

Dynamic Semantics

The **dynamic semantics** of a language defines what happens when you run a program.

There are many approaches to defining the dynamic semantics of a language:

- ▶ Small-step operational semantics
- ▶ Big-step operational semantics
- ▶ Abstract machines
- ▶ Translation to another language
- ▶ Denotational semantics (translation to a mathematical system)
- ▶ Axiomatic semantics

In this class we will primarily be concerned with the small-step operational semantics, which is the most widely used approach.

Recall the language of Arithmetic Expressions

$$e ::= \text{true} \mid \text{false} \mid \text{if } e \text{ then } e \text{ else } e \\ \mid 0 \mid \text{succ } e \mid \text{pred } e \mid \text{iszero } e$$
$$v ::= \text{true} \mid \text{false} \mid nv \\ nv ::= 0 \mid \text{succ } nv$$

Small-step operational semantics

In a small-step semantics, we take textual rewriting view of running a program. That is, we evaluate a program by changing the program text a little bit at a time.

Example

if iszero $(pred (succ 0))$ then 0 else succ 0

\mapsto

if iszero 0 then 0 else succ 0

if *iszero* 0 then 0 else succ 0

\mapsto

if *true* then 0 else succ 0

if true then 0 else succ 0

\mapsto

0

Definition of small-step semantics

We define a relation between two expressions that says when one expression evaluates to the other expression. We'll call this the `ReducesTo` relation and define the relation inductively with the following rules. The first bunch of rules do some real computation

$$(1) \frac{}{(pred\ 0, 0) \in ReducesTo} \qquad (2) \frac{}{(pred\ succ\ nv, nv) \in ReducesTo}$$

$$(3) \frac{}{(iszero\ succ\ nv, false) \in ReducesTo}$$

$$(4) \frac{}{(iszero\ 0, true) \in ReducesTo}$$

$$(5) \frac{}{(if\ true\ then\ e_2\ else\ e_3, e_2) \in ReducesTo}$$

$$(6) \frac{}{(if\ false\ then\ e_2\ else\ e_3, e_3) \in ReducesTo}$$

Before defining the rest of the rules for `ReducesTo`, we introduce the following notation:

$$e \mapsto e' \equiv (e, e') \in \textit{ReducesTo}$$

Congruence Rules

The second bunch of rules, called congruence rules, define the evaluation order by saying where you can reach inside a large expression to evaluate a subexpression.

$$(7) \frac{e \mapsto e'}{\text{succ } e \mapsto \text{succ } e'} \qquad (8) \frac{e \mapsto e'}{\text{pred } e \mapsto \text{pred } e'}$$

$$(9) \frac{e \mapsto e'}{\text{iszero } e \mapsto \text{iszero } e'}$$

$$(10) \frac{e_1 \mapsto e'_1}{(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) \mapsto (\text{if } e'_1 \text{ then } e_2 \text{ else } e_3)}$$

Inversion Lemmas

Suppose you know that for some e and e' , $e \mapsto e'$. What further information can you glean from this based on the rules for ReducesTo? We can do case analysis on the rules.

For example, suppose

$$\text{if true then } e_2 \text{ else } e_3 \mapsto e_4$$

Then you know that $e_2 = e_4$ because the only rule that could have appeared as the final step in the derivation is rule (5).

Why not rule (10)? Because then there would need to be a rule of the form $\text{true} \mapsto -$, but there is no such rule.

A Proof by Rule Induction on ReducesTo

Theorem

If $e \mapsto e'$ and $e \mapsto e''$ then $e' = e''$.

Proof by rule induction on $e \mapsto e'$.

Case (1) $\boxed{\text{pred } 0 \mapsto 0}$:

So $e = \text{pred } 0$, $e' = 0$ and by assumption we have $\text{pred } 0 \mapsto e''$.

There are two rules that could have applied (1) and (8), but (8) can't apply because there are no rules of the form $0 \mapsto -$. Thus, $e'' = 0$ and we conclude that $e' = e''$.

A Proof by Rule Induction on ReducesTo, continued

$$\text{Case (7) } \boxed{\frac{e_1 \mapsto e'_1}{\text{succ } e_1 \mapsto \text{succ } e'_1}} :$$

So $e = \text{succ } e_1$ and $e' = \text{succ } e'_1$. From the induction hypothesis we have

(a) $\forall x. e_1 \mapsto x \text{ implies } x = e'_1$.

Also, by assumption we have $\text{succ } e_1 \mapsto e''$. The only rule that could have applied is (7), so we know there was some e''_1 such that $e_1 \mapsto e''_1$. Then using (a) we have $e''_1 = e'_1$. Thus $e = \text{succ } e''_1 = \text{succ } e'_1 = e''$.

The other cases are left as an exercise.

More Definitions

Definition

An expression e is in **normal form** if $\nexists e'. e \mapsto e'$.

Definition

Multi-step reduction (written \mapsto^*) is the relation inductively defined by the following rules:

$$(1) \frac{}{e \mapsto^* e} \qquad (2) \frac{e_1 \mapsto e_2 \quad e_2 \mapsto^* e_3}{e_1 \mapsto^* e_3}$$

(This is an alternate but equivalent definition than the one in the textbook.)

The Untyped Lambda Calculus

Syntax:

x		Variables
$e ::= x \mid (\lambda x.e) \mid (e e)$		Expressions
$v ::= \lambda x.e$		Values

- ▶ An expression of the form $(\lambda x.e)$ create an anonymous function. The x is the one parameter of the function and e is the function body.
- ▶ An expression of the form $(e_1 e_2)$ is a function call. The expression e_1 should evaluate to a function, which is then called using the result of e_2 as the argument.

Call-by-value Small-step Semantics

$$(1) \frac{}{((\lambda x. e_1) v_2) \mapsto [x \mapsto v_2] e_1}$$

$$(2) \frac{e_1 \mapsto e'_1}{(e_1 e_2) \mapsto (e'_1 e_2)}$$

$$(2) \frac{e_2 \mapsto e'_2}{(v_1 e_2) \mapsto (v_1 e'_2)}$$

(Note where we use v instead of e to restrict the order of evaluation!)

The variable name associated with a λ doesn't really matter. The meaning of the program remains unchanged if you change the variable name and consistently replace the occurrences of that variable in the λ .

$$\lambda y.e = \lambda z.[y \mapsto z]e \quad \text{provided } z \text{ is not in } e$$

Substitution and Free Variables

We write $[x \mapsto e']e$ to say that e' is **substituted** for the free occurrences of x in e .

For example, in the following we substitute the expression $(y z)$ for the variable x in the expression $((\lambda x. x) x)$.

$$[x \mapsto (y z)]((\lambda x. x) x) = ((\lambda x. x) (y z))$$

A **free occurrence** of a variable x within an expression e is an occurrence of x in e that does not have a surrounding λ in e that binds x .

The the following examples, free occurrences are enclosed in a box.

- ▶ $(\boxed{z} \boxed{x})$
- ▶ $(\lambda y. y)$
- ▶ $(\lambda y. (y \boxed{x}))$
- ▶ $((\lambda x. x) \boxed{x})$
- ▶ \boxed{y}

Free Variables

The function FV computes the set of free variables in a given expression.

$$\begin{aligned}FV(x) &= \{x\} \\FV((e_1 e_2)) &= FV(e_1) \cup FV(e_2) \\FV((\lambda x. e)) &= FV(e) - \{x\}\end{aligned}$$

Capture-avoiding Substitution

- ▶ We need to be careful when defining substitution $([x \mapsto e']e)$ to make sure that free variables in e' don't get captured by λs in e .
- ▶ The following is an example of what happens if you don't define substitution correctly:

$$(\lambda y. (\lambda x. \lambda y. x) y) \longrightarrow (\lambda y. (\lambda y. y))$$

- ▶ Here's a correct definition of substitution:

$$\begin{aligned} [x \mapsto e']y &= \text{if } x = y \text{ then } e' \text{ else } y \\ [x \mapsto e'](\lambda y. e) &= (\lambda z. [x \mapsto e'] [y \mapsto z] e) \quad (\text{where } z \text{ is fresh}) \\ [x \mapsto e'](e_1 e_2) &= ([x \mapsto e']e_1 [x \mapsto e']e_2) \end{aligned}$$

Evaluation Contexts

Congruence rules are a verbose way to specify evaluation order. A more succinct way is to use evaluation contexts.

Recall the small-step textual rewriting view of semantics:

if iszero $\boxed{(pred (succ 0))}$ then 0 else succ 0

\mapsto

if iszero $\boxed{0}$ then 0 else succ 0

An **evaluation context** is everything *not* inside an evaluation box. We can use a grammar to define where the boxes are allowed and what the surrounding evaluation context looks like.

$E ::= \boxed{\quad}$ The evaluation box forms a hole in the context.
 $(E e)$
 $(v E)$

Filling an Evaluation Context

$$\begin{aligned} \text{fill}([\], e) &= e \\ \text{fill}((E e'), e) &= (\text{fill}(E, e) e') \\ \text{fill}((\nu E), e) &= (\nu \text{fill}(E, e)) \end{aligned}$$

Notation: $E[e] \equiv \text{fill}(E, e)$

Example

$$((\lambda y. y) [\])[(\lambda x. x)] = ((\lambda y. y) (\lambda x. x))$$

Small-step Semantics with Evaluation Contexts

We separate out the computational reduction rules into the relation \longrightarrow . For the lambda calculus there is just one rule:

$$(\beta) \frac{}{((\lambda x. e_1) v_2) \longrightarrow [x \mapsto v_2] e_1}$$

We then define top-level reduction using evaluation contexts:

$$\frac{e \longrightarrow e'}{E[e] \longmapsto E[e']}$$