)	T	i	[0, 1]
---	------------------------------	---	---	--------

```
  t[i] = 0;
```

t	zero <sub>t</sub> (0, i)	T	i	[0, 1]
---	--------------------------	---	---	--------

```
  i = i + 1;
```

t	zero <sub>t</sub> (0, i - 1)	T	i	[1, 2]
---	------------------------------	---	---	--------

```
}
```

t	T	i	T
---	---	---	---

## Array analysis: example

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

t	T	i	T
---	---	---	---

```
int i = 0;
```

t	zero <sub><math>\bar{\epsilon}</math></sub> (0, i - 1)	T	i	[0, n]
---	--	---	---	--------

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

t	zero <sub><math>\bar{\epsilon}</math></sub> (0, i - 1)	T	i	[0, n - 1]
---	--	---	---	------------

```
  t[i] = 0;
```

t	zero <sub><math>\bar{\epsilon}</math></sub> (0, i)	T	i	[0, 1]
---	--	---	---	--------

```
  i = i + 1;
```

t	zero <sub><math>\bar{\epsilon}</math></sub> (0, i - 1)	T	i	[1, 2]
---	--	---	---	--------

```
}
```

t	T	i	T
---	---	---	---

## Array analysis: example

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

t	T	i	T
---	---	---	---

```
int i = 0;
```

t	zero <sub><math>\bar{\epsilon}</math></sub> (0, i - 1)	T	i	[0, n]
---	--	---	---	--------

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

t	zero <sub><math>\bar{\epsilon}</math></sub> (0, i - 1)	T	i	[0, n - 1]
---	--	---	---	------------

```
  t[i] = 0;
```

t	zero <sub><math>\bar{\epsilon}</math></sub> (0, i)	T	i	[0, n - 1]
---	--	---	---	------------

```
  i = i + 1;
```

t	zero <sub><math>\bar{\epsilon}</math></sub> (0, i - 1)	T	i	[1, 2]
---	--	---	---	--------

```
}
```

t	T	i	T
---	---	---	---

## Array analysis: example

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

t	T	i	T
---	---	---	---

```
int i = 0;
```

t	zero <sub><math>\bar{\epsilon}</math></sub> (0, i - 1)	T	i	[0, n]
---	--	---	---	--------

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

t	zero <sub><math>\bar{\epsilon}</math></sub> (0, i - 1)	T	i	[0, n - 1]
---	--	---	---	------------

```
  t[i] = 0;
```

t	zero <sub><math>\bar{\epsilon}</math></sub> (0, i)	T	i	[0, n - 1]
---	--	---	---	------------

```
  i = i + 1;
```

t	zero <sub><math>\bar{\epsilon}</math></sub> (0, i - 1)	T	i	[1, n]
---	--	---	---	--------

```
}
```

t	T	i	T
---	---	---	---



## Array analysis: example

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

t	T	i	T
---	---	---	---

```
int i = 0;
```

t	$\text{zero}_{\bar{t}}(0, i - 1)$	T	i	$[0, n]$
---	-----------------------------------	---	---	----------

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

t	$\text{zero}_{\bar{t}}(0, i - 1)$	T	i	$[0, n - 1]$
---	-----------------------------------	---	---	--------------

```
  t[i] = 0;
```

t	$\text{zero}_{\bar{t}}(0, i)$	T	i	$[0, n - 1]$
---	-------------------------------	---	---	--------------

```
  i = i + 1;
```

t	$\text{zero}_{\bar{t}}(0, i - 1)$	T	i	$[1, n]$
---	-----------------------------------	---	---	----------

```
}
```

t	$\text{zero}_{\bar{t}}(0, n - 1)$		i	$[n, n]$
---	-----------------------------------	--	---	----------

# Partitioning of arrays

## Array partitions

A **partition** of an array  $t$  of length  $n$  is a **sequence**  $\mathcal{P} = \{e_0, \dots, e_k\}$  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**:

$$\left\{ \begin{array}{ll} i = 1 & [0, 4, 1, 2, 3, 5] \\ i = 2 & [0, 1, 5, 2, 3, 4] \end{array} \right. \quad \begin{array}{ll} i = 3 & [2, 2, 4, 5, 1, 8] \\ i = 5 & [0, 2, 4, 6, 7, 9] \end{array}$$

- **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  $\mathcal{P} = \{e_0, \dots, e_k\}$  and a set of abstract properties  $\{P_0, \dots, P_{k-1}\}$ .

Its concretization is the set of memory states  $m = (e, \hat{h})$  such that

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

- **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 Memory models
- 2 Abstraction of arrays
- 3 Basic pointer analyses**
- 4 Three valued logic heap abstraction
- 5 Conclusion

# Programs with pointers: syntax

**Syntax extension:** quite a few additional constructions

<b>l</b>	::=	<b>l-valules</b>	
		x	( $x \in \mathbb{X}$ )
		...	
		*e	pointer dereference
		l · f	field read
<b>e</b>	::=	<b>expressions</b>	
		l	
		...	
		&l	"address of" operator
<b>s</b>	::=	<b>statements</b>	
		...	
		x = malloc(c)	allocation of c bytes
		free(x)	deallocation of the block pointed to by x

We do not consider **pointer arithmetics here**

# Programs with pointers: semantics

## Case of l-values:

$$\begin{aligned} \llbracket \mathbf{x} \rrbracket(e, h) &= e(\mathbf{x}) \\ \llbracket *e \rrbracket(e, h) &= \begin{cases} h(\llbracket e \rrbracket(e, h)) & \text{if } \llbracket e \rrbracket(e, h) \neq 0 \wedge \llbracket e \rrbracket(e, h) \in \mathbf{Dom}(h) \\ \Omega & \text{otherwise} \end{cases} \\ \llbracket \mathbf{l} \cdot \mathbf{f} \rrbracket(e, h) &= \llbracket \mathbf{l} \rrbracket(e, h) + \mathbf{offset}(\mathbf{f}) \text{ (numeric offset)} \end{aligned}$$

## Case of expressions:

$$\begin{aligned} \llbracket \mathbf{l} \rrbracket(e, h) &= h(\llbracket \mathbf{l} \rrbracket(e, h)) && \text{(evaluates into the contents)} \\ \llbracket \&\mathbf{l} \rrbracket(e, h) &= \llbracket \mathbf{l} \rrbracket(e, h) && \text{(evaluates into the address)} \end{aligned}$$

## Case of statements:

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

# Pointer non relational abstractions

We rely on the **non relational abstraction of heterogeneous states** that was introduced earlier, with a few changes:

- $\mathbb{V} = \mathbb{V}_{\text{addr}} \uplus \mathbb{V}_{\text{int}}$
- $\mathbb{X} = \mathbb{X}_{\text{addr}} \uplus \mathbb{X}_{\text{int}} \uplus \dots$
- **concrete memory cells** now include **structure fields**, and fields of **dynamically allocated regions**
- **abstract cells**  $\mathbb{C}^\#$  finitely summarize concrete cells
- we apply a **non relational abstraction** to pointer locations, based on  $\mathbb{D}_{\text{ptr}}^\#$  and  $\gamma_{\text{ptr}} : \mathbb{D}_{\text{ptr}}^\# \rightarrow \mathcal{P}(\mathbb{V}_{\text{addr}})$  (other location abstracted in the same way as before, e.g., non relationally)

We will see **several instances** of this kind of abstraction

# Pointer non relational abstraction: null pointers

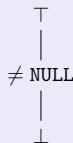
**The dereference of a null pointer will cause a crash**

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

## Null pointer analysis

### Abstract domain for addresses:

- $\gamma_{\text{ptr}}(\perp) = \emptyset$
- $\gamma_{\text{ptr}}(\top) = \mathbb{V}_{\text{addr}}$
- $\gamma_{\text{ptr}}(\neq \text{NULL}) = \mathbb{V}_{\text{addr}} \setminus \{0\}$



- 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

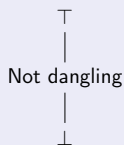
**The dereference of a null pointer will cause a crash**

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

## Null pointer analysis

### Abstract domain for addresses:

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



- very **lightweight**, can typically resolve rather trivial cases
- useful for **C**, useless for Java (initialization requirement + GC)

# Pointer non relational abstraction: points-to sets

## 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**  $\mathbb{A}^\#$  is fixed:

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

- **Concrete addresses** are **abstracted into**  $\mathbb{A}^\#$  by  $\phi_{\mathbb{A}} : \mathbb{A} \rightarrow \mathbb{A}^\# \uplus \{\top\}$
- A pointer value is abstracted by the abstraction of the addresses it may point to, *i.e.*,  $\mathbb{D}_{\text{ptr}}^\# = \mathcal{P}(\mathbb{A}^\#)$   
and  $\gamma_{\text{ptr}}(a^\#) = \{a \in \mathbb{A} \mid \phi_{\mathbb{A}}(a) = a^\#\}$

- **example:**  $p$  may point to  $\{\&x\}$

# Points-to sets computation example

## Example code:

```

1: int x, y;
2: int * p;
3: y = 9;
4: p = &x;
5: *p = 0;
6: ...

```

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

Analysis results:

	$\&x$	$\&y$	$\&p$
1	$\top$	$\top$	$\top$
2	$\top$	$\top$	$\top$
3	$\top$	$\top$	$\top$
4	$\top$	$[9, 9]$	$\top$
5	$\top$	$[9, 9]$	$\{\&x\}$
6	$[0, 0]$	$[9, 9]$	$\{\&x\}$

## 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:**  $\{\&x, \&y, \&p\}$

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

## Imprecise results

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

# Weak-updates

As in array analysis, we encounter:

## 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**
- if the points-to set may contain **more than one element**, the analysis needs to perform a **weak-update**

# 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**:  $\{\{p_0, p_1, \&x\}, \{p_2, p_3\}, \dots\}$ )
- **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

## Weak updates:

- **imprecision in updates** that spread out as soon as points-to set contain several elements
- impact **client analyses** severely (as for array analyses)

## 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

**Shape abstraction:**  
**We will use similar ideas as for array segment analyses**

# Outline

- 1 Memory models
- 2 Abstraction of arrays
- 3 Basic pointer analyses
- 4 Three valued logic heap abstraction**
  - Building an abstract domain
  - Weakening abstract elements
  - Computation of transfer functions
- 5 Conclusion



# An abstract representation of memory states: shape graphs

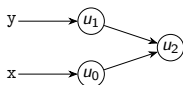
## Goal of the static analysis

Infer structural invariants of programs using unbounded heap

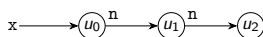
Observation: representation of memory states by shape graphs

- **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:



A list of length 2:



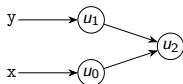
⇒ We need to over-approximate sets of shape graphs

# Shape graphs and their representation

## Description with predicates

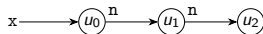
- **Boolean encoding:** nodes are atoms  $u_0, u_1, \dots$
- **Predicates over atoms:**
  - ▶  $x(u)$ : variable  $x$  contains the address of  $u$
  - ▶  $n(u, v)$ : field of  $u$  points to  $v$
- **Truth values:** traditionally noted 0 and 1 in the TVLA litterature

### Two alias pointers:



	x	y	$\mapsto$	$u_0$	$u_1$	$u_2$
$u_0$	1	0	$u_0$	0	0	1
$u_1$	0	1	$u_1$	0	0	1
$u_2$	0	0	$u_2$	0	0	0

### A list of length 2:

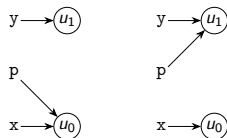


	x	$\cdot n \mapsto$	$u_0$	$u_1$	$u_2$
$u_0$	1	$u_0$	0	1	0
$u_1$	0	$u_1$	0	0	1
$u_2$	0	$u_2$	0	0	0

# Unknown value: three valued logic

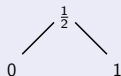
**How to abstract away some information ?** *i.e.*,  
to abstract several graphs into one ?

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

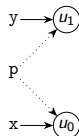


## A boolean lattice

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



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



# Summary nodes

At this point, we cannot talk about **unbounded memory states** with **finitely many** nodes

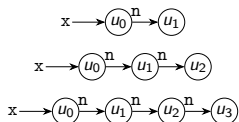
## An idea

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

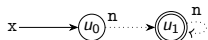
## Definition: summary node

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

### Lists of lengths 1, 2, 3:



Attempt at a **summary** graph:



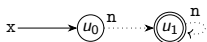
- Edges to  $u_1$  are dotted

# 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



We need to **add more information**, e.g., about **connectedness**

“Is shared”

$\text{sh}(u)$  if and only if

$$\exists v_0, v_1, \begin{cases} v_0 \neq v_1 \\ \wedge n(v_0, u) \\ \wedge n(v_1, u) \end{cases}$$

Predicates defined by transitive closure

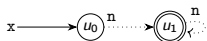
- **Reachability**:  $\underline{r}(u, v)$  if and only if  $u = v \vee \exists u_0, n(u, u_0) \wedge \underline{r}(u_0, v)$
- **Acyclicity**:  $\underline{\text{acy}}(v)$   
similar, with a negation

# Three structures

## Definition: 3-structures

A 3-structure is a tuple  $(\mathcal{U}, \mathcal{P}, \phi)$ :

- a set  $\mathcal{U} = \{u_0, u_1, \dots, u_m\}$  of **atoms**
- a set  $\mathcal{P} = \{p_0, p_1, \dots, p_n\}$  of **predicates**  
(we write  $k_i$  for the arity of predicate  $p_i$ )
- a **truth table**  $\phi$  such that  $\phi(p_i, u_{l_1}, \dots, u_{l_{k_i}})$  denotes the truth value of  $p_i$  for  $u_{l_1}, \dots, u_{l_{k_i}}$   
note: truth values are elements of the lattice  $\{0, \frac{1}{2}, 1\}$



$$\left\{ \begin{array}{l} \mathcal{U} = \{u_0, u_1\} \\ \mathcal{P} = \{\underline{x}(\cdot), \underline{n}(\cdot, \cdot), \underline{\text{sum}}(\cdot)\} \end{array} \right.$$

	x	<u>sum</u>	n	$u_0$	$u_1$
$u_0$	1	0	$u_0$	0	1
$u_1$	0	$\frac{1}{2}$	$u_1$	0	0

**In the following we build up an abstract domain of 3-structures**

# Embedding

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

## The embedding principle

Let  $\mathcal{S}_0 = (\mathcal{U}_0, \mathcal{P}, \phi_0)$  and  $\mathcal{S}_1 = (\mathcal{U}_1, \mathcal{P}, \phi_1)$  be two three structures, with the same sets of predicates. Let  $f : \mathcal{U}_0 \rightarrow \mathcal{U}_1$ , surjective.

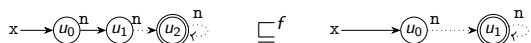
We say that  $f$  **embeds**  $\mathcal{S}_0$  **into**  $\mathcal{S}_1$  iff

for all predicate  $p \in \mathcal{P}$  or arity  $k$ , for all  $u_{l_1}, \dots, u_{l_{k_i}} \in \mathcal{U}_0$ ,  
 $\phi_0(u_{l_1}, \dots, u_{l_{k_i}}) \sqsubseteq \phi_1(f(u_{l_1}), \dots, f(u_{l_{k_i}}))$

Then, **we write**  $\mathcal{S}_0 \sqsubseteq^f \mathcal{S}_1$

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

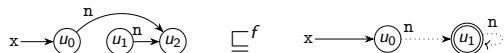
## Embedding examples



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  $f : u_0 \mapsto u_0; u_1 \mapsto u_1; u_2 \mapsto u_1$

- **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

A 3-structure  $(\mathcal{U}, \mathcal{P}, \phi)$  is a **2-structure**, if and only if  $\phi$  always returns in  $\{0, 1\}$

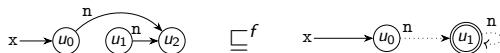
- A **2-structure** defines a set of **concrete memory states**  $(e, h)$  obtained by mapping symbols to addresses, that are **compatible with the predicates** of the structure
- We let **stores** $(\mathcal{S})$  denote the stores corresponding to 2-structure  $\mathcal{S}$

### Concretization of a 3-structure

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

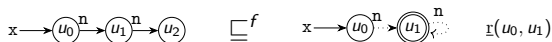
# Concretization examples

## Without reachability:



where  $f : u_0 \mapsto u_0; u_1 \mapsto u_1; u_2 \mapsto u_1; u_3 \mapsto u_1$

## With reachability:



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 ?

- **Concrete states** correspond to **2-structures**
- The **analysis** should track **3-structures**, thus the analysis correctness should **rely on the embedding relation**

### Embedding theorem

- Let  $\mathcal{S}_0 = (\mathcal{U}_0, \mathcal{P}, \phi_0)$  and  $\mathcal{S}_1 = (\mathcal{U}_1, \mathcal{P}, \phi_1)$  be two three structures, with the same sets of predicates
- Let  $f : \mathcal{U}_0 \rightarrow \mathcal{U}_1$ , such that  $\mathcal{S}_0 \sqsubseteq^f \mathcal{S}_1$
- Let  $\Psi$  be a logical formula, with variables in  $X$
- Let  $g : X \rightarrow \mathcal{U}_0$  be an assignment for the variables of  $\Psi$

Then,

$$\llbracket \Psi \upharpoonright_g \rrbracket(\mathcal{S}_0) \sqsubseteq \llbracket \Psi \upharpoonright_{f \circ g} \rrbracket(\mathcal{S}_1)$$

# Principle for the design of sound transfer functions

## Transfer functions for static analysis

- **Semantics of concrete statements encoded into boolean formulas**
- **Evaluation in the abstract is sound (embedding theorem)**

### Example:

analysis of an assignment  $y := x$

- 1 let  $y'$  denote the *new* value of  $y$
- 2 add the constraint  $y'(u) = x(u)$
- 3 rename  $y'$  into  $y$

### Advantages:

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

# A powerset abstraction

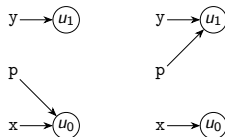
- Do 3-structures allow for a **sufficient level of precision** ?
- How to **over-approximate a set of 2-structures** ?

```

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

```

After the **if** statement:  
 abstracting would be imprecise



## Powerset abstraction

- Shape analyzers usually rely on a **powerset abstract domain** based on **disjunctive completion** of a finite lattice  
*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  $\mathbf{can} : \mathcal{L} \rightarrow \mathcal{L}'$ :

- operator **can** is called **canonicalization** if and only if it defines an **upper closure operator**
- then it defines a **canonicalization operator**  $\mathbf{can} : \mathcal{P}(\mathcal{L}) \rightarrow \mathcal{P}(\mathcal{L}')$ :

$$\mathbf{can}(\mathcal{E}) = \{\mathbf{can}(x) \mid x \in \mathcal{E}\}$$

To make the powerset domain work, we simply need a **can** over 3-structures

## A canonicalization over 3-structures

- We assume there are  $n$  variables  $x_1, \dots, x_n$   
**Thus the number of unary predicates is finite**
- **Sub-lattice**: structures with atoms **distinguished by the values of the unary predicates** (or *abstraction predicates*)  $x_1, \dots, x_n$

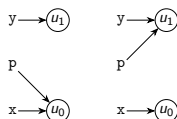
# Canonical abstraction

We assume the analysis relies on unary predicates for canonicalization. The analysis design may choose another set of predicates than the unary predicates for the sub-lattice representation

## Canonical abstraction by truth blurring

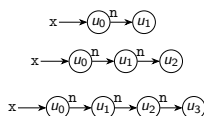
- 1 **Identify** nodes that **have different abstraction predicates**
- 2 When several nodes have the **same abstraction predicate** **introduce a summary node**
- 3 **Compute new predicate values** by doing a **join over truth values**

### Elements not merged:

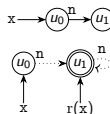


### Elements merged:

#### Lists of lengths 1, 2, 3:



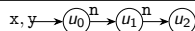
#### Abstract into:



## Assignment: a simple case

**Statement**  $l_0 : y = y \rightarrow n; l_1 : \dots$

**Pre-condition**  $\mathcal{S}$

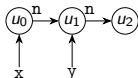


### Transfer function computation:

- It should produce an over-approximation of  $\{m_1 \in \mathbb{M} \mid (l_0, m_0) \rightarrow (l_1, m_1)\}$
- **Encoding** using “**primed predicates**” to denote predicates **after** the evaluation of the assignment, to evaluate them in the same structure (non primed variables are removed afterwards and primed variables renamed):

$$\begin{aligned} \mathbf{x}'(u) &= \mathbf{x}(u) \\ \mathbf{y}'(u) &= \exists v, \mathbf{y}(v) \wedge \mathbf{n}(v, u) \\ \mathbf{n}'(u, v) &= \mathbf{n}(u, v) \end{aligned}$$

- **Result:**



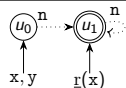
This is exactly the expected result



## Assignment: a more involved case

**Statement**  $l_0 : y = y \rightarrow n; l_1 : \dots$

**Pre-condition**  $\mathcal{S}$



- Let us try to **resolve the update in the same way as before**:

$$\begin{aligned} x'(u) &= x(u) \\ y'(u) &= \exists v, y(v) \wedge n(v, u) \\ n'(u, v) &= n(u, v) \end{aligned}$$

- 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

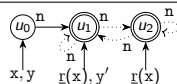
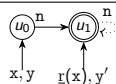
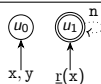
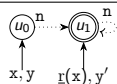
We assume a 3-structure  $\mathcal{S}$  and a boolean formula  $f$  are given, we call a **focusing**  $\mathcal{S}$  on  $f$  the generation of a set  $\hat{\mathcal{S}}$  such that:

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

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

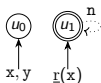
**Example:** focusing on  
 $y'(u) = \exists v, y(v)$   
 $\wedge n(v, u)$

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

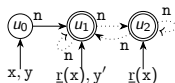


## Focus and coerce

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



$u_1$  is reachable from  $x$ , but there is no sequence of  $n$  fields: this structure has **empty concretization**

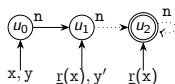
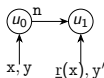


$u_0$  has an  $n$ -field to  $u_1$  so  $u_1$  denotes a unique atom and **cannot be a summary node**

## Coerce operation

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

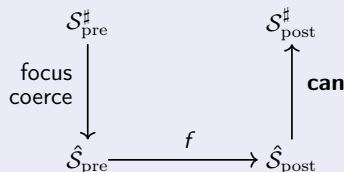
**Result:**



## Focus, transfer, abstract...

## Computation of a transfer function

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



**Soundness proof** steps:

- ① **sound encoding of the semantics of program statements into formulas** (typically, no loss of precision at this stage)
- ② **focusing** produces a **refined** over-approximation (disjunction)
- ③ **canonicalization over-approximates graphs** (truth blurring)

**A common picture in shape analysis**

# Outline

- 1 Memory models
- 2 Abstraction of arrays
- 3 Basic pointer analyses
- 4 Three valued logic heap abstraction
- 5 Conclusion**

Summarization: one abstract cell, many concrete cells

**Large / unbounded numbers of concrete cells need to be abstracted**

- **Array blocks** may have large number of elements
- **Dynamic memory allocation** functions may be called an unbounded number of times

## Summary abstract cell

A **summary abstract cell** describes **several concrete cells**.

A **summary abstract variable** describes **several concrete values**.

- **Formalization** based on a function mapping **concrete cells** into the **abstract cells** that represent them:

$$\phi_{\mathbb{A}} : \mathbb{A} \rightarrow \mathbb{A}^{\#}$$

- Analysis operations should reason on abstract states **up-to**  $\phi_{\mathbb{A}}$

# Updates: weak vs strong

**Memory updates may be very imprecise**

## Several typical cases:

- 1 update to a cell **that cannot be determined precisely**  
*i.e.*, affecting an abstract cell among  $A^\# \subseteq \mathbb{A}^\#$ , where  $|A^\#| > 1$
- 2 update to a **summary cell**

In those cases, the abstract update **joins previous values and new values**

## Weak updates

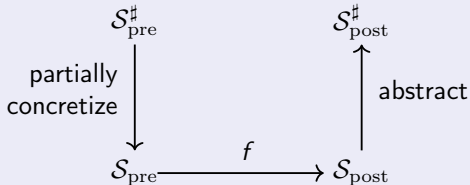
- **The modified concrete cell cannot be mapped into a well identified abstract cell**
- The resulting abstract information is obtained by **joining the new value and the old information**

# Concretize partially, update, abstract

## Summaries can be refined locally for better precision

- **Array segment predicates** can be split into predicates over smaller segments for abstract transfer functions
- The information over **TVLA summary nodes** can be refined using disjunctions for the computation of abstract post-conditions

## A scheme to compute more precise post-conditions





# Bibliography

- [HP'08]: **Discovering properties about arrays in simple programs.**  
Nicolas Halbwachs, Mathias Péron. In PLDI'08, pages 339-348, 2008.
- [CCL'11]: **A parametric segmentation functor for fully automatic and scalable array content analysis.**  
Patrick Cousot, Radhia Cousot, Francesco Logozzo. In POPL'11, pages 105-118, 2011.
- [AD'94]: **Interprocedural may alias analysis for pointers: beyond  $k$ -limiting.**  
Alain Deutsch. In PLDI'94, pages 230-241, 1994.
- [SRW'99]: **Parametric Shape Analysis via 3-Valued Logic.**  
Shmuel Sagiv, Thomas W. Reps et Reinhard Wilhelm. In POPL'99, pages 105-118, 1999.