Semantics of Programming Languages

Theory of Programming Languages (2020-21)

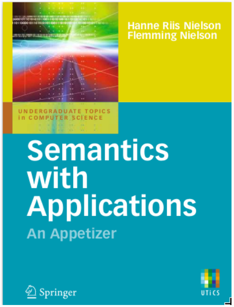
Master's Degree in Formal Methods in Computer Science

Manuel Montenegro (montenegro@fdi.ucm.es)

Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Despacho 219
Universidad Complutense de Madrid

Bibliography

H.R. Nielson, F. Nielson
Semantics with applications: an appetizer
Springer (2007)


Hanne Riis Nielson
Flemming Nielson
UNDERGRADUATE TOPICS in COMPUTER SCIENCE
Semantics with Applications
An Appetizer
Springer
UTiCS

Outline

Why semantics?

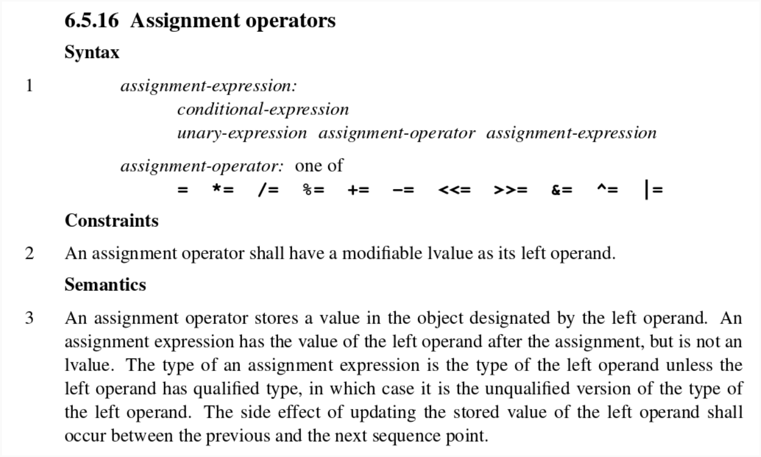
How we specify programming languages?

A programming language is defined by:

- Syntax
  - Context-free grammars, BNF, etc.
  - Determines which programs are well-formed.
- Constraints
  - Pose some restrictions that a CF grammar cannot specify.
  - For example: a variable must be defined before its use.
  - Restrict the set of well-formed programs.
- Semantics $\longleftarrow$ we are here!
  - Specifies the meaning of a program, i.e. how a program is executed, and what does it return.
  - In most cases, it is informal specification in plain English.

Example

C99 Standard - WG14 Draft ver. 1256:



6.5.16 Assignment operators

Syntax

1 *assignment-expression:*
  conditional-expression
  unary-expression assignment-operator assignment-expression

assignment-operator: one of

```
= *= /= %= += -= <<= >>= &= ^= |=
```

Constraints

2 An assignment operator shall have a modifiable lvalue as its left operand.

Semantics

3 An assignment operator stores a value in the object designated by the left operand. An assignment expression has the value of the left operand after the assignment, but is not an lvalue. The type of an assignment expression is the type of the left operand unless the left operand has qualified type, in which case it is the unqualified version of the type of the left operand. The side effect of updating the stored value of the left operand shall occur between the previous and the next sequence point.

Source: `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf`

Disadvantages of informal semantics

- Ambiguity
  - Some corner cases are left unspecified, which leads to varying implementations.
  - Example in C99: `i = i++;`
- Verbosity
  - Particularities must be explicitly mentioned.
  - C18 standard is 520 pages long.
- Inability to reason about programs
  - Do compiler optimizations alter program semantics?

Formal semantics

Formal semantics is concerned with building a mathematical model that reflects the behaviour of programs.

It is useful for:

- reasoning about how a program behaves.
- program analysis and verification.
- avoiding ambiguities and subtle complexities.
- implementing compilers, runtime environments, etc.
- disregarding superfluous concepts: registers, addresses, etc.

Semantic description methods

- Operational semantics

  It defines how a program is executed.

  - Natural semantics (*big-step*)
  - Structural operational semantics (*small-step*)
- Denotational semantics

  It models the result (or effect) of executing a program.
- Axiomatic semantics

  It allows one to prove program properties (e.g. partial correctness).

These methods are complementary.

Simple arithmetic expressions

Simple arithmetic expressions

AExp is the set of expressions generated by the following grammar:

$$\begin{array}{rcll} e & := & n & \{\text{numeric constant } n \in \mathbb{Z}\ \} \\ & \mid & x & \{\text{variable } x \in \mathbf{Var}\ \} \\ & \mid & e_1 + e_2 & \{\text{addition}\} \\ & \mid & e_1 - e_2 & \{\text{substraction}\} \\ & \mid & e_1 * e_2 & \{\text{multiplication}\} \end{array}$$

where **Var** is a set of variables: x, y, etc.

Simple arithmetic expressions

Let us analyse the syntax of simple arithmetic expressions:

$$\begin{array}{rcll} e & := & n & \text{Basic elements} \\ & | & x & \\ & | & e_1 + e_2 & \\ & | & e_1 - e_2 & \text{Composite elements} \\ & | & e_1 * e_2 & \end{array}$$

Composite elements have a unique decomposition into their immediate constituents.

Compositional definitions

- Assume we want to define a function $FV : \mathbf{AExp} \to \mathcal{P}(\mathbf{Var})$ that yields the set of variables in an expression:

$$\begin{array}{rcl} FV(n) & = & \emptyset \\ FV(x) & = & \{x\} \\ FV(e_1 + e_2) & = & FV(e_1) \cup FV(e_2) \\ FV(e_1 - e_2) & = & FV(e_1) \cup FV(e_2) \\ FV(e_1 * e_2) & = & FV(e_1) \cup FV(e_2) \end{array}$$

Example

$$FV(x * (y - 5)) = \{x, y\}$$

Compositional definitions

This is an example of a compositional definition:

$$\begin{array}{rcl} FV(n) & = & \emptyset \\ FV(x) & = & \{x\} \\ FV(e_1 + e_2) & = & FV(e_1) \cup FV(e_2) \\ FV(e_1 - e_2) & = & FV(e_1) \cup FV(e_2) \\ FV(e_1 * e_2) & = & FV(e_1) \cup FV(e_2) \end{array}$$

- There is a clause for each syntactic category.
- The *FV* of a composite expression is defined in terms of the *FV* of its immediate constituents.
    - For example: $FV(e_1 + e_2)$ is defined in terms of $FV(e_1)$ and $FV(e_2)$.

Compositional definitions

Another example: *size* returns the number of nodes in the AST of an expression.

$$size : \mathbf{AExp} \to \mathbb{N}$$

$$\begin{aligned}
size(n) &= 1 \\
size(x) &= 1 \\
size(e_1 + e_2) &= 1 + size(e_1) + size(e_2) \\
size(e_1 - e_2) &= 1 + size(e_1) + size(e_2) \\
size(e_1 * e_2) &= 1 + size(e_1) + size(e_2)
\end{aligned}$$

Structural induction

Recall induction on natural numbers:

Induction principle for natural numbers

To prove that a property P holds for every number $n \in \mathbb{N}$:

- **Base case**: We prove $P(0)$
- **Inductive step**: Assuming that $P(n)$ holds, we prove that $P(n+1)$ holds.

Structural induction

Structural induction is a technique for proving properties on compositional definitions.

Structural induction on expressions

To prove that a property P holds for every expression $e \in \mathbf{AExp}$:

- **Base cases:**
    - Prove $P(n)$ for each $n \in \mathbf{Num}$.
    - Prove $P(x)$ for each $x \in \mathbf{Var}$.
- **Inductive step:** for each $e_1, e_2 \in \mathbf{AExp}$,
    - Assuming that $P(e_1)$ and $P(e_2)$ hold, prove $P(e_1 + e_2)$.
    - Assuming that $P(e_1)$ and $P(e_2)$ hold, prove $P(e_1 - e_2)$.
    - Assuming that $P(e_1)$ and $P(e_2)$ hold, prove $P(e_1 * e_2)$.

Example

Prove that $|FV(e)| \leq size(e)$ for every expression e.

We prove the property by structural induction on e:

- **Base cases:**
  - Assume $e \equiv n$ for some $n \in \mathbf{Num}$.
    We get that $|FV(n)| = 0$ and $size(n) = 1$.
    Since $0 \leq 1$, the property holds.
  - Assume $e \equiv x$ for some $x \in \mathbf{Var}$.
    We get that $|FV(x)| = 1$ and $size(n) = 1$.
    Since $1 \leq 1$, the property holds.

Example

Prove that $|FV(e)| \leq size(e)$ for every expression e.

We prove the property by structural induction on e:

- **Inductive step:**
  - Assume $e \equiv e_1 + e_2$ for some $e_1, e_2 \in$ **AExp**.
    Induction hypothesis: We assume that $|FV(e_1)| \leq size(e_1)$ and that $|FV(e_2)| \leq size(e_2)$.
    Let us prove that $|FV(e_1 + e_2)| \leq size(e_1 + e_2)$:

$$\begin{array}{rcll}
|FV(e_1 + e_2)| & = & |FV(e_1) \cup FV(e_2)| & \{\text{by definition of } FV\} \\
& \leq & |FV(e_1)| + |FV(e_2)| & \{\text{cardinality properties}\} \\
& \leq & size(e_1) + |FV(e_2)| & \{\text{induction hypothesis on } e_1\} \\
& \leq & size(e_1) + size(e_2) & \{\text{induction hypothesis on } e_2\} \\
& \leq & 1 + size(e_1) + size(e_2) & \\
& = & size(e_1 + e_2) & \{\text{definition of } size\}
\end{array}$$

Example

Prove that $|FV(e)| \leq size(e)$ for every expression e.

We prove the property by structural induction on e:

- **Inductive step:**
  - Assume $e \equiv e_1 - e_2$ for some $e_1, e_2 \in$ **AExp**.
    Induction hypothesis: We assume that $|FV(e_1)| \leq size(e_1)$ and that $|FV(e_2)| \leq size(e_2)$.
    We proceed as in the previous case.
  - Assume $e \equiv e_1 * e_2$ for some $e_1, e_2 \in$ **AExp**.
    Induction hypothesis: We assume that $|FV(e_1)| \leq size(e_1)$ and that $|FV(e_2)| \leq size(e_2)$.
    We proceed as in the previous case.

Let us find a semantic definition for expressions

Define a function $\mathcal{A}[\![\_]\!] : \textsf{AExp} \to \mathbb{N}$ that yields the value denoted by an expression.

For example, it must hold that:

$$\mathcal{A}[\![(2 * 4) - 5]\!] = 3$$

Program state

- The semantics of an expression depends on the values assigned to the variables occurring in it.
- A state maps each variable to its current value.
- Formally, a state is a function $\textsf{Var} \to \mathbb{Z}$.
- We use **State** to denote the set of all states.
- We use σ, σ_1, etc. to denote elements from this set.

Program state

Some notational conventions:

- $[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$ denotes the state σ such that

$$\sigma(x_1) = v_1, \ldots, \sigma(x_n) = v_n$$

  and $\sigma(y) = 0$ for every $y \in \mathbf{Var} \backslash \{x_1, \ldots, x_n\}$.
- $[\,]$ denotes the state σ such that $\sigma(x) = 0$ for every $x \in \mathbf{Var}$.
- Given $\sigma \in \mathbf{State}$, the notation $\sigma[x \mapsto v]$ denotes the state σ' such that for all $y \in \mathbf{Var}$:

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

Semantics of arithmetic expressions

- The semantics of an expression now has the following form:

$$\mathcal{A}[\![\_]\!] : \textsf{AExp} \to \textsf{State} \to \mathbb{Z}$$

- It is defined as follows:

$$\begin{array}{rcl} \mathcal{A}[\![n]\!]\,\sigma & = & n \\ \mathcal{A}[\![x]\!]\,\sigma & = & \sigma(x) \\ \mathcal{A}[\![e_1 + e_2]\!]\,\sigma & = & \mathcal{A}[\![e_1]\!]\,\sigma + \mathcal{A}[\![e_2]\!]\,\sigma \\ \mathcal{A}[\![e_1 - e_2]\!]\,\sigma & = & \mathcal{A}[\![e_1]\!]\,\sigma - \mathcal{A}[\![e_2]\!]\,\sigma \\ \mathcal{A}[\![e_1 * e_2]\!]\,\sigma & = & \mathcal{A}[\![e_1]\!]\,\sigma \cdot \mathcal{A}[\![e_2]\!]\,\sigma \end{array}$$

Is $\mathcal{A}[\![\_]\!]$ a compositional definition?

Semantics of arithmetic expressions

- In the equations shown previously, we have to make distinction between syntactic elements of the language and actual integer operators.

$$\mathcal{A}[\![e_1 \underset{\substack{\uparrow\\ \boxed{\text{Syntactic element}}}}{+} e_2]\!]\,\sigma = \mathcal{A}[\![e_1]\!]\,\sigma \underset{\substack{\uparrow\\ \boxed{\text{Addition on integers}}}}{+} \mathcal{A}[\![e_2]\!]\,\sigma$$

- The $+$ on the left-hand side is just a textual representation, in the **While** language, of the addition operator.
- The $+$ at the right-hand side is the addition operator on $\mathbb{Z}$, in the mathematical sense.
- Sometimes we use $+_{\mathbb{Z}}$ in the latter case to highlight this distinction.

Semantics of arithmetic expressions

Example

Assume that $\sigma = [x \mapsto 3, y \mapsto -5, z \mapsto 4]$:

$$
\begin{aligned}
\mathcal{A}[\![(3 * x + y) - z]\!]\,\sigma &= \mathcal{A}[\![3 * x + y]\!]\,\sigma - \mathcal{A}[\![z]\!]\,\sigma \\
&= \mathcal{A}[\![3 * x]\!]\,\sigma + \mathcal{A}[\![y]\!]\,\sigma - \mathcal{A}[\![z]\!]\,\sigma \\
&= \mathcal{A}[\![3]\!]\,\sigma \cdot \mathcal{A}[\![x]\!]\,\sigma + \mathcal{A}[\![y]\!]\,\sigma - \mathcal{A}[\![z]\!]\,\sigma \\
&= 3 \cdot \sigma(x) + \sigma(y) - \sigma(z) \\
&= 3 \cdot 3 + (-5) - 4 \\
&= 0
\end{aligned}
$$

Alternative notation: λ-expressions

- In some programming languages we can define functions without giving them a name (anonymous functions).
- For example, if we want to specify the function f such that $f(x) = x + 1$ for all $x \in \mathbb{Z}$:
  - Javascript (ES6): `x => x + 1`
  - Java: `x -> x + 1`
  - Haskell: `\x -> x + 1`
  - Erlang: `fun(X) -> X + 1 end`
  - Python: `lambda x: x + 1`
- We shall use similar notation to define anonymous functions: $\lambda x \,.\, x + 1$

Alternative notation: λ-expressions

- Our semantic definitions for arithmetic expressions can be rewritten as follows:

$$\begin{aligned}
\mathcal{A}[\![n]\!] &= \lambda\sigma\,.\,n \\
\mathcal{A}[\![x]\!] &= \lambda\sigma\,.\,\sigma(x) \\
\mathcal{A}[\![e_1 + e_2]\!] &= \lambda\sigma\,.\,\mathcal{A}[\![e_1]\!]\,\sigma + \mathcal{A}[\![e_2]\!]\,\sigma \\
\mathcal{A}[\![e_1 - e_2]\!] &= \lambda\sigma\,.\,\mathcal{A}[\![e_1]\!]\,\sigma - \mathcal{A}[\![e_2]\!]\,\sigma \\
\mathcal{A}[\![e_1 * e_2]\!] &= \lambda\sigma\,.\,\mathcal{A}[\![e_1]\!]\,\sigma \cdot \mathcal{A}[\![e_2]\!]\,\sigma
\end{aligned}$$

Semantics of arithmetic expressions

- The $\mathcal{A}[\![\_]\!]$ function defined previously is a denotational definition of the semantics of an arithmetic expression.
- A denotational assigns a **mathematical entity** to each expression.
    - For example, the expression $3 + x * y$ is denoted by the mathematical function $\lambda\sigma \,.\, 3 + \sigma(x) \cdot \sigma(y)$.
- In this denotational semantics, the function $\mathcal{A}[\![\_]\!]$ maps each arithmetic expression to a function **State** $\rightarrow \mathbb{Z}$.
- But there are some other ways to define the semantics of arithmetic expressions...

How would you define equivalence on arithmetic expressions?

Given two arithmetic expressions $e_1, e_2 \in \mathsf{AExp}$, we say that they are equivalent if and only if:

Arithmetic expressions: operational semantics

- Denotational semantics put emphasis on *what* is computed, but not *how* is it computed.
- Operational semantics gives meaning to program constructs by specifying the computational steps involved in the evaluation of an expression.
- There are two styles of operational semantics:
    - Big-step semantics: describes how the final result is obtained.
    - Small-step semantics: describes the steps of the computation.

Arithmetic expressions: big-step semantics

- Let us define the big-step semantics of an expression as a relation BS on states, expressions, and integer numbers.

$$BS \subseteq \textbf{AExp} \times \textbf{State} \times \mathbb{Z}$$

- Informally, the fact that a tuple (e, σ, n) belongs to BS means that the expression e, when evaluated under an state σ, yields n as a result.

Example

Given $\sigma = [x \mapsto 3, y \mapsto -5, z \mapsto 4]$:

$$((3 * x + y) - z, \sigma, 0) \in BS$$

Arithmetic expressions: big-step semantics

- Let us use a less cumbersome notation.
- Instead of writing $(e, \sigma, n) \in BS$, we write $\langle e, \sigma \rangle \Downarrow n$.
- So, $\langle e, \sigma \rangle \Downarrow n$ means: *if we want to evaluate e under an state σ, we obtain n as a result.*
- Now we define this relation by distinguishing cases according to the structure of e.

Arithmetic expressions: big-step semantics

Our big-step relation *BS* should satisfy the following conditions:

1. For every $\sigma \in \textbf{State}$, $n \in \mathbb{Z}$: $\langle n, \sigma \rangle \Downarrow n$.
2. For every $\sigma \in \textbf{State}$, $x \in \textbf{Var}$: $\langle x, \sigma \rangle \Downarrow \sigma(x)$
3. For every $\sigma \in \textbf{State}$, $e_1, e_2 \in \textbf{AExp}$, $n_1, n_2 \in \mathbb{Z}$:

   If $\langle e_1, \sigma \rangle \Downarrow n_1$ and $\langle e_2, \sigma \rangle \Downarrow n_2$ then $\langle e_1 + e_2, \sigma \rangle \Downarrow n_1 + n_2$
4. For every $\sigma \in \textbf{State}$, $e_1, e_2 \in \textbf{AExp}$, $n_1, n_2 \in \mathbb{Z}$:

   If $\langle e_1, \sigma \rangle \Downarrow n_1$ and $\langle e_2, \sigma \rangle \Downarrow n_2$ then $\langle e_1 - e_2, \sigma \rangle \Downarrow n_1 - n_2$
5. For every $\sigma \in \textbf{State}$, $e_1, e_2 \in \textbf{AExp}$, $n_1, n_2 \in \mathbb{Z}$:

   If $\langle e_1, \sigma \rangle \Downarrow n_1$ and $\langle e_2, \sigma \rangle \Downarrow n_2$ then $\langle e_1 * e_2, \sigma \rangle \Downarrow n_1 * n_2$

Arithmetic expressions: big-step semantics

Example

Given $\sigma = [x \mapsto 3, y \mapsto -5, z \mapsto 4]$, prove that

$$\langle (3 * x + y) - z, \sigma \rangle \Downarrow 0$$

- By condition (1), we know that $\langle 3, \sigma \rangle \Downarrow 3$.
- By condition (2), we know that $\langle x, \sigma \rangle \Downarrow 3$.
- Therefore, by condition (5), $\langle 3 * x, \sigma \rangle \Downarrow 9$.
- On the other hand, by (2), $\langle y, \sigma \rangle \Downarrow -5$.
- Therefore, by condition (3), $\langle 3 * x + y \rangle \Downarrow \sigma, 4$.
- Moreover, by (2), $\langle z, \sigma \rangle \Downarrow 4$.
- Therefore, by (4), $\langle (3 * x + y) - z, \sigma \rangle \Downarrow 0$.

Is there more than one relation that satisfies these five conditions?

Arithmetic expressions: big-step semantics

- We define the big-step semantics relation BS as the smallest relation satisfying these five conditions.
- This is equivalent to say that **every tuple** $(e, \sigma, n) \in BS$ **must have been obtained by applying these conditions**.
  - If we had a tuple $(e, \sigma, n) \in BS$ that cannot be obtained from these conditions, we could remove it from BS so as to get another relation BS' that satisfies these conditions. Since $BS' \subset BS$, it follows that BS is not the smallest relation satisfying these conditions, which leads to a contradiction.

Introducing judgements and proof rules

- Each of the five conditions shown above can be expressed as a set of proof rules.
- A proof rule has the following form:

$$\frac{\text{premise}_1 \quad \cdots \quad \text{premise}_n}{\text{conclusion}}$$

  where premises and conclusion are judgements of the form $\langle e, \sigma \rangle \Downarrow n$ for some e, σ, and n.
- If a rule does not have premises, it is an axiom.

Big-step semantics rewritten

- Condition (1) can be expressed with the following axiom:

$$\frac{}{\langle n, \sigma \rangle \Downarrow n} \text{ [Const}_{BS}\text{]}$$

- Condition (2) can be expressed with the following axiom:

$$\frac{}{\langle x, \sigma \rangle \Downarrow \sigma(x)} \text{ [Var}_{BS}\text{]}$$

- Condition (3) can be expressed as follows:

$$\frac{\langle e_1, \sigma \rangle \Downarrow n_1 \quad \langle e_2, \sigma \rangle \Downarrow n_2}{\langle e_1 + e_2, \sigma \rangle \Downarrow n_1 + n_2} \text{ [Add}_{BS}\text{]}$$

Big-step semantics rewritten

- Condition (4) can be expressed as follows:

$$\frac{\langle e_1, \sigma \rangle \Downarrow n_1 \quad \langle e_2, \sigma \rangle \Downarrow n_2}{\langle e_1 - e_2, \sigma \rangle \Downarrow n_1 - n_2} \; [\mathrm{Sub}_{BS}]$$

- Condition (5) can be expressed as follows:

$$\frac{\langle e_1, \sigma \rangle \Downarrow n_1 \quad \langle e_2, \sigma \rangle \Downarrow n_2}{\langle e_1 * e_2, \sigma \rangle \Downarrow n_1 \cdot n_2} \; [\mathrm{Mul}_{BS}]$$

Proof rules & derivations

- With these rules we can define the semantics of arithmetic expressions as follows:

Definition

We say that the expression e, under an state σ evaluates to a number n if and only if we can build a derivation of the judgement $\langle e, \sigma \rangle \Downarrow n$ by using the proof rules shown before.

Proof rules & derivations

Example

Assume that $\sigma = [x \mapsto 3, y \mapsto -5, z \mapsto 4]$.

We have the following derivation for $\langle (3 * x + y) - z, \sigma\rangle \Downarrow 0$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\langle 3, \sigma\rangle \Downarrow 3}\,[\text{Const}] \quad \dfrac{}{\langle x, \sigma\rangle \Downarrow 3}\,[\text{Var}]}{\langle 3 * x, \sigma\rangle \Downarrow 9}\,[\text{Mul}] \quad \dfrac{}{\langle y, \sigma\rangle \Downarrow -5}\,[\text{Var}]}{\langle 3 * x + y, \sigma\rangle \Downarrow 4}\,[\text{Add}] \quad \dfrac{}{\langle z, \sigma\rangle \Downarrow 4}\,[\text{Var}]}{\langle (3 * x + y) - z, \sigma\rangle \Downarrow 0}\,[\text{Sub}]$$

Proof rules & derivations

leaves

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\langle 3,\sigma\rangle \Downarrow 3}\,\text{[Const]} \quad \dfrac{}{\langle x,\sigma\rangle \Downarrow 3}\,\text{[Var]}}{\langle 3*x,\sigma\rangle \Downarrow 9}\,\text{[Mul]} \quad \dfrac{}{\langle y,\sigma\rangle \Downarrow -5}\,\text{[Var]}}{\langle 3*x+y,\sigma\rangle \Downarrow 4}\,\text{[Add]} \quad \dfrac{}{\langle z,\sigma\rangle \Downarrow 4}\,\text{[Var]}}{\langle (3*x+y)-z,\sigma\rangle \Downarrow 0}\,\text{[Sub]}$$

root

- This is a **proof tree**.
- Its root is the judgement we want to prove.
- The children of the root are the premises of the rule we have applied (in this case, [Sub]).
- The leaves of the tree contain judgements obtained when applying axioms.

Properties of our definition

The semantics we have defined for arithmetic expressions is deterministic.

That is, given an expression e and state σ, there is a **single** value n such that $\langle e, \sigma \rangle \Downarrow n$.

Proposition

For all $e \in$ **AExp**, $\sigma \in$ **State**, $n_1, n_2 \in \mathbb{Z}$:

If $\langle e, \sigma \rangle \Downarrow n_1$ and $\langle e, \sigma \rangle \Downarrow n_2$, then $n_1 = n_2$.

Prove the previous theorem

How would you define equivalence on arithmetic expressions by using big-step operational semantics?

Two expressions $e_1, e_2 \in \mathsf{AExp}$ are equivalent if and only if:

An equivalence result

- So far we have defined the semantics of arithmetic expressions in two ways:
  - Denotational semantics: $\mathcal{A}[\![e]\!]\ \sigma = n$.
  - Big-step operational semantics: $\langle e, \sigma \rangle \Downarrow n$.
- Are they equivalent?

Theorem

Given an expression $e \in \mathbf{AExp}$*, a state* $\sigma \in \mathbf{State}$*, and a number* $n \in \mathbb{Z}$*. Then:*

$$\mathcal{A}[\![e]\!]\ \sigma = n \iff \langle e, \sigma \rangle \Downarrow n$$

Let us prove it by structural induction on e.

Proof sketch ($\Longrightarrow$)

Let us prove by **structural induction on e** that $\langle e, \sigma \rangle \Downarrow \mathcal{A}[\![e]\!]\sigma$

- **Base cases:**
  - Case $e \equiv n$ for some $n \in \mathbb{Z}$.
    By using [Const$_{BS}$] rule, we get $\langle n, \sigma \rangle \Downarrow n$, with $n = \mathcal{A}[\![n]\!]\ \sigma$.
  - Case $e \equiv x$ for some $x \in$ **Var**.
    By using [Var$_{BS}$] rule, we get $\langle x, \sigma \rangle \Downarrow \sigma(x)$, with $\sigma(x) = \mathcal{A}[\![x]\!]\ \sigma$.
- **Inductive step:**
  - Case $e \equiv e_1 + e_2$ for some $e_1, e_2 \in$ **AExp**.
    By induction hypothesis, we know that $\langle e_1, \sigma \rangle \Downarrow \mathcal{A}[\![e_1]\!]\ \sigma$ and $\langle e_2, \sigma \rangle \Downarrow \mathcal{A}[\![e_2]\!]\ \sigma$. We use [Add$_{BS}$]:

$$\frac{\langle e_1, \sigma \rangle \Downarrow \mathcal{A}[\![e_1]\!]\ \sigma \quad \langle e_2, \sigma \rangle \Downarrow \mathcal{A}[\![e_2]\!]\ \sigma}{\langle e_2, \sigma \rangle \Downarrow \mathcal{A}[\![e_1]\!]\ \sigma + \mathcal{A}[\![e_2]\!]\ \sigma}\ [\text{Add}_{BS}]$$

    and since $\mathcal{A}[\![e_1 + e_2]\!]\ \sigma = \mathcal{A}[\![e_1]\!]\ \sigma + \mathcal{A}[\![e_2]\!]\ \sigma$, it holds that $\langle e_1 + e_2, \sigma \rangle \Downarrow \mathcal{A}[\![e_1 + e_2]\!]\ \sigma$.

Proof sketch ($\Longleftarrow$)

- Assume that $\langle e, \sigma \rangle \Downarrow n$.
- We have already proved that $\langle e, \sigma \rangle \Downarrow \mathcal{A}[\![e]\!]\ \sigma$.
- Therefore, since big-step semantics is deterministic, we get that $\mathcal{A}[\![e]\!]\ \sigma = n$.

Small-step semantics of arithmetic expressions

- Recall the two styles of operational semantics:
    - **Big-step:** describes how the final result is obtained.
    - **Small-step:** describes the steps of computation.
- Small-step semantics resembles a term rewriting system (TRS), in which an expression is successively transformed until we get a number.
- We define a rewriting relation ($\longrightarrow$) with a set of rules.
- We have judgements of the form $\sigma \vdash e \longrightarrow e'$ specifying that, under a state σ, an expression e becomes e' after a step of computation.

Small-step rules

- The following rule specifies that a variable is evaluated, in a single step, to the value it contains.

$$\frac{}{\sigma \vdash x \longrightarrow \sigma(x)} \ [\text{Var}_{SS}]$$

- If we want to perform a step of computation on a addition, we can try to reduce its left operand.

$$\frac{\sigma \vdash e_1 \longrightarrow e_1'}{\sigma \vdash e_1 + e_2 \longrightarrow e_1' + e_2} \ [\text{Add1}_{SS}]$$

For example, we can derive $[x \mapsto 5] \vdash x + y \longrightarrow 5 + y$.

Small-step rules

- If the first operand is already a number, we can try to reduce the second operand:

$$\frac{\sigma \vdash e_2 \longrightarrow e_2'}{\sigma \vdash n_1 + e_2 \longrightarrow n_1 + e_2'} \; [\mathrm{Add2}_{SS}]$$

- If both operands are numbers, we perform the addition:

$$\frac{}{\sigma \vdash n_1 + n_2 \longrightarrow n_1 +_{\mathbb{Z}} n_2} \; [\mathrm{Add3}_{SS}]$$

Small-step rules

We have similar sets of rules for substraction and multiplication:

$$\frac{\sigma \vdash e_1 \longrightarrow e_1'}{\sigma \vdash e_1 - e_2 \longrightarrow e_1' - e_2}\ [\text{Sub1}_{SS}] \qquad \frac{\sigma \vdash e_2 \longrightarrow e_2'}{\sigma \vdash n_1 - e_2 \longrightarrow n_1 - e_2'}\ [\text{Sub2}_{SS}]$$

$$\frac{}{\sigma \vdash n_1 - n_2 \longrightarrow n_1 -_{\mathbb{Z}} n_2}\ [\text{Sub3}_{SS}]$$

$$\frac{\sigma \vdash e_1 \longrightarrow e_1'}{\sigma \vdash e_1 * e_2 \longrightarrow e_1' * e_2}\ [\text{Mul1}_{SS}] \qquad \frac{\sigma \vdash e_2 \longrightarrow e_2'}{\sigma \vdash n_1 * e_2 \longrightarrow n_1 * e_2'}\ [\text{Mul2}_{SS}]$$

$$\frac{}{\sigma \vdash n_1 * n_2 \longrightarrow n_1 \cdot n_2}\ [\text{Mul3}_{SS}]$$

No rule for constants?

- We cannot perform a step of computation on a constant, since the constant is an already evaluated value.
- Using TRS terminology, we say that constants are normal forms.

Does small-step semantics specify the order in which evaluate the operands in an addition?

Can we do the same with big-step semantics or denotational semantics?

Multiple computation steps

- We have seen that the small-step semantics specifies how to perform a single step of computation.
- But, in order to know the value to which an expression is evaluated, we may need several steps of computation.
- For a $k \in \mathbb{N}$, we define $\sigma \vdash e \longrightarrow^k e'$ as follows:
    - For every $e \in \textbf{AExp}$, $\sigma \vdash e \longrightarrow^0 e$.
        - That is, an expression is transformed into itself in zero steps.
    - For every $e, e', e'' \in \textbf{AExp}$, if $\sigma \vdash e \longrightarrow e'$ and $\sigma \vdash e' \longrightarrow^k e''$, then $\sigma \vdash e \longrightarrow^{k+1} e''$.
        - That is, if we perform one step of computation and, from the result, we do k steps, we get $k+1$ steps from the initial expression.
- We say that $\sigma \vdash e \longrightarrow^* e'$ if and only if there exists some $k \geq 0$ such that $\sigma \vdash e \longrightarrow^k e'$.

Multiple computation steps

Example

Assume $\sigma = [x \mapsto 3, y \mapsto -5, z \mapsto 4]$. We get the following derivation sequence under σ:

$$\begin{aligned}
(3 * x + y) - z &\longrightarrow (3 * 3 + y) - z \\
&\longrightarrow (9 + y) - z \\
&\longrightarrow (9 + (-5)) - z \\
&\longrightarrow 4 - z \\
&\longrightarrow 4 - 4 \\
&\longrightarrow 0
\end{aligned}$$

Therefore, $\sigma \vdash (3 * x + y) - z \longrightarrow^* 0$

Equivalence: big-step semantics and small-step semantics

Theorem

For any $e \in \textsf{AExp}$, $\sigma \in \textsf{State}$, $n \in \mathbb{Z}$:

$$\langle e, \sigma \rangle \Downarrow n \qquad \Longleftrightarrow \qquad \sigma \vdash e \longrightarrow^{*} n$$

Auxiliary lemmas

Lemma (1)

Given $e_1, e_2, e_1', e_2' \in \textbf{AExp}$, $n_1 \in \mathbb{Z}$ *and* $\sigma \in \textbf{State}$:

1. *If* $\sigma \vdash e_1 \longrightarrow^* e_1'$ *then* $\sigma \vdash e_1 + e_2 \longrightarrow^* e_1' + e_2$
2. *If* $\sigma \vdash e_2 \longrightarrow^* e_2'$ *then* $\sigma \vdash n_1 + e_2 \longrightarrow^* n_1 + e_2'$

Lemma (2)

Assume that we have a derivation sequence $\sigma \vdash e_1 + e_2 \longrightarrow^k n$. *Then we get:*

1. $\sigma \vdash e_1 \longrightarrow^{k_1} n_1$ *for some* $k_1 < k$.
2. $\sigma \vdash e_2 \longrightarrow^{k_2} n_2$ *for some* $k_2 < k$.

for some $n_1, n_2 \in \mathbb{Z}$ *such that* $n = n_1 + n_2$.

Similarly with expressions of the form $e_1 - e_2$, $e_1 * e_2$.

Prove the lemmas shown above

Which proof technique have you used?

Hint: It is not proved by structural induction on e.

... AND NOW PROVE THE EQUIVALENCE RESULT

Theorem

For any $e \in \mathsf{AExp}$, $\sigma \in \mathsf{State}$, $n \in \mathbb{Z}$:

$$\langle e, \sigma \rangle \Downarrow n \qquad \Longleftrightarrow \qquad \sigma \vdash e \longrightarrow^* n$$

Interlude: boolean expressions

Boolean expressions: syntax

Let BExp be the set of boolean expressions generated by the following grammar:

$$\begin{array}{rcll} b & := & \mathit{true} & \{\text{boolean constant}\} \\ & \mid & \mathit{false} & \{\text{boolean constant}\} \\ & \mid & e_1 = e_2 & \{\text{where } e_1, e_2 \in \mathsf{AExp}\} \\ & \mid & e_1 \leq e_2 & \{\text{where } e_1, e_2 \in \mathsf{AExp}\} \\ & \mid & \neg b & \{\text{negation}\} \\ & \mid & b_1 \wedge b_2 & \{\text{conjunction}\} \end{array}$$

Boolean expressions: denotational semantics

- The semantics of a boolean expression also depends on the state.
- It is defined as a function:

$$\mathcal{B}[\![b]\!] :: \textbf{BExp} \rightarrow \textbf{State} \rightarrow \{true, false\}$$

Boolean expressions: denotational semantics

$$
\begin{aligned}
\mathcal{B}[\![\mathit{true}]\!]\,\sigma &= \mathit{true} \\
\mathcal{B}[\![\mathit{false}]\!]\,\sigma &= \mathit{false} \\
\mathcal{B}[\![e_1 = e_2]\!]\,\sigma &= \begin{cases} \mathit{true} & \text{if } \mathcal{A}[\![e_1]\!]\,\sigma = \mathcal{A}[\![e_2]\!]\,\sigma \\ \mathit{false} & \text{otherwise} \end{cases} \\
\mathcal{B}[\![e_1 \leq e_2]\!]\,\sigma &= \begin{cases} \mathit{true} & \text{if } \mathcal{A}[\![e_1]\!]\,\sigma \leq_{\mathbb{Z}} \mathcal{A}[\![e_2]\!]\,\sigma \\ \mathit{false} & \text{otherwise} \end{cases} \\
\mathcal{B}[\![\neg b]\!]\,\sigma &= \begin{cases} \mathit{true} & \text{if } \mathcal{B}[\![b]\!]\,\sigma = \mathit{false} \\ \mathit{false} & \text{otherwise} \end{cases} \\
\mathcal{B}[\![b_1 \wedge b_2]\!]\,\sigma &= \begin{cases} \mathit{true} & \text{if } \mathcal{B}[\![b_1]\!]\,\sigma = \mathit{true} \text{ and } \mathcal{B}[\![b_2]\!]\,\sigma = \mathit{true} \\ \mathit{false} & \text{otherwise} \end{cases}
\end{aligned}
$$

While: an imperative language

The While language: syntax

Let **Stm** be the set of program statements generated by the following grammar:

$$
\begin{array}{rcll}
S & := & \texttt{skip} & \{\text{no operation}\} \\
 & \mid & x := e & \{\text{assignment } (x \in \textbf{Var}, e \in \textbf{AExp})\} \\
 & \mid & S_1; S_2 & \{\text{sequence}\} \\
 & \mid & \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 & \{\text{conditional}\} \\
 & \mid & \texttt{while } b \texttt{ do } S & \{\text{while loop}\}
\end{array}
$$

Example

```
x := 1; i := 0;
while i ≤ n − 1 do
      x := 2 ∗ x;
      i := i + 1;
```

Semantics of while language

- The result of executing an arithmetic expression is a value in $\mathbb{Z}$.
- The result of executing a boolean expression is a value in $\{true, false\}$.
- **What is the result of executing a statement?**
  - A statement does not ”return” anything by itself, but it mutates the variables in the program.
  - Therefore, the result of executing a statement is a state that contains the values of the variables after program execution.

Big-step semantics for statements

- We have judgements of the form:

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

  meaning that, if we execute S starting from the state σ, it terminates and the state after the execution is σ'.

- **Skip instruction:**

$$\frac{}{\langle \texttt{skip}, \sigma \rangle \Downarrow \sigma} \ [\text{Skip}_{BS}]$$

- **Assignment:**

$$\frac{}{\langle x := e, \sigma \rangle \Downarrow \sigma[x \mapsto \mathcal{A}[\![e]\!]\ \sigma]} \ [\text{Assign}_{BS}]$$

- Program sequence:

$$\frac{\langle S_1, \sigma\rangle \Downarrow \sigma' \quad \langle S_2, \sigma'\rangle \Downarrow \sigma''}{\langle S_1; S_2, \sigma\rangle \Downarrow \sigma''}\ [\text{Seq}_{BS}]$$

- Conditional expressions:

$$\frac{\mathcal{B}[\![b]\!]\,\sigma = \mathit{true} \quad \langle S_1, \sigma\rangle \Downarrow \sigma'}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, \sigma\rangle \Downarrow \sigma'}\ [\text{IfT}_{BS}]$$

$$\frac{\mathcal{B}[\![b]\!]\,\sigma = \mathit{false} \quad \langle S_2, \sigma\rangle \Downarrow \sigma'}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, \sigma\rangle \Downarrow \sigma'}\ [\text{IfF}_{BS}]$$

Big-step semantics for statements

- While loops:

$$\frac{\mathcal{B}[\![b]\!]\ \sigma = \mathit{false}}{\langle \texttt{while}\ b\ \texttt{do}\ S, \sigma\rangle \Downarrow \sigma}\ [\text{WhileF}_{BS}]$$

$$\frac{\mathcal{B}[\![b]\!]\ \sigma = \mathit{true} \quad \langle S, \sigma\rangle \Downarrow \sigma' \quad \langle \texttt{while}\ b\ \texttt{do}\ S, \sigma'\rangle \Downarrow \sigma''}{\langle \texttt{while}\ b\ \texttt{do}\ S, \sigma\rangle \Downarrow \sigma''}\ [\text{WhileT}_{BS}]$$

Big-step semantics: determinism

Proposition

For any $S \in \mathbf{Stm}, \sigma, \sigma', \sigma'' \in \mathbf{State}$:

$$\textit{If } \langle S, \sigma \rangle \Downarrow \sigma' \textit{ and } \langle S, \sigma \rangle \Downarrow \sigma'' \textit{ then } \sigma' = \sigma''$$

Try to prove the proposition shown in previous slide

Which problem did you encounter?

Induction on the size of derivation trees

In order to prove that a property P holds for any judgement $\langle S, \sigma\rangle \Downarrow \sigma'$:

- **Base case:** Prove P by assuming that $\langle S, \sigma\rangle \Downarrow \sigma'$ has been obtained from an axiom in our set of rules.
- **Inductive step:** Prove P by assuming that $\langle S, \sigma\rangle \Downarrow \sigma'$ has been obtained from applying a rule:

$$\frac{\langle S_1, \sigma_1\rangle \Downarrow \sigma_1' \quad \cdots \quad \langle S_n, \sigma_n\rangle \Downarrow \sigma_n'}{\langle S, \sigma\rangle \Downarrow \sigma'}$$

  and assuming that P holds for every child $\langle S_i, \sigma_i\rangle \Downarrow \sigma_i'$ where $1 \leq i \leq n$ (**induction hypothesis**).

Example

In the proof of the proposition regarding determinism:

- Case $e = \texttt{while}\ b\ \texttt{do}\ e_1$:
  - If $\mathcal{B}[\![b]\!]\ \sigma = \mathit{true}$ then $\sigma = \sigma' = \sigma''$.
  - If $\mathcal{B}[\![b]\!]\ \sigma = \mathit{false}$ we know that there are σ_1, σ_1' such that:

$$\frac{\langle e_1, \sigma\rangle \Downarrow \sigma_1 \quad \langle \texttt{while}\ b\ \texttt{do}\ e_1, \sigma_1\rangle \Downarrow \sigma'}{\langle \texttt{while}\ b\ \texttt{do}\ e_1, \sigma\rangle \Downarrow \sigma'}$$

$$\frac{\langle e_1, \sigma\rangle \Downarrow \sigma_1' \quad \langle \texttt{while}\ b\ \texttt{do}\ e_1, \sigma_1'\rangle \Downarrow \sigma''}{\langle \texttt{while}\ b\ \texttt{do}\ e_1, \sigma\rangle \Downarrow \sigma''}$$

  We have $\langle e_1, \sigma\rangle \Downarrow \sigma_1$ and $\langle e_1, \sigma\rangle \Downarrow \sigma_1'$ so, by i.h., $\sigma_1 = \sigma_1'$.

  Moreover, we have $\langle \texttt{while}\ b\ \texttt{do}\ e_1, \sigma_1\rangle \Downarrow \sigma'$ and $\langle \texttt{while}\ b\ \texttt{do}\ e_1, \sigma_1\rangle \Downarrow \sigma''$, so $\sigma' = \sigma''$.

Does always exist a final state?

Given a program S and an initial state σ. Is there always a σ' such that $\langle S, \sigma \rangle \Downarrow \sigma'$.

Small-step semantics for statements

- The small-step semantics of While puts emphasis on the steps of computation when executing a program.
- Here we have two kinds of judgements:
  - $\langle S, \sigma \rangle \longrightarrow \langle S', \sigma' \rangle$

    Performing a computation step on a program S under a state σ leads to a state σ', and S' is the program that represents the remaining computation.
  - $\langle S, \sigma \rangle \longrightarrow \sigma'$

    Performing a computation step on S under a state σ terminates the program. The final state is σ'.
- We use the word configuration to denote a pair $\langle S, \sigma \rangle$ or a state σ. The $(\longrightarrow)$ is a relation between configurations.

Small-step semantic rules

- Skip statement:

$$\frac{}{\langle \texttt{skip}, \sigma \rangle \longrightarrow \sigma} \text{ [Skip}_{SS}\text{]}$$

- Assignment:

$$\frac{}{\langle x := e, \sigma \rangle \longrightarrow \sigma[x \mapsto \mathcal{A}[\![e]\!]\ \sigma]} \text{ [Assign}_{SS}\text{]}$$

Small-step semantic rules

- Sequence of statements:

$$\frac{\langle S_1, \sigma\rangle \longrightarrow \sigma'}{\langle S_1; S_2, \sigma\rangle \longrightarrow \langle S_2, \sigma'\rangle}\ [\text{Seq1}_{SS}]$$

$$\frac{\langle S_1, \sigma\rangle \longrightarrow \langle S_1', \sigma'\rangle}{\langle S_1; S_2, \sigma\rangle \longrightarrow \langle S_1'; S_2, \sigma'\rangle}\ [\text{Seq2}_{SS}]$$

Small-step semantic rules

- Conditionals:

$$\frac{\mathcal{B}[\![b]\!]\ \sigma = \mathit{true}}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, \sigma\rangle \longrightarrow \langle S_1, \sigma\rangle}\ [\mathrm{If1}_{SS}]$$

$$\frac{\mathcal{B}[\![b]\!]\ \sigma = \mathit{false}}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, \sigma\rangle \longrightarrow \langle S_2, \sigma\rangle}\ [\mathrm{If2}_{SS}]$$

Small-step semantic rules

- While loops:

$$\frac{\mathcal{B}[\![b]\!]\ \sigma = \mathit{false}}{\langle \texttt{while}\ b\ \texttt{do}\ S, \sigma\rangle \longrightarrow \sigma}\ [\text{While1}_{SS}]$$

$$\frac{\mathcal{B}[\![b]\!]\ \sigma = \mathit{true}}{\langle \texttt{while}\ b\ \texttt{do}\ S, \sigma\rangle \longrightarrow \langle S; \texttt{while}\ b\ \texttt{do}\ S, \sigma\rangle}\ [\text{While2}_{SS}]$$

Derivation sequences

- A **derivation sequence** of a statement $S \in \mathbf{Stm}$ is either:
  - A finite sequence $\langle S_1, \sigma_1 \rangle \longrightarrow \cdots \longrightarrow \langle S_n, \sigma_n \rangle \longrightarrow \sigma'$ of length n, or
  - An infinite sequence $\langle S_1, \sigma_1 \rangle \longrightarrow \cdots \longrightarrow \langle S_n, \sigma_n \rangle \longrightarrow \cdots$
- As in arithmetic expressions, we also use the notation:

$$\langle S, \sigma \rangle \longrightarrow^k \langle S', \sigma' \rangle \text{ or } \langle S, \sigma \rangle \longrightarrow^k \sigma'$$

$$\langle S, \sigma \rangle \longrightarrow^* \langle S', \sigma' \rangle \text{ or } \langle S, \sigma \rangle \longrightarrow^* \sigma'$$

to specify k, zero, one, or more computation steps.

Derivation sequences

Example

Given

$$\begin{array}{rl}
 & \langle \texttt{while}\ n \geq 0\ \texttt{do}\ n := n - 1, [n \mapsto 2]\rangle \\
\longrightarrow & \langle n := n - 1; \texttt{while}\ n \geq 0\ \texttt{do}\ n := n - 1, [n \mapsto 2]\rangle \\
\longrightarrow & \langle \texttt{while}\ n \geq 0\ \texttt{do}\ n := n - 1, [n \mapsto 1]\rangle \\
\longrightarrow & \langle n := n - 1; \texttt{while}\ n \geq 0\ \texttt{do}\ n := n - 1, [n \mapsto 1]\rangle \\
\longrightarrow & \langle \texttt{while}\ n \geq 0\ \texttt{do}\ n := n - 1, [n \mapsto 0]\rangle \\
\longrightarrow & \langle n := n - 1; \texttt{while}\ n \geq 0\ \texttt{do}\ n := n - 1, [n \mapsto 0]\rangle \\
\longrightarrow & \langle \texttt{while}\ n \geq 0\ \texttt{do}\ n := n - 1, [n \mapsto -1]\rangle \\
\longrightarrow & [n \mapsto -1]
\end{array}$$

Therefore, $\langle \texttt{while}\ n \geq 0\ \texttt{do}\ n := n - 1, [n \mapsto 2]\rangle \longrightarrow^* [n \mapsto -1]$

How would you define program equivalence on statements?

We say that $S_1, S_2 \in \textbf{Stm}$ are equivalent iff:

An equivalence result

Big-step semantics and small-step semantics are equivalent in the following sense:

Theorem

For any $S \in \textbf{Stm}$, $\sigma, \sigma' \in \textbf{State}$:

$$\langle S, \sigma \rangle \Downarrow \sigma' \quad \Longleftrightarrow \quad \langle S, \sigma \rangle \longrightarrow^* \sigma'$$

Auxiliary lemmas

Lemma (1)

Given $S_1, S_2 \in \textbf{Stm}$ *and* $\sigma, \sigma' \in \textbf{State}$:

$$\text{If } \langle S_1, \sigma\rangle \longrightarrow^* \sigma' \text{ then } \langle S_1; S_2, \sigma\rangle \longrightarrow^* \langle S_2, \sigma'\rangle$$

Lemma (2)

Given $S_1, S_2 \in \textbf{Stm}$ *and* $\sigma, \sigma' \in \textbf{State}$ *such that* $\langle S_1; S_2, \sigma\rangle \longrightarrow^* \sigma'$ *with length* k*, there exists some* $\sigma'' \in \textbf{State}$ *such that:*

- $\langle S_1, \sigma\rangle \longrightarrow^* \sigma''$ *with length* $< k$.
- $\langle S_2, \sigma''\rangle \longrightarrow^* \sigma'$ *with length* $< k$.

Prove the equivalence result between big-step and small-step semantics

How big-step and small-step semantics behave when dealing with nonterminating programs?

For example, assume the program `while` *true* `do skip`.

Big-step vs small-step

- Big-step semantics is useful to study properties that relate initial and final states:
  - After executing S, is x bound to a number?
  - After executing S, does y take an integer value between 5 and 10?
- Small-step semantics is useful to detect situations occurring throughout the execution of a program:
  - Could x take a negative value during the execution of S?
  - Could a dangling pointer be accessed during the execution of S?
  - Could two threads run the same instruction during the execution of S?

Denotational semantics for statements

- A program S transforms an initial state into a final state.
- Therefore, we have to define the following semantic function $\mathcal{S}[\![\_]\!]$:

$$\mathcal{S}[\![\_]\!] : \textbf{Stm} \to \textbf{State} \to \textbf{State}$$

- However, this function is not necessarily defined for every input state.

Example

$$S \equiv \texttt{if}\ (x \leq 0)\ \texttt{then}\ (\texttt{while}\ \mathit{true}\ \texttt{do}\ \texttt{skip})\ \texttt{else}\ x := 3$$

- We want $\mathcal{S}[\![S]\!]\ [x \mapsto 4]$ to be defined, and equal to $[x \mapsto 3]$.
- We want $\mathcal{S}[\![S]\!]\ [x \mapsto -2]$ to be undefined.

Denotational semantics for statements

- Given the above, given a statement S, the semantics of S is a partial function from **State** to **State**:

$$\mathcal{S}[\![\_]\!] : \mathbf{Stm} \to \mathbf{State} \to \mathbf{State}_\perp$$

where $State_\perp$ is defined as $\mathbf{State} \cup \{\perp\}$.
  - $\perp$ denotes an undefined result.

Example

$$S \equiv \mathtt{if}\ (x \leq 0)\ \mathtt{then}\ (\mathtt{while}\ \mathit{true}\ \mathtt{do}\ \mathtt{skip})\ \mathtt{else}\ x := 3$$

$$\mathcal{S}[\![S]\!] = \lambda\sigma. \begin{cases} \sigma[x \mapsto 3] & \text{if } \sigma(x) > 0 \\ \perp & \text{otherwise} \end{cases}$$

Denotational semantics for statements

- The denotation of `skip` is a function returning the input state as-is:
$$\mathcal{S}[\![\texttt{skip}]\!] = \mathit{id}$$
where $\mathit{id} = \lambda\sigma.\sigma$.
- The denotation of an assignment $x := e$ is a function that updates the x variable in the input state.
$$\mathcal{S}[\![x := e]\!] = \lambda\sigma.\sigma\,[x \mapsto (\mathcal{A}[\![e]\!]\,\sigma)]$$

Denotational semantics for statements

- The semantics of a sequence $S_1; S_2$ is defined as follows:

$$\mathcal{S}[\![S_1; S_2]\!] = \lambda\sigma.\mathcal{S}[\![S_2]\!]\ (\mathcal{S}[\![S_1]\!]\ \sigma)$$

or, equivalently

$$\mathcal{S}[\![S_1; S_2]\!] = \mathcal{S}[\![S_2]\!] \circ \mathcal{S}[\![S_1]\!]$$

Denotational semantics for statements

- The semantics of a conditional is defined as follows:

$$\mathcal{S}[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!] = \lambda\sigma. \begin{cases} \mathcal{S}[\![S_1]\!]\ \sigma & \text{if } \mathcal{B}[\![b]\!]\ \sigma = \mathit{true} \\ \mathcal{S}[\![S_2]\!]\ \sigma & \text{if } \mathcal{B}[\![b]\!]\ \sigma = \mathit{false} \end{cases}$$

- Let us introduce the following notation:

$$\mathit{cond}(f, g, h) = \lambda x. \begin{cases} g(x) & \text{if } f(x) = \mathit{true} \\ h(x) & \text{if } f(x) = \mathit{false} \end{cases}$$

- Therefore, we get:

$$\mathcal{S}[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!] = \mathit{cond}(\mathcal{B}[\![b]\!], \mathcal{S}[\![S_1]\!], \mathcal{S}[\![S_2]\!])$$

Denotational semantics for statements

- Now we define the semantics of a `while` loop:

$$\mathcal{S}[\![\texttt{while}\ b\ \texttt{do}\ S]\!] = \lambda\sigma. \begin{cases} \mathcal{S}[\![S; \texttt{while}\ b\ \texttt{do}\ S]\!]\ \sigma & \text{if } \mathcal{B}[\![b]\!]\ \sigma = \mathit{true} \\ \sigma & \text{if } \mathcal{B}[\![b]\!]\ \sigma = \mathit{false} \end{cases}$$

  or, equivalently,

$$\mathcal{S}[\![\texttt{while}\ b\ \texttt{do}\ S]\!] = \mathit{cond}(\mathcal{B}[\![b]\!], \mathcal{S}[\![\texttt{while}\ b\ \texttt{do}\ S]\!] \circ \mathcal{S}[\![S]\!], \mathit{id})$$

Summary

$$
\begin{aligned}
\mathcal{S}\llbracket \mathtt{skip} \rrbracket &= id \\
\mathcal{S}\llbracket x := e \rrbracket &= \lambda\sigma.\ \sigma[x \mapsto \mathcal{A}\llbracket e \rrbracket\ \sigma] \\
\mathcal{S}\llbracket S_1; S_2 \rrbracket &= \mathcal{S}\llbracket S_2 \rrbracket \circ \mathcal{S}\llbracket S_1 \rrbracket \\
\mathcal{S}\llbracket \mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2 \rrbracket &= cond(\mathcal{B}\llbracket b \rrbracket, \mathcal{S}\llbracket S_1 \rrbracket, \mathcal{S}\llbracket S_2 \rrbracket) \\
\mathcal{S}\llbracket \mathtt{while}\ b\ \mathtt{do}\ S \rrbracket &= cond(\mathcal{B}\llbracket b \rrbracket, \mathcal{S}\llbracket \mathtt{while}\ b\ \mathtt{do}\ S \rrbracket \circ \mathcal{S}\llbracket S \rrbracket, id)
\end{aligned}
$$

Warning!

Warning! A circular definition

- Our definition is not compositional!

  $$\mathcal{S}[\![\texttt{while}\ b\ \texttt{do}\ S]\!] = cond(\mathcal{B}[\![b]\!], \mathcal{S}[\![\texttt{while}\ b\ \texttt{do}\ S]\!] \circ \mathcal{S}[\![S]\!], id)$$

- We have a problem of circularity. The object being defined $\mathcal{S}[\![\texttt{while}\ b\ \texttt{do}\ S]\!]$ (left-hand side of $=$) occurs in its definition (right-hand side of $=$).
- If we define an object by such an equation, the definition makes sense only when the equation has a single solution.

Are the following definitions correct?

- x is defined as the number such that $x = 3x + 4$.
  - Yes. There is only a solution. It is the same as writing: *x denotes the number -2.*
- x is defined as the number such that $x = x^2 - 2$.
  - No, because there are two possibilities: $x = -1$ and $x = 2$.
- x is defined as the number such that $x = x + 3$.
  - No, because there is no number satisfying this equation.

Which of the following are solutions of the equation?

$$f = cond(\mathcal{B}[\![x \neq 0]\!], f \circ \mathcal{S}[\![\texttt{skip}]\!], id)$$

1. $f = \lambda\sigma. \begin{cases} \sigma & \text{if } \sigma(x) = 0 \\ \perp & \text{otherwise} \end{cases}$
2. $f = \lambda\sigma. \begin{cases} \sigma & \text{if } \sigma(x) = 1 \\ \perp & \text{otherwise} \end{cases}$
3. $f = \lambda\sigma. \begin{cases} \sigma & \text{if } \sigma(x) = 0 \vee \sigma(x) = 1 \\ \perp & \text{otherwise} \end{cases}$
4. $f = \lambda\sigma.\sigma$
5. $f = \lambda\sigma.\perp$

Given the solutions above...

...which one represents more accurately the semantics of the program?

$$\texttt{while}\; x \neq 0 \;\texttt{do skip}$$

Solving circularity: the fixed point theorems

Choosing the right solution

- In this case, from the three solutions shown above, the most suitable one is the "most partial" one (i.e. those with more inputs mapped to $\perp$).
- Therefore, let us define the following order relation between functions **State** $\to$ **State**$_\perp$:

$$f \sqsubseteq g \quad \iff \quad \forall \sigma \in \textbf{State}.\, f(\sigma) = \perp \vee f(\sigma) = g(\sigma)$$

  - In other words, f is lower than g of both coincide in all states, except those in which f is undefined.

Which one is lower? Which one is greater?

According to the $\sqsubseteq$ order defined above.

- $f_1 = \lambda\sigma.\begin{cases}\sigma & \text{if } \sigma(x) = 0 \\ \bot & \text{otherwise}\end{cases}$
- $f_2 = \lambda\sigma.\begin{cases}\sigma & \text{if } \sigma(x) = 0 \vee \sigma(x) = 1 \\ \bot & \text{otherwise}\end{cases}$
- $f_3 = \lambda\sigma.\sigma$

Is there a bottommost element in the set $(\mathsf{State} \to \mathsf{State}_\perp, \sqsubseteq)$? Is there a topmost element?

Equations as fixed points

- Back to our equation:

$$\mathcal{S}[\![\texttt{while}\ b\ \texttt{do}\ S]\!] = cond(\mathcal{B}[\![b]\!], \mathcal{S}[\![\texttt{while}\ b\ \texttt{do}\ S]\!] \circ \mathcal{S}[\![S]\!], id)$$

  - Is there a solution for the unknown $\mathcal{S}[\![\texttt{while}\ b\ \texttt{do}\ S]\!]$?
  - In case there are several, is there a **lowest** one?
- We can rewrite this problem as follows:

$$F(f) = cond(\mathcal{B}[\![b]\!], f \circ \mathcal{S}[\![S]\!], id)$$

  - Is there a fixed point for F?
  - In case there are several, is there a **lowest** one?

Finding fixed points

$$F(f) = cond(\mathcal{B}[\![b]\!], f \circ \mathcal{S}[\![S]\!], id)$$

- It can be proved that F is monotonically increasing for every b and S.
- Therefore, we can build a Kleene ascending chain:

$$\lambda\sigma.\bot \sqsubseteq F(\lambda\sigma.\bot) \sqsubseteq F(F(\lambda\sigma.\bot)) \sqsubseteq F(F(F(\lambda\sigma.\bot))) \sqsubseteq \cdots$$

  or

$$\lambda\sigma.\bot \sqsubseteq F(\lambda\sigma.\bot) \sqsubseteq F^2(\lambda\sigma.\bot) \sqsubseteq F^3(\lambda\sigma.\bot) \sqsubseteq \cdots$$

- We know that, if the chain stabilizes, it does at the **least fixed point**.

But does this chain always stabilize?

Example

Let us consider the following loop:

$$\texttt{while}\ m > 0\ \texttt{do}\ (\underbrace{n := n * m;\ m := m - 1}_{S})$$

We want to find the least fixed point of F, defined as follows:

$$F(f) = cond(\mathcal{B}[\![m > 0]\!], f \circ \mathcal{S}[\![S]\!], id)$$

We start from the bottommost element: $f_0 = \lambda\sigma.\bot$.

Example

We define f_1 as follows:

$$f_1 = F(f_0) = cond(\mathcal{B}[\![m > 0]\!], f_0 \circ \mathcal{S}[\![S]\!], id)$$

Simplifying:

$$f_1 = \lambda\sigma. \begin{cases} \bot & \text{if } \sigma(m) > 0 \\ \sigma & \text{if } \sigma(m) \leq 0 \end{cases}$$

It holds that $f_0 \sqsubseteq f_1$

Example

We define f_2 as follows:

$$f_2 = F(f_1) = cond(\mathcal{B}[\![m > 0]\!], f_1 \circ \mathcal{S}[\![S]\!], id)$$

Simplifying:

$$f_2 = \lambda\sigma. \begin{cases} \bot & \text{if } \sigma(m) > 1 \\ \sigma[n \mapsto \sigma(n) * 1, m \mapsto 0] & \text{if } \sigma(m) = 1 \\ \sigma & \text{if } \sigma(m) \leq 0 \end{cases}$$

It holds that $f_0 \sqsubseteq f_1 \sqsubseteq f_2$

Example

We define f_3 as follows:

$$f_3 = F(f_2) = cond(\mathcal{B}[\![m > 0]\!], f_2 \circ \mathcal{S}[\![S]\!], id)$$

Simplifying:

$$f_3 = \lambda\sigma. \begin{cases} \bot & \text{if } \sigma(m) > 2 \\ \sigma[n \mapsto \sigma(n) * 1 * 2, m \mapsto 0] & \text{if } \sigma(m) = 2 \\ \sigma[n \mapsto \sigma(n) * 1, m \mapsto 0] & \text{if } \sigma(m) = 1 \\ \sigma & \text{if } \sigma(m) \leq 0 \end{cases}$$

It holds that $f_0 \sqsubseteq f_1 \sqsubseteq f_2 \sqsubseteq f_3$

Example

We define f_4 as follows:

$$f_4 = F(f_3) = cond(\mathcal{B}[\![m > 0]\!], f_3 \circ \mathcal{S}[\![S]\!], id)$$

Simplifying:

$$f_4 = \lambda\sigma. \begin{cases} \bot & \text{if } \sigma(m) > 3 \\ \sigma[n \mapsto \sigma(n) * 1 * 2 * 3, m \mapsto 0] & \text{if } \sigma(m) = 3 \\ \sigma[n \mapsto \sigma(n) * 1 * 2, m \mapsto 0] & \text{if } \sigma(m) = 2 \\ \sigma[n \mapsto \sigma(n) * 1, m \mapsto 0] & \text{if } \sigma(m) = 1 \\ \sigma & \text{if } \sigma(m) \leq 0 \end{cases}$$

It holds that $f_0 \sqsubseteq f_1 \sqsubseteq f_2 \sqsubseteq f_3 \sqsubseteq f_4$

The chain does not seem to stabilize. Can you guess, at least, a general definition for any f_i, where $i > 0$?

$$f_i = \lambda\sigma.\begin{cases} \bot & \text{if } \sigma(m) \geq i \\ \sigma[n \mapsto \sigma(n) * \sigma(m)!, m \mapsto 0] & \text{if } 0 < \sigma(m) < i \\ \sigma & \text{if } \sigma(m) \leq 0 \end{cases}$$

Least upper bounds on chains

- Given a chain of elements l in an ordered set $(L, \sqsubseteq)$

$$l_0 \sqsubseteq l_1 \sqsubseteq l_2 \sqsubseteq l_3 \sqsubseteq \cdots$$

  A **least upper bound** of this chain is the lowest element $l \in L$ such that $l_i \subseteq l$ for all $i \in \mathbb{N}$, if such lowest element exists.
- The upper bound of the chain above is denoted by $\bigsqcup_i l_i$, or $\bigsqcup_{i=0}^{\infty} l_i$.

Is there an upper bound in our example chain?

$$f_i = \lambda\sigma. \begin{cases} \bot & \text{if } \sigma(m) \geq i \\ \sigma[n \mapsto \sigma(n) * \sigma(m)!, m \mapsto 0] & \text{if } 0 < \sigma(m) < i \\ \sigma & \text{if } \sigma(m) \leq 0 \end{cases}$$

$$\bigsqcup_i f_i = \lambda\sigma. \begin{cases} \sigma[n \mapsto \sigma(n) * \sigma(m)!, m \mapsto 0] & \text{if } \sigma(m) > 0 \\ \sigma & \text{if } \sigma(m) \leq 0 \end{cases}$$

CCPOs and continuous functions

- An ordered set $(L, \sqsubseteq)$ is a chain-complete partial ordered set (ccpo) iff every ascending chain:

$$l_0 \sqsubseteq l_1 \sqsubseteq l_2 \sqsubseteq \cdots$$

  has a least upper bound.

- If $(L, \sqsubseteq)$ is a ccpo, we say that $f : L \to L$ is continuous iff for every ascending chain:

$$l_0 \sqsubseteq l_1 \sqsubseteq l_2 \sqsubseteq \cdots$$

  such that $\bigsqcup_i f(l_i)$ exists, it holds that

$$\bigsqcup_i f(l_i) = f\left(\bigsqcup_i l_i\right)$$

CCPOs and continuous functions

Proposition

$(\mathbf{State} \to \mathbf{State}_\perp, \sqsubseteq)$ *is a ccpo.*

Proposition

For every $b \in \mathbf{BExp}$ *and* $S \in \mathbf{Stm}$, *the function*

$$F : (\mathbf{State} \to \mathbf{State}_\perp) \to (\mathbf{State} \to \mathbf{State}_\perp)$$

defined by

$$F(f) = cond(\mathcal{B}[\![b]\!], f \circ \mathcal{S}[\![S]\!], id)$$

is continuous.

The fixed point theorem

Theorem

*Let $(D, \sqsubseteq)$ be a ccpo, and $f : L \to L$ a monotonically increasing and continuous function. Then $\bigsqcup_i f^i(\bot)$ is the **least fixed point** of f.*

In case f satisfies the conditions of the theorem, we denote by *lfp f* the least fixed point of f.

$$lfp\ f = \bigsqcup_{i=0}^{\infty} f^i(\bot)$$

Fixed-point theorem: proof

Let us prove that $\bigsqcup_i f^i(\bot)$ is a fixed point, that is:

$$f\left(\bigsqcup_i f^i(\bot)\right) = \bigsqcup_i f^i(\bot)$$

We get:

$$\begin{array}{rcll}
f\,(\bigsqcup_{i=0}^{\infty} f^i(\bot)) & = & \bigsqcup_{i=0}^{\infty} f(f^i(\bot)) & \{\text{since } f \text{ is continuous}\} \\
 & = & \bigsqcup_{i=0}^{\infty} f^{i+1}(\bot) & \\
 & = & \bigsqcup_{j=1}^{\infty} f^j(\bot) & \{\text{variable change: } j = i+1\} \\
 & = & (\bigsqcup_{j=1}^{\infty} f^j(\bot)) \sqcup \bot & \{\text{properties of } \bot\} \\
 & = & \bigsqcup_{j=0}^{\infty} f^j(\bot) &
\end{array}$$

Fixed-point theorem: proof

Now we prove that $\bigsqcup_i f^i(\bot)$ is the least fixed point.

- Assume there is another fixed point d. Obviously,

$$\bot \sqsubseteq d$$

- Since f is monotonic, we get $f(\bot) \sqsubseteq f(d)$, but d is a fixed point, so $f(d) = d$. Therefore:

$$\bot \sqsubseteq f(\bot) \sqsubseteq d$$

- Since f is monotonic, we get $f(f(\bot)) \sqsubseteq f(d)$, so:

$$\bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq d$$

Fixed-point theorem: proof

In general we get:

$$\bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \cdots \sqsubseteq f^i(\bot) \sqsubseteq \cdots \sqsubseteq d$$

Therefore, d is an upper bound of the ascending chain.

But, by definition, $\bigsqcup_i f^i(\bot)$ is the least upper bound of the chain. Therefore:

$$\bigsqcup_i f^i(\bot) \sqsubseteq d$$

which proves the theorem.

Summarizing

- We had the following

  $$\mathcal{S}[\![\texttt{while}\ b\ \texttt{do}\ S]\!] = cond(\mathcal{B}[\![b]\!], \mathcal{S}[\![\texttt{while}\ b\ \texttt{do}\ S]\!] \circ \mathcal{S}[\![S]\!], id)$$

  which does not define $\mathcal{S}[\![\texttt{while}\ b\ \texttt{do}\ S]\!]$ since there might be, in general, many solutions to this equation.
- In case there are many solutions, we want the **least one**.
- Finding a solution to the equation above is equivalent to finding a fixed point of the following function:

  $$F(f) = cond(\mathcal{B}[\![b]\!], f \circ \mathcal{S}[\![S]\!], id)$$

- In case there are many fixed points, we want the **least fixed point**: *lfp F*.

Summarizing

$$F(f) = cond(\mathcal{B}[\![b]\!], f \circ \mathcal{S}[\![S]\!], id)$$

- F is a monotonically increasing and continuous function in a ccpo, so it has a least fixed point, which is:

$$lfp\ F = \bigsqcup_i F^i(\bot)$$

- Therefore, we can rewrite the previous (incorrect) definition by:

$$\mathcal{S}[\![\texttt{while}\ b\ \texttt{do}\ S]\!] = lfp\ F$$

Denotational semantics: final version

$$\begin{array}{rcl}
\mathcal{S}[\![\texttt{skip}]\!] & = & id \\
\mathcal{S}[\![x := e]\!] & = & \lambda\sigma.\ \sigma[x \mapsto \mathcal{A}[\![e]\!]\ \sigma] \\
\mathcal{S}[\![S_1; S_2]\!] & = & \mathcal{S}[\![S_2]\!] \circ \mathcal{S}[\![S_1]\!] \\
\mathcal{S}[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!] & = & cond(\mathcal{B}[\![b]\!], \mathcal{S}[\![S_1]\!], \mathcal{S}[\![S_2]\!]) \\
\mathcal{S}[\![\texttt{while}\ b\ \texttt{do}\ S]\!] & = & lfp\ F \\
 & & \textbf{where } F(f) = cond(\mathcal{B}[\![b]\!], f \circ \mathcal{S}[\![S]\!], id)
\end{array}$$

An equivalence result

Equivalence between big-step operational semantics and denotational semantics:

Theorem

Given $S \in \mathbf{Stm}$, $\sigma, \sigma' \in \mathbf{State}$:

- $\mathcal{S}[\![S]\!]\ \sigma = \sigma' \iff \langle S, \sigma \rangle \Downarrow \sigma'$
- $\mathcal{S}[\![S]\!]\ \sigma = \bot \iff$ *there is no* σ' *such that* $\langle S, \sigma \rangle \Downarrow \sigma'$.