Armando Solar-Lezama and Martin Rinard
Course Staff

Armando Solar-Lezama

Martin Rinard
What this course is about

The central theme of this course:

How do we get tools to reason about a program’s behavior

This is important in many areas
- finding bugs
- designing languages to prevent bugs
- synthesizing programs
- manipulating programs automatically (refactoring, optimization)

Before we get to that though…
- we must be able to reason about program behaviors ourselves
Course outline

Semantics
- learn to make formal arguments about program behavior
- we’ll cover operational and axiomatic semantics

Abstract Interpretation
- use abstraction to reason about the behavior of the program under all possible inputs.

Type theory
- learn how to design and reason about type systems
- use type-based analysis to find synchronization errors, avoid information leaks and manage your memory efficiently

Model checking
- learn how to reason exhaustively about program states
- learn how abstraction and symbolic reasoning can help you find bugs in device drivers and protocol designs
Grading

3 homework assignments
- 15% of your grade each
- you can work in groups, but each person must submit their own solutions
- start on them early!

Term Project
- apply what you learned in class to a concrete problem
- be original
- make it real
- you’ll start by submitting a project proposal (due 11/9)
- by the end of the course you’ll make a presentation
- and turn in a report (conference format)
And now the fun part...
Introduction to Operational Semantics

Lecture 1

Adapted in part from Winskel, Pierce, Turbak & Gifford and slides by George Necula.
A simple language: $\lambda$-calculus

$\lambda$-calculus pre-dates computers or even Turing machines - developed in 1932 by Alonzo Church as a logic system.

Based entirely around the concept of function application

Arguably one of the simplest Turing-complete languages
The basics of $\lambda$-calculus

Syntax:

\[
e := \quad x \quad \text{(a variable)} \\
\quad \lambda x. e \quad \text{(a function)} \\
\quad e_1 e_2 \quad \text{(function application)}
\]

Example:

\[
(\lambda x. (\lambda y. y x)) (\lambda z. z) (\lambda w. w w)
\]

Application associates to the \textit{left} (it’s just a convention)

The body of a function goes as far as possible to the \textit{right} (another notational convention)
The basics of λ-calculus

1 execution rule:
- Whenever you apply a lambda expression $\lambda x. e$ to an expression $e_2$, replace $x$ in $e$ with $e_2$.

Example:

$(\lambda x. (\lambda y. y x)) (\lambda z. z) (\lambda w. w w)$

$(\lambda y. y (\lambda z. z)) (\lambda w. w w)$

$(\lambda w. w w) (\lambda z. z)$

$(\lambda z. z) (\lambda z. z)$

$(\lambda z. z)$

This is too informal!

What if there are many possible applications?

Do we chose one arbitrarily?

Will you get the same answer regardless?

We need a more precise definition of the semantics!
Why semantics

We want to build mathematical models of programs
- abstract away low-level implementation issues
- construct solid arguments (proofs) about properties of a language
  - is the language deterministic?
  - can I write non-terminating programs?
- reason about soundness of analysis
  - if an analysis says the program never dereferences null pointers, can I trust it?
Why operational semantics

Models the execution of a program in an abstract machine

Simple but extremely powerful
- powerful enough to allow us to prove interesting properties

Easy to relate to an implementation
Big Step Operational Semantics

Model the execution in an abstract machine

Basic Notation: Judgments

\[ \langle \text{configuration} \rangle \rightarrow \text{result} \]

- describe how a program configuration is evaluated into a result
- the configuration is usually a program fragment together with any state.

Basic Notation: Inference rules

- define how to derive judgments for an arbitrary program
- also called derivation rules or evaluation rules
- usually defined recursively

\[ \langle c_1 \rangle \rightarrow r_1 \quad \langle c_2 \rangle \rightarrow r_2 \quad \ldots \quad \langle c_k \rangle \rightarrow r_k \]

\[ \langle \text{configuration} \rangle \rightarrow \text{result} \]
Big Step OS for $\lambda$ calculus

Configuration is simply a lambda expression
- there is no state

Result is a different lambda expression

Inductive definition: Base case

\[
\frac{\quad}{x \rightarrow x}
\]

Inductive definition: recursive cases

\[
\frac{e \rightarrow e'}{\lambda x. e \rightarrow \lambda x. e'}
\]

??

\[
\frac{e_1 e_2 \rightarrow e_3}{\quad}
\]
Big Step OS for $\lambda$ calculus

Many choices for function application

- **call by value**
  
  $e_1 \rightarrow \lambda x. e'_1$  
  $e_2 \rightarrow e'_2$  
  $e'_1[\alpha(e'_2)/x] \rightarrow e_3$

  $e_1 e_2 \rightarrow e_3$

  - $\alpha$-renaming: rename variables to avoid naming conflicts

- **call by name**
  
  $e_1 \rightarrow \lambda x. e'_1$  
  $e'_1[\alpha(e_2)/x] \rightarrow e_3$

  $e_1 e_2 \rightarrow e_3$

  - These produce slightly different semantics!
    
    - with call-by-value you can’t get recursive programs to terminate
Growing the language

We can grow the language with integers and arithmetic

\[
N \rightarrow n \\
\underbrace{e_1 + e_2 \rightarrow n}
\]

Ex: \((\lambda \ y. \ (\lambda \ x. \ 5 + y \ x)) \ (\lambda \ z. \ z + 1) \ 5\)

This is only syntactic sugar
- numbers and arithmetic can be represented in plain \(\lambda\) calculus
Growing the language

We can add predicates and conditionals too:

\[
\begin{align*}
\frac{e_1 \rightarrow n_1 \quad e_2 \rightarrow n_2 \quad n_1 = n_2}{e_1 = e_2 \rightarrow true} & \quad & \frac{e_1 \rightarrow n_1 \quad e_2 \rightarrow n_2 \quad n_1 \neq n_2}{e_1 = e_2 \rightarrow false}
\end{align*}
\]

\[
\begin{align*}
\frac{e_1 \rightarrow true \quad e_t \rightarrow e_t'}{if \ e_1 then \ e_t \ else \ e_f \rightarrow e_t'} & \quad & \frac{e_1 \rightarrow false \quad e_f \rightarrow e_f'}{if \ e_1 then \ e_t \ else \ e_f \rightarrow e_f'}
\end{align*}
\]

Ex: \((\lambda g. \ g\ g\ 3)\ (\lambda f. \ \lambda x. \ if \ x = 1 \ then \ x \ else \ (f\ (x-1)\ f)\ *\ x)\)
Growing the language

Ex: \((\lambda g. \ g \ g \ 3) \ (\lambda f. \ \lambda x. \ if \ x = 1 \ then \ x \ else \ (f \ f \ (x-1)) \ ) \ * \ x)\)

\[
e_1 \rightarrow \lambda x. \ e_1' \quad e_1'[\alpha(e_2)/x] \rightarrow e_3 \quad e_1e_2 \rightarrow e_3
\]

\[
e_1 \rightarrow n_1 \quad e_2 \rightarrow n_2 \quad n = n_1 + n_2 \quad e_1 + e_2 \rightarrow n
\]

\[
e_1 \rightarrow true \quad e_t \rightarrow e_t' \quad if \ e_1 \ \text{then} \ e_t \ \text{else} \ e_f \rightarrow e_t'
\]

\[
e_1 \rightarrow false \quad e_f \rightarrow e_f' \quad if \ e_1 \ \text{then} \ e_t \ \text{else} \ e_f \rightarrow e_f'
\]

\[
e_1 \rightarrow n_1 \quad e_2 \rightarrow n_2 \quad n_1 = n_2 \quad e_1 = e_2 \rightarrow true
\]

\[
e_1 \rightarrow n_1 \quad e_2 \rightarrow n_2 \quad n_1 \neq n_2 \quad e_1 = e_2 \rightarrow false
\]
The same techniques apply to programs with state
- The big difference is that the configuration now includes state

Example: IMP

\[
\begin{align*}
e &:= n \mid x \mid e_1 + e_2 \\
c &:= x := e \mid c_1 ; c_2 \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \mid \text{while } e \text{ do } c
\end{align*}
\]

Now we need two types of judgments
- expressions result in values
- commands change the state

\[
\langle e, \sigma \rangle \rightarrow n \quad \langle c, \sigma \rangle \rightarrow \sigma'
\]
Big Step OS for Imperative Programs

Rules for expressions are very similar to what we had before

\[
\begin{align*}
\langle N, \sigma \rangle \rightarrow n & \quad \frac{\langle e_1, \sigma \rangle \rightarrow n_1 \quad \langle e_2, \sigma \rangle \rightarrow n_2 \quad n = n_1 + n_2}{\langle e_1, \sigma \rangle + \langle e_2, \sigma \rangle \rightarrow n}
\end{align*}
\]

We need a rule to assign values to variables

\[
\frac{\langle x, \sigma \rangle \rightarrow \sigma(x)}{
}\]
Big Step OS for Imperative Programs

Commands mutate the state

\[
\frac{\langle e, \sigma \rangle \rightarrow e' }{ \langle X := e, \sigma \rangle \rightarrow \sigma[X \rightarrow e'] } \quad \quad \frac{\langle c_1, \sigma \rangle \rightarrow \sigma'' }{ \langle c_1; c_2, \sigma \rangle \rightarrow \sigma' } \quad \frac{\langle c_2, \sigma'' \rangle \rightarrow \sigma' }{ } 
\]

\[
\frac{\langle e_1, \sigma \rangle \rightarrow false }{ \langle if \ e_1 \ then \ c_t \ else \ c_f, \sigma \rangle \rightarrow \sigma' } \quad \quad \frac{\langle c_t, \sigma \rangle \rightarrow \sigma' }{ } 
\]

What about loops?
Big Step OS for Imperative Programs

The definition for loops must be recursive

\[
\frac{\langle e_1, \sigma \rangle \rightarrow false}{\langle \text{while } e_1 \text{then } c, \sigma \rangle \rightarrow \sigma}
\]

\[
\frac{\langle e_1, \sigma \rangle \rightarrow true \quad \langle c;\text{while } e_1 \text{then } c, \sigma \rangle \rightarrow \sigma'}{\langle \text{while } e_1 \text{then } c, \sigma \rangle \rightarrow \sigma'}
\]

\[
\frac{\langle e_1, \sigma \rangle \rightarrow true \quad \langle c, \sigma \rangle \rightarrow \sigma'' \quad \langle \text{while } e_1 \text{then } c, \sigma'' \rangle \rightarrow \sigma'}{\langle \text{while } e_1 \text{then } c, \sigma \rangle \rightarrow \sigma'}
\]
Small Step semantics

Big-step inference rules go from initial configuration to final result in one big step.

Sometimes we want to get a “closer look” at how the evaluation proceeds
- for example, in what order operands were evaluated
- or how the configuration evolves along the way
- particularly important if we are dealing with non-terminating computation!!

This can be done with small step semantics.
- model the *evolution* of the program configuration
Example: Small Step OS for $\lambda$ calculus

First, evaluate the body of $\lambda$

\[
\frac{e_1 \rightarrow e'_1}{(\lambda x. e_1)e_2 \rightarrow (\lambda x. e'_1)e_2}
\]

Call by name:
- Once the body can’t be reduced, apply
  \[
  \frac{\text{ired}(e_1)}{(\lambda x. e_1)e_2 \rightarrow e_1[\alpha(e_2)/x]}
  \]

Call by value
- Once the body can’t be reduced, reduce the argument

\[
\frac{\text{ired}(e_1) \quad e_2 \rightarrow e'_2}{(\lambda x. e_1)e_2 \rightarrow (\lambda x. e_1)e'_2}
\]
\[
\frac{\text{ired}(e_1) \quad \text{ired}(e_2)}{(\lambda x. e_1)e_2 \rightarrow e_1[\alpha(e_2)/x]}
\]
Small Step Semantics

Many design decisions
- How small is a step?
- How do we select the next step?

These decisions need to be defined formally
A redex is an expression that can be reduced in one atomic step.

The first step in defining a small step semantics is to define the redexes.

Ex.

- In IMP: $n_1 + n_2 | x = n | \text{skip; c} | \text{if true then c1 else c2} | \text{if false then c1 else c2} | \text{while b do c}$
- In $\lambda$-calculus: $(\lambda x. v) e_2 , (\lambda x. e_1) e_2$
Local reduction rules

One for each redex
- show how to advance one step of the execution

- \( <x, \sigma[x=n]> \to <n, \sigma> \)
- \( <n_1+n_2, \sigma> \to <n, \sigma> \) where \( n = n_1 + n_2 \)
- \( <x = n, \sigma> \to <\text{skip}, \sigma[x \mapsto n]> \)
- \( <\text{skip}; c, \sigma> \to <c, \sigma> \)
- \( <\text{if true then c1 else c2}, \sigma> \to <c_1, \sigma> \)
- \( <\text{if false then c1 else c2}, \sigma> \to <c_2, \sigma> \)
- \( <\text{while b do c}, \sigma> \to <\text{if b then (c; while b do c) else skip}, \sigma> \)
Global reduction rules

A simple algorithm
- start with a program
- identify a redex
- reduce according to local reduction rules
- repeat until you can’t reduce anymore

We need rules to define the next redex
We use $H$ to refer to a context. $H[r]$ is a program fragment consisting of redex $r$ in context $H$. Global reduction rules can be defined from local reduction rules as flows:

If $<r, \sigma> \uparrow <e, \sigma'>$ then $<H[r], \sigma'> \uparrow <H[e], \sigma'>$.

How we define the set of contexts will determine the order in which local reductions are applied.
### Example

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;x := (x + 1) + 2, [x=2]&gt;)</td>
<td>x = (o + 1) + 2</td>
<td>x</td>
</tr>
<tr>
<td>(&lt;x := (2 + 1) + 2, [x=2]&gt;)</td>
<td>x = o + 2</td>
<td>2 + 1</td>
</tr>
<tr>
<td>(&lt;x := 3 + 2, [x=2]&gt;)</td>
<td>x = o;</td>
<td>3 + 2</td>
</tr>
<tr>
<td>(&lt;x := 5, [x=2]&gt;)</td>
<td>o</td>
<td>x:=5</td>
</tr>
<tr>
<td>(&lt;\text{skip}, [x=5]&gt;)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The context is a program with a hole
Contexts

Contexts are defined by a grammar

\[ H ::= \text{o} | n + H | H + e | x := H \]

| if H then c1 else c2 | H; c |

The grammar defines the evaluation order
- Note in \( a + b \), \( a \) is evaluated before \( b \).

We can define redexes and contexts to
- define the order of evaluation
- define short circuit behavior
Contexts

How do we know if our contexts and redexes are well defined?

Decomposition theorem:

If \( c \) is not “skip”, then there exist unique \( H \) and \( r \) such that \( c \) is \( H[r] \)
- Exist guarantees progress
- Unique guarantees determinism