Imperative Languages

Structural Operational Semantics

Maribel Fernández

Department of Informatics
King's College London

Alternative ways of defining operational semantics

In this lecture we will give an alternative definition of the operational semantics of SIMP, where the execution of programs is defined by induction.
Aims of this lecture:

- Understand the use of induction to specify transition systems and prove properties of programs.
- Give alternative definitions of SIMP's operational semantics using induction.

Structure

This lecture has three parts:

Revision:Induction

Small-Step Operational Semantics for SIMP

Big-Step Semantics for SIMP

Revision:Induction

Revision: Induction

Aims of this lecture:

- Understand the use of induction on natural numbers as a proof method.
- Generalise the principle of induction to work directly on lists, trees, etc.
- Use axioms and rules to define inductive sets, and prove properties by rule induction.

The Principle of Mathematical Induction

For any property $P(n)$ of natural numbers
(i.e. $n \in \mathcal{N} = \{0, 1, 2, \ldots\}$)
to prove $\forall n \in \mathcal{N}.P(n)$ it is sufficient to show:

- **Base Case:** $P(0)$
- **Induction Step:** $\forall n \in \mathcal{N}.P(n) \Rightarrow P(n+1)$

Example:

We prove that for all natural numbers n:

$$\sum_{i=1}^{n}(2i-1) = n^2$$

Base: $0 = 0^2$

Induction Step: Assume $\sum_{i=1}^{n}(2i-1) = n^2$.

$$\sum_{i=1}^{n+1}(2i-1) = \sum_{i=1}^{n}(2i-1) + 2(n+1) - 1 = n^2 + 2n + 1 = (n+1)^2$$

Structural Induction

We denote an empty list by nil, and a non-empty list by $cons(h, l)$ where h is the head element and l is the tail of the list.

To prove that a property P holds for every list, it is sufficient to prove:

- **Base Case:** $P(nil)$,
- **Induction Step:** $P(l) \Rightarrow P(cons(h, l))$ for all h, l.

Structural Induction

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):
  $\forall t_1, \ldots, t_n . P(t_1) \wedge \ldots \wedge P(t_n) \Rightarrow P(c(t_1, \ldots, t_n))$

Structural Induction

Example:

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

1. *Base Case:*
   - Prove $P(n)$ for all $n \in Z$
   - Prove $P(!l)$ for all locations l.
2. *Induction Step:*
   For all integer expressions E, E' and operators op:
   Prove that $P(E)$ and $P(E')$ implies $P(E\ op\ E')$.

Remark

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.

Example:
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

We will prove the following property:
The semantics of SIMP guarantees that for any integer expression 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.

In other words:
For any configuration $\langle E \cdot c, r, m \rangle$ where c is any arbitrary control stack and r is also arbitrary, there is a configuration $\langle c, n \cdot r, m \rangle$ such that $\langle E \cdot c, r, m \rangle \rightarrow^* \langle c, n \cdot r, m \rangle$.

Application of the Structural Induction Principle

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

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 $P(E_1 op E_2)$.
  $\langle (E_1 op E_2) \cdot c, r, m\rangle \rightarrow \langle E_1 \cdot E_2 \cdot op \cdot c, r, m\rangle$
  $\rightarrow^* \langle E_2 \cdot op \cdot c, n_1 \cdot r, m\rangle$ for some n_1 by $P(E_1)$
  $\rightarrow^* \langle op \cdot c, n_2 \cdot n_1 \cdot r, m\rangle$ for some n_2 by $P(E_2)$
  $\rightarrow^* \langle c, n \cdot r, m\rangle$ where $n = n_1\ op\ n_2$.

Inductive Definitions

We can also use induction to define subsets of a given set T. We will write inductive definitions 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

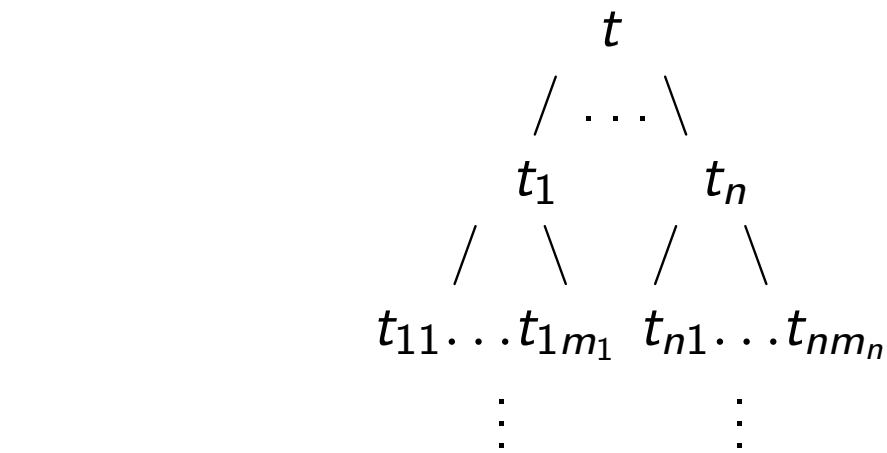
- H is a non-empty subset of T, called the **hypotheses** of the rule
- c is an element of T, called the **conclusion** of the rule.

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

- $t \in A$, or
- there are $t_1, \ldots, t_n \in I$ and a rule (H, c) such that $H = \{t_1, \ldots, t_n\}$ and $t = c$.

Inductive Definitions

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*:



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:

$$\dfrac{\dfrac{\vdots}{t_{11}} \quad \ldots \quad \dfrac{\vdots}{t_{1m_1}} \qquad \ldots \qquad \dfrac{\vdots}{t_n}}{t}$$

Examples of Inductive Definitions

1. **Natural Numbers:**
   Axiom: 0
   Rule: $(\{n\}, n+1)$
   This is usually written:

$$\frac{n}{n+1}$$

2. **Evaluation relation for integer expressions in** SIMP**:**
   Notation:
   $(E, m) \Downarrow n$ means that E evaluates to n in the state m.
   Axioms:

$$(n, m) \Downarrow n \qquad \text{for all integer numbers } n$$
$$(!l, m) \Downarrow n \qquad \text{if } l \in dom(m) \text{ and } m(l) = n$$

   Rule:

$$\frac{(E_1, m) \Downarrow n_1 \quad (E_2, m) \Downarrow n_2}{(E_1 \; op \; E_2, m) \Downarrow n} \text{ if } n = n_1 \; op \; n_2$$

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 that $P(i)$ holds for all $i \in I$, it is sufficient to prove:

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

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

Basis: Trivial, since numbers have unique values.

Induction Step: For non-atomic expressions, 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 $c \in J$,

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

Summary

We have reviewed the principle of induction, including two variants: structural induction and rule induction.

For more details on these topics, see Chapter 2 in the textbook.

In the next part of this lecture we will give an alternative definition of the operational semantics of SIMP, where the execution of programs is defined by induction.
This is called Structural Operational semantics.
There are two styles of structural operational semantics:

- **Small-step semantics:** defined using a *reduction* relation.
- **Big-step semantics:** defined using an *evaluation* relation.

Small-Step Operational Semantics for SIMP

Small-Step Semantics for SIMP: Configurations

We define a transition system with **configurations**

$$\langle P, s \rangle$$

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 $l \neq l'$

Small-Step Semantics for SIMP: Transition Relation

The **transition relation $\rightarrow$ on configurations** is inductively defined by the axioms and rules (below).

- The expression E has the value n in the memory state s if $\langle E, s\rangle \rightarrow^* \langle n, s'\rangle$.
- The command C has a successful execution in the memory state s if $\langle C, s\rangle \rightarrow^* \langle \mathit{skip}, s'\rangle$.
  We say that C in s produces s'.

Small-Step Semantics For Expressions:

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

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

$$\frac{\langle E_1, s\rangle \to \langle E_1', s'\rangle}{\langle E_1 op E_2, s\rangle \to \langle E_1' op E_2, s'\rangle}\ (\text{op}_\text{L}) \qquad \frac{\langle E_2, s\rangle \to \langle E_2', s'\rangle}{\langle n_1 op E_2, s\rangle \to \langle n_1 op E_2', s'\rangle}\ (\text{op}_\text{R})$$

$$\frac{\langle E_1, s\rangle \to \langle E_1', s'\rangle}{\langle E_1 bop E_2, s\rangle \to \langle E_1' bop E_2, s'\rangle}\ (\text{bop}_\text{L}) \qquad \frac{\langle E_2, s\rangle \to \langle E_2', s'\rangle}{\langle n_1 bop E_2, s\rangle \to \langle n_1 bop E_2', s'\rangle}\ (\text{bop}_\text{R})$$

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

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

$$\frac{\langle B_1, s\rangle \to \langle B_1', s'\rangle}{\langle B_1 \wedge B_2, s\rangle \to \langle B_1' \wedge B_2, s'\rangle}\ (\text{and}_\text{L}) \qquad \frac{\langle B_2, s\rangle \to \langle B_2', s'\rangle}{\langle b_1 \wedge B_2, s\rangle \to \langle b_1 \wedge B_2', s'\rangle}\ (\text{and}_\text{R})$$

Small-Step Semantics: Example

Let s be a state such that $s(z) = 0$ and let $E = !z + 1$.
The expression E evaluates to 1 in s, since

$$\langle !z + 1, s\rangle \to \langle 0 + 1, s\rangle \to \langle 1, s\rangle$$

This is shown as follows:
First step:

$$\dfrac{\dfrac{}{\langle !z, s\rangle \to \langle 0, s\rangle}\,(\text{var})}{\langle !z + 1, s\rangle \to \langle 0 + 1, s\rangle}\,(\text{op}_\text{L})$$

Second step:

$$\dfrac{}{\langle 0 + 1, s\rangle \to \langle 1, s\rangle}\,(\text{op})$$

Small-Step Semantics For Commands:

$$\frac{\langle E, s\rangle \to \langle E', s'\rangle}{\langle I := E, s\rangle \to \langle I := E', s'\rangle}\ (:=_{\mathrm{R}}) \qquad \frac{}{\langle I := n, s\rangle \to \langle \mathit{skip}, s[I \mapsto n]\rangle}\ (:=)$$

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

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

$$\frac{}{\langle \mathit{if}\ \mathit{True}\ \mathit{then}\ C_1\ \mathit{else}\ C_2, s\rangle \to \langle C_1, s\rangle}\ (\mathrm{if_T})$$

$$\frac{}{\langle \mathit{if}\ \mathit{False}\ \mathit{then}\ C_1\ \mathit{else}\ C_2, s\rangle \to \langle C_2, s\rangle}\ (\mathrm{if_F})$$

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

Small Step Semantics

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:

$\langle P, s\rangle \to \langle z := 1; (x :=!y; y :=!z), s\rangle$
$\to \langle skip; (x :=!y; y :=!z), s[z \mapsto 1]\rangle$
$\to \langle x :=!y; y :=!z, s[z \mapsto 1]\rangle$
$\to \langle x := 2; y :=!z, s[z \mapsto 1]\rangle$
$\to \langle skip; y :=!z, s[z \mapsto 1, x \mapsto 2]\rangle$
$\to \langle y :=!z, s[z \mapsto 1, x \mapsto 2]\rangle$
$\to \langle y := 1, s[z \mapsto 1, x \mapsto 2]\rangle$
$\to \langle skip, s[z \mapsto 1, x \mapsto 2, y \mapsto 1]\rangle$

Small Step Semantics

Remark:

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

- $\langle n, s \rangle$ where n is an integer
- $\langle b, s \rangle$ where b is a boolean
- $\langle !l, s \rangle$ where $l \notin dom(s)$
- $\langle skip, s \rangle$

These are *terminal configurations*. In the case of $\langle !l, s \rangle$ where $l \notin dom(s)$ we say that the program is *blocked*.

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:
   $\langle \text{if } !l > 0 \text{ then } (C'; C) \text{ else skip}, s\rangle \rightarrow$

   $$\langle \text{if } 4 > 0 \text{ then } (C'; C) \text{ else skip}, s\rangle$$

   can be proved if $s(l) = 4$.
   Proof:

$$\dfrac{\dfrac{}{\langle !l, s\rangle \rightarrow \langle 4, s\rangle}\; l \in dom(s), s(l) = 4}{\langle !l > 0, s\rangle \rightarrow \langle 4 > 0, s\rangle}$$

$$\dfrac{\qquad}{\langle \text{if } !l > 0 \text{ then } (C'; C) \text{ else skip}, s\rangle \rightarrow \langle \text{if } 4 > 0 \text{ then } (C'; C) \text{ else skip}, s\rangle}$$

Summary

We have defined a transition system for SIMP, where the transition relation is defined by induction.
Transition steps represent steps of computation, the result of the program P in a memory state s can be found by performing transition steps starting from $\langle P, s \rangle$ until a terminal configuration is obtained (not all configurations reach a terminal configuration).

For more details on these topics, see Chapter 4 in the textbook.

In the next part of this lecture we will give another specification of the transition system: **Big-step semantics**.

Big-Step Semantics for SIMP

Remark: Determinism

The previous system is *deterministic*:
Given a configuration $\langle P, s \rangle$ there is a unique sequence of transitions from $\langle P, s \rangle$ with maximal length.
This is called the **evaluation sequence for** $\langle P, s \rangle$.

The evaluation sequence for $\langle P, s \rangle$ may be finite or infinite.

Evaluation sequence

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 $\langle n, s\rangle$ where n is an integer, or $\langle b, s\rangle$ where b is a boolean, or $\langle skip, s\rangle$).
- *Blocked (Stuck):* if the sequence eventually reaches a blocked configuration (that is, a configuration of the form $\langle !l, s\rangle$ where $l \notin dom(s)$).
- *Divergent:* if the sequence is infinite.

Examples:

$\langle$while True do skip, $s\rangle$ is divergent.

$\langle$if !x $= 0$ then skip else skip, $s\rangle$ is stuck if $dom(s)$ does not contain x.

$\langle$if $4 = 0$ then skip else skip, $s\rangle$ is terminating.

Big-Step Semantics for SIMP

The goal is to define a binary relation between configurations, which associates a configuration with its corresponding terminal one (if it exists).
That is, define by induction the binary relation $\langle P, s\rangle \to^* \langle P', s'\rangle$ such that $\langle P', s'\rangle$ is terminal.

The usual notation for this is $\langle P, s\rangle \Downarrow \langle P', s'\rangle$
In other words:

$$\langle P, s\rangle \Downarrow \langle P', s'\rangle \text{ if } \langle P, s\rangle \to^* \langle P', s'\rangle$$
$$\text{where } \langle P', s'\rangle \text{ is terminal.}$$

Big-Step Semantics

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

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

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

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

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

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

Big-Step Semantics

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

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

$$\frac{\langle B, s\rangle \Downarrow \langle True, s'\rangle \quad \langle C_1, s'\rangle \Downarrow \langle skip, s''\rangle}{\langle if\ B\ then\ C_1\ else\ C_2, s\rangle \Downarrow \langle skip, s''\rangle}\ (\text{if}_\text{T})$$

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

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

$$\frac{\langle B, s\rangle \Downarrow \langle True, s'\rangle \quad \langle C, s'\rangle \Downarrow \langle skip, s''\rangle \quad \langle while\ B\ do\ C, s''\rangle \Downarrow \langle skip, s'''\rangle}{\langle while\ B\ do\ C, s\rangle \Downarrow \langle skip, s'''\rangle}\ (\text{while}_\text{T})$$

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:

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

Using this we can prove:

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

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.

Adding Variable Declarations to SIMP

Syntax:

We extend the syntax of SIMP with blocks.

$$\text{C} ::= \text{begin loc x} := \text{E; C end}$$

Semantics:

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

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

where

- $l \notin \mathit{dom}(s') \cup \mathit{dom}(s'') \cup \mathit{locations}(C)$, 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).

Adding Variable Declarations to SIMP

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

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

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

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

Summary

We have given alternative specifications of the operational semantics of SIMP: small-step and big-step semantics.

For more details on these topics, see Chapter 4 in the textbook and the references in the Further Reading section (Section 4.5).