Semantics of arithmetic and Boolean expressions

William Steingartner

william.steingartner@tuke.sk

Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice, Slovakia

Semantics of programming languages

Spring 2022/2023

CEEPUS

Theme 02 – Semantics of expressions

Topics

1. Formal definition of programming language.
2. Formal definition of binary numbers language.
3. Semantics of arithmetic expressions.
4. Semantics of Boolean expressions.

Formal definition of programming language

Formal definition of programming language has the following parts:

- definition of **abstract syntax** with:
  - **syntactic domains** – the elements of one syntactic domain must be of the same internal structure,
  - **production rules** – they define acceptable forms of elements in particular syntactic domains,
- definition of **semantics** with:
  - **semantic domains,**
  - specification of **semantic functions**, i.e. their domains and ranges,
  - **semantic equations** or **derivation rules**, that defines particular semantic functions.

Formal definition of language

Semantic domain is a structure which contains meanings of particular syntactic forms from the given syntactic domain.
For simplicity we will use only **semantic domains based on sets**:

- simple sets like sets of integers $\mathbf{Z}$,
- the results of set operations, e.g. union, intersection, etc.,
- sets of functions over defined semantic domains.

Semantic domain of programming language is an union of all semantic domains of language. We say that this set is a **model of programming language**.

Formal definition of language

Semantic function maps syntactic domain into appropriate semantic domain. Its **specification** is denoted in general form:

$$\mathscr{F} : \mathbf{Synt} \rightarrow \mathbf{Sem}$$

where

- $\mathbf{Synt}$ is replaced by concrete syntactic domain, and
- $\mathbf{Sem}$ is replaced by appropriate semantic domain.

We specify one semantic function for each syntactic domain.

Semantic function **is defined** by:

- semantic equations or
- derivation rules,

which define the meaning of particular syntactic forms in production rule for given syntactic domain.

Formal definition of binary numbers language

As an example, we present here the binary numbers language and the definition of its syntax and semantics.

1. Formal syntax:

a. we need only one syntactic domain for binary numerals:

$$n \in \mathbf{Bin},$$

b. the abstract syntax could then be specified by production rule:

$$n ::= 0 \mid 1 \mid n0 \mid n1.$$

Formal definition of binary numbers language

2. Semantics

a. meaning of any numeral shall by unique number in decadic form, which are elements of sets of integer $\mathbf{Z}$.

b. we specify one semantic function (for one syntactic domain):

$$\mathscr{N} : \mathbf{Bin} \rightarrow \mathbf{Z},$$

and we want $\mathscr{N}$ to be a total function because we want to determine a unique number for each numeral of $\mathbf{Bin}$.

Formal definition of binary numbers language

c. we define four **semantic equations**, one for each alternative in production rule.

They define the meaning of particular forms in production rule in terms of semantic domain ($\mathbf{Z}$) elements:

$$\begin{array}{lcl} \mathscr{N}[\![0]\!] & = & \mathbf{0}, \\ \mathscr{N}[\![1]\!] & = & \mathbf{1}, \\ \mathscr{N}[\![n0]\!] & = & \mathbf{2} \otimes \mathscr{N}[\![n]\!], \\ \mathscr{N}[\![n1]\!] & = & \mathbf{2} \otimes \mathscr{N}[\![n]\!] \oplus \mathbf{1}. \end{array}$$

- $'[\!['$ and $']\!]'$ are **semantic brackets**, inside them **syntactic form** is enclosed,
- $\mathscr{N}[\![n]\!]$ is **application of semantic function** on element of syntactic domain, the result here is meaning of binary numeral n, i.e. an integer $\mathscr{N}[\![n]\!] \in \mathbf{Z}$,
- $\otimes, \oplus$ – are real arithmetic operations,
- $\mathbf{0}, \mathbf{1}, \mathbf{2}$ – are numbers in contrast to symbols 0, 1 and 2 in syntax.

Example

The notation 101 is well-formed syntactic form of binary numeral. We find its meaning.

A **semantics** is computed by applying the semantic function on the particular alternatives in production rule.
Numeral 101 is of the 4^{th} form in production rule – $n1$, so we apply the 4^{th} semantic equation and n be 10:

$$\mathscr{N}[\![\, 101 \,]\!] = \mathbf{2} \otimes \mathscr{N}[\![\, 10 \,]\!] \oplus \mathbf{1} =$$

Numeral 10 in semantic brackets is of the 3^{rd} form in production rule, so we apply the 3^{rd} semantic equation. After that we continue until we find the integer which is the meaning of binary numeral 101:

$$\begin{aligned} &= \mathbf{2} \otimes (\mathbf{2} \otimes \mathscr{N}[\![\, 1 \,]\!]) \oplus \mathbf{1} = \\ &= \mathbf{2} \otimes (\mathbf{2} \otimes \mathbf{1}) \oplus \mathbf{1} = \\ &= \mathbf{5} \end{aligned}$$

$\square$

Problem as a motivation

There exists a complete canonical representation in the form of reduced numerals (numerals without leading zeros). The syntax for these is:

$$n' ::= 0 \mid 1 \mid 1n$$

where $n' \in \mathbf{Bin}'$, the set of reduced numerals, and $n \in \mathbf{Bin}$ as defined before. Notice that it is not quite as easy to give a semantic function

$$\mathscr{N}' : \mathbf{Bin}' \to \mathbf{Z},$$

for the reduced numerals.
The most convenient way is by means of an auxiliary function

$$\mathscr{L} : \mathbf{Bin} \to \mathbf{N}$$

which gives the „length“ of a number.

Define $\mathscr{N}'$!

Structural induction

Structural induction is a proof method that is used in mathematical logic, computer science, graph theory, and some other mathematical fields.

Structural induction is used to prove that some proposition $P(x)$ holds for all x of some sort of recursively defined structure.

In our course we will use proofs by structural induction on the **structure** of particular **syntactic domains**.

Mathematical and structural induction

By **mathematical induction** we prove some property P on natural numbers:

1. we **prove** the property for value 1, i.e. $P(1)$,
2. we **formulate** an **induction hypothesis**:
   - we **assume** that the property P holds for all naturals $n \leq k$, i.e. $P(k)$,
3. we **prove** that the property P holds for $k+1$, i.e. $P(k+1)$.

Then the property holds for all naturals: $P(n), n \in \mathbf{N}$.

The **structural induction** proves some property P for some syntactic domain:

1. we **prove**, that the property holds for simple (atomic) elements in syntactic domain,
2. we **formulate** an induction hypothesis: we **assume**, that the property P holds for sub-elements of each composite element,
3. we **prove**, that the property P holds for each composite element.

Then the property holds for all elements in syntactic domain.

Example of proof

Lemma: Semantic function $\mathscr{N} : \mathbf{Bin} \to \mathbf{Z}$ is a total function.

Proof:

$\mathscr{N}$ is total function, if it is defined for **all** arguments, i.e.

if for all arguments $n \in \mathbf{Bin}$ there is **exactly one** number $\mathbf{n} \in \mathbf{Z}$ such that

$$\mathscr{N}[\![\, n \,]\!] = \mathbf{n} \qquad\qquad (*)$$

To prove $(*)$ we have to prove it for all possibilities in production rule.

1. We prove the property for the basis elements of $\mathbf{Bin}$:
   - the case $n = 0$: only one of the semantic clauses defining $\mathscr{N}$ can be used and it gives $\mathscr{N}[\![\, 0 \,]\!] = \mathbf{0}$; so clearly there is exactly one number $\mathbf{n}$ in $\mathbf{Z}$ such that $\mathscr{N}[\![\, n \,]\!] = \mathbf{n}$, namely $\mathbf{0}$;
   - the case $n = 1$: the proof is similar.

Proof by structural induction

2. Composite elements in $\mathbf{Bin}$ are $n0$ and $n1$.
   - the case $n = n'0$:
     we see that only one of the clauses is applicable and we have
     $$\mathscr{N}[\![\, n'0 \,]\!] = \mathbf{2} \otimes \mathscr{N}[\![\, n' \,]\!].$$
     We can now apply the induction hypothesis to n' and get that there is exactly one number $\mathbf{n}'$ such that $\mathscr{N}[\![\, n' \,]\!] = \mathbf{n}'$.

     Then is is clear that there is exactly one number $\mathbf{n}$ (namely $\mathbf{2} \otimes \mathbf{n}'$) such that $\mathscr{N}[\![\, n \,]\!] = \mathbf{n}$.
   - the case $n = n'1$: the proof is similar.

$\square$

Simple imperative language *Jane*

Syntax

Syntactic domains:

$n \in \mathbf{Num}$ — for numerals,
$x \in \mathbf{Var}$ — for variable,
$e \in \mathbf{Expr}$ — for arithmetic expressions,
$b \in \mathbf{Bexp}$ — for Boolean expressions,
$S \in \mathbf{Statm}$ — for statements.

Production rules:

$$e ::= n \mid x \mid e + e \mid e \ - \ e \mid e \ * \ e \mid (e),$$

$$b ::= \texttt{true} \mid \texttt{false} \mid e \ = \ e \mid e \ \leq \ e \mid \neg\, b \mid b \ \wedge \ b \mid (b),$$

$$S ::= x := e \mid \texttt{skip} \mid S;\ S \mid \texttt{if}\ b\ \texttt{then}\ S\ \texttt{else}\ S \mid \texttt{while}\ b\ \texttt{do}\ S.$$

Semantics of arithmetic expressions

Semantics of arithmetic expressions:

- is defined only for untyped expressions here,
- this allows us to define semantics of arithmetic expressions by **uniform** way for **different** methods of semantics.

Semantic domains:

- meaning of each arithmetic expression is its **value**, in our language it is integer, so we define **semantic domain** $\mathbf{Z}$ of integers,
- the meaning of an expression depends on the values bound to the variables that occur in it. We shall therefore introduce the concept of a (memory) **state**.

Semantics of arithmetic expressions

We define **semantic domain of states** $\mathbf{State}$, where the elements are **states** s:

$$s \in \mathbf{State}.$$

We shall represent a state as a function from variables to values:

$$s : \mathbf{Var} \to \mathbf{Z},$$

which assigns to each variable occurring in an expression an exact value from semantic domain $\mathbf{Z}$.

Semantic domain $\mathbf{State}$ is a set of all functions from the set $\mathbf{Var}$ to the set $\mathbf{Z}$.

We call this set also **function space**:

$$\mathbf{State} = \mathbf{Var} \to \mathbf{Z}.$$

Semantics of arithmetic expressions

State s is a function which provides value for variable x

$$s\ x \in \mathbf{Z}.$$

When variables x and y occur in an expression and their values are $\mathbf{3}$ and $\mathbf{5}$, resp., the state can be expressed as a list:

$$s = [x \mapsto \mathbf{3}, y \mapsto \mathbf{5}] \quad \text{or} \quad s\ x = \mathbf{3}, s\ y = \mathbf{5}.$$

State is an **abstraction of computer memory** for the purpose of semantics.

Semantics of arithmetic expressions

Given an arithmetic expression e and a state s, we can determine the value of the expression. Therefore we shall define the meaning of arithmetic expressions as a total function $\mathscr{E}$:

$$\mathscr{E} : \mathbf{Expr} \to \mathbf{State} \to \mathbf{Z}.$$

Function is written in **Curry style**.

Function $\mathscr{E}$ takes two arguments:

- the syntactic construct (an element of $\mathbf{Expr}$), and
- the state, an element of $\mathbf{State}$.

Haskell Curry (1900-1982)

Semantics of arithmetic expressions

Semantic function

$$\mathscr{E} : \mathbf{Expr} \to \mathbf{State} \to \mathbf{Z}$$

is a function of two arguments. It takes its parameters one at a time.

1. We may supply $\mathscr{E}$ with its first parameter, for instance $x + y - 5$, and study the function

   $$\mathscr{E}[\![\, x + y - 5 \,]\!] : \mathbf{State} \to \mathbf{Z}.$$

   Syntactic constructs are always enclosed in semantic brackets.

2. When we supply the function $\mathscr{E}[\![\, x + y - 5 \,]\!]$ with a state s, we obtain the value of the expression $x + y - 5$:

   $$\mathscr{E}[\![\, x + y - 5 \,]\!]\, s \in \mathbf{Z}$$

Here s is the second argument of the function, not an index!

Semantics of arithmetic expressions

The semantics of arithmetic expressions is defined on each arithmetic expression:

$$
\begin{array}{lcl}
\mathscr{E}[\![\, n \,]\!]s & = & \mathscr{N}[\![\, n \,]\!] \\
\mathscr{E}[\![\, x \,]\!]s & = & s\ x \\
\mathscr{E}[\![\, e_1 + e_2 \,]\!]s & = & \mathscr{E}[\![\, e_1 \,]\!]s \oplus \mathscr{E}[\![\, e_2 \,]\!]s \\
\mathscr{E}[\![\, e_1 * e_2 \,]\!]s & = & \mathscr{E}[\![\, e_1 \,]\!]s \otimes \mathscr{E}[\![\, e_2 \,]\!]s \\
\mathscr{E}[\![\, e_1 - e_2 \,]\!]s & = & \mathscr{E}[\![\, e_1 \,]\!]s \ominus \mathscr{E}[\![\, e_2 \,]\!]s \\
\mathscr{E}[\![\, (e) \,]\!]s & = & (\mathscr{E}[\![\, e \,]\!]s)
\end{array}
$$

Here s is an input state, i.e. an input argument for semantic function. After evaluation its value is **unchanged**.

Semantics of arithmetic expressions: Example

Let $x + (y - 5)$ be an arithmetic expression and suppose $s = [x \mapsto \mathbf{2}, y \mapsto \mathbf{10}]$.

An expression is of the form $e + e$, so we start with an application of the third semantic equation:

$$
\begin{aligned}
\mathscr{E}[\![\, x + (y - 5) \,]\!]\; s &= \mathscr{E}[\![\, x \,]\!]\; s \oplus \mathscr{E}[\![\, (y-5) \,]\!]\; s \\
&= s\; x \oplus (\mathscr{E}[\![\, y \,]\!]\; s \ominus \mathscr{E}[\![\, 5 \,]\!]\; s) \\
&= s\; x \oplus (s\; y \ominus \mathscr{N}[\![\, 5 \,]\!]) \\
&= \mathbf{2} \oplus (\mathbf{10} \ominus \mathbf{5}) \\
&= \mathbf{7}
\end{aligned}
$$

$\square$

Problem as motivation. Suppose we add the arithmetic expression $-e$ to our language. Define its semantics!

Semantics of Boolean expressions

Meaning of Boolean expression is a truth value. Semantic domain of truth values is a set:

$$\mathbf{B} = \{\mathbf{tt}, \mathbf{ff}\}$$

where

- $\mathbf{tt}$ is used for true,
- $\mathbf{ff}$ is used for false.

The denotations `true` and `false` are considered as syntactic elements, not truth values!

We define (total) semantic function

$$\mathscr{B} : \mathbf{Bexp} \to \mathbf{State} \to \mathbf{B}.$$

Semantics of Boolean expressions

We define semantic clauses as follows:

$$
\begin{aligned}
\mathscr{B}[\![\, \texttt{true} \,]\!]\ s &= \mathbf{tt}, \\
\mathscr{B}[\![\, \texttt{false} \,]\!]\ s &= \mathbf{ff}, \\
\mathscr{B}[\![\, e_1 = e_2 \,]\!]\ s &= \begin{cases} \mathbf{tt}, & \text{if } \mathscr{E}[\![\, e_1 \,]\!]\ s = \mathscr{E}[\![\, e_2 \,]\!]\ s, \\ \mathbf{ff}, & \text{if } \mathscr{E}[\![\, e_1 \,]\!]\ s \neq \mathscr{E}[\![\, e_2 \,]\!]\ s, \end{cases} \\
\mathscr{B}[\![\, e_1 \leq e_2 \,]\!]\ s &= \begin{cases} \mathbf{tt}, & \text{if } \mathscr{E}[\![\, e_1 \,]\!]\ s \leq \mathscr{E}[\![\, e_2 \,]\!]\ s, \\ \mathbf{ff}, & \text{if } \mathscr{E}[\![\, e_1 \,]\!]\ s > \mathscr{E}[\![\, e_2 \,]\!]\ s, \end{cases} \\
\mathscr{B}[\![\, \neg b \,]\!]\ s &= \begin{cases} \mathbf{tt}, & \text{if } \mathscr{B}[\![\, b \,]\!]\ s = \mathbf{ff}, \\ \mathbf{ff}, & \text{if } \mathscr{B}[\![\, b \,]\!]\ s = \mathbf{tt}, \end{cases} \\
\mathscr{B}[\![\, b_1 \wedge b_2 \,]\!]\ s &= \begin{cases} \mathbf{tt}, & \text{if } \mathscr{B}[\![\, b_1 \,]\!]\ s = \mathbf{tt} \text{ and } \mathscr{B}[\![\, b_2 \,]\!]\ s = \mathbf{tt}, \\ \mathbf{ff}, & \text{if } \mathscr{B}[\![\, b_1 \,]\!]\ s = \mathbf{ff} \text{ or } \mathscr{B}[\![\, b_2 \,]\!]\ s = \mathbf{ff}, \end{cases} \\
\mathscr{B}[\![\, (b) \,]\!]\ s &= (\mathscr{B}[\![\, b \,]\!]\ s)\,.
\end{aligned}
$$

Semantics of Boolean expressions: Example

We find a meaning of an expression $\neg(x + y \leq 10)$. We suppose $s = [x \mapsto \mathbf{2}, y \mapsto \mathbf{1}]$.

Inner expression is of the form $e = e$, the outermost one of the form $\neg b$. Firstly, we determine $x + y$ in the state s.

$$\begin{aligned}
\mathscr{E}[\![\, x + y \,]\!]\; s \quad &= \mathscr{E}[\![\, x \,]\!]\; s \oplus \mathscr{E}[\![\, y \,]\!]\; s = s\; x + s\; y = \mathbf{3}, \\
\mathscr{E}[\![\, 10 \,]\!]\; s \quad &= \mathscr{N}[\![\, 10 \,]\!] = \mathbf{10}, \\
\mathscr{B}[\![\, x + y \leq 10 \,]\!]\; s \quad &= \mathscr{E}[\![\, x + y \,]\!]\; s \leq \mathscr{E}[\![\, 10 \,]\!]\; s = \\
&= \mathbf{3} \leq \mathbf{10} \\
&= \mathbf{tt}.
\end{aligned}$$

It holds that $\mathbf{3} \leq \mathbf{10}$, so it follows that its negation is false:

$$\mathscr{B}[\![\, \neg\, (x + y \leq 10) \,]\!]\; s = \mathbf{ff}.$$

$\square$

Semantics of expressions

When working with arithmetic and Boolean expressions, we need two more concepts:

1. a meaning of expression depends **only** on values of variables that occur in it. The **free variables** of an expression is defined to be the set of variables occurring in it. Formally, we may give a compositional definition of subsets $FV(e)$ of $\mathbf{Var}$. We may define the set $FV(e)$:

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

Lemma: Let s and s' be two states satisfying that

$$s\ x = \ s'\ x$$

for all $x \in FV(e)$ in an arithmetic expression e. Then

$$\mathscr{E}[\![e]\!]\ s = \mathscr{E}[\![e]\!]\ s'$$

Proof: using structural induction on the arithmetic expression. (*Homework*).

Substitutions

2. an occurence of a variable in an arithmetic expression can be replaced with another arithmetic expression.

   Substitution is used also for **state actualisation**.

   Change the initial state s to the new state s_0 is denoted as follows:

$$s' = s[y \mapsto \mathbf{a}]$$

   which means that new state s_0 is a state s except that the value bound to y is $\mathbf{a} \in \mathbf{Z}$. Formally:

$$s'\ x = \ (s[y \mapsto \mathbf{a}])\ x = \begin{cases} \mathbf{a} & \text{if} \quad x = \ y, \\ s\ x & \text{if} \quad x \neq \ y. \end{cases}$$