Solving Shape-Analysis Problems in Languages with Destructive Updating

#SAV Presentation

Authors

Mooly Sagiv, Thomas W. Reps, Reinhard Wilhelm

Presenters

Marco Antognini, Sandro Stucki

May 2015

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Motivation

Analyse the shape of heap allocated data structures.

Verify that a program preserves shape properties such as:

- *list-ness*
- *circular list-ness*
- *tree-ness*

This analysis algorithm can be used to find aliases, and therefore to optimise code (no alias means it can be more easily parallelised).

List Reversal – Normalisation

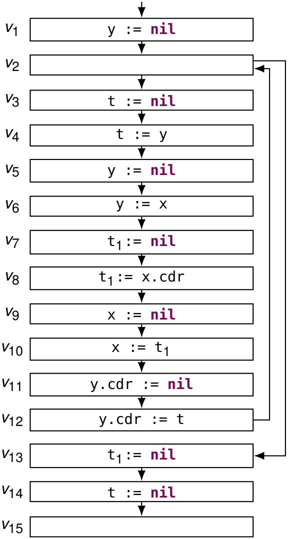
```
// x points to an unshared list
y := nil
while x ≠ nil do

  t := y

  y := x




  x := x.cdr

  y.cdr := t
od

t := nil
```

List Reversal – Normalisation

```
// x points to an unshared list
y := nil
while x ≠ nil do
  t := nil
  t := y
  y := nil
  y := x
  t1 := nil
  t1 := x.cdr
  x := nil
  x := t1
  y.cdr := nil
  y.cdr := t
od
t1 := nil
t := nil
```

List Reversal – Normalisation

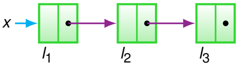
```
// x points to an unshared list
y := nil
while x ≠ nil do
  t := nil
  t := y
  y := nil
  y := x
  t₁ := nil
  t₁ := x.cdr
  x := nil
  x := t₁
  y.cdr := nil
  y.cdr := t
od
t₁ := nil
t := nil
```



Shape-Graph $SG = \langle E_v, E_s \rangle$

What is a **shape-graph**?

- directed graph with nodes and edges
- nodes are called **shape-nodes**
  - runtime locations, i.e. heap memory, or *cons-cells*
  - implicitly defined by edges and $shape\_nodes(SG)$
- edges are divided into 2 categories:
  - E_v: **variable-edges** of the form $[x, n]$
  - E_s: **selector-edges** of the form $\langle s, sel, t \rangle$



DSG: Deterministic Shape-Graph

A shape-graph is deterministic if

1. no variable points to more than one node
2. no node has a selector pointing to more than one node

In other words:

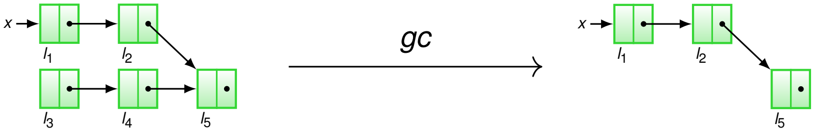
It is deterministic when edges are behaving like pointers.

gc: Garbage Collection

The *gc* function removes runtime location that are not reachable from any program variable:

$gc : SG \rightarrow SG$

$gc\,(\langle E_v, E_s \rangle) \stackrel{\text{def}}{=} \langle E_v, E'_s \rangle$ where $E'_s \subseteq E_s$ and $\langle s, sel, t \rangle \in E'_s$ if and only if there exists $[x, r] \in E_v$ such that there is a path of selector-edges in E_s from r to s.



DSG Transformers

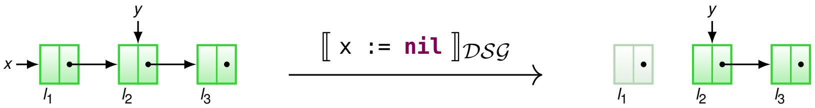
A program statement is represented as a transform function on *DSG*, denoted $[\![\ \texttt{st}\ ]\!]_{\mathcal{DSG}}$.

Thanks to program normalisation, there are only 6 statements and they are simple:

1. $[\![\ \texttt{x := } \textbf{nil}\ ]\!]_{\mathcal{DSG}}$
2. $[\![\ \texttt{x.sel := } \textbf{nil}\ ]\!]_{\mathcal{DSG}}$
3. $[\![\ \texttt{x := } \textbf{new}\ ]\!]_{\mathcal{DSG}}$
4. $[\![\ \texttt{x := y}\ ]\!]_{\mathcal{DSG}}$
5. $[\![\ \texttt{x := y.sel}\ ]\!]_{\mathcal{DSG}}$
6. $[\![\ \texttt{x.sel := y}\ ]\!]_{\mathcal{DSG}}$

DSG Transformers: 1

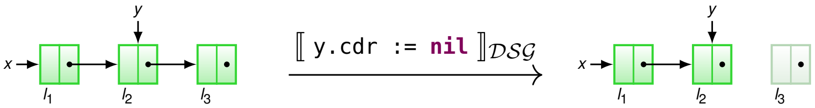
$$[\![\ \texttt{x := nil} \ ]\!]_{\mathcal{DSG}} (\langle E_v, E_s \rangle)$$
$$\stackrel{\text{def}}{=} \langle E_v - \{[\mathrm{x}, *]\}, E_s \rangle$$



NB: *gc* can do some cleaning.

DSG Transformers: 2

$$[\![\ \texttt{x.sel := }\textbf{nil}\ ]\!]_{\mathcal{DSG}}(\langle E_v, E_s\rangle) \stackrel{\text{def}}{=} \langle E_v, E_s - \{\langle s, \texttt{sel}, *\rangle \mid [\mathrm{x}, s] \in E_v\}\rangle$$



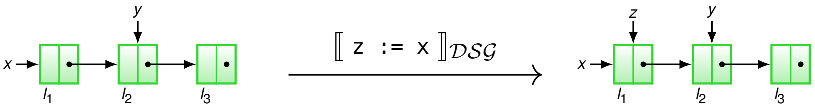
NB: *gc* can do some cleaning.

DSG Transformers: 3

$$
\llbracket\ \texttt{x := new}\ \rrbracket_{\mathcal{DSG}} \left(\langle E_v, E_s \rangle\right) \stackrel{\text{def}}{=} \langle E_v \cup \{[\mathrm{x}, \mathrm{n}_{new}]\}\ ,\ E_s \rangle
$$

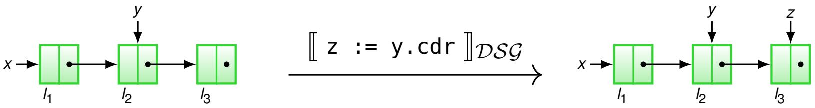
DSG Transformers: 4

$$
\llbracket \texttt{x := y} \rrbracket_{\mathcal{DSG}} (\langle E_v, E_s \rangle) \stackrel{\text{def}}{=} \langle E_v \cup \{[\mathrm{x}, \mathrm{n}] \mid [\mathrm{y}, \mathrm{n}] \in E_v\}, E_s \rangle
$$



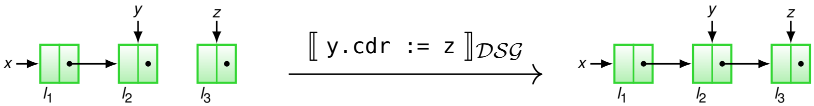
DSG Transformers: 5

$$\llbracket\ \texttt{x := y.sel}\ \rrbracket_{\mathcal{DSG}}\left(\langle E_v, E_s\rangle\right)$$
$$\stackrel{\text{def}}{=} \langle E_v \cup \{[\mathrm{x},\mathrm{t}] \mid [\mathrm{y},\mathrm{s}] \in E_v \wedge \langle \mathrm{s},\mathrm{sel},\mathrm{t}\rangle \in E_s\}\ , E_s\rangle$$



DSG Transformers: 6

$$\llbracket\ \texttt{x.sel := y}\ \rrbracket_{\mathcal{DSG}}\left(\langle E_v, E_s\rangle\right) \stackrel{\text{def}}{=} \langle E_v, E_s \cup \{\langle \mathrm{s}, \mathrm{sel}, \mathrm{t}\rangle \mid [\mathrm{x}, \mathrm{s}] \in E_v \wedge [\mathrm{y}, \mathrm{t}] \in E_v\}\rangle$$



Collecting Semantics

Each DSG models one runtime behaviour (of many).

We need a tool to perform analysis on **all** possible execution paths, not just one.

Hence the collecting function $cv : V \rightarrow 2^{DSG}$

$$cv(v) \stackrel{\text{def}}{=} \{ [\![\texttt{st}(v_k)]\!]_{\mathcal{DSG}} (\cdots ([\![\texttt{st}(v_1)]\!]_{\mathcal{DSG}} (\langle \varnothing, \varnothing \rangle))) \mid [v_1, \ldots, v_k] \in pathsTo(v) \}$$

V is the vertex set of the *regular* control flow graph $G = \langle V, A \rangle$.

Abstraction, Abstractly

DSG

Abstraction, Abstractly

$\subseteq \quad \cup$

$\{DSG\}$

Abstraction, Abstractly

$\subseteq \quad \cup$

$\sqsubseteq \quad \sqcup$

$$\{DSG\} \xrightarrow{\alpha} SSG$$

Abstraction, Abstractly

$\subseteq \quad \cup$

$\sqsubseteq \quad \sqcup$

$$\{DSG\} \xrightarrow{\quad\alpha\quad} SSG$$

Static shape graphs: $SSG = \langle SG, is\_shared \rangle$

- SG is a shape graph
- $is\_shared : shape\_nodes(SG) \rightarrow \{T, F\}$
  is a predicate identifying nodes that were shared in the DSG

Abstraction – Helpers

- Grouping nodes by variable labels

$$\alpha_s[DSG] \colon \mathit{shape\_nodes}(DSG) \to \{n_X \mid X \subseteq \mathit{PVar}\}$$

$$\alpha_s[DSG](r) \stackrel{\text{def}}{=} n_{\{x \in \mathit{PVar} \mid [x,r] \in E_v\}}$$

- Initialisation of the sharing predicate

$$\mathit{induced\_is\_shared}[DSG] \colon \mathit{shape\_nodes}(DSG) \to \{T, F\}$$

$$\mathit{induced\_is\_shared}[DSG](t) \stackrel{\text{def}}{=} |\{\langle *, *, t\rangle \in E_s\}| \leq 2$$

- Projection (a.k.a. quotienting) of SGs with respect to f

$$\langle SG, p\rangle \downarrow f$$

Abstraction

- Projection/quotient of a single DSG

$$\hat{\alpha} : \mathcal{DSG} \to \mathcal{SSG}$$

$$\hat{\alpha}(DSG) \stackrel{\text{def}}{=} \langle DSG', induced\_is\_shared[DSG'] \rangle \downarrow \alpha_s[DSG']$$
$$\text{where } DSG' = gc(DSG)$$

- Abstraction function

$$\alpha : 2^{\mathcal{DSG}} \to \mathcal{SSG}$$

$$\alpha(S) \stackrel{\text{def}}{=} \bigsqcup_{DSG \in S} \hat{\alpha}(DSG)$$

Abstract Interpretation, Abstractly

DSG_1

Abstract Interpretation, Abstractly

$$\begin{array}{c} DSG_1 \\ \Big\downarrow{\scriptstyle [\![s]\!]_{\mathcal{DSG}}} \\ DSG_2 \end{array}$$

Abstract Interpretation, Abstractly

$\subseteq \quad \cup$

$$\begin{array}{c}\{DSG_1\} \\ \Big\downarrow{\scriptstyle \{[\![s]\!]_{\mathcal{DSG}}\}} \\ \{DSG_2\}\end{array}$$

Abstract Interpretation, Abstractly

$\subseteq \quad \cup$ $\qquad\qquad$ $\sqsubseteq \quad \sqcup$

$$
\begin{tikzcd}
\{ DSG_1 \} \arrow[rr, "\alpha"] \arrow[d, "{\{ [\![s]\!]_{\mathcal{DSG}} \}}"'] & & SSG_1 \\
\{ DSG_2 \} & &
\end{tikzcd}
$$

Abstract Interpretation, Abstractly

$\subseteq \quad \cup$ $\qquad\qquad$ $\sqsubseteq \quad \sqcup$

$$
\begin{tikzcd}
\{DSG_1\} \arrow[rr, "\alpha"] \arrow[d, "{\{[\![s]\!]_{\mathcal{DSG}}\}}"'] & & SSG_1 \\
\{DSG_2\} \arrow[r, "\alpha"] & SSG_2 &
\end{tikzcd}
$$

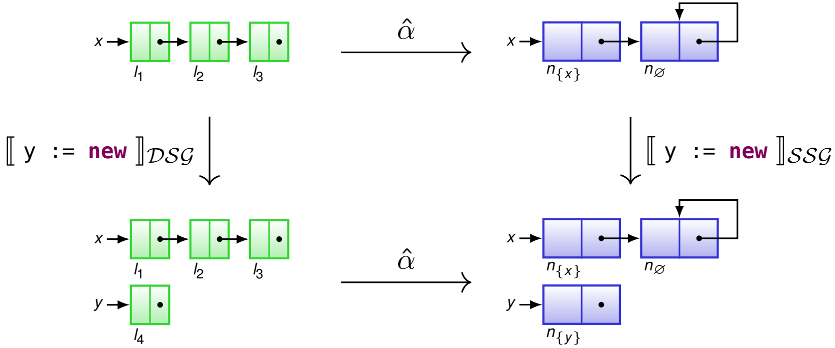
Abstract Interpretation, Abstractly

$$\boxed{\subseteq \quad \cup} \qquad\qquad\qquad \boxed{\sqsubseteq \quad \sqcup}$$

$$\begin{tikzcd}
\{DSG_1\} \arrow[r, "\alpha"] \arrow[d, "{\{[\![s]\!]_{\mathcal{DSG}}\}}"'] & SSG_1 \arrow[d, "{[\![s]\!]_{\mathcal{SSG}}}"] \\
\{DSG_2\} \arrow[r, "\alpha"] & SSG_2 \sqsubseteq SSG_2'
\end{tikzcd}$$

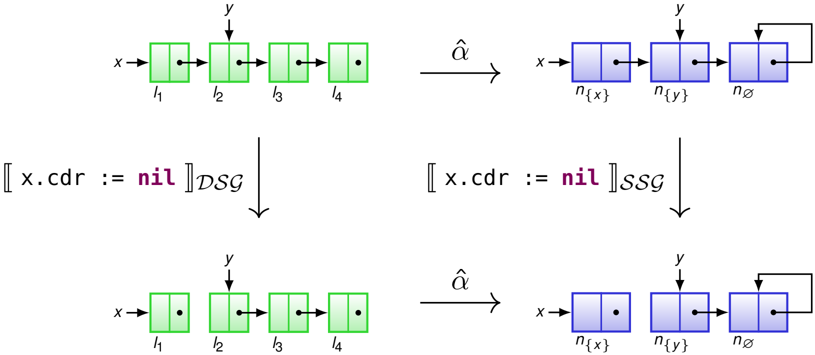
Abstract Interpretation – Examples

Allocating a new node.



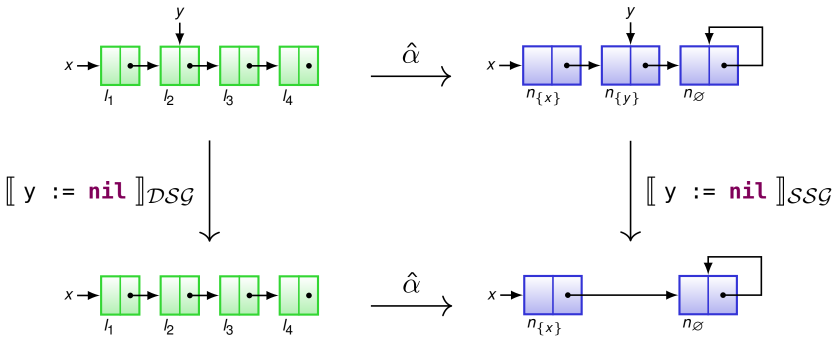
Abstract Interpretation – Examples (cont.)

Assigning **nil** to a field.



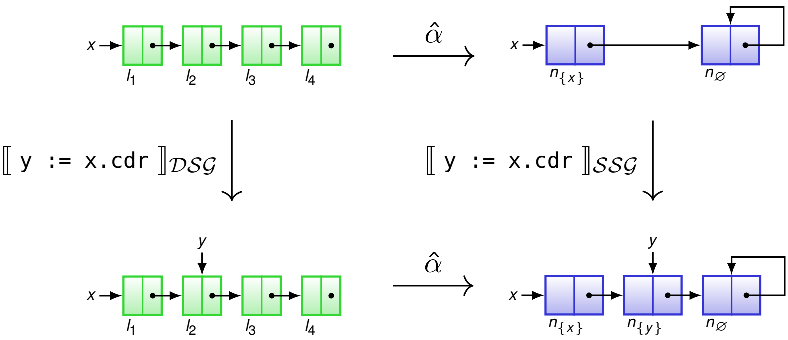
Abstract Interpretation – Examples (cont.)

Assigning **nil** to a variable.



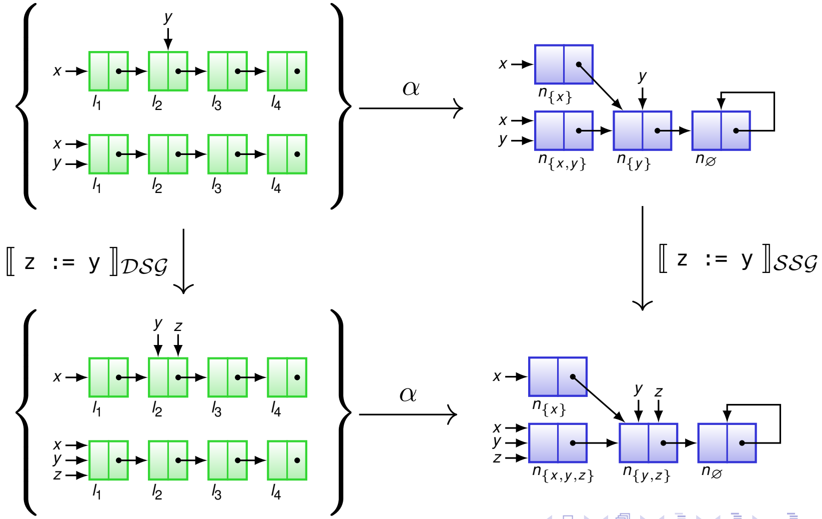
Abstract Interpretation – Examples (cont.)

Materialising a node n_y from the summary node $n_\varnothing$.



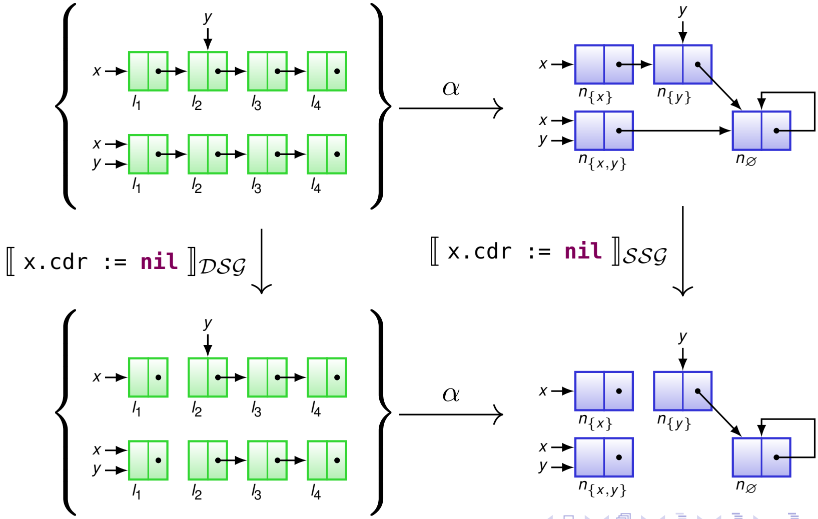
Abstract Interpretation – Examples (cont.)

Variable assignment.



Abstract Interpretation – Examples (cont.)

Strong nullification.



List Insertion – Normalisation

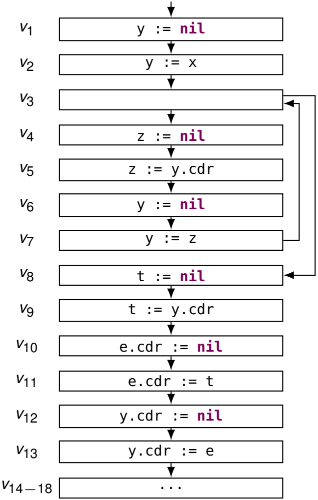
```
// x is an unshared list
// e the element to insert

y := x
while y.cdr ≠ nil ∧... do

  z := y.cdr

  y := z
od

t := y.cdr

e.cdr := t

y.cdr := e
t := nil
z := nil
e := nil
y := nil
```

List Insertion – Normalisation

```
// x is an unshared list
// e the element to insert
y := nil
y := x
while y.cdr ≠ nil ∧... do
  z := nil
  z := y.cdr
  y := nil
  y := z
od
t := nil
t := y.cdr
e.cdr := nil
e.cdr := t
y.cdr := nil
y.cdr := e
t := nil
z := nil
e := nil
y := nil
```

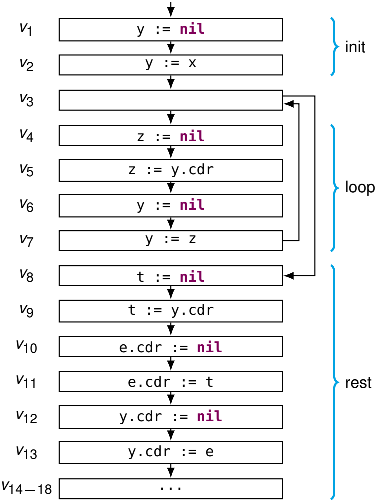
List Insertion – Normalisation

```
// x is an unshared list
// e the element to insert
y := nil
y := x
while y.cdr ≠ nil ∧... do
  z := nil
  z := y.cdr
  y := nil
  y := z
od
t := nil
t := y.cdr
e.cdr := nil
e.cdr := t
y.cdr := nil
y.cdr := e
t := nil
z := nil
e := nil
y := nil
```



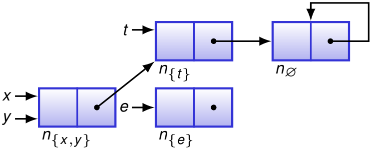
List Insertion – Normalisation

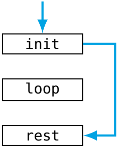
```
// x is an unshared list
// e the element to insert
y := nil
y := x
while y.cdr ≠ nil ∧... do
  z := nil
  z := y.cdr
  y := nil
  y := z
od
t := nil
t := y.cdr
e.cdr := nil
e.cdr := t
y.cdr := nil
y.cdr := e
t := nil
z := nil
e := nil
y := nil
```



Sharing

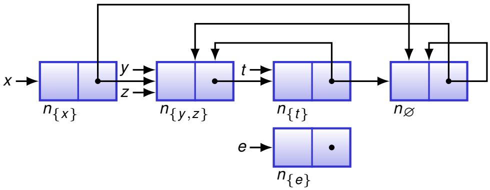
From v_1 to v_{11} *without entering* the loop.
Executing $[\![$ `e.cdr := nil` $]\!]$ – $n_{\{t\}}$ is **not** shared.

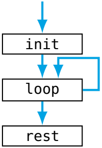




Sharing

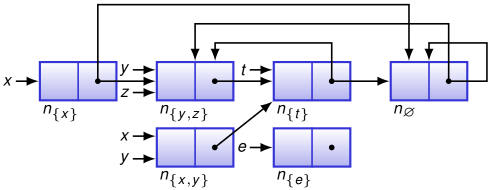
From v_1 to v_{11} *through* the loop.
Executing $[\![$ `e.cdr := nil` $]\!]$ – $n_{\{t\}}$ is **not** shared.

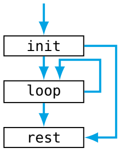




Sharing

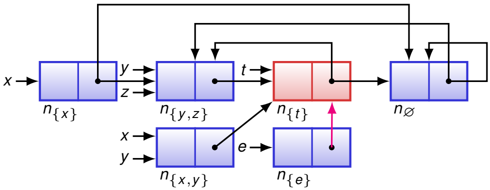
From v_1 to v_{11} by all possible paths.
Executing $[\![$ `e.cdr := nil` $]\!]$ – $n_{\{t\}}$ is still **not** shared.

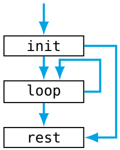




Sharing

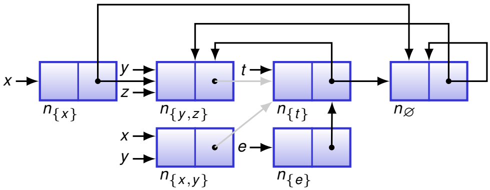
From v_1 to v_{12} by all possible paths.
Executing $[\![$ `e.cdr := t` $]\!]$ – $n_{\{t\}}$ **is** shared.

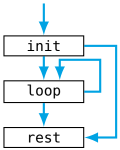




Sharing

From v_1 to v_{13} by all possible paths.
Executing $[\![$ `y.cdr := nil` $]\!]$ – **strong nullification.**





Extensions

Merging Shape-Nodes to avoid a huge number of nodes ($\leq 2^{|PVar|}$), a widening operator can be introduced.

Finding Aliases and Sharing testing whether x and y are aliases at some point of the program can be extended to test whether two paths can alias by introducing two extra variables.

Interprocedural Analysis *shape-graph-transformations* can be introduced to accurately model procedures.

Representing Definitely Circular Structures with extra special nodes ($n_{atom}, n_{nil}, n_{uninit}$), definitely cyclic data structures can be modelled.

Thank you!

Questions?