Shape Analysis

Goal: for each program point, for each variable, a finite description of the heap-allocated data structures resulting from any execution

Problem: mapping of a heap of potentially unbounded size to a graph of bounded size

Different approaches:

- length of paths in the heap can be bound (Jones-Muchnick [JM81])
- heuristics to merge the heap cells created at the same program point (Chase-Wegman-Zadeck [CWZ90])
- heap cells that cannot be kept apart by the set of pointer variables poiting to them are merged (Sagiv-Reps-Wilhelm [SRW96])

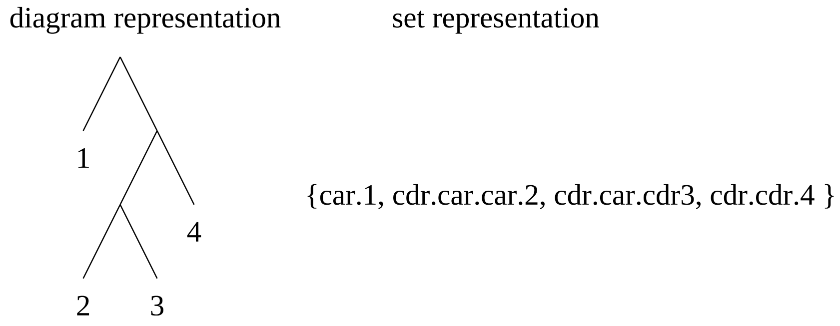
We assume an imperative language, with pointers and integers, and lisp-like heap-allocated data structures, but without functions

Outline

1. Shape-analysis without destructive update [JM81]
2. Shape-analysis with destructive update
   (a) Shape-graphs
   (b) Shape-graph in [JM81]
   (c) Shape-analysis in [JM81]
3. Heuristic Approach [CWZ90]
   (a) Managing multiple instances
   (b) Strong update
   (c) Shape detection
4. Different naming scheme [SRW96]
   (a) Strong nullification
   (b) Materialization and sharing
   (c) Achievements
5. Conclusion

Without Destructive Update [JM81] 1/4

The value of a variable can be seen as a tree:



During the execution, each program point is associated a concrete store:

$$\mathrm{Var} \to \mathcal{P}(\{\mathtt{car}, \mathtt{cdr}\}^* \times \{\mathrm{int}, \mathtt{nil}\})$$

Without Destructive Update [JM81] 2/4

Analysis by abstract interpretation:

1. Construct a system of data-flow equations. For instance, the statement s

   $$\mathtt{x} := \mathtt{cons}(\mathtt{y}_1, \mathtt{y}_2)$$

   is turned into an equation

   $$F(s, \mathtt{x}) = \bigcup_{p \in \mathrm{pred}(s)} \mathrm{cons}(F(p, \mathtt{y}_1), F(p, \mathtt{y}_2))$$

   where $\mathrm{cons}(T_1, T_2) = \{\mathtt{car}.t | t \in T_1\} \cup \{\mathtt{cdr}.t | t \in T_2\}$

2. Find the least-fixed point; each $F(s, \mathtt{x})$ is associated a set with all the possible trees $\mathtt{x}$ may point to at s in any possible execution of the program

   Yet, because of looping, the solutions may be infinite!

Without Destructive Update [JM81] 3/4

Use (tree) grammars whose generated language is a solution to the system of data-flow equations:

1. Each data-flow equation is seen as a production. For instance, the statement s

   $$\mathtt{x} := \mathtt{cons}(\mathtt{y}_1, \mathtt{y}_2)$$

   with one predecessor p is seen as the production

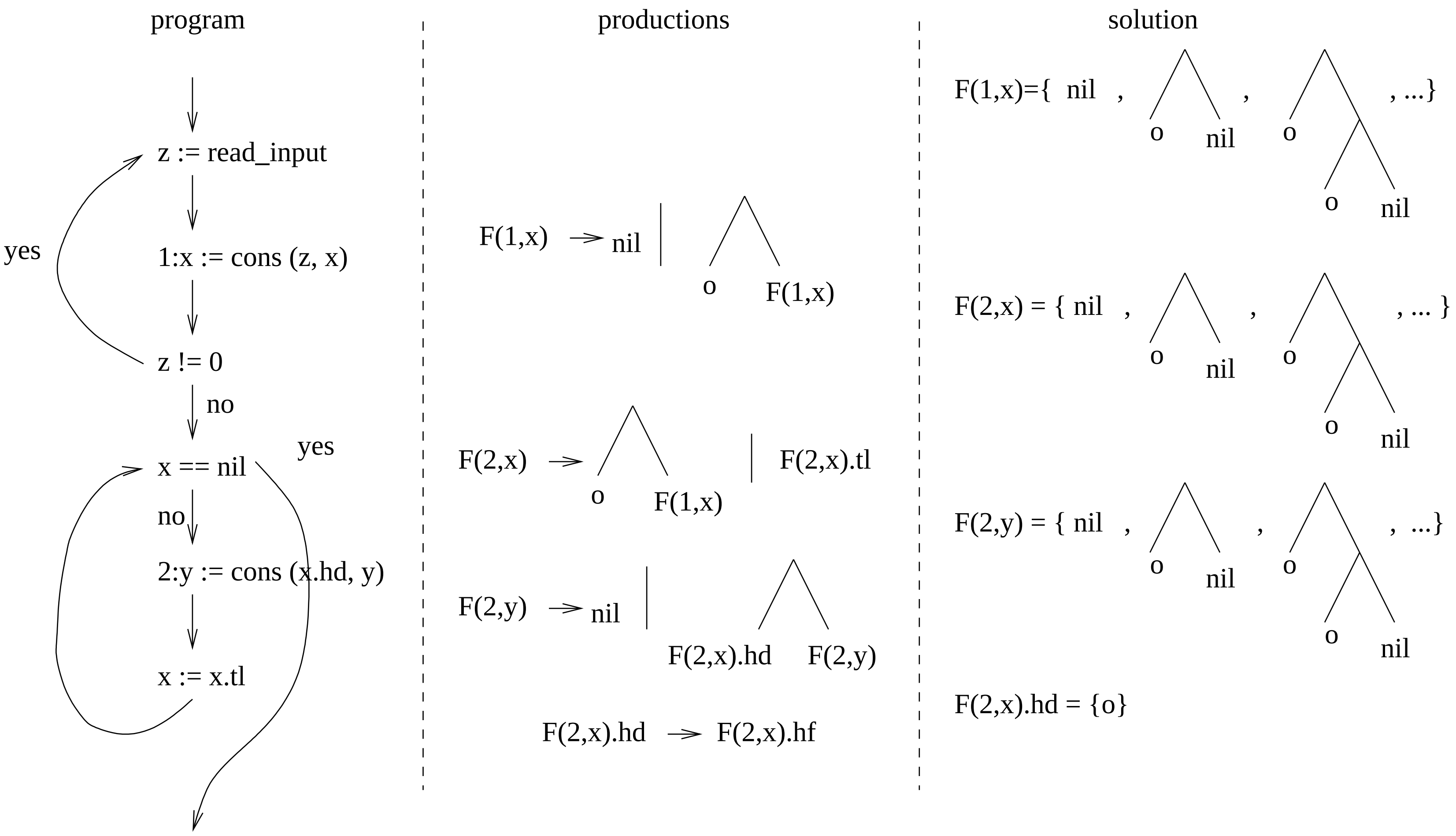
   $$F(s, \mathtt{x}) \longrightarrow \begin{array}{c} \diagup \!\!\!\! \diagdown \\ F(p, \mathtt{y}_1) \quad F(p, \mathtt{y}_2) \end{array}$$

   The nonterminal $F(s, \mathtt{x})$ is seen as the

   set $\left\{ \begin{array}{c} \diagup \!\!\!\! \diagdown \\ t_1 \quad t_2 \end{array} \middle| \, t_1 \in F(p, \mathtt{y}_1), t_2 \in F(p, \mathtt{y}_2) \right\}$

2. Simplify and solve the system.

Without Destructive Update [JM81] 4/4



Need for Shape-graphs

Destructive updates allow for cycles and complicate sharing

$\downarrow$

The value of a variable cannot be seen as a tree anymore

$\downarrow$

The whole memory more closely resembles a directed graph

$\downarrow$

The analysis associates to program points (sets of) *shape-graph*(s)

Shape-graph Interpretation

Let `x` be a variable, n be a shape-node allocated at statement s_n, `f` be a field in the shape-node n, m be a shape-node allocated at statement s_m.

Let us consider a statement s and its associated shape-graph:

- If there is an edge from `x` to n, then, after some execution of s, the variable `x` may point to some data structure allocated as s_n.
- If there is an edge from n to m labeled `f`, then after some execution of s, the field `f` of some data structure allocated at s_n may contain a pointer to some structure allocated at s_m.

Shape-graph Representation in [JM81] 1/2

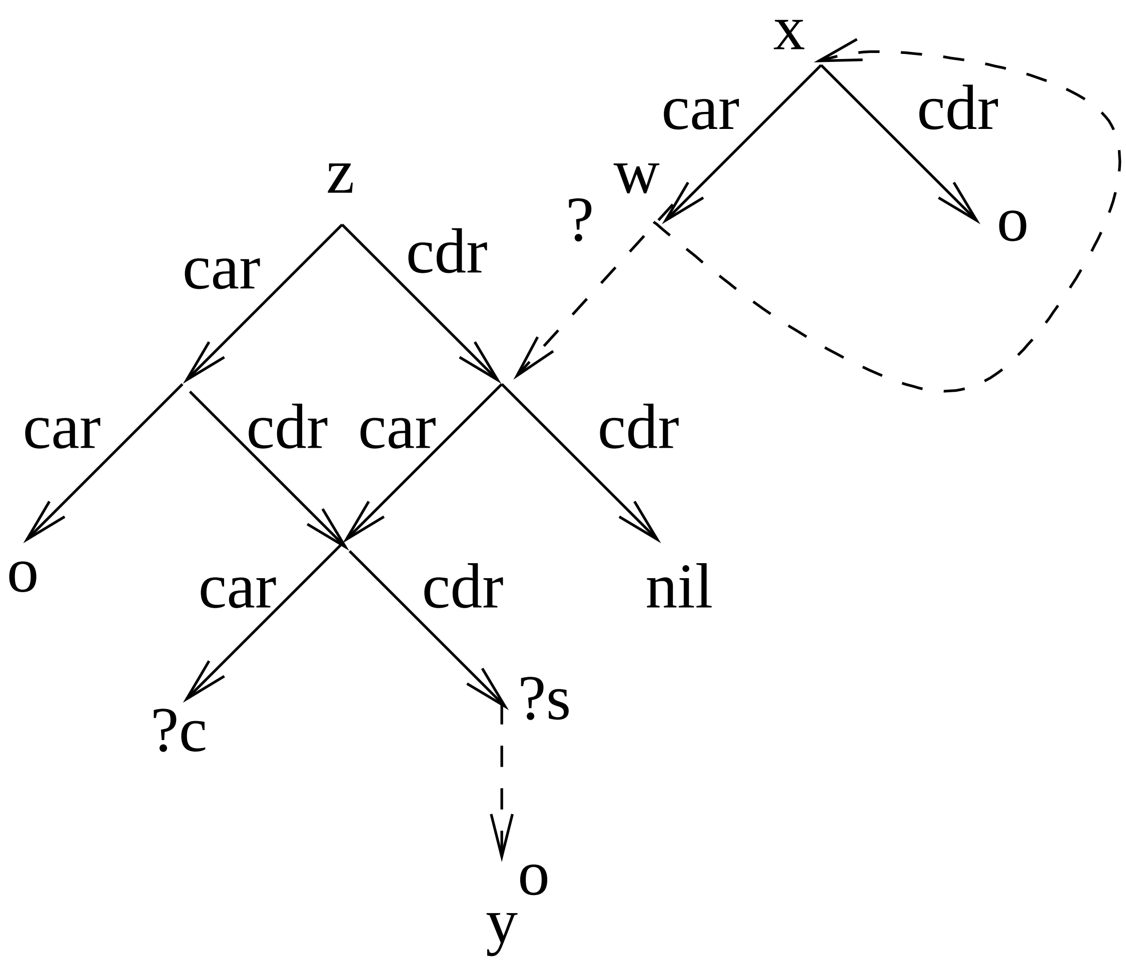
Annotated directed graphs:

- each variable labels one node
- each node is accessible from a variable-labeled node
- each (known) node has outgoing `car` and `cdr` edges to other nodes
- *summary nodes*[a] are marked as *unknown* (`?`)
  - unknown nodes may contain *shared* nodes (`?s`)
  - unknown nodes may contain *cycles* (`?c`)
- each unknown node may have any number of outgoing edges
- each leaf is either an integer, `nil`, `?`, `?c`, or `?s`

[a] 'contraction' of a subgraph into a single (meta-)node

Shape-graph Representation in [JM81] 2/2

Example[a]:



[a] 3-limited

Shape-analysis in [JM81]

The shape-analysis

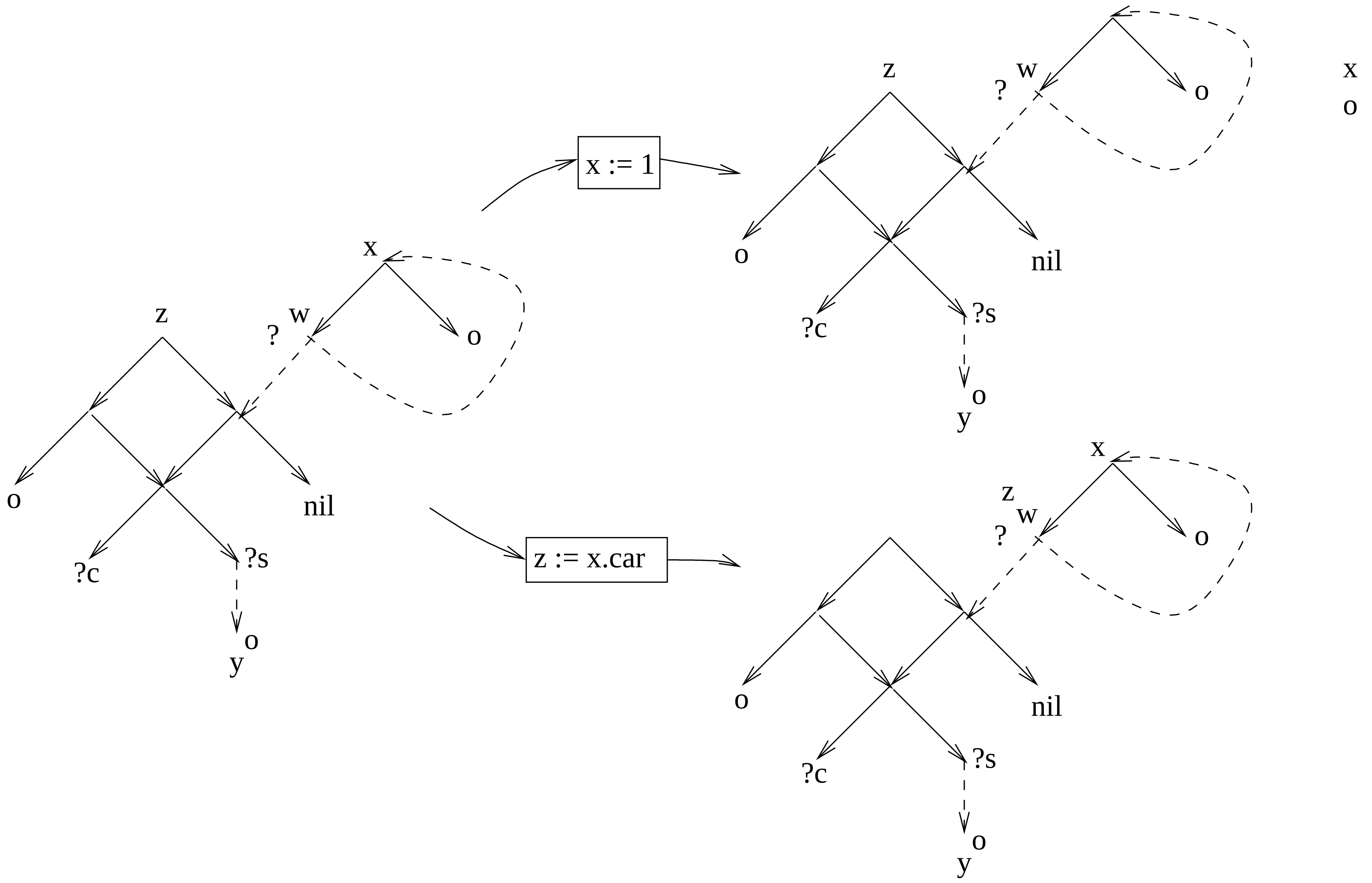
- is merely a 'sets of shape-graphs' transformer
- ultimately associates with each statement s a set of graphs $F(s)$

For instance, an assignment results in the following graph transformations:

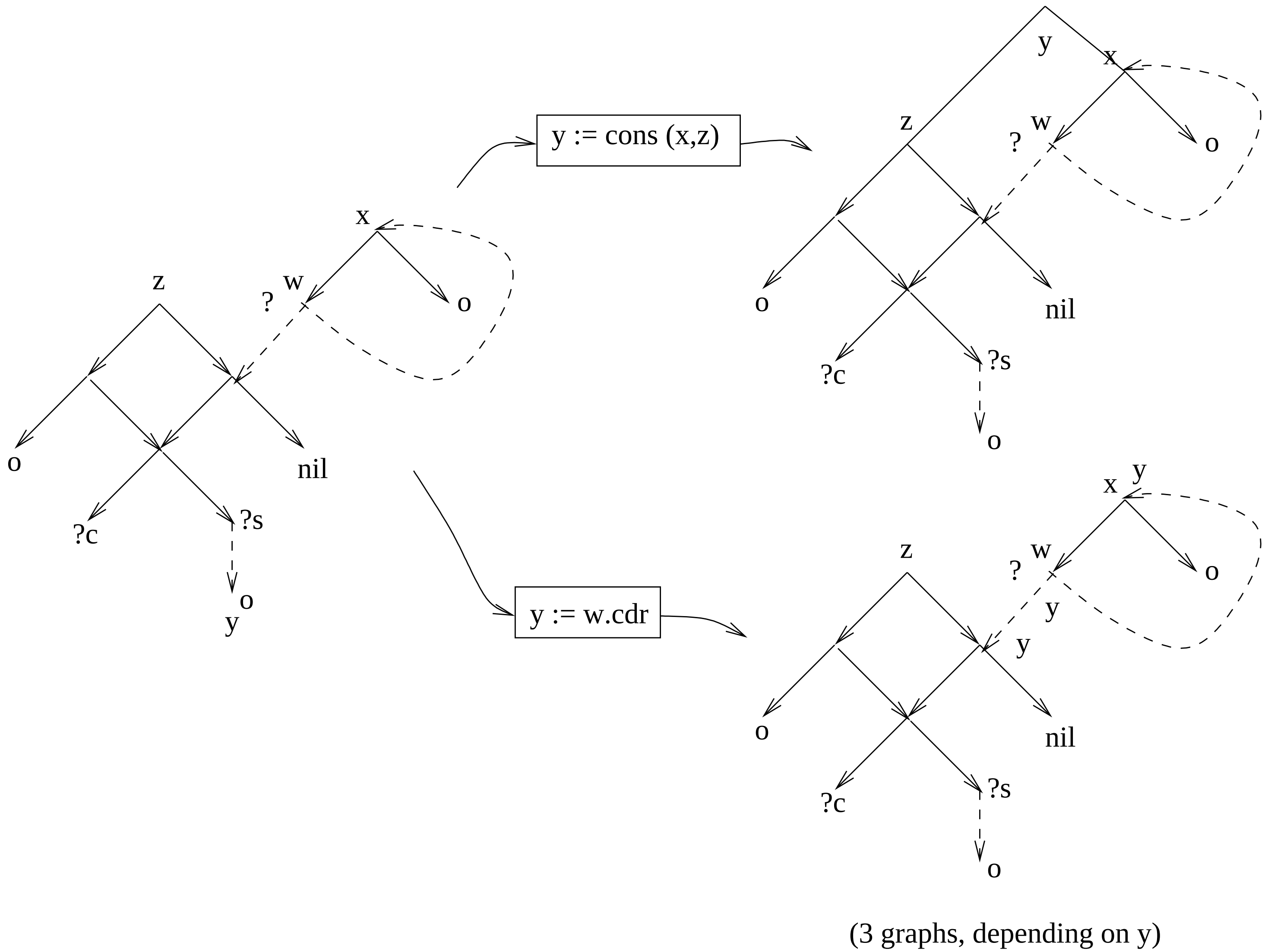
$$F(s) = \bigcup_{p \in \mathrm{pred}(s)} \mathrm{next}(F(I_p))$$

where next is the function illustrated hereafter

Shape-analysis in [JM81]



Shape-analysis in [JM81]



Shape-analysis in [JM81]

Shape-graphs may still grow without bounds in loops!

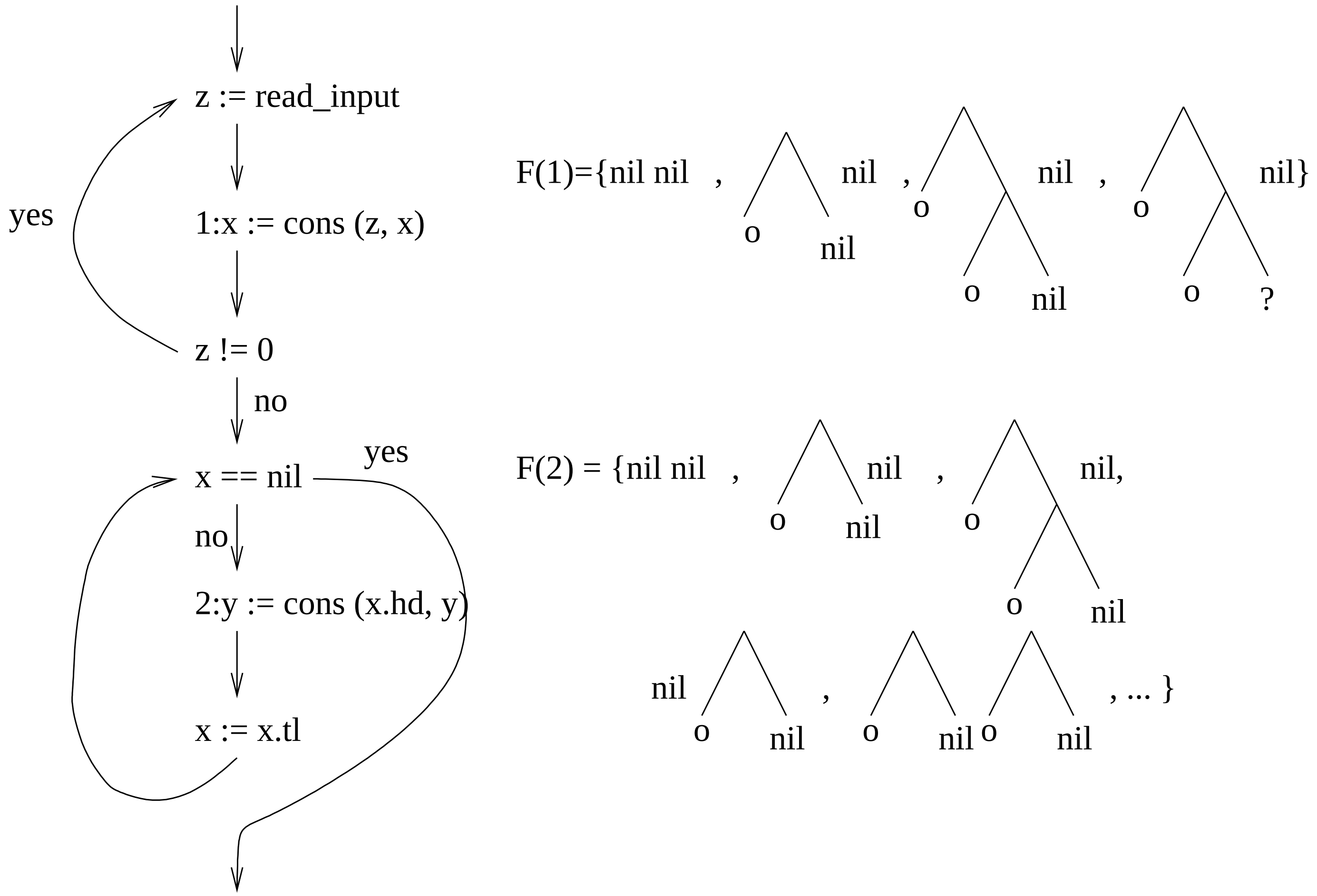
The shape-analysis contracts out-of-bound nodes into summary nodes (concretely, only node accessible from a variable by a selector-path of length $\leq k$ are explicitly represented)

For instance, any assignment results in the following graph transformations:

$$F(s) = \bigcup_{p \in \mathrm{pred}(s)} \mathrm{clean}(\mathrm{next}(F(p)))$$

where clean is a function that restore the k-limited character of graphs

Shape-analysis in [JM81]



Heuristic Approach - Overview [CWZ90]

Differences:

- single shape-graph (instead of set of shape-graphs)
- heuristic method (instead of k-approximation)
- heap reference count (instead of sharing annotations)

Achievements:

- better practical implementation
- preservation of more information about unbounded data structures

Heuristic Approach - Simple Algorithm [CWZ90]

Data structures: one shape-graph per statement with one node per variable and one node per `cons` statement

Process: add edges to the shape-graphs and eventally find the conservatively smallest number of edges

Two important points:

- at join[a] points, edge sets from predecessors are unioned to form the new edge set
- at assignments
  1. to a simple variable: the set of edges leaving the variable is replaced with the r-value (*strong update*)
  2. to a field: the set of edges leaving the field is added the r-value (*weak update*)

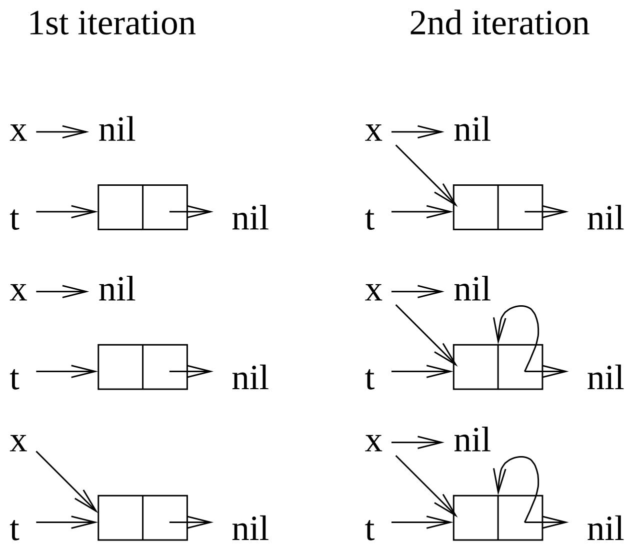
[a]control-flow join points

Managing Multiple Instances 1/4 [CWZ90]

Only one node for one `cons` statement?

For instance, bottom-up construction of a list:

```
x := nil

while ... do

    t := cons ()

    t.cdr := x

    x := t
```

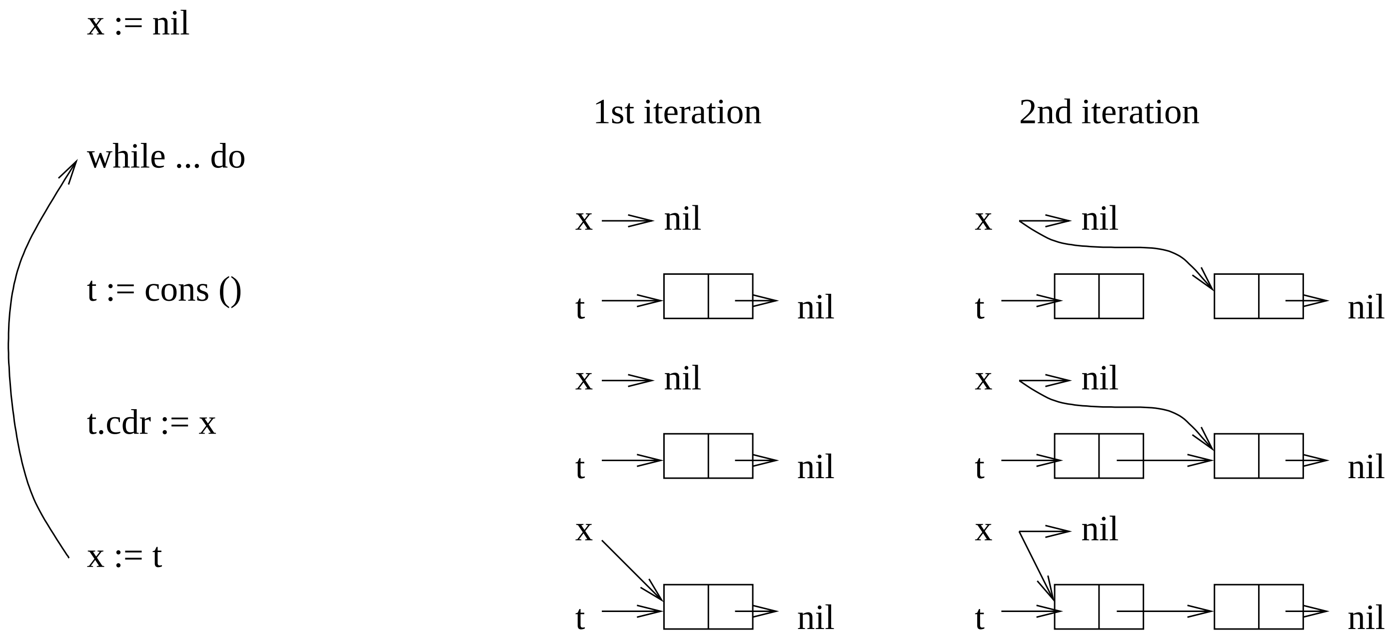


We cannot deduce that the data structure is acyclic!

Managing Multiple Instance 2/4 [CWZ90]

Several nodes for one `cons` statement?

For instance, bottom-up construction of a list:



The structure grows without bound!

Managing Multiple Instances 3/4 [CWZ90]

Two definitions:

1. A *deterministic* variable points to one shape-node and may possibly point to `nil`.
2. An shape-node is *interesting* if it is pointed to by a deterministic variable.

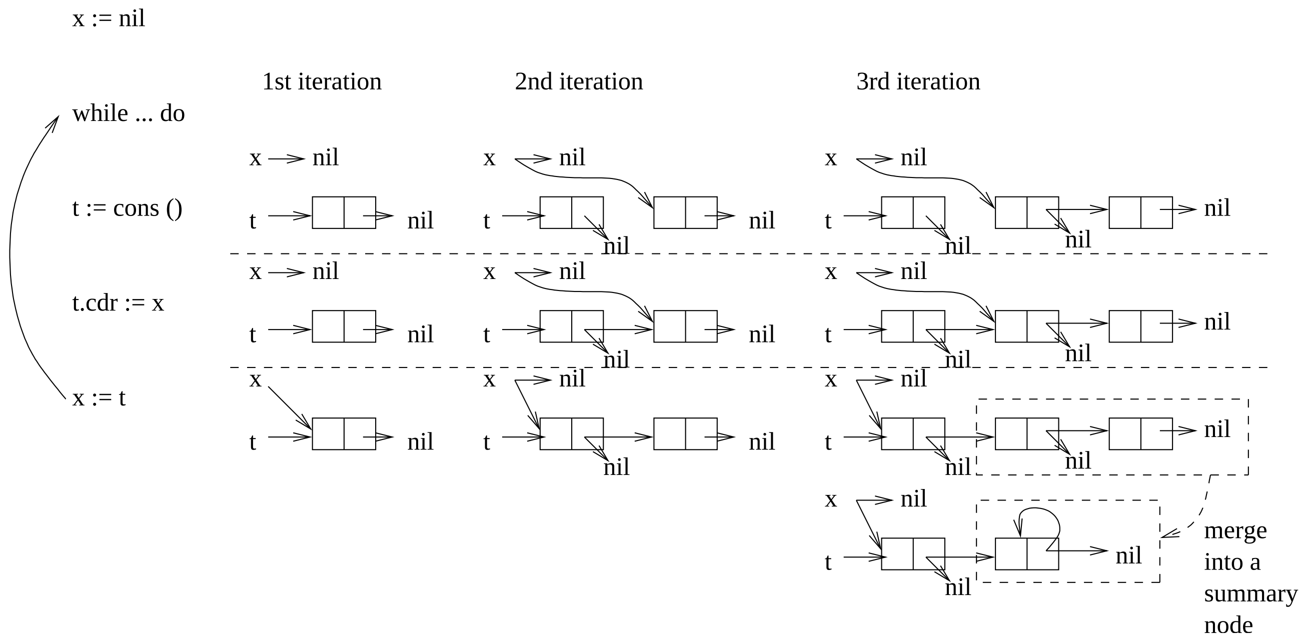
Merge rules:

- instances of the same `cons` statement are merged when they become uninteresting at assignments and join points[a]
- for each `cons` statement there can be at most one instance not pointed to by a deterministic variable: the summary node[b]

[a] join point $\neq$ merge point
[b] similar to [JM81]

Managing Multiple Instances 4/4 [CWZ90]

Merging at assignments. Example:



Merging at join points. Example:

(insert complicated decision procedure here)

Strong Updates 1/2 [CWZ90]

Weak updates are undesirable.

Strong update are possible:

- when the l-value is a single shape-node[a] and that shape-node corresponds on all execution paths to at most one `cons`-cell[b]
- when the l-value is a variable pointing to several shape-nodes and there are no other edges into any of those shape-nodes

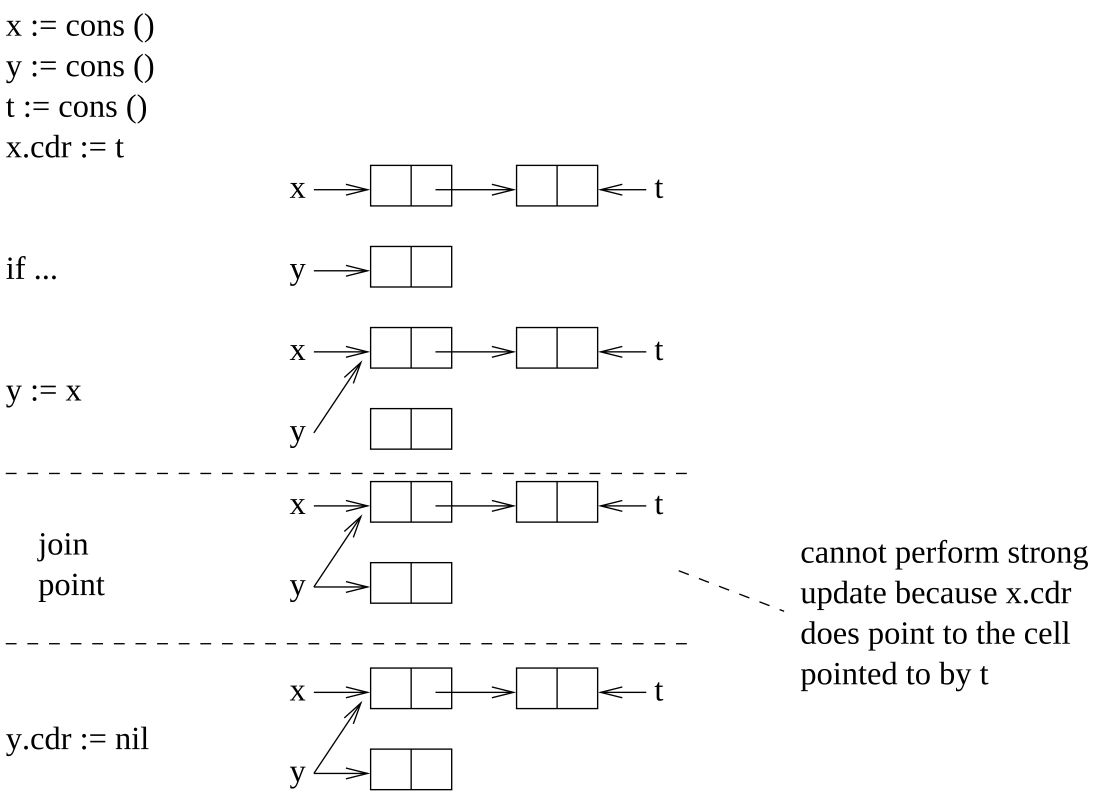
---

[a] assignment to a variable or to a shape-node reached by a deterministic variable

[b] the shape-node has not been merged with another shape-node on any execution path

Strong Updates 2/2 [CWZ90]

Limitation:



In assignment `y.cdr` := `nil`, the l-value is not a single shape-node

Shape Detection 1/2 [CWZ90]

Goal: Discover subgraphs of the shape-graph that represent lists or trees (i.e., graphs without sharing nor cycles)

Approach: Statically approximates the number of references from other heap cells with *heap reference count*

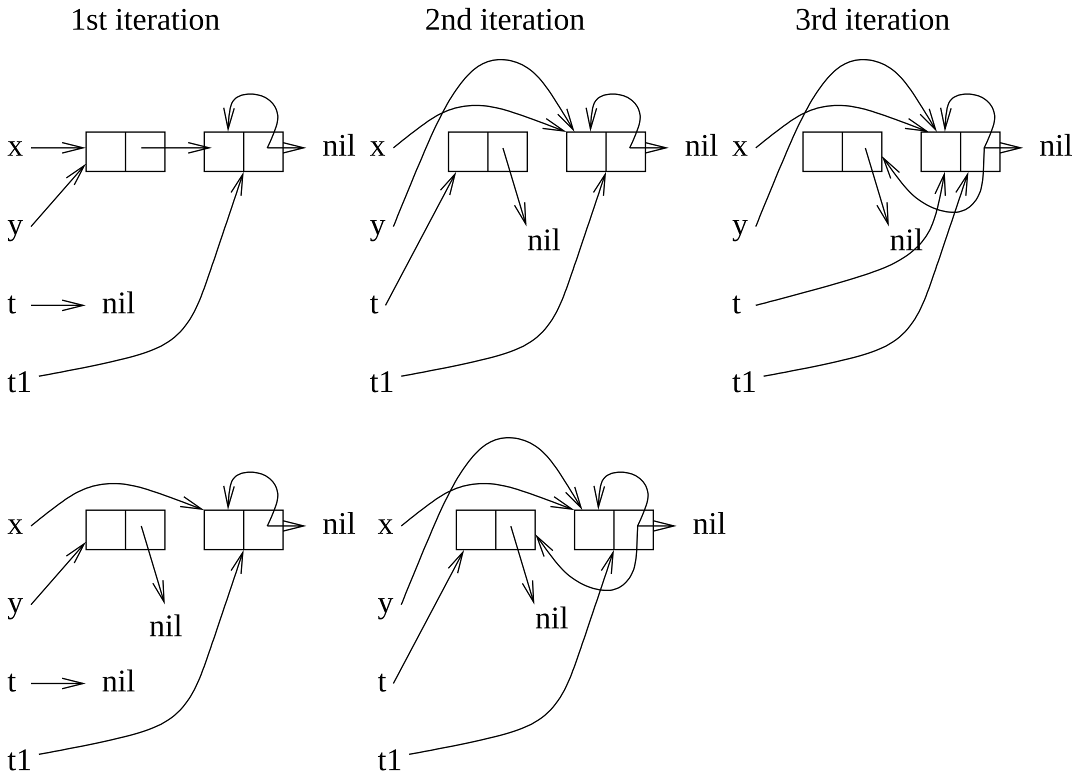
Key fact: A subgraph such that the root is not reachable from the other shape-nodes in the subgraph and all nodes except the root have their count set to one is composed of unshared shape-nodes

Achievements: Bottom-up construction of a list, top-down construction of a tree, destructive concatenation of two lists

Shape Detection 2/2 [CWZ90]

Limitation:

```
/* x points to a list */

y := nil

while x != nil

t := nil; t := y

y := nil; y := x

t1 := nil; t1 := x.cdr

x := nil; x := t1

y.cdr := nil; y.cdr := t

t1 := nil; t := nil

/* y points to the reversed list */
```



Problem: `y` may point to a possibly cyclic data structure and that the structures pointed to be `x` and `y` might share `cons`-cells

Sagiv-Reps-Wilhelm [SRW96]

Shape-graph:

- representation: one shape-node per statement, named with a set of variables (not with allocation sites)
- interpretation: a shape-node with variable set Z represents the unique `cons`-cell that is simultaneously pointed to by all (and only) the variables in Z; as a consequence, the partition of memory by the graph is not fixed

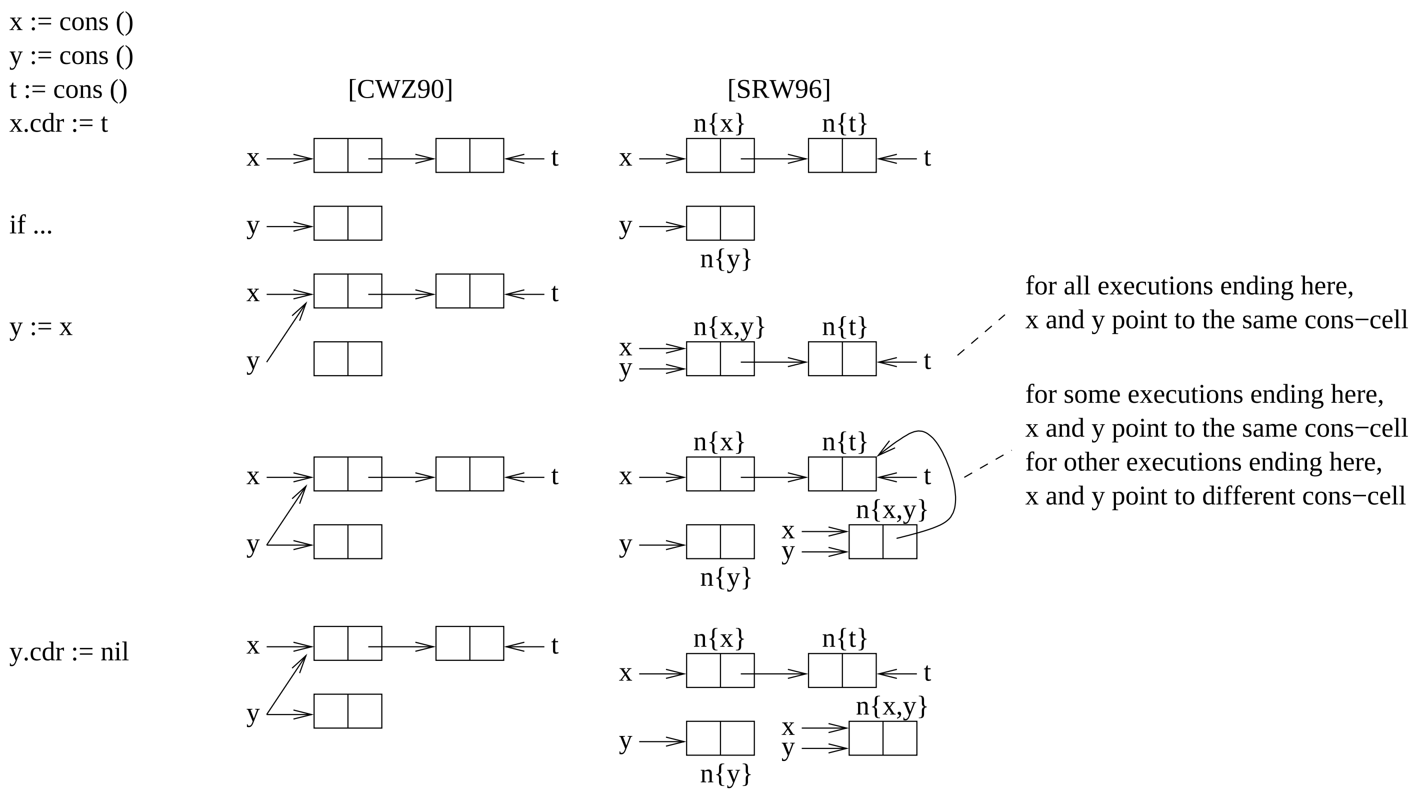
$\Rightarrow$ the analysis tracks all the possible aliasing configurations by performing operations on the variable sets that name the shape-nodes

Gain: strong nullification, materialization[a] (coupled with sharing information)

[a] of summary nodes, hereafter n_ϕ

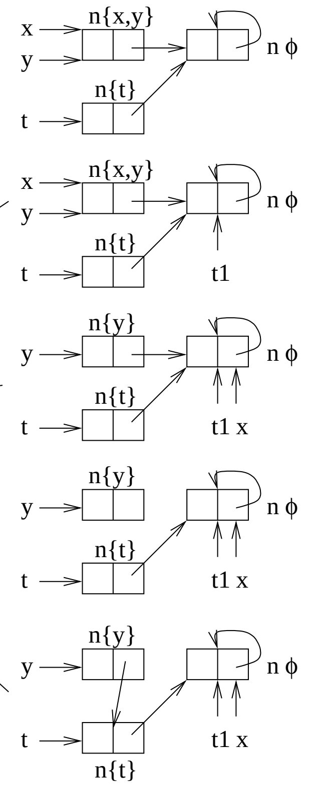
Strong nullification [SRW96]

For an assignment `x.cdr := nil`, the algorithm always removes the `cdr` edges emanating from what `x` points to. For instance:



Materialization and Sharing 1/3 [SRW96]

```
/* x points to a list */

y := nil

while x != nil

t := nil; t := y

y := nil; y := x

t1 := nil; t1 := x.cdr

x := nil; x := t1

y.cdr := nil; y.cdr := t

t1 := nil; t := nil

/* y points to the reversed list */
```



Problem: After `y.cdr := t`, `y`'s cdr points to `t` whose cdr points to n_ϕ which is pointed to by `x`: the two lists are no longer known to be separate![a]

---

[a]Nodes that are not pointed to by variables are clustered into a single node

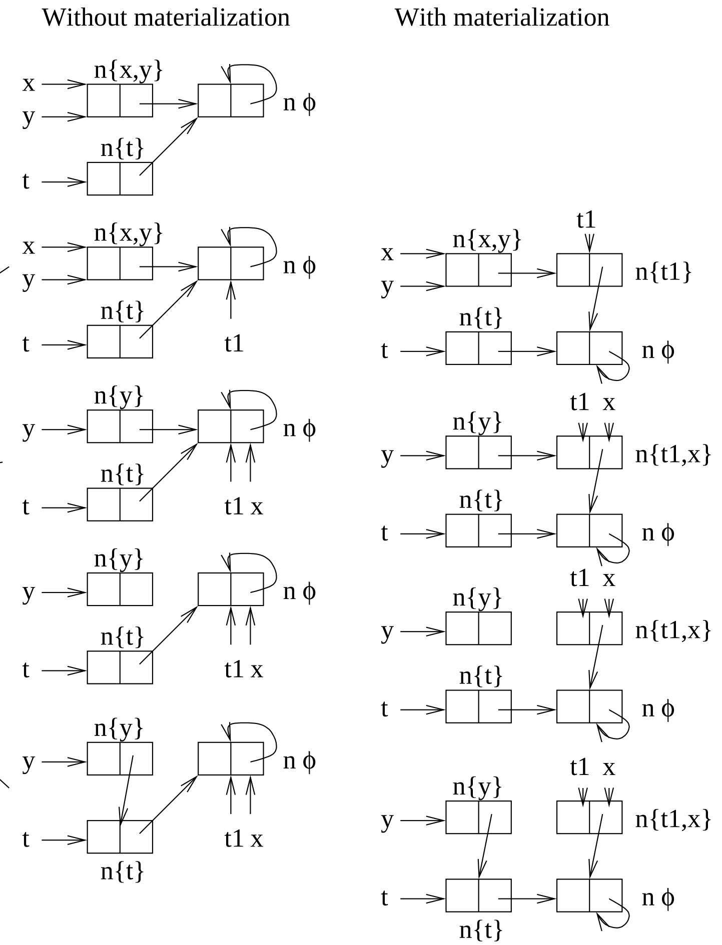
Materialization and Sharing 2/3 [SRW96]

For an assignment `x := y.cdr`, if `y`'s `cdr` was assigned a summary node (n_ϕ), we *materialize* a copy as a non-summary node.

We use *sharing information* to increase the accuracy of the primitive operations used by our method (boolean flag indicating whether or not the shape-node may be the target of more than one pointer from the heap)

Materialization and Sharing 3/3 [SRW96]

```
/* x points to a list */
y := nil
while x != nil
t := nil; t := y
y := nil; y := x
t1 := nil; t1 := x.cdr
x := nil; x := t1
y.cdr := nil; y.cdr := t
t1 := nil; t := nil
/* y points to the reversed list */
```



Achievements [SRW96]

For certain programs, able to determine that:

- if the input is a list, the output is also a list (e.g., list reversal via destructive update, list search and insertion[a])
- if the input is a tree, the output is also a tree
- if the input is a circular list, the output is also a circular list (e.g., insertion)

---

[a]Uses the original property that a shared node may become unshared

Conclusion

- Shape-analysis addresses multiple problems (from pointer analysis to type-checking)
- Different results can be extracted from the results of shape-analysis (shared data structures detection and may-aliases [SRW96])