Automated Program Verification and Testing
15414/15614 Fall 2016
Lecture 10:
Introduction to Program Semantics

Matt Fredrikson
mfredrik@cs.cmu.edu

October 4, 2016

Today's Lecture

- See how to reason about programs mathematically
- Formalize meaning of programs: **operational semantics**
- Review inductive principles, see how to generalize to semantics
- Prove properties about programs

Lanugage Semantics

Language semantics specify what happens when programs evaluate

- Does the program terminate?
- Does an invariant hold on every execution?
- Is the language deterministic?
- Are two programs equivalent?

Think of a mathematical definition of the language

Approaches

How might we do this?

- Why not write a compiler? **Lots of irrelevant details.** Which way does the stack grow? How are registers allocated? Which instructions do we use?
- Why not write natural language docs? **Written language is ambiguous.** Easy to miss cases, difficult to make sure it's been done right.

Well-constructed semantics give us a way to specify meaning with assurances:

- Execution won't get "stuck" where it shouldn't
- Programs don't exhibit unexplained behavior
- Specifications mean what we intend

Operational Semantics

Today we'll look at **operational semantics**

- Define an abstract "machine" to execute programs on
- Describe how values are computed from machine states
- Describe how statements change machine states

Together, these elements define the meaning of programs

Imp: Syntax

We will examine an imperative language Imp

Before talking about semantics, we need to define syntax

- **Concrete syntax**: rules for expressing programs as sequences of characters
- **Abstract syntax**: simplified rules that ignore tokens without semantic meaning

Concrete syntax is important in practice for parsing, readability, etc.

When talking about semantics, we'll use abstract syntax

Imp: Syntactic Entities

The syntax of Imp has three categories

- **Arithmetic expressions** AExp denoted by $a, a_1, a_2, \ldots$
- **Boolean expressions** BExp denoted by $b, b_1, b_2, \ldots$
- **Commands** Com denoted by $c, c_1, c_2, \ldots$

Arithmetic expressions take values $n, n_1, n_2, \ldots$ in $\mathbb{Z}$

Boolean expressions take values in $\{$*true*$,$ *false*$\}$

Imp programs are always commands

We draw variables $x, x_1, x_2, \ldots$ from a set Var

Imp: Abstract Syntax

$$a \in \textbf{AExp} ::= n \in \mathbb{Z} \mid x \in \mathsf{Var} \mid a_1 + a_2 \mid a_1 \times a_2$$

$$b \in \textbf{BExp} ::= \mathsf{true} \mid \mathsf{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2$$

$$\begin{aligned} c \in \textbf{Com} ::= {} & \textbf{skip} \mid x := a \mid c_1; c_2 \\ & \mid \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \\ & \mid \textbf{while } b \textbf{ do } c \end{aligned}$$

Note: AExp and BExp can be **syntactic constants** $0, 1, \ldots, \mathsf{true}, \mathsf{false}$

These are in one-to-one correspondence with $\mathbb{Z}$ and $\{\textit{true}, \textit{false}\}$

Program States

Programs in Imp operate over integers

Their variables have values stored in the *environment*

We model the environment as a map $\sigma : \mathsf{Var} \mapsto \mathbb{Z}$

For Imp, we always assume that σ is **total**

To completely specify program state, we define a **configuration**

Configuration

A *configuration* is a pair $\langle c, \sigma \rangle$, where $c \in \mathsf{Com}$ is a command and σ is an environment. A configuration represents a *moment in time* during the computation of a program, where σ is the current assignment to variables and c is the next command to be executed.

Imp in Dafny

```
type Var = string
datatype AExp = N(n: int)
              | V(x: Var)
              | Plus(0: AExp, 1: AExp)
datatype BExp = B(v: bool)
              | Less(a0: AExp, a1: AExp)
              | Not(op: BExp)
              | And(0: BExp, 1: BExp)
datatype Com  = Skip
              | Assign(vname, aexp)
              | Seq(com, com)
              | If(bexp, com, com)
              | While(bexp, com)

type Env = map<Var, int>
type Config = Com * Env
```

Small-Step Operational Semantics

Idea: Specify operations **one step at a time**

- Formalize semantics as **transition relation over configurations**
- For each syntactic element, provide **inference rules**
- Apply transition rules until **final configuration** $\langle \textbf{skip}, \sigma \rangle$
- If the program reaches $\langle \textbf{skip}, \sigma \rangle$, we say that it **terminates**

We need to define three transition relations:

- $\rightarrow_a$: $(\text{AExp} \times \mathit{Env}) \mapsto \mathbb{Z}$ for evaluating arithmetic expressions
- $\rightarrow_b$: $(\text{BExp} \times \mathit{Env}) \mapsto \{\mathit{true}, \mathit{false}\}$ for Boolean expressions
- $\rightarrow$: $(\text{Com} \times \mathit{Env}) \mapsto (\text{Com} \times \mathit{Env})$ for commands

Imp: Small-step AExp (1)

$$a \in \textbf{AExp} \quad ::= \quad n \in \mathbb{Z} \mid x \in \mathsf{Var} \mid a_1 + a_2 \mid a_1 \times a_2$$

Let's start by defining the relation for $\rightarrow_a$

To evaluate a **variable** expression:

$$\text{Var}\ \frac{}{\langle x, \sigma \rangle \rightarrow_a \langle n, \sigma \rangle}\ \text{where } n = \sigma(x)$$

Why no rule for constants?

Constants are **irreducable**

No rules on irreducable entities, so no further computation

Imp: Small-step AExp (2)

$$a \in \textbf{AExp} \quad ::= \quad n \in \mathbb{Z} \mid x \in \mathsf{Var} \mid a_1 + a_2 \mid a_1 \times a_2$$

Now let's move on to the arithmetic operators

$$\text{Add}\ \frac{}{\langle n_1 + n_2, \sigma\rangle \to_a \langle n_3, \sigma\rangle} \text{ where } n_3 \text{ is the sum of } n_1, n_2$$

$$\text{LAdd}\ \frac{\langle a_1, \sigma\rangle \to_a a_1'}{\langle a_1 + a_2, \sigma\rangle \to_a \langle a_1' + a_2, \sigma\rangle} \qquad \text{RAdd}\ \frac{\langle a_2, \sigma\rangle \to_a a_2'}{\langle n + a_2, \sigma\rangle \to_a \langle n + a_2', \sigma\rangle}$$

The rules specify the order in which computations are performed

In this case, evaluate the left operand before the right

Imp: Small-step BExp (1)

$$b \in \textbf{BExp} \quad ::= \quad \mathsf{true} \mid \mathsf{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2$$

We can define semantics for Boolean expressions similarly

$$\text{EqTrue}\ \frac{}{\langle n_1 = n_2, \sigma\rangle \to_b \langle \mathsf{true}, \sigma\rangle}\ \text{if } n_1 \text{ equals } n_2$$

$$\text{EqFalse}\ \frac{}{\langle n_1 = n_2, \sigma\rangle \to_b \langle \mathsf{false}, \sigma\rangle}\ \text{if } n_1 \text{ not equals } n_2$$

$$\text{EqLeft}\ \frac{\langle a_1, \sigma\rangle \to_a a_1'}{\langle a_1 = a_2, \sigma\rangle \to_b \langle a_1' = a_2, \sigma\rangle} \qquad \text{EqRight}\ \frac{\langle a_2, \sigma\rangle \to_a a_2'}{\langle n = a_2, \sigma\rangle \to_b \langle n = a_2', \sigma\rangle}$$

The inequality operator is defined by replacing = with $\leq$

Imp: Small-step BExp (2)

$$b \in \textbf{BExp} \quad ::= \quad \mathsf{true} \mid \mathsf{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2$$

For Boolean connectives:

$$\text{NotTrue}\ \frac{}{\langle \neg \mathsf{true}, \sigma \rangle \to_b \langle \mathsf{false}, \sigma \rangle} \qquad \text{NotFalse}\ \frac{}{\langle \neg \mathsf{false}, \sigma \rangle \to_b \langle \mathsf{true}, \sigma \rangle}$$

$$\text{Not}\ \frac{\langle b, \sigma \rangle \to_b \langle b', \sigma \rangle}{\langle \neg b, \sigma \rangle \to_b \langle \neg b', \sigma \rangle}$$

For $\wedge$, we need four rules:

- AndLeft, AndRight to evaluate the operands in order
- AndTrue, AndFalse to reduce $\wedge$ over Boolean values

Example

Evaluate $(\mathsf{x} + 2) \times y$ under $\sigma = [\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 3]$

Start by applying MulLeft:

$$\text{MulLeft}\ \frac{\langle \mathsf{x} + 2, \sigma \rangle \to_a \langle 3, \sigma \rangle}{\langle (\mathsf{x} + 2) \times \mathsf{y}, \sigma \rangle \to_a \langle 3 \times \mathsf{y}, \sigma \rangle}$$

Now we must show that the premise $\langle \mathsf{x} + 2, \sigma \rangle \to_a \langle 3, \sigma \rangle$ holds

We apply AddLeft:

$$\text{AddLeft}\ \frac{\langle \mathsf{x}, \sigma \rangle \to_a \langle 1, \sigma \rangle}{\langle \mathsf{x} + 2, \sigma \rangle \to_a \langle 1 + 2, \sigma \rangle}$$

Example Contd.

Evaluate $(\mathsf{x}+2)\times y$ under $\sigma = [\mathsf{x}\mapsto 1, \mathsf{y}\mapsto 3]$

Now we need to show the premise $\langle \mathsf{x}, \sigma\rangle \to_a \langle 1, \sigma\rangle$

We apply Var:

$$\text{Var}\ \frac{}{\langle \mathsf{x}, \sigma\rangle \to_a \langle 1, \sigma\rangle}$$

because $\sigma(x) = 1$

Now we have $\langle \mathsf{x}+2, \sigma\rangle \to_a \langle 1+2, \sigma\rangle$

Apply Add:

$$\text{Add}\ \frac{}{\langle 1+2, \sigma\rangle \to_a \langle 3, \sigma\rangle}$$

Example Contd.

Evaluate $(\mathsf{x} + 2) \times y$ under $\sigma = [\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 3]$

Now we've justified application of the rule:

$$\text{MulLeft}\ \frac{\langle \mathsf{x} + 2, \sigma \rangle \to_a \langle 3, \sigma \rangle}{\langle (\mathsf{x} + 2) \times \mathsf{y}, \sigma \rangle \to_a \langle 3 \times \mathsf{y}, \sigma \rangle}$$

We did this by deriving a proof using rules from the semantics

We can summarize our reasoning with the **proof tree**:

$$\text{MulLeft}\ \frac{\text{AddLeft}\ \dfrac{\text{Var}\ \dfrac{}{\langle \mathsf{x}, \sigma \rangle \to_a \langle 1, \sigma \rangle}}{\langle \mathsf{x} + 2, \sigma \rangle \to_a \langle 1 + 2, \sigma \rangle} \qquad \text{Add}\ \dfrac{}{\langle 1 + 2, \sigma \rangle \to_a \langle 3, \sigma \rangle}}{\langle (\mathsf{x} + 2) \times \mathsf{y}, \sigma \rangle \to_a \langle 3 \times \mathsf{y}, \sigma \rangle}$$

Example Contd.

Evaluate $(\mathsf{x} + 2) \times y$ under $\sigma = [\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 3]$

But, we're not done:

$$\langle 3 \times \mathsf{y}, \sigma \rangle \text{ is reducible}$$

Next steps:

1. Apply MulRight to evaluate y in $3 \times \mathsf{y}$
2. Apply Var to evaluate y alone
3. From 3×3, apply Mul to derive 9
4. Now, 9 is irreducible

Imp: Small-step commands (1)

$$\begin{array}{rcl} c \in \textbf{Com} & ::= & \textbf{skip} \mid x := a \mid c_1; c_2 \\ & & \mid \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \\ & & \mid \textbf{while } b \textbf{ do } c \end{array}$$

Now let's assign semantics to the commands

Unlike expressions, commands can change the environment

skip has no rule

Assignment:

$$\text{Asgn1}\ \frac{\langle a, \sigma \rangle \rightarrow_a \langle a', \sigma \rangle}{\langle x := a, \sigma \rangle \rightarrow \langle x := a', \sigma \rangle}\ \text{Asgn2}\ \frac{}{\langle x := n, \sigma \rangle \rightarrow \langle \textbf{skip}, \sigma[x \mapsto n] \rangle}$$

$$
\begin{array}{rcl}
c \in \textbf{Com} & ::= & \textbf{skip} \mid x := a \mid c_1; c_2 \\
& & \mid \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \\
& & \mid \textbf{while } b \textbf{ do } c
\end{array}
$$

Composition $c_1; c_2$ requires two rules:

$$
\text{Seq1}\ \frac{\langle c_1, \sigma\rangle \to \langle c_1', \sigma'\rangle}{\langle c_1; c_2, \sigma\rangle \to \langle c_1'; c_2, \sigma'\rangle} \qquad \text{Seq2}\ \frac{}{\langle \textbf{skip}; c, \sigma\rangle \to \langle c, \sigma\rangle}
$$

Notice: in `Seq1`, the environment σ changes to σ'

Evaluating c_1 might have updated a variable, we account for this

Imp: Small-step commands (3)

$$c \in \textbf{Com} \quad ::= \quad \textbf{skip} \mid x := a \mid c_1; c_2 \mid \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \mid \textbf{while } b \textbf{ do } c$$

if commands introduce branching:

$$\text{If}\ \frac{\langle b, \sigma \rangle \to \langle b', \sigma \rangle}{\langle \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, \sigma \rangle \to \langle \textbf{if } b' \textbf{ then } c_1 \textbf{ else } c_2, \sigma \rangle}$$

$$\text{IfTrue}\ \frac{}{\langle \textbf{if } \mathsf{true} \textbf{ then } c_1 \textbf{ else } c_2, \sigma \rangle \to \langle c_1, \sigma \rangle}$$

$$\text{IfFalse}\ \frac{}{\langle \textbf{if } \mathsf{false} \textbf{ then } c_1 \textbf{ else } c_2, \sigma \rangle \to \langle c_2, \sigma \rangle}$$

Imp: Small-step commands (4)

$$c \in \textbf{Com} \quad ::= \quad \textbf{skip} \mid x := a \mid c_1 ; c_2 \mid \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \mid \textbf{while } b \textbf{ do } c$$

while command fits in a single rule!

$$\text{While}\ \frac{}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \rightarrow \langle \textbf{if } b \textbf{ then } (c;\ \textbf{while } b \textbf{ do } c) \textbf{ else skip}, \sigma\rangle}$$

Unroll a while loop one iteration

Only break when the **if** command evaluates *false*

Big-step operational semantics

Now we've defined a full semantics for Imp

We can talk about evaluations using $\rightarrow^*$, the transitive closure of $\rightarrow$

If $\langle c, \sigma \rangle$ is an initial configuration, we derive a sequence of intermediate configurations to reach $\langle \textbf{skip}, \sigma' \rangle$

We could have defined the semantics to directly give the result σ'

This is called **big-step operational semantics**, or **natural** semantics

Here, we define inference rules that give us judgements of the form:

$$\langle c, \sigma \rangle \Downarrow \sigma'$$

Imp: Big-step AExp

BigConst $\dfrac{}{\langle n, \sigma \rangle \Downarrow n}$ BigVar $\dfrac{}{\langle x, \sigma \rangle \Downarrow_a n}$ where $n = \sigma(x)$

BigAdd $\dfrac{\langle a_1, \sigma \rangle \Downarrow_a n_1 \qquad \langle a_2, \sigma \rangle \Downarrow_a n_2}{\langle a_1 + a_2, \sigma \rangle \Downarrow_a n}$ where n is the sum of n_1, n_2

BigMul $\dfrac{\langle a_1, \sigma \rangle \Downarrow_a n_1 \qquad \langle a_2, \sigma \rangle \Downarrow_a n_2}{\langle a_1 \times a_2, \sigma \rangle \Downarrow_a n}$ where n is the product of n_1, n_2

The rules for defining Boolean expression are similar

Imp: Big-step commands

$$\text{BigAsgn}\ \frac{\langle a, \sigma \rangle \Downarrow_a n}{\langle x := a, \sigma \rangle \Downarrow \sigma[x \mapsto n]} \qquad \text{BigSkip}\ \frac{}{\langle \textbf{skip}, \sigma \rangle \Downarrow \sigma}$$

$$\text{BigSeq}\ \frac{\langle c_1, \sigma_1 \rangle \Downarrow \sigma_1' \qquad \langle c_2, \sigma_1' \rangle \Downarrow \sigma_2}{\langle c_1; c_2, \sigma_1 \rangle \Downarrow \sigma_2}$$

$$\text{BigIfT}\ \frac{\langle b, \sigma \rangle \Downarrow_b \textit{true} \qquad \langle c_1, \sigma \rangle \Downarrow \sigma_2}{\langle \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2, \sigma \rangle \Downarrow \sigma_2} \qquad \text{BigIfF}\ \frac{\langle b, \sigma \rangle \Downarrow_b \textit{false} \qquad \langle c_2, \sigma \rangle \Downarrow \sigma_2}{\langle \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2, \sigma \rangle \Downarrow \sigma_2}$$

$$\text{BigWhileFalse}\ \frac{\langle b, \sigma \rangle \Downarrow_b \textit{false}}{\langle \textbf{while}\ b\ \textbf{do}\ c, \sigma \rangle \Downarrow \sigma}$$

$$\text{BigWhileTrue}\ \frac{\langle b, \sigma \rangle \Downarrow_b \textit{true} \qquad \langle c, \sigma \rangle \Downarrow \sigma' \qquad \langle \textbf{while}\ b\ \textbf{do}\ c, \sigma' \rangle \Downarrow \sigma''}{\langle \textbf{while}\ b\ \textbf{do}\ c, \sigma \rangle \Downarrow \sigma''}$$

Big-step vs. Small-step Semantics

Now we have two ways to assign meaning to Imp programs

Why have both?

- Big-step semantics are more natural in the sense that they model the recursive definition of the language
- Fewer rules in big-step semantics makes proving things easier; no need to worry about order of evaluation
- However, there are no intermediate states to speak of in big-step
- To the point, all non-terminating executions look the same—no derivable judgement!
- Small-step semantics can model properties of non-terminating executions
- They can also model things like concurrency and run-time errors

Example: Program Equialence (1)

We can prove program equivalence using the semantics

Let's try using big-step. What is the property?

$$c_0 \sim c_1 \text{ iff } \forall \sigma, \sigma'. \langle c_0, \sigma \rangle \Downarrow \sigma' \Leftrightarrow \langle c_1, \sigma \rangle \Downarrow \sigma'$$

The programs we'll prove:

c_0 = **while** b **do** c $\qquad$ c_1 = **if** b **then** $c;$ (**while** b **do** c) **else skip**

We need to show both directions of $\Leftrightarrow$

First we prove: $\forall \sigma, \sigma'. \langle c_0, \sigma \rangle \Downarrow \sigma' \Rightarrow \langle c_1, \sigma \rangle \Downarrow \sigma'$

Example: Program Equialence (2)

First we prove: $\forall \sigma, \sigma'. \langle c_0, \sigma \rangle \Downarrow \sigma' \Rightarrow \langle c_1, \sigma \rangle \Downarrow \sigma'$

Assuming $\langle \textbf{while } b \textbf{ do } c, \sigma \rangle \Downarrow \sigma'$

One of two cases holds regarding b. Either:

- b is *true*, so the last rule was BigWhileTrue.
- b is *false*, so the last rule was BigWhileFalse.

Suppose the former case, so BigWhileTrue.

Then there must be some derivation that takes the shape:

$$\text{BigWhileTrue}\ \frac{\dfrac{T_1}{\langle b, \sigma \rangle \Downarrow \textit{true}} \qquad \dfrac{T_2}{\langle c, \sigma \rangle \Downarrow \sigma''} \qquad \dfrac{T_3}{\langle \textbf{while } b \textbf{ do } c, \sigma'' \rangle \Downarrow \sigma'}}{\langle \textbf{while } b \textbf{ do } c, \sigma \rangle \Downarrow \sigma'}$$

Example: Program Equialence (3)

$$\text{BigWhileTrue}\ \frac{\dfrac{T_1}{\langle b, \sigma\rangle \Downarrow \textit{true}} \qquad \dfrac{T_2}{\langle c, \sigma\rangle \Downarrow \sigma''} \qquad \dfrac{T_3}{\langle \textbf{while}\ b\ \textbf{do}\ c, \sigma''\rangle \Downarrow \sigma'}}{\langle \textbf{while}\ b\ \textbf{do}\ c, \sigma\rangle \Downarrow \sigma'}$$

Recall, our goal is to show that:

$$\langle \textbf{if}\ b\ \textbf{then}\ c;\ (\textbf{while}\ b\ \textbf{do}\ c)\ \textbf{else skip}, \sigma\rangle \Downarrow \sigma'$$

We can use T_3 and T_3 with BigSeq to show:

$$\text{BigSeq}\ \frac{T_2 \qquad T_3}{\langle c;\ (\textbf{while}\ b\ \textbf{do}\ c), \sigma\rangle \Downarrow \sigma'}$$

Then T_1 and BigIfTrue to show:

$$\text{BigIfT}\ \frac{T_1 \qquad \text{BigSeq}\ \dfrac{T_2 \qquad T_2}{\langle c;\ (\textbf{while}\ b\ \textbf{do}\ c), \sigma\rangle \Downarrow \sigma'}}{\langle \textbf{if}\ b\ \textbf{then}\ c;\ (\textbf{while}\ b\ \textbf{do}\ c)\ \textbf{else skip}, \sigma\rangle \Downarrow \sigma'}$$

Example: Program Equialence (4)

This does it for the case where b is *true*.

Now for b is *false*.

In this case the derivation tree ends with:

$$\text{BigWhileF}\ \dfrac{\dfrac{T_4}{\langle b, \sigma \rangle \Downarrow \textit{false}}}{\langle \textbf{while}\ b\ \textbf{do}\ c, \sigma \rangle \Downarrow \sigma}$$

We can use T_4 with BigSkip and BigIfF:

$$\text{BigIfF}\ \dfrac{T_4 \qquad \text{BigSkip}\ \dfrac{}{\langle \textbf{skip}, \sigma \rangle \Downarrow \sigma}}{\langle \textbf{if}\ b\ \textbf{then}\ c;\ (\textbf{while}\ b\ \textbf{do}\ c)\ \textbf{else skip}, \sigma \rangle \Downarrow \sigma}$$

This concludes the direction $\forall \sigma, \sigma'. \langle c_0, \sigma \rangle \Downarrow \sigma' \Rightarrow \langle c_1, \sigma \rangle \Downarrow \sigma'$

Example: Program Equialence (5)

Now for the direction $\forall \sigma, \sigma'. \langle c_1, \sigma \rangle \Downarrow \sigma' \Rightarrow \langle c_0, \sigma \rangle \Downarrow \sigma'$

The last rule in the derivation is either BigIfT or BigIfF

Suppose that BigIfT:

$$\textsf{BigIfT}\ \frac{\dfrac{T_1}{\langle b, \sigma \rangle \Downarrow \textit{true}} \qquad \textsf{BigSeq}\ \dfrac{\dfrac{T_2}{\langle c, \sigma \rangle \Downarrow \sigma''} \qquad \dfrac{T_3}{\langle \textbf{while}\ b\ \textbf{do}\ c, \sigma'' \rangle \Downarrow \sigma'}}{\langle c;\ \textbf{while}\ b\ \textbf{do}\ c, \sigma \rangle \Downarrow \sigma'}}{\langle \textbf{if}\ b\ \textbf{then}\ c;\ (\textbf{while}\ b\ \textbf{do}\ c)\ \textbf{else skip}, \sigma \rangle \Downarrow \sigma'}$$

Now we can use $\mathsf{BigWhileTrue}$ with T_1, T_2, T_3:

$$\textsf{BigWhileTrue}\ \frac{T_1 \qquad T_2 \qquad T_3}{\langle \textbf{while}\ b\ \textbf{do}\ c, \sigma \rangle \Downarrow \sigma'}$$

Example: Program Equialence (6)

Now we move on to BigIfF:

$$\text{BigIfF}\ \dfrac{\dfrac{T_4}{\langle b, \sigma\rangle \Downarrow \textit{false}} \qquad \text{BigSkip}\ \dfrac{}{\langle \textbf{skip}, \sigma\rangle \Downarrow \sigma}}{\langle \textbf{if}\ b\ \textbf{then}\ c;\ (\textbf{while}\ b\ \textbf{do}\ c)\ \textbf{else skip}, \sigma\rangle \Downarrow \sigma}$$

Now we can use BigWhileFalse with T_4:

$$\text{BigWhileFalse}\ \dfrac{T_4}{\langle \textbf{while}\ b\ \textbf{do}\ c, \sigma\rangle \Downarrow \sigma}$$

This completes the proof.

Semantic Properties

We can also prove important properties about the semantics

- **Determinism**: For any $\sigma_1, \sigma_2, \sigma$ and command c, if $\langle c, \sigma \rangle \Downarrow \sigma_1$ and $\langle c, \sigma \rangle \Downarrow \sigma_2$, then $\sigma_1 = \sigma_2$:

$$\forall \sigma, \sigma_1, \sigma_2, c.(\langle c, \sigma \rangle \Downarrow \sigma_1 \land \langle c, \sigma \rangle \Downarrow \sigma_2) \rightarrow \sigma_1 = \sigma_2$$

- **Expression termination**: For any σ and arithmetic (Boolean) expression $e \in$ AExp ($e \in$ BExp), there is a value v such that $\langle e, \sigma \rangle \Downarrow v$:

$$\forall \sigma, e. \exists v. \langle e, \sigma \rangle \Downarrow v$$

To prove statements like these, we'll need to use induction

Induction

Recall our inductive axiom from T_{PA}

$$(F[0] \land (\forall x.F[x] \to F[x+1])) \to \forall x.F[x]$$

The goal is to prove $\forall x.F[x]$, i.e., F holds for all numbers

1. We begin by proving that $F[0]$ holds
2. We then prove that if $F[x]$ holds, then $F[x+1]$ holds

$F[0]$ is the **basis** of the induction

The assumption $F[x]$ is the **inductive hypothesis**

Establishing $F[x] \to F[x+1]$ is the **inductive step**

Inductive Sets

An **inductive set** is constructed using axioms and inference rules

For example, the syntax of Imp defines an inductive set:

$$a \in \textbf{AExp} \quad ::= \quad n \in \mathbb{Z} \mid x \in \mathsf{Var} \mid a_1 + a_2 \mid a_1 \times a_2$$

$$\frac{}{n \in \mathsf{AExp}}\ n \in \mathbb{Z} \qquad \frac{}{x \in \mathsf{AExp}}\ x \in \mathsf{Var} \qquad \frac{a_1 \in \mathsf{AExp} \qquad a_2 \in \mathsf{AExp}}{a_1 + a_2 \in \mathsf{AExp}}$$

Recall that rules without antecedents are called **axioms**

The semantic relations $\rightarrow, \rightarrow^*, \Downarrow$ are also inductive sets

As the name suggests, we can prove facts about these sets using inductive reasoning

Structural Induction

Structural Induction generalizes inductive reasoning to these sets

To prove that some property F holds on an inductively-defined set S:

1. **Basis**: Prove the base case for each axiom defining S. In other words, for each rule

$$\frac{}{s \in S}$$

prove $F[s]$

2. **Inductive step**: Unlike "traditional" induction, there are several inductive steps. For each inference rule:

$$\frac{s_1 \in S \qquad \cdots \qquad s_n \in S}{s \in S}$$

prove that $(s_1 \in S \land \cdots \land s_n \in S) \rightarrow s \in S$. Note the **inductive hypotheses** come from the antecedents of the rules.

Proving Semantic Properties

There are two primary ways to apply structural induction:

- **On program syntax**: Use the inductive set defined by Imp syntax rules, and induce on all possible syntactic constructions.
- **On semantic derivations**: Use the inductive set defined by either $\rightarrow$ or $\Downarrow$. This is often called **induction on derivations**.

Let's apply this to proving determinism of Imp:

$$\forall \sigma, \sigma_1, \sigma_2, c.(\langle c, \sigma \rangle \Downarrow \sigma_1 \wedge \langle c, \sigma \rangle \Downarrow \sigma_2) \rightarrow \sigma_1 = \sigma_2$$

This will be an induction on derivations for commands, structural induction for expressions

Proving Determinism of Imp (1)

$$\forall \sigma, a, n_1, n_2.(\langle a, \sigma \rangle \Downarrow n_1 \wedge \langle a, \sigma \rangle \Downarrow n_2) \rightarrow n_1 = n_2$$

First the expressions. We'll do AExp.

The base cases:

$$\text{BigConst}\ \frac{}{\langle n, \sigma \rangle \Downarrow n} \qquad \text{BigVar}\ \frac{}{\langle x, \sigma \rangle \Downarrow_a n}\ \text{where}\ n = \sigma(x)$$

- If the expression is a constant, there is only one rule (BigConst). We have that for all σ, $n_1 = n_2$.
- If the expression is a variable, then we have BigVar. Because σ is the same in both evaluations, we have $n_1 = n_2$.

Proving Determinism of Imp (2)

$$\forall \sigma, a, n, n'.(\langle a, \sigma\rangle \Downarrow n \wedge \langle a, \sigma\rangle \Downarrow n') \rightarrow n = n'$$

Now the inductive case:

$$\text{BigAdd}\ \frac{\langle a_1, \sigma\rangle \Downarrow_a n_1 \qquad \langle a_2, \sigma\rangle \Downarrow_a n_2}{\langle a_1 + a_2, \sigma\rangle \Downarrow_a n}\ \text{where } n \text{ is the sum of } n_1, n_2$$

If the expression is a sum, then the rule BigAdd applies.

We take as our inductive hypothesis that a_1 and a_2 are deterministic.

- Any derivation $\langle a, \sigma\rangle \Downarrow n$ must have $\langle a_1, \sigma\rangle \Downarrow n_1$ and $\langle a_1, \sigma\rangle \Downarrow n_2$ as premises.
- Any derivation $\langle a, \sigma\rangle \Downarrow n'$ must have $\langle a_1, \sigma\rangle \Downarrow n'_1$ and $\langle a_1, \sigma\rangle \Downarrow n'_2$ as premises.
- By the inductive hypothesis $n_1 + n_2 = n'_1 + n'_2 = n = n'$

Proving Determinism of Imp (3)

$$\forall \sigma, \sigma_1, \sigma_2, c.(\langle c, \sigma\rangle \Downarrow \sigma_1 \wedge \langle c, \sigma\rangle \Downarrow \sigma_2) \rightarrow \sigma_1 = \sigma_2$$

We said induction on derivations. Why not induction on syntax?

One of the cases will be for **while** b **do** c

Recall the rule BigWhileTrue:

$$\text{BigWhileTrue}\ \frac{\langle b, \sigma\rangle \Downarrow_b \textit{true} \qquad \langle c, \sigma\rangle \Downarrow \sigma' \qquad \langle \textbf{while}\ b\ \textbf{do}\ c, \sigma'\rangle \Downarrow \sigma''}{\langle \textbf{while}\ b\ \textbf{do}\ c, \sigma\rangle \Downarrow \sigma''}$$

One of the inductive hypotheses is not a proper sub-component of the original program!

This is not a well-founded induction.

Proving Determinism of Imp (4)

$$F : \forall \sigma, \sigma_1, \sigma_2, c.(\langle c, \sigma \rangle \Downarrow \sigma_1 \wedge \langle c, \sigma \rangle \Downarrow \sigma_2) \rightarrow \sigma_1 = \sigma_2$$

Instead, we'll show that if

$$\frac{T_1}{\langle c, \sigma \rangle \Downarrow \sigma_1} \qquad \frac{T_2}{\langle c, \sigma \rangle \Downarrow \sigma_2}$$

then $\sigma_1 = \sigma_2$

Our inductive hypothesis will be that T_1 and T_2 satisfy F

For the inductive step, we need to consider each operational semantics rule

Proving Determinism of Imp (5)

$$F : \forall \sigma, \sigma_1, \sigma_2, c. (\langle c, \sigma \rangle \Downarrow \sigma_1 \wedge \langle c, \sigma \rangle \Downarrow \sigma_2) \rightarrow \sigma_1 = \sigma_2$$

Begin with BigAsgn:

$$\text{BigAsgn} \; \frac{\langle a, \sigma \rangle \Downarrow_a n'}{\langle x := a, \sigma \rangle \Downarrow \sigma[x \mapsto n]}$$

So we have:

$$\text{BigAsgn} \; \frac{\dfrac{T_1}{\langle a, \sigma \rangle \Downarrow_a n}}{\langle x := a, \sigma \rangle \Downarrow \sigma[x \mapsto n]} \qquad \text{BigAsgn} \; \frac{\dfrac{T_2}{\langle a, \sigma \rangle \Downarrow_a n'}}{\langle x := a, \sigma \rangle \Downarrow \sigma[x \mapsto n']}$$

Because expressions are deterministic, we have $n = n'$, so $\sigma[x \mapsto n] = \sigma[x \mapsto n']$

Proving Determinism of Imp (6)

$$F : \forall \sigma, \sigma_1, \sigma_2, c.(\langle c, \sigma\rangle \Downarrow \sigma_1 \land \langle c, \sigma\rangle \Downarrow \sigma_2) \rightarrow \sigma_1 = \sigma_2$$

We'll jump to BigWhileTrue:

$$\text{BigWhileTrue}\ \frac{\langle b, \sigma\rangle \Downarrow_b \textit{true} \qquad \langle c, \sigma\rangle \Downarrow \sigma' \qquad \langle \textbf{while}\ b\ \textbf{do}\ c, \sigma'\rangle \Downarrow \sigma''}{\langle \textbf{while}\ b\ \textbf{do}\ c, \sigma\rangle \Downarrow \sigma''}$$

So we have:

$$\text{BigWhileTrue}\ \frac{\dfrac{T_1}{\langle b, \sigma\rangle \Downarrow_b \textit{true}} \qquad \dfrac{T_2}{\langle c, \sigma\rangle \Downarrow \sigma_1'} \qquad \dfrac{T_3}{\langle \textbf{while}\ b\ \textbf{do}\ c, \sigma_1'\rangle \Downarrow \sigma_1}}{\langle \textbf{while}\ b\ \textbf{do}\ c, \sigma\rangle \Downarrow \sigma_1}$$

$$\text{BigWhileTrue}\ \frac{\dfrac{T_4}{\langle b, \sigma\rangle \Downarrow_b \textit{true}} \qquad \dfrac{T_5}{\langle c, \sigma\rangle \Downarrow \sigma_2'} \qquad \dfrac{T_6}{\langle \textbf{while}\ b\ \textbf{do}\ c, \sigma_2'\rangle \Downarrow \sigma_2}}{\langle \textbf{while}\ b\ \textbf{do}\ c, \sigma\rangle \Downarrow \sigma_2}$$

Proving Determinism of Imp (7)

$$F : \forall \sigma, \sigma_1, \sigma_2, c.(\langle c, \sigma\rangle \Downarrow \sigma_1 \wedge \langle c, \sigma\rangle \Downarrow \sigma_2) \rightarrow \sigma_1 = \sigma_2$$

$$\text{BigWhileTrue}\ \frac{\dfrac{T_1}{\langle b, \sigma\rangle \Downarrow_b \textit{true}} \quad \dfrac{T_2}{\langle c, \sigma\rangle \Downarrow \sigma_1'} \quad \dfrac{T_3}{\langle \textbf{while}\ b\ \textbf{do}\ c, \sigma_1'\rangle \Downarrow \sigma_1}}{\langle \textbf{while}\ b\ \textbf{do}\ c, \sigma\rangle \Downarrow \sigma_1}$$

$$\text{BigWhileTrue}\ \frac{\dfrac{T_4}{\langle b, \sigma\rangle \Downarrow_b \textit{true}} \quad \dfrac{T_5}{\langle c, \sigma\rangle \Downarrow \sigma_2'} \quad \dfrac{T_6}{\langle \textbf{while}\ b\ \textbf{do}\ c, \sigma_2'\rangle \Downarrow \sigma_2}}{\langle \textbf{while}\ b\ \textbf{do}\ c, \sigma\rangle \Downarrow \sigma_2}$$

By ind. hypothesis on T_2, T_5, we have $\sigma_1' = \sigma_2'$

So we can apply ind. hyp. on T_3, T_6 giving $\sigma_1 = \sigma_2$.

Next Lecture

We'll leave the remaining cases as an exercise

Next lecture, we'll see how to automate some of this with Dafny

We'll move on to specifications of correctness