

# Memory abstraction 1

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

Xavier Rival

INRIA, ENS, CNRS

Jan, 10th. 2021

# Overview of the lecture

So far, we have shown **numerical abstract domains**

- non relational: intervals, congruences...
- relational: polyhedra, octagons, ellipsoids...

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

⇒ **a very broad topic**, and two lectures:

## This lecture

- **overview memory models and memory properties**
- non relational **pointer structures abstraction**
- **predicates based shape abstraction**

**Next lecture:** separation logic and shape abstraction, shape/numerical abstraction

# Outline

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

- 2 Pointer Abstractions

- 3 Shape analysis in Three-Valued Logic (TVL)

- 4 Conclusion

# Assumptions for the two lectures on memory abstraction

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:  $\mathbb{M} = \mathbb{X} \rightarrow \mathbb{V}$ ) to change
- **error state**:  $\Omega$
- **states**:  $\mathbb{S}$

$$\mathbb{S} = \mathbb{L} \times \mathbb{M}$$

$$\mathbb{S}_\Omega = \mathbb{S} \uplus \{\Omega\}$$

- **transition relation**:

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

**Abstraction** of sets of states

- **abstract domain**  $\mathbb{D}^\#$
- **concretization**  $\gamma : (\mathbb{D}^\#, \sqsubseteq^\#) \longrightarrow (\mathcal{P}(\mathbb{S}), \subseteq)$

# Assumptions: syntax of programs

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

<b>l</b>	<b>::= l-values</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>	(lvalue)
	<b><math>e \oplus e</math></b>	(arith operation, comparison)
<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)

# Assumptions: semantics of programs

We assume **classical definitions for:**

- **I-values:**  $\llbracket 1 \rrbracket : \mathbb{M} \rightarrow \mathbb{X}$  → to change
- **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 transitions** ( $\rightarrow$ )

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

## Reachable states semantics

The reachable states are computed as  $\llbracket S \rrbracket_{\mathcal{R}} = \text{lfp } F$  where

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

and  $\mathbb{S}_I$  denotes the set of initial states.

$$= \{\ell_0\} \times \mathbb{M} \rightarrow \ell_0 : \text{entry point}$$

# Assumptions: general form of the abstraction

We assume an **abstraction for sets of memory states**:

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

*↖ context sensitivity*

## 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  $\mathbb{D}_{\text{mem}}^{\#}, \gamma_{\text{mem}} \dots$**

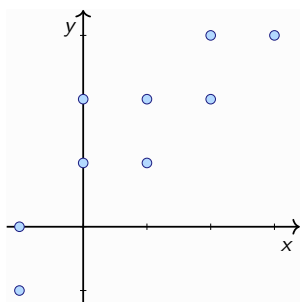
- previous lectures:
  - $\mathbb{X}$  is fixed and finite and,  $\mathbb{V}$  is scalars (integers or floats), thus,  $\mathbb{M} \equiv \mathbb{V}^n$
- today:
  - we will **extend the language** thus, also **need to extend**  $\mathbb{D}_{\text{mem}}^{\#}, \gamma_{\text{mem}}$

# Abstraction of purely numeric memory states

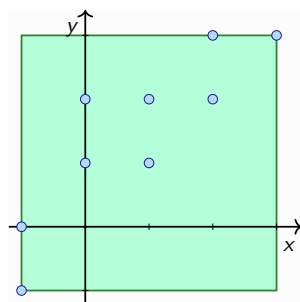
## Purely numeric case

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

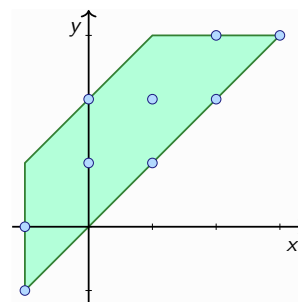
**Example:**  $N = 2$ ,  $\mathbb{X} = \{x, y\}$



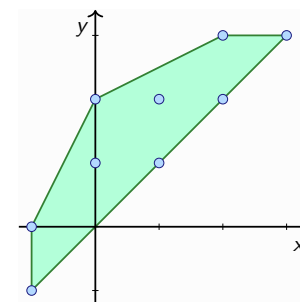
concrete set



interval domain



octagon domain



polyhedra 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...** *records...*

$\mathcal{E}_0$ : scalars  $\rightarrow$  integers  
 $\mathcal{E}_1$ : pointers

## Heterogeneous memory states and non relational abstraction

- **types**  $t_0, t_1, \dots$  and **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**:  $\mathbb{M} = \mathbb{X}_{t_0} \rightarrow \mathbb{V}_{t_0} \times \mathbb{X}_{t_1} \rightarrow \mathbb{V}_{t_1} \dots$

$\mathcal{E}_0$        $\mathcal{E}_1$

**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)

$$\gamma_{nr} : \mathcal{P}(\mathbb{M}_0) \times \mathcal{P}(\mathbb{M}_1) \longrightarrow \mathcal{P}(\mathbb{M})$$

$$(m_0^\#, m_1^\#) \longmapsto \{(m_0, m_1) \mid \forall i, m_i \in \gamma_i(m_i^\#)\}$$

*naive*

# Memory structures

## Common structures (non exhaustive list)

- **Structures, records, tuples:**  
sequences of cells accessed with fields
  - **Arrays:**  
similar to structures; indexes are integers in  $[0, n - 1]$
  - **Pointers:** *+ dynamic allocation*  
numerical values corresponding to the address of a memory cell
  - **Strings and buffers:**  
blocks with a sequence of elements and a terminating element (e.g.,  $0x0$ )
  - **Closures** (functional languages):  
pointer to function code and (partial) list of arguments
- no fixed cells set of*

To describe memories, the definition  $M = X \rightarrow V$  is **too restrictive**

**Generally, non relational, heterogeneous abstraction cannot handle many such structures all at once: relations are needed!**

# Specific properties to verify

## Memory safety

Absence of memory errors (crashes, or undefined behaviors)

### Pointer errors:

- Dereference of a **null pointer** / of an **invalid pointer**

### Access errors:

- **Out of bounds** array access, **buffer overruns** (often used for attacks)

## Invariance properties

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

### Examples:

- **Preservation of structures**, e.g., **lists should remain connected**
- **Preservation of invariants**, e.g., **of balanced trees**

# Properties to verify: examples

## A program closing a list of file descriptors

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

## Correctness properties

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

## Examples of structure preservation properties

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

## A more realistic model

$$M = X \rightarrow V, \text{ inadequate}$$

### No one-to-one relation between memory cells and program variables

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

### Environment + Heap

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

$$\mathbb{E} = X \rightarrow V_{\text{addr}}$$

$$\mathbb{H} = V_{\text{addr}} \rightarrow V$$

$h$  is actually only a partial function

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

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

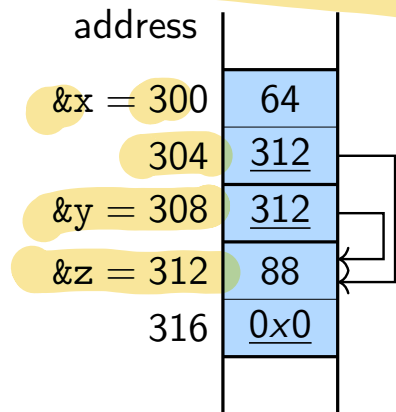
# Example of a concrete memory state (variables)

## Example setup:

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

## Memory layout

(pointer values underlined)



$e :$

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

$f :$

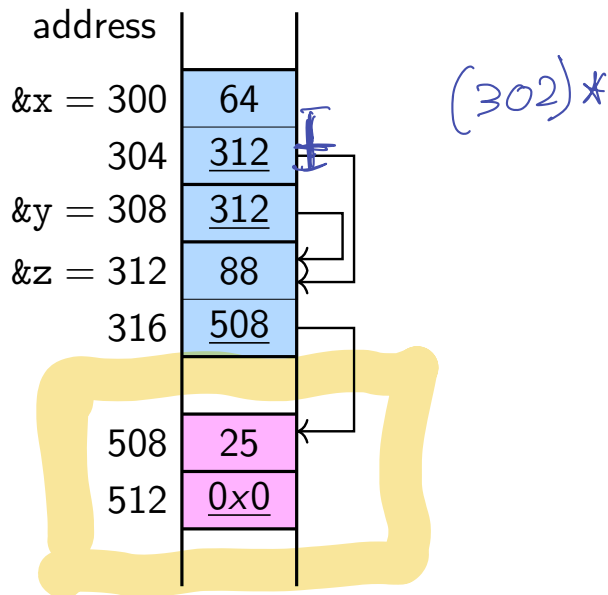
300	$\mapsto$	64
304	$\mapsto$	312
308	$\mapsto$	312
312	$\mapsto$	88
316	$\mapsto$	0

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

## Example setup:

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

## Memory layout



<i>e</i> :	x	↦	300
	y	↦	308
	z	↦	312
<i>h</i> :	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:

- **Addresses are values:**  $\mathbb{V}_{\text{addr}} \subseteq \mathbb{V}$

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

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

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

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

where

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



# 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 such concrete memory states:  
see **CompCert** project source files (Coq formalization)

# Language semantics: program crash

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

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

## Error state

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

## 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

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

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

## Modeling of undefined behavior

- Very hard to capture what a program operation may modify
- Abnormal situation at  $(\ell_0, m_0)$  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

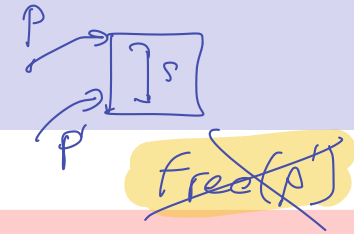
### Arithmetics on pointers, requiring a more precise model

- **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

# Manual memory management

$(t) \text{ malloc } (s)$

$\text{free}(p)$



## Allocation of unbounded memory space

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

### OCaml memory management

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

### C memory management

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

**Manual memory management** is not safe:

- **memory leaks**: growing unreachable memory region; memory exhaustion
- **dangling pointers** if freeing a block that is still referred to

# Summary on the memory model

## Language dependent items

- **Clear error cases** or **undefined behaviors**  
for analysis, a semantics with clear error cases is preferable
- **Composite objects**: structure fully exposed or not
- **Pointers to 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 study specific features one by one and in isolation from the others



# Rest of these two lectures

## Abstraction for pointers and dynamic data-structures:

- **pointer abstractions** → short, simple techniques
- **three-valued logic**-based abstraction for **dynamic structures**
- **separation logic**-based abstraction for **dynamic structures**
- **combination** of **value** and **structure** abstractions

## Abstract operations:

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

# Outline

- 1 Memory models
- 2 **Pointer Abstractions**
- 3 Shape analysis in Three-Valued Logic (TVL)
- 4 Conclusion

# Programs with pointers: syntax

**Syntax extension:** we add pointer operations

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

We do not consider **pointer arithmetics here**

# Programs with pointers: semantics

## Case of l-values:

env  
heap

$e \in \mathbb{E} ::= X \rightarrow \mathbb{V}_{\text{addr}}$

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

$$\llbracket x \rrbracket(e, h) = e(x)$$

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

$$\llbracket l \cdot f \rrbracket(e, h) = \llbracket l \rrbracket(e, h) + \mathbf{offset}(f) \text{ (numeric offset)}$$

## Case of expressions:

$\llbracket e \rrbracket : \mathbb{M} \rightarrow \mathbb{V}$

$$\llbracket l \rrbracket(e, h) = h(\llbracket l \rrbracket(e, h)) \quad \text{(evaluates into the contents)}$$

$$\llbracket \&l \rrbracket(e, h) = \llbracket l \rrbracket(e, h) \quad \text{(evaluates into the address)}$$

## Case of statements:

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

- we let  $\mathbb{V} = \mathbb{V}_{\text{addr}} \uplus \mathbb{V}_{\text{int}}$  and  $\mathbb{X} = \mathbb{X}_{\text{addr}} \uplus \mathbb{X}_{\text{int}}$
- **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**:

## Non relational pointer abstraction

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

We will see **several instances** of this kind of abstraction, and show how such abstraction lift into abstraction for sets of heaps

# Pointer non relational abstraction: null pointers

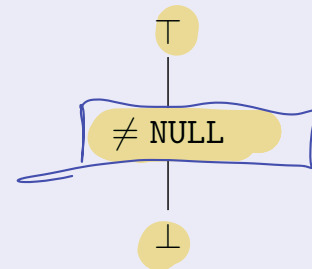
**The dereference of a null pointer will cause a crash**

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

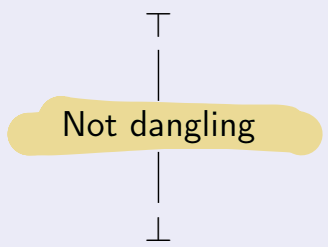
**The dereference of a null pointer will cause a crash**

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

## Null pointer analysis

### Abstract domain for addresses:

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



Not dangling

- 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

*Finite*

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

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

where  $a_0^\#, \dots, a_N^\#$  is a collection of  $N + 1$  fixed abstract addresses

- **Concrete addresses** are **abstracted into**  $\mathbb{A}^\#$  by  $\phi_{\mathbb{A}} : \mathbb{V}_{\text{addr}} \rightarrow \mathbb{A}^\# \uplus \{\top\}$

Assumption:  $\phi_{\mathbb{A}}$  surjective (no useless abstract address).

- 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

$$\gamma_{\text{ptr}}(a^\#) = \{a \in \mathbb{V}_{\text{addr}} \mid \phi_{\mathbb{A}}(a) = a^\#\}$$



# Abstraction of pointer states

We consider all values are of pointer type, *i.e.*, heaps are of the form  $h : \mathbb{V}_{\text{addr}} \rightarrow \mathbb{V}_{\text{addr}}$ .

## Intuition:

- collect information separately for each element of  $\mathbb{A}^\#$
- use a pointer value abstract element for each abstract address

## Lifting a pointer abstraction to heap abstraction

We let  $\mathbb{D}_{\text{mem}}^\# = \mathbb{A}^\# \rightarrow \mathbb{D}_{\text{ptr}}^\#$  and define

$$\gamma_{\text{mem}}(h^\#) = \{h \in \mathbb{H} \mid \forall a \in \mathbb{V}_{\text{addr}}, \forall a^\# \in \mathbb{A}^\#, \phi_{\mathbb{A}}(a) = a^\# \implies \phi_{\mathbb{A}}(h(a)) \in \gamma_{\text{ptr}}(h^\#(a^\#))\}$$

**Examples** of properties described by this abstraction:

- $p$  may point to  $\{\&x\}$
- $p$  points to some address described by  $a^\#$  and, at all addresses described by  $a^\#$ , we can read another address described by  $a^\#$

# 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:

*intervals*  
↗ ↘

	$\&x$	$\&y$	$\&p$
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: ...

```

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

$[-10, -5] \not\subseteq [-10, 0]$

Case 1 :  $p :: \&x$   
 $y \in [5, 10]$      $x = 0$

Case 2 :  $p :: \&y$   
 $y = 0$      $x \in [-10, -5]$

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

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

## Imprecise results

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

# Weak updates

We can formalize this imprecision a bit more:

## Weak updates

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

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

- if the points-to set contains **exactly one element**, the analysis can perform a **strong update**  
as in the first example:  $p \mapsto \{\&x\}$
- if the points-to set may contain **more than one element**, the analysis needs to perform a **weak-update**  
as in the second example:  $p \mapsto \{\&x, \&y\}$

# Weak updates

We recall:

- $\mathbb{A}^\# = \{\&x, \&y, \dots, \&t, a_0^\#, a_1^\#, \dots, a_N^\#\}$
- $\phi_{\mathbb{A}} : \mathbb{V}_{\text{addr}} \rightarrow \mathbb{A}^\# \uplus \{\top\}$ , surjective

Moreover, we assume an abstract state  $h^\#$  and an assignment  $l := c$  where  $l$  is an l-value. We note the abstract evaluation of the l-value:

$$\mathcal{L} ::= \phi_{\mathbb{A}}^{-1}(\llbracket l \rrbracket^\#(h^\#)) = \{a \in \mathbb{A}^\# \mid \phi_{\mathbb{A}}(a) \in \llbracket l \rrbracket^\#(h^\#)\}$$

We have **two cases**, based on **the cardinality of  $\mathcal{L}$** :

- 1  $|\mathcal{L}| \leq 1$ : *strong update*  
then, exactly one abstract value needs to be updated ( $\phi_{\mathbb{A}}(a)$  if  $\mathcal{L} = \{a\}$ )

- 2  $|\mathcal{L}| > 1$ :  
then, there exists two distinct addresses  $a_0, a_1 \in \mathcal{L}$ ; since the assignment **overwrites one cell exactly**:

- ▶ if the content of  $a_0$  is modified, then that of  $a_1$  stays the same...
- ▶ the other way around too, of course

**thus the post-condition need to map  $\phi_{\mathbb{A}}(a_0)$  to something weaker than  $h^\#(a_0)$ , and the same for  $a_1$ , which means we have a weak update**

# Weak updates

We consider:

- abstract heap  $h^\#$
- assignment  $l := c$
- the abstract evaluation of the l-value:

$$\mathcal{L} ::= \phi_{\mathbb{A}}^{-1}(\llbracket l \rrbracket^\#(h^\#)) = \{a \in \mathbb{A}^\# \mid \phi_{\mathbb{A}}(a) \in \llbracket l \rrbracket^\#(h^\#)\}$$

So, **when does the weak update happen ?**

There are two (non exclusive) situations:

- 1 **when**  $|\llbracket l \rrbracket^\#(h^\#)| > 1$ :  
this includes that the evaluation of  $l$  is not precise in the abstract
- 2 **when there exists**  $a \in \llbracket l \rrbracket^\#(h^\#)$  **such that**  $|\phi_{\mathbb{A}}^{-1}(\{a\})| > 1$ :  
this means that one of the addresses  $l$  may evaluate to corresponds to several distinct concrete cells

Both cases can be expected to happen frequently in pointer analysis...

# Pointer aliasing based on equivalence on access paths

## Aliasing relation

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

## Abstraction to infer pointer aliasing properties

- An **access path** describes a sequence of dereferences to resolve an l-value (*i.e.*, an address); *e.g.*:

$$a ::= x \mid a \cdot f \mid * a$$

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

## Examples of aliasing abstractions:

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

## Weak updates:

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

## Unsatisfactory abstraction of unbounded memory:

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

## Unable to express well structural invariants:

- for instance, that a structure should be a **list**, a **tree**...
- **very indirect** abstraction in numeric / path equivalence abstraction

**A common solution:  
shape abstraction**



# Outline

1 Memory models

2 Pointer Abstractions

3 Shape analysis in Three-Valued Logic (TVL)

- Principles of Three-Valued Logic based abstraction
- Comparing and concretizing Three-Valued Logic abstractions
- Weakening Three-Valued Logic abstractions
- Transfer functions
- Focusing

4 Conclusion

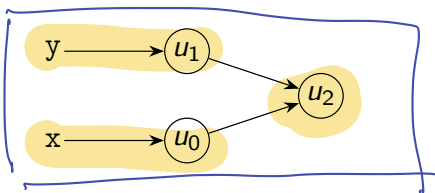
# Representation of memory states: memory graphs

Observation: representation of memory states by graphs

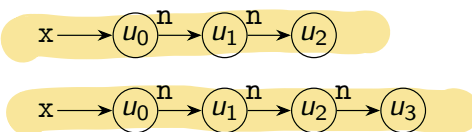
- **Nodes** (aka, atoms) denote **variables, memory locations**
- **Edges** denote **properties of addresses / pointers**, such as:
  - ▶ “field  $f$  of location  $u$  points to  $v$ ”
  - ▶ “variable  $x$  is stored at location  $u$ ”
- This representation is also relevant in the case of **separation logic** based shape abstraction

A couple of examples:

Two alias pointers:



A list of length 2 or 3:



We need to over-approximate **sets of shape graphs**

# Memory graphs and predicates: variables

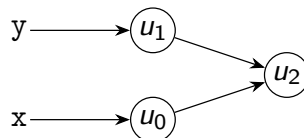
Before we apply some abstraction, we **formalize memory graphs** using some **predicates**, such as:

## “Variable content” predicate

We note  $x(u) = 1$  if node  $u$  represents the contents of  $x$ .

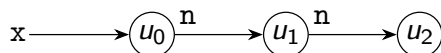
### Examples:

- **Two alias pointers:**



Then, we have  $x(u_0) = 1$  and  $y(u_1) = 1$ , and  $x(u) = 0$  (resp.,  $y(u) = 0$ ) in all the other cases

- **A list of length 2:**



Then, we have  $x(u_0) = 1$  and  $x(u) = 0$  in all the other cases

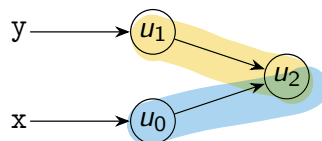
# Memory graphs and predicates: (field) pointers

## “Field content pointer” predicate

- We note  $n(u, v)$  if the field  $n$  of  $u$  stores a pointer to  $v$
- We note  $\underline{0}(u, v)$  if  $u$  stores a pointer to  $v$  (base address field is at offset 0)

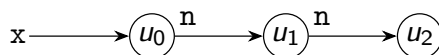
### Examples:

- **Two alias pointers:**



Then, we have  $\underline{0}(u_0, u_2) = 1$  and  $\underline{0}(u_1, u_2) = 1$ , and  $\underline{0}(u, v) = 0$  in all the other cases

- **A list of length 2:**



Then, we have  $n(u_0, u_1) = 1$  and  $n(u_1, u_2) = 1$ , and  $n(u, v) = 0$  in all the other cases

## 2-structures and concretization

We can represent the memory graphs using **tables of predicate values**:

### Two-structures and concretization

We assume 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 formal representation of a memory graph is a **2-structure**  $(\mathcal{U}, \phi) \in \mathbb{D}_2^\#$  defined by:

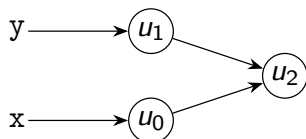
- a set  $\mathcal{U} = \{u_0, u_1, \dots, u_m\}$  of **atoms**
- a **truth table**  $\phi$  such that  $\phi(p_i, u_{l_1}, \dots, u_{l_{k_i}})$  denotes the truth value of  $p_i$  for  $u_{l_1}, \dots, u_{l_{k_i}}$  (where arities of predicates are respected)

Then,  $\gamma_2(\mathcal{U}, \phi)$  is the set of  $(e, h, \nu)$  where  $\nu : \mathcal{U} \rightarrow \mathbb{V}_{\text{addr}}$  and that satisfy exactly the truth tables defined by  $\phi$ :

- $(e, h, \nu)$  satisfies  $\mathbf{x}(u)$  iff  $e(\mathbf{x}) = \nu(u)$
- $(e, h, \nu)$  satisfies  $\mathbf{f}(u, v)$  iff  $h(\nu(u), \mathbf{f}) = \nu(v)$
- the name “two-structure” will become clear (very) soon
- the set of two-structures is parameterized by the data of a set of predicates  $\mathbf{x}(\cdot), \mathbf{y}(\cdot), \mathbf{0}(\cdot, \cdot), \mathbf{n}(\cdot, \cdot)$  (additional predicates will be added soon...)

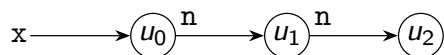
# Examples of two-structures

## Two alias pointers:



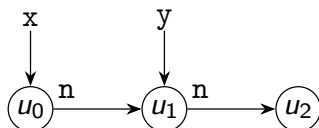
	x	y	$\mapsto$	$u_0$	$u_1$	$u_2$
$u_0$	1	0	$u_0$	0	0	1
$u_1$	0	1	$u_1$	0	0	1
$u_2$	0	0	$u_2$	0	0	0

## 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

## A list of length 2:



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

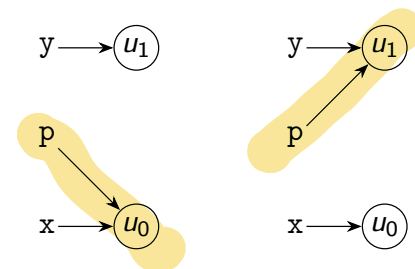
Lists of **arbitrary length** ? More on this **later**

# Unknown value: three valued logic

## How to abstract away some information ?

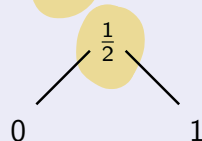
*i.e.*, how to abstract several graphs into one ?

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

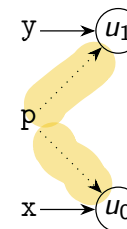


## A boolean lattice

- 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, since one node represents at most one memory cell

## An idea

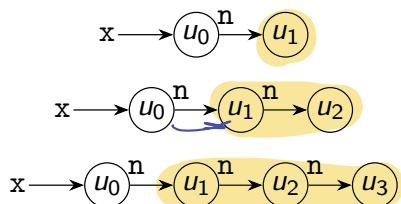
- Choose a node to represent **several** concrete nodes
- Similar to **smashing** of arrays using segments

## Definition: summary node

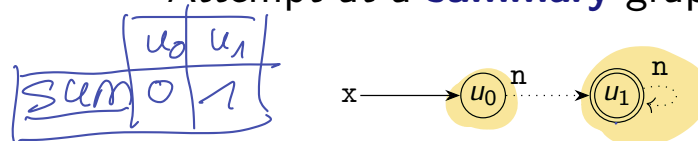
A **summary node** is an atom that may denote several concrete atoms

- intuition: we are using a **non injective function**  $\phi_{\Delta} : \mathbb{V}_{\text{addr}} \longrightarrow \mathbb{A}^{\#}$
- representation: double circled nodes

## Lists of lengths 1, 2, 3:



Attempt at a **summary** graph:



- Edges to  $u_1$  are dotted



# Additional graph predicate: sharing

We now define a few **higher level predicates** based on the previously seen **atomic predicates** describing the graphs.

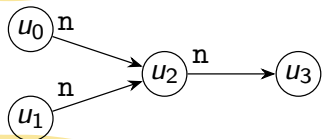
Example: a cell is **shared** if and only if there exists several distinct pointers to it

## “Is shared” predicate

The predicate  $\underline{\text{sh}}(u)$  holds if and only if

$$\exists v_0, v_1, \left\{ \begin{array}{l} v_0 \neq v_1 \\ \wedge \text{n}(v_0, u) \\ \wedge \text{n}(v_1, u) \end{array} \right.$$

(for concision, we assume only  $\text{n}$  pointers)



- $\underline{\text{sh}}(u_0) = \underline{\text{sh}}(u_1) = \underline{\text{sh}}(u_3) = 0$
- $\underline{\text{sh}}(u_2) = 1$

## Additional graph predicate: reachability

We can also define higher level predicates **using induction**:

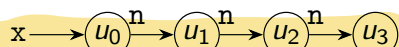
For instance, a cell is **reachable** from  $u$  if and only if it is  $u$  or it is reachable from a cell pointed to by  $u$ .

### “Reachability” predicate

The predicate  $\underline{r}(u, v)$  holds if and only if:

$$\vee \quad \begin{array}{l} u = v \\ \exists u_0, n(u, u_0) \wedge \underline{r}(u_0, v) \end{array} \quad \text{①} \xrightarrow{\wedge} \text{②} \xrightarrow{\cdot} \text{③} \xrightarrow{\cdot} \text{④}$$

(for concision, we assume only  $n$  pointers)



- $\underline{r}(u_1, u_0) = \underline{r}(u_2, u_0) = \underline{r}(u_3, u_1) = 0$

- $\underline{r}(u_0, u_0) = \underline{r}(u_0, u_2) = \underline{r}(u_0, u_3) = 1$

### “Acyclicity” predicate

The predicate  $\underline{acy}(u)$  holds iff  $\exists v, v \neq u \wedge \underline{r}(u, v) \wedge \underline{r}(v, u)$  does not hold

# Three structures

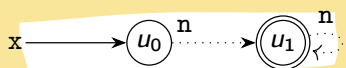
As for 2-structures, we assume a set  $\mathcal{P} = \{p_0, p_1, \dots, p_n\}$  of **predicates** fixed and write  $k_i$  for the arity of predicate  $p_i$ .

## Definition: 3-structures

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

- a set  $\mathcal{U} = \{u_0, u_1, \dots, u_m\}$  of **atoms**
  - 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  $\{0, \frac{1}{2}, 1\}$

We write  $\mathbb{D}_3^\#$  for the set of three-structures.



$$\begin{cases} \mathcal{U} = \{u_0, u_1\} \\ \mathcal{P} = \{x(\cdot), n(\cdot, \cdot), \text{sum}(\cdot)\} \end{cases}$$

$x \rightarrow u_0$        $u_1$

	x	sum	n	$u_0$	$u_1$
$u_0$	1	0	$u_0$	0	$\frac{1}{2}$
$u_1$	0	$\frac{1}{2}$	$u_1$	0	$\frac{1}{2}$

**In the following we build up an abstract domain of 3-structures (but a bit more work is needed for the definition of the concretization)**

# Main predicates and concretization

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 $\frac{1}{2}$ )
$\text{sh}(u)$	whether there exists several distinct pointers to $u$
$\text{r}(u, v)$	whether $v$ is reachable starting from $u$
$\text{acy}(v)$	$v$ may not be on a cycle

## Concretization for 2 structures:

$$(e, h, \nu) \in \gamma_2(\mathcal{U}, \phi) \iff \bigwedge_{p \in \mathcal{P}} (env, h, \nu) \text{ evaluates } p \text{ as specified in } \phi$$

## Concretization for 3 structures:

- predicates with value  $\frac{1}{2}$  may concretize either to true or to false
- but the concretization of summary nodes is still unclear...

# Outline

1 Memory models

2 Pointer Abstractions

3 Shape analysis in Three-Valued Logic (TVL)

- Principles of Three-Valued Logic based abstraction
- Comparing and concretizing Three-Valued Logic abstractions
- Weakening Three-Valued Logic abstractions
- Transfer functions
- Focusing

4 Conclusion

# Embedding

Reasons why we need to set up a **relation among structures**:

- learn how to **compare** two 3-structures
- describe the **concretization** of 3-structures into 2-structures

## The embedding principle

Let  $\mathcal{S}_0 = (\mathcal{U}_0, \phi_0)$  and  $\mathcal{S}_1 = (\mathcal{U}_1, \phi_1)$  be two three structures, with the same sets of predicates  $\mathcal{P}$ . 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  $p \in \mathcal{P}$  of arity  $k$ , for all  $u_{l_1}, \dots, u_{l_{k_i}} \in \mathcal{U}_0$ ,

$$\phi_0(u_{l_1}, \dots, u_{l_{k_i}}) \sqsubseteq \phi_1(f(u_{l_1}), \dots, f(u_{l_{k_i}}))$$

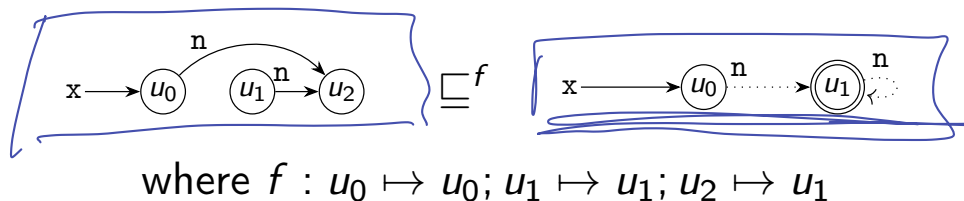
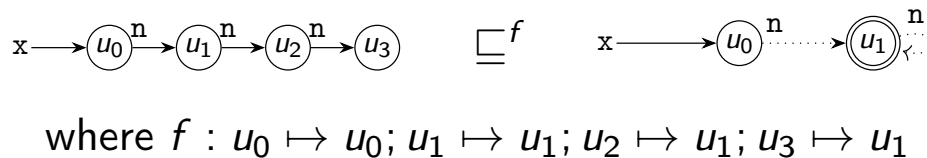
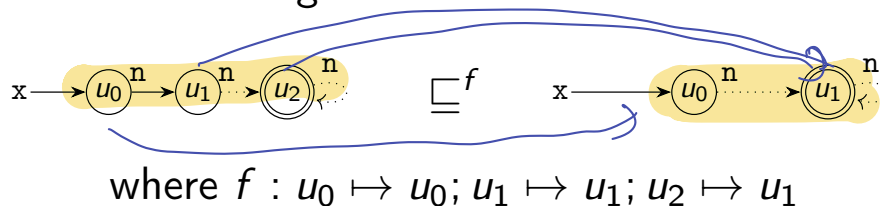
Then, we write  $\mathcal{S}_0 \sqsubseteq^f \mathcal{S}_1$



- Note: we use the order  $\sqsubseteq$  of the lattice  $\{0, \frac{1}{2}, 1\}$
- Intuition: **embedding** defines an **abstract pre-order**  
i.e., when  $\mathcal{S}_0 \sqsubseteq^f \mathcal{S}_1$ , any property that is satisfied by  $\mathcal{S}_0$  is also satisfied by  $\mathcal{S}_1$

# Embedding examples

A few examples of the embedding relation:



The last example shows summary nodes are not enough to capture just lists:

- **reachability** would be necessary to constrain it be a list
- alternatively: list cells **should not be shared**

# Concretization of three-structures

## Intuitions:

- concrete memory states correspond to 2-structures
- embedding applies uniformly to 2-structures and 3-structures (in fact, 2-structures are a subset of 3-structures)
- 2-structures can be embedded into 3-structures, that abstract them

This suggests **a concretization of 3-structures in two steps:**

- 1 turn it into a set of 2-structures that can be embedded into it
- 2 concretize these 2-structures

## Concretization of 3-structures

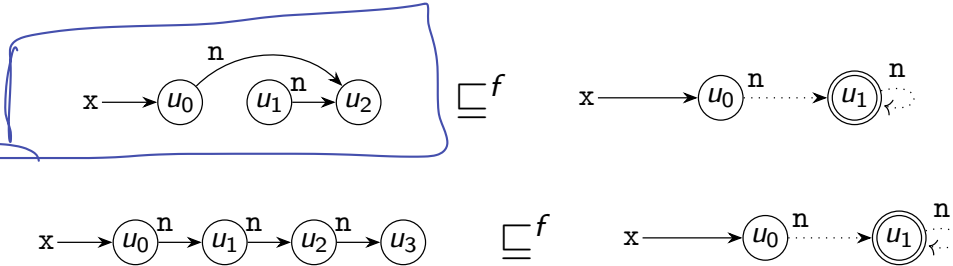
Let  $\mathcal{S}$  be a 3-structure. Then:

$$\gamma_3(\mathcal{S}) = \bigcup \{ \gamma_2(\mathcal{S}') \mid \mathcal{S}' \text{ 2-structure s.t. } \exists f, \mathcal{S}' \sqsubseteq^f \mathcal{S} \}$$



# Concretization examples

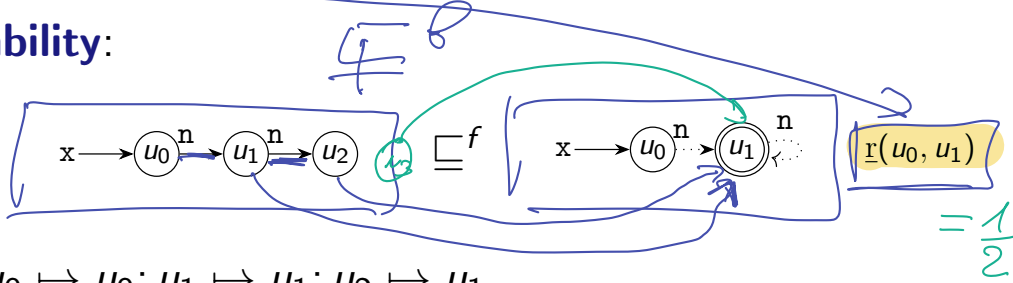
## Without reachability:



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

## With reachability:

$\Gamma(u_0, u_1) = 1$   
 $\Gamma(u_0, u_2) = 1$



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

Note the first item of the above case does not work here

# Disjunctive completion

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

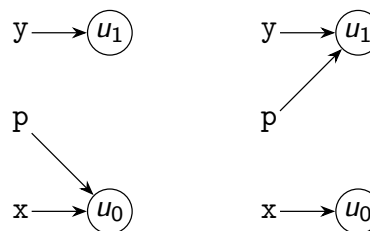
```

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



## Abstraction based on disjunctive completion

- In the following, we use **partial disjunctive completion** *i.e.*, TVLA manipulates **finite disjunctions** of 3-structures  
We write  $\mathbb{D}_{\mathcal{P}(3)}^\#$  for the abstract domain made of finite sets of 3-structures in  $\mathbb{D}_3^\#$
- How to ensure disjunctions **will not grow infinite** ?  
the set of atoms is **unbounded**, so it is not necessarily true!

# Outline

1 Memory models

2 Pointer Abstractions

3 Shape analysis in Three-Valued Logic (TVL)

- Principles of Three-Valued Logic based abstraction
- Comparing and concretizing Three-Valued Logic abstractions

 Weakening Three-Valued Logic abstractions

 Transfer functions

 Focusing

4 Conclusion

# Canonical abstraction

To **prevent disjunctions** from growing infinite, we propose to **normalize (in a precision losing manner)** abstract states:

- the analysis may use all 3-structures at most points
- at selected points (including loop heads), only 3-structures in a finite set  $\mathbb{D}_{\text{can}(3)}^\#$  are allowed
- there is a function to coarsen 3-structures into elements of  $\mathbb{D}_{\text{can}(3)}^\#$

## Canonicalization function

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

- operator  $\text{can}$  is called **canonicalization** if and only if it defines an **upper closure operator**
- then it extends into a **canonicalization operator**  $\text{can} : \mathcal{P}(\mathcal{L}) \rightarrow \mathcal{P}(\mathcal{L}')$  for the disjunctive completion domain:

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

- proof of the extension to disjunctive completion domains: left as an exercise
- to make the powerset domain work, we simply need a  $\text{can}$  over 3-structures

# Canonical abstraction

## Definition of a finite lattice $\mathbb{D}_{\text{can}(3)}^\#$

We partition the set of predicates  $\mathcal{P}$  into two subsets  $\mathcal{P}_a$  and  $\mathcal{P}_o$ :

- $\mathcal{P}_a$  defines **abstraction predicates** and should contain only unary predicates and have a finite truth table whatever the number of atoms
- $\mathcal{P}_o$  denotes **non-abstraction predicates**, and may define truth tables of unbounded size

Then, we let  $\mathbb{D}_{\text{can}(3)}^\#$  be the set of 3-structures such that **no pair of atoms have the same value of the  $\mathcal{P}_a$  predicates**. It defines a finite set of 3-structures.

This sub-lattice defines a clear “canonicalization” algorithm:

## 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**

*can weaken*

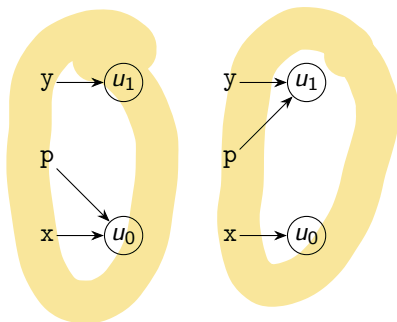
# Canonical abstraction examples

Most common TVLA instantiation:

- we assume there are  $n$  variables  $x_1, \dots, x_n$   
 thus the number of unary predicates is finite, and provides a good choice for  $\mathcal{P}_a$
- **sub-lattice**: structures with atoms distinguished by the values of the unary predicates  $x_1, \dots, x_n$

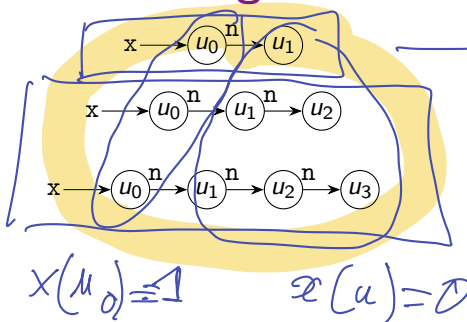
Examples:

Elements not merged:

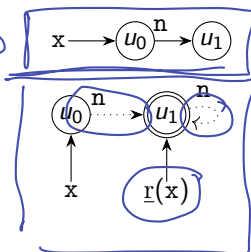


Elements merged:

Lists of lengths 1, 2, 3:



Abstract into:



# Outline

1 Memory models

2 Pointer Abstractions

3 Shape analysis in Three-Valued Logic (TVL)

- Principles of Three-Valued Logic based abstraction
- Comparing and concretizing Three-Valued Logic abstractions
- Weakening Three-Valued Logic abstractions
- Transfer functions
- Focusing

4 Conclusion

# Principle for the design of sound transfer functions

- Intuitively, **concrete states** correspond to **2-structures**
- The **analysis** should track **3-structures**, thus the analysis and its soundness proof need to **rely on the embedding relation**

## Embedding theorem

We assume that

- $\mathcal{S}_0 = (\mathcal{U}_0, \phi_0)$  and  $\mathcal{S}_1 = (\mathcal{U}_1, \phi_1)$  define a pair of 3-structures
- $f : \mathcal{U}_0 \rightarrow \mathcal{U}_1$ , is such that  $\mathcal{S}_0 \sqsubseteq^f \mathcal{S}_1$  (embedding)
- $\Psi$  is a logical formula, with variables in  $X$
- $g : X \rightarrow \mathcal{U}_0$  is an assignment for the variables of  $\Psi$

Then, the semantics (evaluation) of logical formulae is such that

$$\begin{array}{ccc}
 \text{2 struct} & \leftarrow \llbracket \Psi \mid_g \rrbracket (\mathcal{S}_0) \sqsubseteq \llbracket \Psi \mid_{f \circ g} \rrbracket (\mathcal{S}_1) \longrightarrow & \text{3 struct} \\
 \text{conc} & & \text{abs}
 \end{array}$$

**Intuition:** this theorem ties the evaluation of conditions in the concrete and in the abstract in a general manner



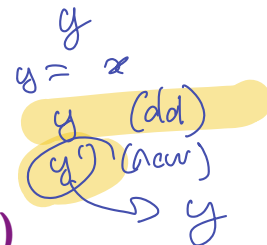
# Principle for the design of sound transfer functions

## Transfer functions for static analysis

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

**Example:** analysis of an assignment  $y := x$

- 1 let  $y'$  be a new predicate that denotes the *new* value of  $y$
- 2 then we can add the constraint  $y'(u) = x(u)$   
(using the embedding theorem to prove soundness)
- 3 rename  $y'$  into  $y$

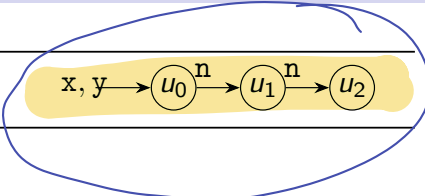


## Advantages:

- **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**

**Disadvantage:** precision will require some care, more on this later!

# Assignment: a simple case

<b>Statement</b> $l_0 : y = y -> n; l_1 : \dots$	<b>Pre-condition</b> $\mathcal{S}$ 
--	--

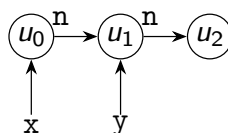
## Transfer function computation:

- 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):

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

$$\begin{aligned}
 y'(u_0) &= 0 \\
 y'(u_1) &= 1 \\
 y'(u_2) &= 0
 \end{aligned}$$

- **resulting structure:**



This is exactly the expected result

# Outline

1 Memory models

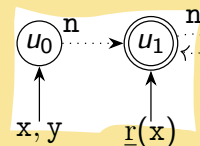
2 Pointer Abstractions

3 Shape analysis in Three-Valued Logic (TVL)

- Principles of Three-Valued Logic based abstraction
- Comparing and concretizing Three-Valued Logic abstractions
- Weakening Three-Valued Logic abstractions
- Transfer functions
- Focusing

4 Conclusion

## Assignment: a more involved case

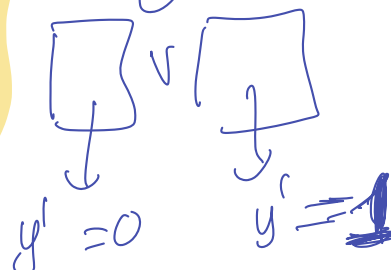
Statement  $l_0 : y = y \rightarrow n; l_1 : \dots$ Pre-condition  $\mathcal{S}$ 

- 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  $\hat{\mathcal{S}}$  of 3-structures such that:

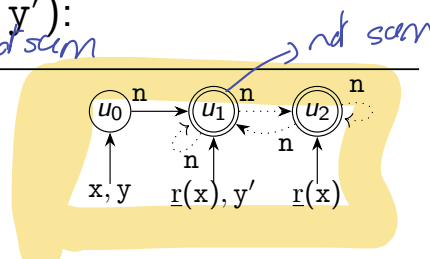
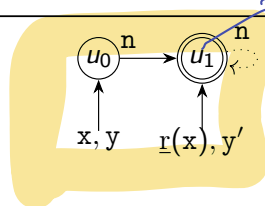
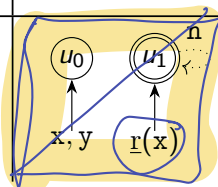
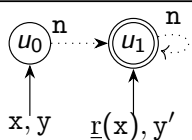
- $f$  evaluates to 0 or 1 on all elements of  $\hat{\mathcal{S}}$
- **precision was gained**:  $\forall S' \in \hat{\mathcal{S}}, S' \sqsubseteq \mathcal{S}$  (embedding)
- **soundness is preserved**:  $\gamma(\mathcal{S}) = \bigcup \{ \gamma(S') \mid S' \in \hat{\mathcal{S}} \}$

- Details of focusing algorithms are rather complex: not detailed here
- They involve splitting of summary nodes, solving of boolean constraints

**Example:** focusing on

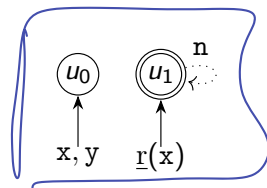
$$y'(u) = \exists v, y(v) \wedge n(v, u)$$

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

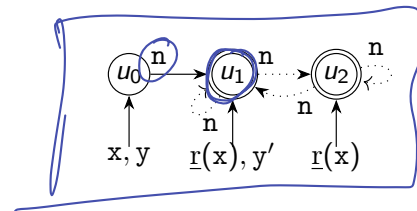


# Focus and coerce

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**

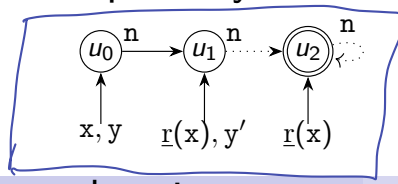
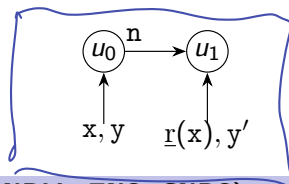


$u_0$  has an  $n$ -field to  $u_1$  so  $u_1$  denotes a unique atom and **cannot be a summary node**

## Coerce operation

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

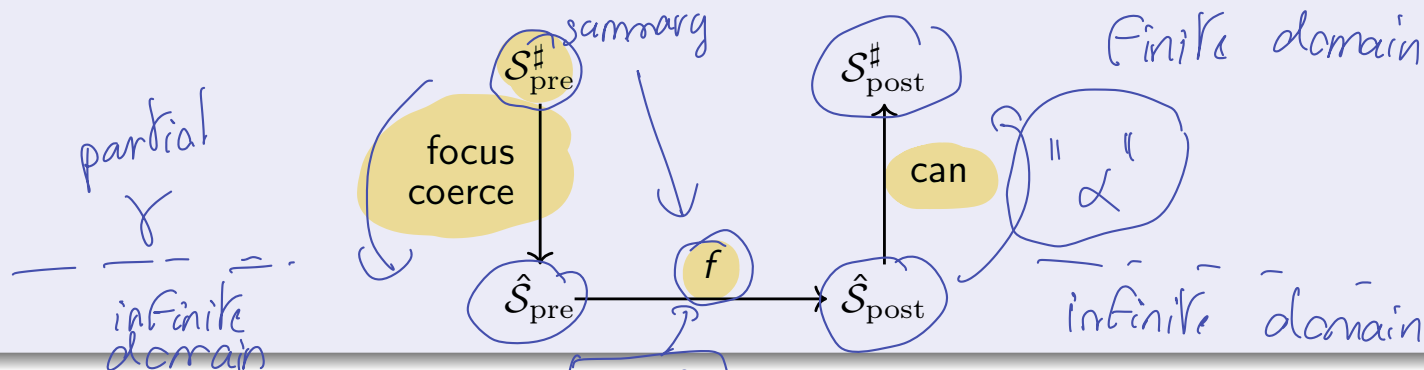
**Result:** one case removed (bottom), two possibly summary nodes non summary



## 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**

# Shape analysis with three valued logic

**Abstract states; two abstract domains** are used:

- **infinite domain**  $\mathbb{D}_{\mathcal{P}(3)}^\#$ : finite disjunctions of 3-structures in  $\mathbb{D}_3^\#$  for general abstract computations *infinitely many atoms*
- **finite domain**  $\mathbb{D}_{\mathcal{P}(\text{can}(3))}^\#$ : disjunctions of finite domain  $\mathbb{D}_{\text{can}(3)}^\#$  to simplify abstract states and for loop iteration
- **concretization** via  $\mathbb{D}_2^\#$

**Abstract post-conditions:**

- 1 start from  $\mathbb{D}_{\mathcal{P}(3)}^\#$  or  $\mathbb{D}_{\text{can}(3)}^\#$
- 2 focus and coerce when needed
- 3 apply the concrete transformation
- 4 apply can to weaken abstract states; result in  $\mathbb{D}_{\mathcal{P}(\text{can}(3))}^\#$

**Analysis of loops:**

- iterations in  $\mathbb{D}_{\mathcal{P}(\text{can}(3))}^\#$  terminate, as it is finite



# Outline

- 1 Memory models
- 2 Pointer Abstractions
- 3 Shape analysis in Three-Valued Logic (TVL)
- 4 Conclusion

# Updates and summarization

## Weak updates cause significant precision loss...

- Basic pointer abstractions suffer weak update issues leading to high precision loss
- Various techniques exist to mitigate this effect
- Today, we saw shape analysis based on three-valued predicates as a way to circumvent it  
Next week, another technique will be presented...

## A novel family of abstract interpretation based static analyses:

- Some analysis operations require **local concretization** of abstract predicates
- A reverse operation makes **abstract states more abstract**

## Internships

# Assignment: formalization and paper reading

## Formalization of the concretization of 2-structures:

- describe the concretization formula, assuming that we consider the predicates discussed in the course
- run it on the list abstraction example (from the 3-structure to a few select 2-structures, and down to memory states)
- ~~prove the correctness and termination of the widening of the cofibered abstract domain~~

## Reading:

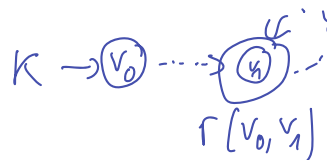
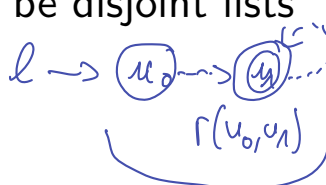
**Parametric Shape Analysis via 3-Valued Logic.**

**Shmuel Sagiv, Thomas W. Reps et Reinhard Wilhelm.**

In POPL'99, pages 105–118, 1999.

# Assignment: a simple analysis in TVLA

l, k assumed to be disjoint lists



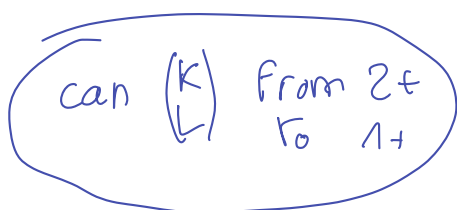
while(l ≠ 0){

t = l -> n;

focus on u<sub>0</sub>

l -> n = k;

k = t;



l = t;

