Harvard School of Engineering and Applied Sciences — CS 152: Programming Languages

Notes on Semantics

Spring 2017

1 Intro to semantics

What is the meaning of a program? When we write a program, we use a sequence of characters to represent the program. But this *syntax* is just how we represent the program: it is not what the program means.

Maybe we could define the meaning of a program to be whatever happens when we execute the program (perhaps using an interpreter, or by compiling it first). But we can have bugs in interpreters and compilers! That is, an interpreter or compiler may not accurately reflect the meaning of a program. So we must look elsewhere for a definition of what a program means.

One place to look for the meaning of a program is in the language specification manual. Such manuals typically give an informal description of the language constructs.

Another option is to give a formal, mathematical definition of the language semantics. A formal mathematical definition can have the following advantages over an informal description.

- **Less ambiguous.** The behavior of the language is clearer, which is useful for anyone who needs to write programs in the language, implement a compiler or interpreter for the language, add a new feature to the language, etc.

- **More concise.** Mathematical concepts and notation can clearly and concisely describe a language, and state restrictions on legal programs in the language. For example, the Java Language Specification (2rd edition) devotes a chapter (26 pages) to describing the concept of *definite assignment*, most of which is describing, in English, a dataflow analysis that can be expressed more succinctly using mathematics.

- **Formal arguments.** Most importantly, a formal semantics allows us to state, and prove, program properties that we're interested in. For example: we can state and prove that *all* programs in a language are guaranteed to be free of certain run-time errors, or free of certain security violations; we can state the specification of a program and prove that the program meets the specification (i.e., that the program is guaranteed to produce the correct output for all possible inputs).

However, the drawback of formal semantics is that they can lead to fairly complex mathematical models, especially if one attempts to describe all details in a full-featured modern language. Few real programming languages have a formal semantics, since modeling all the details of a real-world language is hard: real languages are complex, with many features. In order to describe these features, both the mathematics and notation for modeling can get very dense, making the formal semantics difficult to understand. Indeed, sometimes novel mathematics and notation needs to be developed to accurately model language features. So while there can be many benefits to having a formal semantics for a programming language, they do not yet outweigh the costs of developing and using a formal semantics for real programming languages.

There are three main approaches to formally specify the semantics of programming languages:

- operational semantics: describes how a program would execute on an abstract machine;

- denotational semantics: models programs as mathematical functions;

- axiomatic semantics: defines program behavior in terms of the logical formulae that are satisfied before and after a program;

Each of these approaches has different advantages and disadvantages in terms of how mathematically sophisticated they are, how easy it is to use them in proofs, or how easy it is to implement an interpreter or compiler based on them.

2 A simple language of arithmetic expressions

To understand some of the key concepts of semantics, we will start with a very simple language of integer arithmetic expressions, with assignment. A program in this language is an expression; executing a program means evaluating the expression to an integer.

To describe the structure of this language we will use the following domains:

$$\begin{aligned} x, y, z &\in \mathbf{Var} \\ n, m &\in \mathbf{Int} \\ e &\in \mathbf{Exp} \end{aligned}$$

Var is the set of program variables (e.g., foo, bar, baz, i, etc.). **Int** is the set of constant integers (e.g., 42, −40, 7). **Exp** is the domain of expressions, which we specify using a BNF (Backus-Naur Form) grammar:

$$e ::= x \mid n \mid e_1 + e_2 \mid e_1 \times e_2 \mid x := e_1; e_2$$

Informally, the expression $x := e_1; e_2$ means that x is assigned the value of e_1 before evaluating e_2. The result of the entire expression is that of e_2.

This grammar specifies the syntax for the language. An immediate problem here is that the grammar is ambiguous. Consider the expression $1 + 2 \times 3$. One can build two abstract syntax trees:

```
    +                                          *
   / \                                        / \
  1   *                                      +   3
     / \                                    / \
    2   3                                  1   2
```

There are several ways to deal with this problem. One is to rewrite the grammar for the same language to make it unambiguous. But that makes the grammar more complex, and harder to understand. Another possibility is to extend the syntax to require parentheses around all expressions:

$$x \mid n \mid (e_1 + e_2) \mid (e_1 \times e_2) \mid x := e_1; e_2$$

However, this also leads to unnecessary clutter and complexity.

Instead, we separate the "concrete syntax" of the language (which specifies how to unambiguously parse a string into program phrases) from the "abstract syntax" of the language (which describes, possibly ambiguously, the structure of program phrases). In this course we will use the abstract syntax and assume that the abstract syntax tree is known. When writing expressions, we will occasionally use parenthesis to indicate the structure of the abstract syntax tree, but the parentheses are not part of the language itself. (For details on parsing, grammars, and ambiguity elimination, see or take the compiler course Computer Science 153.)

3 Small-step operational semantics

At this point we have defined the syntax of our simple arithmetic language. We have some informal, intuitive notion of what programs in this language mean. For example, the program $7 + (4 \times 2)$ should equal 15, and the program foo $:= 6 + 1; 2 \times 3 \times$ foo should equal 42.

We would like now to define a formal semantics for this language.

Operational semantics describe how a program would execute on an abstract machine. A *small-step operational semantics* describe how such an execution in terms of successive reductions of an expression, until we reach a number, which represents the result of the computation.

The state of the abstract machine is usually referred to as a configuration, and for our language it must include two pieces of information:

- a store (aka environment or state), which assigns integer values to variables. During program execution, we will refer to the store to determine the values associated with variables, and also update the store to reflect assignment of new values to variables.
- the expression left to evaluate.

Thus, the domain of stores is functions from **Var** to **Int** (written **Var** → **Int**), and the domain of configurations is pairs of expressions and stores.

$$\begin{aligned}\textbf{Config} &= \textbf{Exp} \times \textbf{Store}\\ \textbf{Store} &= \textbf{Var} \to \textbf{Int}\end{aligned}$$

We will denote configurations using angle brackets. For instance, $\langle (\mathsf{foo} + 2) \times (\mathsf{bar} + 1), \sigma\rangle$, where σ is a store and $(\mathsf{foo} + 2) \times (\mathsf{bar} + 1)$ is an expression that uses two variables, foo and bar.

The small-step operational semantics for our language is a relation $\longrightarrow \subseteq \textbf{Config} \times \textbf{Config}$ that describes how one configuration transitions to a new configuration. That is, the relation $\longrightarrow$ shows us how to evaluate programs, one step at a time. We use infix notation for the relation $\longrightarrow$. That is, given any two configurations $\langle e_1, \sigma_1\rangle$ and $\langle e_2, \sigma_2\rangle$, if $(\langle e_1, \sigma_1\rangle, \langle e_2, \sigma_2\rangle)$ is in the relation $\longrightarrow$, then we write $\langle e_1, \sigma_1\rangle \longrightarrow \langle e_2, \sigma_2\rangle$.

For example, we have $\langle (4 + 2) \times \mathsf{y}, \sigma\rangle \longrightarrow \langle 6 \times \mathsf{y}, \sigma\rangle$. That is, we can evaluate the configuration $\langle (4 + 2) \times \mathsf{y}, \sigma\rangle$ by one step, to get the configuration $\langle 6 \times \mathsf{y}, \sigma\rangle$.

Now defining the semantics of the language boils down to defining the relation $\longrightarrow$ that describes the transitions between machine configurations.

One issue here is that the domain of integers is infinite, and so is the domain of expressions. Therefore, there is an infinite number of possible machine configurations, and an infinite number of possible one-step transitions. We need to use a finite description for the infinite set of transitions.

We can compactly describe the transition function $\longrightarrow$ using inference rules:

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

$$\textsc{LAdd}\ \frac{\langle e_1, \sigma\rangle \longrightarrow \langle e_1', \sigma'\rangle}{\langle e_1 + e_2, \sigma\rangle \longrightarrow \langle e_1' + e_2, \sigma'\rangle} \qquad \textsc{RAdd}\ \frac{\langle e_2, \sigma\rangle \longrightarrow \langle e_2', \sigma'\rangle}{\langle n + e_2, \sigma\rangle \longrightarrow \langle n + e_2', \sigma'\rangle}$$

$$\textsc{Add}\ \frac{}{\langle n + m, \sigma\rangle \longrightarrow \langle p, \sigma\rangle}\ \text{where } p \text{ is the sum of } n \text{ and } m$$

$$\textsc{LMul}\ \frac{\langle e_1, \sigma\rangle \longrightarrow \langle e_1', \sigma'\rangle}{\langle e_1 \times e_2, \sigma\rangle \longrightarrow \langle e_1' \times e_2, \sigma'\rangle} \qquad \textsc{RMul}\ \frac{\langle e_2, \sigma\rangle \longrightarrow \langle e_2', \sigma'\rangle}{\langle n \times e_2, \sigma\rangle \longrightarrow \langle n \times e_2', \sigma'\rangle}$$

$$\textsc{Mul}\ \frac{}{\langle n \times m, \sigma\rangle \longrightarrow \langle p, \sigma\rangle}\ \text{where } p \text{ is the product of } n \text{ and } m$$

$$\textsc{Asg1}\ \frac{\langle e_1, \sigma\rangle \longrightarrow \langle e_1', \sigma'\rangle}{\langle x := e_1; e_2, \sigma\rangle \longrightarrow \langle x := e_1'; e_2, \sigma'\rangle} \qquad \textsc{Asg}\ \frac{}{\langle x := n; e_2, \sigma\rangle \longrightarrow \langle e_2, \sigma[x \mapsto n]\rangle}$$

The meaning of an inference rule is that if the fact above the line holds and the side conditions also hold, then the fact below the line holds. The fact(s) above the line are called premises; the fact below the line is called the conclusion. The rules without premises are axioms; and the rules with premises are inductive rules.

Also, we use the notation $\sigma[x \mapsto n]$ for a store that maps the variable x to integer n, and maps every other variable to whatever σ maps it to. More explicitly, if f is the function $\sigma[x \mapsto n]$, then we have

$$f(y) = \begin{cases} n & \text{if } y = x \\ \sigma(y) & \text{otherwise} \end{cases}$$

4 Using the Semantic Rules

Let's see how we can use these rules. Suppose we want to evaluate expression $(\mathsf{foo} + 2) \times (\mathsf{bar} + 1)$ in a store σ where $\sigma(\mathsf{foo}) = 4$ and $\sigma(\mathsf{bar}) = 3$. That is, we want to find the transition for configuration $\langle (\mathsf{foo} + 2) \times (\mathsf{bar} + 1), \sigma \rangle$. For this, we look for a rule with this form of a configuration in the conclusion. By inspecting the rules, we find that the only matching rule is LMul, where $e_1 = \mathsf{foo} + 2$, $e_2 = \mathsf{bar} + 1$, but e_1' is not yet known. We can *instantiate* the rule LMul, replacing the metavariables e_1 and e_2 with appropriate expressions.

$$\textsc{LMul}\ \frac{\langle \mathsf{foo} + 2, \sigma \rangle \longrightarrow \langle e_1', \sigma \rangle}{\langle (\mathsf{foo} + 2) \times (\mathsf{bar} + 1), \sigma \rangle \longrightarrow \langle e_1' \times (\mathsf{bar} + 1), \sigma \rangle}$$

Now we need to show that the premise actually holds and find out what e_1' is. We look for a rule whose conclusion matches $\langle \mathsf{foo} + 2, \sigma \rangle \longrightarrow \langle e_1', \sigma \rangle$. We find that LAdd is the only matching rule:

$$\textsc{LAdd}\ \frac{\langle \mathsf{foo}, \sigma \rangle \longrightarrow \langle e_1'', \sigma \rangle}{\langle \mathsf{foo} + 2, \sigma \rangle \longrightarrow \langle e_1'' {+} 2, \sigma \rangle}$$

where $e_1' = e_1'' {+} 2$. We repeat this reasoning for $\langle \mathsf{foo}, \sigma \rangle \longrightarrow \langle e_1'', \sigma \rangle$, and we find that the only applicable rule is the axiom Var:

$$\textsc{Var}\ \frac{}{\langle \mathsf{foo}, \sigma \rangle \longrightarrow \langle 4, \sigma \rangle}$$

because we have $\sigma(\mathsf{foo}) = 4$. Since this is an axiom and has no premises, there is nothing left to prove. Hence, $e'' = 4$ and $e_1' = 4 + 2$. We can put together the above pieces and build the following proof:

$$\textsc{LMul}\ \frac{\textsc{LAdd}\ \dfrac{\textsc{Var}\ \dfrac{}{\langle \mathsf{foo}, \sigma \rangle \longrightarrow \langle 4, \sigma \rangle}}{\langle \mathsf{foo} + 2, \sigma \rangle \longrightarrow \langle 4 + 2, \sigma \rangle}}{\langle (\mathsf{foo} + 2) \times (\mathsf{bar} + 1), \sigma \rangle \longrightarrow \langle (4 + 2) \times (\mathsf{bar} + 1), \sigma \rangle}$$

This proves that, given our inference rules, the one-step transition $\langle (\mathsf{foo} + 2) \times (\mathsf{bar} + 1), \sigma \rangle \longrightarrow \langle (4 + 2) \times (\mathsf{bar} + 1), \sigma \rangle$ is possible. The above proof structure is called a "proof tree" or "derivation". It is important to keep in mind that proof trees must be finite for the conclusion to be valid.

We can use a similar reasoning to find out the next evaluation step:

$$\textsc{LMul}\ \frac{\textsc{Add}\ \dfrac{}{\langle 4 + 2, \sigma \rangle \longrightarrow \langle 6, \sigma \rangle}}{\langle (4 + 2) \times (\mathsf{bar} + 1), \sigma \rangle \longrightarrow \langle 6 \times (\mathsf{bar} + 1), \sigma \rangle}$$

And we can continue this process. At the end, we can put together all of these transitions, to get a view of the entire computation:

$$\begin{aligned} \langle (\mathsf{foo} + 2) \times (\mathsf{bar} + 1), \sigma \rangle &\longrightarrow \langle (4 + 2) \times (\mathsf{bar} + 1), \sigma \rangle \\ &\longrightarrow \langle 6 \times (\mathsf{bar} + 1), \sigma \rangle \\ &\longrightarrow \langle 6 \times (3 + 1), \sigma \rangle \\ &\longrightarrow \langle 6 \times 4, \sigma \rangle \\ &\longrightarrow \langle 24, \sigma \rangle \end{aligned}$$

The result of the computation is a number, 24. The machine configuration that contains the final result is the point where the evaluation stops; they are called *final configurations*. For our language of expressions, the final configurations are of the form $\langle n, \sigma \rangle$ where n is a number and σ is a store.

We write $\longrightarrow^*$ for the reflexive transitive closure of the relation $\longrightarrow$. That is, if $\langle e, \sigma \rangle \longrightarrow^* \langle e', \sigma' \rangle$, then using zero or more steps, we can evaluate the configuration $\langle e, \sigma \rangle$ to the configuration $\langle e', \sigma' \rangle$. Thus, we can write

$$\langle (\mathsf{foo} + 2) \times (\mathsf{bar} + 1), \sigma \rangle \longrightarrow^* \langle 24, \sigma \rangle.$$

5 Expressing Program Properties

Now that we have defined our small-step operational semantics, we can formally express different properties of programs. For instance:

- **Progress:** For each store σ and expression e that is not an integer, there exists a possible transition for $\langle e, \sigma \rangle$:
$$\forall e \in \mathbf{Exp}.\ \forall \sigma \in \mathbf{Store}.\ \text{either } e \in \mathbf{Int} \text{ or } \exists e', \sigma'.\ \langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle$$

- **Termination:** The evaluation of each expression terminates:
$$\forall e \in \mathbf{Exp}.\ \forall \sigma_0 \in \mathbf{Store}.\ \exists \sigma \in \mathbf{Store}.\ \exists n \in \mathbf{Int}.\ \langle e, \sigma_0 \rangle \longrightarrow^* \langle n, \sigma \rangle$$

- **Deterministic Result:** The evaluation result for any expression is deterministic:
$$\forall e \in \mathbf{Exp}.\ \forall \sigma_0, \sigma, \sigma' \in \mathbf{Store}.\ \forall n, n' \in \mathbf{Int}.$$
$$\text{if } \langle e, \sigma_0 \rangle \longrightarrow^* \langle n, \sigma \rangle \text{ and } \langle e, \sigma_0 \rangle \longrightarrow^* \langle n', \sigma' \rangle \text{ then } n = n' \text{ and } \sigma = \sigma'.$$

How can we prove such kinds of properties? *Inductive proofs* allow us to prove statements such as the properties above. We first introduce inductive sets, introduce inductive proofs, and then show how we can prove progress (the first property above) using inductive techniques.

6 Inductive sets

Induction is an important concept in the theory of programming language. We have already seen it used to define language syntax, and to define the small-step operational semantics for the arithmetic language.

An inductively-defined set[1] A is a set that is built using a set of axioms and inductive (inference) rules. Axioms of the form

$$\frac{}{a \in A}$$

indicate that a is in the set A. Inductive rules

$$\frac{a_1 \in A \qquad \ldots \qquad a_n \in A}{a \in A}$$

indicate that if $a_1, \ldots, a_n$ are all elements of A, then a is also an element of A.

The set A is the set of all elements that can be inferred to belong to A using a (finite) number of applications of these rules, starting only from axioms. In other words, for each element a of A, we must be able to construct a finite proof tree whose final conclusion is $a \in A$.

Example 1. The language of a grammar is an inductive set. For instance, the set of arithmetic expressions can be described with 2 axioms, and 3 inductive rules:

[1] or inductive set for short — but watch out, the term is pretty overloaded in the literature

$$\textsc{Var}\ \frac{}{x \in \mathbf{Exp}}\ x \in \mathbf{Var} \qquad \textsc{Int}\ \frac{}{n \in \mathbf{Exp}}\ n \in \mathbf{Int}$$

$$\textsc{Add}\ \frac{e_1 \in \mathbf{Exp} \quad e_2 \in \mathbf{Exp}}{e_1 + e_2 \in \mathbf{Exp}} \qquad \textsc{Mul}\ \frac{e_1 \in \mathbf{Exp} \quad e_2 \in \mathbf{Exp}}{e_1 \times e_2 \in \mathbf{Exp}} \qquad \textsc{Ass}\ \frac{e_1 \in \mathbf{Exp} \quad e_2 \in \mathbf{Exp}}{x := e_1; e_2 \in \mathbf{Exp}}\ x \in \mathbf{Var}$$

This is equivalent to the grammar $e ::= x \mid n \mid e_1 + e_2 \mid e_1 \times e_2 \mid x := e_1; e_2$.

To show that $(\mathsf{foo} + 3) \times \mathsf{bar}$ is an element of the set **Exp**, it suffices to show that $\mathsf{foo} + 3$ and bar are in the set **Exp**, since the inference rule Mul can be used, with $e_1 \equiv \mathsf{foo} + 3$ and $e_2 \equiv \mathsf{foo}$, and, since if the premises $\mathsf{foo} + 3 \in \mathbf{Exp}$ and $\mathsf{bar} \in \mathbf{Exp}$ are true, then the conclusion $(\mathsf{foo} + 3) \times \mathsf{bar} \in \mathbf{Exp}$ is true.

Similarly, we can use rule Add to show that if $\mathsf{foo} \in \mathbf{Exp}$ and $3 \in \mathbf{Exp}$, then $(\mathsf{foo} + 3) \in \mathbf{Exp}$. We can use axiom Var (twice) to show that $\mathsf{foo} \in \mathbf{Exp}$ and $\mathsf{bar} \in \mathbf{Exp}$ and rule Int to show that $3 \in \mathbf{Exp}$. We can put these all together into a derivation whose conclusion is $(\mathsf{foo} + 3) \times \mathsf{bar} \in \mathbf{Exp}$:

$$\textsc{Mul}\ \frac{\textsc{Add}\ \dfrac{\textsc{Var}\ \dfrac{}{\mathsf{foo} \in \mathbf{Exp}} \quad \textsc{Int}\ \dfrac{}{3 \in \mathbf{Exp}}}{(\mathsf{foo} + 3) \in \mathbf{Exp}} \qquad \textsc{Var}\ \dfrac{}{\mathsf{bar} \in \mathbf{Exp}}}{(\mathsf{foo} + 3) \times \mathsf{bar} \in \mathbf{Exp}}$$

Example 2. The natural numbers can be inductively defined:

$$\frac{}{0 \in \mathbb{N}} \qquad \frac{n \in \mathbb{N}}{\mathit{succ}(n) \in \mathbb{N}}$$

where $\mathit{succ}(n)$ is the successor of n.

Example 3. The small-step evaluation relation $\longrightarrow$ is an inductively defined set. The definition of this set is given by the semantic rules.

Example 4. The transitive, reflexive closure $\longrightarrow^*$ (i.e., the multi-step evaluation relation) can be inductively defined:

$$\frac{}{\langle e, \sigma \rangle \longrightarrow^* \langle e, \sigma \rangle} \qquad \frac{\langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle \qquad \langle e', \sigma' \rangle \longrightarrow^* \langle e'', \sigma'' \rangle}{\langle e, \sigma \rangle \longrightarrow^* \langle e'', \sigma'' \rangle}$$

7 Inductive proofs

We can prove facts about elements of an inductive set using an inductive reasoning that follows the structure of the set definition.

7.1 Inductive reasoning principle

The inductive reasoning principle for natural numbers can be stated as follows.

> For any property P,
> **If**
>
> - $P(0)$ holds
> - For all natural numbers n, if $P(n)$ holds then $P(n + 1)$ holds
>
> **then**
>
> for all natural numbers k, $P(k)$ holds.

This inductive reasoning principle gives us a technique to prove that a property holds for all natural numbers, which is an infinite set. Why is the inductive reasoning principle for natural numbers sound? That is, why does it work? One intuition is that for any natural number k you choose, k is either zero, or the result of applying the successor operation a finite number of times to zero. That is, we have a finite proof tree that k is a natural number, using the inference rules given in Example 2 of Lecture 2. Given this proof tree, the leaf of this tree is that $0 \in \mathbb{N}$. We know that $P(0)$ holds. Moreover, since we have for all natural numbers n, if $P(n)$ holds then $P(n+1)$ holds, and we have $P(0)$, we also have $P(1)$. Since we have $P(1)$, we also have $P(2)$, and so on. That is, for each node of the proof tree, we are showing that the property holds of that node. Eventually we will reach the root of the tree, that $k \in N$, and we will have $P(k)$.

For every inductively defined set, we have a corresponding inductive reasoning principle (often called *structural induction*). The template for this inductive reasoning principle, for an inductively defined set A, is as follows.

For any property P,
If

- **Base cases:** For each axiom

$$\frac{}{a \in A}\ ,$$

  $P(a)$ holds.

- **Inductive cases:** For each inference rule

$$\frac{a_1 \in A \quad \dots \quad a_n \in A}{a \in A}\ ,$$

  if $P(a_1)$ and $\dots$ and $P(a_n)$ then $P(a)$.

then

for all $a \in A$, $P(a)$ holds.

The intuition for why the inductive reasoning principle works is that same as the intuition for why mathematical induction works, i.e., for why the inductive reasoning principle for natural numbers works.

Let's consider a specific inductively defined set, and consider the inductive reasoning principle for that set: the set of arithmetic expressions **Exp**, inductively defined by the grammar

$$e ::= x \mid n \mid e_1 + e_2 \mid e_1 \times e_2 \mid x := e_1; e_2$$

Here is the inductive reasoning principle for the set **Exp**.

For any property P,
If

- For all variables x, $P(x)$ holds.
- For all integers n, $P(n)$ holds.
- For all $e_1 \in \mathbf{Exp}$ and $e_2 \in \mathbf{Exp}$, if $P(e_1)$ and $P(e_2)$ then $P(e_1 + e_2)$ holds.
- For all $e_1 \in \mathbf{Exp}$ and $e_2 \in \mathbf{Exp}$, if $P(e_1)$ and $P(e_2)$ then $P(e_1 \times e_2)$ holds.
- For all variables x and $e_1 \in \mathbf{Exp}$ and $e_2 \in \mathbf{Exp}$, if $P(e_1)$ and $P(e_2)$ then $P(x := e_1; e_2)$ holds.

then

for all $e \in \mathbf{Exp}$, $P(e)$ holds.

Here is the inductive reasoning principle for the small step relation on arithmetic expressions, i.e., for the set $\longrightarrow$.

For any property P,
If

- Var: For all variables x, stores σ and integers n such that $\sigma(x) = n$, $P(\langle x, \sigma\rangle \longrightarrow \langle n, \sigma\rangle)$ holds.
- Add: For all integers n, m, p such that $p = n + m$, and stores σ, $P(\langle n + m, \sigma\rangle \longrightarrow \langle p, \sigma\rangle)$ holds.
- Mul: For all integers n, m, p such that $p = n \times m$, and stores σ, $P(\langle n \times m, \sigma\rangle \longrightarrow \langle p, \sigma\rangle)$ holds.
- Asg: For all variables x, integers n and expressions $e \in \mathbf{Exp}$, $P(\langle x := n; e, \sigma\rangle \longrightarrow \langle e, \sigma[x \mapsto n]\rangle)$ holds.
- LAdd: For all expressions $e_1, e_2, e_1' \in \mathbf{Exp}$ and stores σ and σ', if $P(\langle e_1, \sigma\rangle \longrightarrow \langle e_1', \sigma'\rangle)$ holds then $P(\langle e_1 + e_2, \sigma\rangle \longrightarrow \langle e_1' + e_2, \sigma'\rangle)$ holds.
- RAdd: For all integers n, expressions $e_2, e_2' \in \mathbf{Exp}$ and stores σ and σ', if $P(\langle e_2, \sigma\rangle \longrightarrow \langle e_2', \sigma'\rangle)$ holds then $P(\langle n + e_2, \sigma\rangle \longrightarrow \langle n + e_2', \sigma'\rangle)$ holds.
- LMul: For all expressions $e_1, e_2, e_1' \in \mathbf{Exp}$ and stores σ and σ', if $P(\langle e_1, \sigma\rangle \longrightarrow \langle e_1', \sigma'\rangle)$ holds then $P(\langle e_1 \times e_2, \sigma\rangle \longrightarrow \langle e_1' \times e_2, \sigma'\rangle)$ holds.
- RMul: For all integers n, expressions $e_2, e_2' \in \mathbf{Exp}$ and stores σ and σ', if $P(\langle e_2, \sigma\rangle \longrightarrow \langle e_2', \sigma'\rangle)$ holds then $P(\langle n \times e_2, \sigma\rangle \longrightarrow \langle n \times e_2', \sigma'\rangle)$ holds.
- Asg1: For all variables x, expressions $e_1, e_2, e_1' \in \mathbf{Exp}$ and stores σ and σ', if $P(\langle e_1, \sigma\rangle \longrightarrow \langle e_1', \sigma'\rangle)$ holds then $P(\langle x := e_1; e_2, \sigma\rangle \longrightarrow \langle x := e_1'; e_2, \sigma'\rangle)$ holds.

then

for all $\langle e, \sigma\rangle \longrightarrow \langle e', \sigma'\rangle$, $P(\langle e, \sigma\rangle \longrightarrow \langle e', \sigma'\rangle)$ holds.

Note that there is one case for each inference rule: 4 axioms (Var, Add, Mul and Asg) and 5 inductive rules (LAdd, RAdd, LMul, RMul, Asg1).

The inductive reasoning principles give us a technique for showing that a property holds of every element in an inductively defined set. Let's consider some examples. Make sure you understand how the appropriate inductive reasoning principle is being used in each of these examples.

7.2 Example: Proving progress

Let's consider the progress property defined above, and repeated here:

Progress: For each store σ and expression e that is not an integer, there exists a possible transition for $\langle e, \sigma\rangle$:

$$\forall e \in \mathbf{Exp}.\ \forall \sigma \in \mathbf{Store}.\ \text{either } e \in \mathbf{Int} \text{ or } \exists e', \sigma'.\ \langle e, \sigma\rangle \longrightarrow \langle e', \sigma'\rangle$$

Let's rephrase this property as: for all expressions e, $P(e)$ holds, where:

$$P(e) = \forall \sigma.\ (e \in \mathbf{Int}) \vee (\exists e', \sigma'.\ \langle e, \sigma\rangle \longrightarrow \langle e', \sigma'\rangle)$$

The idea is to build a proof that follows the inductive structure in the grammar of expressions:

$$e ::= x \mid n \mid e_1 + e_2 \mid e_1 \times e_2 \mid x := e_1; e_2.$$

This is called "structural induction on the expressions e". We must examine each case in the grammar and show that $P(e)$ holds for that case. Since the grammar productions $e = e_1 + e_2$ and $e = e_1 \times e_2$ and $e = x := e_1; e_2$ are inductive definitions of expressions, they are inductive steps in the proof; the other two cases $e = x$ and $e = n$ are the basis of induction. The proof goes as follows:

We will show by structural induction that for all expressions e we have

$$P(e) = \forall \sigma.\ (e \in \mathbf{Int}) \vee (\exists e', \sigma'.\ \langle e, \sigma\rangle \longrightarrow \langle e', \sigma'\rangle).$$

Consider the possible cases for e.

- Case $e = x$. By the Var axiom, we can evaluate $\langle x, \sigma \rangle$ in any state: $\langle x, \sigma \rangle \longrightarrow \langle n, \sigma \rangle$, where $n = \sigma(x)$. So $e' = n$ is a witness that there exists e' such that $\langle x, \sigma \rangle \longrightarrow \langle e', \sigma \rangle$, and $P(x)$ holds.

- Case $e = n$. Then $e \in \textbf{Int}$, so $P(n)$ trivially holds.

- Case $e = e_1 + e_2$. This is an inductive step. The inductive hypothesis is that P holds for subexpressions e_1 and e_2. We need to show that P holds for e. In other words, we want to show that $P(e_1)$ and $P(e_2)$ implies $P(e)$. Let's expand these properties. We know that the following hold:

$$P(e_1) = \forall \sigma.\ (e_1 \in \textbf{Int}) \vee (\exists e', \sigma'.\ \langle e_1, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle)$$
$$P(e_2) = \forall \sigma.\ (e_2 \in \textbf{Int}) \vee (\exists e', \sigma'.\ \langle e_2, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle)$$

  and we want to show:

$$P(e) = \forall \sigma.\ (e \in \textbf{Int}) \vee (\exists e', \sigma'.\ \langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle)$$

  We must inspect several subcases.

  First, if both e_1 and e_2 are integer constants, say $e_1 = n_1$ and $e_2 = n_2$, then by rule Add we know that the transition $\langle n_1 + n_2, \sigma \rangle \longrightarrow \langle n, \sigma \rangle$ is valid, where n is the sum of n_1 and n_2. Hence, $P(e) = P(n_1 + n_2)$ holds (with witness $e' = n$).

  Second, if e_1 is not an integer constant, then by the inductive hypothesis $P(e_1)$ we know that $\langle e_1, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle$ for some e' and σ'. We can then use rule LAdd to conclude $\langle e_1 + e_2, \sigma \rangle \longrightarrow \langle e' + e_2, \sigma' \rangle$, so $P(e) = P(e_1 + e_2)$ holds.

  Third, if e_1 is an integer constant, say $e_1 = n_1$, but e_2 is not, then by the inductive hypothesis $P(e_2)$ we know that $\langle e_2, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle$ for some e' and σ'. We can then use rule RAdd to conclude $\langle n_1 + e_2, \sigma \rangle \longrightarrow \langle n_1 + e', \sigma' \rangle$, so $P(e) = P(n_1 + e_2)$ holds.

- Case $e = e_1 \times e_2$ and case $e = x := e_1; e_2$. These are also inductive cases, and their proofs are similar to the previous case. [Note that if you were writing this proof out for a homework, you should write these cases out in full.]

7.3 A recipe for inductive proofs

In this class, you will be asked to write inductive proofs. Until you are used to doing them, inductive proofs can be difficult. Here is a recipe that you should follow when writing inductive proofs. Note that this recipe was followed above.

1. State what you are inducting over. In the example above, we are doing structural induction on the expressions e.

2. State the property P that you are proving by induction. (Sometimes, as in the proof above the property P will be essentially identical to the theorem/lemma/property that you are proving; other times the property we prove by induction will need to be stronger than theorem/lemma/property you are proving in order to get the different cases to go through.)

3. Make sure you know the inductive reasoning principle for the set you are inducting on.

4. Go through each case. For each case, don't be afraid to be verbose, spelling out explicitly how the meta-variables in an inference rule are instantiated in this case.

7.4 Example: the store changes incremental

Let's see another example of an inductive proof, this time doing an induction on the derivation of the small step operational semantics relation. The property we will prove is that for all expressions e and stores σ, if $\langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle$ then either $\sigma = \sigma'$ or there is some variable x and integer n such that $\sigma' = \sigma[x \mapsto n]$. That is, in one small step, either the new store is identical to the old store, or is the result of updating a single program variable.

Theorem 1. *For all expressions e and stores σ, if $\langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle$ then either $\sigma = \sigma'$ or there is some variable x and integer n such that $\sigma' = \sigma[x \mapsto n]$.*

Proof of Theorem 1. We proceed by induction on the derivation of $\langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle$. Suppose we have e, σ, e' and σ' such that $\langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle$. The property P that we will prove of e, σ, e' and σ', which we will write as $P(\langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle)$, is that either $\sigma = \sigma'$ or there is some variable x and integer n such that $\sigma' = \sigma[x \mapsto n]$:

$$P(\langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle) \triangleq \sigma = \sigma' \vee (\exists x \in \mathbf{Var}, n \in \mathbf{Int}.\ \sigma' = \sigma[x \mapsto n]).$$

Consider the cases for the derivation of $\langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle$.

- Case Add. This is an axiom. Here, $e \equiv n + m$ and $e' = p$ where p is the sum of m and n, and $\sigma' = \sigma$. The result holds immediately.
- Case LAdd. This is an inductive case. Here, $e \equiv e_1 + e_2$ and $e' \equiv e_1' + e_2$ and $\langle e_1, \sigma \rangle \longrightarrow \langle e_1', \sigma' \rangle$. By the inductive hypothesis, applied to $\langle e_1, \sigma \rangle \longrightarrow \langle e_1', \sigma' \rangle$, we have that either $\sigma = \sigma'$ or there is some variable x and integer n such that $\sigma' = \sigma[x \mapsto n]$, as required.
- Case Asg. This is an axiom. Here $e \equiv x := n; e_2$ and $e' \equiv e_2$ and $\sigma' = \sigma[x \mapsto n]$. The result holds immediately.
- We leave the other cases (Var, RAdd, LMul, RMul, Mul, and Asg1) as exercises for the reader. Seriously, try them. Make sure you can do them. Go on, you're reading these notes, you may as well try the exercise.

□

8 Large-step semantics

So far we have defined the small-step evaluation relation $\longrightarrow \subseteq \mathbf{Config} \times \mathbf{Config}$ for our simple language of arithmetic expressions, and used its transitive and reflexive closure $\longrightarrow^*$ to describe the execution of multiple steps of evaluation. In particular, if $\langle e, \sigma \rangle$ is some start configuration, and $\langle n, \sigma' \rangle$ is a final configuration, the evaluation $\langle e, \sigma \rangle \longrightarrow^* \langle n, \sigma' \rangle$ shows that by executing expression e starting with the store σ, we get the result n, and the final store σ'.

Large-step semantics is an alternative way to specify the operational semantics of a language. Large-step semantics directly give the final result.

We'll use the same configurations as before, but define a large-step evaluation relation:

$$\Downarrow \subseteq \mathbf{Config} \times \mathbf{FinalConfig}$$

where

$$\mathbf{Config} = \mathbf{Exp} \times \mathbf{Store}$$
$$\text{and } \mathbf{Final\ Config} = \mathbf{Int} \times \mathbf{Store} \subseteq \mathbf{Config}.$$

We write $\langle e, \sigma \rangle \Downarrow \langle n, \sigma' \rangle$ to mean that $(\langle e, \sigma \rangle, \langle n, \sigma' \rangle) \in \Downarrow$. In other words, configuration $\langle e, \sigma \rangle$ evaluates in one large step directly to final configuration $\langle n, \sigma' \rangle$. In general, the large-step semantics takes a configuration to an “answer”. For our language of arithmetic expressions, “answers” are a subset of configurations, but this is not always true in general.

The large-step semantics boils down to defining the relation $\Downarrow$. We use inference rules to inductively define the relation $\Downarrow$, similar to how we specified the small-step operational semantics $\longrightarrow$.

$$\textsc{Int}_{\textsc{Lrg}}\ \frac{}{\langle n, \sigma\rangle \Downarrow \langle n, \sigma\rangle} \qquad\qquad \textsc{Var}_{\textsc{Lrg}}\ \frac{}{\langle x, \sigma\rangle \Downarrow \langle n, \sigma\rangle}\ \text{where } \sigma(x) = n$$

$$\textsc{Add}_{\textsc{Lrg}}\ \frac{\langle e_1, \sigma\rangle \Downarrow \langle n_1, \sigma''\rangle \qquad \langle e_2, \sigma''\rangle \Downarrow \langle n_2, \sigma'\rangle}{\langle e_1 + e_2, \sigma\rangle \Downarrow \langle n, \sigma'\rangle}\ \text{where } n \text{ is the sum of } n_1 \text{ and } n_2$$

$$\textsc{Mul}_{\textsc{Lrg}}\ \frac{\langle e_1, \sigma\rangle \Downarrow \langle n_1, \sigma''\rangle \qquad \langle e_2, \sigma''\rangle \Downarrow \langle n_2, \sigma'\rangle}{\langle e_1 \times e_2, \sigma\rangle \Downarrow \langle n, \sigma'\rangle}\ \text{where } n \text{ is the product of } n_1 \text{ and } n_2$$

$$\textsc{Asg}_{\textsc{Lrg}}\ \frac{\langle e_1, \sigma\rangle \Downarrow \langle n_1, \sigma''\rangle \qquad \langle e_2, \sigma''[x \mapsto n_1]\rangle \Downarrow \langle n_2, \sigma'\rangle}{\langle x := e_1; e_2, \sigma\rangle \Downarrow \langle n_2, \sigma'\rangle}$$

To see how we use these rules, here is a proof tree that shows that $\langle \mathsf{foo} := 3; \mathsf{foo} \times \mathsf{bar}, \sigma\rangle \Downarrow \langle 21, \sigma'\rangle$ for a store σ such that $\sigma(\mathsf{bar}) = 7$, and $\sigma' = \sigma[\mathsf{foo} \mapsto 3]$.

$$\textsc{Asg}_{\textsc{Lrg}}\ \frac{\textsc{Int}_{\textsc{Lrg}}\ \dfrac{}{\langle 3, \sigma\rangle \Downarrow \langle 3, \sigma\rangle} \qquad \textsc{Mul}_{\textsc{Lrg}}\ \dfrac{\textsc{Var}_{\textsc{Lrg}}\ \dfrac{}{\langle \mathsf{foo}, \sigma'\rangle \Downarrow \langle 3, \sigma'\rangle} \qquad \textsc{Var}_{\textsc{Lrg}}\ \dfrac{}{\langle \mathsf{bar}, \sigma'\rangle \Downarrow \langle 7, \sigma'\rangle}}{\langle \mathsf{foo} \times \mathsf{bar}, \sigma'\rangle \Downarrow \langle 21, \sigma'\rangle}}{\langle \mathsf{foo} := 3; \mathsf{foo} \times \mathsf{bar}, \sigma\rangle \Downarrow \langle 21, \sigma'\rangle}$$

A closer look to this structure reveals the relation between small step and large-step evaluation: a depth-first traversal of the large-step proof tree yields the sequence of one-step transitions in small-step evaluation.

9 Equivalence of semantics

So far, we have specified the semantics of our language of arithmetic expressions in two different ways: small-step operational semantics and large-step operational semantics. Are they expressing the same meaning of arithmetic expressions? Can we show that they express the same thing?

Theorem (Equivalence of semantics). *For all expressions e, stores σ and σ', and integers n, we have:*

$$\langle e, \sigma\rangle \Downarrow \langle n, \sigma'\rangle \iff \langle e, \sigma\rangle \longrightarrow^* \langle n, \sigma'\rangle.$$

Proof sketch.

- $\Longrightarrow$. We proceed by structural induction on expressions e. The inductive hypothesis is:

  $$P(e) = \forall \sigma, \sigma' \in \textbf{Store}.\ , \forall n \in \textbf{Int}.\ \langle e, \sigma\rangle \Downarrow \langle n, \sigma'\rangle \Longrightarrow \langle e, \sigma\rangle \longrightarrow^* \langle n, \sigma'\rangle$$

  We have to consider each of the possible axioms and inference rules for constructing an expression.

  – **Case $e \equiv n$.**
  Here, we are consider the case where expression e is equal to some integer n. But then $\langle n, \sigma\rangle \longrightarrow^* \langle n, \sigma\rangle$ holds trivially because of reflexivity of $\longrightarrow^*$.

- **Case** $e \equiv x$**.**

  Here, we are considering the case where the expression e is equal to some variable x. Assume that for some σ, σ', and n we have $\langle x, \sigma\rangle \Downarrow \langle n, \sigma'\rangle$. That means that there is some derivation using the axioms and inference rules of the large-step operational semantics, whose conclusion is $\langle x, \sigma\rangle \Downarrow \langle n, \sigma'\rangle$. There is only one rule whose conclusion could look like this, the rule $\mathrm{Var}_{\mathrm{Lrg}}$. That rule requires that $n = \sigma(x)$, and that $\sigma' = \sigma$.

  (This reasoning is an example of *inversion*: using the inference rules in reverse. That is, we know that some conclusion holds—$\langle x, \sigma\rangle \Downarrow \langle n, \sigma'\rangle$—and we examine the inference rules to determine which rule must have been used in the derivation, and thus which premises must be true, and which side conditions satisfied.)

  Since $n = \sigma(x)$ we know that $\langle x, \sigma\rangle \longrightarrow \langle n, \sigma\rangle$ also holds, by using the small-step axiom $\textsc{Var}$. So we can conclude that $\langle x, \sigma\rangle \longrightarrow^* \langle n, \sigma\rangle$ holds, which is what we needed to show.

- **Case** $e \equiv e_1 + e_2$**.**

  This is an inductive case. Expressions e_1 and e_2 are subexpressions of e, and so we can assume that $P(e_1)$ and $P(e_2)$ hold. We need to show that $P(e)$ holds. Let's write out $P(e_1)$, $P(e_2)$, and $P(e)$ explicitly.

$$\begin{aligned}
P(e_1) &= \forall n, \sigma, \sigma' : \langle e_1, \sigma\rangle \Downarrow \langle n, \sigma'\rangle \Longrightarrow \langle e_1, \sigma\rangle \longrightarrow^* \langle n, \sigma'\rangle \\
P(e_2) &= \forall n, \sigma, \sigma' : \langle e_2, \sigma\rangle \Downarrow \langle n, \sigma'\rangle \Longrightarrow \langle e_2, \sigma\rangle \longrightarrow^* \langle n, \sigma'\rangle \\
P(e) &= \forall n, \sigma, \sigma' : \langle e_1 + e_2, \sigma\rangle \Downarrow \langle n, \sigma'\rangle \Longrightarrow \langle e_1 + e_2, \sigma\rangle \longrightarrow^* \langle n, \sigma'\rangle
\end{aligned}$$

  Assume that for some σ, σ' and n we have $\langle e_1 + e_2, \sigma\rangle \Downarrow \langle n, \sigma'\rangle$. We now need to show that $\langle e_1 + e_2, \sigma\rangle \longrightarrow^* \langle n, \sigma'\rangle$.

  We assumed that $\langle e_1 + e_2, \sigma\rangle \Downarrow \langle n, \sigma'\rangle$. Let's use inversion again: there is some derivation whose conclusion is $\langle e_1 + e_2, \sigma\rangle \Downarrow \langle n, \sigma'\rangle$. By looking at the large-step semantic rules, we see that only one rule could possible have a conclusion of this form: the rule $\textsc{Add}_{\textsc{Lrg}}$. So that means that the last rule use in the derivation was $\textsc{Add}_{\textsc{Lrg}}$. But in order to use the rule $\textsc{Add}_{\textsc{Lrg}}$, it must be the case that $\langle e_1, \sigma\rangle \Downarrow \langle n_1, \sigma''\rangle$ and $\langle e_2, \sigma''\rangle \Downarrow \langle n_2, \sigma'\rangle$ hold for some n_1 and n_2 such that $n = n_1 + n_2$ (i.e., there is a derivation whose conclusion is $\langle e_1, \sigma\rangle \Downarrow \langle n_1, \sigma''\rangle$ and a derivation whose conclusion is $\langle e_2, \sigma''\rangle \Downarrow \langle n_2, \sigma'\rangle$).

  Using the inductive hypothesis $P(e_1)$, since $\langle e_1, \sigma\rangle \Downarrow \langle n_1, \sigma''\rangle$, we must have $\langle e_1, \sigma\rangle \longrightarrow^* \langle n_1, \sigma''\rangle$. Similarly, by $P(e_2)$, we have $\langle e_2, \sigma''\rangle \longrightarrow^* \langle n_2, \sigma\rangle$. By Lemma 1 below, we have

$$\langle e_1 + e_2, \sigma\rangle \longrightarrow^* \langle n_1 + e_2, \sigma''\rangle$$

  and by another application of Lemma 1 we have

$$\langle n_1 + e_2, \sigma''\rangle \longrightarrow^* \langle n_1 + n_2, \sigma'\rangle$$

  and by the rule $\textsc{Add}$ we have

$$\langle n_1 + n_2, \sigma'\rangle \longrightarrow \langle n, \sigma'\rangle.$$

  Thus, we have $\langle e_1 + e_2, \sigma\rangle \longrightarrow^* \langle n, \sigma'\rangle$, which proves this case.

- **Case** $e \equiv e_1 \times e_2$**.** Similar to the case $e = e_1 + e_2$ above.
- **Case** $e \equiv x := e_1; e_2$**.** Omitted. Try it as an exercise.

- $\Longleftarrow$. We proceed by mathematical induction on the number of steps $\langle e, \sigma\rangle \longrightarrow^* \langle n, \sigma'\rangle$.

  - **Base case.** If $\langle e, \sigma\rangle \longrightarrow^* \langle n, \sigma'\rangle$ in zero steps, then we must have $e \equiv n$ and $\sigma' = \sigma$. Then, $\langle n, \sigma\rangle \Downarrow \langle n, \sigma\rangle$ by the large-step operational semantics rule $\textsc{Int}_{\textsc{Lrg}}$.
  - **Inductive case.** Assume that $\langle e, \sigma\rangle \longrightarrow \langle e'', \sigma''\rangle \longrightarrow^* \langle n, \sigma'\rangle$, and that (the inductive hypothesis) $\langle e'', \sigma''\rangle \Downarrow \langle n, \sigma'\rangle$. That is, $\langle e'', \sigma''\rangle \longrightarrow^* \langle n, \sigma'\rangle$ takes m steps, and we assume that the property holds for it ($\langle e'', \sigma''\rangle \Downarrow \langle n, \sigma'\rangle$), and we are considering $\langle e, \sigma\rangle \longrightarrow^* \langle n, \sigma'\rangle$, which takes $m+1$ steps. We need to show that $\langle e, \sigma\rangle \Downarrow \langle n, \sigma'\rangle$. This follows immediately from Lemma 2 below.

□

Lemma 1. *If $\langle e, \sigma \rangle \longrightarrow^* \langle n, \sigma' \rangle$ then for all n_1, e_2 the following hold.*

- $\langle e + e_2, \sigma \rangle \longrightarrow^* \langle n + e_2, \sigma' \rangle$
- $\langle e \times e_2, \sigma \rangle \longrightarrow^* \langle n \times e_2, \sigma' \rangle$
- $\langle n_1 + e, \sigma \rangle \longrightarrow^* \langle n_1 + n, \sigma' \rangle$
- $\langle n_1 \times e, \sigma \rangle \longrightarrow^* \langle n_1 \times n, \sigma' \rangle$

Proof. By (mathematical) induction on the number of evaluation steps in $\longrightarrow^*$. □

Lemma 2. *For all e, e', σ, and n, if $\langle e, \sigma \rangle \longrightarrow \langle e', \sigma'' \rangle$ and $\langle e', \sigma'' \rangle \Downarrow \langle n, \sigma' \rangle$, then $\langle e, \sigma \rangle \Downarrow \langle n, \sigma' \rangle$.*

10 IMP: a simple imperative language

We shall now consider a more realistic programming language, one where we can assign values to variables and execute control constructs such as $\textsf{if}$ and $\textsf{while}$. The syntax for this simple imperative language, called IMP, is as follows:

arithmetic expressions	$a \in \mathbf{Aexp}$	$a ::= \; x \mid n \mid a_1 + a_2 \mid a_1 \times a_2$
boolean expressions	$b \in \mathbf{Bexp}$	$b ::= \; \textsf{true} \mid \textsf{false} \mid a_1 < a_2$
commands	$c \in \mathbf{Com}$	$c ::= \; \textsf{skip} \mid x := a \mid c_1; c_2$
		$\mid \textsf{if } b \textsf{ then } c_1 \textsf{ else } c_2$
		$\mid \textsf{while } b \textsf{ do } c$

10.1 Small-step operational semantics

We'll first give a small-step operational semantics for IMP. The configurations in this language are of the form $\langle c, \sigma \rangle$, $\langle b, \sigma \rangle$, and $\langle a, \sigma \rangle$, where σ is a store. The final configurations are of the form $\langle \textsf{skip}, \sigma \rangle$, $\langle \textsf{true}, \sigma \rangle$, $\langle \textsf{false}, \sigma \rangle$, and $\langle n, \sigma \rangle$. There are three different small-step operational semantics relations, one each for commands, boolean expressions, and arithmetic expressions.

$$\begin{aligned}
\longrightarrow_{\mathbf{Com}} &\subseteq \mathbf{Com} \times \mathbf{Store} \times \mathbf{Com} \times \mathbf{Store} \\
\longrightarrow_{\mathbf{Bexp}} &\subseteq \mathbf{Bexp} \times \mathbf{Store} \times \mathbf{Bexp} \times \mathbf{Store} \\
\longrightarrow_{\mathbf{Aexp}} &\subseteq \mathbf{Aexp} \times \mathbf{Store} \times \mathbf{Aexp} \times \mathbf{Store}
\end{aligned}$$

For brevity, we will overload the symbol $\longrightarrow$ and use it to refer to all of these relations. Which relation is being used will be clear from context.

The evaluation rules for arithmetic and boolean expressions are similar to the ones we've seen before. However, note that since the arithmetic expressions no longer contain assignment, arithmetic and boolean expressions cannot update the store.

Arithmetic expressions

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

$$\frac{\langle e_1, \sigma \rangle \longrightarrow \langle e_1', \sigma \rangle}{\langle e_1 + e_2, \sigma \rangle \longrightarrow \langle e_1' + e_2, \sigma \rangle} \qquad \frac{\langle e_2, \sigma \rangle \longrightarrow \langle e_2', \sigma \rangle}{\langle n + e_2, \sigma \rangle \longrightarrow \langle n + e_2', \sigma \rangle} \qquad \frac{}{\langle n + m, \sigma \rangle \longrightarrow \langle p, \sigma \rangle} \text{ where } p = n + m$$

$$\frac{\langle e_1, \sigma \rangle \longrightarrow \langle e_1', \sigma \rangle}{\langle e_1 \times e_2, \sigma \rangle \longrightarrow \langle e_1' \times e_2, \sigma \rangle} \qquad \frac{\langle e_2, \sigma \rangle \longrightarrow \langle e_2', \sigma \rangle}{\langle n \times e_2, \sigma \rangle \longrightarrow \langle n \times e_2', \sigma \rangle} \qquad \frac{}{\langle n \times m, \sigma \rangle \longrightarrow \langle p, \sigma \rangle} \text{ where } p = n \times m$$

Boolean expressions

$$\frac{\langle a_1, \sigma\rangle \longrightarrow \langle a_1', \sigma\rangle}{\langle a_1 < a_2, \sigma\rangle \longrightarrow \langle a_1' < a_2, \sigma\rangle} \qquad \frac{\langle a_2, \sigma\rangle \longrightarrow \langle a_2', \sigma\rangle}{\langle n < a_2, \sigma\rangle \longrightarrow \langle n < a_2', \sigma\rangle}$$

$$\frac{}{\langle n < m, \sigma\rangle \longrightarrow \langle \textsf{\textbf{true}}, \sigma\rangle} \text{ where } n < m \qquad \frac{}{\langle n < m, \sigma\rangle \longrightarrow \langle \textsf{\textbf{false}}, \sigma\rangle} \text{ where } n \geq m$$

Commands

$$\frac{\langle e, \sigma\rangle \longrightarrow \langle e', \sigma\rangle}{\langle x := e, \sigma\rangle \longrightarrow \langle x := e', \sigma\rangle} \qquad \frac{}{\langle x := n, \sigma\rangle \longrightarrow \langle \textsf{\textbf{skip}}, \sigma[x \mapsto n]\rangle}$$

$$\frac{\langle c_1, \sigma\rangle \longrightarrow \langle c_1', \sigma'\rangle}{\langle c_1; c_2, \sigma\rangle \longrightarrow \langle c_1'; c_2, \sigma'\rangle} \qquad \frac{}{\langle \textsf{\textbf{skip}}; c_2, \sigma\rangle \longrightarrow \langle c_2, \sigma\rangle}$$

For $\textsf{if}$ commands, we gradually reduce the test until we get either **true** or **false**; then, we execute the appropriate branch:

$$\frac{\langle b, \sigma\rangle \longrightarrow \langle b', \sigma\rangle}{\langle \textsf{\textbf{if}}\ b\ \textsf{\textbf{then}}\ c_1\ \textsf{\textbf{else}}\ c_2, \sigma\rangle \longrightarrow \langle \textsf{\textbf{if}}\ b'\ \textsf{\textbf{then}}\ c_1\ \textsf{\textbf{else}}\ c_2, \sigma\rangle}$$

$$\frac{}{\langle \textsf{\textbf{if true then}}\ c_1\ \textsf{\textbf{else}}\ c_2, \sigma\rangle \longrightarrow \langle c_1, \sigma\rangle} \qquad \frac{}{\langle \textsf{\textbf{if false then}}\ c_1\ \textsf{\textbf{else}}\ c_2, \sigma\rangle \longrightarrow \langle c_2, \sigma\rangle}$$

For while loops, the above strategy doesn't work (why?). Instead, we use the following rule, which can be thought of as "unrolling" the loop, one iteration at a time.

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

We can now take a concrete program and see how it executes under the above rules. Consider we start with state σ where $\sigma(\textsf{foo}) = 0$ and we execute the program

$$\textsf{foo} := 3; \textsf{\textbf{while}}\ \textsf{foo} < 4\ \textsf{\textbf{do}}\ \textsf{foo} := \textsf{foo} + 5$$

The execution works as follows:

$$\begin{array}{lll}
 & \langle \textsf{foo} := 3; \textsf{\textbf{while}}\ \textsf{foo} < 4\ \textsf{\textbf{do}}\ \textsf{foo} := \textsf{foo} + 5, \sigma\rangle & \\
\longrightarrow & \langle \textsf{\textbf{skip}}; \textsf{\textbf{while}}\ \textsf{foo} < 4\ \textsf{\textbf{do}}\ \textsf{foo} := \textsf{foo} + 5, \sigma'\rangle & \text{where } \sigma' = \sigma[\textsf{foo} \mapsto 3] \\
\longrightarrow & \langle \textsf{\textbf{while}}\ \textsf{foo} < 4\ \textsf{\textbf{do}}\ \textsf{foo} := \textsf{foo} + 5, \sigma'\rangle & \\
\longrightarrow & \langle \textsf{\textbf{if}}\ \textsf{foo} < 4\ \textsf{\textbf{then}}\ (\textsf{foo} := \textsf{foo} + 5; W)\ \textsf{\textbf{else skip}}, \sigma'\rangle & \\
\longrightarrow & \langle \textsf{\textbf{if}}\ 3 < 4\ \textsf{\textbf{then}}\ (\textsf{foo} := \textsf{foo} + 5; W)\ \textsf{\textbf{else skip}}, \sigma'\rangle & \\
\longrightarrow & \langle \textsf{\textbf{if true then}}\ (\textsf{foo} := \textsf{foo} + 5; W)\ \textsf{\textbf{else skip}}, \sigma'\rangle & \\
\longrightarrow & \langle \textsf{foo} := \textsf{foo} + 5; \textsf{\textbf{while}}\ \textsf{foo} < 4\ \textsf{\textbf{do}}\ \textsf{foo} := \textsf{foo} + 5, \sigma'\rangle & \\
\longrightarrow & \langle \textsf{foo} := 3 + 5; \textsf{\textbf{while}}\ \textsf{foo} < 4\ \textsf{\textbf{do}}\ \textsf{foo} := \textsf{foo} + 5, \sigma'\rangle & \\
\longrightarrow & \langle \textsf{foo} := 8; \textsf{\textbf{while}}\ \textsf{foo} < 4\ \textsf{\textbf{do}}\ \textsf{foo} := \textsf{foo} + 5, \sigma'\rangle & \\
\longrightarrow & \langle \textsf{\textbf{while}}\ \textsf{foo} < 4\ \textsf{\textbf{do}}\ \textsf{foo} := \textsf{foo} + 5, \sigma''\rangle & \text{where } \sigma'' = \sigma'[\textsf{foo} \mapsto 8] \\
\longrightarrow & \langle \textsf{\textbf{if}}\ \textsf{foo} < 4\ \textsf{\textbf{then}}\ (\textsf{foo} := \textsf{foo} + 5; W)\ \textsf{\textbf{else skip}}, \sigma''\rangle & \\
\longrightarrow & \langle \textsf{\textbf{if}}\ 8 < 4\ \textsf{\textbf{then}}\ (\textsf{foo} := \textsf{foo} + 5; W)\ \textsf{\textbf{else skip}}, \sigma''\rangle & \\
\longrightarrow & \langle \textsf{\textbf{if false then}}\ (\textsf{foo} := \textsf{foo} + 5; W)\ \textsf{\textbf{else skip}}, \sigma''\rangle & \\
\longrightarrow & \langle \textsf{\textbf{skip}}, \sigma''\rangle &
\end{array}$$

(where W is an abbreviation for the while loop $\textbf{while}\ \mathsf{foo} < 4\ \textbf{do}\ \mathsf{foo} := \mathsf{foo} + 5$).

10.2 Large-step operational semantics

We define large-step evaluation relations for arithmetic expressions, boolean expressions, and commands. The relation for arithmetic expressions relates an arithmetic expression and store to the integer value that the expression evaluates to. For boolean expressions, the final value is in $\mathbf{Bool} = \{\textbf{true}, \textbf{false}\}$. For commands, the final value is a store.

$$\begin{aligned}
\Downarrow_{\mathbf{Aexp}} &\subseteq \mathbf{Aexp} \times \mathbf{Store} \times \mathbf{Int} \\
\Downarrow_{\mathbf{Bexp}} &\subseteq \mathbf{Bexp} \times \mathbf{Store} \times \mathbf{Bool} \\
\Downarrow_{\mathbf{Com}} &\subseteq \mathbf{Com} \times \mathbf{Store} \times \mathbf{Store}
\end{aligned}$$

Again, we overload the symbol $\Downarrow$ and use it for any of these three relations; which relation is intended will be clear from context. We also use infix notation, for example writing $\langle c, \sigma \rangle \Downarrow \sigma'$ if $(c, \sigma, \sigma') \in \Downarrow_{\mathbf{Com}}$.
Arithmetic expressions.

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

$$\frac{\langle e_1, \sigma \rangle \Downarrow n_1 \qquad \langle e_2, \sigma \rangle \Downarrow n_2}{\langle e_1 + e_2, \sigma \rangle \Downarrow n} \text{ where } n = n_1 + n_2 \qquad \frac{\langle e_1, \sigma \rangle \Downarrow n_1 \qquad \langle e_2, \sigma \rangle \Downarrow n_2}{\langle e_1 \times e_2, \sigma \rangle \Downarrow n} \text{ where } n = n_1 \times n_2$$

Boolean expressions.

$$\frac{}{\langle \textbf{true}, \sigma \rangle \Downarrow \textbf{true}} \qquad\qquad \frac{}{\langle \textbf{false}, \sigma \rangle \Downarrow \textbf{false}}$$

$$\frac{\langle a_1, \sigma \rangle \Downarrow n_1 \qquad \langle a_2, \sigma \rangle \Downarrow n_2}{\langle a_1 < a_2, \sigma \rangle \Downarrow \textbf{true}} \text{ where } n_1 < n_2 \qquad \frac{\langle a_1, \sigma \rangle \Downarrow n_1 \qquad \langle a_2, \sigma \rangle \Downarrow n_2}{\langle a_1 < a_2, \sigma \rangle \Downarrow \textbf{false}} \text{ where } n_1 \geq n_2$$

Commands.

$$\textsc{Skip}\ \frac{}{\langle \textbf{skip}, \sigma \rangle \Downarrow \sigma} \qquad \textsc{Asg}\ \frac{\langle e, \sigma \rangle \Downarrow n}{\langle x := e, \sigma \rangle \Downarrow \sigma[x \mapsto n]} \qquad \textsc{Seq}\ \frac{\langle c_1, \sigma \rangle \Downarrow \sigma' \qquad \langle c_2, \sigma' \rangle \Downarrow \sigma''}{\langle c_1; c_2, \sigma \rangle \Downarrow \sigma''}$$

$$\textsc{If-T}\ \frac{\langle b, \sigma \rangle \Downarrow \textbf{true} \qquad \langle c_1, \sigma \rangle \Downarrow \sigma'}{\langle \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2, \sigma \rangle \Downarrow \sigma'} \qquad \textsc{If-F}\ \frac{\langle b, \sigma \rangle \Downarrow \textbf{false} \qquad \langle c_2, \sigma \rangle \Downarrow \sigma'}{\langle \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2, \sigma \rangle \Downarrow \sigma'}$$

$$\textsc{While-F}\ \frac{\langle b, \sigma \rangle \Downarrow \textbf{false}}{\langle \textbf{while}\ b\ \textbf{do}\ c, \sigma \rangle \Downarrow \sigma} \qquad \textsc{While-T}\ \frac{\langle b, \sigma \rangle \Downarrow \textbf{true} \qquad \langle c, \sigma \rangle \Downarrow \sigma' \qquad \langle \textbf{while}\ b\ \textbf{do}\ c, \sigma' \rangle \Downarrow \sigma''}{\langle \textbf{while}\ b\ \textbf{do}\ c, \sigma \rangle \Downarrow \sigma''}$$

It's interesting to see that the rule for while loops does not rely on using an if command (as we needed in the case of small-step semantics). Why does this rule work?

10.3 Command equivalence

The small-step operational semantics suggest that the loop **while** b **do** c should be equivalent to the command **if** b **then** $(c;$ **while** b **do** $c)$ **else skip**. Can we show that this indeed the case that the language is defined using the above large-step evaluation?

First, we need to to be more precise about what "equivalent commands" mean. Our formal model allows us to define this concept using large-step evaluations as follows. (One can write a similar definition using $\longrightarrow^*$ in small-step semantics.)

Definition (Equivalence of commands)**.** Two commands c and c' are equivalent (written $c \sim c'$) if, for any stores σ and σ', we have

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

We can now state and prove the claim that **while** b **do** c and **if** b **then** $(c;$ **while** b **do** $c)$ **else skip** are equivalent.

Theorem. *For all $b \in \boldsymbol{Bexp}$ and $c \in \boldsymbol{Com}$ we have*

$$\textbf{while } b \textbf{ do } c \sim \textbf{if } b \textbf{ then } (c; \textbf{while } b \textbf{ do } c) \textbf{ else skip}.$$

Proof. Let W be an abbreviation for **while** b **do** c. We want to show that for all stores σ, σ', we have:

$$\langle W, \sigma\rangle \Downarrow \sigma' \text{ if and only if } \textbf{if } b \textbf{ then } (c; W) \textbf{ else skip} \Downarrow \sigma'$$

For this, we must show that both directions ($\Longrightarrow$ and $\Longleftarrow$) hold. We'll show only direction $\Longrightarrow$; the other is similar.

Assume that σ and σ' are stores such that $\langle W, \sigma\rangle \Downarrow \sigma'$. It means that there is some derivation that proves for this fact. Inspecting the evaluation rules, we see that there are two possible rules whose conclusions match this fact: While-F and While-T. We analyze each of them in turn.

- While-F. The derivation must look like the following.

$$\textsc{While-F}\ \frac{\dfrac{\vdots^1}{\langle b, \sigma\rangle \Downarrow \textbf{false}}}{\langle W, \sigma\rangle \Downarrow \sigma}$$

  Here, we use $\vdots^1$ to refer to the derivation of $\langle b, \sigma\rangle \Downarrow \textbf{false}$. Note that in this case, $\sigma' = \sigma$.

  We can use $\vdots^1$ to derive a proof tree showing that the evaluation of **if** b **then** $(c; W)$ **else skip** yields the same final state σ:

$$\textsc{If-F}\ \frac{\dfrac{\vdots^1}{\langle b, \sigma\rangle \Downarrow \textbf{false}} \qquad \textsc{Skip}\ \dfrac{}{\langle \textbf{skip}, \sigma\rangle \Downarrow \sigma}}{\langle \textbf{if } b \textbf{ then } (c; W) \textbf{ else skip}, \sigma\rangle \Downarrow \sigma}$$

- While-T. In this case, the derivation has the following form.

$$\textsc{While-T}\ \frac{\dfrac{\vdots^2}{\langle b, \sigma\rangle \Downarrow \textbf{true}} \qquad \dfrac{\vdots^3}{\langle c, \sigma\rangle \Downarrow \sigma''} \qquad \dfrac{\vdots^4}{\langle W, \sigma''\rangle \Downarrow \sigma'}}{\langle W, \sigma\rangle \Downarrow \sigma'}$$

  We can use subderivations $\vdots^2$, $\vdots^3$, and $\vdots^4$ to show that the evaluation of **if** b **then** $(c; W)$ **else skip** yields the same final state σ.

$$\textsc{If-T}\ \frac{\dfrac{\vdots^2}{\langle b, \sigma\rangle \Downarrow \textbf{true}} \qquad \textsc{Seq}\ \dfrac{\dfrac{\vdots^3}{\langle c, \sigma\rangle \Downarrow \sigma''} \qquad \dfrac{\vdots^4}{\langle W, \sigma''\rangle \Downarrow \sigma'}}{\langle c; W, \sigma\rangle \Downarrow \sigma'}}{\langle \textbf{if } b \textbf{ then } (c; W) \textbf{ else skip}, \sigma\rangle \Downarrow \sigma'}$$

Hence, we showed that in each of the two possible cases, the command **if** b **then** $(c; W)$ **else skip** evaluates to the same final state as the command W. □

10.4 Some properties of IMP

10.4.1 Equivalence of semantics

The small-step and large-step semantics are equivalent. We state this formally in the following theorem.

Theorem (Equivalence of IMP semantics)**.** *For all commands $c \in$ **Com** and stores $\sigma, \sigma' \in$ **Store** we have*

$$\langle c, \sigma \rangle \longrightarrow^* \langle \mathbf{skip}, \sigma' \rangle \iff \langle c, \sigma \rangle \Downarrow \sigma'.$$

10.4.2 Non-termination

For a command c and initial state σ, the execution of the command may *terminate* with some final store σ', or it may *diverge* and never yield a final state. For example, the command **while true do** $\mathsf{foo} := \mathsf{foo} + 1$ always diverges; the command **while** $0 < \mathsf{i}$ **do** $\mathsf{i} := \mathsf{i} + 1$ will diverge if and only if the value of variable i in the initial state is positive.

If $\langle c, \sigma \rangle$ is a configuration that diverges, then there is no state σ' such that $\langle c, \sigma \rangle \Downarrow \sigma'$ or $\langle c, \sigma \rangle \longrightarrow^* \langle \mathbf{skip}, \sigma' \rangle$. However, in small-step semantics, a diverging computation has an infinite sequence of configurations: $\langle c, \sigma \rangle \longrightarrow \langle c_1, \sigma_1 \rangle \longrightarrow \langle c_2, \sigma_2 \rangle \longrightarrow \ldots$. Small-step semantics can allow us to state, and prove, properties about programs that may diverge. Later in the course, we will specify and prove properties that are of interest in potentially diverging computations.

10.4.3 Determinism of commands

The semantics of IMP (both small-step and large-step) are *deterministic*. That is, each IMP command c and each initial store σ evaluates to at most one final store. We state this formally for the large-step semantics below.

Theorem. *For all commands $c \in$ **Com** and stores $\sigma, \sigma_1, \sigma_2 \in$ **Store**, if $\langle c, \sigma \rangle \Downarrow \sigma_1$ and $\langle c, \sigma \rangle \Downarrow \sigma_2$ then $\sigma_1 = \sigma_2$.*

We need an inductive proof to prove this theorem. However, induction on the structure of command c does not work. (Why? Which of the cases does it fail for?) Instead, we need to perform induction on the derivation of $\langle c, \sigma \rangle \Downarrow \sigma_1$.

Before we commence the proof of the theorem, we will need two lemmas, related to the determinism of the arithmetic and boolean semantics, $\Downarrow_{\mathbf{Aexp}}$ and $\Downarrow_{\mathbf{Bexp}}$.

Lemma 3. *For all arithmetic expressions $a \in$ **Aexp**, stores $\sigma \in$ **Store**, and integers $n_1, n_2 \in$ **Int**, if $\langle a, \sigma \rangle \Downarrow n_1$ and $\langle a, \sigma \rangle \Downarrow n_2$ then $n_1 = n_2$.*

Lemma 4. *For all boolean expressions $b \in$ **Aexp**, stores $\sigma \in$ **Store**, and integers $b_1, b_2 \in$ **Bool**, if $\langle b, \sigma \rangle \Downarrow b_1$ and $\langle a, \sigma \rangle \Downarrow b_2$ then $b_1 = b_2$.*

These lemmas are straightforward to prove, and can be proved using strucutral induction on arithmetic and boolean expressions respectively.

Proof. We proceed by induction on the derivation of $\langle c, \sigma \rangle \Downarrow \sigma_1$. The inductive hypothesis P is

$$P(\langle c, \sigma \rangle \Downarrow \sigma_1) = \forall \sigma_2 \in \mathbf{Store}, \text{if } \langle c, \sigma \rangle \Downarrow \sigma_2 \text{ then } \sigma_1 = \sigma_2.$$

Suppose we have a derivation for $\langle c, \sigma \rangle \Downarrow \sigma_1$, for some c, σ, and σ_1. Assume that the inductive hypothesis holds for any subderivation $\langle c', \sigma' \rangle \Downarrow \sigma''$ used in the derivation of $\langle c, \sigma \rangle \Downarrow \sigma_1$.

Assume that for some σ_2 we have $\langle c, \sigma \rangle \Downarrow \sigma_2$. We need to show that $\sigma_1 = \sigma_2$.

We consider the possible cases for the last rule used in derivation of

$$\langle c, \sigma\rangle \Downarrow \sigma_1$$

.

- $\textsc{Skip}$. In this case, the derivation looks like

  $$\textsc{Skip}\,\frac{\vdots}{\langle \textbf{skip}, \sigma\rangle \Downarrow \sigma}\ ,$$

  and we have $c \equiv \textbf{skip}$ and $\sigma_1 = \sigma$. Since by assumption we have $\langle c, \sigma\rangle \Downarrow \sigma_2$, there must be a derivation of $\langle c, \sigma\rangle \Downarrow \sigma_2$. Moreover, the last rule used in this derivation must be $\textsc{Skip}$, as it is the only rule that has the command **skip** in its conclusion. So we have $\sigma_2 = \sigma$ and the result holds.

- $\textsc{Asg}$

  In this case, the derivation looks like

  $$\textsc{Asg}\,\frac{\dfrac{\vdots}{\langle a, \sigma\rangle \Downarrow n}}{\langle x := a, \sigma\rangle \Downarrow \sigma_1}\ ,$$

  and we have $c \equiv x := a$ and $\sigma_1 = \sigma[x \mapsto n]$. The last rule used in the derivation of $\langle c, \sigma\rangle \Downarrow \sigma_2$ must also be $\textsc{Asg}$, and so we have $\sigma_2 = \sigma[x \mapsto m]$, where $\langle a, \sigma\rangle \Downarrow m$. By the determinism of arithmetic expressions, $m = n$ and so $\sigma_2 = \sigma_1$ and the result holds.

- $\textsc{Seq}$

  In this case, the derivation looks like

  $$\textsc{Seq}\,\frac{\dfrac{\vdots}{\langle c_1, \sigma\rangle \Downarrow \sigma'} \qquad \dfrac{\vdots}{\langle c_2, \sigma'\rangle \Downarrow \sigma_1}}{\langle c_1; c_2, \sigma\rangle \Downarrow \sigma_1}\ ,$$

  and we have $c \equiv c_1; c_2$. The last rule used in the derivation of $\langle c, \sigma\rangle \Downarrow \sigma_2$ must also be $\textsc{Seq}$, and so we have

  $$\textsc{Seq}\,\frac{\dfrac{\vdots}{\langle c_1, \sigma\rangle \Downarrow \sigma''} \qquad \dfrac{\vdots}{\langle c_2, \sigma''\rangle \Downarrow \sigma_2}}{\langle c_1; c_2, \sigma\rangle \Downarrow \sigma_2}\ .$$

  By the inductive hypothesis applied to the derivation $\dfrac{\vdots}{\langle c_1, \sigma\rangle \Downarrow \sigma'}$, we have $\sigma' = \sigma''$. By another application of the inductive hypothesis, to the derivation $\dfrac{\vdots}{\langle c_2, \sigma'\rangle \Downarrow \sigma_1}$, we have $\sigma_1 = \sigma_2$ and the result holds.

- $\textsc{If-T}$

  In this case, the derivation looks like

  $$\textsc{If-T}\,\frac{\dfrac{\vdots}{\langle b, \sigma\rangle \Downarrow \textbf{true}} \qquad \dfrac{\vdots}{\langle c_1, \sigma\rangle \Downarrow \sigma_1}}{\langle \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2, \sigma\rangle \Downarrow \sigma_1}\ ,$$

and we have $c \equiv \textsf{\textbf{if}}\ b\ \textsf{\textbf{then}}\ c_1\ \textsf{\textbf{else}}\ c_2$. The last rule used in the derivation of $\langle c, \sigma\rangle \Downarrow \sigma_2$ must be either If-T or If-F (since these are the only rules that can be used to derive a conclusion of the form $\langle \textsf{\textbf{if}}\ b\ \textsf{\textbf{then}}\ c_1\ \textsf{\textbf{else}}\ c_2, \sigma\rangle \Downarrow \sigma_2$). But by the determinism of boolean expressions, we must have $\langle b, \sigma\rangle \Downarrow \textsf{\textbf{true}}$, and so the derivation of $\langle c, \sigma\rangle \Downarrow \sigma_2$ must have the following form.

$$\textsc{If-T}\ \dfrac{\dfrac{\vdots}{\langle b, \sigma\rangle \Downarrow \textsf{\textbf{true}}} \qquad \dfrac{\vdots}{\langle c_1, \sigma\rangle \Downarrow \sigma_2}}{\langle \textsf{\textbf{if}}\ b\ \textsf{\textbf{then}}\ c_1\ \textsf{\textbf{else}}\ c_2, \sigma\rangle \Downarrow \sigma_2}$$

The result holds by the inductive hypothesis applied to the derivation $\dfrac{\vdots}{\langle c_1, \sigma\rangle \Downarrow \sigma_1}$.

- If-F

  Similar to the case for If-T.

- While-F

  Straightforward, similar to the case for Skip.

- While-T

  Here we have

$$\textsc{While-T}\ \dfrac{\dfrac{\vdots}{\langle b, \sigma\rangle \Downarrow \textsf{\textbf{true}}} \qquad \dfrac{\vdots}{\langle c_1, \sigma\rangle \Downarrow \sigma'} \qquad \dfrac{\vdots}{\langle c, \sigma'\rangle \Downarrow \sigma_1}}{\langle \textsf{\textbf{while}}\ b\ \textsf{\textbf{do}}\ c_1, \sigma\rangle \Downarrow \sigma_1},$$

  and we have $c \equiv \textsf{\textbf{while}}\ b\ \textsf{\textbf{do}}\ c_1$. The last rule used in the derivation of $\langle c, \sigma\rangle \Downarrow \sigma_2$ must also be While-T (by the determinism of boolean expressions), and so we have

$$\textsc{While-T}\ \dfrac{\dfrac{\vdots}{\langle b, \sigma\rangle \Downarrow \textsf{\textbf{true}}} \qquad \dfrac{\vdots}{\langle c_1, \sigma\rangle \Downarrow \sigma''} \qquad \dfrac{\vdots}{\langle c, \sigma''\rangle \Downarrow \sigma_2}}{\langle \textsf{\textbf{while}}\ b\ \textsf{\textbf{do}}\ c_1, \sigma\rangle \Downarrow \sigma_2}.$$

  By the inductive hypothesis applied to the derivation $\dfrac{\vdots}{\langle c_1, \sigma\rangle \Downarrow \sigma'}$, we have $\sigma' = \sigma''$. By another application of the inductive hypothesis, to the derivation $\dfrac{\vdots}{\langle c, \sigma'\rangle \Downarrow \sigma_1}$, we have $\sigma_1 = \sigma_2$ and the result holds.

  Note: Even though the command $c \equiv \textsf{\textbf{while}}\ b\ \textsf{\textbf{do}}\ c_1$ appears in the derivation of $\langle \textsf{\textbf{while}}\ b\ \textsf{\textbf{do}}\ c_1, \sigma\rangle \Downarrow \sigma_1$, we do not run in to problems, as the induction is over the *derivation*, not over the structure of the command.

So we have shown that $P(\langle c, \sigma\rangle \Downarrow \sigma_1)$ for any c, σ, and σ_1 such that $\langle c, \sigma\rangle \Downarrow \sigma_1$. This is equivalent to

$$\forall c \in \textbf{Com}.\ \forall \sigma, \sigma_1, \sigma_2 \in \textbf{Store}, \text{if } \langle c, \sigma\rangle \Downarrow \sigma_1 \text{ and } \langle c, \sigma\rangle \Downarrow \sigma_2 \text{ then } \sigma_1 = \sigma_2$$

which proves the theorem. □