Shape Analysis

The Shape Analysis Approach

- *Pointers* and *heap-allocated* storage are features of all modern imperative programming languages.
- They are ignored by most semantic descriptions of imperative programming languages, because they complicate it.
- Using pointers often causes errors. Two common errors:
    - Dereferencing NULL pointers
    - Accessing previously deallocated storage.

- The *Shape Analysis* is useful for:
    - Debugging and optimization of the code.
    - Program verification

Concrete questions:

- Alias: Do two pointer expressions reference the same heap cell?
    - Yes (for every state): Trigger a prefetch or predict a cache hit
- Sharing: Is a heap cell shared?
    - Yes (for some state): explicit deallocation may run into an inconsistent state
- Reachability: Is a heap cell reachable from a specific variable or from any pointer variable?
- Disjointness: Do two data structures pointed to by two distinct pointer variables ever have common elements?
    - No (for every state): Distribute data structures to different processors
- Ciclicity: Is a heap cell part of a cycle?
    - No (for every state): Perform garbage collection by reference counting
- Shape: What will be the «shape» of (some part of) the heap contents?

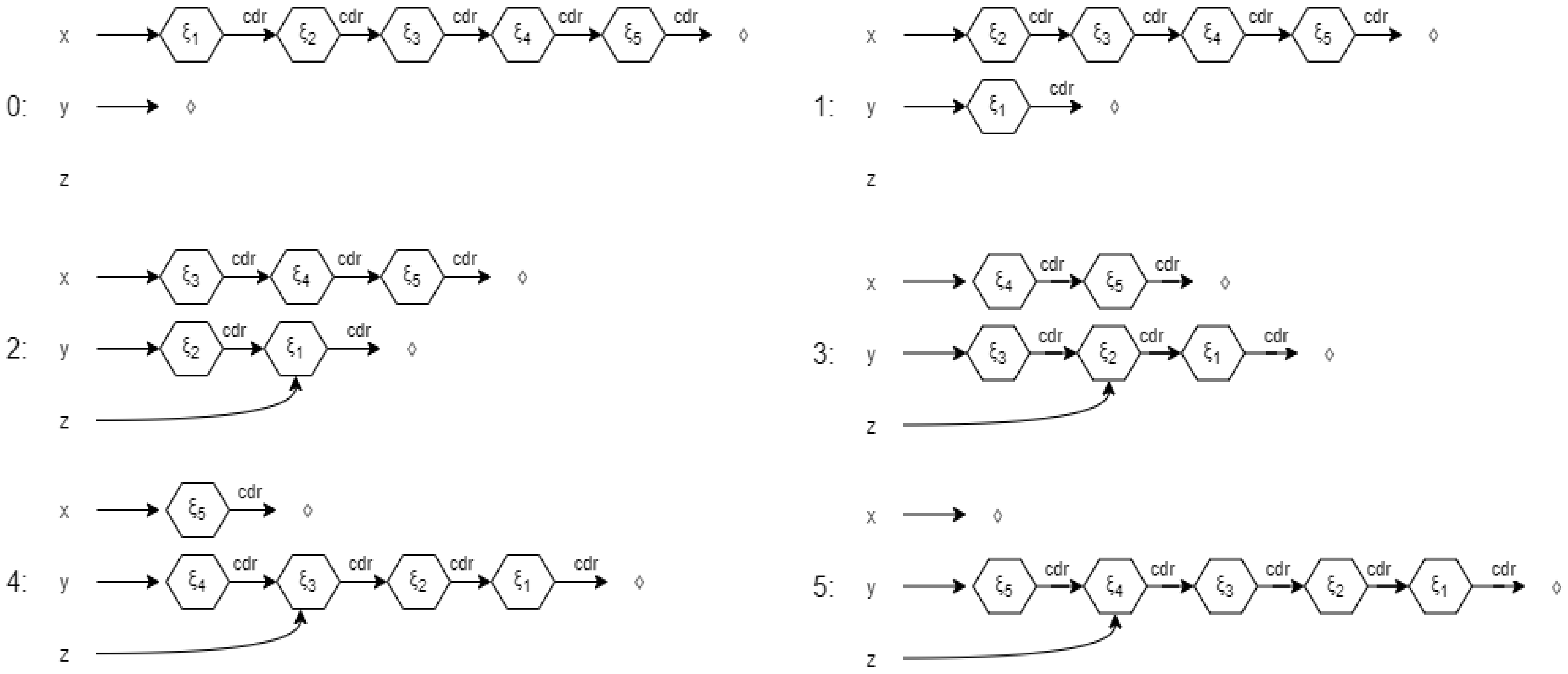
Formally:

- **Goal**: for each program point, for each variable, obtain a *finite description* of the heap-allocated data structures resulting from any execution.
- **Problem:** mapping a heap of potentially unbounded size to a graph of bounded size.

Definition. A (concrete) *heap configuration* is given by $(Loc, Sel, Var, \sigma, \mathcal{H})$, where:

- Loc is an infinite set of locations (or addresses) for the heap cells $\xi \in Loc$
- Sel is a finite set of selector names
- Var is a finite set of program variables
- $\sigma \in State = Var \rightarrow (Z + Loc + \{\lozenge\})$ is a variable valuation
- $\mathcal{H} \in Heap = (Loc \times Sel) \rightarrow_{fin} (Z + Loc + \{\lozenge\})$ is a (concrete) heap

```
[y:=nil]1;
while [not is-nil(x)]2 do
        [z: =y]3;
        [y: =x]4;
        [x: =x.cdr]5;
        [y.cdr: =z]6;
[z:=nil]7
```



Shape graphs

- We have to explicitly abstract from a concrete heap to the form of a bounded graph: a *shape graph*.
- A shape graph is defined from the concept of *abstract location*, the representative for one (or more) heap cells of the program heap.

$$ALoc = \{n_X \mid X \subseteq Var\} \qquad abstract\ locations$$

- Idea: if $x \in Var$ points to ξ_{I}, then it belongs to the set X of n_X
- We introduce the abstract *summary location* $\mathbf{n}_\emptyset$ that will represent all the heap cells that are not directly pointed by a state variable.

Definition. A shape graph (S,H,is) consists of

- An abstract state S - maps variables to abstract locations.

$$S \in AState = \mathcal{P}(Var \times ALoc)$$

- Abstract heap *H* - maps abstract locations to abstract locations via selectors.

$$H \in AHeap = \mathcal{P}(ALoc \times Sel \times ALoc)$$

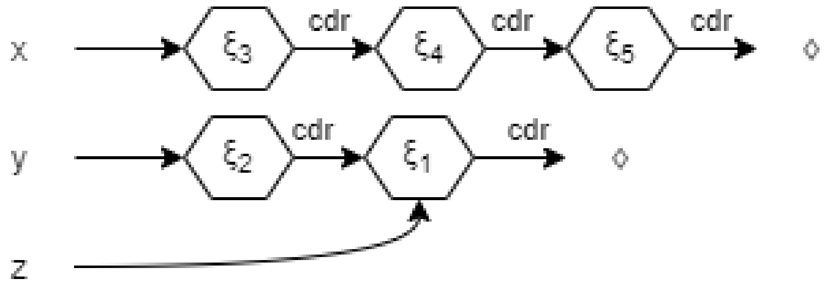
Idea: ($\xi_1 \xrightarrow{sel} \xi_2 \quad \wedge \quad (\xi_1 \longmapsto \mathrm{n}_V$ and $\xi_2 \longmapsto \mathrm{n}_W)) \Rightarrow (n_V, sel, n_W) \in H$.

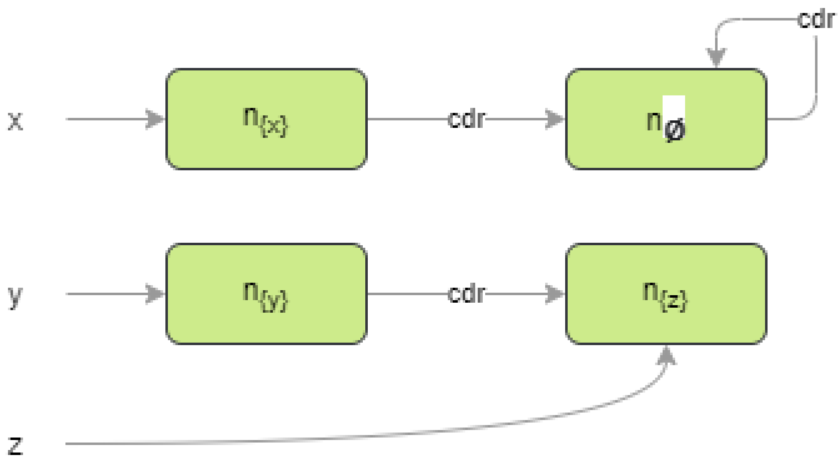
- An IsShared set *is* - abstract locations that represents locations that are shared due to pointers in the heap.

Nodes → Abstract locations
Labelled Edges → defined by H
Unlabelled Edges → defined by S

- Variables x, y and z point to diffent locations, so:
  - $\xi_3 \longmapsto n_{\{x\}}$
  - $\xi_2 \longmapsto n_{\{y\}}$
  - $\xi_1 \longmapsto n_{\{z\}}$
  - $\xi_4, \xi_5 \longmapsto n_{\emptyset}$.
- $S = \{(x, n_{\{x\}}), (y, n_{\{y\}}), (z, n_{\{z\}})\}$
- $H = \{(n_{\{x\}}, cdr, n_{\emptyset}), (n_{\emptyset}, cdr, n_{\emptyset}), (n_{\{y\}}, cdr, n_{\{z\}})\}$
- No abstract locations are shared

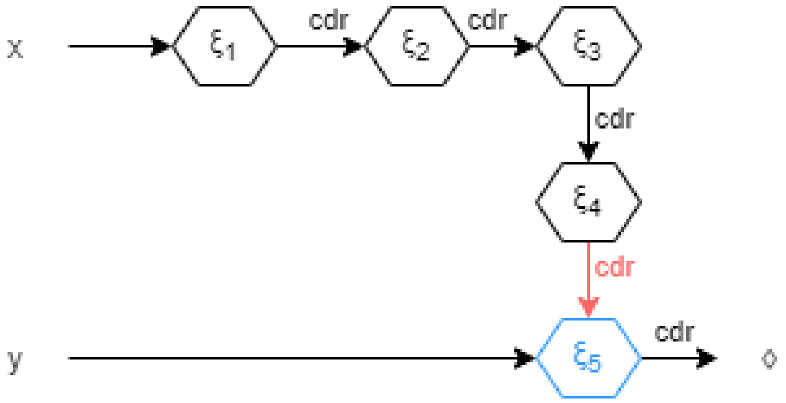
2:

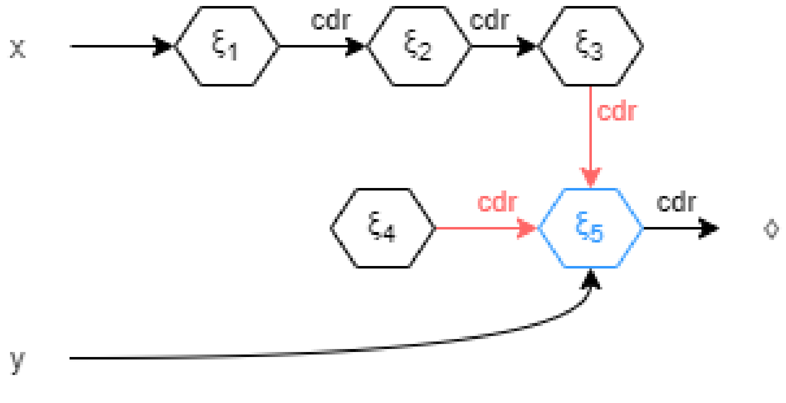


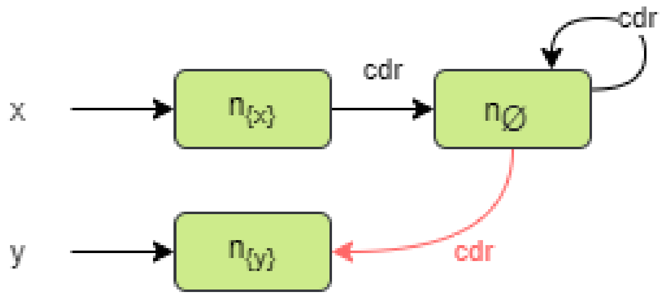


When is the *is* set useful?

- An abstract location n_X will be included in is if it does represent a location that targets more that one pointer in the heap.
- In the first row abstract location $n_{\{y\}}$ representing location ξ_5 is not shared, so $n\{y\} \notin is$.
- In the second case, ξ_5 is shared, so $n_{\{y\}} \epsilon\ is$.







- To summerise, a shape graph is a triple $(\mathrm{S}, \mathrm{H}, \mathrm{is}) \in AState \times AHeap \times IsShared$, with:

$$S \in AState = \mathcal{P}(Var \times ALoc)$$
$$H \in AHeap = \mathcal{P}(ALoc \times Sel \times ALoc)$$
$$is \in IsShared = \mathcal{P}(ALoc)$$

- Given the CFG of the program, determine for all nodes ℓ all the possible shape graphs entering and leaving the program nodes that summarize the possible heap configurations for that node.
- **Basically**: We have to find a fixpoint solution for $Shape(\ell) = < Shape_{enter}(\ell), Shape_{exit}(\ell) >$ for every ℓ.

$Shape(\ell)$ will operate over sets of shape graphs, i.e. elements of $\mathcal{P}(SG)$.

Domain: $(D, \sqsubseteq) \coloneqq (2^{SG}, \subseteq)$ $\quad (Var, ALoc, Sel$ finite $\Rightarrow SG$ finite $\Rightarrow 2^{SG}$ finite $\Rightarrow ACC)$

$power\ set \Longrightarrow complete\ lattice$

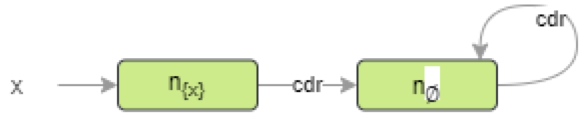
The Analysis

$$Shape_{enter}(\ell) = \begin{cases} \iota, & if\ \ell = init(S) \\ \bigcup \{Shape_{exit}(\ell') \mid \ell' \in pre[\ell]\}, & otherwise \end{cases}$$

$$Shape_{exit}(\ell) = f_\ell^{SA}(Shape_{enter}(\ell))$$

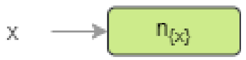
Where:

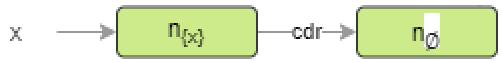
- $pre[\ell]$ is the set of predecessors of the node ℓ.
- $init(S)$ computes the initial label for the statement S.
- ι is an initial set of shape graph for possible initial values of variables. In the case of our reverse program:
  - "x points to a (finite) acyclic list of at least 3 elements"

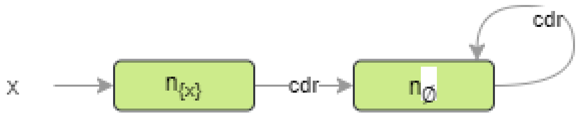
$\iota = \{$ 

$\}$

  - "x points to any (finite) acyclic list"

$\iota = \{ (\emptyset, \emptyset, \emptyset),$ 

, 

, 

$\}$

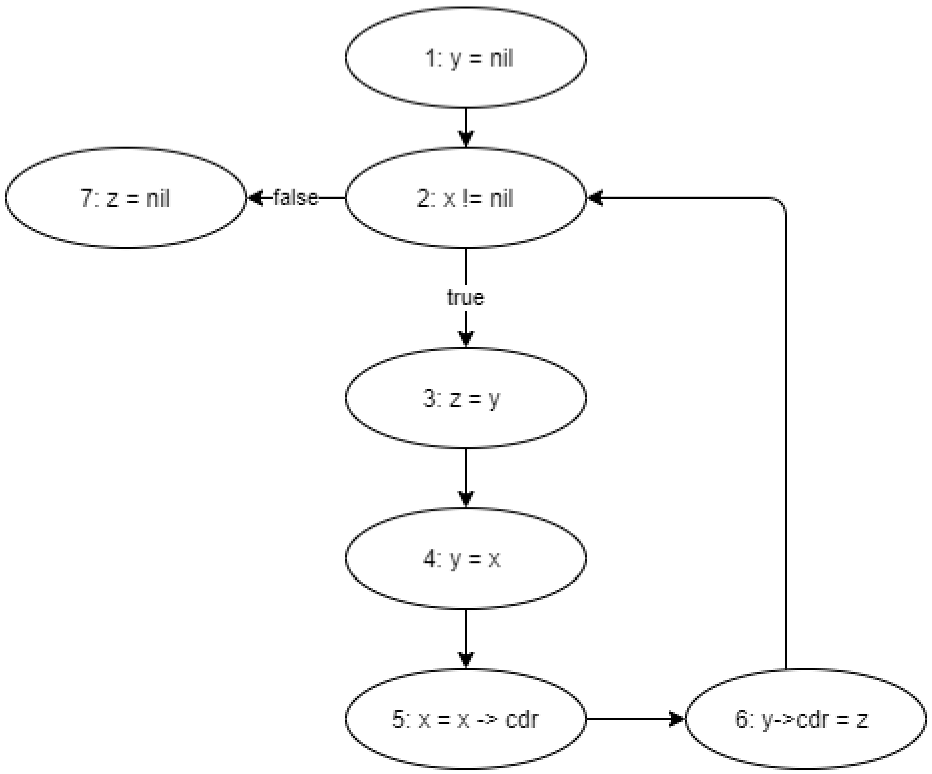
- Forward analysis
- Possible analysis
- Some aspects of a must analysis

Consider again the list reversal program:

$[y{:=}nil]_1;$
while $[\text{not is-nil}(x)]_2$ do
$([z:=y]_3; [y:=x]_4; [x:=x.cdr]_5; [y.cdr:=z]_6);$
$[z{:=}nil]_7$

Assume that x initially points to an unshared list with at least two elements and that y and z are initially undefined.

$$Shape_{exit}(1) = f_1^{SA}(Shape_{enter}(1)) = f_1^{SA}(\iota)$$
$$Shape_{exit}(2) = f_2^{SA}(Shape_{enter}(2)) = f_2^{SA}(Shape_{exit}(1) \cup Shape_{exit}(6))$$
$$Shape_{exit}(3) = f_3^{SA}(Shape_{enter}(3)) = f_3^{SA}(Shape_{exit}(2))$$
$$Shape_{exit}(4) = f_4^{SA}(Shape_{enter}(4)) = f_4^{SA}(Shape_{exit}(3))$$
$$Shape_{exit}(5) = f_5^{SA}(Shape_{enter}(5)) = f_5^{SA}(Shape_{exit}(4))$$
$$Shape_{exit}(6) = f_6^{SA}(Shape_{enter}(6)) = f_6^{SA}(Shape_{exit}(5))$$
$$Shape_{exit}(7) = f_7^{SA}(Shape_{enter}(7)) = f_7^{SA}(Shape_{exit}(2))$$

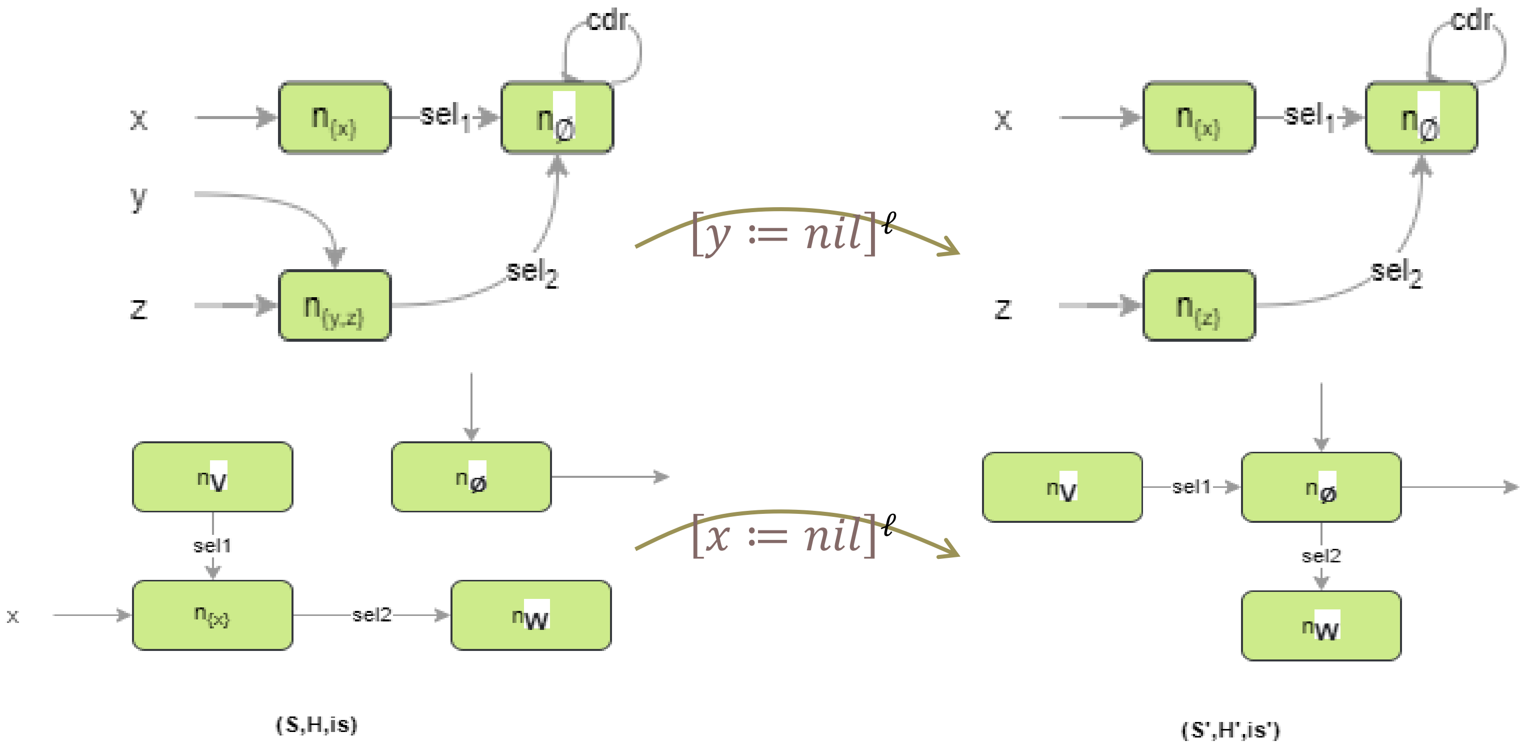


An SG is modified by evaluation of assignments.

Transfer function $f_\ell^{SA}: P(SG) \rightarrow P(SG)$ defines how to modify input shape graphs' components (S, H, is) to represent all possible shape graph that can be generated by effects of the elementary block labelled ℓ.

$[b]^\ell$ *and* $[skip]^\ell$ These commands does not modify heap's content.

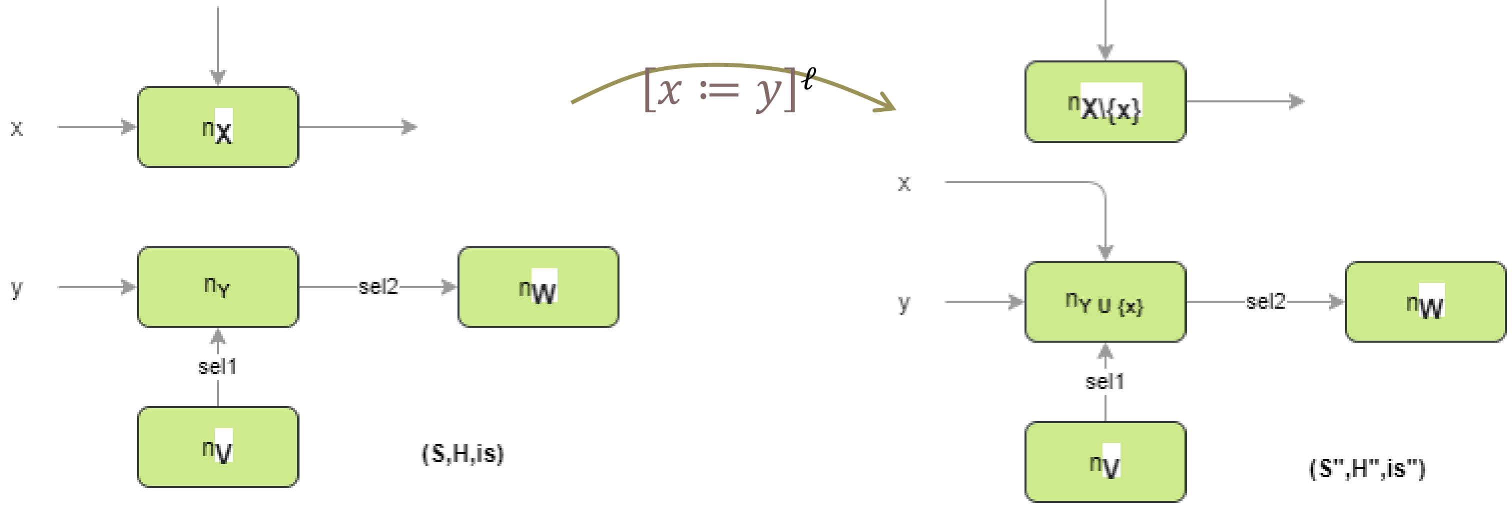
$[x := nil]^\ell$ The effects will be to remove the binding to x, and to rename all abstract locations so that they do not include x in their name.



$[x := y]^\ell$

If $x \neq y$:

- First visible effect: remove the old bindings to x.
- Second visible effect: the new bindings to x is recorded.
  All abstract locations are renamed to include x in their name if they already have y.

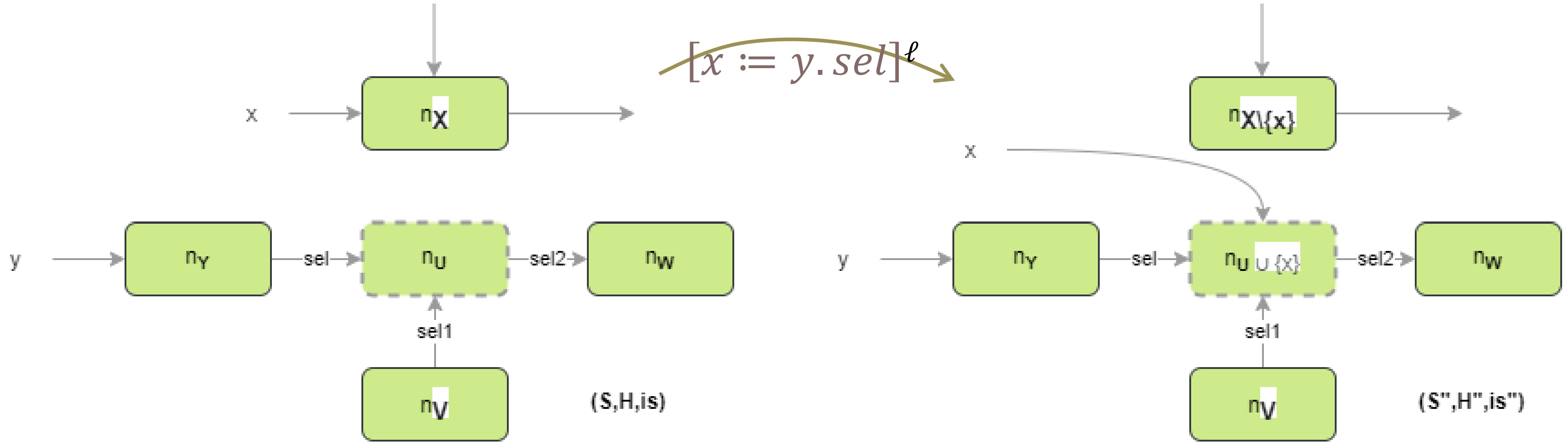


$[x := y.sel]^\ell$

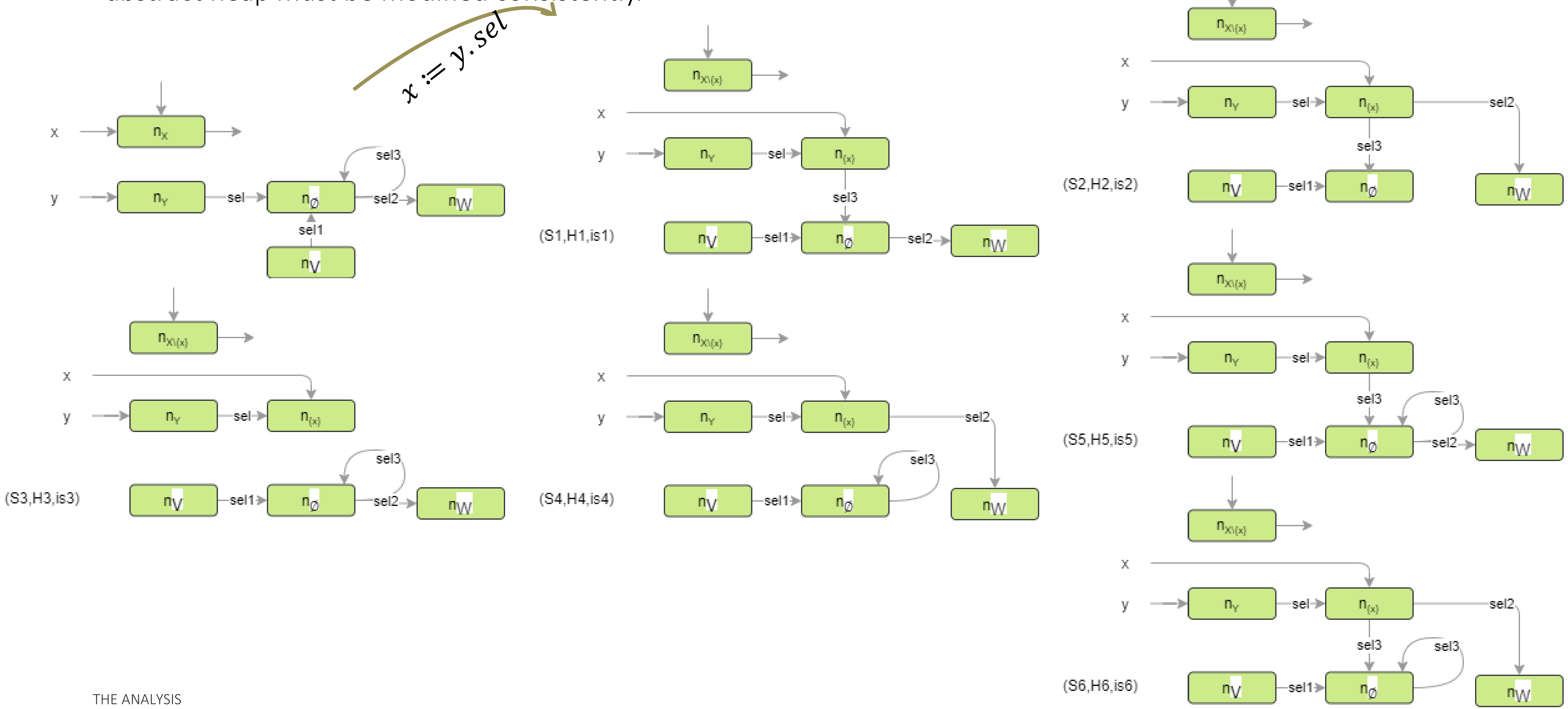
Assume that $x \neq y$.

- First visible effect: remove the old binding for x.
- Second visible effect: rename abstract location corresponding to y.sel to include x in its name and to establish binding of x to that abstract location.
  Who is y.sel pointed to? We have 3 possibilities…

1. $(y, n_Y) \notin S'$ or $(y, n_Y) \in S'$, but there is no n_Z such that $(n_Y, sel, n_Z) \in H'$.
    I. In the first case, we have no effect.
    II. In the second case, only remove the old bindings to x.
2. $(y, n_Y) \in S'$ and there is an abstract location $n_U \neq n_\emptyset$ such that $(n_Y, sel, n_U) \in H'$.
   The abstract location n_U will be renamed to include the variable x.

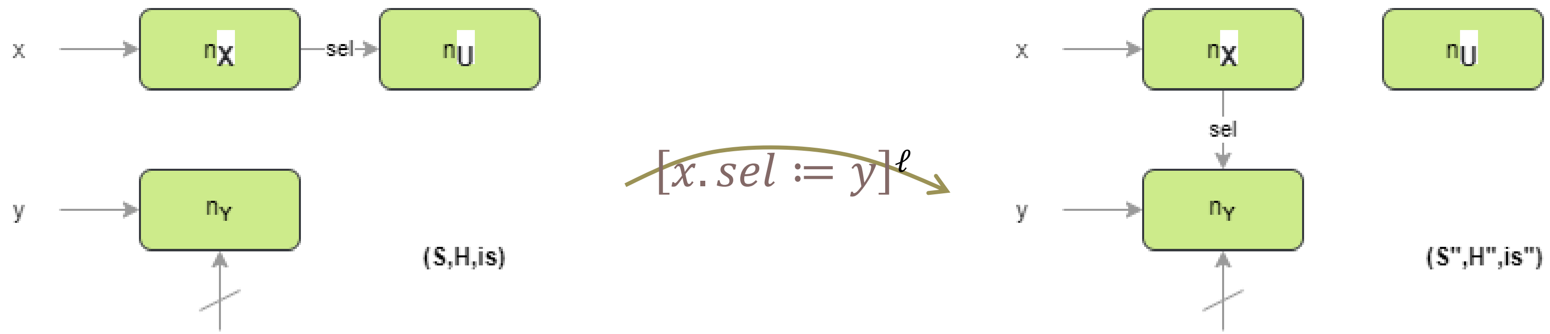


3. There is an abstract location n_Y such that $(y, n_Y) \in S'$ and $(n_Y, sel, n_\emptyset) \in H'$.
The location $n_\emptyset$ describes location for y.sel as well as a set of other locations.
   - Intuitively, the statement $[x \coloneqq y.sel]\ell$ in this case outputs a new abstract location $n_{\{x\}}$ from $n_\emptyset$ that describes the location for y.sel and $n_\emptyset$ will continue to represent remaining locations. As it is introduced a new abstract location. the abstract heap must be modified consistently.



$[x.sel := y]^{\ell}$

- Assume that $x \neq y$. As usual,if $(x, nX) \notin S$, x will not point to a cell in the heap, so the statement will have no effect on the shape of the heap.
- Let's assume that $(x, n_X) \in S$. We need to remove from H all triples $(n_X, sel, n_W) \in H$.
- Let's assume that $(x, n_X) \in S$ and $(y, n_Y) \in S$. It this case, we must establish the new binding given by the assignment.



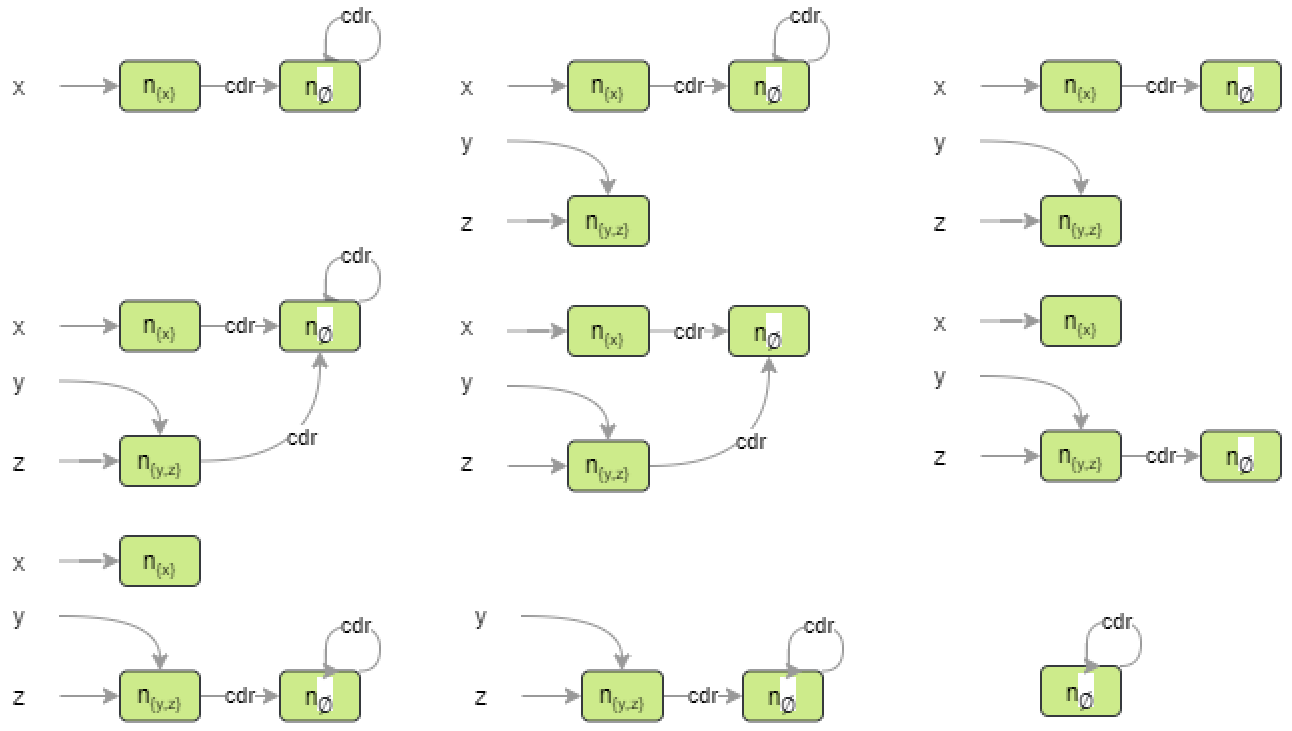
Fixpoint solution yields $SG_\ell \subseteq SG$, for each $\ell \in Lab$.

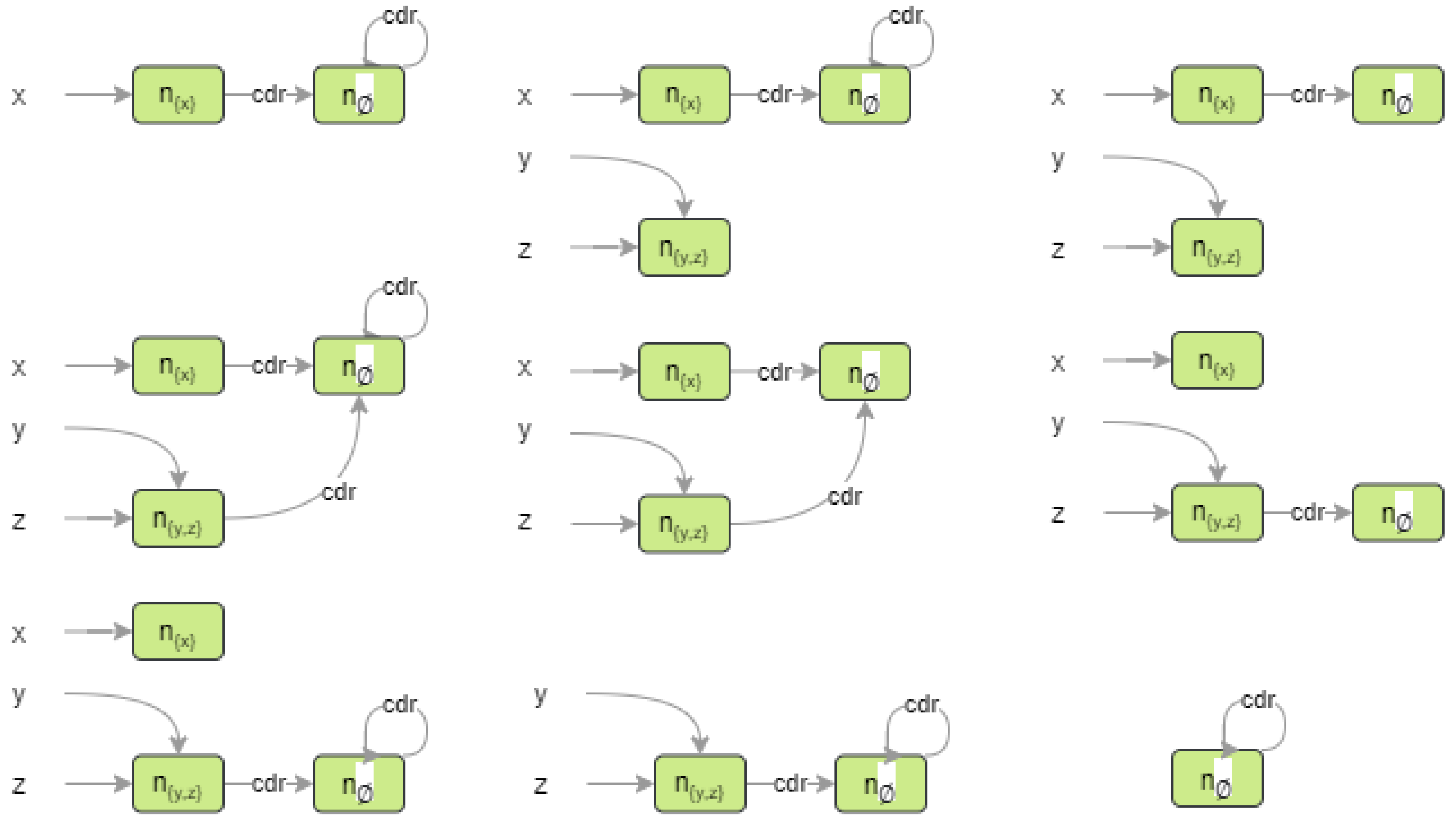
Solving shape analysis's equations for our reverse program requires too much time (and generates a lot of shape graphs… approx 50). Let's show only the potential of this analysis with this particular result:

$[y{:=}nil]_1$;
while $[\text{not is-nil}(x)]_2$ do
    $([z{:} {=}y]_3; [y{:} {=}x]_4; [x{:} {=}x.cdr]_5; [y.cdr{:} {=}z]_6)$;
$[z{:=}nil]_7$

For example, we could have the following shape graphs, given by $Shape_{exit}(3)$:

- The description of the lists occurring during execution is finite: there are 9 shape graphs describing all x- and y-lists arising after 3.





Some conclusions we can draw after 3:

- No heap cell is shared
- x and y point to acyclic data structure
- z and y are alias or both point to nil.

Other (correct) conclusions we can't draw after 3:

- The lists to which x and y point are disjoint.
- x never points to nil.