Shape Analysis

Reinhard Wilhelm Mooly Sagiv Thomas Reps

Shape-analysis algorithms

- statically analyze a program
- determine information about the heap-allocated data structures that the program manipulates.

Help

- to understand, debug, or verify programs.
- in compile-time garbage collection, instruction scheduling, and parallelization.

Pointers

- are error prone,
- have complex (derived) semantics, e.g. reachability,
- complicate language semantics, e.g. side effects of assignments,
- introduce alias problems into dependence analysis.

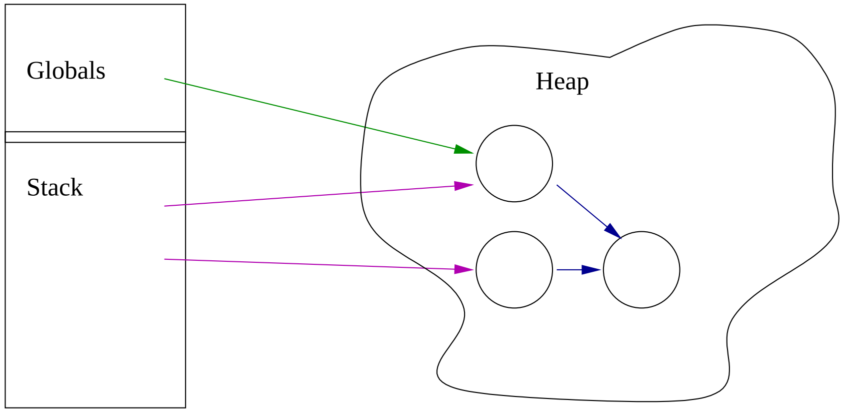
Contents of the talk

1. questions about the contents of the heap
2. destructive list reverse as a running example
3. shape analysis
   - summarization
   - parametricity
4. answers as given by shape analysis
5. applications

A reduced view of programs

Shape analysis of a program is (only) interested in

- linked heap structures,
- the pointers into the heap
  - from global memory
  - from the stack or
  - from cells in the heap.



Execution state to mean

- the heap cells and the values of their pointer components
- the values of pointer variables in the store.

Questions about execution states

At each point in a program:

NULL-pointers: Is a pointer value NULL before dereferencing?

- **true:** issue error message
- **false:** eliminate check for NULL
- **maybe:** warn about dereference

Alias: Do two pointer expressions refer to the same heap cell?

- **true:** predict a cache hit or trigger a prefetch
- **false:** improve program dependence information

Sharing: Is a heap cell shared?

- **true:** warn about explicit deallocation
- **false:** generate explicit deallocation

More Questions

Reachability: Is a heap cell reachable from a specific variable/any pointer variable?

true: Used in program verification

false: Identify as garbage

Disjointness: Will two data structures ever have common elements?

false: Distribution possible

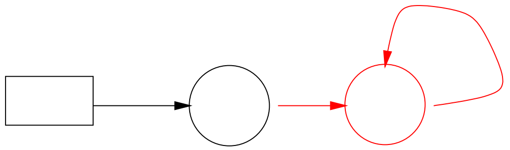
Cyclicity: Can a heap cell never be part of a cycle?

false: garbage collection by reference counting

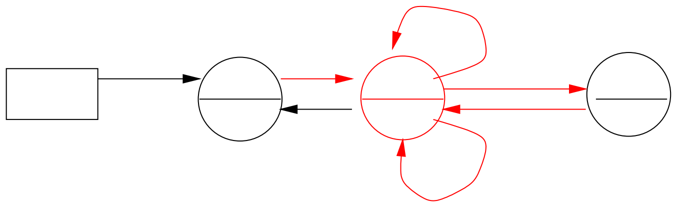
The Ultimate Question: Shape

What will be the “shape” of the heap contents?

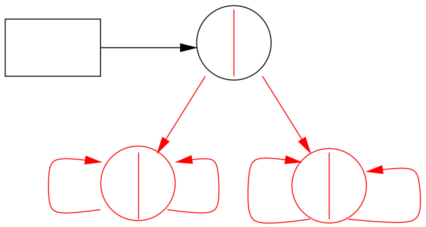
Shapes/Shape Descriptors characterize data structures,
could indicate whether the heap contains



a singly linked list



a doubly linked list

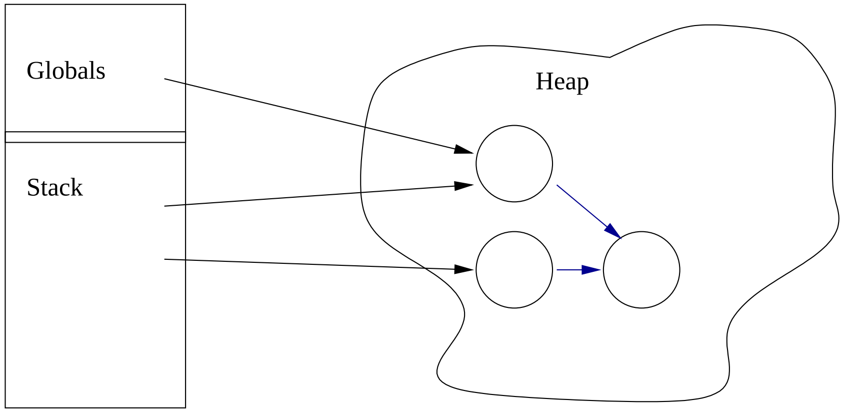


a binary tree

Shape information implies answers to questions above!

A Word about Sharing

- "shared" will mean "heap-shared"
- Sharing through pointer variables represented explicitly (warning: more complicated in the case of interprocedural analysis!)



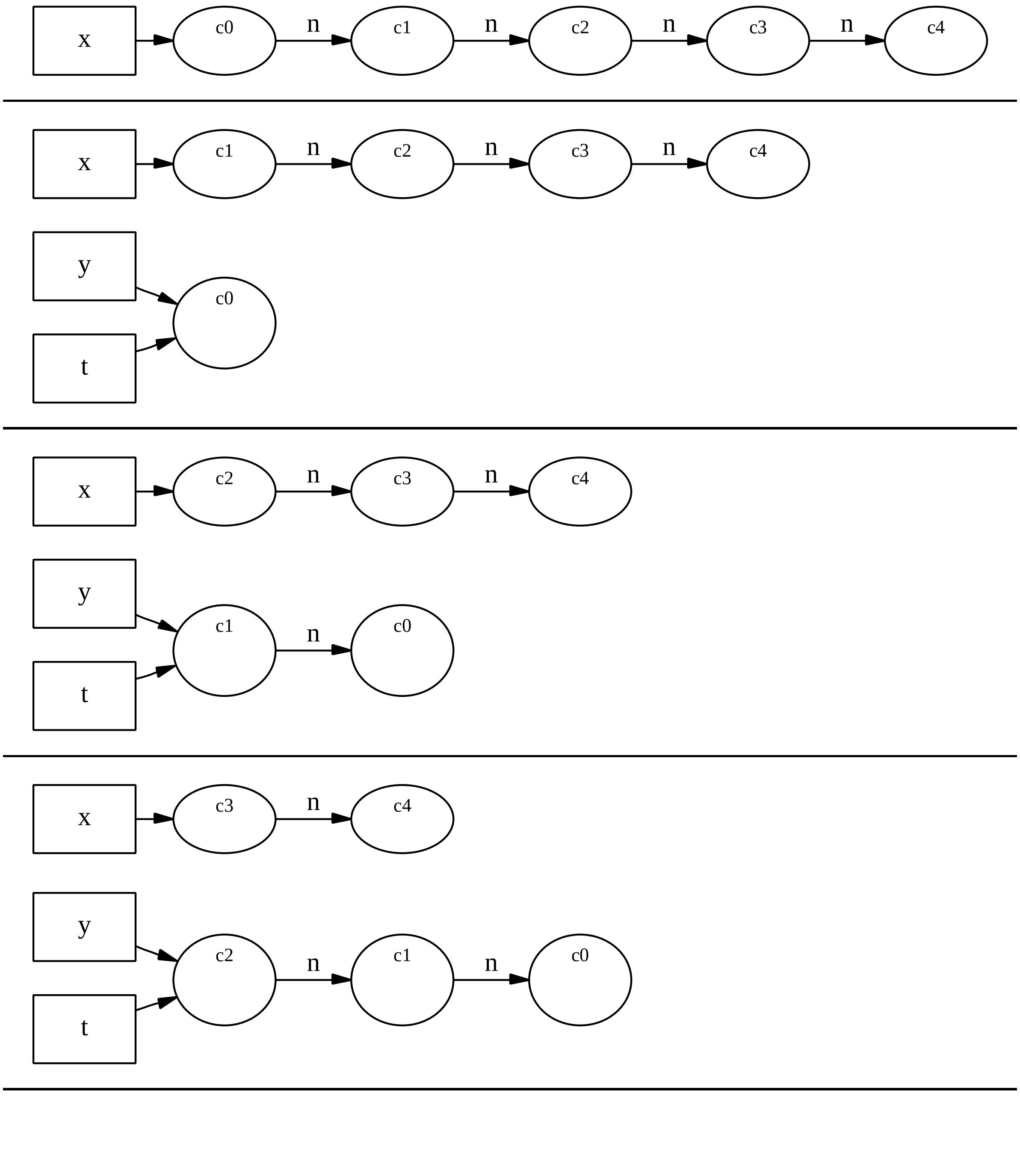
Example: Destructive List Reversal

```
/* reverse.c */
#include "list.h"
List reverse(List x) {
  List y, t;
  y = NULL;
```

```
/* list.h */
typedef struct node {
  struct node *n;
  int data;
} *List
```

```
 while (x != NULL)  {
   t = y;      /* n_3: t holds on to the y-list */
   y = x;      /* n_4: y grabs head of the x-list */
   x = x->n;   /* n_5: x moves to tail of the x-list */
   y->n = NULL;
   y->n = t;   /* new head of y-list connected to tail */
 return y;
}
```

First loop iterations in `reverse` after n_3



Answers

Input: a non-empty acyclic singly linked list of unshared elements

after n_3:

- `x` and `y` point to acyclic singly linked lists; the `y`-list may be empty,
- `t` and `y` are may- and must-aliases or both point to NULL.
- `x` never points to NULL,
- the lists to which `x` and `y` point are disjoint,
- no list element is heap-shared.

after n_4: `x` and `y` are must-aliases, therefore also may-aliases.

after n_5: `x` and `y` are not may-aliases, therefore not must-aliases.

everywhere: there are no garbage cells.

Shape Analysis: Summarization

- programs work for data structures of arbitrary size,
- descriptions of data structures must have bounded size,

$\Longrightarrow$ conservative approximation: *Summarization*

Example

- `reverse` works for lists of arbitrary lengths.
- we'll see: 8 "shape descriptors" describe all `x`- and `y`-lists arising after n_3.

Conservativism

Claim $\mathcal{P} \equiv$ "pointer variable p or pointer component p`->`c never has the value NULL at this program point"

Correctness: $\mathcal{P}$ only results from the analysis if true for all executions of the program and all program paths leading to the program point

Incompleteness: analysis may be unable to derive this information although $\mathcal{P}$ holds

Program analysis is allowed to (only) "err on the safe side."

Shape Analysis

computes for a given program and each point in the program:

> a finite, conservative representation of the heap-allocated data structures that could arise when a path to this program point is executed.

Summarization

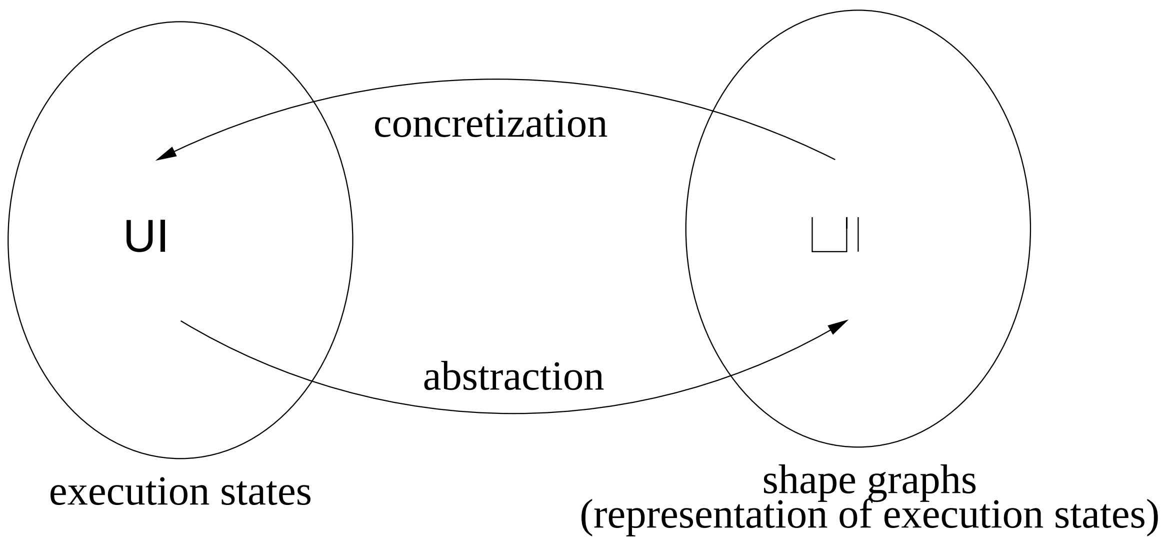
Bounded representation $\Longrightarrow$ loss of information

- Joint representation of some heap cells (lost identity)
- Conservatively represented connectivity among jointly represented heap cells
- Conservatively represented properties of these heap cells:
  - a property that holds for all (for none of the) summarized cells will be found to hold (not to hold) for their summary location
  - a property that holds for some but not all of the summarized cells will have value “don’t know” for the summary location
- Size information: lengths of lists, depths of trees, ...

Shape Analysis is a Static Program Analysis

Abstraction function relates two worlds:

- *concrete world* of program execution states
- *abstract world* of shape graphs, their representation



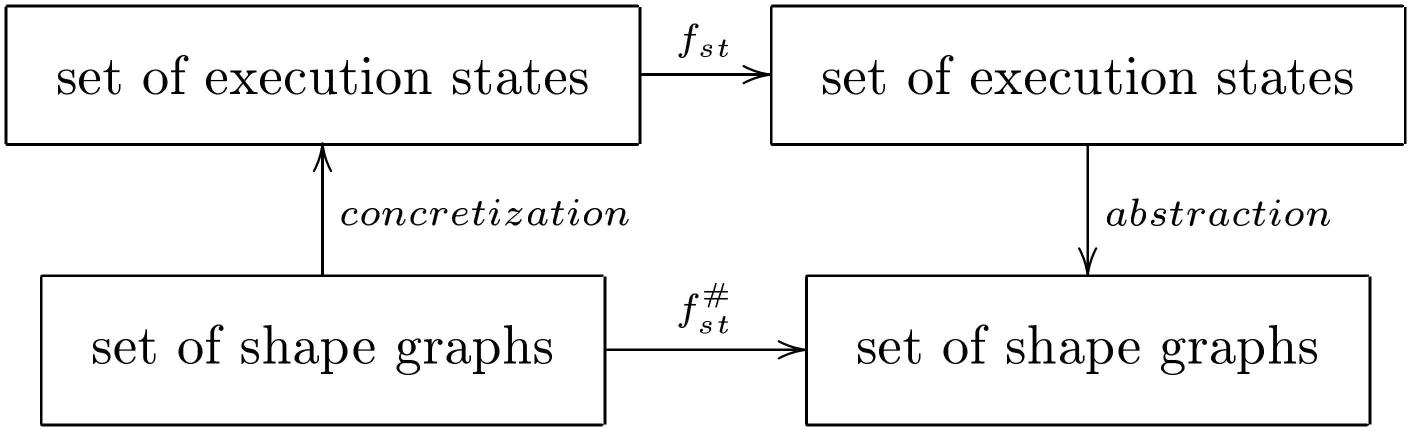
Semantics Based

Concrete semantics given by

- a set of functions f_{st} for each statement st of the language
- f_{st} describes the effect of st on an execution state

Abstract semantics given by

- a set of *abstract transfer functions* $f_{st}^{\#}$ for each statement st
- $f_{st}^{\#}$ describes the effect of st on the repr. of execution states
- Applying $f_{st}^{\#}$ called *abstractly executing st*



The best abstract semantics of a statement *st* a-la-Cousot

Parametric Shape Analysis

"Shape analysis" generic term, stands for a whole class of algorithms. Particular shape-analysis algorithm is determined by a set of properties of heap cells:

Core properties: part of any pointer semantics:

- "pointed-to-by-p" for each program pointer variable p
- "connected-through-c"

Instrumentation properties: additional means of observation,

- depending on the purpose of the analysis
- expressed in terms of the core properties
- for acyclic singly linked lists: sharing, reachability

Core properties in analysis of `reverse`

unary

- "pointed-to-by-`x`", denoted by `x`
- "pointed-to-by-`y`", denoted by `y`
- "pointed-to-by-`t`", denoted by `t`

binary

- "connected-through-`n`" denoted by `n`

Instrumentation properties in the analysis of `reverse`

- "is-heap-shared", denoted by `is`
- "reachable-from-`x`-through-`n`", denoted by `r[n,x]`
- "reachable-from-`y`-through-`n`", denoted by `r[n,y]`
- "reachable-from-`t`-through-`n`", denoted by `r[n,t]`

A property existing in every shape analysis is "not-summarized"

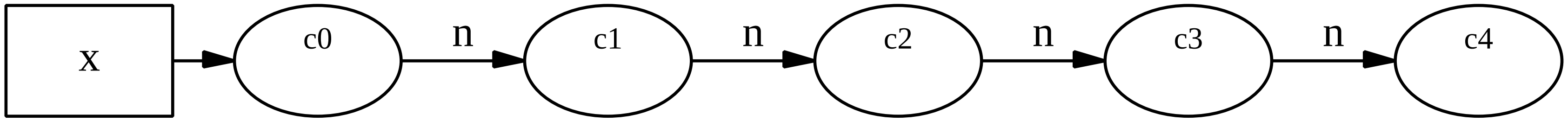
Abstraction Functions

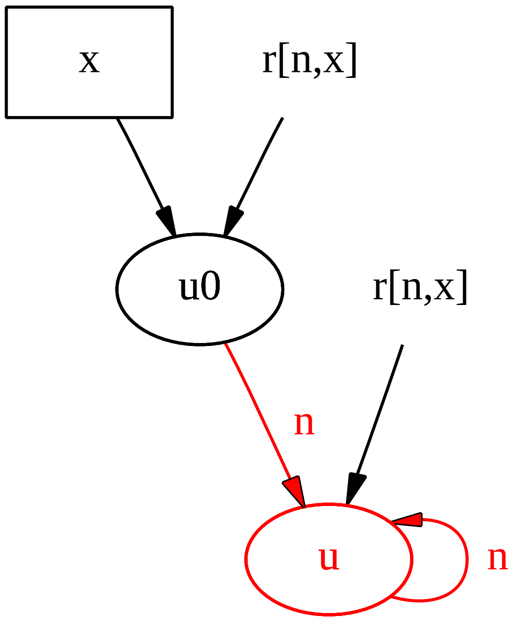
Determined by the *abstraction properties*,
a subset of the unary core and instrumentation properties.

Abstraction principle:

> Heap cells that have the same definite values (true, false) for the abstraction properties are summarized to the same abstract location.

Heap cells summarized by one summary location have no observable difference.





Abstract location u represents all the cells in the tail of the input list since

- all have the property `r[n,x]`
- all do not have the properties `x`, `y`, `t`, `r[n,y]`, `r[n,t]`, `is`.

Abstract location $u0$ represents exactly the first cell of the input list since it

- has the properties `x` and `r[n,x]`
- has no other of the abstraction properties.

The Legend

$\boxed{\mathbf{q}}$ represents pointer variable q

m represents exactly one heap cell (”not-summ.”)

m may represent one or more heap cells

$\mathbf{m} \xrightarrow{\mathbf{c}} \mathbf{m'}$ the c-component of the m-cell points to the m’-cell

$\mathbf{m} \xrightarrow{\mathbf{c}} \mathbf{m'}$ the c-component of one of the m-cells may point to one of the m’-cells

$m = m'$: edge may or may not represent a cycle.

$\mathbf{p} \longrightarrow \mathbf{m}$ properties possessed by the m-cell

$\mathbf{p} \longrightarrow \mathbf{m}$ properties possessed by all m-cells

$\mathbf{p} \longrightarrow \mathbf{m}$ properties possibly possessed by the m-cells

no edge cells dont possess property

Legend Summary

The shape graphs portray information of three kinds:

solid black meaning “always holds” for properties (incl. ”not-summ.”),

absent meaning “never holds” for properties, and

red meaning “don’t know” for properties (incl. ”not-summ.”),

Three Values

- *definite* values

  “always holds” corresponds to *true*,

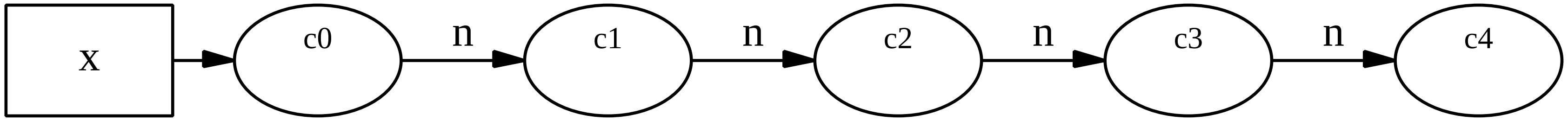
  “never holds” corresponds to *false*,

- *indefinite* value

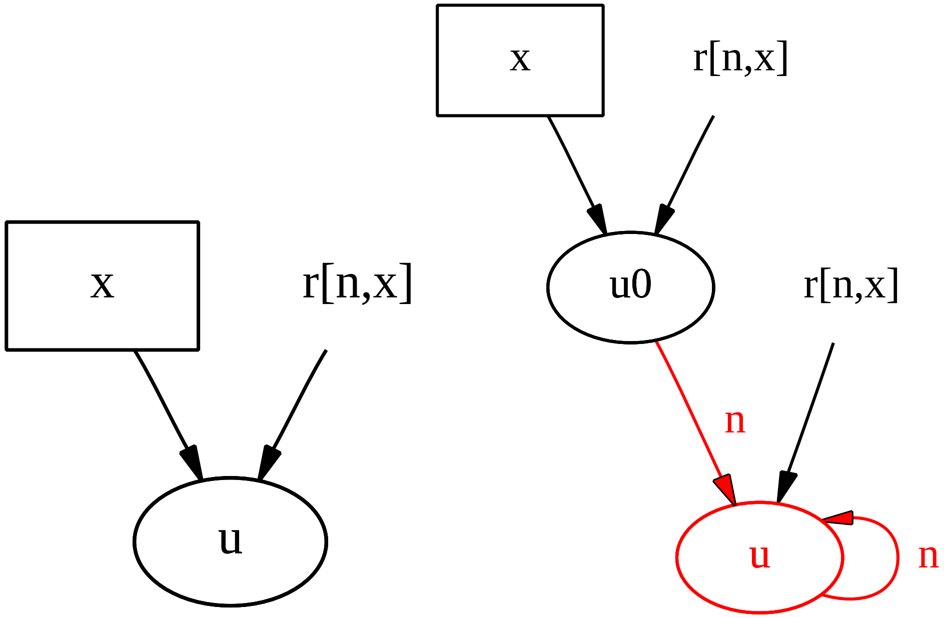
  “don’t know” corresponds to the third value in Kleenes three-valued semantics (“potentially true).

Input to `reverse`: Concrete and Abstract

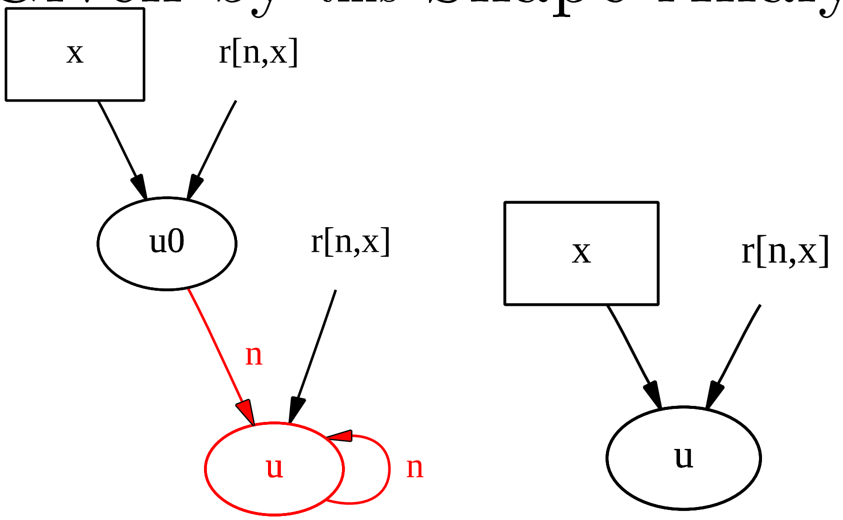
one concrete:



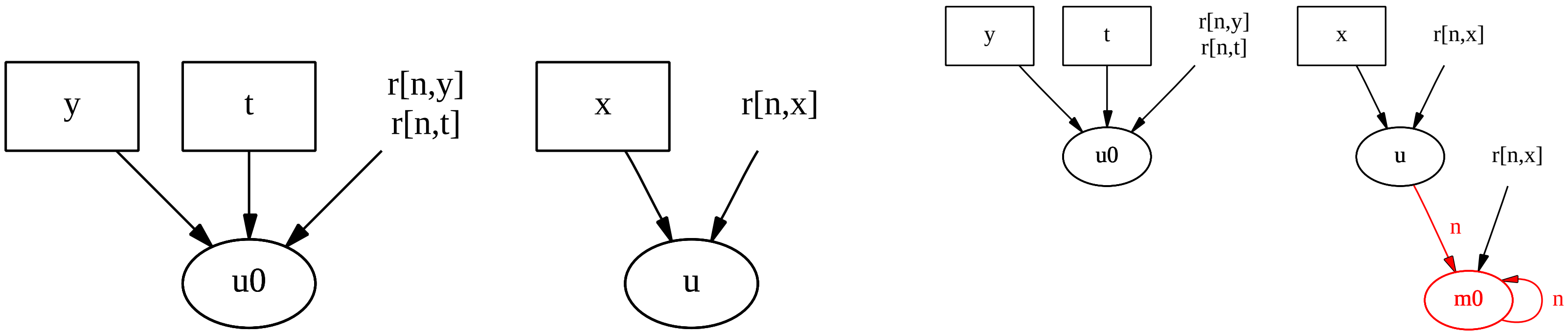
all abstract descriptions of all concrete inputs:



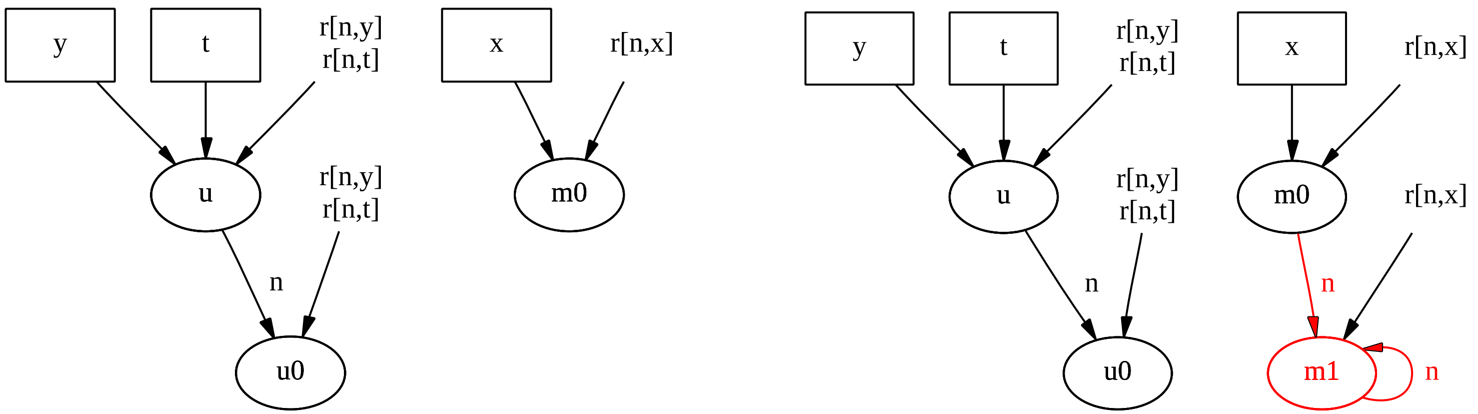
Answers as Given by this Shape Analysis after n_3



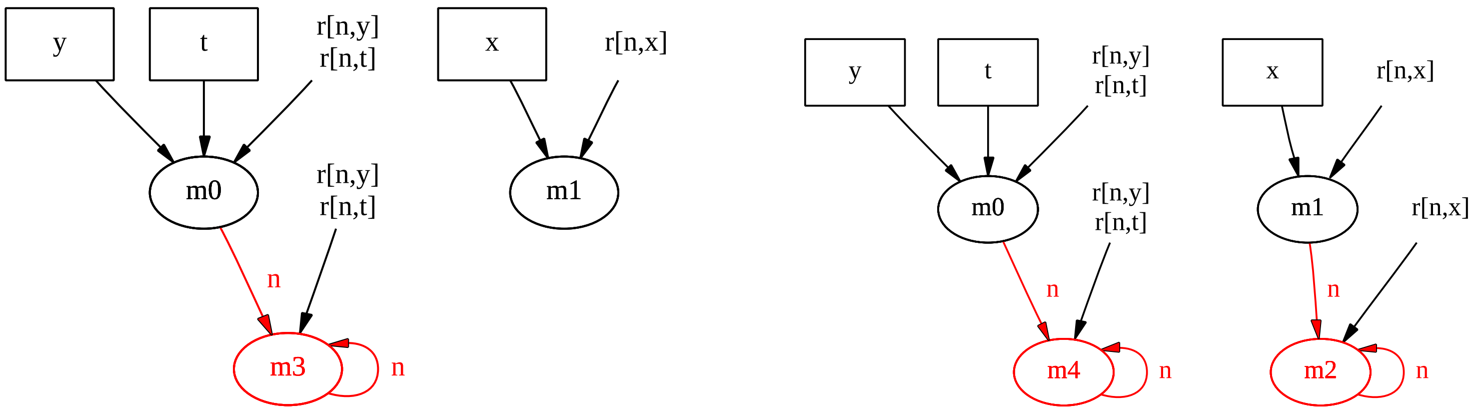
Empty y-list



One-element y-lists



Two-element y-lists



Longer y-lists

Concretization

At a program point	describes
A shape graph	a set of possible execution states
Set of all graphs	(a superset of) all the possible execution states

To check claims about the properties of the heap contents at some program point:
Do they hold for all the shape graphs?

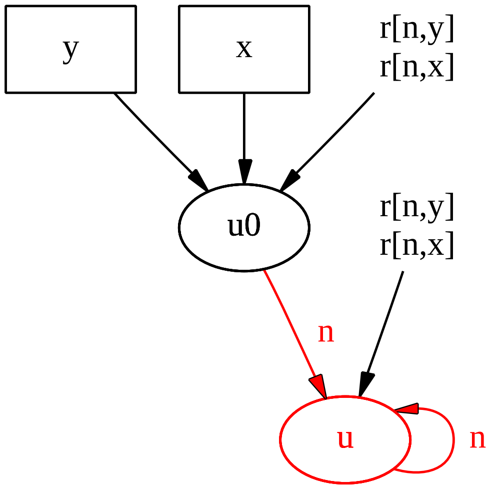
Analyzing a Statement

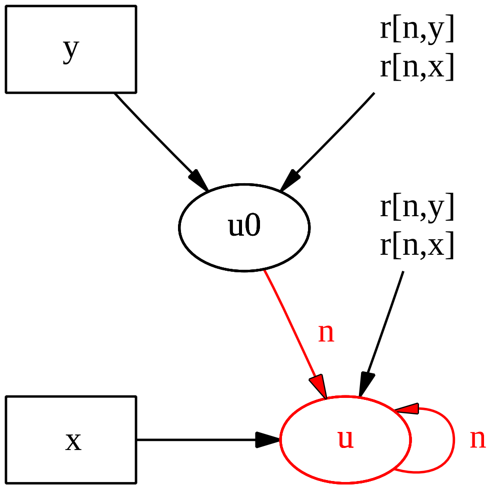
Abstract semantics of the pointer statements, $f^{\#}_{st}$, specified in some formalism (here logic):
Change of the core and the instrumentation properties when a statement is executed

Most complex statement `x = x -> n`:

Semantics described by means of an example

First Attempt





describes some of the lists that can arise before program point n_5

the resulting shape graph resulting from (naively) executing **`x=x->n`**

Unacceptable shape graph!

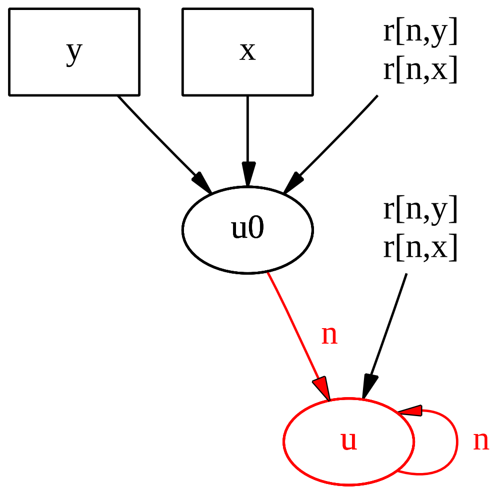
- not in the image of the abstraction function
- summarized abstract location representing one or more heap cells cannot have the definite property "pointed-to-by-`x`"
- semantics of languages with pointers implies that any pointer variable or pointer component can only point to at most one cell at any one time
- only two cases are possible:
  - `x` points to NULL, or
  - `x` points to exactly one cell

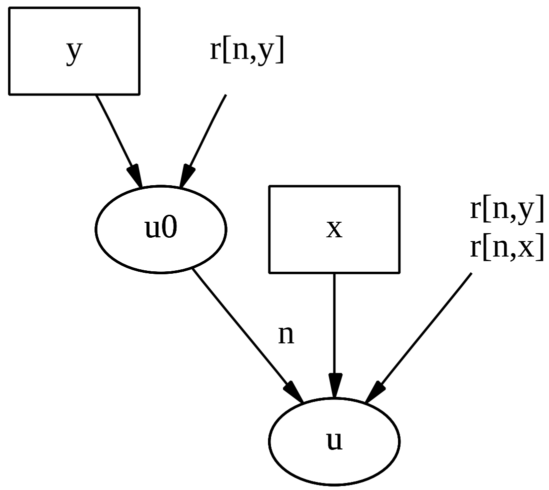
Better approach: case analysis of the reasons why u's "not-summarized" property and the two `n`-edges are indefinite

Case 1:

- a definite **n**-edge exists starting at $u0$, and
- u represents exactly one heap cell
- the cell represented by u is definitely not shared
- $\Longrightarrow$ back-edge from u to u cannot represent a concrete back pointer
- $\Longrightarrow$ indefinite back-edge from u to u has to be eliminated

Now apply predicate update formulae

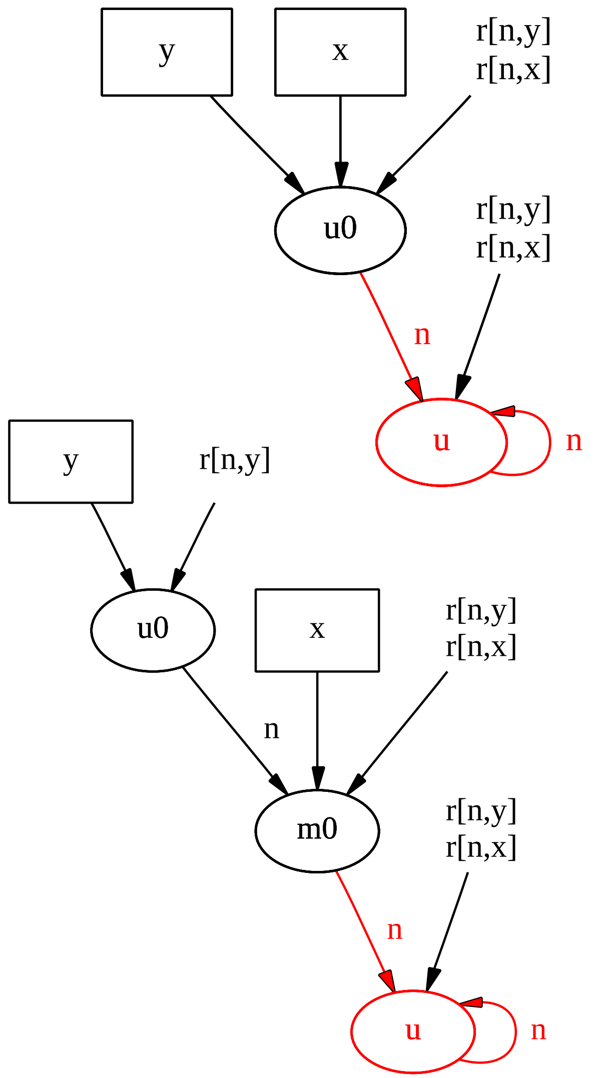




Case 2:

- a definite `n`-edge exists starting at $u0$
- u represents more than one heap cell
- new definite abstract location, $m0$, is “materialized” out of u
- represents the first of the cells in the tail of the `x`/`y`-list
- definite edge starting at $u0$ is made to point to it
- same argumentation about a potential back-edge to it

Now apply predicate update formulae



Case 3:

- the indefinite `n`-edge representing the absence of a pointer from the cell represented by $u0$ to any of the cells represented by u
- incompatible with the fact that the `r[n,x]` and `r[n,y]` properties of u have the (definite) value 1
- $\Longrightarrow$ this case is impossible

Foundation of the Parametric Approach

Specification Language: 1st order logic with transitive closure

Constituents of a specification:

- core predicates
- instrumentation predicates
- abstraction predicates
- predicate update formulae

Core Predicates

Predicate	**Intended Meaning**
$x(v)$	Does pointer variable `x` point to element v?
$sm(v)$	Does element v represent more than one concrete element?
$n(v_1, v_2)$	Does the `n` field of v_1 point to v_2?

Instrumentation Predicates

Predicate	**Intended Meaning**	**Defining Formula**
$r[n, x](v)$	Is v reachable from program variable `x` using component `n`?	$\exists v_1 : (x(v_1) \wedge n^*(v_1, v))$
$c[n](v)$	Does v reside on a directed cycle of `n`-components?	$n^+(v, v)$
$is[n](v)$	Is v pointed to by more than one `n`-component?	$\exists v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2$

some Predicate Update Formulae

st	φ_p^{st}
`x = NULL`	$\varphi_x^{st}(v) \stackrel{\text{def}}{=} \mathbf{0}$ $\varphi_z^{st}(v) \stackrel{\text{def}}{=} z(v)$, for each $\mathtt{z} \in (PVar - \{\mathtt{x}\})$ $\varphi_n^{st}(v_1, v_2) \stackrel{\text{def}}{=} n(v_1, v_2)$
`x = t->n`	$\varphi_x^{st}(v) \stackrel{\text{def}}{=} \exists v_1 : t(v_1) \wedge n(v_1, v)$ $\varphi_z^{st}(v) \stackrel{\text{def}}{=} z(v)$, for each $\mathtt{z} \in (PVar - \{\mathtt{x}\})$ $\varphi_n^{st}(v_1, v_2) \stackrel{\text{def}}{=} n(v_1, v_2)$
`x->n = t` (assuming `x->n == NULL`)	$\varphi_z^{st}(v) \stackrel{\text{def}}{=} z(v)$, for each $\mathtt{z} \in PVar$ $\varphi_n^{st}(v_1, v_2) \stackrel{\text{def}}{=} n(v_1, v_2) \vee (x(v_1) \wedge t(v_2))$

Concrete and Abstract Semantics

Concrete Semantics:

Execution State: Two-valued interpretation of the predicates
Statement Execution: “Application” of the predicate update formulae.

Abstract Semantics:

Description of the execution state: Three-valued interpretation of the predicates
Abstract Statement Execution:

1. Focusing on some predicates (case analysis above);
2. Application of predicate update formulae;
3. Cleaning up (semantic reduction)

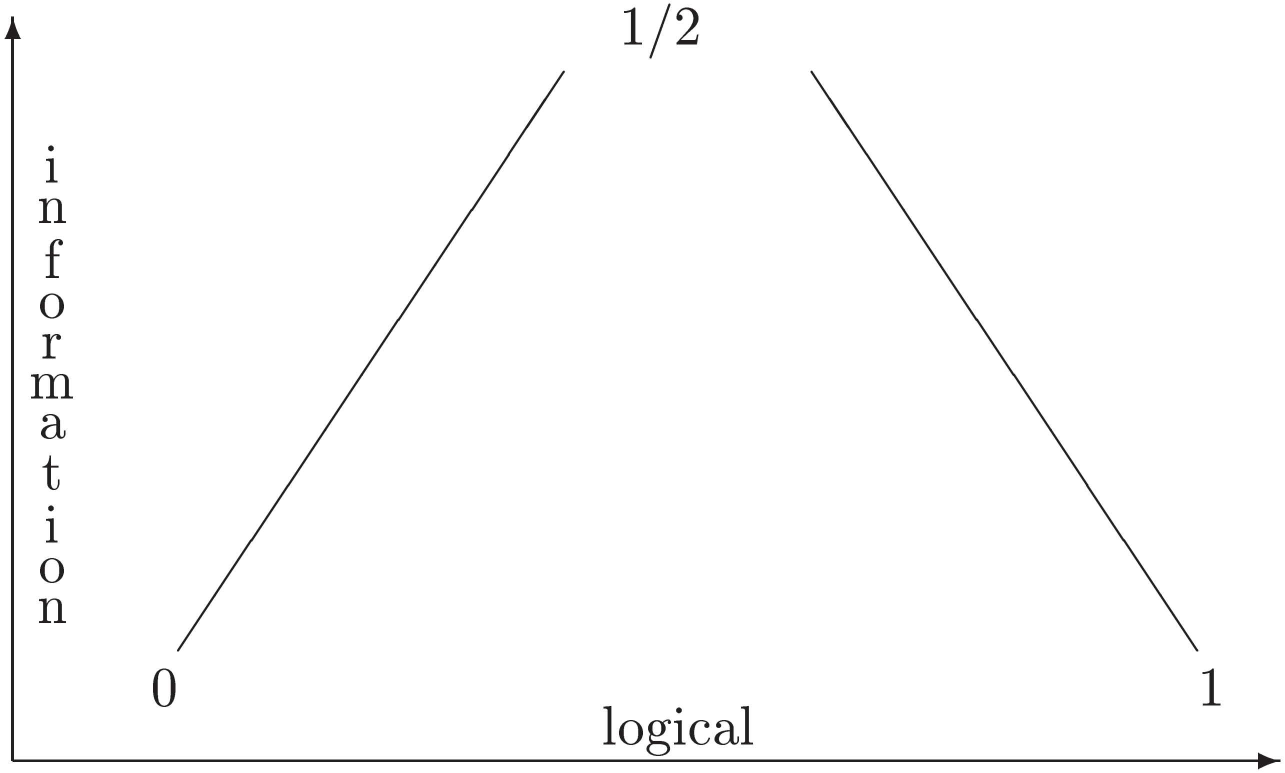
NB: Different interpretations of the same formulae!

Kleene's Three-valued Logic

The semi-bilattice of three-valued logic:

definite values 0, **1**

designated values 1/2, **1** (potentially true)



Embedding

Relates 2-valued and 3-valued interpretations S_1 and S_2 by a surjective function between their universes.

Properties:

1. no contradiction in definite values of predicates, and
2. effect of summarization reflected in value of sm.

Tight Embedding

1. compute values in S_2 as least upper bound of values in S_1,
2. use least upper bound on summarization information.

The Embedding Theorem

Interpretation S_1 embedded into interpretation S_2.

Values of formulae in S_2 do not contradict the ones in S_1.

Implementation and Experience

Implementation (TVLA-engine)

- by Tal Lev-Ami at Tel-Aviv University
- in Java
- good for experimenting with analysis designs.
- Small examples like `reverse` can be analyzed in a few seconds on a Pentium 166 MHz PC.
- Bigger examples require patience.
- Surprising experience: more precise analysis does not necessarily cost more time and space.

Applications

- Dependence Analysis, Ross/Sagiv, ESOP'98
- Compiler Optimizations, Ghiya/Hendren, POPL'98
- Paralellization, Corbera/Asenjo/Zapata, ICS'99
- Memory Cleanness Checking, Dor/Rodeh/Sagiv, SAS'2000
- Focusing in Algorithm Explanation, Braune/Wilhelm, IEEE TVCG'2000
- Program Verification

Program Verification

Lev-Ami/Sagiv/Reps/Wilhelm'2000:

Proof of partial correctness of abstract data type *Sorted linked list*.

Operations:

- sort (insertion-sort, bubble-sort),
- delete,
- insert,
- merge,
- reverse

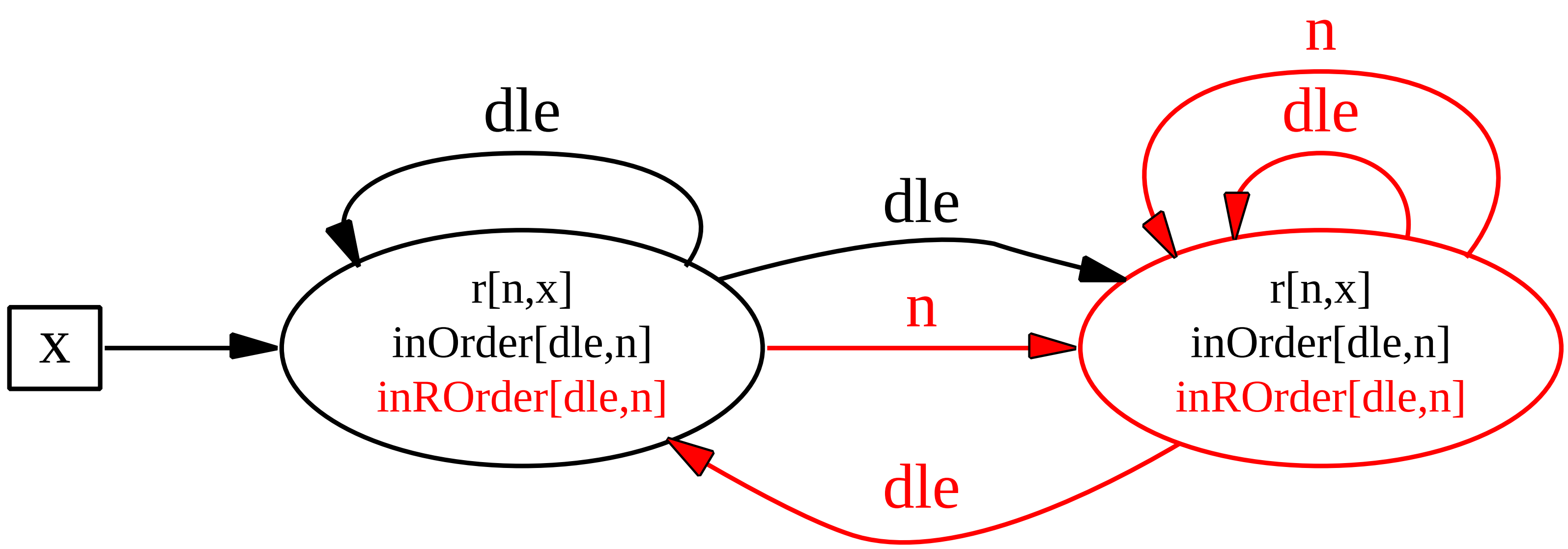
The Core Predicates

Predicate	**Intended Meaning**
$x(v)$	Is v pointed to by variable `x`?
$n(v_1, v_2)$	Does the `n`-component of v_1 point to v_2?
$dle(v_1, v_2)$	Is the `d`-component of v_1 less-than-or-equal-to the `d`-component of v_2?

The Instrumentation Predicates

Predicate	**Intended Meaning**	**Defining Formula**
$r[n,x](v)$	Is v reachable from variable `x` using component `n`?	$\exists v_1 : (x(v_1) \wedge n^*(v_1, v))$
$c[n](v)$	Is v on a directed cycle of `n`-components?	$n^+(v, v)$
$is[n](v)$	Is v pointed to by more than one `n`-component?	$\exists v_1, v_2 : n(v_1, v) \wedge n(v_2, v)$ $\wedge v_1 \neq v_2$
$inOrder[n, dle](v)$	Is v in place (non-decr.)?	$\forall v_1 : n(v, v_1) \Longrightarrow dle(v, v_1)$
$inROrder[n, dle](v)$	Is v in place (non-incr.)?	$\forall v_1 : n(v, v_1) \Longrightarrow dle(v_1, v)$

Shape Graph for a Sorted List



Conclusions

Parametric Framework for Shape Analysis

specification: properties of interest of heap cells,

1st order logic as specification language,

generative mechanism: produces the instance analysis,

TVLA is a prototype implementation.

Different instantiations of the framework

- use different classes of shape graphs,
- identify different kinds of linked data structures

Presented instantiation of the framework is able to analyze

- the `reverse` program and
- many other programs on singly linked lists.

This instantiation would fail on programs on doubly linked lists,

An instantiation for doubly linked lists would need the properties:

- forward-backward-pairing,
- backward-forward-pairing

Two Values vs. Three Values

Classical program analysis uses two Boolean values. It is

- *definite* on one of the values and
- *conservative* on the other.

Program analysis based on 3-valued logic,

- definite truth and definite falseness both tracked,
- the third value captures indefiniteness.