Shape Analysis

Tal Zelmanovich

Seminar in automatic tools for analyzing programs with dynamic memory 2013/2014B

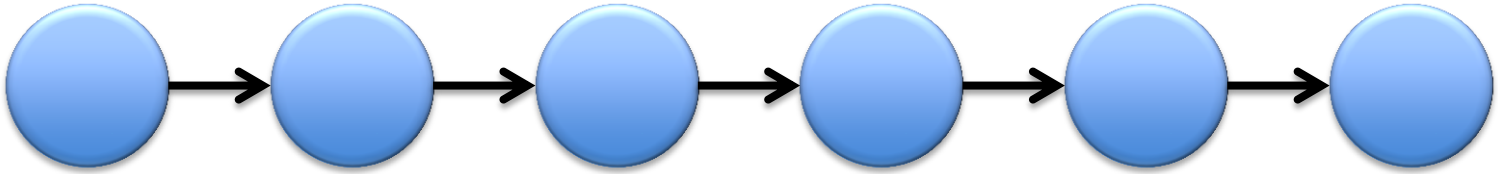
Subjects

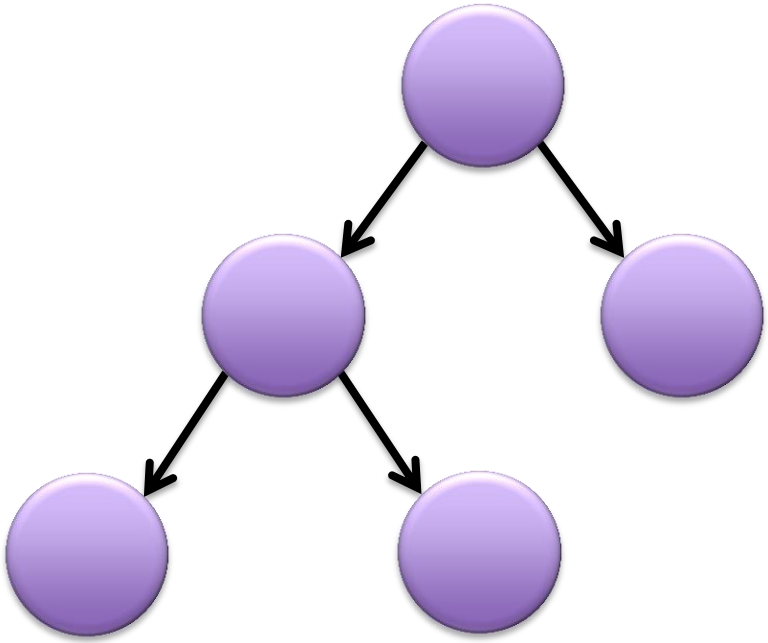
- Introducing shape analysis
- TVLA method
- Cutpoint-free method
- Separation logic method
- Conclusion & Personal view

Part 1 – General shape analysis

- The idea behind shape analysis
- Goals
- Analysis scope & limits
- Termination problem
- Common definitions & symbols

What is the best way to describe a list or a binary tree?





The concept

Analyze program behavior through shapes of data structures occurring in the heap

- In-depth analysis that answers advanced questions about the program
- Static analysis
- No single algorithm – a family of methods with common principles

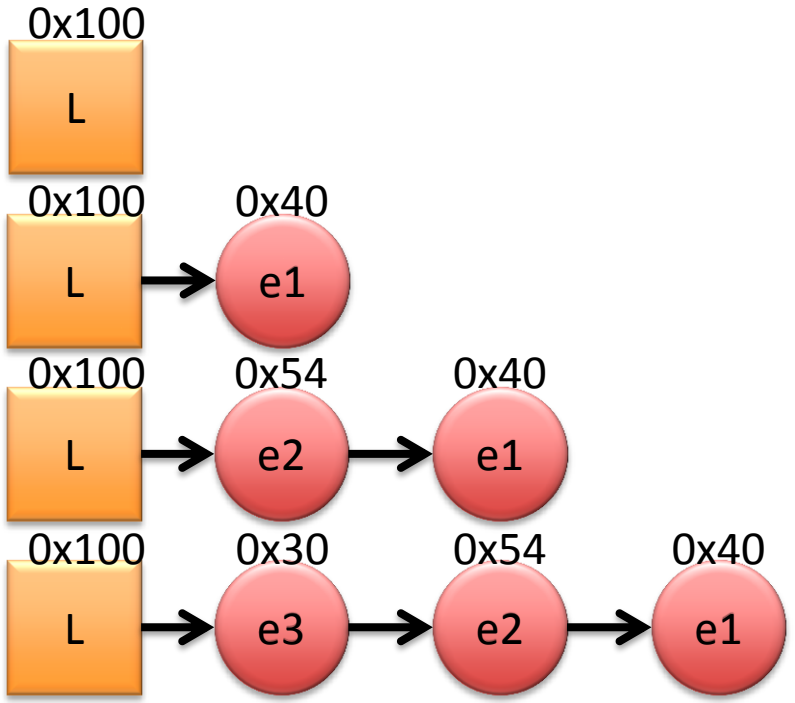
The concept

Structures are usually kept as pointing-graphs or logical statements

Example:

```
void three_func()
{
    List_element * L = NULL;
    for (int i=0; i<3; i++)
        L = append_element(L, i)
}
```

Possible states inside loop:



Goals

The analysis allows us to answer some common pointer-analysis questions:

- Does a pointer points at NULL?
- Are two pointers aliasing?
- Can we reach y from x?
- Is there an access violations?

Using shape analysis we can get answers about both stack pointers and heap locations

Goals

Shape analysis also answers more complicated questions:

- How many places points to a single location?
- Is x a part of a pointing cycle?
- Is there a memory leak?
- Does x points to a list\double list\tree?

In some shape analysis methods it is even possible to define questions\properties on our own

Analysis scope

Shape analysis may be a part of a complete analysis system, but the basic version cannot answer questions about:

- Pointer arithmetic
- Arrays
- Data values (follows pointer only)
- Flow questions (is code reachable?)

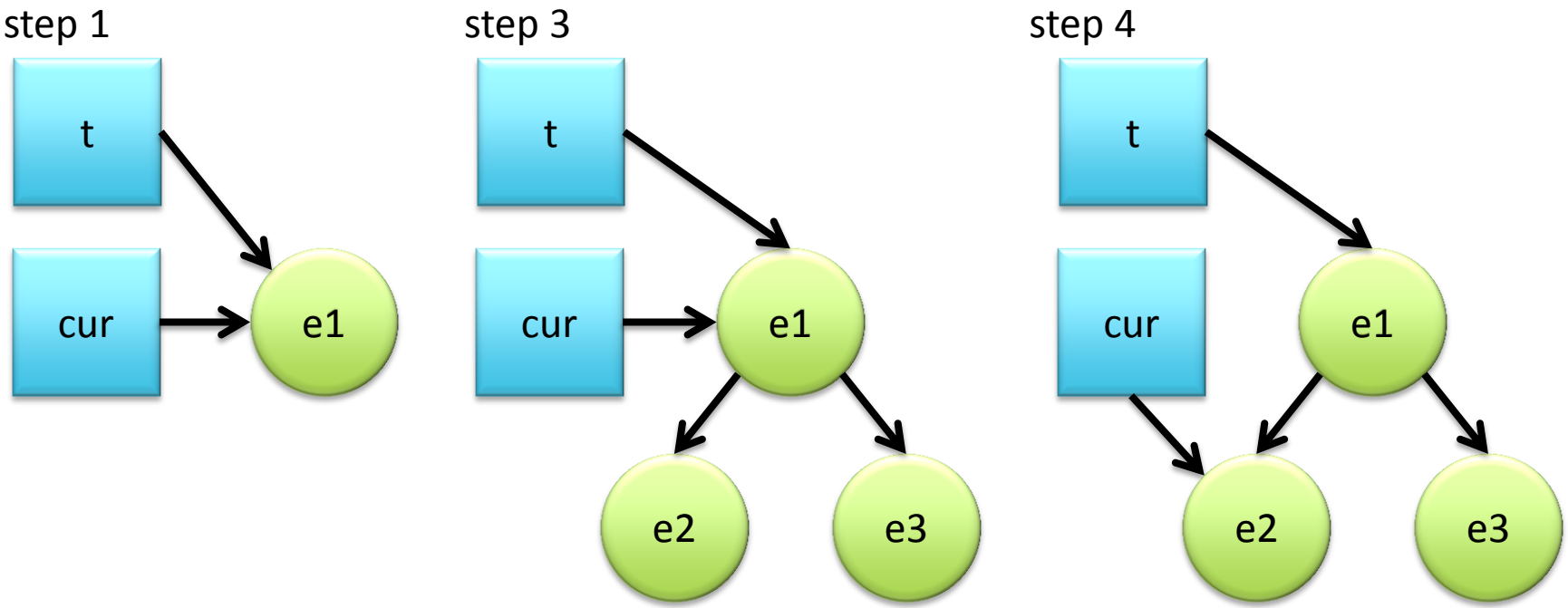
It only gives info about memory structures!

Analysis example

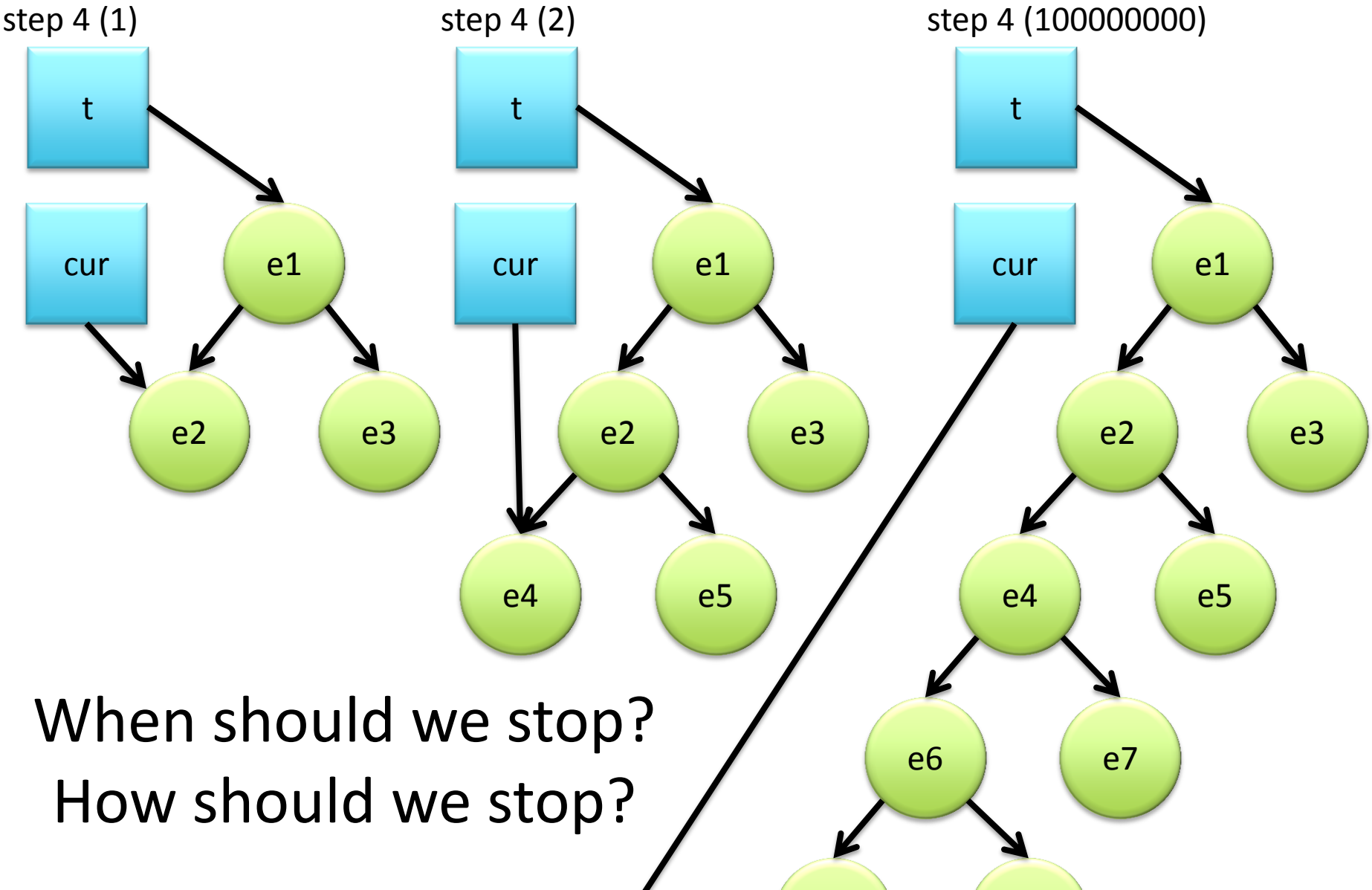
```
struct Tree {int data = DC, Tree * left = NULL, Tree * right = NULL};

Tree * generate_tree(int times)
{
    Tree * t = new Tree();
    Tree * cur_node = t;
    for (int i=0; i<times; i++)
    {
        Tree * left_son = new Tree();
        Tree * right_son = new_Tree();
        cur_node->left = left_son; cur_node->right = right_son;
        cur_node = cur_node->left
    }
    return t;
}
```

Analysis example

1. Tree * t = new Tree(); Tree * cur_node = t;
2. for (int i=0; i<times; i++)
   …
3. cur_node->left = left_son; cur_node->right = right_son;
4. cur_node = cur_node->left



Analysis example



When should we stop?

How should we stop?

Summarization

Recall abstraction from a few lectures ago:

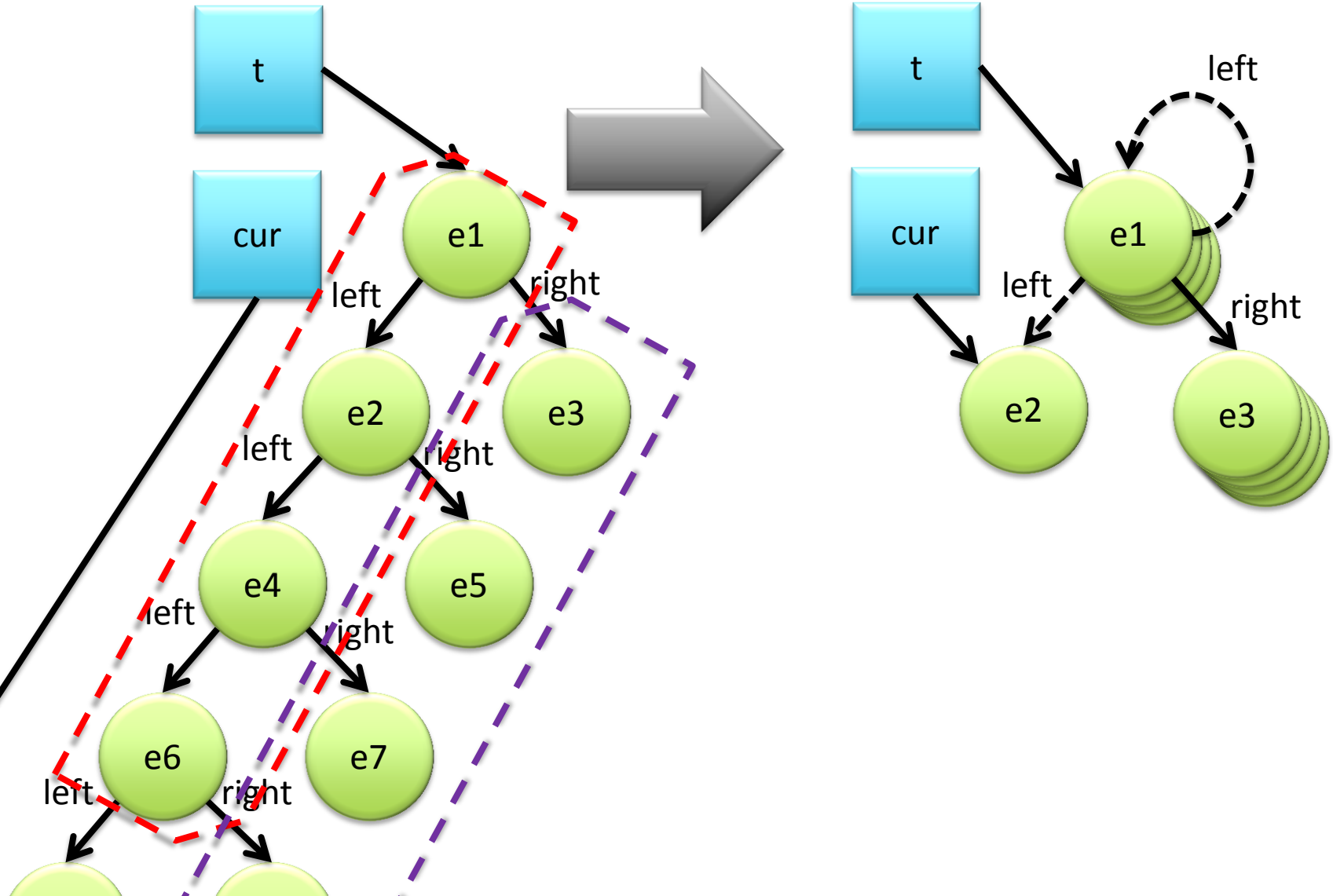
- {1,2,3} → [1,3]
- {1,2,3} → T

How can we do the same for pointing graphs?

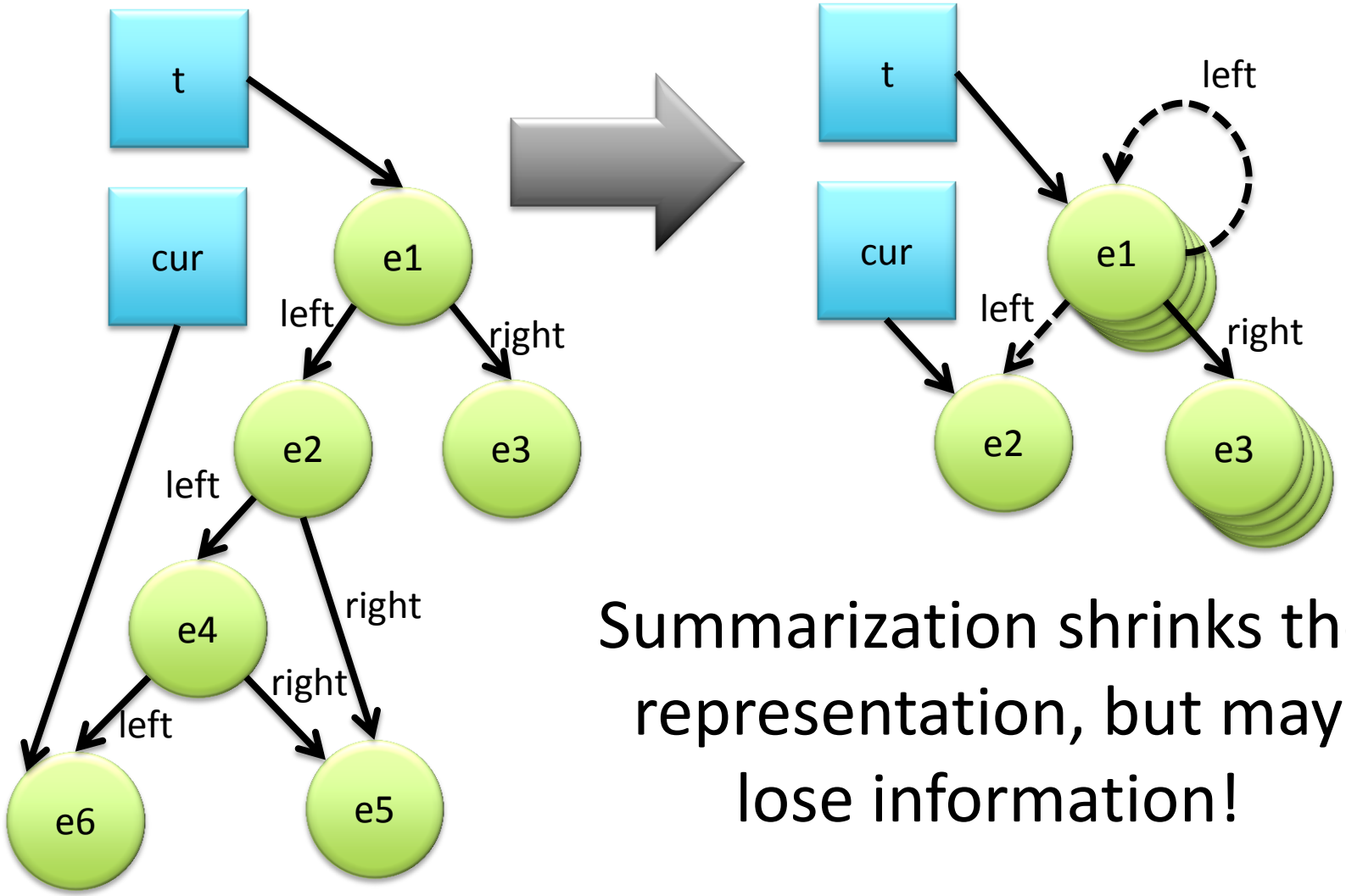
Summarize – represent memory locations with similar connectivity attributes as one node\place

Summarization allows us to treat a set of (possibly infinite) graphs as if it was a single graph

Summarization

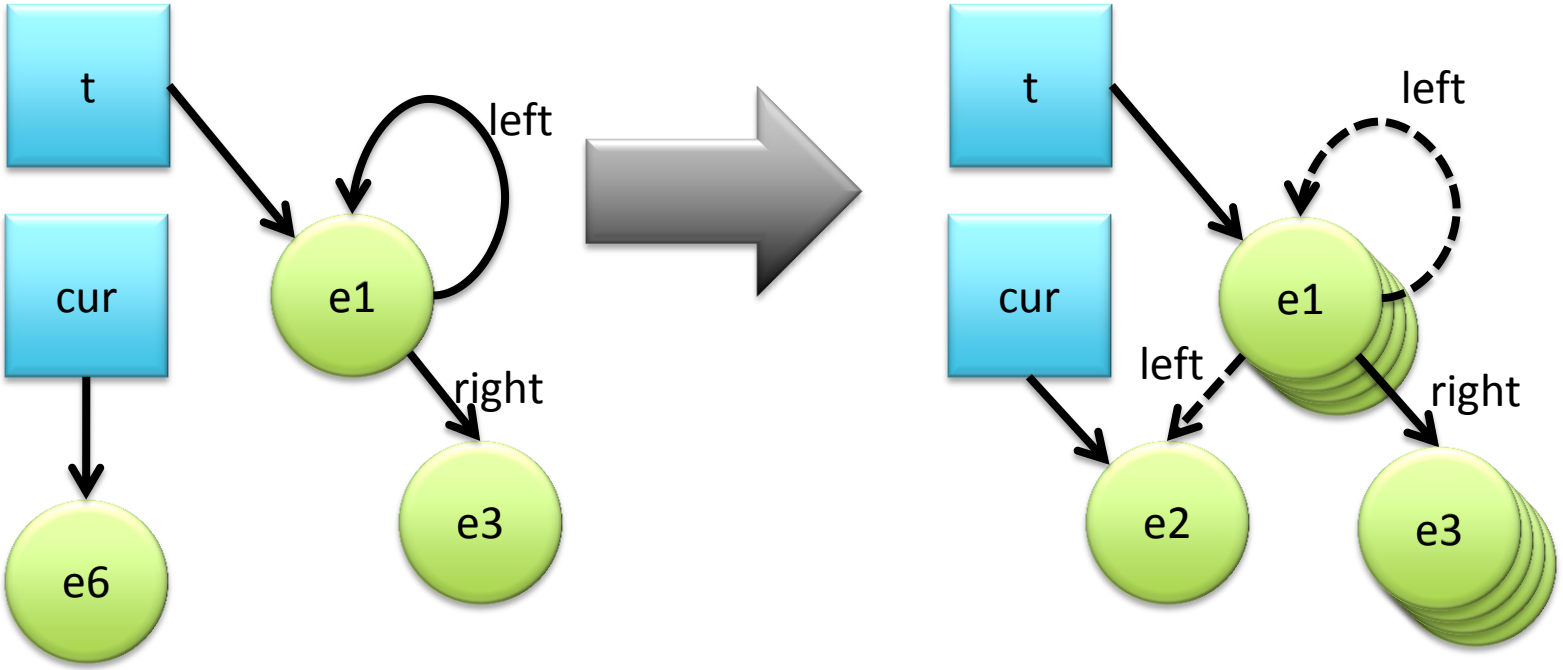


Summarization



Summarization shrinks the representation, but may lose information!

Summarization

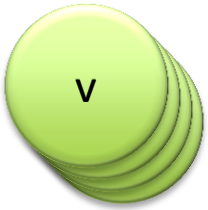


A good summarization method must keep the traits we care about correct

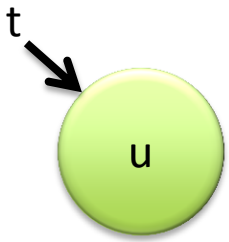
Symbols & conventions

Pointer placed on stack

Single heap cell\struct

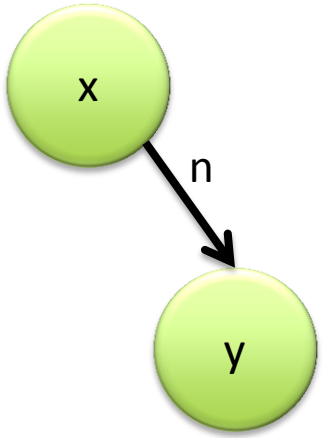


Collection of heap cells\structs (at least 1)

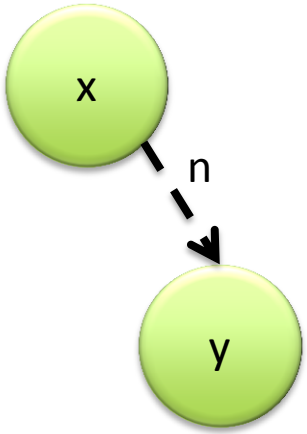


Has attribute t (examples: points_to_NULL, is_on_cycle, reachable_from_pointer_P)

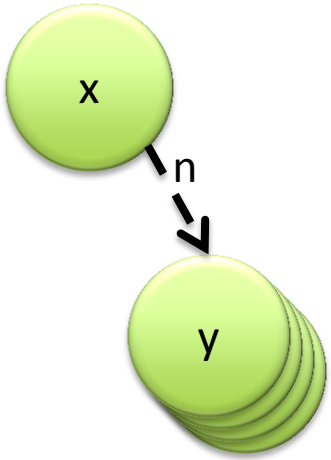
Symbols & conventions



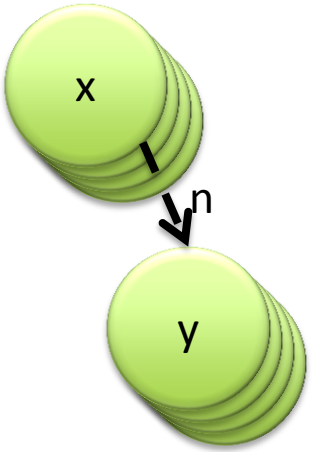
x points to y by n field



x may point to y by n field



x may point to one element of y by n field



Some elements of x may point to some elements of y by n field

Part 2 – the TVLA method

- About the TLVA method
- 3 valued – logics
- Predicates used in TLVA
- Command translation in TLVA
- Special uses and versions of TLVA
- Runtime & bottleneck

The TVLA method

- Method: Mooly Sagiv, Tom Reps & Reinhard Wilhelm
- Tool: Mooly Sagiv, Tal Lev Ami & Roman Manevich

Three valued logic

- Instead of {T, F} use {1, ½, 0} where ½ means "don't know"
- Expressions are evaluated as expected:
  - $T \wedge \frac{1}{2} = \frac{1}{2}$
  - $T \vee \frac{1}{2} = T$
- Attributes and connections may have value ½ (represented as dotted lines in graphs)

Predicates

- Attributes and connections are represented as unary and binary predicates operating on heap locations
- Core predicates – basic shape analysis properties such as points-to
- Instrumentation predicates – additional properties we'd like to follow (reachability for example)
- Predicates have {0, ½, 1} values

Core predicates

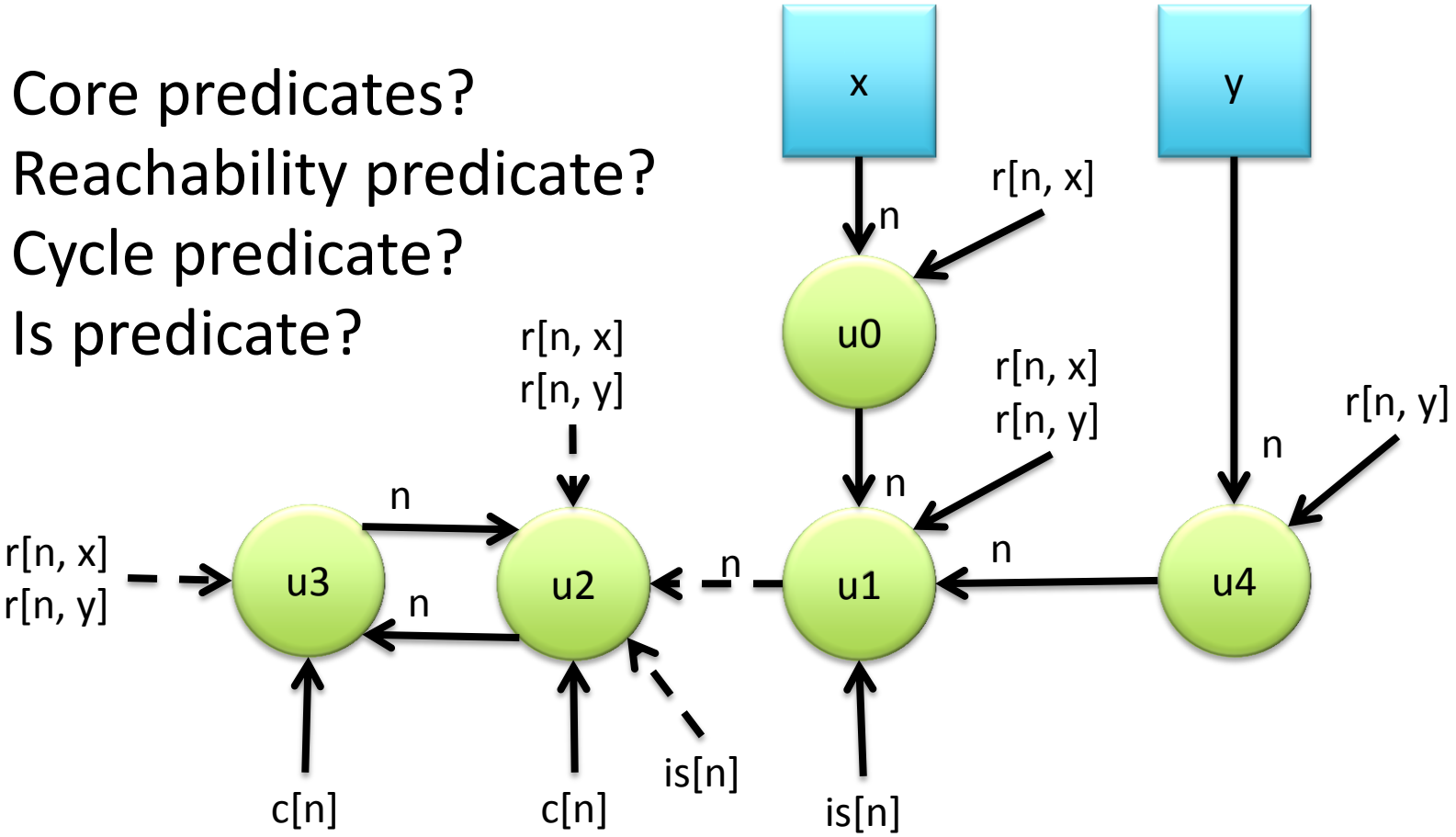
- points_to_by_x(y) – stack pointer x points to heap location y
- connected_through_n(x,y) – n property of heap location x points to y
- sm(x) – special predicate stating whether x is a summarized location (cannot be ½)

Examples of instrumentation predicates

- r[n, p](x) – location x can be reached by going throw n-fields of stack pointer p
- Is_Null(x) – x is not an actual heap location, but NULL
- Is[n](x) – is x heap shared, meaning does more then one element points to x
- c[n](x) – x is a part of a cycle using n field
- we can even define instrumentation predicates of our own

Predicates

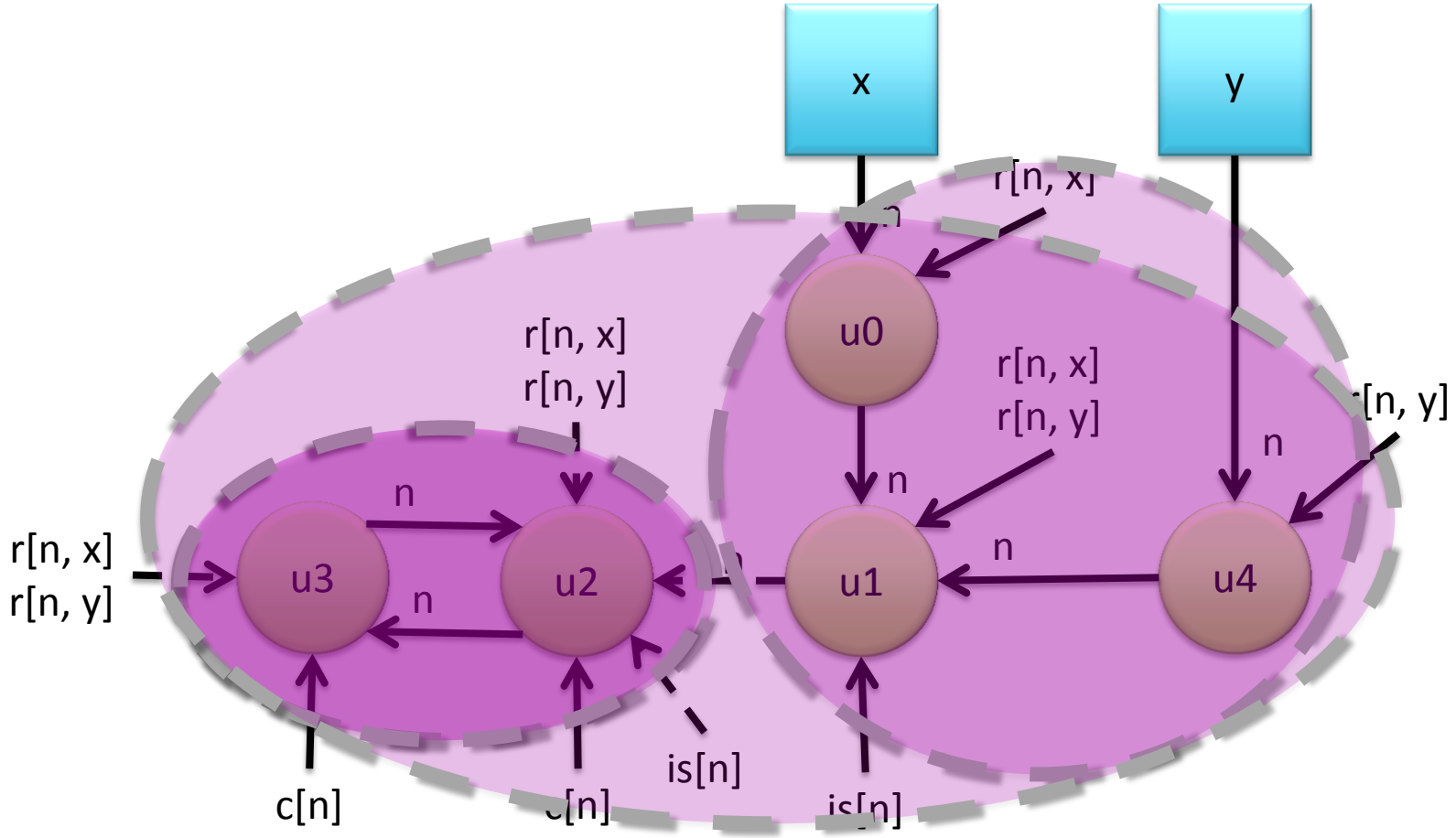
Core predicates?
Reachability predicate?
Cycle predicate?
Is predicate?



Summary operation

- In TVLA summary is done by grouping together connected elements sharing the same set of abstraction predicates
- abstraction predicates are a set of unary predicates (can be chosen however you like)
- abstraction predicates are the properties that summary will conserve
- more abstraction predicates means better analysis and usually (although not always) longer running time

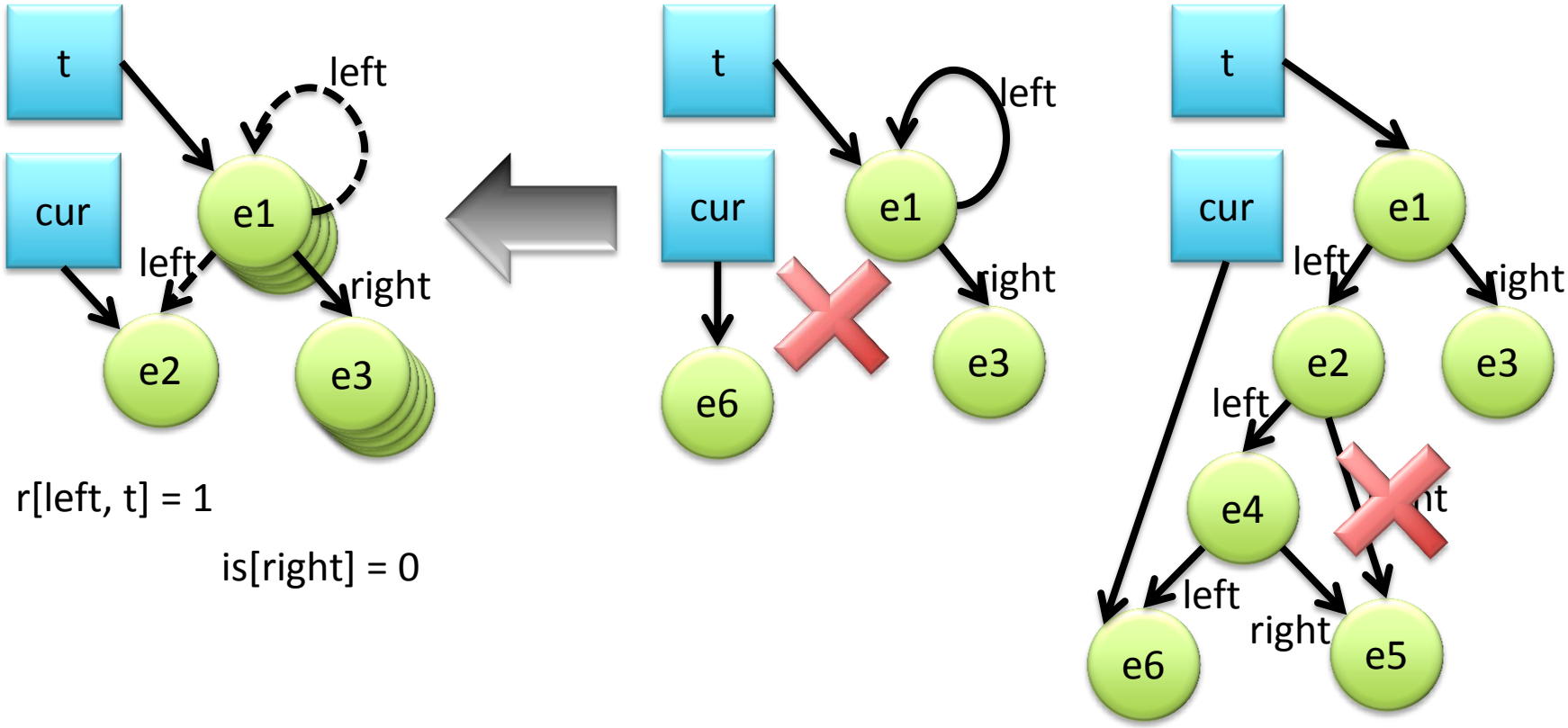
Summary operation



Possibilities for abstraction predicates:

{r[n,x], r[n,y]} {c[n]} {}

Revisit: summary information lost

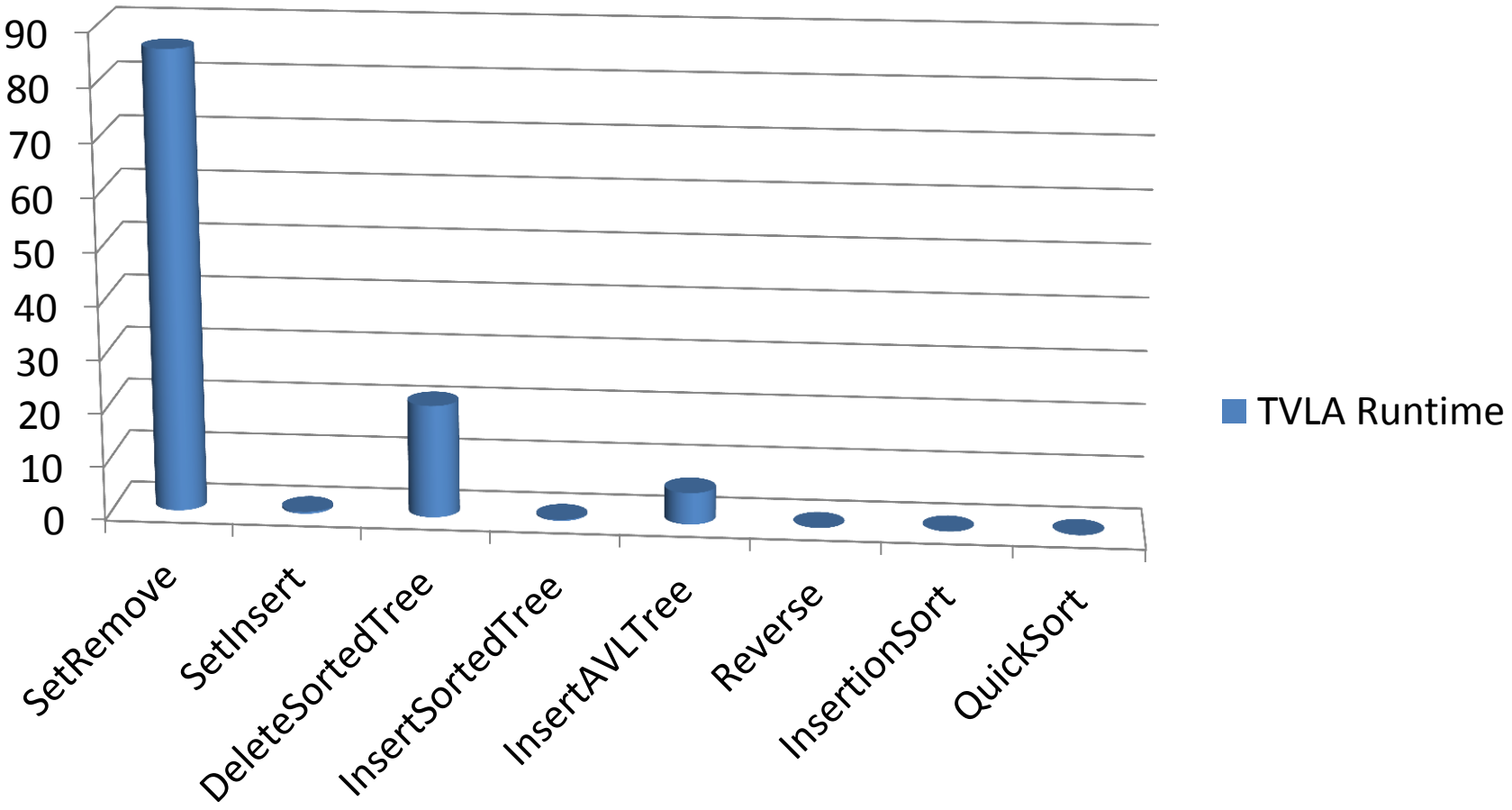


Command Translation

The TVLA process for translating a command:

- Focus – if the command relates a property we're not sure of (for example x.n=u0 is ½), instantiate it for all possible values
- Update – preform command on current state graph + update predicates
- Coerce – remove impossible structures
- Blur – perform summary operation (promises process termination)

Runtime



2.6GHz Pentium, 1GB Ram, Win XP

Time unit: minutes

Runtime

TLVA works well on small programs, but when trying to scale up the solution running time may reach double exponent!

Most of the time is wasted due to the fact even a simple command may affect all predicates along the way. That means that every function call\loop cannot be analyzed out of its context – function analysis cannot be reused.

Next up: two different methods to ease this runtime bottleneck

More uses & versions of TVLA

- TLVA is very versatile and may be used to analyze (or relay on) other properties beside structures:
  - Determining program correctness (sort example)
  - Adding type predicates
  - Adding allocation position predicates
  - Time stamping heap cells creation

Things we learned up to now…

Shape analysis is a form of static \ dynamic program analysis.

Summary is the process of:

Converging multiple heap locations with similar attributes (predicates) to a single representation

The core predicates are: pointed_by_x \ c[n] \ connected_through_n \ r[x,n] \ is[n]

TLVA's runtime bottleneck is:

A single update may require pass on the entire structure, no analysis reuse

Break

Part 3 – cutting down on runtime

- Cutpoint-free & separation logic methods:
  - Main concept
  - Algorithm implementation & examples
  - Runtime
- Comparing both methods

Cutpoint-free shape analysis

Noam Rinetzky, Mooly Sagiv & Eran Yahav

(based on TVLA)

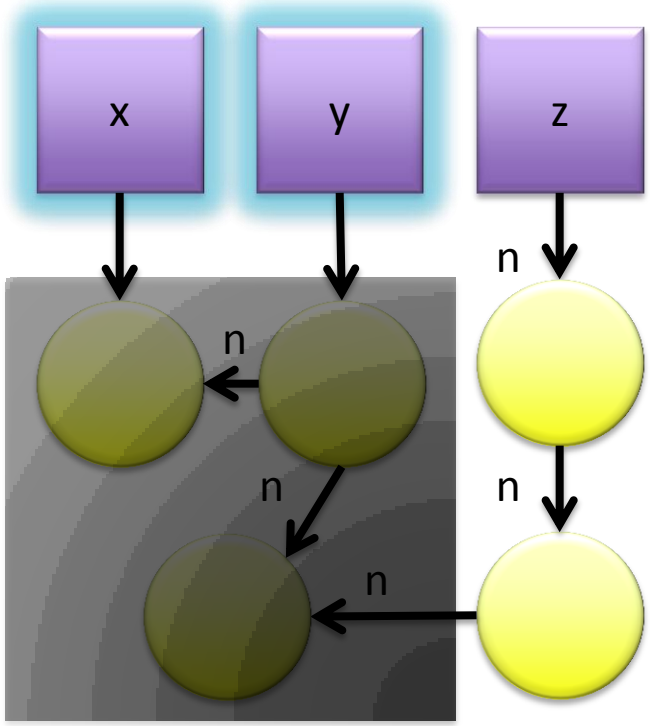
Cutpoint-free concept

- Function calls usually affects only memory pointed by the function arguments, and not other pointers\heap cells
- Such calls are called cutpoint-free
- A cutpoint-free call can be analyzed considering only the heap accessible through the function arguments – faster analysis
- Caller function analysis will treat calle analysis as sort of a black box

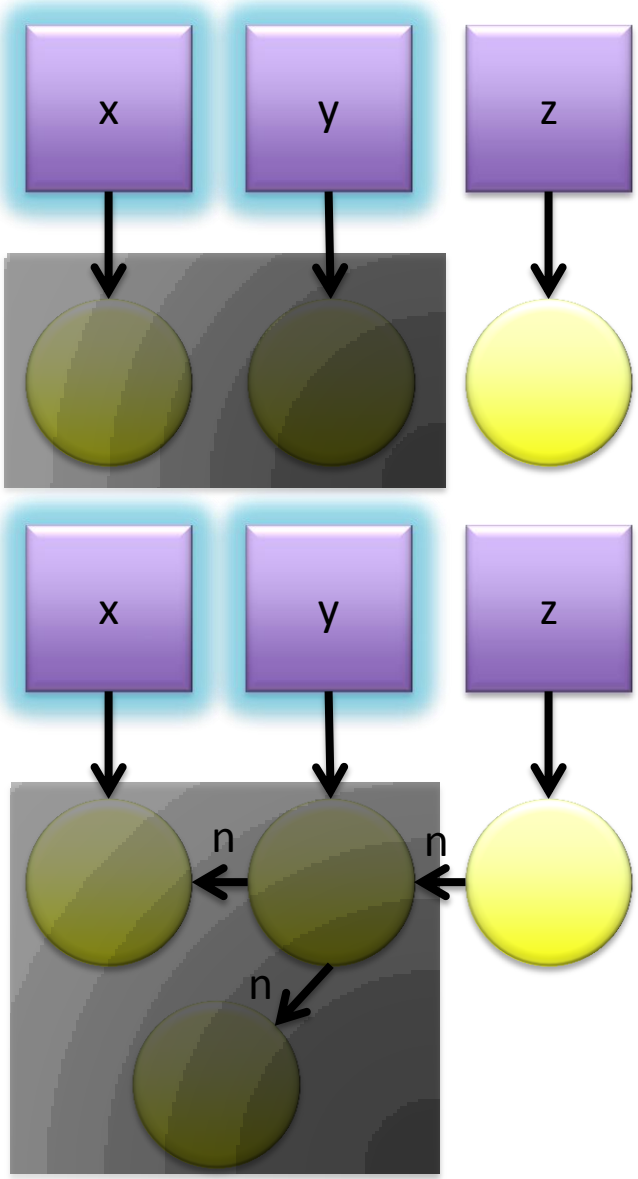
Cutpoints

Call func(x,y)

Is the call cutpoint free?



Definition of cutpoint?

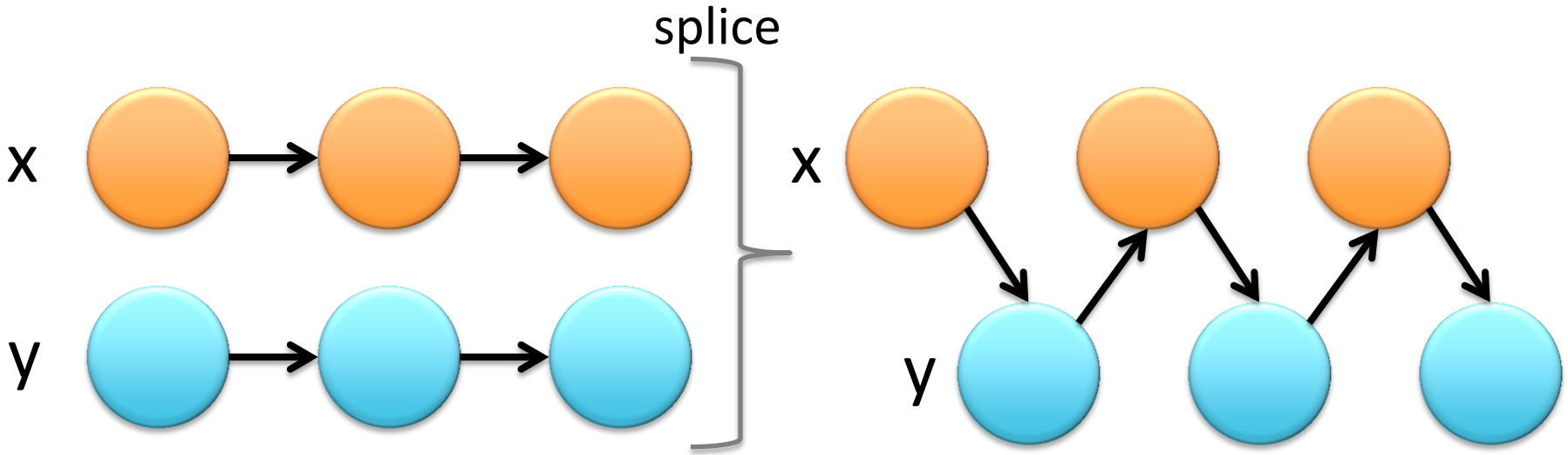


Cutpoint-free concept

- Cutpoint: a location reachable from a function argument, as well as reachable from a non-argument pointer while not passing through an argument.
- Exception: cutpoints cannot be pointed directly by a parameter
- Cutpoint-free algorithm can analyze only cutpoint -free programs (happens a lot, yet not always)
- If some call is not cutpoint free the algorithm can detect it using is-cutpoint[func] predicate

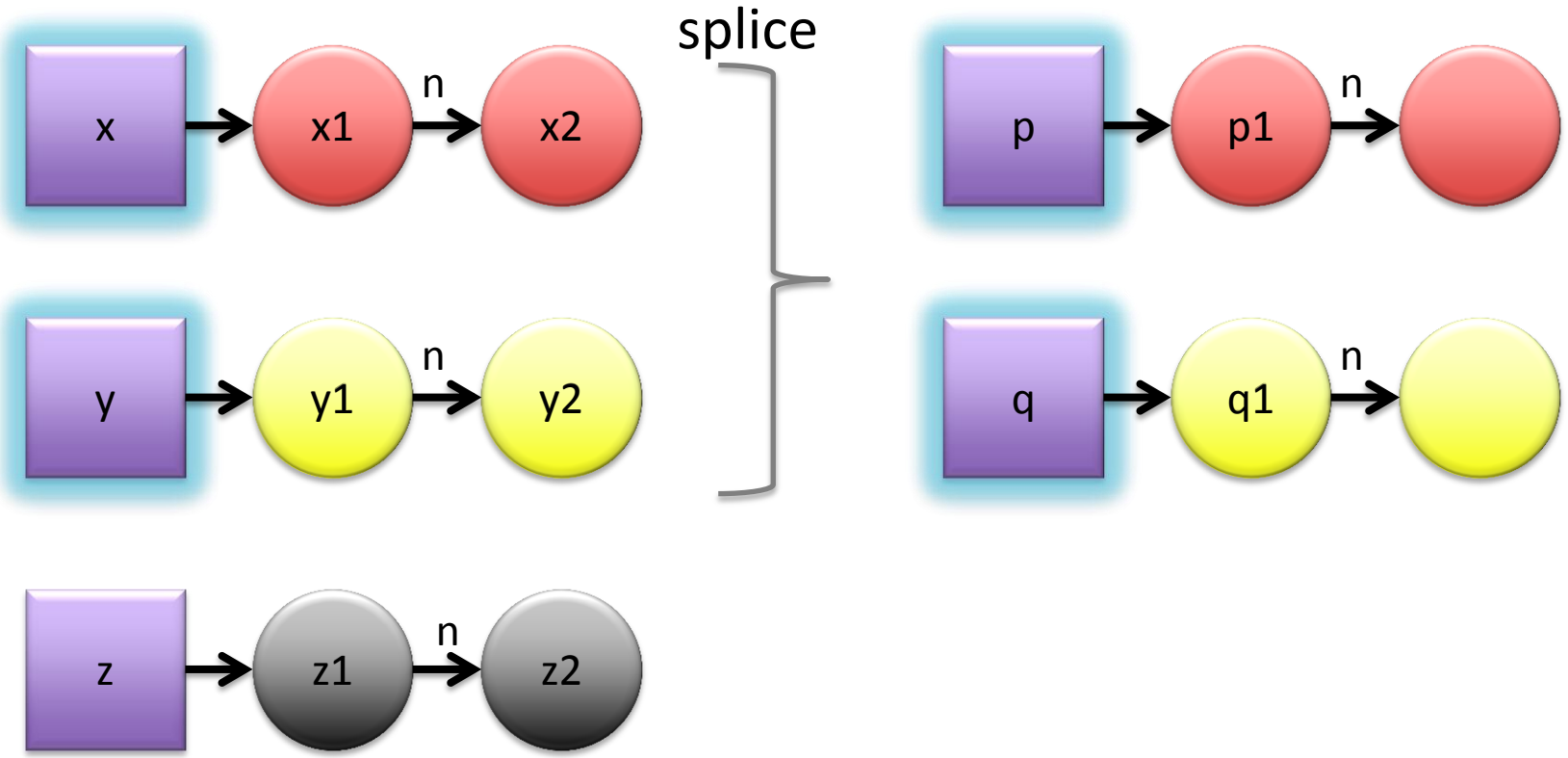
Cutpoint-free analysis example

List splice operation:



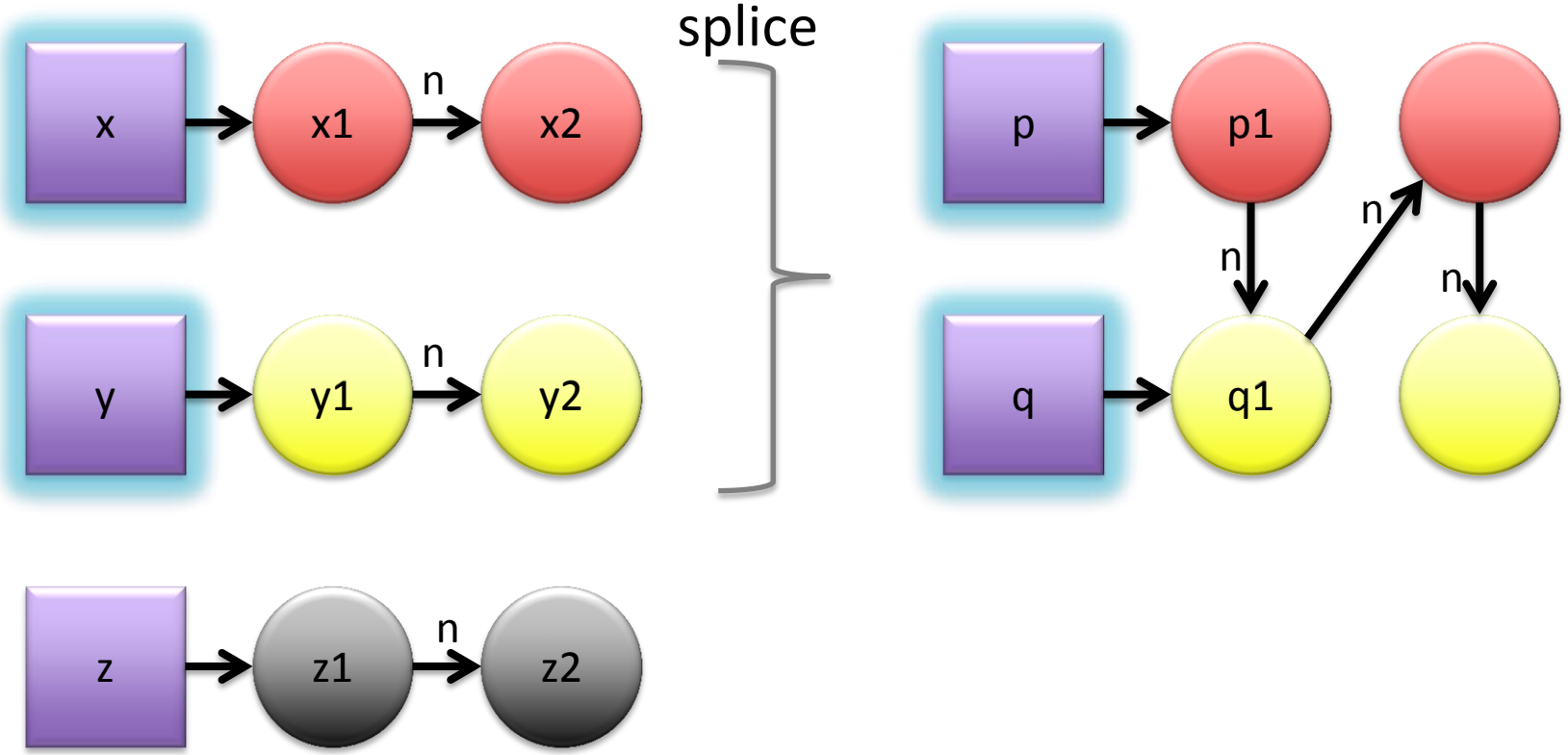
Cutpoint-free analysis example

Splice(x, y)



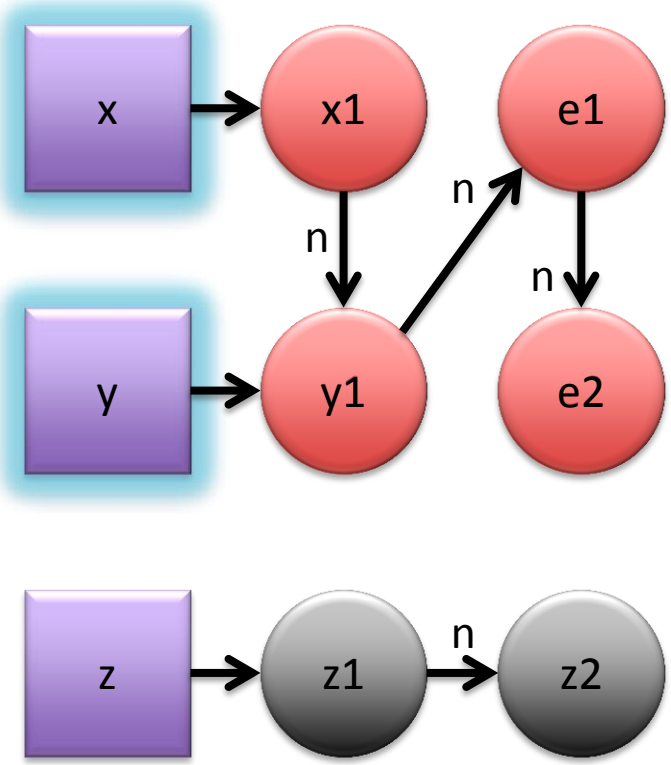
Cutpoint-free analysis example

Splice(x, y)



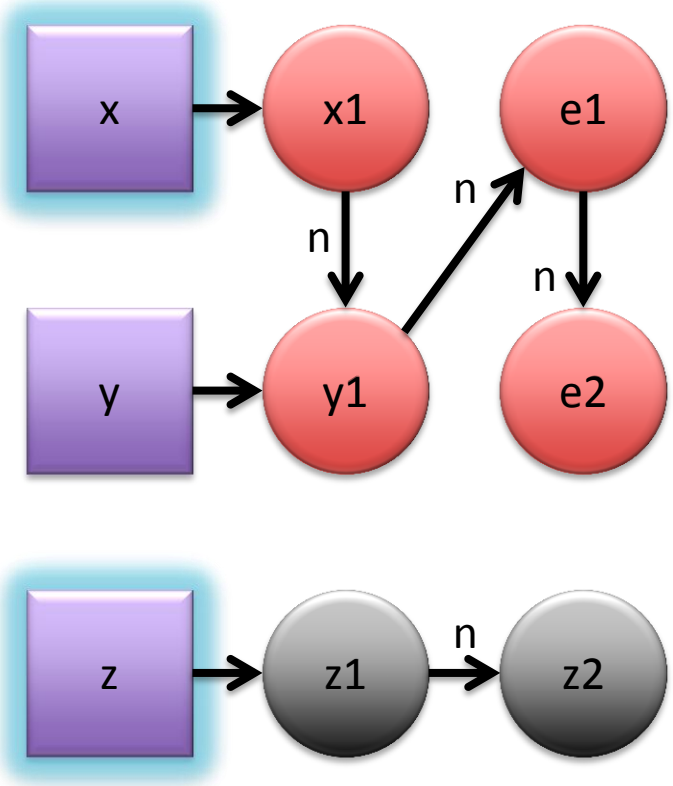
Cutpoint-free analysis example

Splice(x, y)



Cutpoint-free analysis example

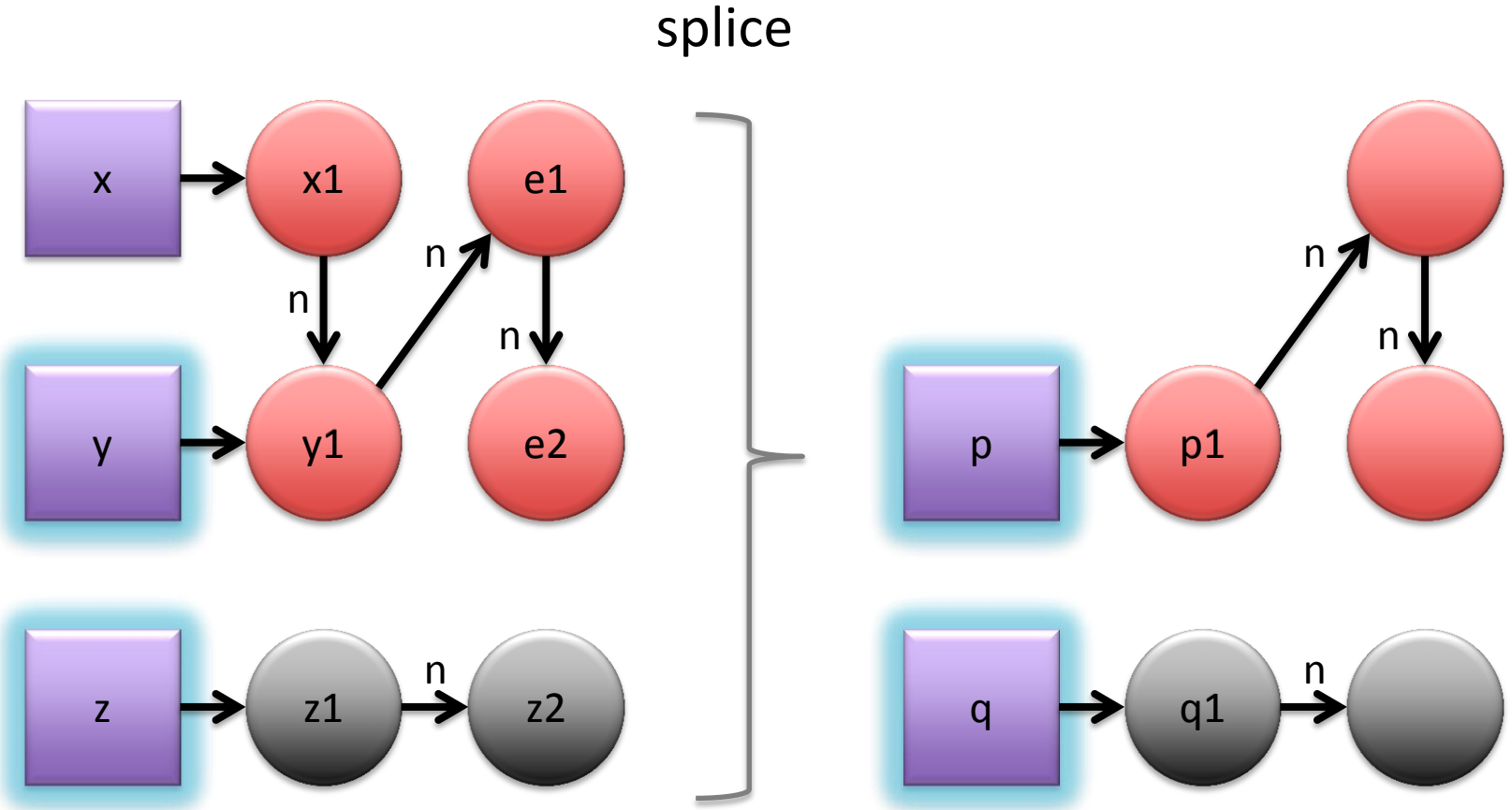
Splice(x, z)



Cutpoint!

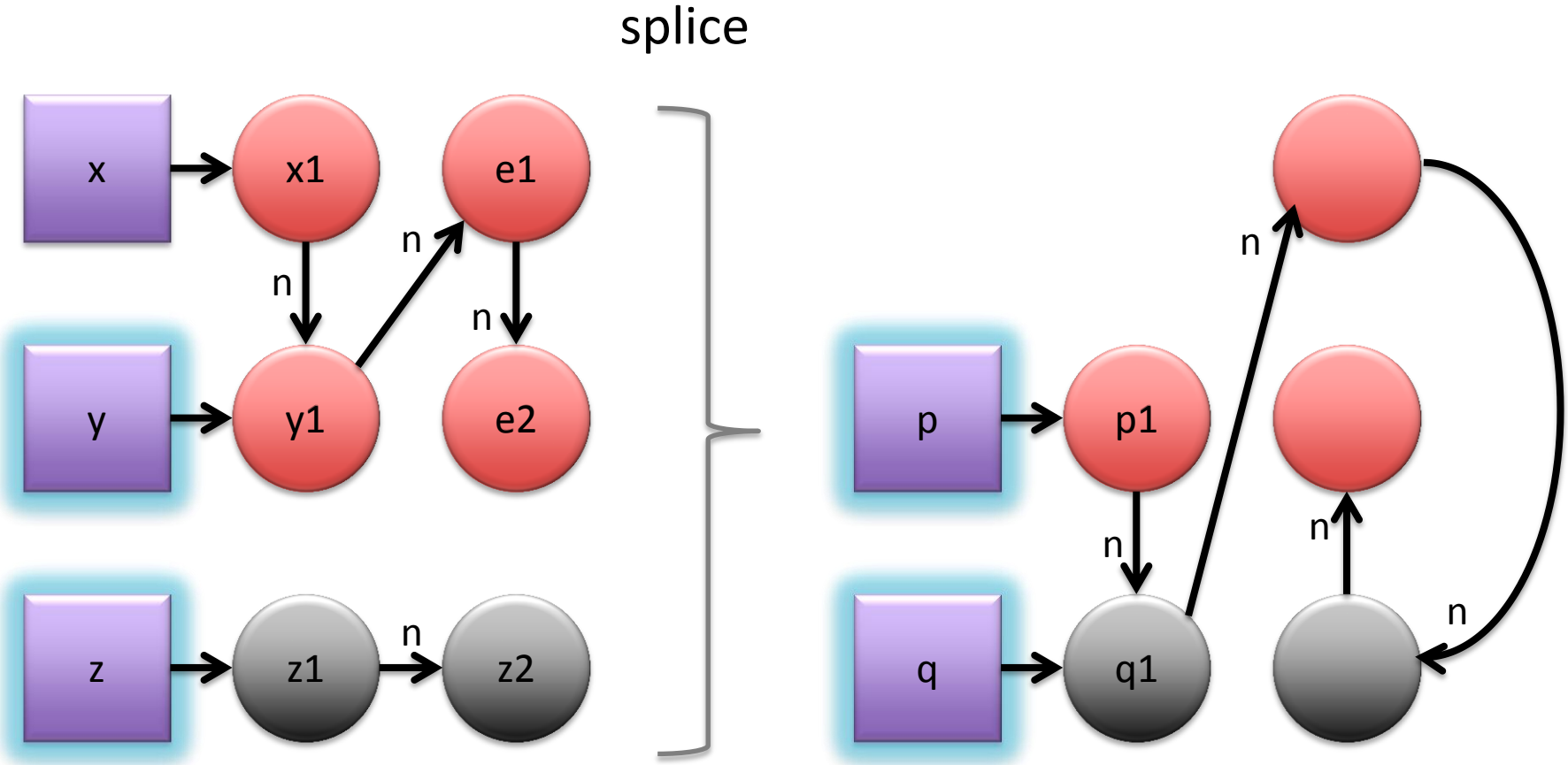
Cutpoint-free analysis example

Splice(y, z)



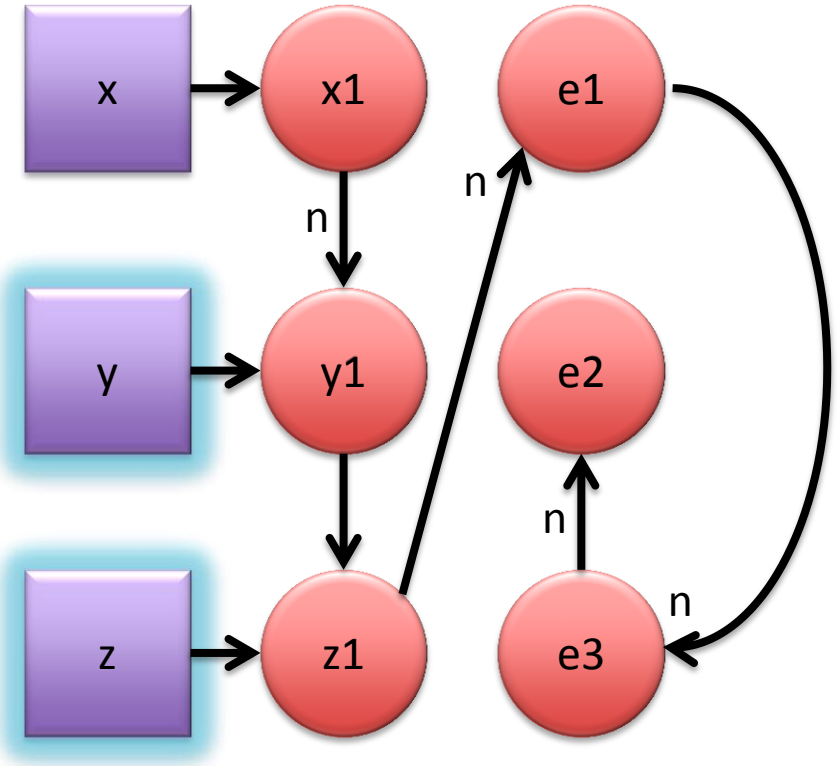
Cutpoint-free analysis example

Splice(y, z)



Cutpoint-free analysis example

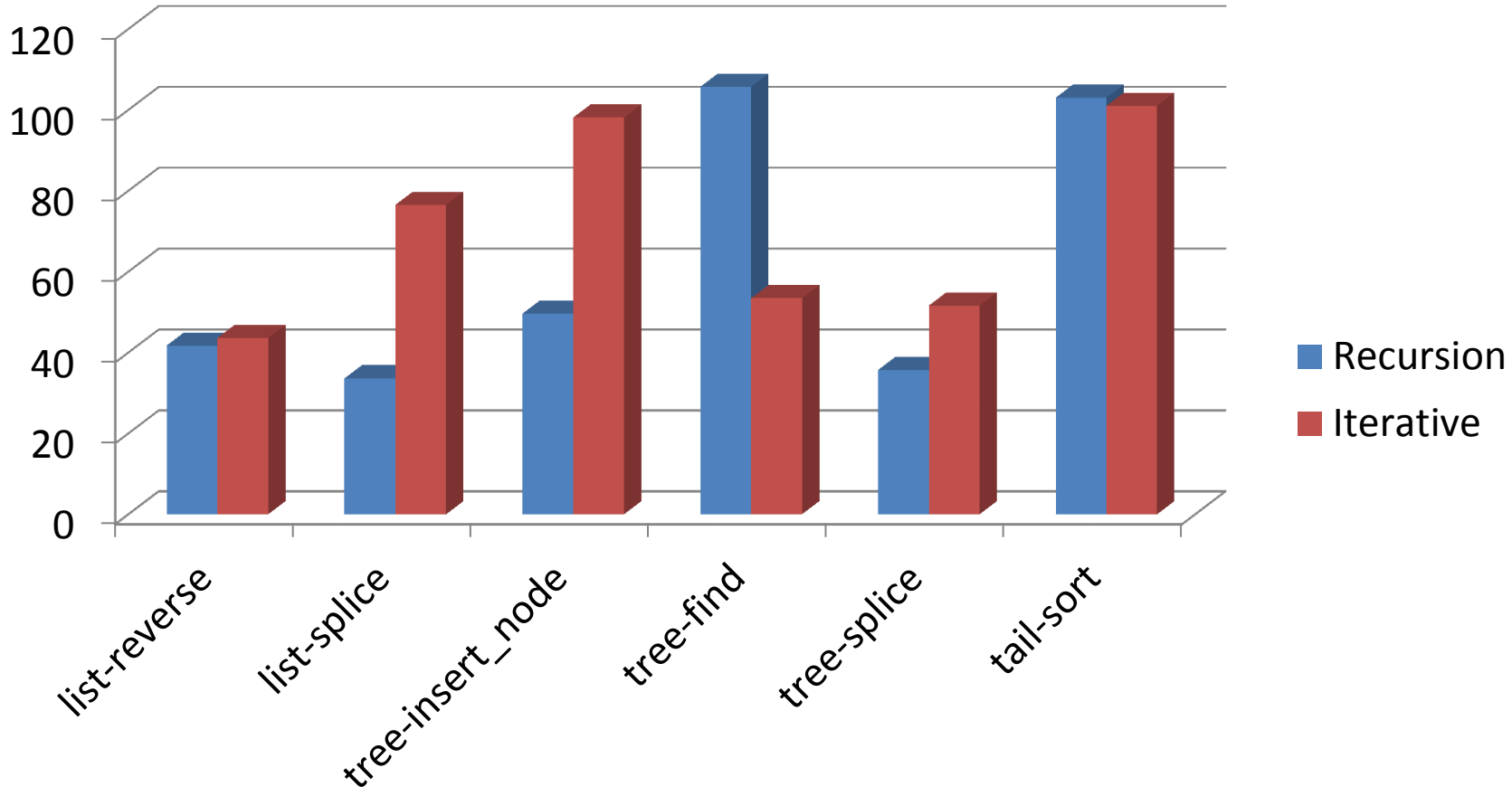
Splice(y, z)



Tabulation

- Beside that time saved by not updating properties of the entire heap, the algorithm employs another useful technique to save time
- Since functions are analyzed separately, we can remember results of analyzed calls with various inputs and re-use them (Tabulation)
- This even allows us to treat different call locations the same way – and therefore compute them only once.
- Separation of functions from calling context reduces runtime to single-exponent!

Cutpoint-free analysis runtime



1.5GHz Pentium, 1GB Ram, Win XP
Time unit: seconds

Separation logic based shape analysis

Method: Peter O'Hearn & John C. Reynolds

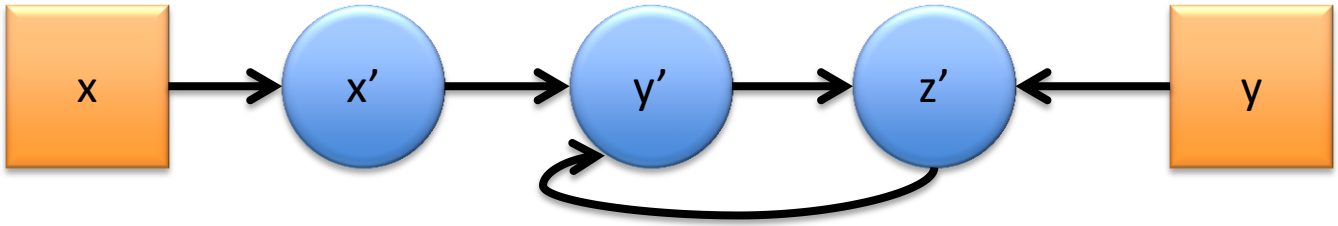
Tool: Dino Distefano, Peter W. O'Hearn & Hongseok Yang

Separation logic method

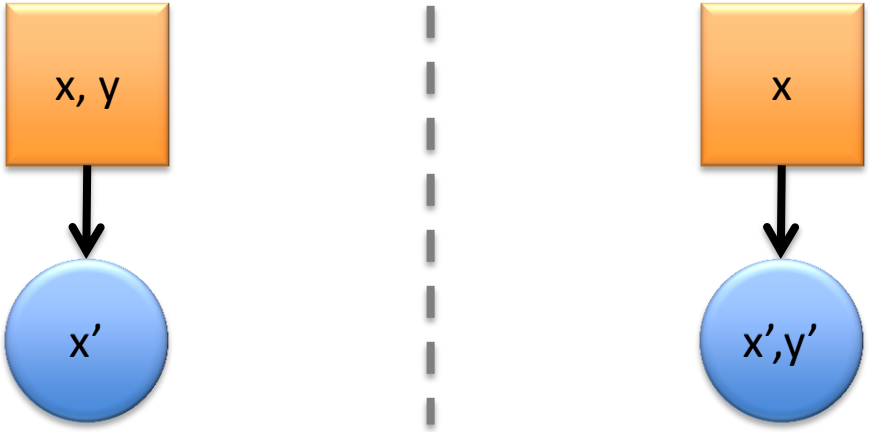
- Use specific logic with specific set of rules to represent memory pointing structure
- taking completely different approach from TVLA
- Commands affects the logical state with O'Heran logic style – {P} C {Q}
- Use reasoning to bound the locations command c might update to reduce runtime
- Presented version works only for lists (each cell has at most one pointer in it)

Separation logic – memory presentation

- Explicit pointers addresses – x, y, z…
- Implicit pointers addresses – x’, y’, z’…



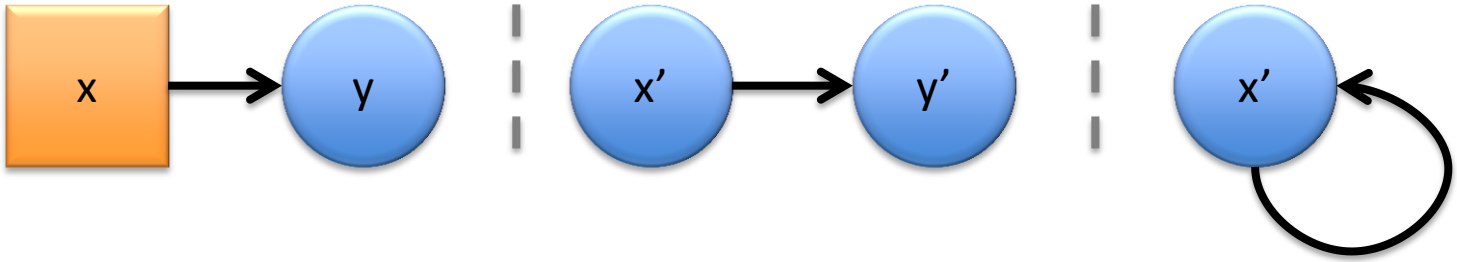
- Locations aliasing x=y, x’=y’:



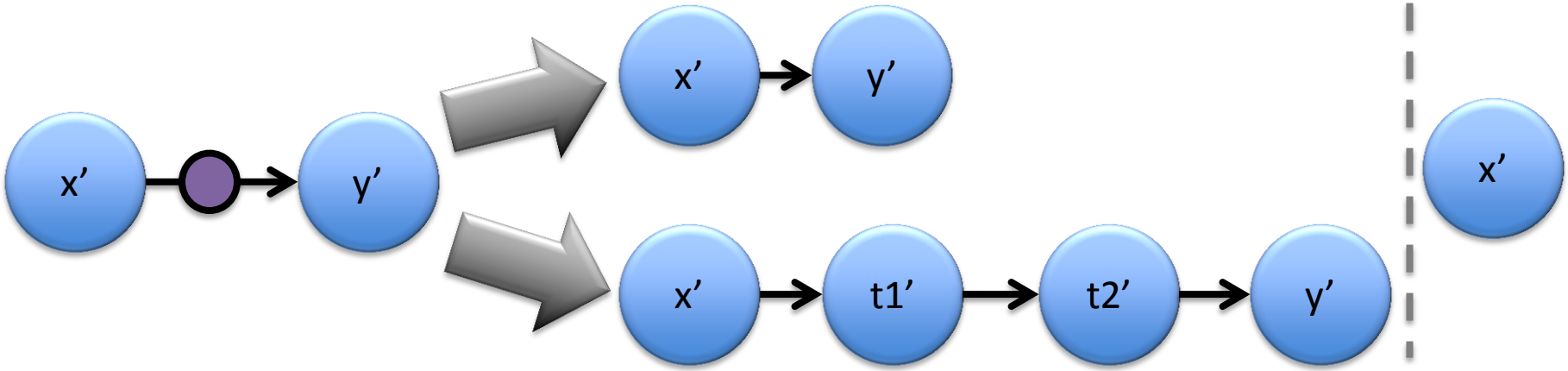
Separation logic – memory presentation

Two types of pointing:

- Straight forward pointing: $x \mapsto y$, $x' \mapsto y'$, $x' \mapsto x'$



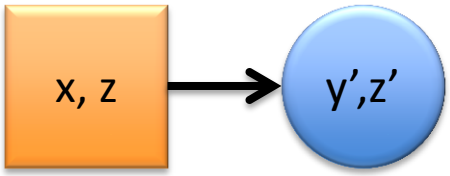
- Path indirect acyclic pointing: ls(x', y'), ls(x', x')



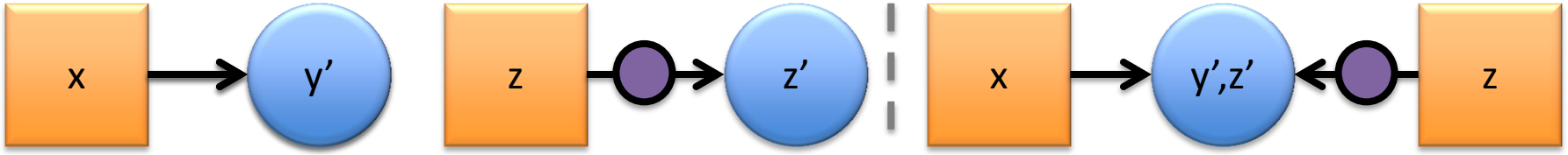
Separation logic – memory presentation

Operations between stacks\heaps:

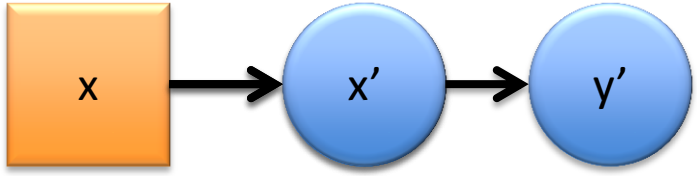
- $s_1,h_1 \wedge s_2,h_2$ – a structure that matches both:
  $\{x \mapsto y' \wedge ls(z, z')\}$



- $s_1,h_1 * s_2,h_2$ – guarantees separation:
  $\{x \mapsto y' * ls(z, z')\}$

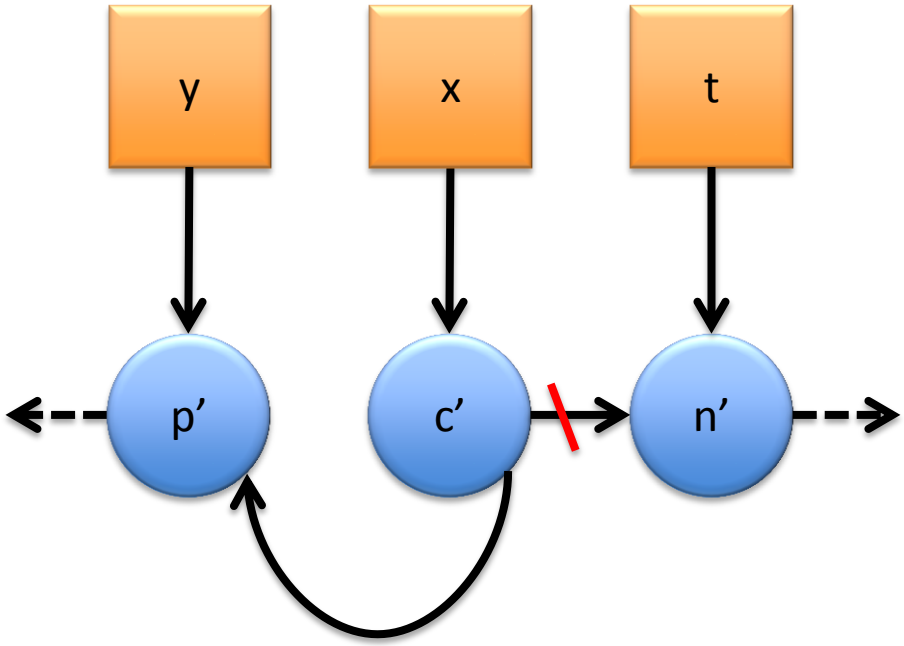


$\{x \mapsto x' * x' \mapsto y\}$



Separation logic example

```
void reverse_list(List * x)
{
    List *t = NULL, *y=NULL;
    while (x != NULL)
    {
        t = x->n;
        x->n = y;
        y=x;
        x=t;
    }
}
```



Separation logic example

```
void reverse_list(List * x)
{
    List *t = NULL, *y=NULL;
    while (x != NULL)
    {
        t = x->n;
        x->n = y;
        y=x;
        x=t;
    }
}
```

{x ≠ NULL∧ t=x ∧ ls(x) ∗ls(y)}

Unfold: {∃x'.t=x∧x↦x'∗ls(x')∗ls(y)}

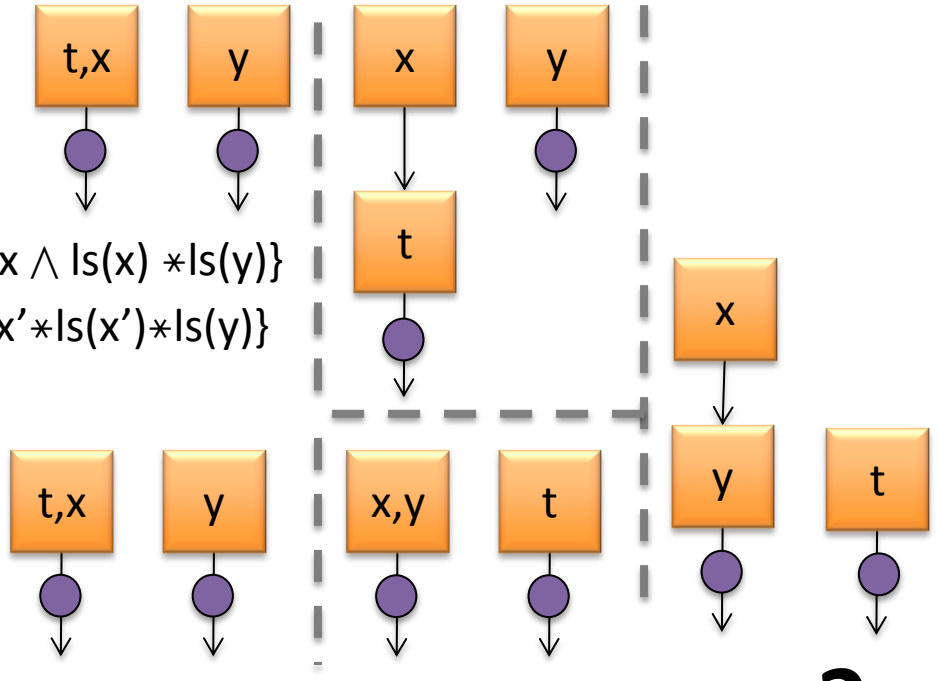
{x↦t∗ls(t) ∗ls(y) }

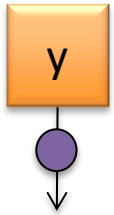
{x↦y∗ls(t) ∗ls(y) }

{x=y ∧ ls(t) ∗ls(y) }

{t=x ∧ ls(x) ∗ls(y)}

{t=x ∧ x=NULL ∗ls(y)}





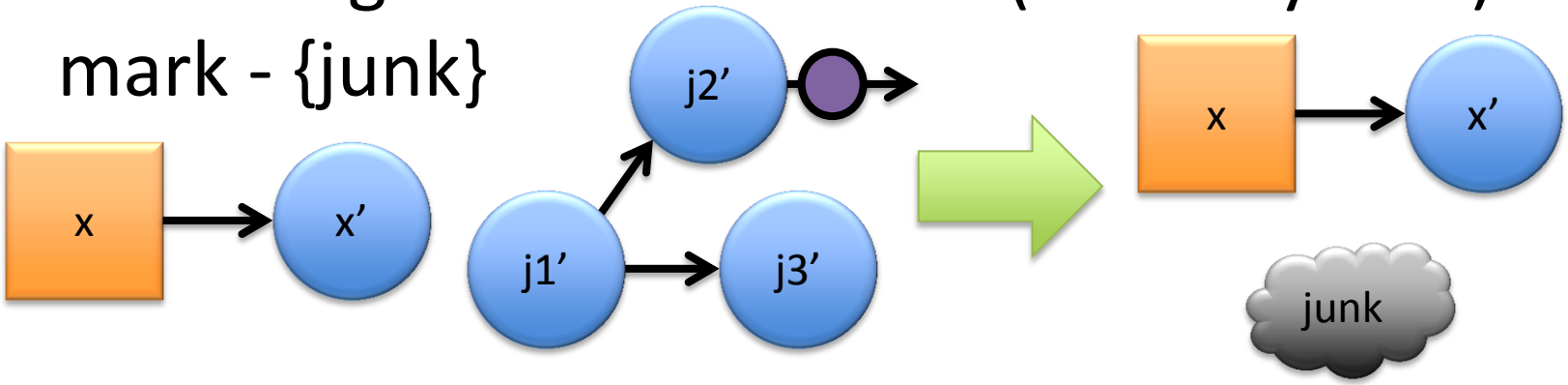
What about abstraction?

Is it needed?

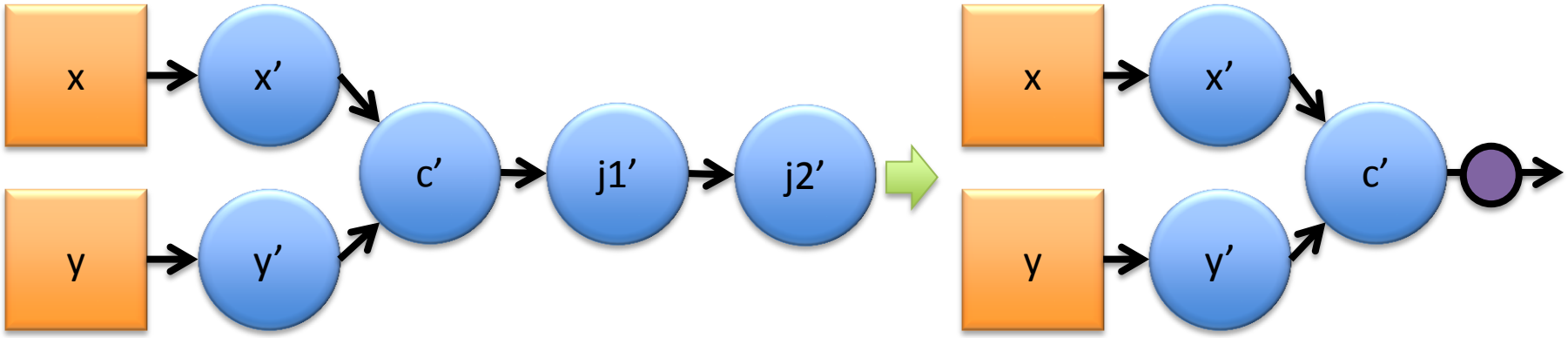
Abstraction of separation logic

We allow two types of abstraction:

- Collecting unreachable cells (memory leak): mark - {junk}

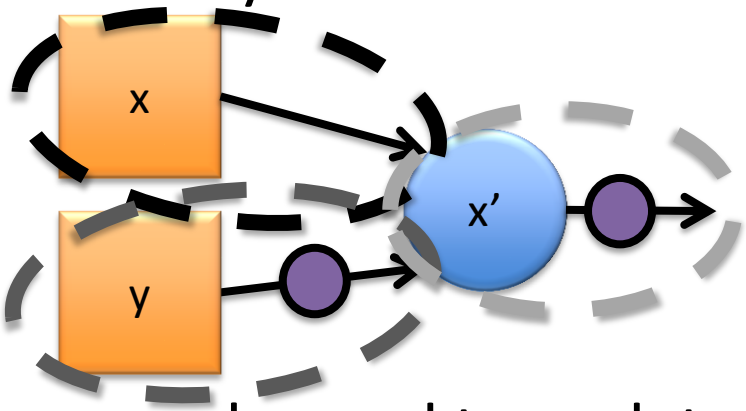


- Trimming sequences of primed locations:



Locality principle

- What do we gain from analyzing the structure using separation logic?
- "$*$" separates different memory slices
  $\{x \mapsto x' * ls(y, x') * ls(x')\}$



- When an update occurs we only need to update slices directly affected
- saves a lot of time when the slices are relatively small

TVLA VS Separation

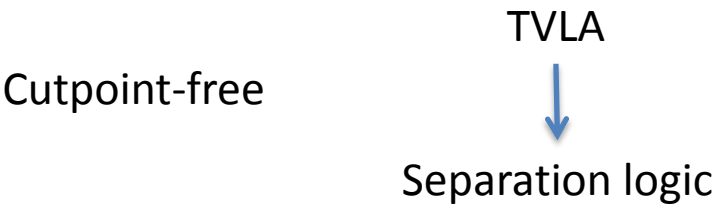
Category	TVLA\Cutpoint-free	Separation logic
Model	Abstraction by grouping predicates (graph oriented)	Logical proof
Predicates based on	Mainly reachability properties	Inductive predicate (ls for example)
Coverage	Soundness	Soundness
Operation	Automatic only	Automatic or manually
Achilles' heal	Small updates can effect everything and impact runtime	Lower expressability
Locality principle	Function calls separation & tabulation	Locality & Tabulation
Reception	One of the two leading methods for shape analysis	The other of the two leading methods (Linux kernel analyzed)

Part 3 – Conclusions & Personal View

- Summary
- My thoughts
- My idea
- Questions
- Discussion

Summary

- Shape analysis allows us to analyze the heap structure
- It can answer advanced questions (is this a doubly liked list? Is this a part of a cycle?)
- We've seen 3 methods of shape analysis:



- Last two attempt to solve runtime bottleneck

Summary

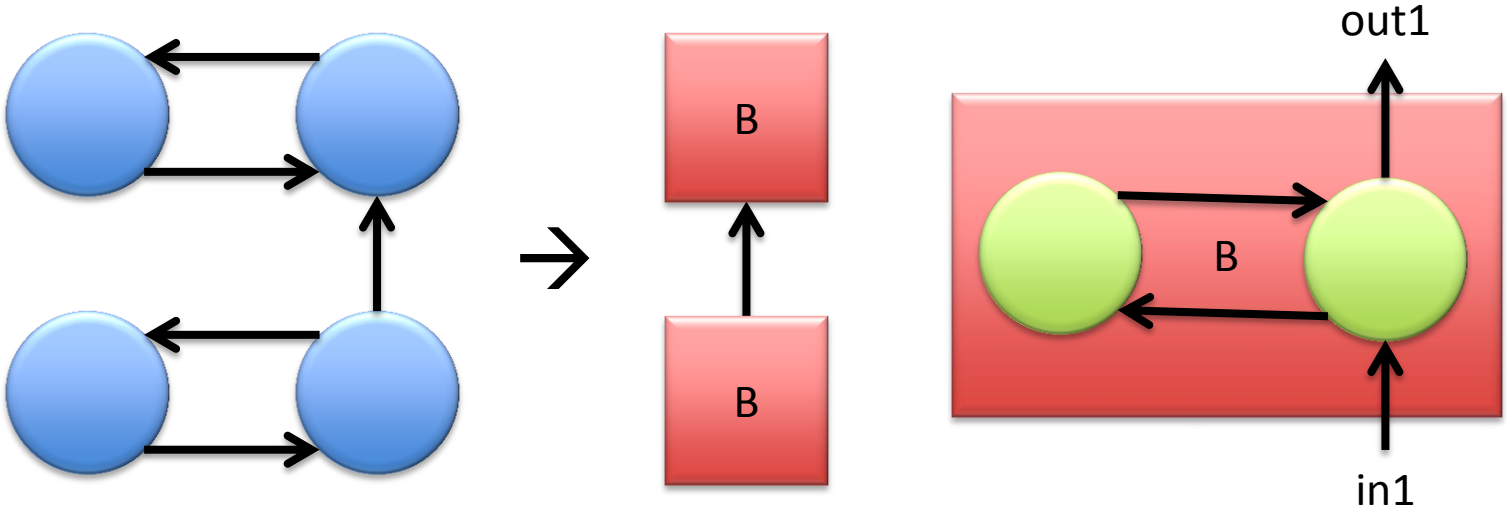
- TVLA - uses three valued logic, easily allows definition of user properties (predicates)
- Cutpoints algorithm – attempts to decrease runtime by separating function points from their calling context (based on TVLA)
- Separation logic – uses tailored logic reasoning to bound the area requiring updates

My thoughts & Conclusions

- Ground breaking idea & techniques
- Presented algorithms are complex, but are also straight forward and very versatile
- Competitive field
- The distance to practical use is still far:
  - Long runtime (hard time scaling up)
  - Not a complete solution (structures only, libs support)
- Maybe general idea may solve other problems?
  - Image analysis
  - Pattern recognition

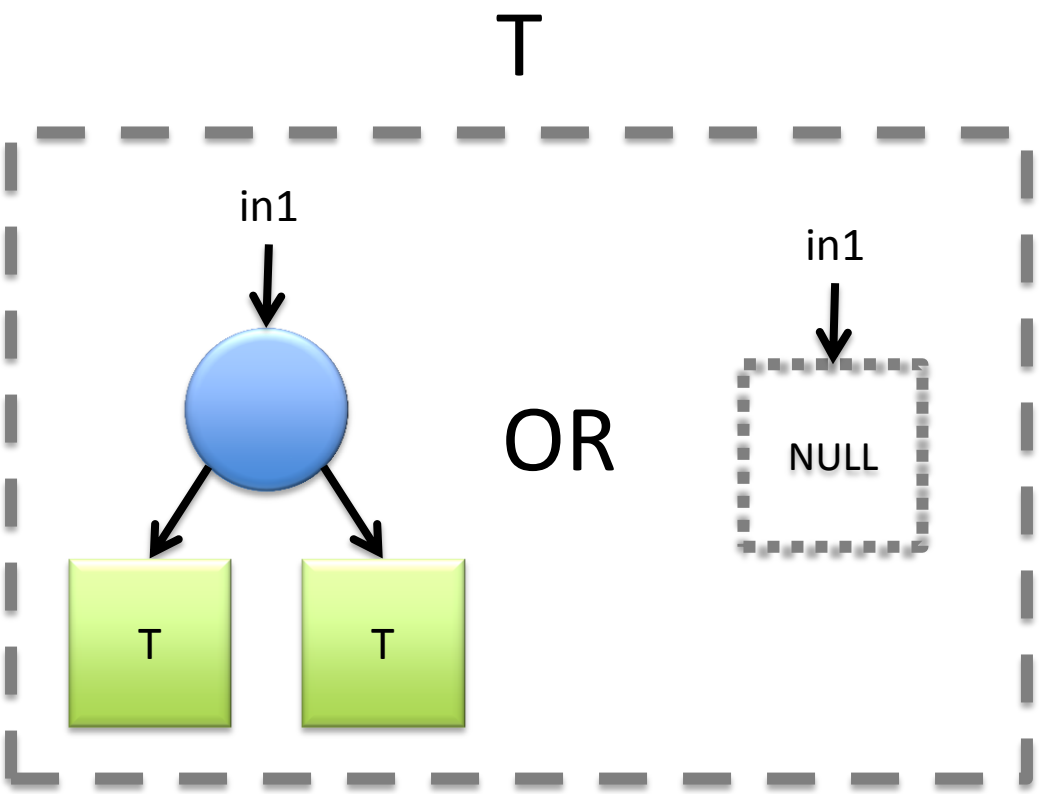
My* idea – Template analysis

- Compress structure representation by identifying reoccurring structures
- For each new (small) heap state build a template, reuse templates to define entire heap



My idea – template analysis

- Representation can be recursive (abstraction):



My idea – template analysis

Open Questions:

- How to make pattern search feasible without loss of quality?
  (subgraph isomorphism is NP-complete)
- How to select between few possible matches?
- How to generate recursive structures?

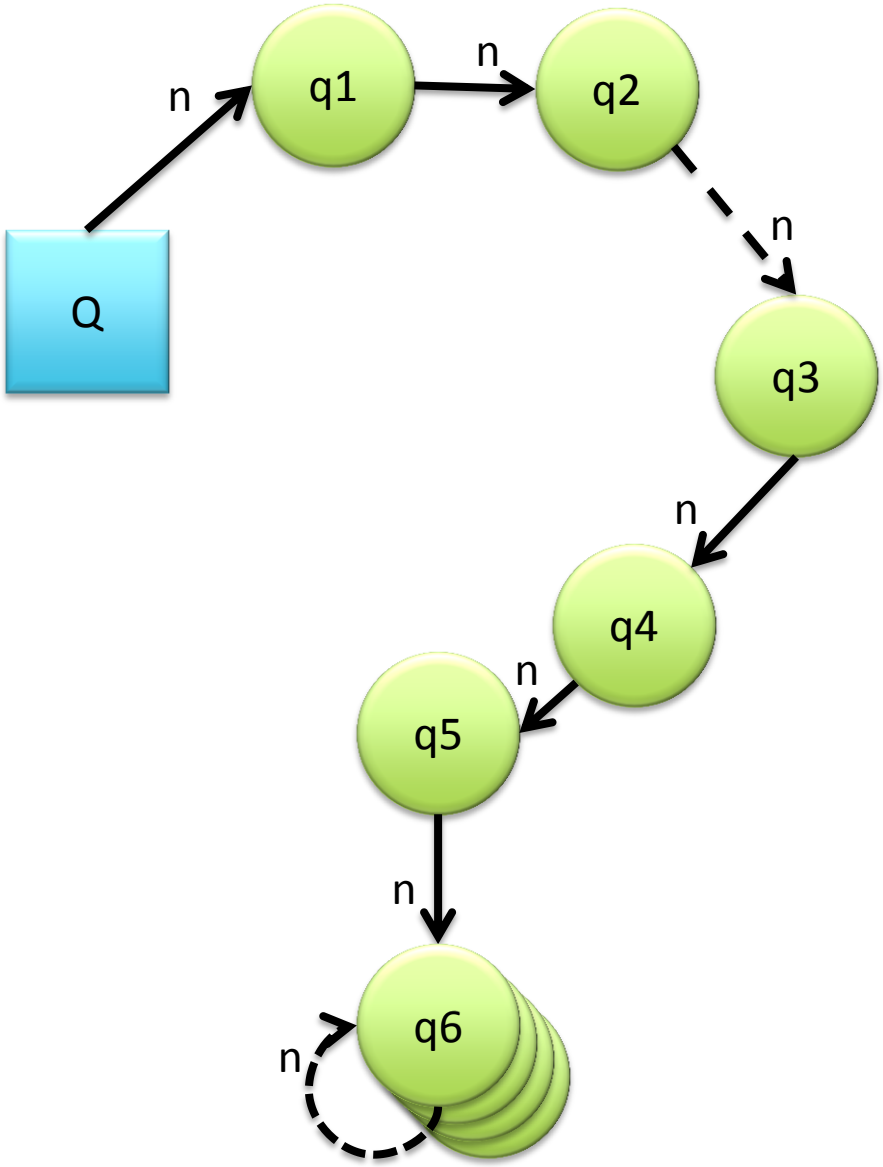
Shape analysis vs Template analysis

Template analysis advantages:

- Properties calculated only once per shape
- Utilizes recursion definition of structures
- Allows short representation of common objects (similar to dictionary contraction)

Template analysis disadvantages:

- Many open questions – not even sure possible
- Runtime (probably) longer


Q
n
q1
n
q2
n
q3
n
q4
n
q5
n
q6
n

Discussion

- Which method is better?
- Which properties\predicates would you define?
- Would you use shape analysis?

- Any comments about the lecture itself? (don’t be afraid to be rough)

References

- Shape analysis terms:
  Shape Analysis
  by Reinhard Wilhelm, Mooly Sagiv & Thomas Reps
- TVLA algorithm:
  TVLA: a system for implementing static analyses
  by Tal Lev-Ami & Mooly Sagiv
- Cutpoint-free algorithm:
  Interprocedural shape analysis for cutpoint-free programs
  by Noam Rinetzky, Mooly Sagiv and Eran Yahav
- Separation logic algorithm:
  A local shape analysis based on separation logic
  by Dino Distefano, Peter W. O'Hearn & Hongseok Yang
- TVLA runtime examples:
  Revamping TVLA: making parametric shape analysis competative
  by Igor Bogudlov, Tal Lev-Ami, Thomas Reps & Mooly Sagiv