Dynamic Semantics

- **Dynamic semantics** = how a program is executed.
  - There are many methods for specifying dynamic semantics.
- We'll use a **language-based** model of computation.
  - Step-by-step evaluation of expressions.
  - No need to refer to an implementation.

Dynamic Semantics

- Contrast: **machine-based** models.
  - Specify an abstract machine.
  - Specify how the language is compiled.
  - Dynamic semantics given via compilation.
  - Eg, C <-> RAM (15-211).

Language vs Machine Models

- Language models are
  - portable across implementations
  - more direct, and hence easier to understand
  - inherently abstracted from machine-level details

Language vs Machine Models

- Machine-based models are
  - not as portable (depending on the abstract machine)
  - inherently less direct (requiring a compilation model)
  - able to address machine-specific details

Language vs Machine Models

- Abstract (language-based) models will dominate concrete (machine-based) models in the future.
  - Abstraction is a virtue in a heterogeneous, distributed environment.
  - Low-level hacking is a very small percentage of software development.
  - Emphasis on bit-level efficiency is often misplaced.

Structured Operational Semantics (SOS)

- We'll define the dynamic semantics by a **transition relation,** $e \mapsto e'$, read "e **steps to** e'".
- Each step consists of executing one **instruction**.
- Complete when e is a **value**: a number, boolean, or function.
  - Write v for values, e for gen'l expr's.

SOS, cont'd

- The transition relation is inductively-defined by two sets of rules:
  - **primitive instructions**: basic steps of computation
  - **search rules**: find the next primitive instruction to execute
- The search rules determine the **order of evaluation** of the primitive instructions.

Primitive Instructions

- Primitive operations
  - $o(v1,\ldots,vn) \mapsto v$, if v is the result of executing o on arguments v1, …, vn
  - eg, $m+n \mapsto p$, if p is the sum of m and n
- Conditionals
  - if true then e1 else e2 $\mapsto$ e1
  - if false then e1 else e2 $\mapsto$ e2

Primitive Instructions

- Function call
  - apply (v, v1) $\mapsto$ [v,v1/f,x]e, where v = fun f(x:t):t' is e
- Two aspects:
  - plug in the argument for x
  - plug in the function itself for f (to implement recursion)

Search Rules

- Evaluation arguments to primop's left-to-right
  - if ei $\mapsto$ ei', then o(v1,…,ei,…, en) $\mapsto$ o(v1,…,ei', …, en)
  - (ei is the leftmost non-value)
  - eg, if e1 $\mapsto$ e1', then e1 + e2 $\mapsto$ e1' + e2; if e2 $\mapsto$ e2', then v1 + e2 $\mapsto$ v1 + e2'.

Search Rules

- Evaluate "test" expression of conditional before branching.
  - if e $\mapsto$ e', then if e then e1 else e2 fi $\mapsto$ if e' then e1 else e2 fi
  - do not evaluate "then" or "else" clauses until decision is made!

Search Rules

- Evaluate function, then argument.
  - if e1 $\mapsto$ e1', then apply(e1,e2) $\mapsto$ apply(e1',e2)
  - if e2 $\mapsto$ e2', then apply(v1, e2) $\mapsto$ apply(v1, e2')
- Could swap order.
- Could defer evaluation of argument.
  - "lazy language"

Example

- Let v be the factorial function.
  - fun f(x:int):int is if x=0 then 1 else x*f(x-1) end
- Evaluate v(3) = apply(v,3):
  - $\mapsto$ if 3=0 then 1 else 3 * v (3-1)
  - $\mapsto^*$ if false then 1 else 3 * v (3-1)
  - $\mapsto$ 3 * v (3-1)
  - $\mapsto$ 3 * v (2)
  - $\mapsto$ 3 * if 2=0 then 1 else 2 * v (2-1)

Example

- Cont'd,
  - $\mapsto^*$ 3 * 2 * v (2-1)
  - $\mapsto$ 3 * 2 * v (1)
  - $\mapsto^*$ 3 * 2 * 1 * v(0)
  - $\mapsto$ 3 * 2 * 1 * if 0=0 then 1 else 0 * v (0-1)
  - $\mapsto^*$ 3 * 2 * 1 * 1
  - $\mapsto^*$ 6

Example

- Observe that the expression first grows longer in proportion to the magnitude of the argument:
  - ... $\mapsto$ n * (n-1) * (n-2) * ... * 2 * 1
- Then shrinks as pending operations are performed
  - ... $\mapsto$ n!

Determinacy of Evaluation

- For every closed expression e, there exists at most one expression e' such that e $\mapsto$ e'.
  - Proof is by induction on the evaluation rules.
- Corollary: every expression has at most one value.
  - The relation $\mapsto^*$ is a partial function!

Determinacy, cont'd

- Proof is by induction on expressions.
  - If e is a number n, then there is no e' such that e $\mapsto$ e'. Ditto the other values.
  - If e = apply(e1, e2), where e1 is not a value, then by induction there exists at most one e1' st e1 $\mapsto$ e1'. Therefore there exists at most one e' such that e $\mapsto$ e', namely apply(e1',e2).

Determinacy, cont'd

- Suppose e = apply(v1,e2). By induction there exists at most one e2' st e2 $\mapsto$ e2'. Hence there exists at most one e', namely apply(v1,e2'), st e $\mapsto$ e'.
- Suppose e = apply(v1,v2). If v1 is not a function, there is no e' st e $\mapsto$ e'. If v1 = fun f(x:t):t' is e'' end, then there is exactly one e' st e $\mapsto$ e'.

Dynamic Semantics: Summary

- We employ a **language-based model** of computation.
  - Define execution of expressions, not translation to machine code.
- Given by an inductively-defined **transition relation** between expressions.
  - Compute by calculation.