

# IA159 Formal Methods for Software Analysis

## Shape Analysis via 3-Valued Logic

Jan Strejček

Faculty of Informatics  
Masaryk University

## focus

- shape analysis in general
- 3-valued logic approach
  - the logic and shape graphs
  - algorithm
  - TVLA and (semi)demo
- other approaches

## sources

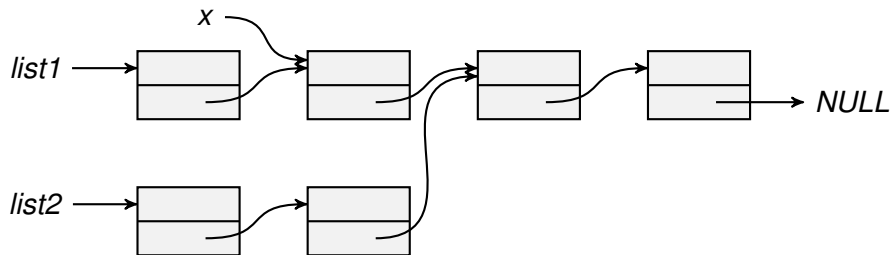
- M. Sagiv, T. Reps, R. Wilhelm: **Parametric Shape Analysis via 3-Valued Logic**, ACM Trans. Program. Lang. Syst. 24(3), 2002.
- B. Jeannet, A. Loginov, T. Reps, M. Sagiv: **A Relational Approach to Interprocedural Shape Analysis**, SAS 2004.

**Shape analysis** is a static analysis focused on program properties related to dynamically allocated memory. In particular, it aims to detect or verify the absence of heap-specific errors like

- **null dereference**
- **memory leaking**
- **dangling pointer** – a pointer to a deallocated memory
- violation of expected properties of dynamic datastructures (e.g., the datastructure is a cyclic list)
- ...

# Basic idea

For each program location, we want to compute all reachable **memory configurations**.



- The number of reachable memory configurations can be very large or even unbounded.
- We need to find finite representations of potentially infinite sets of memory configurations.
- We compute over-approximations of sets of reachable memory configurations (an abstraction).
- The over-approximations are represented by finite **shape graphs**.
- Shape graphs can be represented using **logics**, graph structures, automata, ...

Representing concrete memory configurations  
with 2-valued logical structures

# Logical representation of concrete configurations

- configurations are represented by predicate logic formulas over the following **core predicates**:

unary predicate  $x(v)$  for each pointer variable  $x$

binary predicate  $n(v_1, v_2)$  for each structure field  $n$  serving as a pointer

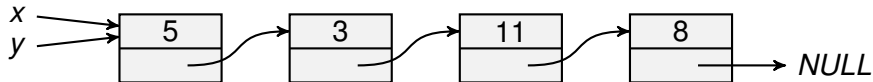
binary predicate  $eq(v_1, v_2)$

predicate	intended meaning
$x(v)$	variable $x$ points to memory cell $v$
$n(v_1, v_2)$	field $n$ of $v_1$ (i.e., $v_1.n$ ) points to $v_2$
$eq(v_1, v_2)$	$v_1$ and $v_2$ denote the same memory cell

- memory configurations correspond to interpretations
- allocated memory cells correspond to domain elements

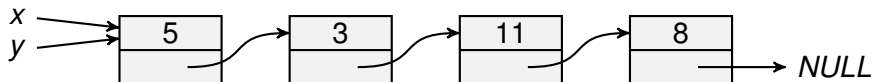
# Example

```
typedef struct node {  
    struct node *n;  
    int data;  
} *List;
```





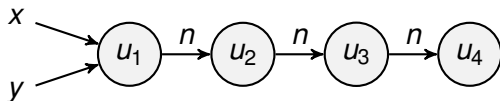
# Example



## logical representation

- domain  $\{u_1, u_2, u_3, u_4\}$
- $x(u_1) = y(u_1) = 1$
- $n(u_1, u_2) = n(u_2, u_3) = n(u_3, u_4) = 1$
- $eq(u_1, u_1) = eq(u_2, u_2) = eq(u_3, u_3) = eq(u_4, u_4) = 1$
- values of all predicates on other arguments is 0

## visualisation of the logical representation

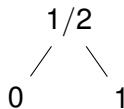


- **storeless** approach – it does not model precise location of allocated cells in the memory
- it cannot handle pointer arithmetics
- some interpretations do not represent any memory configuration, e.g., if  $n(u, v) = n(u, w) = 1$  for some  $v \neq w$
- these interpretations are eliminated by formulas called **integrity constraints**, e.g.,  $n(u, v) \wedge n(u, w) \implies eq(v, w)$
- the size of a configuration (and its logical representation) can be unbounded  
→ we use an abstraction to get a less precise, but bounded representation

3-valued logical structures and shape graphs

# 3-valued logic

- uses 3 truth values: 0, 1,  $1/2$  (indefinite value)
- new operation the least upper bound  $\sqcup$
- operations  $\wedge, \vee, \neg$  are extended



$\sqcup$	0	1	$1/2$
0	0	$1/2$	$1/2$
1	$1/2$	1	$1/2$
$1/2$	$1/2$	$1/2$	$1/2$

$\wedge$	0	1	$1/2$
0	0	0	0
1	0	1	$1/2$
$1/2$	0	$1/2$	$1/2$

$\vee$	0	1	$1/2$
0	0	1	$1/2$
1	1	1	1
$1/2$	$1/2$	1	$1/2$

$\neg$	
0	1
1	0
$1/2$	$1/2$

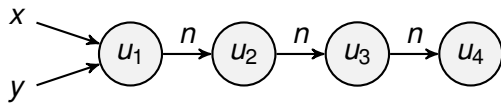
## abstraction

- we merge cells with identical values of all unary predicates
- values of unary predicates on merged cells keep unchanged (these are always 0 or 1)
- values of binary predicates on merged cells are defined as the least upper bound of the values on the original cells

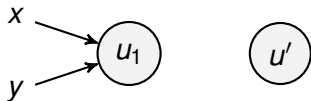
**Example:** if  $u_2, u_3, u_4$  is merged into  $u'$  and  $u_1$  is not, then

$$n(u_1, u') = n(u_1, u_2) \sqcup n(u_1, u_3) \sqcup n(u_1, u_4)$$

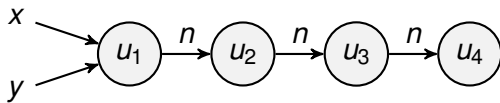
# Example



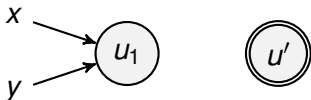
- let  $u_2$ ,  $u_3$ , and  $u_4$  be merged into  $u'$



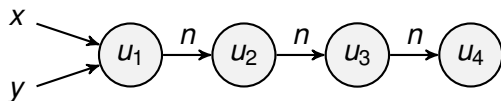
# Example



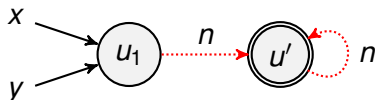
- let  $u_2, u_3$ , and  $u_4$  be merged into  $u'$
- $eq(u', u') = eq(u_2, u_2) \sqcup eq(u_2, u_3) \sqcup \dots \sqcup eq(u_4, u_4) = 1/2$
- cells with  $eq(u, u) = 1/2$  are called **summary nodes**



# Example

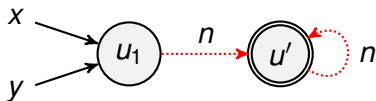


- let  $u_2$ ,  $u_3$ , and  $u_4$  be merged into  $u'$
- $eq(u', u') = eq(u_2, u_2) \sqcup eq(u_2, u_3) \sqcup \dots \sqcup eq(u_4, u_4) = 1/2$
- cells with  $eq(u, u) = 1/2$  are called **summary nodes**
- $n(u_1, u') = 1/2$  and  $n(u', u') = 1/2$





# Shape graph interpretation



This **shape graph** may represent:

- an acyclic list of 2+ elements pointed by  $x$  and  $y$
- a cyclic list of 2+ elements pointed by  $x$  and  $y$ , with the first element not lying on the cycle
- besides of these,  $u'$  can also represent another cyclic or acyclic lists not pointed by anything (i.e., garbage)

To refine the abstraction, we add **instrumentation predicates**.

# Instrumentation predicates

## instrumentation predicates

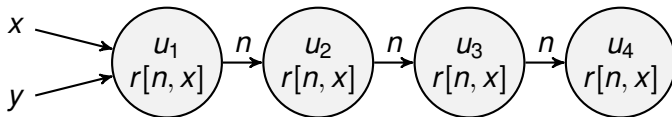
- are defined by first-order formulas over core predicates
- may also use transitive (or reflexive and transitive) closures of binary predicates

typical instrumentation predicates for linked lists

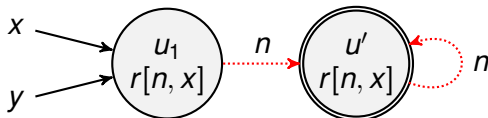
predicate	meaning	definition
$t[n](v_1, v_2)$	$v_2$ is reachable from $v_1$ via $n$ -fields	$n^*(v_1, v_2)$
$r[n, x](v)$	$v$ is reachable from variable $x$ via $n$ -fields	$\exists v_1 . x(v_1) \wedge t[n](v_1, v)$
$c[n](v)$	$v$ lies on a cycle of $n$ -fields	$\exists v_1 . n(v, v_1) \wedge t[n](v_1, v)$

# Example

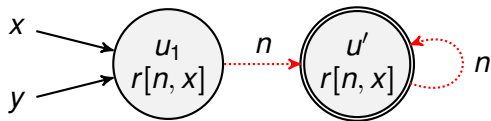
- we add instrumentation predicates  $r[n, x]$  and  $c[n]$



- there are more unary predicates determining cell merging



# Example



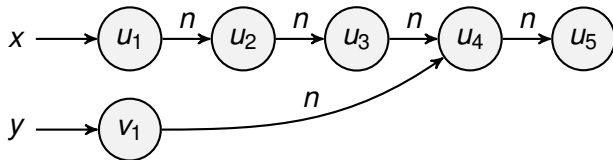
Now it represents exactly all acyclic lists of 2+ elements:

- all nodes satisfy  $r[n, x]$ , hence they are reachable from  $x$  (i.e., there is no garbage)
- $c[n]$  does not hold in any node, hence the list is acyclic

The choice of instrumentation predicates is crucial for obtaining some useful output.

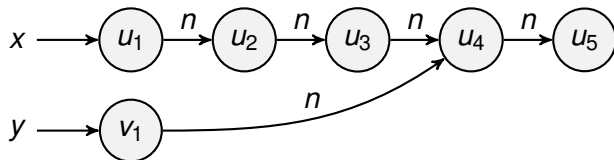
# Example

Compute the shape graph given by core predicates and instrumentation predicates  $r[n, x]$ ,  $r[n, y]$ :

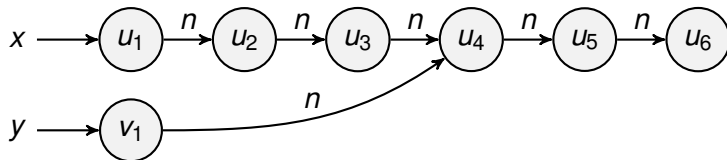


# Example

Compute the shape graph given by core predicates and instrumentation predicates  $r[n, x]$ ,  $r[n, y]$ :

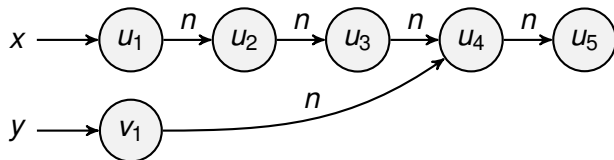


Decide whether the shape graph represents also the configuration below.

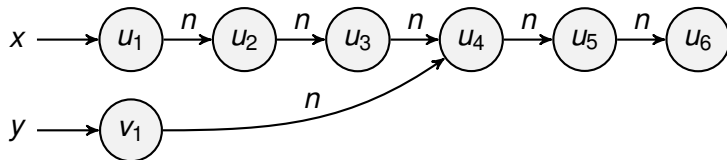


# Example

Compute the shape graph given by core predicates and instrumentation predicates  $r[n, x]$ ,  $r[n, y]$ :



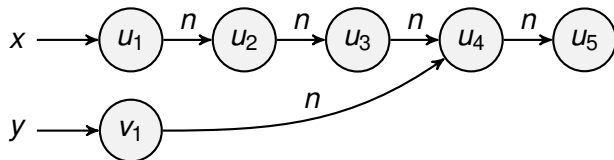
Decide whether the shape graph represents also the configuration below.



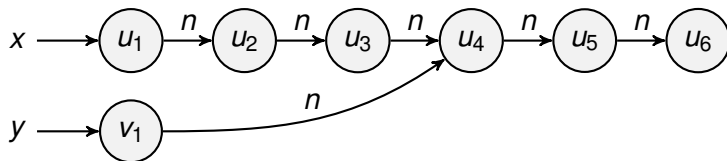
Suggest an instrumentation predicate that would make shape graphs for the two configurations different.

# Example

Compute the shape graph given by core predicates and instrumentation predicates  $r[n, x], r[n, y]$ :



Decide whether the shape graph represents also the configuration below.



Suggest an instrumentation predicate that would make shape graphs for the two configurations different.

**Solution:**  $is[n](v)$  defined by  $\exists v_1, v_2 . n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2$



Algorithm: the first look

# Algorithm: the first look

- there are only finitely many different shape graphs for a fixed finite set of core and instrumentation predicates
- the algorithm is a standard abstract interpretation

## algorithm

**input:** a program and  
shape graphs describing possible initial memory configurations

- 1 assign the input shape graphs to the initial program location
- 2 **repeat**
- 3 |   **foreach** program statement **do**
- 4 |   |   take the shape graphs assigned to the location before the statement
- 5 |   |   and update shape graphs in the locations after the statement
- 6 **until** a fixpoint is reached

- for each core predicate and each program statement, there is a **predicate-update** formula describing the values of the predicate after the statement using the values of core predicates before the statement
- using the predicate-update formulae, it is easy to compute the effect of the statement on concrete memory configurations
- to compute the effect of a statement on shape graphs is harder: values of instrumentation predicates are given by their definition formulas and values of core predicates, but this approach would quickly lead to loss of precision (values 1/2)
- to get better results, we define also **predicate-update** formulas for instrumentation predicates, which may use values of both core and instrumentation predicates before the statement

TVLA and (semi)demo

- = **Three Valued Logic Analysis Engine**
- developed at Tel Aviv University under supervision of Mooly Sagiv
- written in Java
- currently in version 3 (extended with heap decomposition)
- available for academic purposes
- `http://www.cs.tau.ac.il/~tvla/`

**program** has to be specified in four parts

- 1 declaration of predicates and integrity constraints
    - core predicates are just declared
    - instrumentation predicates have to be defined by formulas
  - 2 operation semantics of all program statements
    - for each statement used in the program, the corresponding predicate-update formulas have to be given
    - each statement can be accompanied by an error detection formula (e.g., null dereference)
  - 3 program flowgraph (including asserts)
  - 4 the list of locations for which we want to get all reachable shape graphs
- parts 1 and 2 can be used repeatedly and they are available for certain classes of programs (e.g., for programs manipulating linked lists or trees)
  - part 4 is optional

## initial shape graphs

- described using a simple text format

```
tvla <program> <initial_graphs>
```

**output** file contains

- picture of the program flowgraph
- reachable shape graphs for specified locations
- potential error messages



# Example

```
typedef struct node {
    struct node *n;
    int data;
} *List;

List reverse(List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = x->n;
        x->n = y;
        y = x;
        x = t;
    }
    return y;
}
```

# Example

```
typedef struct node {  
    struct node *n;  
    int data;  
} *List;
```

```
List reverse(List x) {  
    List y, t;  
    y = NULL;  
    while (x != NULL) {  
        t = x->n;  
        x->n = y;  
        y = x;  
        x = t;  
    }  
    return y;  
}
```

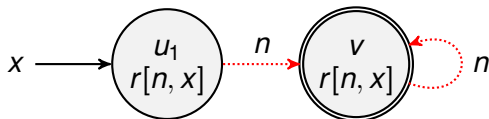
(SEMI)DEMO

Algorithm: a closer look

# Computing the effect of a statement on a shape graph

- 1 operation Focus
- 2 evaluation of statement guards
- 3 computing new values of predicates
- 4 operation Coerce
- 5 operation Blur

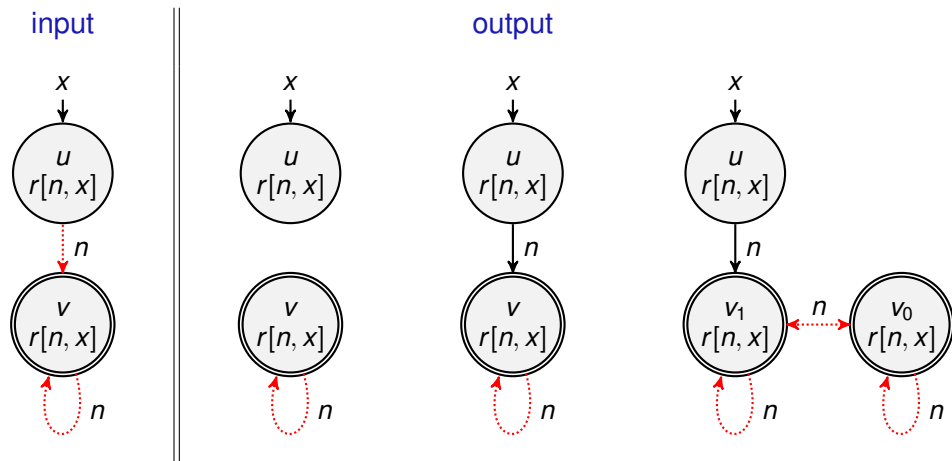
We will compute the effect of  $t = x \rightarrow n$  on the shape graph:



- applied on statements with defined **focus formula**, which is a formula with exactly one free variable
- operation Focus takes the shape graph and returns the set of shape graphs representing the same configurations and such that the focus formula is not evaluated to  $1/2$  on any node of any of the graphs.
- operation Focus modifies only values of predicates in the focus formula, values of other predicates are not recomputed
- hence, some resulting graphs may not satisfy integrity constraints

# Operation Focus: example

- focus formula for  $t = x \rightarrow n$  is  $f(w) = \exists v_1. x(v_1) \wedge n(v_1, w)$
- formula ensures that after the statement, the predicate  $t(v)$  cannot have value 1/2



# Evaluation of statement guards

- for each statement, there can be defined a **guard**, which is again a formula
- the statements is not performed on the shape graphs for which the guard evaluates to 0
- it is typically used to handle program branching
- statement  $t = x \rightarrow n$  has no guard

# Computing new values of predicates

- we use predicate-update formulas corresponding to the statement to compute new predicate values
- predicates with no predicate-update formulas keep their value

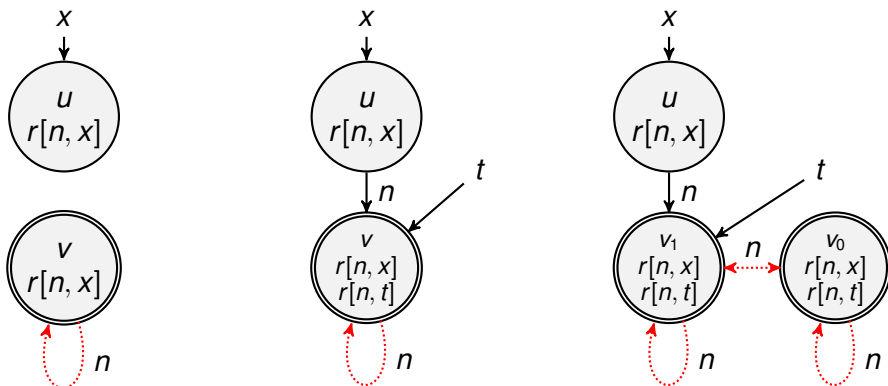


# Computing new values of predicates: example

Predicate-update formulas for  $t = x \rightarrow n$

predicate	predicate-update formula
$t(v)$	$\exists v_1 . x(v_1) \wedge n(v_1, v)$
$r[n, t](v)$	$r[n, x](v) \wedge (c[n](v) \vee \neg x(v))$

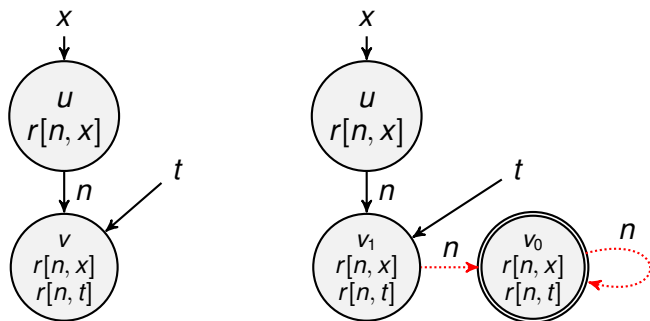
output



- removes shape graphs not satisfying integrity constraints
- makes values of some predicates more precise

# Operation Coerce: example

- shape graph on the left is corrupted as  $r[n, x](v)$  cannot hold  $\implies$  the graph is removed
- in the shape graph in the middle,  $v$  cannot be a summary node as  $t(v)$  holds
- on the right,  $v_1$  cannot be a summary node for the same reason, and moreover  $c[n](v_1)$  does not hold and thus  $n(v_1, v_1)$ ,  $n(v_0, v_1)$  cannot hold



- can further merge nodes with same values of unary predicates
- consequently, some shape graphs can become identical
- in our example, Blur has no effect

- TVLA works automatically, but the user has to
  - provide semantics of program statements
  - select/supply suitable instrumentation predicates
  - process the results and filter out false alarms
- studied extensions and applications
  - interprocedural shape analysis (can handle also recursive programs)
  - lazy shape analysis
  - shape analysis and CEGAR
  - shape analysis for parallel processes
  - mix of shape analysis and data-related abstract interpretation (can be used e.g., to prove that sorting algorithms output sorted linked lists)
  - can be used also to analyse liveness of java objects and their timely deallocation
  - ...

Other approaches and tools for shape analysis

Other approaches to analysis of dynamically allocated memory are based on

- separation logic and (bi-)abduction (**Infer**)
- translation to first-order logic and automated theorem proving (**HAVOC**)
- symbolic memory graphs (**Predator**)
- tree automata (**Forester**)
- ...