CS150 Adv Prog Lang: Dynamic Semantics

Guannan Wei

`guannan.wei@tufts.edu`

Sept 4, 2025

Tufts University

Last time

- Logistics update: paper reading/discussion can be selected from advanced topics in textbooks
- Defining a programming language
    - Syntax
    - Dynamic semantics
    - Static semantics

Last time: operational semantics

Syntax of the λ-calculus:

$$\begin{array}{rcll} n & \in & \mathbb{N} & \\ t & ::= & n \mid x \mid \lambda x.t \mid t_1\, t_2 & \textbf{terms} \end{array}$$

Many flavors of operational semantics:

- *Structural operational semantics (i.e. small-step semantics)*
- *Contextual reduction semantics*
- Abstract machines
- Natural semantics (i.e. big-step semantics)
- Evaluators

Last time: Call-by-value

Structural operational semantics

$$\frac{}{(\lambda x.t)\,v \to t[x := v]}\;\beta_v \qquad \frac{t_1 \to t_1'}{t_1\,t_2 \to t_1'\,t_2}\;\textsc{App1} \qquad \frac{t_2 \to t_2'}{v\,t_2 \to v\,t_2'}\;\textsc{App2}$$

Reduction semantics

$$E \quad ::= \quad \square \mid v\,E \mid E\,t \quad \textbf{reduction contexts}$$

$$\frac{}{(\lambda x.t)\,v \to t[x := v]}\;\beta_v \qquad \frac{t_1 \to t_1'}{E[t_1] \to E[t_1']}\;\textsc{Ctx}$$

Last time: Call-by-name

Structural operational semantics

$$\frac{}{(\lambda x.t_1)\, t_2 \to t_1[x := t_2]}\;\beta \qquad\qquad \frac{t_1 \to t_1'}{t_1\, t_2 \to t_1'\, t_2}\;\textsc{App}$$

Reduction semantics

- Question: define the evaluation context for CBN.

Last time: Call-by-name

Structural operational semantics

$$\frac{}{(\lambda x.t_1)\,t_2 \to t_1[x := t_2]}\ \beta \qquad\qquad \frac{t_1 \to t_1'}{t_1\,t_2 \to t_1'\,t_2}\ \textsc{App}$$

Reduction semantics

$$E \quad ::= \quad \Box \mid \xcancel{v\,E} \mid E\,t \quad \textbf{reduction contexts}$$

$$\frac{}{(\lambda x.t_1)\,t_2 \to t_1[x := t_2]}\ \beta \qquad\qquad \frac{t_1 \to t_1'}{E[t_1] \to E[t_1']}\ \textsc{Ctx}$$

Some properties

- Decomposition is unique
  - Given t, there exists only one E and $(\lambda x.t)\, v$ such that $t = E[(\lambda x.t)\, v]$
  - Uniqueness of decomposition implies the determinism of evaluation
- Equivalence between SOS and reduction semantics

From reduction semantics to abstract machines

Given t, find an E and t_1 such that $t = E[t_1]$, if $t_1 \to t_1'$, plug t_1' into E to obtain $E[t_1']$.

- It postulates $decompose : \text{Term} \to (\text{Ctx}, \text{Term})$ and $plugin : (\text{Ctx}, \text{Term}) \to \text{Term}$ (meta)-functions.

From reduction semantics to abstract machines

Given t, find an E and t_1 such that $t = E[t_1]$, if $t_1 \to t_1'$, plug t_1' into E to obtain $E[t_1']$.

- It postulates $decompose$: Term $\to$ (Ctx, Term) and $plugin$: (Ctx, Term) $\to$ Term (meta)-functions.
- They need to search for the innermost redex $(\lambda x.t)\, v$ in the AST and reconstruct the AST replacing $\square$ (complexity: both $O(\text{height}(t))$).

From reduction semantics to abstract machines

Given t, find an E and t_1 such that $t = E[t_1]$, if $t_1 \to t_1'$, plug t_1' into E to obtain $E[t_1']$.

- It postulates $decompose : \text{Term} \to (\text{Ctx}, \text{Term})$ and $plugin : (\text{Ctx}, \text{Term}) \to \text{Term}$ (meta)-functions.
- They need to search for the innermost redex $(\lambda x.t)\, v$ in the AST and reconstruct the AST replacing $\Box$ (complexity: both $O(\text{height}(t))$).
- Neither a faithful description of an “implementation”, nor can be used as an efficient one.

The CC abstract machine

- Idea: materialize the search of redex by maintaining a pair of focused term and its context, and directly manipulate context.
- CC machine stands for “control string”-“context” machine

CC Machine: $\langle t, E\rangle \rightarrow_{cc} \langle t', E'\rangle$

$$E ::= \square \mid v\,E \mid E\,t \quad \textbf{reduction contexts}$$

$$\begin{array}{rcll}
\langle t_1\ t_2, E\rangle & \rightarrow_{cc} & \langle t_1, E[(\square\ t_2)]\rangle \text{ if } t_1 \text{ not value} & \text{[cc-app1]} \\
\langle v\ t_2, E\rangle & \rightarrow_{cc} & \langle t_2, E[(v\ \square)]\rangle \text{ if } t_2 \text{ not value} & \text{[cc-app2]} \\
\langle (\lambda x.t)\ v, E\rangle & \rightarrow_{cc} & \langle t[x := v], E\rangle & \text{[cc-}\beta\text{]} \\
\langle v, E[(\square\ t)]\rangle & \rightarrow_{cc} & \langle v\ t, E\rangle & \text{[cc-use1]} \\
\langle v_2, E[(v_1\ \square)]\rangle & \rightarrow_{cc} & \langle v_1\ v_2, E\rangle & \text{[cc-use2]}
\end{array}$$

The CC abstract machine

- Example in class:

  $((\lambda f.\lambda x.f\ x)\ (\lambda y.y))\ 1$

Simplifying the CC machine

CC Machine: $\langle t, E\rangle \rightarrow_{cc} \langle t', E'\rangle$

$$E \quad ::= \quad \square \mid v\,E \mid E\,t \quad \textbf{reduction contexts}$$

$$\begin{array}{rcll}
\langle t_1\ t_2, E\rangle & \rightarrow_{cc} & \langle t_1, E[(\square\ t_2)]\rangle \text{ if } t_1 \text{ not value} & \text{[cc-app1]} \\
\langle v\ t_2, E\rangle & \rightarrow_{cc} & \langle t_2, E[(v\ \square)]\rangle \text{ if } t_2 \text{ not value} & \text{[cc-app2]} \\
\langle (\lambda x.t)\ v, E\rangle & \rightarrow_{cc} & \langle t[x := v], E\rangle & \text{[cc-}\beta\text{]} \\
\langle v, E[(\square\ t)]\rangle & \rightarrow_{cc} & \langle v\ t, E\rangle & \text{[cc-use1]} \\
\langle v_2, E[(v_1\ \square)]\rangle & \rightarrow_{cc} & \langle v_1\ v_2, E\rangle & \text{[cc-use2]}
\end{array}$$

- What is the rule used after cc-use1?

Simplifying the CC machine

CC Machine: $\langle t, E\rangle \to_{cc} \langle t', E'\rangle$

$$E \quad ::= \quad \Box \mid v\,E \mid E\,t \quad \textbf{reduction contexts}$$

$$\begin{array}{rcll}
\langle t_1\ t_2, E\rangle & \to_{cc} & \langle t_1, E[(\Box\ t_2)]\rangle \text{ if } t_1 \text{ not value} & \text{[cc-app1]} \\
\langle v\ t_2, E\rangle & \to_{cc} & \langle t_2, E[(v\ \Box)]\rangle \text{ if } t_2 \text{ not value} & \text{[cc-app2]} \\
\langle (\lambda x.t)\ v, E\rangle & \to_{cc} & \langle t[x := v], E\rangle & \text{[cc-}\beta\text{]} \\
\langle v, E[(\Box\ t)]\rangle & \to_{cc} & \langle v\ t, E\rangle & \text{[cc-use1]} \\
\langle v_2, E[(v_1\ \Box)]\rangle & \to_{cc} & \langle v_1\ v_2, E\rangle & \text{[cc-use2]}
\end{array}$$

- What is the rule used after cc-use1?
- What is the rule used after cc-use2?

The Simplified CC abstract machine

SCC Machine: $\langle t, E\rangle \rightarrow_{scc} \langle t', E'\rangle$

$$E \quad ::= \quad \Box \mid v\,E \mid E\,t \quad \textbf{reduction contexts}$$

$$\begin{array}{rcll}
\langle t_1\ t_2, E\rangle & \rightarrow_{scc} & \langle t_1, E[(\Box\ t_2)]\rangle & \text{[scc-app1]} \\
\langle v, E[(\Box\ t)]\rangle & \rightarrow_{scc} & \langle t, E[(v\ \Box)]\rangle & \text{[scc-app2]} \\
\langle v, E[((\lambda x.t)\ \Box)]\rangle & \rightarrow_{scc} & \langle t[x := v], E\rangle & \text{[scc-}\beta\text{]}
\end{array}$$

From Contexts to Continuations

SCC Machine: $\langle t, E\rangle \rightarrow_{scc} \langle t', E'\rangle$

$$E \quad ::= \quad \Box \mid v\,E \mid E\,t \quad \textbf{reduction contexts}$$

$$\begin{array}{rcll}
\langle t_1\ t_2, E\rangle & \rightarrow_{scc} & \langle t_1, E[(\Box\ t_2)]\rangle & \text{[scc-app1]} \\
\langle v, E[(\Box\ t)]\rangle & \rightarrow_{scc} & \langle t, E[(v\ \Box)]\rangle & \text{[scc-app2]} \\
\langle v, E[((\lambda x.t)\ \Box)]\rangle & \rightarrow_{scc} & \langle t[x := v], E\rangle & \text{[scc-}\beta\text{]}
\end{array}$$

- SCC machine still needs the “decompose” and “plugin” meta-functions
- But, also observe that context E is used as a *stack*:
  - scc-app1 “pushes” a new $(\Box\ t_2)$ frame to the top of context E
  - scc-app2 “peeks” the top frame of E and replace it
  - scc-beta “pops” the top frame of E

From Contexts to Continuations

- Idea: use a list-like data structure to represent contexts
- CK machine stands for “control-string”-“continuation” machine

Continuation

$$\begin{array}{rcll} \kappa & ::= & \mathsf{halt} & \\ & \mid & \mathsf{fun}(v, \kappa) & \text{hold the value at the function position} \\ & \mid & \mathsf{arg}(t, \kappa) & \text{hold the term at the argument position} \end{array}$$

From Contexts to Continuations

- Idea: use a list-like data structure to represent contexts
- CK machine stands for “control-string”-“continuation” machine

Continuation

κ	$::=$	halt	
	$\mid$	fun(v, κ)	hold the value at the function position
	$\mid$	arg(t, κ)	hold the term at the argument position

Or, in a programming language, such as Standard ML

```
datatype cont = Halt
              | Fun of value * cont
              | Arg of term * cont
```

From Contexts to Continuations

- Idea: use a list-like data structure to represent contexts
- CK machine stands for "control-string"-"continuation" machine

Continuation

κ	$::=$	halt	
	$\mid$	$\mathsf{fun}(v, \kappa)$	hold the value at the function position
	$\mid$	$\mathsf{arg}(t, \kappa)$	hold the term at the argument position

- Context, continuations, stack: what should be done after evaluating the current expression

Continuation as stack, explicitly

f	$::=$	$\mathsf{fun}(v) \mid \mathsf{arg}(t)$	**stack frames**
κ	$::=$	$\mathsf{halt} \mid f :: \kappa$	**continuation/stack**

The CK Machine

Continuation

$$\begin{array}{rcll} \kappa & ::= & \mathsf{halt} & \\ & | & \mathsf{fun}(v, \kappa) & \text{hold the value at the function position} \\ & | & \mathsf{arg}(t, \kappa) & \text{hold the term at the argument position} \end{array}$$

CK Machine: $\langle t, \kappa \rangle \rightarrow_{ck} \langle t', \kappa' \rangle$

$$\begin{array}{rcll} \langle t_1\ t_2, \kappa \rangle & \rightarrow_{ck} & \langle t_1, \mathsf{arg}(t_2, \kappa) \rangle & \text{[ck-app1]} \\ \langle v, \mathsf{arg}(t, \kappa) \rangle & \rightarrow_{ck} & \langle t, \mathsf{fun}(v, \kappa) \rangle & \text{[ck-app2]} \\ \langle v, \mathsf{fun}(\lambda x.t, \kappa) \rangle & \rightarrow_{ck} & \langle t[x := v], \kappa \rangle & \text{[ck-}\beta\text{]} \end{array}$$

On Substitution

$$\langle v, \mathsf{fun}(\lambda x.t, \kappa)\rangle \rightarrow_{ck} \langle \mathbf{t}[\mathbf{x} := \mathbf{v}], \kappa\rangle$$

- Eager textual substitution:
  - Needs to **traverse** the term's AST and find the free occurrences of x in t to replace with v.
  - But an actual implementation would not perform substitutions.
  - Caveat: if v is not closed (i.e. containing free variables), then substitution needs to be capturing avoiding.

On Substitution

$$\langle v, \mathsf{fun}(\lambda x.t, \kappa)\rangle \rightarrow_{ck} \langle \mathbf{t}[\mathbf{x} := \mathbf{v}], \kappa\rangle$$

- Eager textual substitution:
  - Needs to **traverse** the term's AST and find the free occurrences of x in t to replace with v.
  - But an actual implementation would not perform substitutions.
  - Caveat: if v is not closed (i.e. containing free variables), then substitution needs to be capturing avoiding.
- Alternative 1: don't substitute eagerly, but keep track of the binding values **in the syntax of the calculus** (explicit substitution).
- Alternative 2: don't substitute eagerly, but keep track of the binding values **at the meta-level** (environment).

The CEK abstract machine

- Idea: a partial mapping (i.e. environment) from variables to their values

$$\begin{array}{rcll} v \in \text{Value} & ::= & n \mid \lambda x.t & \textbf{values} \\ \rho \in \text{Env} & ::= & \text{Var} \rightharpoonup (\text{Value} \times \text{Env}) & \textbf{environment} \\ \kappa \in \text{Cont} & ::= & \text{halt} \mid \text{fun}(v, \rho, \kappa) \mid \text{arg}(t, \rho, \kappa) & \textbf{continuation} \end{array}$$

CEK Machine: $\langle t, \rho, \kappa \rangle \rightarrow_{cek} \langle t', \rho', \kappa' \rangle$

$$\begin{array}{rcll} \langle x, \rho, \kappa \rangle & \rightarrow_{cek} & \langle v, \rho', \kappa \rangle \text{ if } \rho(x) = (v, \rho') & \text{[cek-var]} \\ \langle t_1\ t_2, \rho, \kappa \rangle & \rightarrow_{cek} & \langle t_1, \rho, \text{arg}(t_2, \rho, \kappa) \rangle & \text{[cek-app1]} \\ \langle v, \rho, \text{arg}(t, \rho', \kappa) \rangle & \rightarrow_{cek} & \langle t, \rho', \text{fun}(v, \rho, \kappa) \rangle & \text{[cek-app2]} \\ \langle v, \rho, \text{fun}(\lambda x.t, \rho', \kappa) \rangle & \rightarrow_{cek} & \langle t, \rho'[x \mapsto (v, \rho)], \kappa \rangle & \text{[cek-app3]} \end{array}$$

The CEK abstract machine

- Example in class: extend the CEK machine with arithmetics $t_1 + t_2$
- Example in class:

  $((\lambda f.\lambda x.f\ x)\ (\lambda w.w + 1))\ 2$

The CEK abstract machine

- Closure = Value $\times$ Env

  The environment provides values for free variables in the value (thus “closes” the value).

- Lexical scoping: free variables bind in the environment at the time a function is defined
- Dynamic scoping: free variables bind in the environment at the time a function is called (very few languages in this way)

Natural Semantics

- So far, all semantics executes with discrete steps
  - These steps relate intermediate terms/states
  - We can observe intermediate states during evaluation

Natural Semantics

- Alternative: directly relating the initial term and final value

Natural semantics: $(t, \rho) \Downarrow v$

$$\frac{\rho(x) = v}{(x, \rho) \Downarrow v} \qquad \frac{}{(\lambda x.t, \rho) \Downarrow (\lambda x.t, \rho)}$$

$$\frac{(t_1, \rho) \Downarrow (\lambda x.t, \rho') \quad (t_2, \rho) \Downarrow v_2 \quad (t, \rho'[x \mapsto v_2]) \Downarrow v}{(t_1\, t_2, \rho) \Downarrow v}$$

Natural Semantics

- What if the program does not terminate (i.e. diverging)?
- What if the langauge has some concurrency primitives?

Evaluator

- Now read $\Downarrow$ as a function: $\Downarrow (t, \rho) = v$
- Directly correspond to a recursive, direct-style evaluator, implementing the natural semantics

```
def eval(t: Term, env: Map[Var, Closure]): Closure =
  t match
    case Var(x) => env(x)
    case App(t1, t2) =>
      val Closure(Lam(x, t), env1) = eval(t1, env)
      val v2 = eval(t2, env)
      eval(t, env1 + (x -> v2))
    case Lam(x, t) =>
      Closure(Lam(x, t), env)
```

Summary

Different ways of specifying semantics, describing different level of execution

- Structural operatioanal semantics (SOS): purely term rewriting
- Reduction semantics: evaluation strategy defined by contexts
- Abstract machines: more mechanical and efficient
- Natural semantics: relating the initial term and final result
- Direct-style evaluator: direct implementation of natural semantics

Summary

Different ways of specifying semantics, describing different level of execution

- Structural operatioanal semantics (SOS): purely term rewriting
- Reduction semantics: evaluation strategy defined by contexts
- Abstract machines: more mechanical and efficient
- Natural semantics: relating the initial term and final result
- Direct-style evaluator: direct implementation of natural semantics

Some exercises:

- Extend reduction semantics with arithmetic operations
- Implement the compose/plugin function for reduction semantics
- Implement the CC/SCC/CEK machine and extend it with numbers and arithmetic operations

Further reading

- Are there call-by-name abstract machines?

  Yes, look at Krivine's machine.

- Does the CEK machine correspond to an evaluator?

  A functional correspondence between evaluators and abstract machines. PPDP '03.

- What if our language imperative features (e.g. assignment, mutation, etc.)?

  Look at CESK machine ("S" for store/heap).

References

- Programming Languages and Lambda Calculi, Ch6 and Ch7

  https://users.cs.utah.edu/~mflatt/past-courses/cs7520/public_html/s06/notes.pdf

- Control operators, the SECD-machine, and the -calculus. Matthias Felleisen, Daniel P. Friedman

- Definitional Interpreters for Higher-Order Programming Languages. John Reynolds