Shape Analysis

Nicola Corti & Alessandro Baroni

University of Pisa
Static Analysis Techniques course

8 May 2014

Index

What is the **Shape Analysis**?

Shape Analysis

An intraprocedural analysis aimed to figure out the shape of an heap-allocated memory.

1. **Extend** the WHILE language with command for heap management,
2. Present an **abstract** representation for the heap memory,
3. Present the analysis like a **monotone framework**.

Use case of the **Shape Analysis**

- `nil`-pointer dereferencing,
- Checking field existence (e.g. `a.sel := 1`, what if `a` does not have a `sel` field?),
- Validating properties of data structure shape (e.g. a non-cyclic structure is still non-cyclic after a computation).

Selectors and Pointers

Selectors

$$sel \in \mathbf{Sel}$$

Pointers

$$p \in \mathbf{PExp}$$

$$p ::= x \mid x.sel$$

Extended Syntax

The extended syntax with pointers

$$
\begin{aligned}
a &::= p \mid n \mid a_1\ op_a\ a_2 \mid \texttt{nil} \\
b &::= \texttt{true} \mid \texttt{false} \mid \texttt{not}\ b \mid b_1\ op_b\ b_2 \mid a_1\ op_r\ a_2 \mid op_p\ p \\
S &::= [p := a]^\ell \mid [\texttt{skip}]^\ell \mid S_1; S_2 \mid \texttt{if}\ [b]^\ell\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \\
&\quad\ \texttt{while}\ [b]^\ell\ \texttt{do}\ S \mid [\texttt{malloc}\ p]^\ell
\end{aligned}
$$

Note that op_r now accept two operands of type a, such as two pointer (for an operation such as `are-equals`) and the operator op_p accept one pointer operands (think at operations like `is-nil`).

The operator $[\texttt{malloc}\ p]^\ell$ allow to allocate new space in the heap.

Structural Operational Semantics

We add values for locations:

$$\xi \in \mathbf{Loc}$$

From now on a configuration of the semantics will be composed by a **state** and a **heap**

$$\sigma \in \mathbf{State} = \mathbf{Var}_* \to (\mathbf{Z} + \mathbf{Loc} + \{\diamond\})$$
$$\mathcal{H} \in \mathbf{Heap} = (\mathbf{Loc} \times \mathbf{Sel}) \to_{\text{fin}} (\mathbf{Z} + \mathbf{Loc} + \{\diamond\})$$

Note that the heap $\mathcal{H}$ need a $\mathbf{Loc}$ and a $\mathbf{Sel}$ to return a value. The $\to_{\text{fin}}$ represent the fact that not all the selector fields will be defined. The value $\diamond$ represent the `nil` value.

Structural Operational Semantics

We add values for locations:

$$\xi \in \mathbf{Loc}$$

From now on a configuration of the semantics will be composed by a **state** and a **heap**

$$\sigma \in \mathbf{State} = \mathbf{Var}_* \to (\mathbf{Z} + \mathbf{Loc} + \{\diamond\})$$
$$\mathcal{H} \in \mathbf{Heap} = (\mathbf{Loc} \times \mathbf{Sel}) \to_{\text{fin}} (\mathbf{Z} + \mathbf{Loc} + \{\diamond\})$$

Note that the heap $\mathcal{H}$ need a $\mathbf{Loc}$ and a $\mathbf{Sel}$ to return a value. The $\to_{\text{fin}}$ represent the fact that not all the selector fields will be defined. The value $\diamond$ represent the `nil` value.

Structural Operational Semantics

We add values for locations:

$$\xi \in \mathbf{Loc}$$

From now on a configuration of the semantics will be composed by a **state** and a **heap**

$$\sigma \in \mathbf{State} = \mathbf{Var}_* \to (\mathbf{Z} + \mathbf{Loc} + \{\diamond\})$$
$$\mathcal{H} \in \mathbf{Heap} = (\mathbf{Loc} \times \mathbf{Sel}) \to_{\text{fin}} (\mathbf{Z} + \mathbf{Loc} + \{\diamond\})$$

Note that the heap $\mathcal{H}$ need a $\mathbf{Loc}$ and a $\mathbf{Sel}$ to return a value. The $\to_{\text{fin}}$ represent the fact that not all the selector fields will be defined. The value $\diamond$ represent the `nil` value.

Pointer Expressions

We need to define a new **semantic function** for **pointers**

$$\wp : \mathbf{PExp}_* \to (\mathbf{State} \times \mathbf{Heap}) \to_{\text{fin}} (\mathbf{Z} + \mathbf{Loc} + \{\diamond\})$$

$$\wp[\![x]\!](\sigma, \mathcal{H}) = \sigma(x)$$

$$\wp[\![x.sel]\!](\sigma, \mathcal{H}) = \begin{cases} \mathcal{H}(\sigma(x), sel) \\ \quad \text{if } \sigma(x) \in \mathbf{Loc} \quad \wedge \\ \quad \mathcal{H} \text{ is defined on } (\sigma(x), sel) \\ \\ undef \\ \quad \text{if } \sigma(x) \notin \mathbf{Loc} \quad \vee \\ \quad \mathcal{H} \text{ is undefined on } (\sigma(x), sel) \end{cases}$$

Arithmetic & Boolean Expressions

We need to update the older semantic function to work with the new heap:

$$\mathcal{A} : \mathbf{AExp} \to (\mathbf{State} \times \mathbf{Heap}) \to_{\text{fin}} (\mathbf{Z} + \mathbf{Loc} + \{\diamond\})$$
$$\mathcal{B} : \mathbf{BExp} \to (\mathbf{State} \times \mathbf{Heap}) \to_{\text{fin}} \mathbf{T}$$

The new clause for arithmetic function are:

$$\begin{aligned}
\mathcal{A}[\![p]\!](\sigma, \mathcal{H}) &= \wp[\![p]\!](\sigma, \mathcal{H}) \\
\mathcal{A}[\![n]\!](\sigma, \mathcal{H}) &= \mathcal{N}[\![n]\!] \\
\mathcal{A}[\![a_1\ op_a\ a_2]\!](\sigma, \mathcal{H}) &= \mathcal{A}[\![a_1]\!](\sigma, \mathcal{H})\ \mathbf{op}_a\ \mathcal{A}[\![a_2]\!](\sigma, \mathcal{H}) \\
\mathcal{A}[\![\texttt{nil}]\!](\sigma, \mathcal{H}) &= \diamond
\end{aligned}$$

Arithmetic & Boolean Expressions

The new clause for boolean function are:

$$\begin{aligned} \mathcal{B}[\![a_1\ op_r\ a_2]\!](\sigma,\mathcal{H}) &= \mathcal{A}[\![a_1]\!](\sigma,\mathcal{H})\ \mathbf{op}_r\ \mathcal{A}[\![a_2]\!](\sigma,\mathcal{H}) \\ \mathcal{B}[\![op_p\ p]\!](\sigma,\mathcal{H}) &= \mathbf{op}_p\ (\wp[\![p]\!](\sigma,\mathcal{H})) \end{aligned}$$

Note that the meaning of $\mathbf{op}_a$ and $\mathbf{op}_r$ must be undefined if the types are not the same (e.g. two integers or two pointers).

$$\mathbf{is\text{-}nil}(v) = \begin{cases} \text{tt} & \text{if } v = \diamond \\ \text{ff} & \text{otherwise} \end{cases}$$

Statements

We extended the statements rules with the heap:

$$\langle [x := a]^\ell, \sigma, \mathcal{H} \rangle \rightarrow \langle \sigma[x \mapsto \mathcal{A}[\![a]\!](\sigma, \mathcal{H})], \mathcal{H} \rangle$$
$$\text{if } \mathcal{A}[\![a]\!](\sigma, \mathcal{H}) \text{ is defined}$$

$$\langle [x.sel := a]^\ell, \sigma, \mathcal{H} \rangle \rightarrow \langle \sigma, \mathcal{H}[(\sigma(x), sel) \mapsto \mathcal{A}[\![a]\!](\sigma, \mathcal{H})] \rangle$$
$$\text{if } \sigma(x) \in \mathbf{Loc} \text{ and } \mathcal{A}[\![a]\!](\sigma, \mathcal{H}) \text{ is defined}$$

Statements

We extended the statements rules with the heap:

$$\langle [x := a]^\ell, \sigma, \mathcal{H} \rangle \rightarrow \langle \sigma[x \mapsto \mathcal{A}[\![a]\!](\sigma, \mathcal{H})], \mathcal{H} \rangle$$
$$\text{if } \mathcal{A}[\![a]\!](\sigma, \mathcal{H}) \text{ is defined}$$

$$\langle [x.sel := a]^\ell, \sigma, \mathcal{H} \rangle \rightarrow \langle \sigma, \mathcal{H}[(\sigma(x), sel) \mapsto \mathcal{A}[\![a]\!](\sigma, \mathcal{H})] \rangle$$
$$\text{if } \sigma(x) \in \mathbf{Loc} \text{ and } \mathcal{A}[\![a]\!](\sigma, \mathcal{H}) \text{ is defined}$$

Statements

We add rules for the `malloc` statement, to allow allocation of new cells.

$$\langle [\texttt{malloc}\ x]^\ell, \sigma, \mathcal{H} \rangle \ \rightarrow\ \langle \sigma[x \mapsto \xi], \mathcal{H} \rangle$$
$$\text{where } \xi \text{ is fresh within } \sigma \text{ and } \mathcal{H}$$

$$\langle [\texttt{malloc}\ x.sel]^\ell, \sigma, \mathcal{H} \rangle \ \rightarrow\ \langle \sigma, \mathcal{H}[(\sigma(x), sel) \mapsto \xi] \rangle$$
$$\text{where } \xi \text{ is fresh within } \sigma \text{ and } \mathcal{H} \text{ and } \sigma(x) \in \mathbf{Loc}$$

Statements

We add rules for the `malloc` statement, to allow allocation of new cells.

$$\langle [\texttt{malloc}\ x]^\ell, \sigma, \mathcal{H} \rangle \ \rightarrow\ \langle \sigma[x \mapsto \xi], \mathcal{H} \rangle$$
$$\text{where } \xi \text{ is fresh within } \sigma \text{ and } \mathcal{H}$$

$$\langle [\texttt{malloc}\ x.sel]^\ell, \sigma, \mathcal{H} \rangle \ \rightarrow\ \langle \sigma, \mathcal{H}[(\sigma(x), sel) \mapsto \xi] \rangle$$
$$\text{where } \xi \text{ is fresh within } \sigma \text{ and } \mathcal{H} \text{ and } \sigma(x) \in \mathbf{Loc}$$

Why we need shape graphs?

Obviously the heap can grow arbitrarily large, but we want a way

- to work with a **finite representation**,
- to combine the location of the semantics in a finite number of **abstract locations**.

Shape Graphs

We introduce the **shape graphs**, an abstract representation for heap and state composed by:

- S abstract state,
- H abstract heap,
- is sharing informations.

The is component allow us to recover the imprecision due to combining a set of location into an abstract location.

Why we need shape graphs?

Obviously the heap can grow arbitrarily large, but we want a way

- to work with a **finite representation**,
- to combine the location of the semantics in a finite number of **abstract locations**.

Shape Graphs

We introduce the **shape graphs**, an abstract representation for heap and state composed by:

S abstract state,

H abstract heap,

is sharing informations.

The is component allow us to recover the imprecision due to combining a set of location into an abstract location.

Why we need shape graphs?

Obviously the heap can grow arbitrarily large, but we want a way

- to work with a **finite representation**,
- to combine the location of the semantics in a finite number of **abstract locations**.

Shape Graphs

We introduce the **shape graphs**, an abstract representation for heap and state composed by:

S abstract state,

H abstract heap,

is sharing informations.

The is component allow us to recover the imprecision due to combining a set of location into an abstract location.

How do we proceed

First of all we will define what are

- abstract location,
- abstract state,
- abstract heap,
- sharing information.

Then we will present how to go from a couple $(\sigma, \mathcal{H})$ to a shape graph (S, H, is).

We will do it by introducing **Five Invariants**.

How do we proceed

First of all we will define what are

- abstract location,
- abstract state,
- abstract heap,
- sharing information.

Then we will present how to go from a couple $(\sigma, \mathcal{H})$ to a shape graph $(\mathsf{S}, \mathsf{H}, \mathsf{is})$.

We will do it by introducing **Five Invariants**.

Abstract Location

We define an abstract location such as:

$$\mathbf{ALoc} = \{n_X \mid X \subseteq \mathbf{Var}_*\}$$

The idea is that if $x \in X$ then the abstract location n_X will represent the location $\sigma(x)$.

We introduce even the *abstract summary location* $n_\emptyset$ that will represent all the location that we can't reach directly from σ.

Abstract Location

We define an abstract location such as:

$$\mathbf{ALoc} = \{n_X \mid X \subseteq \mathbf{Var}_*\}$$

The idea is that if $x \in X$ then the abstract location n_X will represent the location $\sigma(x)$.

We introduce even the *abstract summary location* $n_\emptyset$ that will represent all the location that we can't reach directly from σ.

Abstract Location

Abstract location represent **disjoint sets** of locations.

If we consider two different abstract locations n_X and n_Y, they could be the same ($X = Y$) or the are disjoint ($X \cap Y = \emptyset$). It can be easily proved, considering $X \neq Y$ and taking a $z \in X \cap Y$.

Invariant 1

If two abstract location n_X and n_Y occur in the same shape graph the either $X = Y$ or $X \cap Y = \emptyset$.

Abstract Location

Abstract location represent **disjoint sets** of locations.

If we consider two different abstract locations n_X and n_Y, they could be the same ($X = Y$) or the are disjoint ($X \cap Y = \emptyset$). It can be easily proved, considering $X \neq Y$ and taking a $z \in X \cap Y$.

Invariant 1

If two abstract location n_X and n_Y occur in the same shape graph the either $X = Y$ or $X \cap Y = \emptyset$.

Abstract State

The abstract state S is used to map variables to abstract locations

$$\mathsf{S} \in \mathbf{AState} = \mathcal{P}(\mathbf{Var}_* \times \mathbf{ALoc})$$

We shall ensure that

Invariant 2

If x is mapped to n_X by the abstract state then $x \in X$

From Invariant 1 follows that it will be **at most one** abstract location in the shape graph for each variable in the state.

Abstract State

The abstract state S is used to map variables to abstract locations

$$\mathsf{S} \in \mathbf{AState} = \mathcal{P}(\mathbf{Var}_* \times \mathbf{ALoc})$$

We shall ensure that

Invariant 2

If x is mapped to n_X by the abstract state then $x \in X$

From Invariant 1 follows that it will be **at most one** abstract location in the shape graph for each variable in the state.

Abstract Heaps

The abstract state H is used to specify links within abstract locations.

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

The links are specified by triples such as (n_X, sel, n_Y). The idea is that if $\mathcal{H}(\xi_1, sel) = \xi_2$ and ξ_1 and ξ_2 are represented by n_X and n_Y respectively, then $(n_X, sel, n_Y) \in \mathsf{H}$.

Abstract Heaps

The abstract state H is used to specify links within abstract locations.

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

The links are specified by triples such as (n_X, sel, n_Y). The idea is that if $\mathcal{H}(\xi_1, sel) = \xi_2$ and ξ_1 and ξ_2 are represented by n_X and n_Y respectively, then $(n_X, sel, n_Y) \in \mathsf{H}$.

Abstract Heaps

Please note that in heap $\mathcal{H}$ there will be **at most one** location ξ_2 such that $\mathcal{H}(\xi_1, sel) = \xi_2$.

This is **not completely true** in abstract heaps: consider the location $n_\emptyset$, it will represent several locations pointing to several locations (all the not reaching locations).

We shall ensure that

Invariant 3

Whenever (n_V, sel, n_W) and $(n_V, sel, n_{W'})$ are in the abstract heap, then either $V = \emptyset$ or $W = W'$.

Abstract Heaps

Please note that in heap $\mathcal{H}$ there will be **at most one** location ξ_2 such that $\mathcal{H}(\xi_1, sel) = \xi_2$.

This is **not completely true** in abstract heaps: consider the location $n_\emptyset$, it will represent several locations pointing to several locations (all the not reaching locations).

We shall ensure that

Invariant 3

Whenever (n_V, sel, n_W) and $(n_V, sel, n_{W'})$ are in the abstract heap, then either $V = \emptyset$ or $W = W'$.

Abstract Heaps

Please note that in heap $\mathcal{H}$ there will be **at most one** location ξ_2 such that $\mathcal{H}(\xi_1, sel) = \xi_2$.

This is **not completely true** in abstract heaps: consider the location $n_\emptyset$, it will represent several locations pointing to several locations (all the not reaching locations).

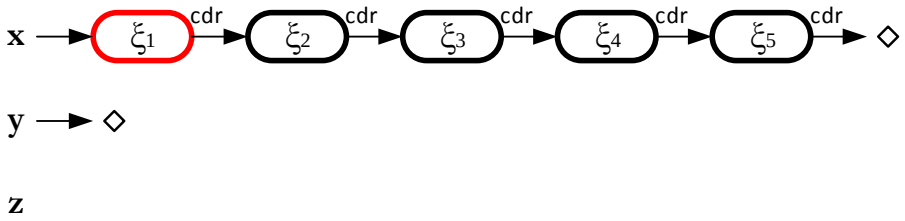
We shall ensure that

Invariant 3

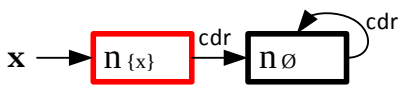
Whenever (n_V, sel, n_W) and $(n_V, sel, n_{W'})$ are in the abstract heap, then either $V = \emptyset$ or $W = W'$.

A little Example

$$[\mathtt{y} := \mathtt{nil}]^1;$$
$$\mathtt{while}\ [\mathtt{not\ is\text{-}nil(x)}]^2\ \mathtt{do}$$
$$([\mathtt{z} := \mathtt{y}]^3; [\mathtt{y} := \mathtt{x}]^4; [\mathtt{x} := \mathtt{x.cdr}]^5; [\mathtt{y.cdr} := \mathtt{z}]^6);$$
$$[\mathtt{z} := \mathtt{nil}]^7$$



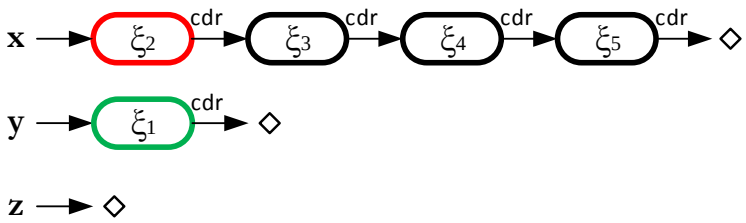
Heap



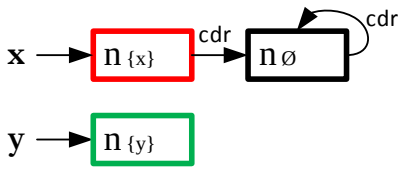
Shape Graph

A little Example

$$[\mathtt{y} := \mathtt{nil}]^1;$$
$$\mathtt{while}\ [\mathtt{not\ is\text{-}nil(x)}]^2\ \mathtt{do}$$
$$([\mathtt{z} := \mathtt{y}]^3; [\mathtt{y} := \mathtt{x}]^4; [\mathtt{x} := \mathtt{x.cdr}]^5; [\mathtt{y.cdr} := \mathtt{z}]^6);$$
$$[\mathtt{z} := \mathtt{nil}]^7$$



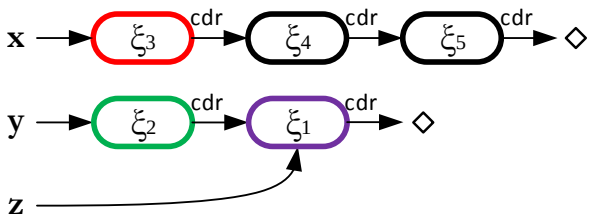
Heap



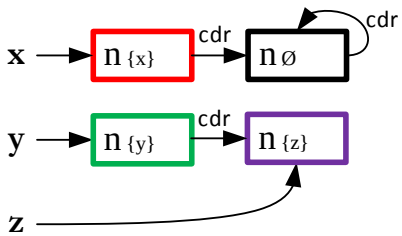
Shape Graph

A little Example

$$[\mathtt{y} := \mathtt{nil}]^1;$$
$$\mathtt{while}\ [\mathtt{not\ is\text{-}nil(x)}]^2\ \mathtt{do}$$
$$([\mathtt{z} := \mathtt{y}]^3; [\mathtt{y} := \mathtt{x}]^4; [\mathtt{x} := \mathtt{x.cdr}]^5; [\mathtt{y.cdr} := \mathtt{z}]^6);$$
$$[\mathtt{z} := \mathtt{nil}]^7$$



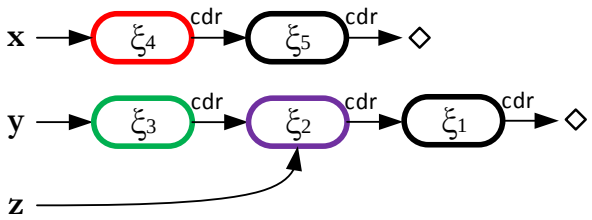
Heap



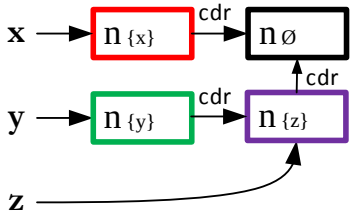
Shape Graph

A little Example

$$[\mathtt{y} := \mathtt{nil}]^1;$$
$$\mathtt{while}\ [\mathtt{not\ is\text{-}nil(x)}]^2\ \mathtt{do}$$
$$([\mathtt{z} := \mathtt{y}]^3; [\mathtt{y} := \mathtt{x}]^4; [\mathtt{x} := \mathtt{x.cdr}]^5; [\mathtt{y.cdr} := \mathtt{z}]^6);$$
$$[\mathtt{z} := \mathtt{nil}]^7$$



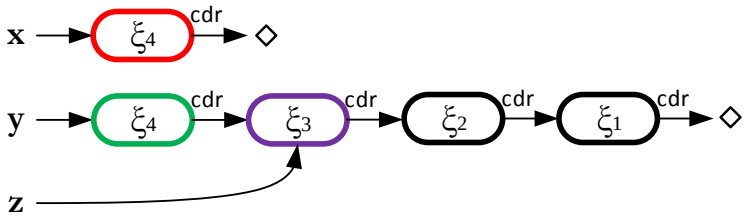
Heap



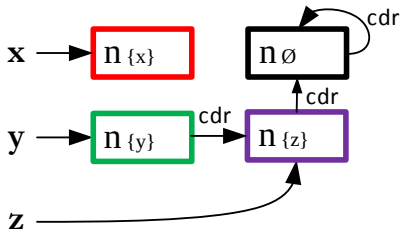
Shape Graph

A little Example

$$[\mathtt{y} := \mathtt{nil}]^1;$$
$$\mathtt{while}\ [\mathtt{not\ is\text{-}nil(x)}]^2\ \mathtt{do}$$
$$([\mathtt{z} := \mathtt{y}]^3; [\mathtt{y} := \mathtt{x}]^4; [\mathtt{x} := \mathtt{x.cdr}]^5; [\mathtt{y.cdr} := \mathtt{z}]^6);$$
$$[\mathtt{z} := \mathtt{nil}]^7$$



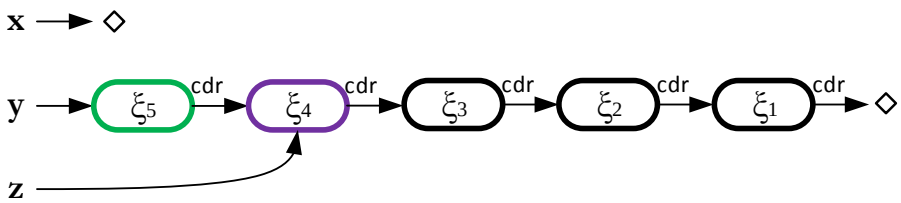
Heap



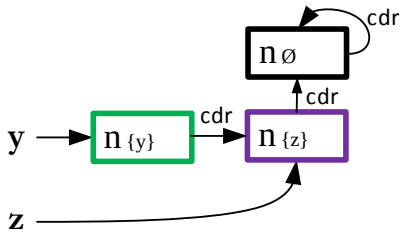
Shape Graph

A little Example

$$[\mathtt{y} := \mathtt{nil}]^1;$$
$$\mathtt{while}\ [\mathtt{not\ is\text{-}nil(x)}]^2\ \mathtt{do}$$
$$([\mathtt{z} := \mathtt{y}]^3; [\mathtt{y} := \mathtt{x}]^4; [\mathtt{x} := \mathtt{x.cdr}]^5; [\mathtt{y.cdr} := \mathtt{z}]^6);$$
$$[\mathtt{z} := \mathtt{nil}]^7$$



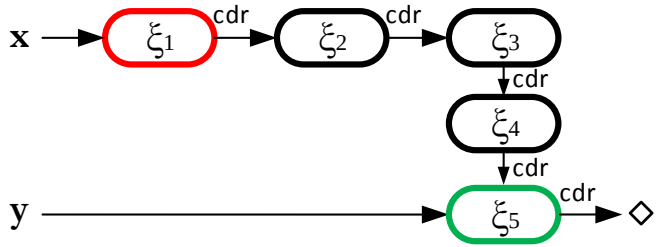
Heap



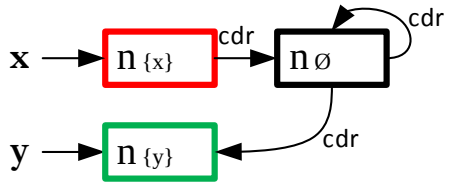
Shape Graph

Sharing Informations

Heap Representation



Shape Graph

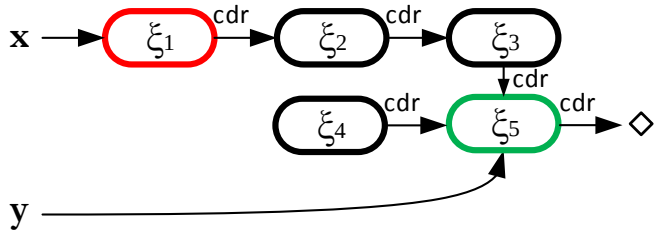


We want to represent a set is of locations that are **shared** due to pointers in the heap.
The idea is that if an abstract location n_X will be included in is if it is a target of **two or more** pointer in the heap.

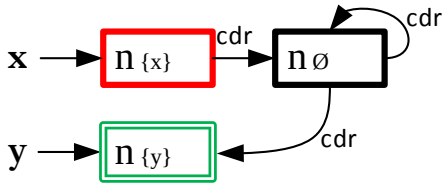
We must assure that the sharing information inside is is **consistent** with the information inside the abstract heap H.

Sharing Informations

Heap Representation



Shape Graph

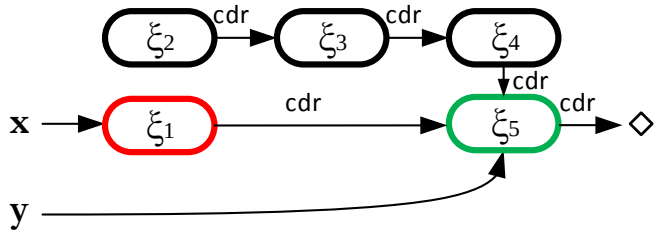


We want to represent a set is of locations that are **shared** due to pointers in the heap.
The idea is that if an abstract location n_X will be included in is if it is a target of **two or more** pointer in the heap.

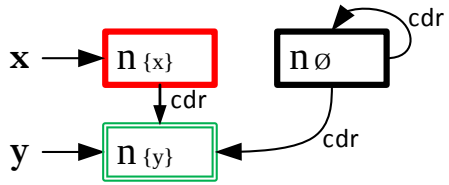
We must assure that the sharing information inside is is **consistent** with the information inside the abstract heap H.

Sharing Informations

Heap Representation



Shape Graph



We want to represent a set is of locations that are **shared** due to pointers in the heap.
The idea is that if an abstract location n_X will be included in is if it is a target of **two or more** pointer in the heap.

We must assure that the sharing information inside is is **consistent** with the information inside the abstract heap H.

Sharing Informations

So we impose two different invariants

Invariant 4

If $n_X \in$ is then either

(a) $(n_\emptyset, sel, n_X)$ is in the abstract heap for some sel, or

(b) there exists two distinct triples (n_V, sel_1, n_X) and (n_W, sel_2, n_X) into the abstract heap (that is either $sel_1 \neq sel_2$ or $V \neq W$).

Invariant 4 imposes that the information in the sharing component is is reflected into the abstract heap.

Case 4(a) takes care of cases where there are links between $n_\emptyset$ and n_X in the heap; Case 4(b) takes care of cases where there are link between different pointers (distinct source or selector) to n_X.

Sharing Informations

So we impose two different invariants

Invariant 4

If $n_X \in$ is then either

(a) $(n_\emptyset, sel, n_X)$ is in the abstract heap for some sel, or

(b) there exists two distinct triples (n_V, sel_1, n_X) and (n_W, sel_2, n_X) into the abstract heap (that is either $sel_1 \neq sel_2$ or $V \neq W$).

Invariant 4 imposes that the information in the sharing component is is reflected into the abstract heap.

Case 4(a) takes care of cases where there are links between $n_\emptyset$ and n_X in the heap; Case 4(b) takes care of cases where there are link between different pointers (distinct source or selector) to n_X.

Sharing Informations

So we impose two different invariants

Invariant 4

If $n_X \in$ is then either

(a) $(n_\emptyset, sel, n_X)$ is in the abstract heap for some sel, or

(b) there exists two distinct triples (n_V, sel_1, n_X) and (n_W, sel_2, n_X) into the abstract heap (that is either $sel_1 \neq sel_2$ or $V \neq W$).

Invariant 4 imposes that the information in the sharing component is is reflected into the abstract heap.

Case 4(a) takes care of cases where there are links between $n_\emptyset$ and n_X in the heap; Case 4(b) takes care of cases where there are link between different pointers (distinct source or selector) to n_X.

Sharing Informations

Invariant 5 imposes that the information in the abstract heap H is reflected into the sharing component.

Invariant 5

Whenever there are two distinct triples (n_V, sel_1, n_X) and (n_W, sel_2, n_X) in the abstract heap and $n_X \neq n_\emptyset$ then $n_X \in$ is.

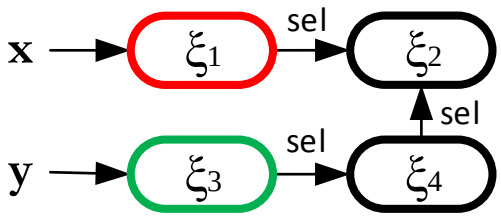
Note that Invariant 5 is the "inverse" of 4(b). We don't have an "inverse" of case 4(a), the presence of a pointer from $n_\emptyset$ to n_X in H does not give information concerning sharing.

Sharing Informations and $n_\emptyset$

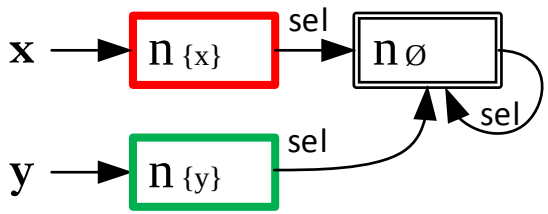
Taking in consideration the abstract summary location $n_\emptyset$:

If $n_\emptyset \in$ is then it will be **at least one** location represented by $n_\emptyset$ that is a target by two or more pointers.

Heap



Shape Graph

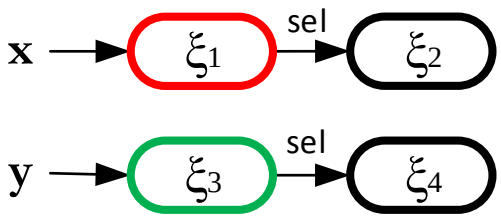


Sharing Informations and $n_\emptyset$

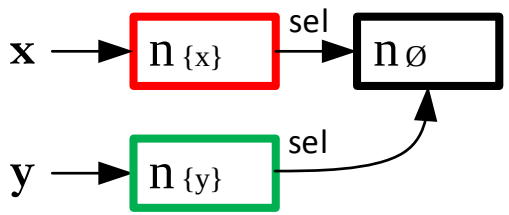
Taking in consideration the abstract summary location $n_\emptyset$:

If $n_\emptyset \notin$ is then **all** the location represented by $n_\emptyset$ will be target by **at most one** pointer (Does not exist a location with two incoming edge).

Heap



Shape Graph



Summarize

To summarize a shape graph is composed by:

$$\begin{array}{lcl} \mathsf{S} \in \mathbf{AState} & = & \mathcal{P}(\mathbf{Var}_* \times \mathbf{ALoc}) \\ \mathsf{H} \in \mathbf{AHeap} & = & \mathcal{P}(\mathbf{ALoc} \times \mathbf{Sel} \times \mathbf{ALoc}) \\ \mathsf{is} \in \mathbf{IsShared} & = & \mathcal{P}(\mathbf{ALoc}) \end{array}$$

where

$$\mathbf{ALoc} = \{n_Z \mid Z \subseteq \mathbf{Var}_*\}$$

Compatible Shape Graph

We say that a shape graph is **compatible** if it fulfil the 5 invariants:

1. $\forall n_V, n_W \in ALoc(\mathsf{S}) \cup ALoc(\mathsf{H}) \cup \mathsf{is} : (V = W) \vee (V \cap W = \emptyset)$
2. $\forall (x, n_X) \in \mathsf{S} : x \in X$
3. $\forall (n_V, sel, n_W), (n_V, sel, n_{W'}) \in \mathsf{H} : (V = \emptyset) \vee (W = W')$
4. $\forall n_X \in \mathsf{is} : \quad (\exists sel : (n_\emptyset, sel, n_X) \in \mathsf{H}) \vee (\exists (n_V, sel_1, n_X), (n_W, sel_2, n_X) \in \mathsf{H} : sel_1 \neq sel_2 \vee V \neq W)$
5. $\forall (n_V, sel_1, n_X), (n_W, sel_2, n_X) \in \mathsf{H} : ((sel_1 \neq sel_2 \vee V \neq W) \wedge X \neq \emptyset) \Rightarrow n_X \in \mathsf{is}$

Compatible Shape Graph

The sets of compatible shape graphs is denoted by

$$\mathbf{SG} = \{(\text{S,H,is}) \mid (\text{S,H,is}) \text{ is compatible}\}$$

Our analysis will work on *sets* of compatible shape graphs (elements of $\mathcal{P}(\mathbf{SG})$).

$\mathcal{P}(\mathbf{SG})$ is trivially a **complete lattice**, with $\sqcup$ being $\cup$ and $\sqsubseteq$ being $\subseteq$.

$\mathcal{P}(\mathbf{SG})$ is obviously **finite** because $\mathbf{SG} \subseteq \mathbf{AState} \times \mathbf{AHeap} \times \mathbf{IsShared}$, and $\mathbf{AState}$, $\mathbf{AHeap}$ and $\mathbf{IsShared}$ are **finite**.

Compatible Shape Graph

The sets of compatible shape graphs is denoted by

$$\mathbf{SG} = \{(\text{S,H,is}) \mid (\text{S,H,is}) \text{ is compatible}\}$$

Our analysis will work on *sets* of compatible shape graphs (elements of $\mathcal{P}(\mathbf{SG})$).

$\mathcal{P}(\mathbf{SG})$ is trivially a **complete lattice**, with $\sqcup$ being $\cup$ and $\sqsubseteq$ being $\subseteq$.

$\mathcal{P}(\mathbf{SG})$ is obviously **finite** because $\mathbf{SG} \subseteq \mathbf{AState} \times \mathbf{AHeap} \times \mathbf{IsShared}$, and $\mathbf{AState}$, $\mathbf{AHeap}$ and $\mathbf{IsShared}$ are **finite**.

Compatible Shape Graph

The sets of compatible shape graphs is denoted by

$$\mathbf{SG} = \{(\text{S,H,is}) \mid (\text{S,H,is}) \text{ is compatible}\}$$

Our analysis will work on *sets* of compatible shape graphs (elements of $\mathcal{P}(\mathbf{SG})$).

$\mathcal{P}(\mathbf{SG})$ is trivially a **complete lattice**, with $\sqcup$ being $\cup$ and $\sqsubseteq$ being $\subseteq$.

$\mathcal{P}(\mathbf{SG})$ is obviously **finite** because $\mathbf{SG} \subseteq \mathbf{AState} \times \mathbf{AHeap} \times \mathbf{IsShared}$, and $\mathbf{AState}$, $\mathbf{AHeap}$ and $\mathbf{IsShared}$ are **finite**.

Compatible Shape Graph

The sets of compatible shape graphs is denoted by

$$\mathbf{SG} = \{\text{(S,H,is)} \mid \text{(S,H,is) is compatible}\}$$

Our analysis will work on *sets* of compatible shape graphs (elements of $\mathcal{P}(\mathbf{SG})$).

$\mathcal{P}(\mathbf{SG})$ is trivially a **complete lattice**, with $\sqcup$ being $\cup$ and $\sqsubseteq$ being $\subseteq$.

$\mathcal{P}(\mathbf{SG})$ is obviously **finite** because
$\mathbf{SG} \subseteq \mathbf{AState} \times \mathbf{AHeap} \times \mathbf{IsShared}$, and $\mathbf{AState}$, $\mathbf{AHeap}$ and $\mathbf{IsShared}$ are **finite**.

The Analysis

We introduce the analysis $Shape$ as an instance of a **Monotone Framework**. For every labeled program S_* we produce a sets of equations of the form.

$$Shape_\circ(\ell) = \begin{cases} \iota & \text{if} \ell = init(S_*) \\ \bigcup\{Shape_\bullet(\ell') \mid (\ell', \ell) \in flow(S_*)\} & \text{otherwise} \end{cases}$$

$$Shape_\bullet(\ell) = f_\ell^{\mathsf{SA}}(Shape_\circ(\ell))$$

Where $\iota \in \mathcal{P}(\mathbf{SG})$ is the extremal value at entry of S_* and f_ℓ^{SA} is the transfer function.

The Analysis

We introduce the analysis $Shape$ as an instance of a **Monotone Framework**. For every labeled program S_* we produce a sets of equations of the form.

$$Shape_\circ(\ell) = \begin{cases} \iota & \text{if} \ell = init(S_*) \\ \bigcup\{Shape_\bullet(\ell') \mid (\ell', \ell) \in flow(S_*)\} & \text{otherwise} \end{cases}$$

$$Shape_\bullet(\ell) = f_\ell^{\mathsf{SA}}(Shape_\circ(\ell))$$

Where $\iota \in \mathcal{P}(\mathbf{SG})$ is the extremal value at entry of S_* and f_ℓ^{SA} is the transfer function.

The Analysis

We introduce the analysis $Shape$ as an instance of a **Monotone Framework**. For every labeled program S_* we produce a sets of equations of the form.

$$Shape_\circ(\ell) = \begin{cases} \iota & \text{if} \ell = init(S_*) \\ \bigcup\{Shape_\bullet(\ell') \mid (\ell', \ell) \in flow(S_*)\} & \text{otherwise} \end{cases}$$

$$Shape_\bullet(\ell) = f_\ell^{\mathsf{SA}}(Shape_\circ(\ell))$$

Where $\iota \in \mathcal{P}(\mathbf{SG})$ is the extremal value at entry of S_* and f_ℓ^{SA} is the transfer function.

The Analysis

The analysis *Shape* is a **forward analysis**, because it's defined using the set $\mathit{flow}(S_*)$.

It's also a **may analysis** since we are using the $\bigcup$ operator for combining results gathered from the $\mathit{flow}(S_*)$.

The **transfer function** $f_\ell^{\mathsf{SA}} : \mathcal{P}(\mathbf{SG}) \to \mathcal{P}(\mathbf{SG})$ has the form:

$$f_\ell^{\mathsf{SA}}(SG) = \bigcup \{ \phi_\ell^{\mathsf{SA}}((\mathsf{S}, \mathsf{H}, \mathsf{is})) \mid (\mathsf{S}, \mathsf{H}, \mathsf{is}) \in SG \}$$

The function ϕ_ℓ^{SA} has to be developed right now.

The Analysis

The analysis *Shape* is a **forward analysis**, because it's defined using the set $\mathit{flow}(S_*)$.

It's also a **may analysis** since we are using the $\bigcup$ operator for combining results gathered from the $\mathit{flow}(S_*)$.

The **transfer function** $f_\ell^{\mathsf{SA}} : \mathcal{P}(\mathbf{SG}) \to \mathcal{P}(\mathbf{SG})$ has the form:

$$f_\ell^{\mathsf{SA}}(SG) = \bigcup \{ \phi_\ell^{\mathsf{SA}}((\mathsf{S}, \mathsf{H}, \mathsf{is})) \mid (\mathsf{S}, \mathsf{H}, \mathsf{is}) \in SG \}$$

The function ϕ_ℓ^{SA} has to be developed right now.

The Analysis

The analysis *Shape* is a **forward analysis**, because it's defined using the set $\mathit{flow}(S_*)$.

It's also a **may analysis** since we are using the $\bigcup$ operator for combining results gathered from the $\mathit{flow}(S_*)$.

The **transfer function** $f_\ell^{\mathsf{SA}} : \mathcal{P}(\mathbf{SG}) \to \mathcal{P}(\mathbf{SG})$ has the form:

$$f_\ell^{\mathsf{SA}}(SG) = \bigcup \{ \phi_\ell^{\mathsf{SA}}((\mathsf{S}, \mathsf{H}, \mathsf{is})) \mid (\mathsf{S}, \mathsf{H}, \mathsf{is}) \in SG \}$$

The function ϕ_ℓ^{SA} has to be developed right now.

The function ϕ_ℓ^{SA}

The function $\phi_\ell^{\mathsf{SA}} : \mathbf{SG} \to \mathcal{P}(\mathbf{SG})$ specifies how a **single** graph in $Shape_\circ(\ell)$ is transformed into a **set** of graph in $Shape_\bullet(\ell)$

We present the ϕ_ℓ^{SA} function for all the different kind of statements.

The function ϕ_ℓ^{SA}

The function $\phi_\ell^{\mathsf{SA}} : \mathbf{SG} \to \mathcal{P}(\mathbf{SG})$ specifies how a **single** graph in $Shape_\circ(\ell)$ is transformed into a **set** of graph in $Shape_\bullet(\ell)$

We present the ϕ_ℓ^{SA} function for all the different kind of statements.

$[b]^\ell$ and $[\texttt{skip}]^\ell$

$[b]^\ell$ and $[\texttt{skip}]^\ell$

Since $[b]^\ell$ and $[\texttt{skip}]^\ell$ does **not modify** the heap, the ϕ_ℓ^{SA} function is just the **identity** functions, remember that we are interested in the heap shape.

$$\phi_\ell^{\mathsf{SA}}((\mathsf{S},\mathsf{H},\mathsf{is})) = \{(\mathsf{S},\mathsf{H},\mathsf{is})\}$$

$[b]^\ell$ and $[\texttt{skip}]^\ell$

Since $[b]^\ell$ and $[\texttt{skip}]^\ell$ does **not modify** the heap, the ϕ_ℓ^{SA} function is just the **identity** functions, remember that we are interested in the heap shape.

$$\phi_\ell^{\mathsf{SA}}((\mathsf{S},\mathsf{H},\mathsf{is})) = \{(\mathsf{S},\mathsf{H},\mathsf{is})\}$$

$$[x := a]^{\ell}$$

$[x := a]^\ell$

We consider $[x := a]^\ell$ in the case where a is of the form n, $a_1\ op_a\ a_2$ or `nil`.

We must **remove the binding** of x and **rename** all the abstract location that contains x.

$$k_x(n_Z) = n_{Z\setminus\{x\}}$$

So we take

$$\phi^{\mathsf{SA}}_\ell((\mathsf{S},\mathsf{H},\mathsf{is})) = \{kill_x(\mathsf{S},\mathsf{H},\mathsf{is})\}$$

$[x := a]^\ell$

We consider $[x := a]^\ell$ in the case where a is of the form n, $a_1\ op_a\ a_2$ or `nil`.

We must **remove the binding** of x and **rename** all the abstract location that contains x.

$$k_x(n_Z) = n_{Z\setminus\{x\}}$$

So we take

$$\phi^{\mathsf{SA}}_\ell((\mathsf{S},\mathsf{H},\mathsf{is})) = \{kill_x(\mathsf{S},\mathsf{H},\mathsf{is})\}$$

$[x := a]^\ell$

We consider $[x := a]^\ell$ in the case where a is of the form n, $a_1\ op_a\ a_2$ or `nil`.

We must **remove the binding** of x and **rename** all the abstract location that contains x.

$$k_x(n_Z) = n_{Z \setminus \{x\}}$$

So we take

$$\phi^{\mathsf{SA}}_\ell((\mathsf{S}, \mathsf{H}, \mathsf{is})) = \{kill_x(\mathsf{S}, \mathsf{H}, \mathsf{is})\}$$

$[x := a]^\ell$

Where $kill_x(\mathsf{S}, \mathsf{H}, \mathsf{is}) = (\mathsf{S}', \mathsf{H}', \mathsf{is}')$ is given by

$$\begin{aligned}
\mathsf{S}' &= \{(z, k_x(n_Z)) \mid (z, n_Z) \in \mathsf{S} \wedge z \neq x\} \\
\mathsf{H}' &= \{(k_x(n_V), sel, k_x(n_W)) \mid (n_V, sel, n_W) \in \mathsf{H}\} \\
\mathsf{is}' &= \{(k_x(n_X)) \mid n_X \in \mathsf{is}\}
\end{aligned}$$

It's easy to prove that if $(\mathsf{S}, \mathsf{H}, \mathsf{is})$ is compatible, even $(\mathsf{S}', \mathsf{H}', \mathsf{is}')$ is compatible.

$[x := a]^\ell$

Where $kill_x(\mathsf{S}, \mathsf{H}, \mathsf{is}) = (\mathsf{S}', \mathsf{H}', \mathsf{is}')$ is given by

$$\begin{aligned}
\mathsf{S}' &= \{(z, k_x(n_Z)) \mid (z, n_Z) \in \mathsf{S} \wedge z \neq x\} \\
\mathsf{H}' &= \{(k_x(n_V), sel, k_x(n_W)) \mid (n_V, sel, n_W) \in \mathsf{H}\} \\
\mathsf{is}' &= \{(k_x(n_X)) \mid n_X \in \mathsf{is}\}
\end{aligned}$$

It's easy to prove that if $(\mathsf{S}, \mathsf{H}, \mathsf{is})$ is compatible, even $(\mathsf{S}', \mathsf{H}', \mathsf{is}')$ is compatible.

If $(x, n_{\{x\}})$ is in **S**?

If $(x, n_{\{x\}})$ is in **S** then the two abstract location $n_{\{x\}}$ and $n_{\emptyset}$ will be **merged**.

From this we can deduce that $n_{\emptyset}$ will be **unshared** if **both** $n_{\{x\}}$ and $n_{\emptyset}$ was **unshared**.

Garbage Collection

Please note that our analysis does not provide a **garbage collector**, so if $n_{\{x\}}$ has no heap pointer, it is **unreachable**. Consider that there will be a pointer from $n_{\emptyset}$ to the location that $n_{\{x\}}$ might point to.

If $(x, n_{\{x\}})$ is in **S**?

If $(x, n_{\{x\}})$ is in **S** then the two abstract location $n_{\{x\}}$ and $n_{\emptyset}$ will be **merged**.

From this we can deduce that $n_{\emptyset}$ will be **unshared** if **both** $n_{\{x\}}$ and $n_{\emptyset}$ was **unshared**.

Garbage Collection

Please note that our analysis does not provide a **garbage collector**, so if $n_{\{x\}}$ has no heap pointer, it is **unreachable**. Consider that there will be a pointer from $n_{\emptyset}$ to the location that $n_{\{x\}}$ might point to.

If $(x, n_{\{x\}})$ is in **S**?

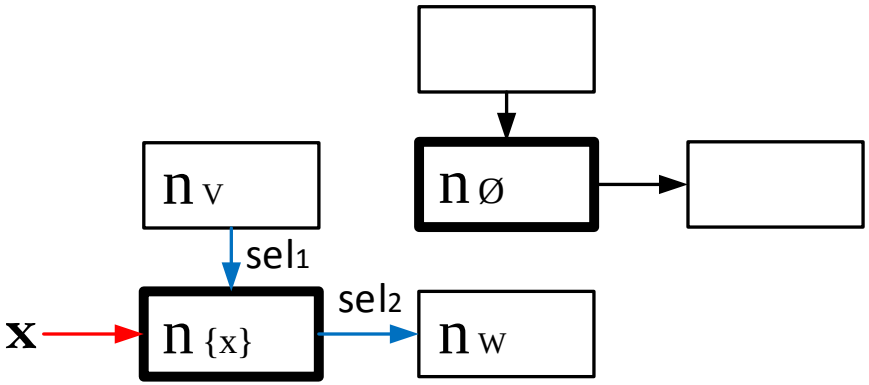
If $(x, n_{\{x\}})$ is in **S** then the two abstract location $n_{\{x\}}$ and $n_{\emptyset}$ will be **merged**.

From this we can deduce that $n_{\emptyset}$ will be **unshared** if **both** $n_{\{x\}}$ and $n_{\emptyset}$ was **unshared**.

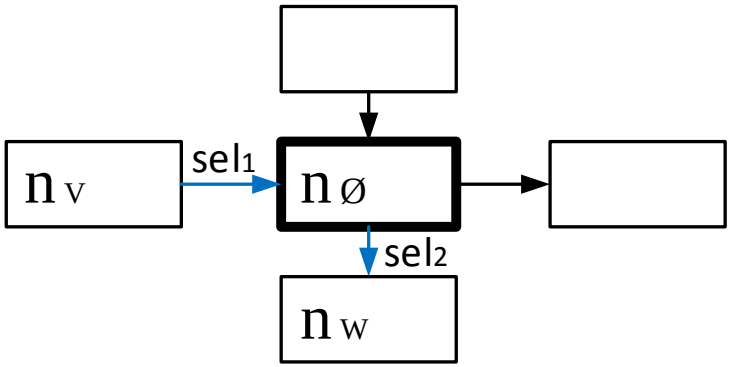
Garbage Collection

Please note that our analysis does not provide a **garbage collector**, so if $n_{\{x\}}$ has no heap pointer, it is **unreachable**. Consider that there will be a pointer from $n_{\emptyset}$ to the location that $n_{\{x\}}$ might point to.

$[x := a]^\ell$



(S, H, is)



(S', H', is')

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

$[x := y]^\ell$

if $x = y$ the f_ℓ^{SA} is just the identity function.

if $x \neq y$ then we execute the following steps:

1. We remove the old binding for x with $kill_x$,
2. We update the abstract location that contains y adding $\{x\}$ to the variable sets.

This can be done with

$$g_x^y(n_Z) = \begin{cases} n_{Z \cup \{x\}} & \text{if } y \in Z \\ n_Z & \text{otherwise} \end{cases}$$

$[x := y]^\ell$

if $x = y$ the f_ℓ^{SA} is just the identity function.

if $x \neq y$ then we execute the following steps:

1. We remove the old binding for x with $kill_x$,
2. We update the abstract location that contains y adding $\{x\}$ to the variable sets.

This can be done with

$$g_x^y(n_Z) = \begin{cases} n_{Z \cup \{x\}} & \text{if } y \in Z \\ n_Z & \text{otherwise} \end{cases}$$

$[x := y]^\ell$

if $x = y$ the f_ℓ^{SA} is just the identity function.

if $x \neq y$ then we execute the following steps:

1. We remove the old binding for x with $kill_x$,
2. We update the abstract location that contains y adding $\{x\}$ to the variable sets.

This can be done with

$$g_x^y(n_Z) = \begin{cases} n_{Z \cup \{x\}} & \text{if } y \in Z \\ n_Z & \text{otherwise} \end{cases}$$

$[x := y]^\ell$

So we take

$$\phi^{\mathsf{SA}}_\ell((\mathsf{S},\mathsf{H},\mathsf{is})) = \{(\mathsf{S}'',\mathsf{H}'',\mathsf{is}'')\}$$

Where $(\mathsf{S}',\mathsf{H}',\mathsf{is}') = kill_x((\mathsf{S},\mathsf{H},\mathsf{is}))$ and

$$\begin{aligned}
\mathsf{S}'' &= \{(z, g^y_x(n_Z)) \mid (z, n_Z) \in \mathsf{S}'\} \\
&\quad \cup \{(x, g^y_x(n_Y)) \mid (y', n_Y) \in \mathsf{S}' \wedge y' = y\} \\
\mathsf{H}'' &= \{(g^y_x(n_V), sel, g^y_x(n_W)) \mid (n_V, sel, n_W) \in \mathsf{H}'\} \\
\mathsf{is}'' &= \{g^y_x(n_Z) \mid n_Z \in \mathsf{is}'\}
\end{aligned}$$

Again it's easy to prove that if $(\mathsf{S},\mathsf{H},\mathsf{is})$ is compatible, even $(\mathsf{S}'',\mathsf{H}'',\mathsf{is}'')$ is compatible.

$[x := y]^\ell$

So we take

$$\phi_\ell^{\mathsf{SA}}((\mathsf{S},\mathsf{H},\mathsf{is})) = \{(\mathsf{S}'',\mathsf{H}'',\mathsf{is}'')\}$$

Where $(\mathsf{S}',\mathsf{H}',\mathsf{is}') = kill_x((\mathsf{S},\mathsf{H},\mathsf{is}))$ and

$$\begin{aligned}
\mathsf{S}'' &= \{(z, g_x^y(n_Z)) \mid (z, n_Z) \in \mathsf{S}'\} \\
&\quad \cup \{(x, g_x^y(n_Y)) \mid (y', n_Y) \in \mathsf{S}' \wedge y' = y\} \\
\mathsf{H}'' &= \{(g_x^y(n_V), sel, g_x^y(n_W)) \mid (n_V, sel, n_W) \in \mathsf{H}'\} \\
\mathsf{is}'' &= \{g_x^y(n_Z) \mid n_Z \in \mathsf{is}'\}
\end{aligned}$$

Again it's easy to prove that if $(\mathsf{S},\mathsf{H},\mathsf{is})$ is compatible, even $(\mathsf{S}'',\mathsf{H}'',\mathsf{is}'')$ is compatible.

$[x := y]^\ell$

So we take

$$\phi_\ell^{\mathsf{SA}}((\mathsf{S},\mathsf{H},\mathsf{is})) = \{(\mathsf{S}'',\mathsf{H}'',\mathsf{is}'')\}$$

Where $(\mathsf{S}',\mathsf{H}',\mathsf{is}') = kill_x((\mathsf{S},\mathsf{H},\mathsf{is}))$ and

$$\begin{aligned}
\mathsf{S}'' &= \{(z, g_x^y(n_Z)) \mid (z, n_Z) \in \mathsf{S}'\} \\
&\quad \cup \{(x, g_x^y(n_Y)) \mid (y', n_Y) \in \mathsf{S}' \wedge y' = y\} \\
\mathsf{H}'' &= \{(g_x^y(n_V), sel, g_x^y(n_W)) \mid (n_V, sel, n_W) \in \mathsf{H}'\} \\
\mathsf{is}'' &= \{g_x^y(n_Z) \mid n_Z \in \mathsf{is}'\}
\end{aligned}$$

Again it's easy to prove that if $(\mathsf{S},\mathsf{H},\mathsf{is})$ is compatible, even $(\mathsf{S}'',\mathsf{H}'',\mathsf{is}'')$ is compatible.

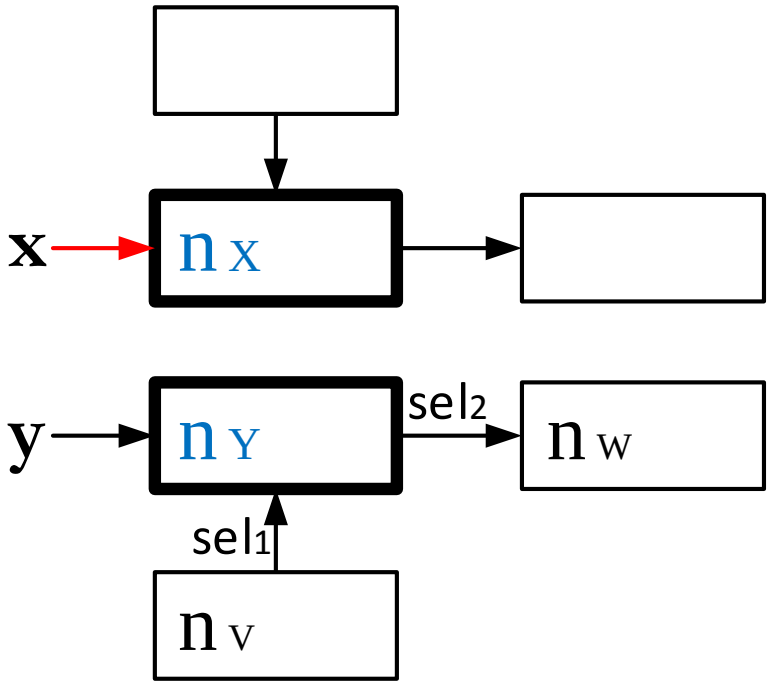
$[x := y]^\ell$

$$
\begin{aligned}
\mathsf{S}'' &= \{(z, g_x^y(n_Z)) \mid (z, n_Z) \in \mathsf{S}'\} \\
&\quad \cup \{(x, g_x^y(n_Y)) \mid (y', n_Y) \in \mathsf{S}' \wedge y' = y\} \\
\mathsf{H}'' &= \{(g_x^y(n_V), sel, g_x^y(n_W)) \mid (n_V, sel, n_W) \in \mathsf{H}'\} \\
\mathsf{is}'' &= \{g_x^y(n_Z) \mid n_Z \in \mathsf{is}'\}
\end{aligned}
$$

Please note that the clause in blue is the **adding** of the binding of x.
The sharing information of $n_{Y \cup \{x\}}$ in inherited from n_Y.

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

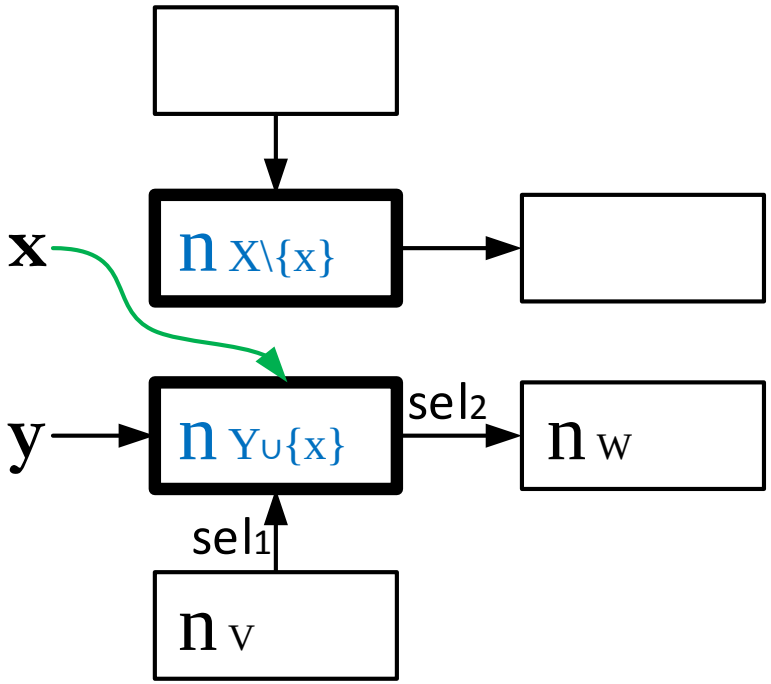
A little example when $x \neq y$.



(S, H, is)

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

A little example when $x \neq y$.



(S'', H'', is'')

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

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

If $x = y$ the assignment is equivalent to:

$$[t := y.sel]^{\ell_1};\ [x := t]^{\ell_2};\ [t := \mathtt{nil}]^{\ell_3};$$

Where t is a **fresh variable** and ℓ_1, ℓ_2 and ℓ_3 are **fresh labels**.

We can obtain the transfer function f^{SA}_{ℓ} as a **composition**:

$$f^{\mathsf{SA}}_{\ell} = f^{\mathsf{SA}}_{\ell_3} \circ f^{\mathsf{SA}}_{\ell_2} \circ f^{\mathsf{SA}}_{\ell_1}$$

So $f^{\mathsf{SA}}_{\ell_2}$ and $f^{\mathsf{SA}}_{\ell_3}$ can be computed using the same pattern as before. We concentrate on $f^{\mathsf{SA}}_{\ell_1}$ (or similar in the case when $x \neq y$).

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

If $x = y$ the assignment is equivalent to:

$$[t := y.sel]^{\ell_1};\ [x := t]^{\ell_2};\ [t := \mathtt{nil}]^{\ell_3};$$

Where t is a **fresh variable** and ℓ_1, ℓ_2 and ℓ_3 are **fresh labels**.

We can obtain the transfer function f^{SA}_ℓ as a **composition**:

$$f^{\mathsf{SA}}_\ell = f^{\mathsf{SA}}_{\ell_3} \circ f^{\mathsf{SA}}_{\ell_2} \circ f^{\mathsf{SA}}_{\ell_1}$$

So $f^{\mathsf{SA}}_{\ell_2}$ and $f^{\mathsf{SA}}_{\ell_3}$ can be computed using the same pattern as before. We concentrate on $f^{\mathsf{SA}}_{\ell_1}$ (or similar in the case when $x \neq y$).

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

First of all we remove the old binding for x:

$$(\mathsf{S}', \mathsf{H}', \mathsf{is}') = kill_x((\mathsf{S}, \mathsf{H}, \mathsf{is}))$$

Then we must rename the abstract location corresponding to $y.sel$ to include x into the variable set, then we must add the new binding for x. We can have 3 different cases:

1. There is no abstract location n_Y such that $(y, n_Y) \in \mathsf{S}'$, or it exist but there is no n_Z such that $(n_Y, sel, n_Z) \in \mathsf{H}'$.
2. There is an abstract location n_Y such that $(y, n_Y) \in \mathsf{S}'$ and there is a $n_U \neq n_\emptyset$ such that $(n_Y, sel, n_U) \in \mathsf{H}'$.
3. There is an abstract location n_Y such that $(y, n_Y) \in \mathsf{S}'$ and there is a triple such that $(n_Y, sel, n_\emptyset) \in \mathsf{H}'$.

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

First of all we remove the old binding for x:

$$(\mathsf{S}', \mathsf{H}', \mathsf{is}') = kill_x((\mathsf{S}, \mathsf{H}, \mathsf{is}))$$

Then we must rename the abstract location corresponding to $y.sel$ to include x into the variable set, then we must add the new binding for x. We can have 3 different cases:

1. There is no abstract location n_Y such that $(y, n_Y) \in \mathsf{S}'$, or it exist but there is no n_Z such that $(n_Y, sel, n_Z) \in \mathsf{H}'$.
2. There is an abstract location n_Y such that $(y, n_Y) \in \mathsf{S}'$ and there is a $n_U \neq n_\emptyset$ such that $(n_Y, sel, n_U) \in \mathsf{H}'$.
3. There is an abstract location n_Y such that $(y, n_Y) \in \mathsf{S}'$ and there is a triple such that $(n_Y, sel, n_\emptyset) \in \mathsf{H}'$.

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

First of all we remove the old binding for x:

$$(\mathsf{S}', \mathsf{H}', \mathsf{is}') = kill_x((\mathsf{S}, \mathsf{H}, \mathsf{is}))$$

Then we must rename the abstract location corresponding to $y.sel$ to include x into the variable set, then we must add the new binding for x. We can have 3 different cases:

1. There is no abstract location n_Y such that $(y, n_Y) \in \mathsf{S}'$, or it exist but there is no n_Z such that $(n_Y, sel, n_Z) \in \mathsf{H}'$.
2. There is an abstract location n_Y such that $(y, n_Y) \in \mathsf{S}'$ and there is a $n_U \neq n_\emptyset$ such that $(n_Y, sel, n_U) \in \mathsf{H}'$.
3. There is an abstract location n_Y such that $(y, n_Y) \in \mathsf{S}'$ and there is a triple such that $(n_Y, sel, n_\emptyset) \in \mathsf{H}'$.

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

First of all we remove the old binding for x:

$$(\mathsf{S}', \mathsf{H}', \mathsf{is}') = kill_x((\mathsf{S}, \mathsf{H}, \mathsf{is}))$$

Then we must rename the abstract location corresponding to $y.sel$ to include x into the variable set, then we must add the new binding for x. We can have 3 different cases:

1. There is no abstract location n_Y such that $(y, n_Y) \in \mathsf{S}'$, or it exist but there is no n_Z such that $(n_Y, sel, n_Z) \in \mathsf{H}'$.
2. There is an abstract location n_Y such that $(y, n_Y) \in \mathsf{S}'$ and there is a $n_U \neq n_\emptyset$ such that $(n_Y, sel, n_U) \in \mathsf{H}'$.
3. There is an abstract location n_Y such that $(y, n_Y) \in \mathsf{S}'$ and there is a triple such that $(n_Y, sel, n_\emptyset) \in \mathsf{H}'$.

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

First of all we remove the old binding for x:

$$(\mathsf{S}', \mathsf{H}', \mathsf{is}') = kill_x((\mathsf{S}, \mathsf{H}, \mathsf{is}))$$

Then we must rename the abstract location corresponding to $y.sel$ to include x into the variable set, then we must add the new binding for x. We can have 3 different cases:

1. There is no abstract location n_Y such that $(y, n_Y) \in \mathsf{S}'$, or it exist but there is no n_Z such that $(n_Y, sel, n_Z) \in \mathsf{H}'$.
2. There is an abstract location n_Y such that $(y, n_Y) \in \mathsf{S}'$ and there is a $n_U \neq n_\emptyset$ such that $(n_Y, sel, n_U) \in \mathsf{H}'$.
3. There is an abstract location n_Y such that $(y, n_Y) \in \mathsf{S}'$ and there is a triple such that $(n_Y, sel, n_\emptyset) \in \mathsf{H}'$.

$[x := y.sel]^\ell$ - Case 1

In the case where there is no abstract location n_Y such that $(y, n_Y) \in \mathsf{S}'$ we simply take:

$$\phi_\ell^{\mathsf{SA}}((\mathsf{S}, \mathsf{H}, \mathsf{is})) = kill_x((\mathsf{S}, \mathsf{H}, \mathsf{is}))$$

In the case where there is no n_Z such that $(n_Y, sel, n_Z) \in \mathsf{H}'$ we again take

$$\phi_\ell^{\mathsf{SA}}((\mathsf{S}, \mathsf{H}, \mathsf{is})) = kill_x((\mathsf{S}, \mathsf{H}, \mathsf{is}))$$

In both case there is no binding to add, so we simply remove the old binding for x. The first case is the case of de-referencing of `nil`-pointer, the second case is the case of de-referencing of non-existing fields.

$[x := y.sel]^\ell$ - Case 1

In the case where there is no abstract location n_Y such that $(y, n_Y) \in \mathsf{S}'$ we simply take:

$$\phi^{\mathsf{SA}}_\ell((\mathsf{S}, \mathsf{H}, \mathsf{is})) = kill_x((\mathsf{S}, \mathsf{H}, \mathsf{is}))$$

In the case where there is no n_Z such that $(n_Y, sel, n_Z) \in \mathsf{H}'$ we again take

$$\phi^{\mathsf{SA}}_\ell((\mathsf{S}, \mathsf{H}, \mathsf{is})) = kill_x((\mathsf{S}, \mathsf{H}, \mathsf{is}))$$

In both case there is no binding to add, so we simply remove the old binding for x. The first case is the case of de-referencing of `nil`-pointer, the second case is the case of de-referencing of non-existing fields.

$[x := y.sel]^\ell$ - Case 1

In the case where there is no abstract location n_Y such that $(y, n_Y) \in \mathsf{S}'$ we simply take:

$$\phi^{\mathsf{SA}}_\ell((\mathsf{S}, \mathsf{H}, \mathsf{is})) = kill_x((\mathsf{S}, \mathsf{H}, \mathsf{is}))$$

In the case where there is no n_Z such that $(n_Y, sel, n_Z) \in \mathsf{H}'$ we again take

$$\phi^{\mathsf{SA}}_\ell((\mathsf{S}, \mathsf{H}, \mathsf{is})) = kill_x((\mathsf{S}, \mathsf{H}, \mathsf{is}))$$

In both case there is no binding to add, so we simply remove the old binding for x. The first case is the case of de-referencing of `nil`-pointer, the second case is the case of de-referencing of non-existing fields.

$[x := y.sel]^\ell$ - Case 2

We consider the case where there is an abstract location n_Y such that $(y, n_Y) \in \mathsf{S}'$ and there is a $n_U \neq n_\emptyset$ such that $(n_Y, sel, n_U) \in \mathsf{H}'$.

We must rename n_U to include x into the variable set using the function:

$$h_x^U(n_Z) = \begin{cases} n_{Z \cup \{x\}} & \text{if } Z = U \\ n_Z & \text{otherwise} \end{cases}$$

$[x := y.sel]^{\ell}$ - Case 2

We consider the case where there is an abstract location n_Y such that $(y, n_Y) \in \mathsf{S}'$ and there is a $n_U \neq n_\emptyset$ such that $(n_Y, sel, n_U) \in \mathsf{H}'$.

We must rename n_U to include x into the variable set using the function:

$$h_x^U(n_Z) = \begin{cases} n_{Z \cup \{x\}} & \text{if } Z = U \\ n_Z & \text{otherwise} \end{cases}$$

$[x := y.sel]^\ell$ - Case 2

So we take

$$\phi^{\mathsf{SA}}_\ell((\mathsf{S},\mathsf{H},\mathsf{is})) = \{(\mathsf{S}'',\mathsf{H}'',\mathsf{is}'')\}$$

Where $(\mathsf{S}',\mathsf{H}',\mathsf{is}') = kill_x((\mathsf{S},\mathsf{H},\mathsf{is}))$ and

$$\begin{aligned}
\mathsf{S}'' &= \{(z, h^U_x(n_Z)) \mid (z, n_Z) \in \mathsf{S}'\} \cup \{(x, h^U_x(n_U))\} \\
\mathsf{H}'' &= \{(h^U_x(n_V), sel, h^U_x(n_W)) \mid (n_V, sel, n_W) \in \mathsf{H}'\} \\
\mathsf{is}'' &= \{h^U_x(n_Z) \mid n_Z \in \mathsf{is}'\}
\end{aligned}$$

$[x := y.sel]^\ell$ - Case 2

So we take

$$\phi_\ell^{\mathsf{SA}}((\mathsf{S},\mathsf{H},\mathsf{is})) = \{(\mathsf{S}'',\mathsf{H}'',\mathsf{is}'')\}$$

Where $(\mathsf{S}',\mathsf{H}',\mathsf{is}') = kill_x((\mathsf{S},\mathsf{H},\mathsf{is}))$ and

$$\begin{aligned}
\mathsf{S}'' &= \{(z, h_x^U(n_Z)) \mid (z, n_Z) \in \mathsf{S}'\} \cup \{(x, h_x^U(n_U))\} \\
\mathsf{H}'' &= \{(h_x^U(n_V), sel, h_x^U(n_W)) \mid (n_V, sel, n_W) \in \mathsf{H}'\} \\
\mathsf{is}'' &= \{h_x^U(n_Z) \mid n_Z \in \mathsf{is}'\}
\end{aligned}$$

$[x := y.sel]^\ell$ - Case 2

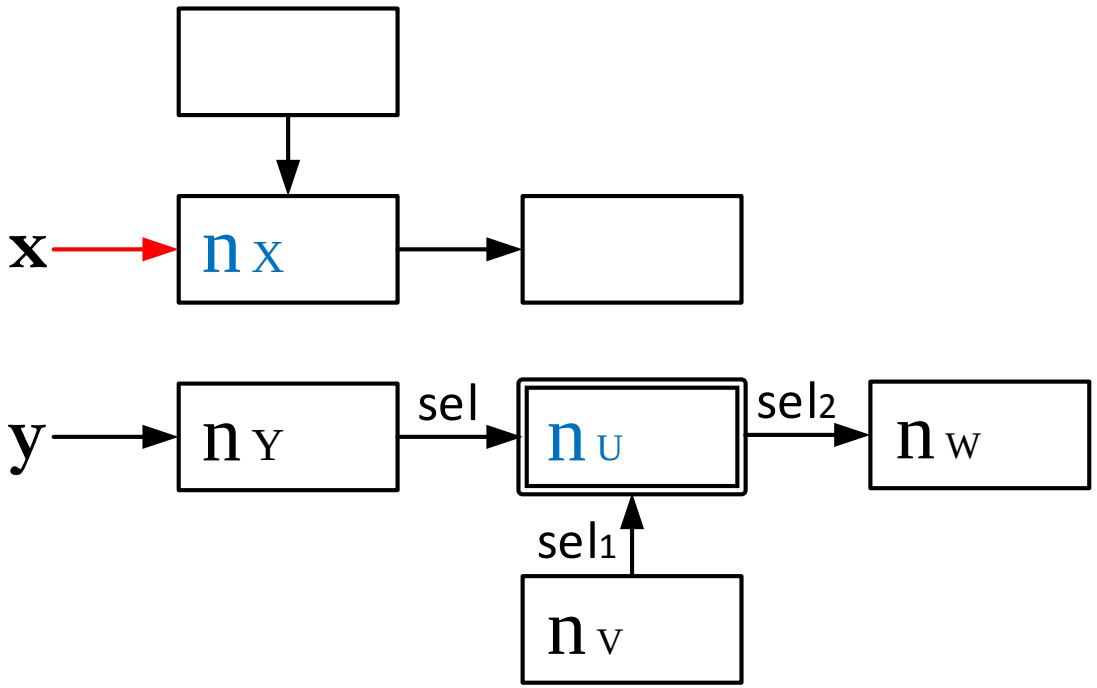
$$\mathsf{S}'' = \{(z, h_x^U(n_Z)) \mid (z, n_Z) \in \mathsf{S}'\} \cup \{(x, h_x^U(n_U))\}$$

The clause in blue represent the adding of the new binding for x.

As before $n_{U \cup \{x\}}$ is shared in (H'') if and only if n_U was shared in (H').

$[x := y.sel]^\ell$ - Case 2

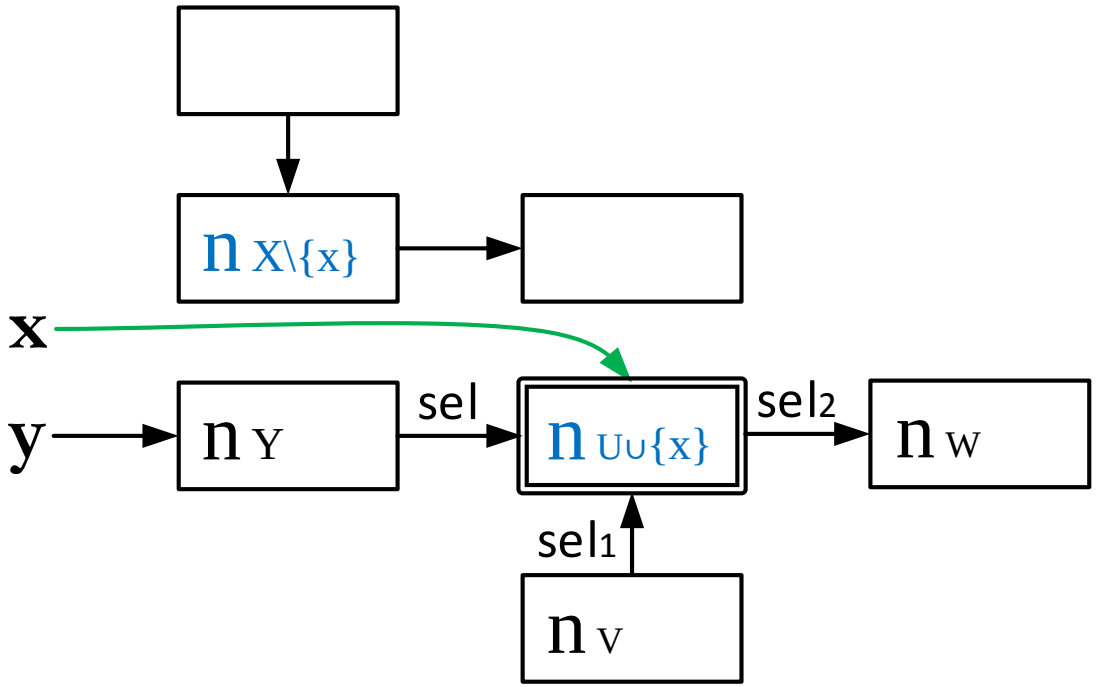
Let's show this case with a little example when $x \neq y$, and in case when $n_U \in$ is.



(S, H, is)

$[x := y.sel]^{\ell}$ - Case 2

Let's show this case with a little example when $x \neq y$, and in case when $n_U \in$ is.



(S'', H'', is'')

$[x := y.sel]^\ell$ - Case 3

Consider now the case where there is an abstract location n_Y such that $(y, n_Y) \in \mathsf{S}'$ and there is a triple such that $(n_Y, sel, n_\emptyset) \in \mathsf{H}'$.

The location for $y.sel$ is represented by $n_\emptyset$ (that represent even other locations). We must **materialize** an abstract location $n_{\{x\}}$ from $n_\emptyset$, doing so $n_{\{x\}}$ will represent $y.sel$ and $n_\emptyset$ will represent the remaining locations.

This operation is a bit hard so let's consider an example

$[x := y.sel]^\ell$ - Case 3

Consider now the case where there is an abstract location n_Y such that $(y, n_Y) \in \mathsf{S}'$ and there is a triple such that $(n_Y, sel, n_\emptyset) \in \mathsf{H}'$.

The location for $y.sel$ is represented by $n_\emptyset$ (that represent even other locations). We must **materialize** an abstract location $n_{\{x\}}$ from $n_\emptyset$, doing so $n_{\{x\}}$ will represent $y.sel$ and $n_\emptyset$ will represent the remaining locations.

This operation is a bit hard so let's consider an example

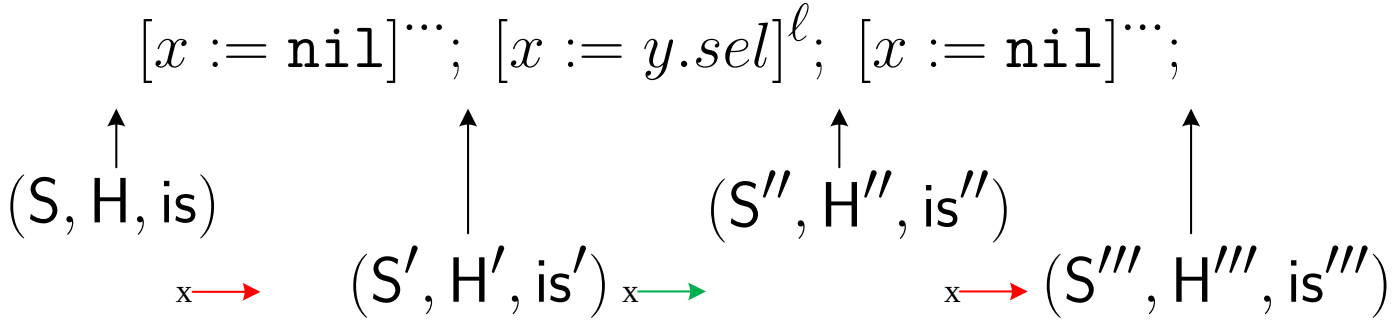
$[x := y.sel]^\ell$ - Case 3

Consider now the case where there is an abstract location n_Y such that $(y, n_Y) \in \mathsf{S}'$ and there is a triple such that $(n_Y, sel, n_\emptyset) \in \mathsf{H}'$.

The location for $y.sel$ is represented by $n_\emptyset$ (that represent even other locations). We must **materialize** an abstract location $n_{\{x\}}$ from $n_\emptyset$, doing so $n_{\{x\}}$ will represent $y.sel$ and $n_\emptyset$ will represent the remaining locations.

This operation is a bit hard so let's consider an example

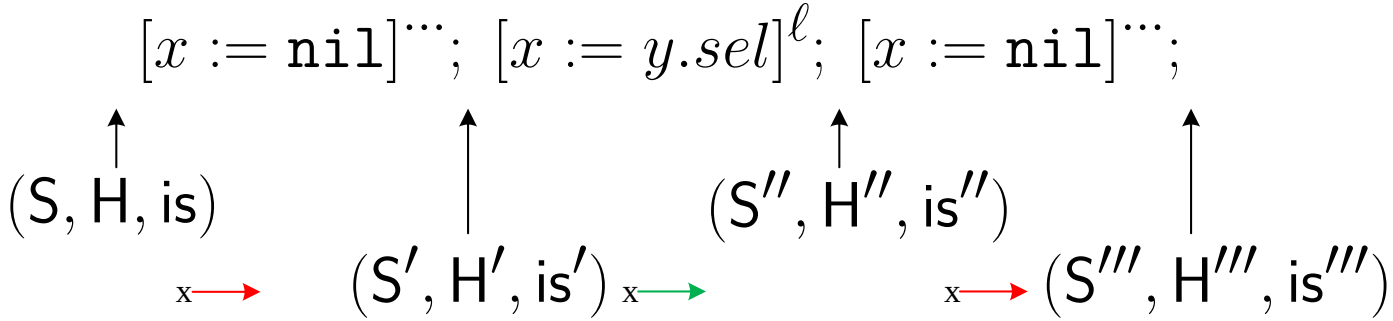
$[x := y.sel]^{\ell}$ - Case 3



We can determinate obviously that

- $[x := \mathtt{nil}]^{\cdots}$; $[x := y.sel]^{\ell}$ is equivalent to $[x := y.sel]^{\ell}$
- $(S', H', is') = kill_x((S, H, is))$
- $(S''', H''', is''') = kill_x((S'', H'', is''))$
- $(S''', H''', is''') = (S', H', is')$

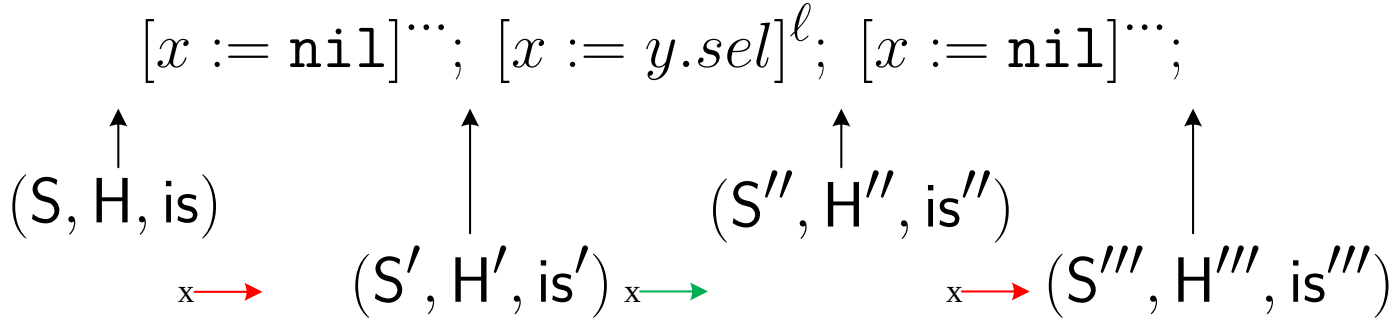
$[x := y.sel]^{\ell}$ - Case 3



We can determinate obviously that

- $[x := \mathtt{nil}]^{\cdots}$; $[x := y.sel]^{\ell}$ is equivalent to $[x := y.sel]^{\ell}$
- $(\mathsf{S}', \mathsf{H}', \mathsf{is}') = kill_x((\mathsf{S}, \mathsf{H}, \mathsf{is}))$
- $(\mathsf{S}''', \mathsf{H}''', \mathsf{is}''') = kill_x((\mathsf{S}'', \mathsf{H}'', \mathsf{is}''))$
- $(\mathsf{S}''', \mathsf{H}''', \mathsf{is}''') = (\mathsf{S}', \mathsf{H}', \mathsf{is}')$

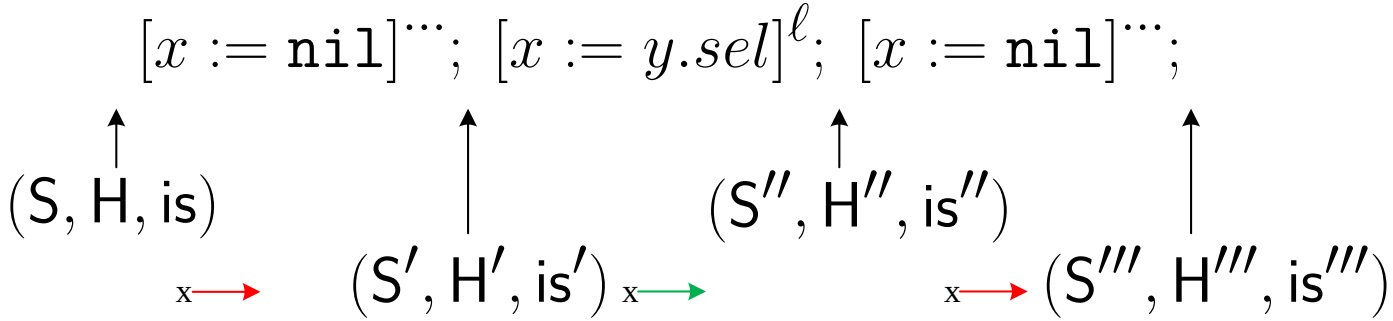
$[x := y.sel]^{\ell}$ - Case 3



We can determinate obviously that

- $[x := \mathtt{nil}]^{\cdots}$; $[x := y.sel]^{\ell}$ is equivalent to $[x := y.sel]^{\ell}$
- $(\mathsf{S}', \mathsf{H}', \mathsf{is}') = kill_x((\mathsf{S}, \mathsf{H}, \mathsf{is}))$
- $(\mathsf{S}''', \mathsf{H}''', \mathsf{is}''') = kill_x((\mathsf{S}'', \mathsf{H}'', \mathsf{is}''))$
- $(\mathsf{S}''', \mathsf{H}''', \mathsf{is}''') = (\mathsf{S}', \mathsf{H}', \mathsf{is}')$

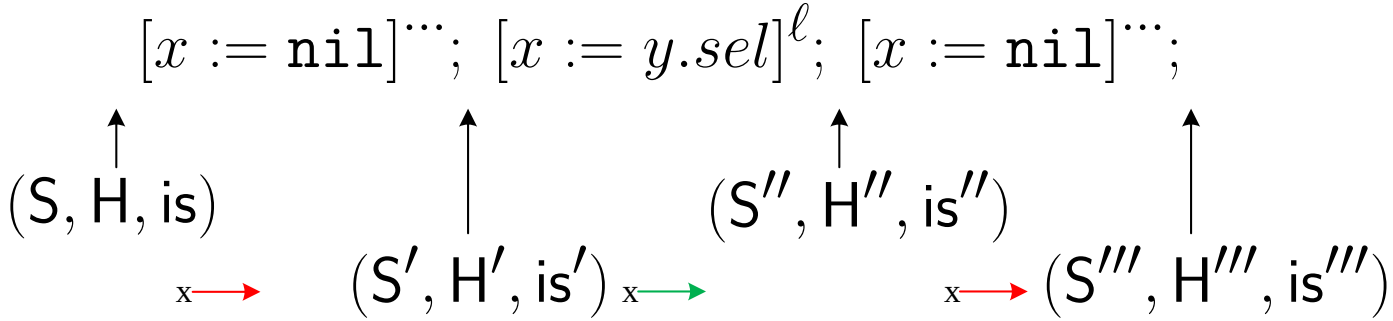
$[x := y.sel]^{\ell}$ - Case 3



We can determinate obviously that

- $[x := \mathtt{nil}]^{\cdots}$; $[x := y.sel]^{\ell}$ is equivalent to $[x := y.sel]^{\ell}$
- $(\mathsf{S}', \mathsf{H}', \mathsf{is}') = kill_x((\mathsf{S}, \mathsf{H}, \mathsf{is}))$
- $(\mathsf{S}''', \mathsf{H}''', \mathsf{is}''') = kill_x((\mathsf{S}'', \mathsf{H}'', \mathsf{is}''))$
- $(\mathsf{S}''', \mathsf{H}''', \mathsf{is}''') = (\mathsf{S}', \mathsf{H}', \mathsf{is}')$

$[x := y.sel]^{\ell}$ - Case 3



We can determinate obviously that

- $[x := \mathtt{nil}]^{\cdots}$; $[x := y.sel]^{\ell}$ is equivalent to $[x := y.sel]^{\ell}$
- $(\mathsf{S}', \mathsf{H}', \mathsf{is}') = kill_x((\mathsf{S}, \mathsf{H}, \mathsf{is}))$
- $(\mathsf{S}''', \mathsf{H}''', \mathsf{is}''') = kill_x((\mathsf{S}'', \mathsf{H}'', \mathsf{is}''))$
- $(\mathsf{S}''', \mathsf{H}''', \mathsf{is}''') = (\mathsf{S}', \mathsf{H}', \mathsf{is}')$

$[x := y.sel]^\ell$ - Case 3

So finally

$$kill_x((\mathsf{S}'', \mathsf{H}'', \mathsf{is}'')) = (\mathsf{S}', \mathsf{H}', \mathsf{is}')$$

Then we can also say that $(x, n_{\{x\}}) \in \mathsf{S}''$ and that $(n_Y, sel, n_{\{x\}}) \in \mathsf{H}''$.

So we can take

$$\begin{aligned}\phi^{\mathsf{SA}}_\ell((\mathsf{S}, \mathsf{H}, \mathsf{is})) \;=\; \{&(\mathsf{S}'', \mathsf{H}'', \mathsf{is}'') \mid \\ &(\mathsf{S}'', \mathsf{H}'', \mathsf{is}'') \text{ is compatible } \wedge \\ &kill_x((\mathsf{S}'', \mathsf{H}'', \mathsf{is}'')) = (\mathsf{S}', \mathsf{H}', \mathsf{is}') \wedge \\ &(x, n_{\{x\}}) \in \mathsf{S}'' \wedge (n_Y, sel, n_{\{x\}}) \in \mathsf{H}''\}\end{aligned}$$

$[x := y.sel]^\ell$ - Case 3

So finally

$$kill_x((\mathsf{S}'', \mathsf{H}'', \mathsf{is}'')) = (\mathsf{S}', \mathsf{H}', \mathsf{is}')$$

Then we can also say that $(x, n_{\{x\}}) \in \mathsf{S}''$ and that $(n_Y, sel, n_{\{x\}}) \in \mathsf{H}''$.

So we can take

$$\begin{aligned}\phi^{\mathsf{SA}}_\ell((\mathsf{S}, \mathsf{H}, \mathsf{is})) \;=\; \{&(\mathsf{S}'', \mathsf{H}'', \mathsf{is}'') \mid \\ &(\mathsf{S}'', \mathsf{H}'', \mathsf{is}'') \text{ is compatible } \wedge \\ &kill_x((\mathsf{S}'', \mathsf{H}'', \mathsf{is}'')) = (\mathsf{S}', \mathsf{H}', \mathsf{is}') \wedge \\ &(x, n_{\{x\}}) \in \mathsf{S}'' \;\wedge\; (n_Y, sel, n_{\{x\}}) \in \mathsf{H}''\}\end{aligned}$$

$[x := y.sel]^\ell$ - Case 3

We are sure that we have included all the possible solutions. So our function is **sound**.

Is it possible that we have included too much solutions, so we are **losing precision**?

We will now argue that the amount of imprecision in **not excessive**.

$[x := y.sel]^\ell$ - Case 3

We are sure that we have included all the possible solutions. So our function is **sound**.

Is it possible that we have included too much solutions, so we are **losing precision**?

We will now argue that the amount of imprecision in **not excessive**.

$[x := y.sel]^\ell$ - Case 3

We are sure that we have included all the possible solutions. So our function is **sound**.

Is it possible that we have included too much solutions, so we are **losing precision**?

We will now argue that the amount of imprecision in **not excessive**.

$[x := y.sel]^\ell$ - Case 3

First we prove that

$$\mathsf{S}'' = \mathsf{S}' \cup \{(x, n_{\{x\}})\}$$

Proof

$\mathsf{S}'' \subseteq \mathsf{S}' \cup \{(x, n_{\{x\}})\}$ Consider $(z, n_Z) \in \mathsf{S}''$. If $z = x$ then from compatibility $n_Z = n_{\{x\}}$. If $z \neq x$ then from $(x, n_{\{x\}}) \in \mathsf{S}''$ and the compatibility we deduce that $x \notin Z$ and so $(z, n_Z) = (z, k_x(n_Z))$ (and k_x is used to define $kill_x$).

$\mathsf{S}'' \supseteq \mathsf{S}' \cup \{(x, n_{\{x\}})\}$ Consider $(u, n_U) \in \mathsf{S}'$. We know from definition of S' and from compatibility that $u \neq x$ and $x \notin U$. There must be a $(u, n'_U) \in \mathsf{S}''$ such that $k_x(n'_U) = n_U$, but since $x \neq y$ we obtain $n'_U = n_U$.

□

$[x := y.sel]^\ell$ - Case 3

First we prove that

$$\mathsf{S}'' = \mathsf{S}' \cup \{(x, n_{\{x\}})\}$$

Proof

$\mathsf{S}'' \subseteq \mathsf{S}' \cup \{(x, n_{\{x\}})\}$ Consider $(z, n_Z) \in \mathsf{S}''$. If $z = x$ then from compatibility $n_Z = n_{\{x\}}$. If $z \neq x$ then from $(x, n_{\{x\}}) \in \mathsf{S}''$ and the compatibility we deduce that $x \notin Z$ and so $(z, n_Z) = (z, k_x(n_Z))$ (and k_x is used to define $kill_x$).

$\mathsf{S}'' \supseteq \mathsf{S}' \cup \{(x, n_{\{x\}})\}$ Consider $(u, n_U) \in \mathsf{S}'$. We know from definition of S' and from compatibility that $u \neq x$ and $x \notin U$. There must be a $(u, n'_U) \in \mathsf{S}''$ such that $k_x(n'_U) = n_U$, but since $x \neq y$ we obtain $n'_U = n_U$.

□

$[x := y.sel]^\ell$ - Case 3

Then we prove that

$$\begin{array}{rcl} \mathsf{is}'\backslash\{n_\emptyset\} & = & \mathsf{is}''\backslash\{n_\emptyset, n_{\{x\}}\} \\ n_\emptyset \in \mathsf{is}' & \text{iff} & n_\emptyset \in \mathsf{is}'' \vee n_{\{x\}} \in \mathsf{is}'' \end{array}$$

Equal as showing that

- Sharing information of location apart $n_\emptyset$ are conserved,
- If $n_\emptyset$ is shared then it can rise share to $n_\emptyset$ or to $n_{\{x\}}$ (or both),
- If $n_\emptyset$ is not shared, we can't introduce a sharing for $n_\emptyset$ neither for $n_{\{x\}}$.

$[x := y.sel]^\ell$ - Case 3

Then we prove that

$$\begin{array}{rcl} \mathsf{is}'\backslash\{n_\emptyset\} & = & \mathsf{is}''\backslash\{n_\emptyset, n_{\{x\}}\} \\ n_\emptyset \in \mathsf{is}' & \text{iff} & n_\emptyset \in \mathsf{is}'' \vee n_{\{x\}} \in \mathsf{is}'' \end{array}$$

Equal as showing that

- Sharing information of location apart $n_\emptyset$ are conserved,
- If $n_\emptyset$ is shared then it can rise share to $n_\emptyset$ or to $n_{\{x\}}$ (or both),
- If $n_\emptyset$ is not shared, we can't introduce a sharing for $n_\emptyset$ neither for $n_{\{x\}}$.

$[x := y.sel]^\ell$ - Case 3

Then we prove that

$$\begin{array}{rcl} \mathsf{is}'\backslash\{n_\emptyset\} & = & \mathsf{is}''\backslash\{n_\emptyset, n_{\{x\}}\} \\ n_\emptyset \in \mathsf{is}' & \text{iff} & n_\emptyset \in \mathsf{is}'' \vee n_{\{x\}} \in \mathsf{is}'' \end{array}$$

Proof

Since $(\mathsf{S}', \mathsf{H}', \mathsf{is}')$ and $(\mathsf{S}'', \mathsf{H}'', \mathsf{is}'')$ are compatible, if $n_U \in \mathsf{is}'$ then $x \notin U$ and if $n_U \in \mathsf{is}''$ then $x \notin U \vee \{x\} = U$.

Remember that $\mathsf{is}' = \{k_x(n_U) \mid n_U \in \mathsf{is}''\}$, so we can deduce that $\mathsf{is}'\backslash\{n_\emptyset\} = \mathsf{is}''\backslash\{n_\emptyset, n_{\{x\}}\}$, because $k_x(n_U) = n_U \neq n_\emptyset$ for all $n_U \in \mathsf{is}''\backslash\{n_\emptyset, n_{\{x\}}\}$.

More $n_\emptyset \in \mathsf{is}'' \vee n_{\{x\}} \in \mathsf{is}'' \Rightarrow n_\emptyset \in \mathsf{is}'$ and more $n_\emptyset \notin \mathsf{is}'' \wedge n_{\{x\}} \notin \mathsf{is}'' \Rightarrow n_\emptyset \notin \mathsf{is}'$

$[x := y.sel]^\ell$ - Case 3

Then we prove that

$$\begin{array}{rcl} \mathsf{is}'\backslash\{n_\emptyset\} & = & \mathsf{is}''\backslash\{n_\emptyset, n_{\{x\}}\} \\ n_\emptyset \in \mathsf{is}' & \text{iff} & n_\emptyset \in \mathsf{is}'' \vee n_{\{x\}} \in \mathsf{is}'' \end{array}$$

Proof

Since $(\mathsf{S}', \mathsf{H}', \mathsf{is}')$ and $(\mathsf{S}'', \mathsf{H}'', \mathsf{is}'')$ are compatible, if $n_U \in \mathsf{is}'$ then $x \notin U$ and if $n_U \in \mathsf{is}''$ then $x \notin U \vee \{x\} = U$.

Remember that $\mathsf{is}' = \{k_x(n_U) \mid n_U \in \mathsf{is}''\}$, so we can deduce that $\mathsf{is}'\backslash\{n_\emptyset\} = \mathsf{is}''\backslash\{n_\emptyset, n_{\{x\}}\}$, because $k_x(n_U) = n_U \neq n_\emptyset$ for all $n_U \in \mathsf{is}''\backslash\{n_\emptyset, n_{\{x\}}\}$.

More $n_\emptyset \in \mathsf{is}'' \vee n_{\{x\}} \in \mathsf{is}'' \Rightarrow n_\emptyset \in \mathsf{is}'$ and more $n_\emptyset \notin \mathsf{is}'' \wedge n_{\{x\}} \notin \mathsf{is}'' \Rightarrow n_\emptyset \notin \mathsf{is}'$

□

$[x := y.sel]^{\ell}$ - Case 3

Now consider the abstract heap H, we first classify the edge depending on if the source and the target are either $n_{\emptyset}$ or $n_{\{x\}}$.

$$(n_V, sel', n_W) \text{ is } external \quad \text{iff} \quad \{n_V, n_W\} \cap \{n_{\emptyset}, n_{\{x\}}\} = \emptyset$$
$$(n_V, sel', n_W) \text{ is } internal \quad \text{iff} \quad \{n_V, n_W\} \subseteq \{n_{\emptyset}, n_{\{x\}}\}$$
$$(n_V, sel', n_W) \text{ is } going\text{-}out \quad \text{iff} \quad n_V \in \{n_{\emptyset}, n_{\{x\}}\} \wedge n_W \notin \{n_{\emptyset}, n_{\{x\}}\}$$
$$(n_V, sel', n_W) \text{ is } going\text{-}in \quad \text{iff} \quad n_V \notin \{n_{\emptyset}, n_{\{x\}}\} \wedge n_W \in \{n_{\emptyset}, n_{\{x\}}\}$$

We say that two edge (n_V, sel', n_W) and (n'_V, sel'', n'_W) are **related** if and only if $k_x(n_V) = k_x(n'_V)$, $sel' = sel''$ and $k_x(n_W) = k_x(n'_W)$.

$[x := y.sel]^\ell$ - Case 3

Now consider the abstract heap H, we first classify the edge depending on if the source and the target are either $n_\emptyset$ or $n_{\{x\}}$.

$$
\begin{array}{rcl}
(n_V, sel', n_W) \text{ is } external & \text{iff} & \{n_V, n_W\} \cap \{n_\emptyset, n_{\{x\}}\} = \emptyset \\
(n_V, sel', n_W) \text{ is } internal & \text{iff} & \{n_V, n_W\} \subseteq \{n_\emptyset, n_{\{x\}}\} \\
(n_V, sel', n_W) \text{ is } going\text{-}out & \text{iff} & n_V \in \{n_\emptyset, n_{\{x\}}\} \wedge n_W \notin \{n_\emptyset, n_{\{x\}}\} \\
(n_V, sel', n_W) \text{ is } going\text{-}in & \text{iff} & n_V \notin \{n_\emptyset, n_{\{x\}}\} \wedge n_W \in \{n_\emptyset, n_{\{x\}}\}
\end{array}
$$

We say that two edge (n_V, sel', n_W) and (n'_V, sel'', n'_W) are **related** if and only if $k_x(n_V) = k_x(n'_V)$, $sel' = sel''$ and $k_x(n_W) = k_x(n'_W)$.

$[x := y.sel]^{\ell}$ - Case 3

Now consider the abstract heap H, we first classify the edge depending on if the source and the target are either $n_{\emptyset}$ or $n_{\{x\}}$.

$$
\begin{aligned}
(n_V, sel', n_W) \text{ is } external &\quad \text{iff} \quad \{n_V, n_W\} \cap \{n_{\emptyset}, n_{\{x\}}\} = \emptyset \\
(n_V, sel', n_W) \text{ is } internal &\quad \text{iff} \quad \{n_V, n_W\} \subseteq \{n_{\emptyset}, n_{\{x\}}\} \\
(n_V, sel', n_W) \text{ is } going\text{-}out &\quad \text{iff} \quad n_V \in \{n_{\emptyset}, n_{\{x\}}\} \wedge n_W \notin \{n_{\emptyset}, n_{\{x\}}\} \\
(n_V, sel', n_W) \text{ is } going\text{-}in &\quad \text{iff} \quad n_V \notin \{n_{\emptyset}, n_{\{x\}}\} \wedge n_W \in \{n_{\emptyset}, n_{\{x\}}\}
\end{aligned}
$$

We say that two edge (n_V, sel', n_W) and (n'_V, sel'', n'_W) are **related** if and only if $k_x(n_V) = k_x(n'_V)$, $sel' = sel''$ and $k_x(n_W) = k_x(n'_W)$.

$[x := y.sel]^\ell$ - Case 3

We can show that

- H′ and H″ have the same external edges,
- each internal edge in H′ is related to an internal edge in H″ and vice versa,
- each going-out edge in H′ is related to a going-out edge in H″ and vice versa,
- each going-in edge in H′ is related to a going-in edge in H″ and vice versa,

Obviously the edge $(n_Y, sel, n_\emptyset) \in \mathsf{H}'$ must be update with $(n_Y, sel, n_{\{x\}}) \in \mathsf{H}''$. We impose that $(n_Y, sel, n_{\{x\}}) \in \mathsf{H}''$, and because of compatibility we deduce that $(n_Y, sel, n_{\{x\}}) \notin \mathsf{H}'$.

$[x := y.sel]^\ell$ - Case 3

We can show that

- H' and H'' have the same external edges,
- each internal edge in H' is related to an internal edge in H'' and vice versa,
- each going-out edge in H' is related to a going-out edge in H'' and vice versa,
- each going-in edge in H' is related to a going-in edge in H'' and vice versa,

Obviously the edge $(n_Y, sel, n_\emptyset) \in \mathsf{H}'$ must be update with $(n_Y, sel, n_{\{x\}}) \in \mathsf{H}''$. We impose that $(n_Y, sel, n_{\{x\}}) \in \mathsf{H}''$, and because of compatibility we deduce that $(n_Y, sel, n_{\{x\}}) \notin \mathsf{H}'$.

$[x := y.sel]^\ell$ - Case 3

We can show that

- H' and H'' have the same external edges,
- each internal edge in H' is related to an internal edge in H'' and vice versa,
- each going-out edge in H' is related to a going-out edge in H'' and vice versa,
- each going-in edge in H' is related to a going-in edge in H'' and vice versa,

Obviously the edge $(n_Y, sel, n_\emptyset) \in \mathsf{H}'$ must be update with $(n_Y, sel, n_{\{x\}}) \in \mathsf{H}''$. We impose that $(n_Y, sel, n_{\{x\}}) \in \mathsf{H}''$, and because of compatibility we deduce that $(n_Y, sel, n_{\{x\}}) \notin \mathsf{H}'$.

$[x := y.sel]^{\ell}$ - Case 3

We can show that

- H' and H'' have the same external edges,
- each internal edge in H' is related to an internal edge in H'' and vice versa,
- each going-out edge in H' is related to a going-out edge in H'' and vice versa,
- each going-in edge in H' is related to a going-in edge in H'' and vice versa,

Obviously the edge $(n_Y, sel, n_\emptyset) \in \mathsf{H}'$ must be update with $(n_Y, sel, n_{\{x\}}) \in \mathsf{H}''$. We impose that $(n_Y, sel, n_{\{x\}}) \in \mathsf{H}''$, and because of compatibility we deduce that $(n_Y, sel, n_{\{x\}}) \notin \mathsf{H}'$.

$[x := y.sel]^{\ell}$ - Case 3

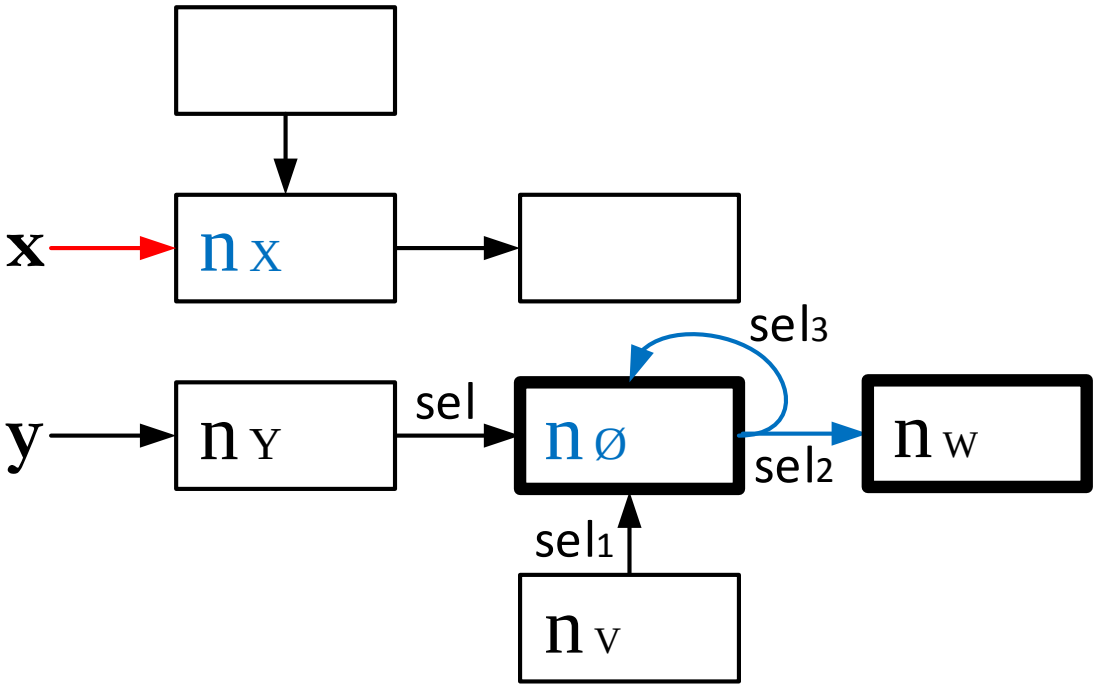
We can show that

- H′ and H″ have the same external edges,
- each internal edge in H′ is related to an internal edge in H″ and vice versa,
- each going-out edge in H′ is related to a going-out edge in H″ and vice versa,
- each going-in edge in H′ is related to a going-in edge in H″ and vice versa,

Obviously the edge $(n_Y, sel, n_\emptyset) \in$ H′ must be update with $(n_Y, sel, n_{\{x\}}) \in$ H″. We impose that $(n_Y, sel, n_{\{x\}}) \in$ H″, and because of compatibility we deduce that $(n_Y, sel, n_{\{x\}}) \notin$ H′.

$[x := y.sel]^\ell$ - Case 3

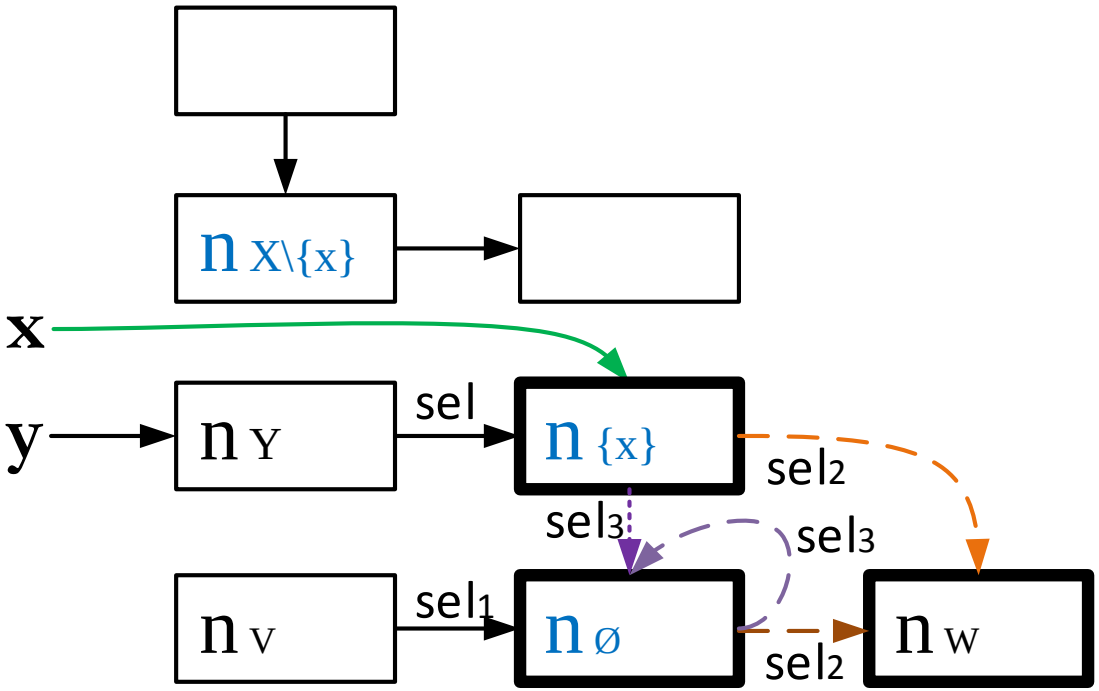
Consider the following scenario and let's see what happens after $[x := y.sel]^\ell$.



(S, H, is)

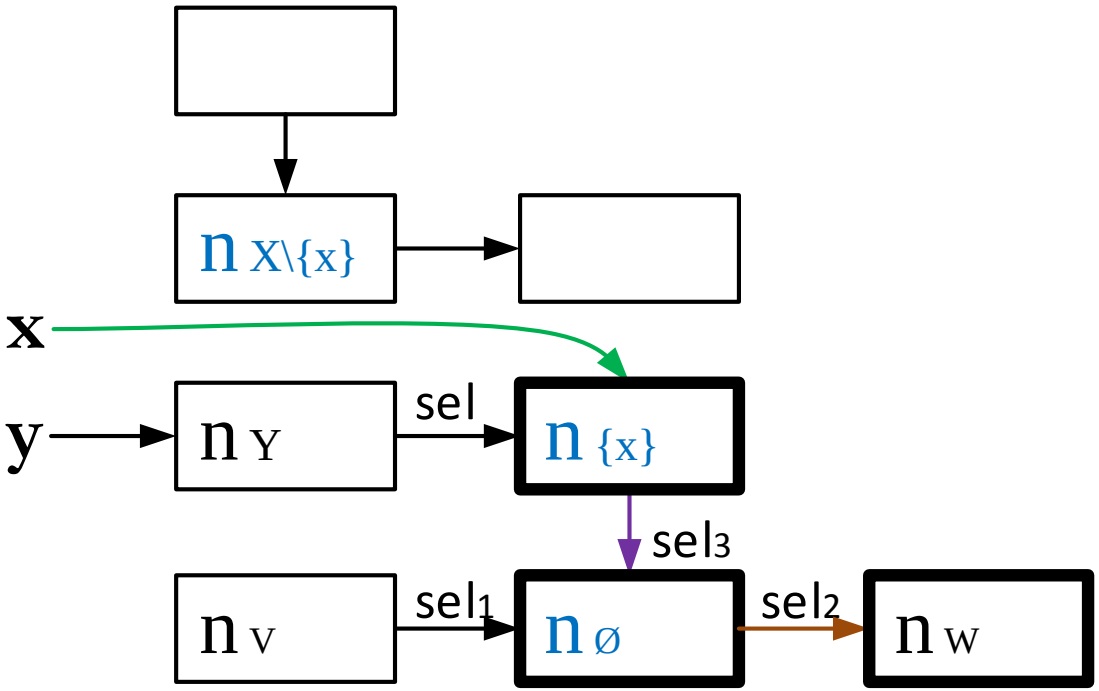
$[x := y.sel]^{\ell}$ - Case 3

We extract the location n_X from $n_{\emptyset}$, and depending on the triple involving sel_2 and sel_3 we can have 6 different and compatible shape graphs.



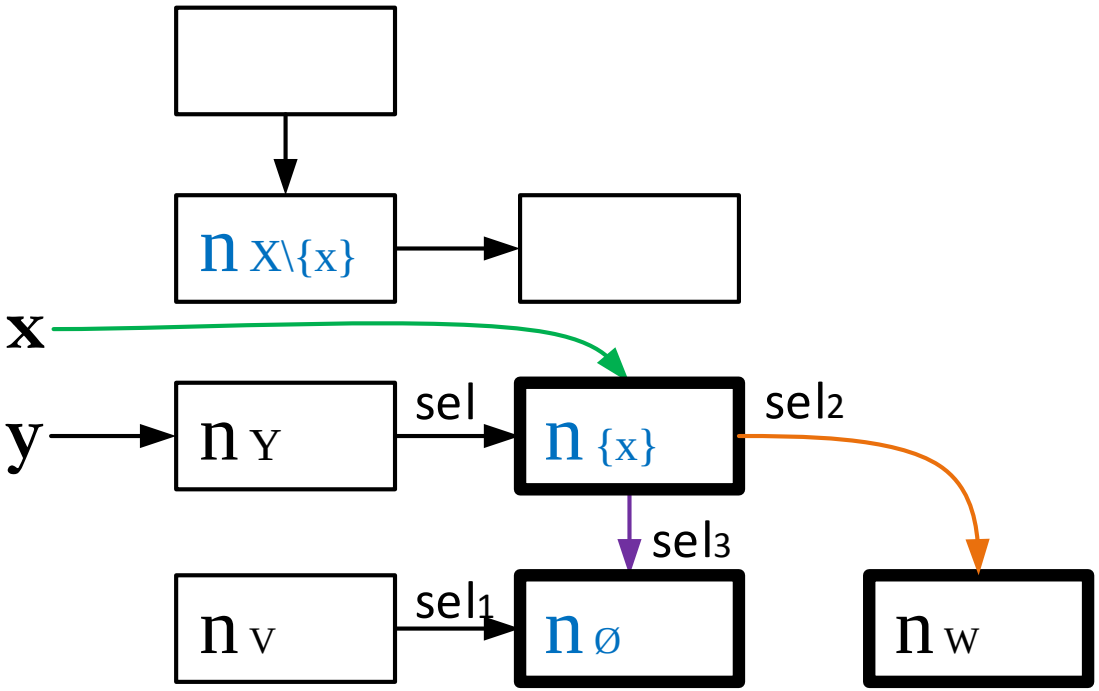
(S'', H'', is'')

$[x := y.sel]^{\ell}$ - Case 3



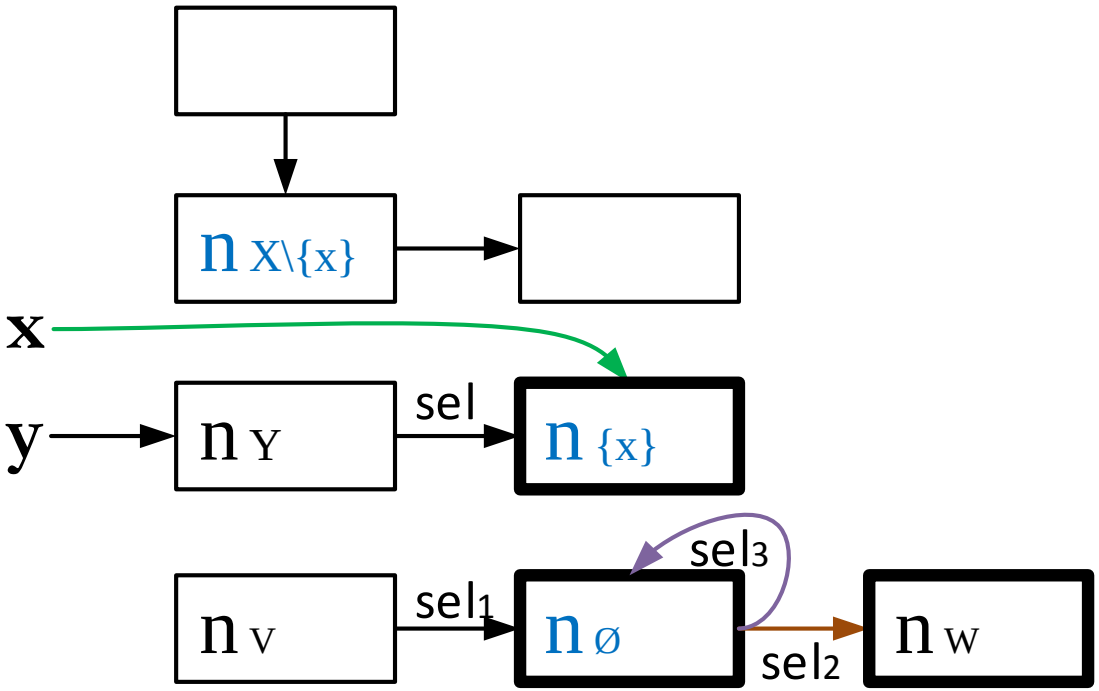
(S_1'', H_1'', is_1'')

$[x := y.sel]^{\ell}$ - Case 3



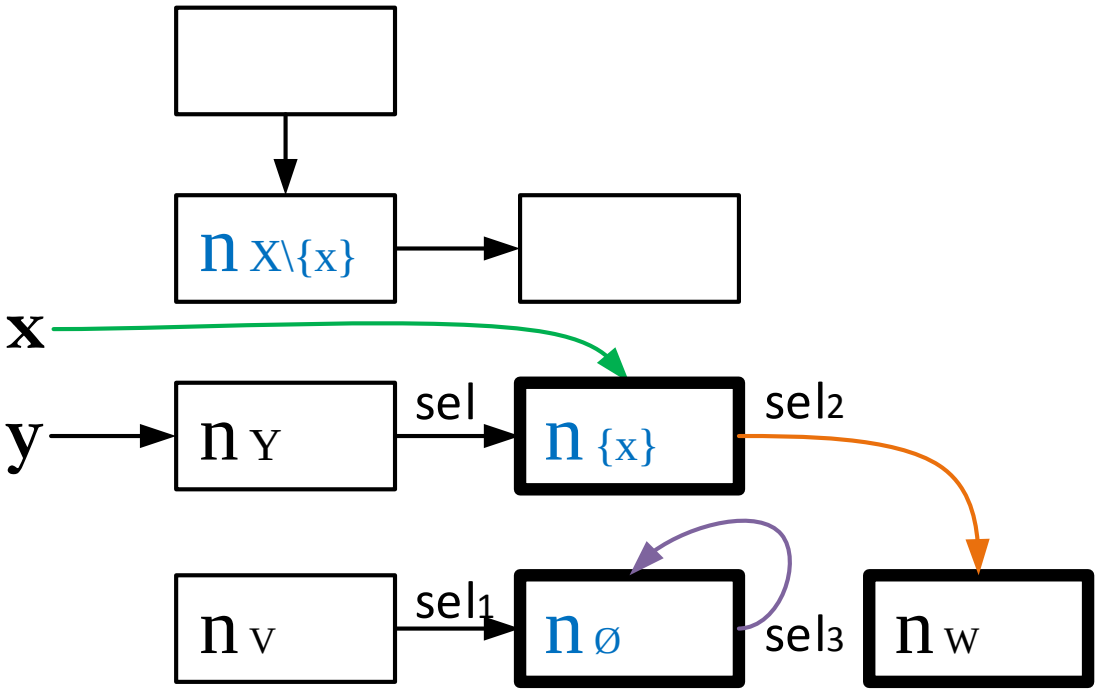
(S_2'', H_2'', is_2'')

$[x := y.sel]^{\ell}$ - Case 3



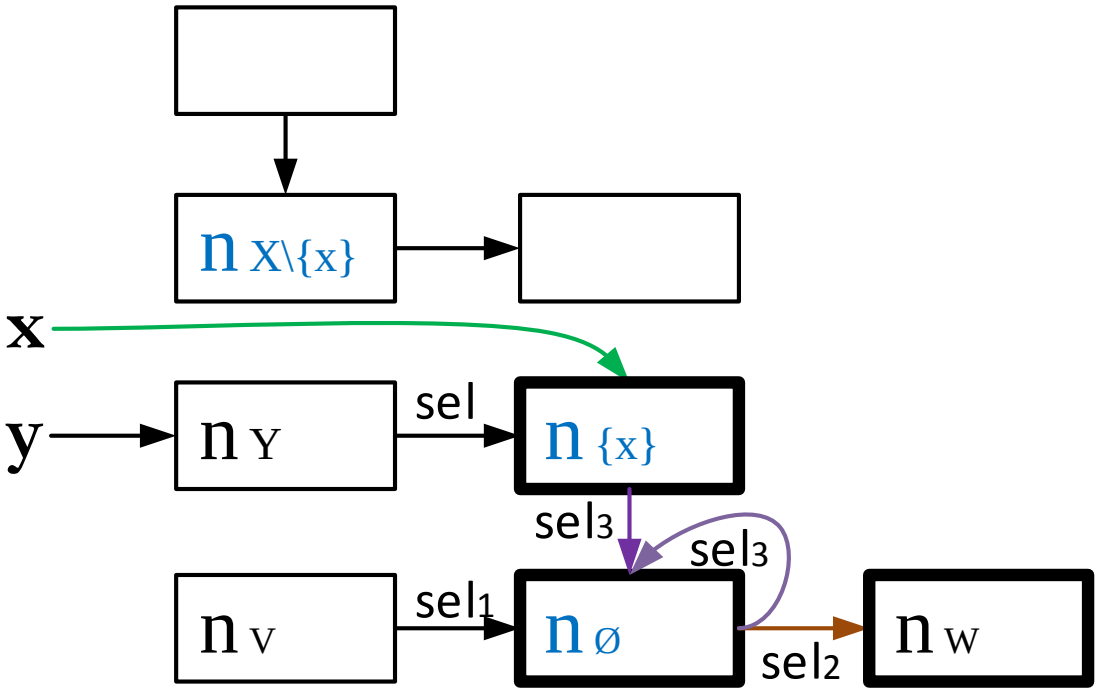
(S''_3, H''_3, is''_3)

$[x := y.sel]^{\ell}$ - Case 3



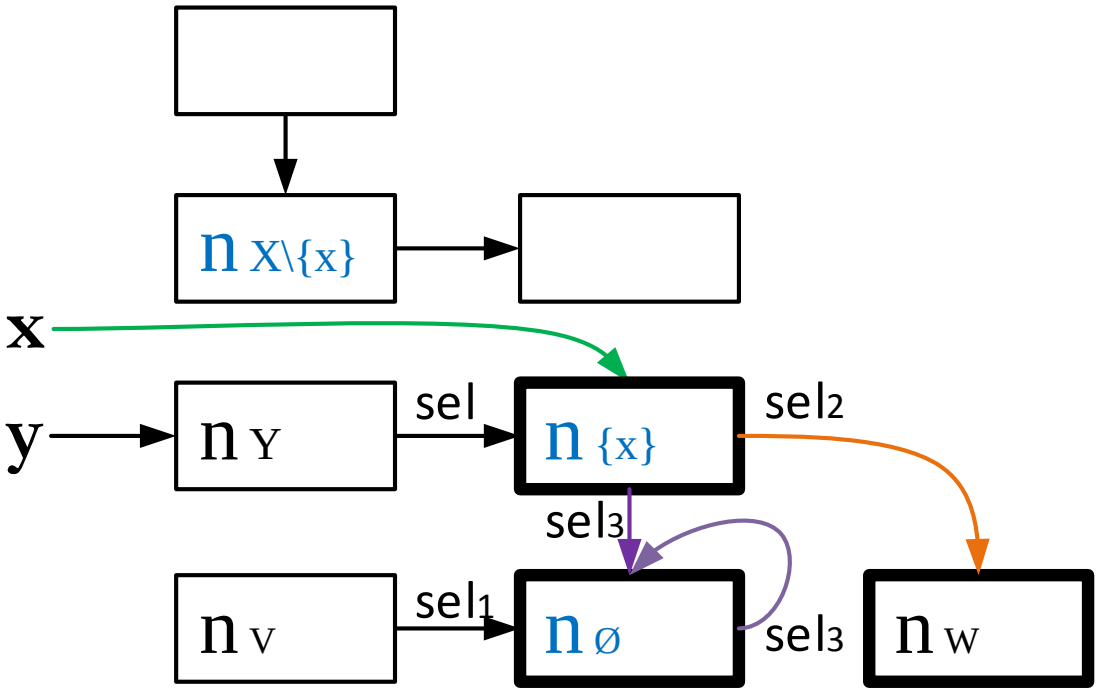
(S''_4, H''_4, is''_4)

$[x := y.sel]^{\ell}$ - Case 3



(S''_5, H''_5, is''_5)

$[x := y.sel]^\ell$ - Case 3



(S''_6, H''_6, is''_6)

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

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

We consider $[x.sel := a]^\ell$ in the case where a is of the form n, $a_1\ op_a\ a_2$ or `nil`.

Let's consider a compatible graph $(\mathsf{S}, \mathsf{H}, \mathsf{is})$.

The easy case are:

- The case where there is no n_X such that $(x, n_X) \in \mathsf{S}$, the statement will have no effect (x does not point to a cell in the heap). In this case the transfer function f_ℓ^{SA} is just the identity.

- The case where there is a n_X such that $(x, n_X) \in \mathsf{S}$, but there is no n_U such that $(n_X, sel, n_U) \in \mathsf{H}$; even in this case the statement will have no effect (the cell pointed by sel does not point to another cell in the heap). In this case the transfer function f_ℓ^{SA} is again the identity.

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

We consider $[x.sel := a]^\ell$ in the case where a is of the form n, $a_1\ op_a\ a_2$ or `nil`.

Let's consider a compatible graph $(\mathsf{S}, \mathsf{H}, \mathsf{is})$.

The easy case are:

- The case where there is no n_X such that $(x, n_X) \in \mathsf{S}$, the statement will have no effect (x does not point to a cell in the heap). In this case the transfer function f_ℓ^{SA} is just the identity.
- The case where there is a n_X such that $(x, n_X) \in \mathsf{S}$, but there is no n_U such that $(n_X, sel, n_U) \in \mathsf{H}$; even in this case the statement will have no effect (the cell pointed by sel does not point to another cell in the heap). In this case the transfer function f_ℓ^{SA} is again the identity.

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

We consider $[x.sel := a]^\ell$ in the case where a is of the form n, $a_1\ op_a\ a_2$ or `nil`.

Let's consider a compatible graph $(\mathsf{S}, \mathsf{H}, \mathsf{is})$.

The easy case are:

- The case where there is no n_X such that $(x, n_X) \in \mathsf{S}$, the statement will have no effect (x does not point to a cell in the heap). In this case the transfer function f_ℓ^{SA} is just the identity.

- The case where there is a n_X such that $(x, n_X) \in \mathsf{S}$, but there is no n_U such that $(n_X, sel, n_U) \in \mathsf{H}$; even in this case the statement will have no effect (the cell pointed by sel does not point to another cell in the heap). In this case the transfer function f_ℓ^{SA} is again the identity.

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

Consider the case where there is a n_X such that $(x, n_X) \in \mathsf{S}$ and there is a unique n_U such that $(n_X, sel, n_U) \in \mathsf{H}$.

The effect will be to **remove** the triple (n_X, sel, n_U):

$$\phi^{\mathsf{SA}}_\ell((\mathsf{S}, \mathsf{H}, \mathsf{is})) = \{kill_{x.sel}(\mathsf{S}, \mathsf{H}, \mathsf{is})\}$$

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

Consider the case where there is a n_X such that $(x, n_X) \in \mathsf{S}$ and there is a unique n_U such that $(n_X, sel, n_U) \in \mathsf{H}$.

The effect will be to **remove** the triple (n_X, sel, n_U):

$$\phi^{\mathsf{SA}}_\ell((\mathsf{S}, \mathsf{H}, \mathsf{is})) = \{kill_{x.sel}(\mathsf{S}, \mathsf{H}, \mathsf{is})\}$$

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

$$\phi^{\mathsf{SA}}_\ell((\mathsf{S},\mathsf{H},\mathsf{is})) = \{kill_{x.sel}(\mathsf{S},\mathsf{H},\mathsf{is})\}$$

where $kill_{x.sel}(\mathsf{S},\mathsf{H},\mathsf{is}) = (\mathsf{S}',\mathsf{H}',\mathsf{is}')$ is given by

$$\begin{array}{lcl}
\mathsf{S}' & = & \mathsf{S} \\
\mathsf{H}' & = & \{(n_V, sel', n_W) | (n_V, sel', n_W) \in \mathsf{H} \wedge \\
 & & \neg(X = V \wedge sel = sel')\} \\
\mathsf{is}' & = & \begin{cases} \mathsf{is}\backslash\{n_U\} & \text{if } n_U \in \mathsf{is} \wedge \#into(n_U, \mathsf{H}') \leq 1 \wedge \\ & \quad \neg\exists sel' : (n_\emptyset, sel', n_U) \in \mathsf{H}' \\ \mathsf{is} & \text{otherwise} \end{cases}
\end{array}$$

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

$$\phi^{\mathsf{SA}}_\ell((\mathsf{S},\mathsf{H},\mathsf{is})) = \{kill_{x.sel}(\mathsf{S},\mathsf{H},\mathsf{is})\}$$

where $kill_{x.sel}(\mathsf{S},\mathsf{H},\mathsf{is}) = (\mathsf{S}',\mathsf{H}',\mathsf{is}')$ is given by

$$\begin{array}{lcl}
\mathsf{S}' & = & \mathsf{S} \\
\mathsf{H}' & = & \{(n_V, sel', n_W) | (n_V, sel', n_W) \in \mathsf{H} \wedge \\
 & & \neg(X = V \wedge sel = sel')\} \\
\mathsf{is}' & = & \begin{cases} \mathsf{is}\backslash\{n_U\} & \text{if } n_U \in \mathsf{is} \wedge \#into(n_U, \mathsf{H}') \leq 1 \wedge \\ & \qquad \neg\exists sel' : (n_\emptyset, sel', n_U) \in \mathsf{H}' \\ \mathsf{is} & \text{otherwise} \end{cases}
\end{array}$$

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

$$\mathsf{is}' = \begin{cases} \mathsf{is} \backslash \{n_U\} & \text{if } n_U \in \mathsf{is} \ \wedge \ \#into(n_U, \mathsf{H}') \leq 1 \ \wedge \\ & \quad \neg \exists sel' : (n_\emptyset, sel', n_U) \in \mathsf{H}' \\ \mathsf{is} & \text{otherwise} \end{cases}$$

We use $\#into(n_U, \mathsf{H}')$ to indicate the number of pointer to n_U in H'.

With the first clause we check if it is possible to remove n_U from is. If the number of pointers to n_U is less then 2 and there is no $(n_\emptyset, sel', n_U)$ triple in H' then we can remove it.

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

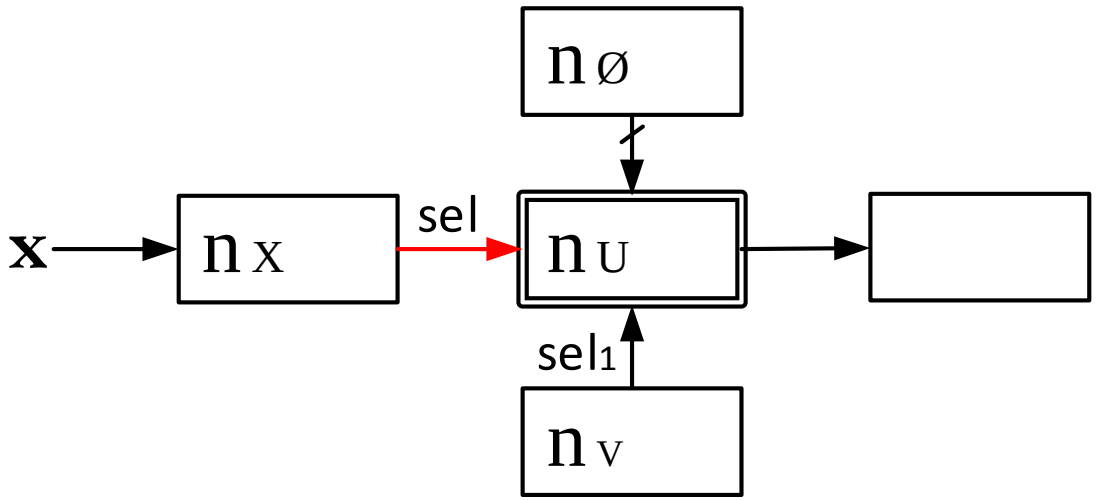
$$\mathsf{is}' = \begin{cases} \mathsf{is} \backslash \{n_U\} & \text{if } n_U \in \mathsf{is} \ \wedge \ \#into(n_U, \mathsf{H}') \leq 1 \ \wedge \\ & \quad \neg \exists sel' : (n_\emptyset, sel', n_U) \in \mathsf{H}' \\ \mathsf{is} & \text{otherwise} \end{cases}$$

We use $\#into(n_U, \mathsf{H}')$ to indicate the number of pointer to n_U in H'.

With the first clause we check if it is possible to remove n_U from is. If the number of pointers to n_U is less then 2 and there is no $(n_\emptyset, sel', n_U)$ triple in H' then we can remove it.

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

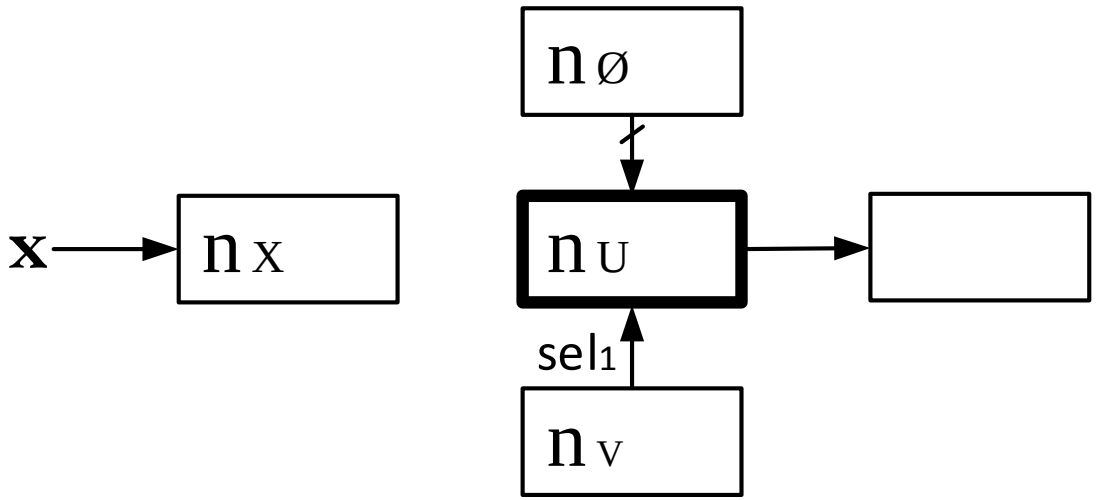
Let's view the effect of $[x.sel := \texttt{nil}]^\ell$ when $\#into(n_U, \mathsf{H}') \leq 1$.



(S, H, is)

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

Let's view the effect of $[x.sel := \texttt{nil}]^\ell$ when $\#into(n_U, \mathsf{H}') \leq 1$.



$(\mathsf{S}', \mathsf{H}', \mathsf{is}')$

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

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

If $x = y$ the assignment is equivalent to:

$$[t := y]^{\ell_1};\ [x.sel := t]^{\ell_2};\ [t := \mathtt{nil}]^{\ell_3};$$

Where t is a **fresh variable** and ℓ_1, ℓ_2 and ℓ_3 are **fresh labels**.

We can obtain the transfer function f^{SA}_{ℓ} as a **composition**:

$$f^{\mathsf{SA}}_{\ell} = f^{\mathsf{SA}}_{\ell_3} \circ f^{\mathsf{SA}}_{\ell_2} \circ f^{\mathsf{SA}}_{\ell_1}$$

So $f^{\mathsf{SA}}_{\ell_1}$ and $f^{\mathsf{SA}}_{\ell_3}$ can be computed using the same pattern as before. We concentrate on $f^{\mathsf{SA}}_{\ell_2}$ (or similar in the case when $x \neq y$).

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

If $x = y$ the assignment is equivalent to:

$$[t := y]^{\ell_1};\ [x.sel := t]^{\ell_2};\ [t := \mathtt{nil}]^{\ell_3};$$

Where t is a **fresh variable** and ℓ_1, ℓ_2 and ℓ_3 are **fresh labels**.

We can obtain the transfer function f^{SA}_ℓ as a **composition**:

$$f^{\mathsf{SA}}_\ell = f^{\mathsf{SA}}_{\ell_3} \circ f^{\mathsf{SA}}_{\ell_2} \circ f^{\mathsf{SA}}_{\ell_1}$$

So $f^{\mathsf{SA}}_{\ell_1}$ and $f^{\mathsf{SA}}_{\ell_3}$ can be computed using the same pattern as before. We concentrate on $f^{\mathsf{SA}}_{\ell_2}$ (or similar in the case when $x \neq y$).

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

Assume $x \neq y$ and let $(\mathsf{S}, \mathsf{H}, \mathsf{is})$ be a compatible shape graph.

In the case where there is no n_X such that $(x, n_X) \in \mathsf{S}$ then the f_ℓ^{SA} is just the identity function.

Consider the case where there is a n_X such that $(x, n_X) \in \mathsf{S}$, but there is no n_Y such that $(y, n_Y) \in \mathsf{S}$. It's the case where y is an integer value, or the `nil` value, it can be treated as $[x.sel := a]^\ell$:

$$\phi_\ell^{\mathsf{SA}}((\mathsf{S}, \mathsf{H}, \mathsf{is})) = \{kill_{x.sel}(\mathsf{S}, \mathsf{H}, \mathsf{is})\}$$

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

Assume $x \neq y$ and let $(\mathsf{S}, \mathsf{H}, \mathsf{is})$ be a compatible shape graph.

In the case where there is no n_X such that $(x, n_X) \in \mathsf{S}$ then the f_ℓ^{SA} is just the identity function.

Consider the case where there is a n_X such that $(x, n_X) \in \mathsf{S}$, but there is no n_Y such that $(y, n_Y) \in \mathsf{S}$. It's the case where y is an integer value, or the `nil` value, it can be treated as $[x.sel := a]^\ell$:

$$\phi_\ell^{\mathsf{SA}}((\mathsf{S}, \mathsf{H}, \mathsf{is})) = \{kill_{x.sel}(\mathsf{S}, \mathsf{H}, \mathsf{is})\}$$

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

Assume $x \neq y$ and let $(\mathsf{S}, \mathsf{H}, \mathsf{is})$ be a compatible shape graph.

In the case where there is no n_X such that $(x, n_X) \in \mathsf{S}$ then the f_ℓ^{SA} is just the identity function.

Consider the case where there is a n_X such that $(x, n_X) \in \mathsf{S}$, but there is no n_Y such that $(y, n_Y) \in \mathsf{S}$. It's the case where y is an integer value, or the `nil` value, it can be treated as $[x.sel := a]^\ell$:

$$\phi_\ell^{\mathsf{SA}}((\mathsf{S}, \mathsf{H}, \mathsf{is})) = \{kill_{x.sel}(\mathsf{S}, \mathsf{H}, \mathsf{is})\}$$

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

In the case where $x \neq y$, $(x, n_X) \in \mathsf{S}$ and $(y, n_Y) \in \mathsf{S}$ we proceed with 2 steps.

1. First we remove the binding for $x.sel$,
2. Then we add the new binding.

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

In the case where $x \neq y$, $(x, n_X) \in \mathsf{S}$ and $(y, n_Y) \in \mathsf{S}$ we proceed with 2 steps.

1. First we remove the binding for $x.sel$,
2. Then we add the new binding.

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

$$\phi^{\mathsf{SA}}_\ell((\mathsf{S},\mathsf{H},\mathsf{is})) = \{(\mathsf{S}'',\mathsf{H}'',\mathsf{is}'')\}$$

where $(\mathsf{S}',\mathsf{H}',\mathsf{is}') = kill_{x.sel}((\mathsf{S},\mathsf{H},\mathsf{is}))$ and

$$\begin{array}{rcl} \mathsf{S}'' & = & \mathsf{S}' \quad (= \mathsf{S}) \\ \mathsf{H}'' & = & \mathsf{H}' \cup \{(n_X, sel, n_Y)\} \\ \mathsf{is}'' & = & \begin{cases} \mathsf{is}' \cup \{n_Y\} & \text{if } \#into(n_Y, \mathsf{H}') \geq 1 \\ \mathsf{is}' & \text{otherwise} \end{cases} \end{array}$$

Note that node n_Y can become shared if we add a new edge pointing to it.

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

$$\phi_\ell^{\mathsf{SA}}((\mathsf{S},\mathsf{H},\mathsf{is})) = \{(\mathsf{S}'',\mathsf{H}'',\mathsf{is}'')\}$$

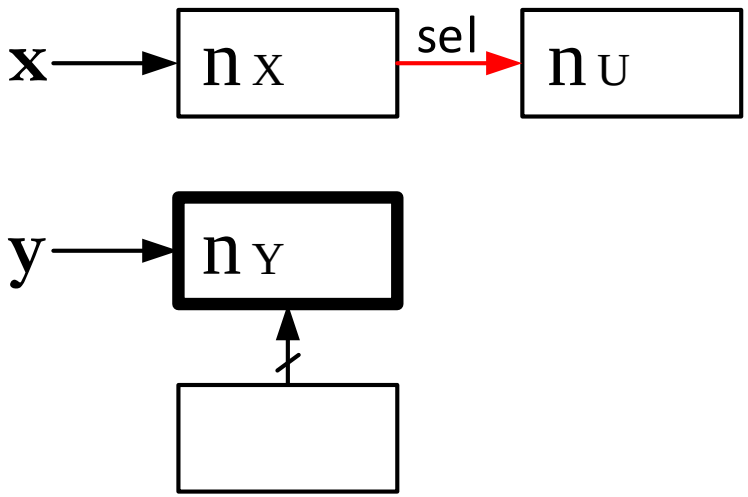
where $(\mathsf{S}',\mathsf{H}',\mathsf{is}') = kill_{x.sel}((\mathsf{S},\mathsf{H},\mathsf{is}))$ and

$$\begin{array}{lcl} \mathsf{S}'' & = & \mathsf{S}' \quad (=\mathsf{S}) \\ \mathsf{H}'' & = & \mathsf{H}' \cup \{(n_X, sel, n_Y)\} \\ \mathsf{is}'' & = & \begin{cases} \mathsf{is}' \cup \{n_Y\} & \text{if } \#into(n_Y,\mathsf{H}') \geq 1 \\ \mathsf{is}' & \text{otherwise} \end{cases} \end{array}$$

Note that node n_Y can become shared if we add a new edge pointing to it.

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

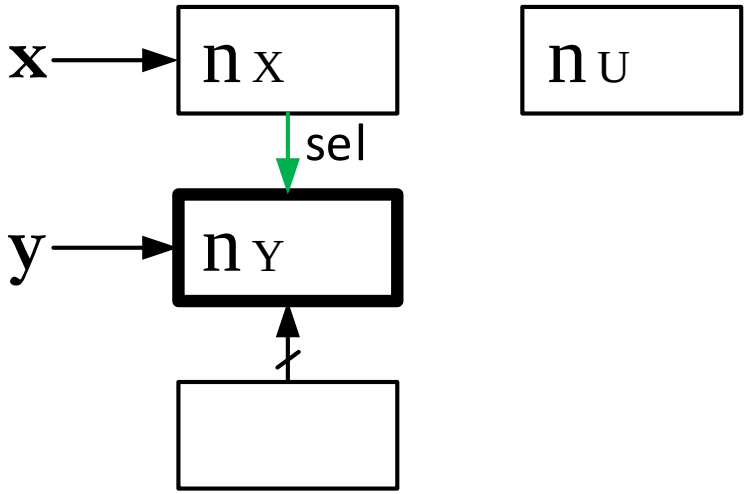
Let's view the effect of $[x.sel := y]^\ell$ when $\#into(n_Y, \mathsf{H}') < 1$.



(S, H, is)

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

Let's view the effect of $[x.sel := y]^\ell$ when $\#into(n_Y, \mathsf{H}') < 1$.



$(\mathsf{S}'', \mathsf{H}'', \mathsf{is}'')$

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

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

The statement is equivalent to:

$$[t := y.sel']^{\ell_1};\ [x.sel := t]^{\ell_2};\ [t := \texttt{nil}]^{\ell_3};$$

Where t is a **fresh variable** and ℓ_1, ℓ_2 and ℓ_3 are **fresh labels**.

We can obtain the transfer function f^{SA}_{ℓ} as a **composition**:

$$f^{\mathsf{SA}}_{\ell} = f^{\mathsf{SA}}_{\ell_3} \circ f^{\mathsf{SA}}_{\ell_2} \circ f^{\mathsf{SA}}_{\ell_1}$$

So $f^{\mathsf{SA}}_{\ell_1}$, $f^{\mathsf{SA}}_{\ell_2}$ and $f^{\mathsf{SA}}_{\ell_3}$ follow the pattern that we have seen before in the other cases.

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

The statement is equivalent to:

$$[t := y.sel']^{\ell_1};\ [x.sel := t]^{\ell_2};\ [t := \texttt{nil}]^{\ell_3};$$

Where t is a **fresh variable** and ℓ_1, ℓ_2 and ℓ_3 are **fresh labels**.

We can obtain the transfer function f^{SA}_ℓ as a **composition**:

$$f^{\mathsf{SA}}_\ell = f^{\mathsf{SA}}_{\ell_3} \circ f^{\mathsf{SA}}_{\ell_2} \circ f^{\mathsf{SA}}_{\ell_1}$$

So $f^{\mathsf{SA}}_{\ell_1}$, $f^{\mathsf{SA}}_{\ell_2}$ and $f^{\mathsf{SA}}_{\ell_3}$ follow the pattern that we have seen before in the other cases.

$$[\texttt{malloc}\ x]^{\ell}$$

$[\texttt{malloc}\ x]^{\ell}$

We remove the binding for x (using $kill_x$) and then we introduce a new (unshared) location pointed by x.

$$\phi^{SA}_{\ell}((S, H, is)) = \{(S' \cup \{(x, n_{\{x\}})\}, H', is')\}$$

where $(S', H', is') = kill_x((S, H, is))$.

$[\texttt{malloc}\ x]^\ell$

We remove the binding for x (using $kill_x$) and then we introduce a new (unshared) location pointed by x.

$$\phi^{\mathsf{SA}}_\ell((\mathsf{S},\mathsf{H},\mathsf{is})) = \{(\mathsf{S}' \cup \{(x, n_{\{x\}})\}, \mathsf{H}', \mathsf{is}')\}$$

where $(\mathsf{S}', \mathsf{H}', \mathsf{is}') = kill_x((\mathsf{S},\mathsf{H},\mathsf{is}))$.

$[\texttt{malloc}\ x.sel]^{\ell}$

$[\texttt{malloc}\ x.sel]^{\ell}$

The statement is equivalent to:

$$[\texttt{malloc}\ t]^{\ell_1};\ [x.sel := t]^{\ell_2};\ [t := \texttt{nil}]^{\ell_3};$$

Where t is a **fresh variable** and ℓ_1, ℓ_2 and ℓ_3 are **fresh labels**.

We can obtain the transfer function f^{SA}_{ℓ} as a **composition**:

$$f^{\mathsf{SA}}_{\ell} = f^{\mathsf{SA}}_{\ell_3} \circ f^{\mathsf{SA}}_{\ell_2} \circ f^{\mathsf{SA}}_{\ell_1}$$

So $f^{\mathsf{SA}}_{\ell_1}$, $f^{\mathsf{SA}}_{\ell_2}$ and $f^{\mathsf{SA}}_{\ell_3}$ follow the pattern that we have seen before in the other cases.

$[\texttt{malloc}\ x.sel]^{\ell}$

The statement is equivalent to:

$$[\texttt{malloc}\ t]^{\ell_1};\ [x.sel := t]^{\ell_2};\ [t := \texttt{nil}]^{\ell_3};$$

Where t is a **fresh variable** and ℓ_1, ℓ_2 and ℓ_3 are **fresh labels**.

We can obtain the transfer function f^{SA}_{ℓ} as a **composition**:

$$f^{\mathsf{SA}}_{\ell} = f^{\mathsf{SA}}_{\ell_3} \circ f^{\mathsf{SA}}_{\ell_2} \circ f^{\mathsf{SA}}_{\ell_1}$$

So $f^{\mathsf{SA}}_{\ell_1}$, $f^{\mathsf{SA}}_{\ell_2}$ and $f^{\mathsf{SA}}_{\ell_3}$ follow the pattern that we have seen before in the other cases.