Structural Operational Semantics (SOS) for SIMP

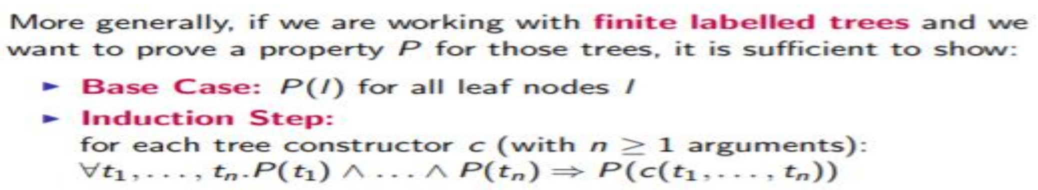
The structural operational semantics of SIMP is an alternative to the ABSTRACT MACHINE.

*Structural induction – **To prove that a property P holds for every list, it is sufficient to prove***

- **Base case:** P(nil) – the case when the list is empty

If P(l) hold will p(l) + 1 element also hold

- **Induction Step:** P(l) => P (cons (h, l)) for all h, I. where H – **HEAD,** l – **TAIL**, cons – **LIST CONSTRUCTOR ADDS ELEMENT TO THE LIST.** *IN THIS EXAMPLE CONS (H, L) puts element H as a HEAD of list L.*



More generally, if we are working with **finite labelled trees** and we want to prove a property P for those trees, it is sufficient to show:

- **Base Case:** $P(l)$ for all leaf nodes l
- **Induction Step:**
  for each tree constructor c (with $n \geq 1$ arguments):
  $\forall t_1, \ldots, t_n . P(t_1) \wedge \ldots \wedge P(t_n) \Rightarrow P(c(t_1, \ldots, t_n))$

This means that the property holds for all subtrees from t1 to tn

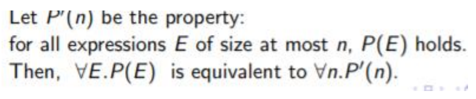
Example

To prove that a property *P* holds for all integer expressions in **SIMP**:

Want to prove property P for N.

1. ***Base case:***
   a. Prove P(n) for all n belongs to Z – Does it give a result for N, is there a rule in a abstract machine to give me N
   b. Prove P (! l) for all locations l – is there a rule to accessing location L assuming l is valid address in memory
2. ***Induction Step:***
   a. For all integer expression E, E' and operators op: prove that P(E) and P(E') implies P (E op E'). – We can compute E Op E' if we can compute P (E) and P (E')

The principle of structural induction can be justified by the principle of mathematical induction. This is because we work with *finite* trees, so we can understand structural induction as induction on the size of the tree.

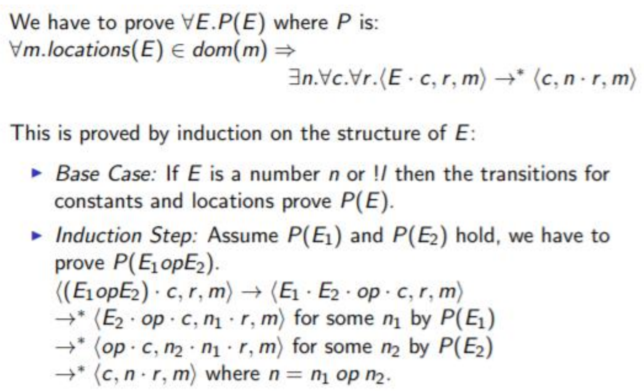


Let $P'(n)$ be the property:
for all expressions E of size at most n, $P(E)$ holds.
Then, $\forall E.P(E)$ is equivalent to $\forall n.P'(n)$.

Application of the Structural Induction Principle

The semantics of SIMP guarantees that for any integer E appearing in a program working on a memory m, if E uses locations that are defined in m then the value of E is defined.

So, there is a configuration < E * c, r, m > where c is any arbitrary control stack is also arbitrary, there is a configuration <c, n * r, m> such that <E * c, r, m> ->* <c, n * r, m>



We have to prove $\forall E.P(E)$ where P is:
$\forall m.locations(E) \in dom(m) \Rightarrow$
$\exists n.\forall c.\forall r.\langle E \cdot c, r, m\rangle \to^* \langle c, n \cdot r, m\rangle$

This is proved by induction on the structure of E:

- *Base Case:* If E is a number n or $!l$ then the transitions for constants and locations prove $P(E)$.
- *Induction Step:* Assume $P(E_1)$ and $P(E_2)$ hold, we have to prove $P(E_1 op E_2)$.
  $\langle (E_1 op E_2) \cdot c, r, m\rangle \to \langle E_1 \cdot E_2 \cdot op \cdot c, r, m\rangle$
  $\to^* \langle E_2 \cdot op \cdot c, n_1 \cdot r, m\rangle$ for some n_1 by $P(E_1)$
  $\to^* \langle op \cdot c, n_2 \cdot n_1 \cdot r, m\rangle$ for some n_2 by $P(E_2)$
  $\to^* \langle c, n \cdot r, m\rangle$ where $n = n_1\ op\ n_2$.

Inductive Definition

We can also use induction to define subsets of a given set T. We will write inductive definition using **axioms** (representing the base case and **rules** (representing the induction step)

DEFINITION:

- An **axiom** is an element of **T.**
- A **rule** is a pair **(H, c)** where
  - **H** is a non-empty subset of **T**, called the **hypotheses** of the rule
  - **C** is an element of **T,** called the **conclusion** of the rule.

*The subset **I** of **T*** inductively defined by a collection of axioms *A* and rules *R* consists of those **t** ∈ **T** such that

- t ∈ A or
- there are t1,…,tn ∈ *I* and a rule **(H, c)** such that **H** = {t1,…tn} and **t** = **c**

To show that an element t of T is in I it is sufficient to show that t is an axiom, or that there is a *proof*:

$$\begin{tikzcd}[column sep=small] & & t \arrow[dl, dash] \arrow[dr, dash] & & \\ & t_1 \arrow[dl, dash] \arrow[d, dash] & \cdots & t_n \arrow[d, dash] \arrow[dr, dash] & \\ t_{11}\cdots & t_{1m_1} & & t_{n1}\cdots & t_{nm_n} \\ \vdots & & & \vdots & \end{tikzcd}$$

where the leaves are axioms and for each non-leaf node t_i there is a rule $(\{t_{i1}, \ldots, t_{im_i}\}, t_i)$.
This kind of proof is usually written:

$$\cfrac{\cfrac{\cfrac{\vdots}{t_{11}} \quad \cdots \quad \cfrac{\vdots}{t_{1m_1}}}{t_1} \qquad \cdots \qquad \cfrac{\vdots}{t_n}}{t}$$

Building a Inductive Tree means that we put the rule to be a root of a tree and then we evaluate the axioms/rules in order to get to leaves which are axioms.

Example of Inducive Definition

1. **Natural Numbers:**
   Axiom:
   $$0$$
   Rule:
   $$\frac{n}{n+1}$$
2. **Evaluation relation for integer expressions in SIMP:**
   Notation:
   $(E, m) \Downarrow n$ means that E evaluates to n in the state m.
   Axioms:
   $(n, m) \Downarrow n$ for all integer numbers n
   $(!l, m) \Downarrow n$ if $l \in dom(m)$ and $m(l) = n$
   Rule:
   $$\frac{(E_1, m) \Downarrow n_1 \quad (E_2, m) \Downarrow n_2}{(E_1\ op\ E_2, m) \Downarrow n} \text{ if } n = n_1\ op\ n_2$$

From the top axiom we can construct rule like this.

Assume E1 evaluates to n1, Assume E2 evaluates to n2, Assume E1 op E2 evaluates to n.

*And this example is an inductive definition, giving us the same information that the transitions we gave for abstract machine gave us for the expressions. But the difference is that we didn't need to explain Stack, push, pop etc… With inducvite definition it took 4 axioms. This is a way of operational semantics REPRESENTED IN A STRUCTURAL WAY therefore and that is why its called **STRUCTURAL OPERATIONAL SEMANTICS** because the Abstract machine transition system is a OPERATIONAL SEMANTICS and this INDUCTIVE METHOD IS A STRUCTURAL WAY.*

For each abstract syntax tree, we will give axioms or rules.

Rule Induction

We can write a principle of induction for this kind of definitions, called rule induction.

PRINCIPLE OF RULE INDUCTION:

Let *I* be a set defined by induction with axioms and rules (*A, R*). To show tat *P(i)* holds for all ***i* ∈ *I***, it is sufficient PROVE:

- *Base Case:* $\forall a \in A. P(a)$
- *Induction Step:* $\forall(\{h_1, \dots h_n\}, c) \in R.\ P(h_1) \wedge \dots \wedge P(h_n) \Rightarrow P(c)$.

This shows that if all hypotheses hold for h1, …, hn then it C must also hold.

Example: Each integer expression in SIMP has a unique value under the evaluation relation ⇓.

Basis: Trivial, since number have unique values,

Induction Step: For non-atomic expression, we remark that the value of an arithmetic expression is uniquely determined by the values of its arguments which are unique by the induction hypotheses.

Special Principle of Rule Induction

*In some cases, we are only interested in proving a property for a subset J of the inductive set **I** defined by **(A, R)** Then we can use a special case of the principle of rule induction, which says that to prove that a property Q(x) holds for all the elements of* J, it is sufficient to show

- *Basis:* $\forall a \in (A \cap J). Q(a)$
- *Induction Step:* for all $(H, c) \in R$ such that $c \in J$,

  $(\forall h \in H \cap J. Q(h)) \Rightarrow Q(c)$.

Structural Operational Semantics(SOS)

The structural operational semantics of SIMP is an alternative to the abstract syntax where everything is at low level, (abstract syntax – operational semantics / induction – structural operational semantics)

There are two styles of structural operational semantics:

- ***Small-step semantics:*** defined using a *reduction* relation – We are going to define transition system where each transition is a small step of computation

- ***Big*-step semantics:** defined using an evaluation relation – We are going to define transition system that in the definition of semantics we will specify the FINAL result.

Small-Step Semantics

We always must define the memory state at the start because it is an imperative language, so we always work with configuration that will have a memory state. [ ] update memory ( ) gets from memory

We define a transition system with configurations: **<P, s>**

Where *P* is a SIMP program and *s* is a store (memory) represented by a partial function from locations to integers.

Notation: $s[l \mapsto n]$ denotes the function s' that coincides with s except that it associates to l the value n. More precisely:
$s[l \mapsto n](l) = n$
$s[l \mapsto n](l') = s(l')$ if $l \neq l'$
The transition relation is inductively defined by the axioms and rules:

If we update the value n at location L and then we ask for value of L it will display N.

Small Step Semantics for Expressions

This can be read from bottom to top, to evaluate the program, we must apply rules given, we achieve the same effect as abstract machine.

$$\frac{}{\langle !l, s\rangle \to \langle n, s\rangle,\ \text{if } s(l) = n}\ (\text{var}) \qquad \frac{}{\langle n_1\ op\ n_2, s\rangle \to \langle n, s\rangle,\ \text{if } n = (n_1\ op\ n_2)}\ (\text{op})$$

$$\frac{}{\langle n_1\ bop\ n_2, s\rangle \to \langle b, s\rangle,\ \text{if } b = (n_1\ bop\ n_2)}\ (\text{bop})$$

$$\frac{\langle E_1, s\rangle \to \langle E_1', s'\rangle}{\langle E_1 op E_2, s\rangle \to \langle E_1' op E_2, s'\rangle}\ (\text{op}_L) \qquad \frac{\langle E_2, s\rangle \to \langle E_2', s'\rangle}{\langle E_1 op E_2, s\rangle \to \langle E_1 op E_2', s'\rangle}\ (\text{op}_R)$$

$$\frac{\langle E_1, s\rangle \to \langle E_1', s'\rangle}{\langle E_1 bop E_2, s\rangle \to \langle E_1' bop E_2, s'\rangle}\ (\text{bop}_L) \qquad \frac{\langle E_2, s\rangle \to \langle E_2', s'\rangle}{\langle E_1 bop E_2, s\rangle \to \langle E_1 bop E_2', s'\rangle}\ (\text{bop}_R)$$

$$\frac{}{\langle b_1 \wedge b_2, s\rangle \to \langle b, s\rangle,\ \text{if } b = (b_1\ and\ b_2)}\ (\text{and})$$

$$\frac{}{\langle \neg b, s\rangle \to \langle b', s\rangle,\ \text{if } b' = not\ b}\ (\text{not}) \qquad \frac{\langle B_1, s\rangle \to \langle B_1', s'\rangle}{\langle \neg B_1, s\rangle \to \langle \neg B_1', s'\rangle}\ (\text{notArg})$$

$$\frac{\langle B_1, s\rangle \to \langle B_1', s'\rangle}{\langle B_1 \wedge B_2, s\rangle \to \langle B_1' \wedge B_2, s'\rangle}\ (\text{and}_L) \qquad \frac{\langle B_2, s\rangle \to \langle B_2', s'\rangle}{\langle B_1 \wedge B_2, s\rangle \to \langle B_1 \wedge B_2', s'\rangle}\ (\text{and}_R)$$

Axioms don't have anything on top of the horizontal line.

Small-Step Semantics for Commands

For example, the if, SIMP must evaluate B which is the condition if this can be evaluated to B'# then we can use axioms provided for both cases TRUE and FALSE.

Sequence can be only read from left to right.

$$\frac{\langle E, s\rangle \to \langle E', s'\rangle}{\langle l := E, s\rangle \to \langle l := E', s'\rangle}\ (:=_R) \qquad \frac{}{\langle l := n, s\rangle \to \langle skip, s[l \mapsto n]\rangle}\ (:=)$$

$$\frac{\langle C_1, s\rangle \to \langle C_1', s'\rangle}{\langle C_1; C_2, s\rangle \to \langle C_1'; C_2, s'\rangle}\ (\text{seq}) \qquad \frac{}{\langle skip; C, s\rangle \to \langle C, s\rangle}\ (\text{skip})$$

$$\frac{\langle B, s\rangle \to \langle B', s'\rangle}{\langle if\ B\ then\ C_1\ else\ C_2, s\rangle \to \langle if\ B'\ then\ C_1\ else\ C_2, s'\rangle}\ (\text{if})$$

$$\frac{}{\langle if\ True\ then\ C_1\ else\ C_2, s\rangle \to \langle C_1, s\rangle}\ (\text{if}_T)$$

$$\frac{}{\langle if\ False\ then\ C_1\ else\ C_2, s\rangle \to \langle C_2, s\rangle}\ (\text{if}_F)$$

$$\frac{}{\langle while\ B\ do\ C, s\rangle \to \langle if\ B\ then\ (C; while\ B\ do\ C)\ else\ skip, s\rangle}\ (\text{while})$$

REMARKS:

There is no axiom or rule for programs of the form:

- <n, s> where n is an integer
- <b, s> where b is a Boolean
- <!l, s> where l not∈ dom(s)
- <skip, s>

There are *terminal configurations*. In the case of <!l, s> where l is not valid address we say that the program is *blocked*.

Small steps that could be performed on machine as it is a low level.

Example:
Let P be the program z:= !x; x:= !y; y:= !z and s a state such that $s(z) = 0$, $s(x) = 1$, $s(y) = 2$.

There is a sequence of transitions:

$$\begin{aligned}
&\langle P, s\rangle \to \langle z := 1; (x := !y; y := !z), s\rangle \\
&\to \langle skip; (x := !y; y := !z), s[z \mapsto 1]\rangle \\
&\to \langle x := !y; y := !z, s[z \mapsto 1]\rangle \\
&\to \langle x := 2; y := !z, s[z \mapsto 1]\rangle \\
&\to \langle skip; y := !z, s[z \mapsto 1, x \mapsto 2]\rangle \\
&\to \langle y := !z, s[z \mapsto 1, x \mapsto 2]\rangle \\
&\to \langle y := 1, s[z \mapsto 1, x \mapsto 2]\rangle \\
&\to \langle skip, s[z \mapsto 1, x \mapsto 2, y \mapsto 1]\rangle
\end{aligned}$$

Comparison with the Abstract Machine

1. Each transition here is doing a part of the computation that leads to the result, whereas some of the transitions of the machine were only manipulating syntax.
2. On the other hand, to show that a sequence of reductions is valid we need a proof. For example:
   $\langle \text{if } !l > 0 \text{ then } (C'; C) \text{ else skip}, s\rangle \to$
   $\langle \text{if } 4 > 0 \text{ then } (C'; C) \text{ else skip}, s\rangle$
   can be proved if $s(l) = 4$.
   Proof:

$$\cfrac{\cfrac{\cfrac{}{\langle !l, s\rangle \to \langle 4, s\rangle}\ l \in dom(s), s(l) = 4}{\langle !l > 0, s\rangle \to \langle 4 > 0, s\rangle}}{\langle \text{if } !l > 0 \text{ then } (C'; C) \text{ else skip}, s\rangle \to \langle \text{if } 4 > 0 \text{ then } (C'; C) \text{ else skip}, s\rangle}$$

Evaluation Semantics for SIMP: Big-Step Semantics

REMARK:

The previous system is *deterministic*. Given a configuration <**P, s**> there is a unique sequence of transitions from <**P, s>** with maximal length. This called the evaluation sequence for <**P, s>**

- The evaluation sequence for <**P, s**> may be finite or infinite

Big-Step Semantics

We will classify evaluation sequences in three categories:

- **Terminating:** if the sequence eventually reaches a terminal non-blocked configuration that is, a configuration of the form:
  - **<n, s>** where n is an integer or
  - **<b, s>** where b is a Boolean
  - **<skip, s>**
- **Blocked:** If the sequence eventually reaches a blocked configuration (that is a configuration of the form **<!l, s>** where l NOT ∈ **dom(s)**
- **Divergent:** if the sequence is infinite.

EXAMPLE:

- <while True do skip, s> is divergent.
- <if !x = 0 then skip else skip, s> is stuck if dom(s) does not contain x
- <if 4 = 0 then skip else skip, s> is terminating

Transition System

The following system defines all the terminating evaluation sequences: It defines by induction the binary relation **<P, s> ->* <P', s'>** such that **<P', s'>** is terminal.

The usual notation for this is **<P,s> ⇓ <P', s'>**

Double arrow down is a relation between <P, s> and last thing we get when we do small step, assuming the program terminates.

In other words: **<P,s> ⇓ <P', s'> if <P,s> ->* <P', s'>** where **<P', s'>** is terminal.

Week 4 Week 5

Big-Step Semantics

$$\frac{}{\langle c, s\rangle \Downarrow \langle c, s\rangle,\ \ if\ c \in Z \cup \{True, False\}}\ (\text{const})$$

c- any number or true or false, this is an axiom which can't be simplified further, so if you have a number or Boolean it evaluates to a number or a Boolean, result done.

$$\frac{}{\langle !l, s\rangle \Downarrow \langle n, s\rangle,\ \ if\ s(l) = n}\ (\text{var})$$

!l – we want access the content at address l, we don't update memory we just read. If l is not valid address there is no value that we can associate with it.

$$\frac{\langle B_1, s\rangle \Downarrow \langle b_1, s'\rangle \quad \langle B_2, s'\rangle \Downarrow \langle b_2, s''\rangle}{\langle B_1 \wedge B_2, s\rangle \Downarrow \langle b, s''\rangle,\ \ if\ b = b_1\ and\ b_2}\ (\text{and})$$

B1 ^ B2 if **B1** can be evaluated to Boolean value and **B2** can be evaluated value, therefore by induction **B** value can be evaluated from **B1 ^ B2** if **B1 = B1** and **B2**

$$\frac{\langle B_1, s\rangle \Downarrow \langle b_1, s'\rangle}{\langle \neg B_1, s\rangle \Downarrow \langle b, s''\rangle,\ \ if\ b = not\ b_1}\ (\text{not})$$

$$\frac{\langle E_1, s\rangle \Downarrow \langle n_1, s'\rangle \quad \langle E_2, s'\rangle \Downarrow \langle n_2, s''\rangle}{\langle E_1\ op\ E_2, s\rangle \Downarrow \langle n, s''\rangle,\ \ if\ n = n_1\ op\ n_2}\ (\text{op})$$

$$\frac{\langle E_1, s\rangle \Downarrow \langle n_1, s'\rangle \quad \langle E_2, s'\rangle \Downarrow \langle n_2, s''\rangle}{\langle E_1\ bop\ E_2, s\rangle \Downarrow \langle b, s''\rangle,\ \ if\ b = n_1\ bop\ n_2}\ (\text{bop})$$

$$\frac{}{\langle skip, s\rangle \Downarrow \langle skip, s\rangle}\ (\text{skip}) \qquad \frac{\langle E, s\rangle \Downarrow \langle n, s'\rangle}{\langle l := E, s\rangle \Downarrow \langle skip, s'[l \mapsto n]\rangle}\ (:=)$$

$$\frac{\langle C_1, s\rangle \Downarrow \langle skip, s'\rangle \quad \langle C_2, s'\rangle \Downarrow \langle skip, s''\rangle}{\langle C_1; C_2, s\rangle \Downarrow \langle skip, s''\rangle}\ (\text{seq})$$

$$\frac{\langle B, s\rangle \Downarrow \langle True, s'\rangle \quad \langle C_1, s'\rangle \Downarrow \langle skip, s''\rangle}{\langle if\ B\ then\ C_1\ else\ C_2, s\rangle \Downarrow \langle skip, s''\rangle}\ (\text{if}_T)$$

$$\frac{\langle B, s\rangle \Downarrow \langle False, s'\rangle \quad \langle C_2, s'\rangle \Downarrow \langle skip, s''\rangle}{\langle if\ B\ then\ C_1\ else\ C_2, s\rangle \Downarrow \langle skip, s''\rangle}\ (\text{if}_F)$$

$$\frac{\langle B, s\rangle \Downarrow \langle False, s'\rangle}{\langle while\ B\ do\ C, s\rangle \Downarrow \langle skip, s'\rangle}\ (\text{while}_F)$$

$$\frac{, s\rangle \Downarrow \langle True, s'\rangle \quad \langle C, s'\rangle \Downarrow \langle skip, s''\rangle \quad \langle while\ B\ do\ C, s''\rangle \Downarrow \langle skip, s'''\rangle}{\langle while\ B\ do\ C, s\rangle \Downarrow \langle skip, s'''\rangle}\ (\text{while}_T$$

To start sequence from the other side we can just put C2 in the S.

Big-Step Semantics Example

Consider the program P: $(z :=!x; x :=!y); y :=!z$
and a state s such that $s(z) = 0$, $s(x) = 1$, $s(y) = 2$.
We can prove that $P \Downarrow \langle skip, s'\rangle$ where $s'(z) = 1, s'(x) = 2, s'(y) = 1$.
First notice that:

$$\frac{\dfrac{}{\langle !x, s\rangle \Downarrow \langle 1, s\rangle}}{\langle z :=!x, s\rangle \Downarrow \langle skip, s[z \mapsto 1]\rangle} \quad \frac{\dfrac{}{\langle !y, s[z \mapsto 1]\rangle \Downarrow \langle 2, s[z \mapsto 1]\rangle}}{\langle x :=!y, s[z \mapsto 1]\rangle \Downarrow \langle skip, s[\begin{smallmatrix} z \mapsto 1 \\ x \mapsto 2 \end{smallmatrix}]\rangle}$$
$$\overline{\langle z :=!x; x :=!y, s\rangle \Downarrow \langle skip, s[\begin{smallmatrix} z \mapsto 1 \\ x \mapsto 2 \end{smallmatrix}]\rangle}$$

Using this we can prove:

$$\frac{\langle z :=!x; x :=!y, s\rangle \Downarrow \langle skip, s[\begin{smallmatrix} z \mapsto 1 \\ x \mapsto 2 \end{smallmatrix}]\rangle \quad \dfrac{\dfrac{}{\langle !z, s[\begin{smallmatrix} z \mapsto 1 \\ x \mapsto 2 \end{smallmatrix}]\rangle \Downarrow \langle 1, s[\begin{smallmatrix} z \mapsto 1 \\ x \mapsto 2 \end{smallmatrix}]\rangle}}{\langle y :=!z, s[\begin{smallmatrix} z \mapsto 1 \\ x \mapsto 2 \end{smallmatrix}]\rangle \Downarrow \langle skip, s[\begin{smallmatrix} z \mapsto 1 \\ x \mapsto 2 \\ y \mapsto 1 \end{smallmatrix}]\rangle}}{\langle P, s\rangle \Downarrow \langle skip, s'\rangle}$$

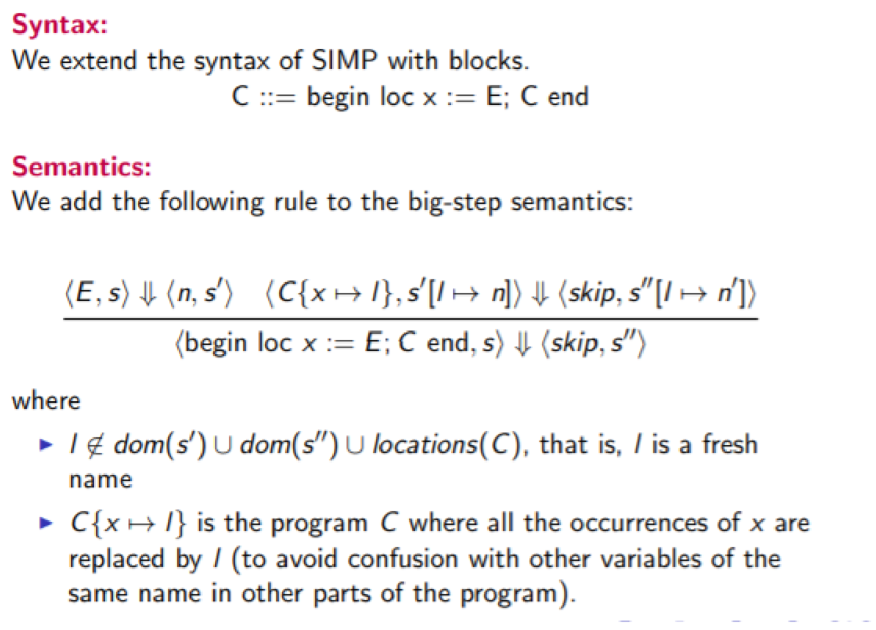
We don't show intermediate evaluations we define that the program P in state s will terminated with this result.

Adding Variable Declarations to SIMP

We can add local declarations to SIMP by using a local state in which the scope of a newly created location corresponds precisely with the block where the location is created and initialised (static scope)

For this we will need a stack-based implementation of the state

Run C will local memory state



Syntax:
We extend the syntax of SIMP with blocks.

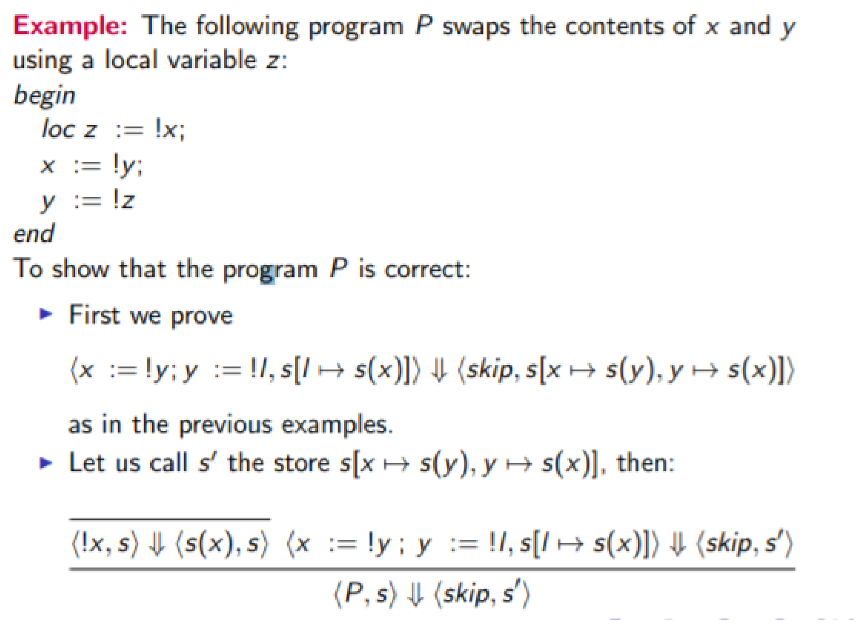
C ::= begin loc x := E; C end

Semantics:
We add the following rule to the big-step semantics:

$$\frac{\langle E, s\rangle \Downarrow \langle n, s'\rangle \quad \langle C\{x \mapsto l\}, s'[l \mapsto n]\rangle \Downarrow \langle skip, s''[l \mapsto n']\rangle}{\langle \text{begin loc } x := E; C \text{ end}, s\rangle \Downarrow \langle skip, s''\rangle}$$

where

- $l \notin dom(s') \cup dom(s'') \cup locations(C)$, that is, l is a fresh name
- $C\{x \mapsto l\}$ is the program C where all the occurrences of x are replaced by l (to avoid confusion with other variables of the same name in other parts of the program).



Example: The following program P swaps the contents of x and y using a local variable z:

```
begin
  loc z  := !x;
  x  := !y;
  y  := !z
end
```

To show that the program P is correct:

- First we prove

  $\langle x := !y; y := !l, s[l \mapsto s(x)]\rangle \Downarrow \langle skip, s[x \mapsto s(y), y \mapsto s(x)]\rangle$

  as in the previous examples.
- Let us call s' the store $s[x \mapsto s(y), y \mapsto s(x)]$, then:

$$\frac{\dfrac{}{\langle !x, s\rangle \Downarrow \langle s(x), s\rangle} \quad \langle x := !y ; y := !l, s[l \mapsto s(x)]\rangle \Downarrow \langle skip, s'\rangle}{\langle P, s\rangle \Downarrow \langle skip, s'\rangle}$$