

Principles of Programming Languages

Semantics of Programming Languages and Types

Table of Contents

Operational Semantics of Programming Languages

Expressions vs. Values

Evaluation of Expressions into Values

Types in Programming Languages

Types with TypeScript

Operational Semantics of Programming Languages

Semantics is the study of the meaning of languages. The word *meaning* is complex and we must specify what is meant by *meaning of a programming language*.

A computer language is specified by two main components:

- a (formal) syntax – describing the structure of programs as they are written by programmers.
- semantics – which describe the intended meaning of programs.

Various tools manipulate programs (compilers, interpreters, debuggers, IDEs, verification tools).

All of these tools must agree upon a formal specification of what is expected when we execute the program. The formal semantics of a programming language provides an unambiguous definition of what the execution should achieve.

- People learning the language can understand what each of the programming language construct is expected to achieve.

- Implementers of tools for the language (parsers, compilers, interpreters, debuggers, etc) have a formal reference for their tool and a formal definition of its correctness/completeness.

- Two different programs in the language can be proved formally as equivalent/non-equivalent.

Formal semantics gives rules for translation from one domain (usually the program's syntax) to another formally defined domain.

We adopt the **Operational Semantics** approach which determines that: the meaning of an expression in the programming language is specified by the computation it induces when it is executed on a machine.

The operational semantics of a programming language tells us how to execute the program step-by-step. When we follow this specification, we record the steps and keep track of the **state** of the computation as steps are executed.

The specification determines what is meant by the state - in a first approximation, it can be the list of defined variables and their value at each step.

The result of the execution of a program according to the semantic specification is a derivation sequence which represents the computation history. Each step in the derivation indicates which part of the program was executed, and the state that was reached as a consequence.

Take as example an imperative programming language with global variables and assignment to variables. The state of the program holds the list of defined variables and their values. The program is a sequence of assignments:

Semantic Domain

Initial state: [x:5, y:3, z:7]

Program: "z = x; x = y; y = z;"

Step 1: execute "z=x;"

Remaining program: "x=y; y=z;"

New state: [x:5; y:3, z:5]

Step 2: execute "x=y;"

Remaining program: "y=z;"

New state: [x:3, y:3, z:5]

Step 3: execute "y=z;"

Remaining program: ""

New state: [x:3, y:5, z:5]

This result is the *history of the computation* - it traces the steps of the execution into primitive steps and the successive states of the computation.

Note that it is an **abstraction** - it does not provide all the details of what should happen in a concrete computation on specific hardware. For example, it does not mention registers, translation to machine language, encoding of data types. This computation history is a formal mathematical object which is in the **semantic domain**.

The rules of the semantics of the specific programming language indicate how to select a sub-expression to evaluate at each step of the computation, and what are the effects on the state of the computation each time a primitive sub-expression is executed.

In summary, the operational semantics of the language maps a program and an initial state to a formal structure - we informally refer to this structure as a computation history.

Operational Semantics of Programming Languages

Expressions vs. Values

In **imperative** programs, programs are constructed from **statements**. A statement is a unit of program execution.

- **Atomic statements** such as variable assignment or print-statements have a single effect either on the world (screen is updated, disk is updated, information is sent in the network) or on the internal state of the program.

- **Compound statements** are built from multiple sub-statements. For example, an **if-then-else** construct is used to build a compound statement, or a **for-loop** construct.

When a statement is executed, the state of the program is modified.

In contrast, in **Functional Programming (FP)**,
programs are **expressions**.

- **Atomic expressions** - for example, the number expression **-12** or the boolean expression **true** or the variable expression **x**.

- **Compound expressions** - which are made up of sub-expressions according to the syntax of the language. For example the expression `12 >= 7` is a compound expression made up of 3 sub-expressions, or `(12 === 13) ? -1 : 2` is also a compound expression (called a conditional expression).

In FP, we do not execute an expression, but instead we compute its value - a process we call **evaluation**. The *evaluation function* maps expressions to values.

The operational semantics of an FP language describes how the evaluation function operates over all possible expressions in the language. It is defined inductively over the syntactic structure of expressions.

In the rest of the course, we will focus on specifying the operational semantics of functional languages.

The JavaScript language mixes both expressions and statements - it is a multi-paradigm language. This makes it difficult to describe completely its operational semantics in a concise manner. We will completely define a smaller language in Chapter 2 - and only provide an informal description for JavaScript.

- Atomic expressions:
number expressions (`1`, `-12`),
string expressions (`"abc"`),
boolean expressions (`true`, `false`).

- **Compound expressions:**

arithmetic expressions

`(10 + v) * (w / 5),`

relational expressions `(x === 5),`

ternary conditional expression

`<condition> ? <expr1> : <expr2>.`

- **Variable bindings:** these are expressions of the form

`let <var> = <expr1>; <expr2>`

which define a block in which the variable `<var>` is defined and bound to the value of `<expr1>` and then evaluate the value of `<expr2>`.

- Function invocation: of the form $f(\langle \text{expr1} \rangle, \dots, \langle \text{exprn} \rangle)$.

- Function definition: of the form
 $(\langle \text{var1} \rangle, \dots) \Rightarrow \langle \text{expr} \rangle.$

The list of these expression types and their structure defines the *syntax of the programming language*. Note that we did not include in this partial description the syntax of statements in JavaScript.

Expressions are combined recursively to form trees of compound expressions with atomic expressions at the leaves. For example:

```
let v = 12;  
    (v >= 7) ? (v * 3) : 9
```

is a compound expression of type **let**, with 2 sub-expressions - a binding expression **v = 12** and a body expression which is a conditional ternary expression; itself this sub-expression has 3 sub-expressions.

Once an expression is evaluated by applying the *evaluation function*, it becomes a **value**. The *evaluation function* maps expressions to values.

As we can distinguish types of expressions, we can also characterize the set of all possible values in an inductive manner - that is, start from **atomic values** and then build up **compound values** which are made up of smaller value parts.

This lecture focuses on the characterization of the **domain of values** in JavaScript.

Operational Semantics of Programming Languages

Evaluation of Expressions into Values

The operational semantics of an FP language
define the evaluation function.

It determines how a top-level expression (the program) is evaluated into a value. This definition is inductive. For each syntactic construct, an evaluation rule indicates how the sub-expressions are evaluated - in which order.

At each step of the computation, a sub-expression is evaluated and then substituted by its value in the embedding expression.

There may be a need to describe the state of the computation during this evaluation process

- similar to the set of bindings **var:value** we observed above when describing the operational semantics of a very simple imperative language.

The operations on this state will be quite different from what we observed, as in pure FP languages there is no need to perform variable assignment (because variables are immutable).

The output of the operational semantics is a computation history - but in this chapter we will not describe the computation history - we will only describe the values produced by the evaluation function.

Consider the complex expression we presented above:

```
let v = 12;  
(v >= 7) ? (v * 3) : 9
```

The evaluation algorithm consists of the following steps: given an expression E :

```
let v = 12;  
  (v >= 7) ? (v * 3) : 9
```

1. Identify the toplevel syntactic construct of E -
in our case a **let** construct.

```
let v = 12;  
  (v >= 7) ? (v * 3) : 9
```

2. Identify the immediate sub-expressions of E
 - in our case - the binding construct $v = 12$
and the body (rest of E).

```
let v = 12;  
(v >= 7) ? (v * 3) : 9
```

3. Perform the specific *evaluation rule* defined for the construct of E - in this case - the *let-evaluation rule* is:

Evaluation Algorithm

- A. Compute the value of the `<expr>` on the right-hand-side of the binding expression - call it `v1`.
- B. Add to the state of the computation the binding `{v:v1}`.
- C. Compute the value of the body sub-expression - call it `v2`.
- D. Remove the binding `{v:v1}` from the state of the computation.
- E. The value of the overall `let` construct is `v2`.

This definition of the evaluation function is **recursive**: in steps 3.A and 3.C - we invoked the evaluation function recursively on sub-expressions of E .

The evaluation function refers to a *state* which describes the set of known variables at every single step of the computation, and the values to which they are bound.

This recursive definition must have a base case to terminate. The base case of the recursion consists of evaluating **atomic expressions** or to invoke **atomic primitive operators or functions**.

For example:

- The expression v is an atomic expression (it has no sub-expressions); it is an expression of type *variable*. The computation rule for evaluating a variable is to lookup its value in the current state of the computation. In our case, the value of the expression v is **12**.

- The expression **7** is an atomic expression of type **number**; The computation rule for evaluating a number is to convert the way numbers are written in the language into a number value.

- The expression (`v >= 7`) is a compound expression of type **relational operator** with a primitive atomic operator (`>=`); The computation rule for evaluating a relational operator is to (1) evaluate the left side sub-expression (we obtain **12**); (2) evaluate the right-side sub-expression (we obtain **7**); (3) pass the 2 values to the primitive comparison operator which must return a boolean value (in this case **true**).

A useful way to think about the evaluation of complex expressions is in terms of **reduction**.

We want to evaluate the following expression, which is a formula for computing the wage of a worker given the number of hours she worked:

```
let hours = 45;  
  (hours <= 40) ? hours * 10 :  
  (hours > 40) ? (40 * 10) + ((hours - 40) * 15) :  
  0
```

This formula in a more explicit form states that:

```
if (hours <= 40)
    hours * 10
else if (hours > 40)
    (40 * 10) + ((hours - 40) * 15)
else
    0    // This should not happen.
```

Let's now see how this expression is reduced to a value, using a step-by-step process:

The first step is to **substitute the hours variable** with 45:

$(45 \leq 40) ? 45 * 10 :$

$(45 > 40) ? (40 * 10) + ((45 - 40) * 15) :$

0

Next, the conditional part of the ternary expression is evaluated, which in this case is false.

```
false ? 45 * 10 :  
(45 > 40) ? (40 * 10) + ((45 - 40) * 15) :  
0
```

Since the condition is false, the next branch is tried. Note that at each step, a sub-expression is evaluated and replaced by its value:

false ? 45 * 10 :

true ? (40 * 10) + ((45 - 40) * 15) :

0

Since the condition is true, the expression reduces to the body of that branch. After that, it's just arithmetic (but each step is done in a specific order):

$$\Rightarrow (40 * 10) + ((45 - 40) * 15)$$

$$\Rightarrow 400 + (5 * 15)$$

$$\Rightarrow 475$$

This style of reduction is the best way to think about the evaluation of complex expressions. It is called a process of **reduction** - because the complex expression is incrementally reduced into a value.

Types in Programming Languages

In the same way as expressions can be atomic or compound and built-up into complex recursive programs, the values that are generated by programs can be:

- **atomic values:** values which have no “sub-components” - such as numbers or booleans.
- **compound values:** values which are constructed from sub-values, such as arrays for example.

Compound values can be created in the following manner:

- Constant literal values, *e.g.*, `[1, 2, 3]` denotes an array in JavaScript.
- Computed by primitive constructor functions, *e.g.*, `[1].concat([12])` returns a new array `[1, 12]`.
- Deserialized from strings that represent compound values according to a value syntax.

NOTE 1: the capability to specify compound values as literal constants is a characteristic of dynamic languages such as JavaScript, Scheme or Python. It is much more difficult to *write down* compound values in languages such as Java or C++. We will return to this important distinction later.

NOTE 2: the distinction that a value is atomic depends on a language design decision. For example, strings in JavaScript are considered atomic, while in C or C++ they are compound (an array of characters).

As you have noted, JavaScript is an **untyped language** - this means: when we declare variables (using the **let** construct) or functions (with the **function** or **=>** constructs), we do not specify the type of the variables or parameters. We also do not need to specify the type returned by the function.

This is in contrast to **typed languages** such as Java and C++ - which require the programmer to specify the type of all variables before they can be defined or used.

Yet - even if **variables** are not declared with a type in JavaScript, **values** in JavaScript **do have** a type - they can be either **number** or **string** or **boolean** or compound values.

Why would we want or not want to declare the type of variables?

What are the benefits of each of the approaches?

In the description of the evaluation function above, we indicated that expressions are traversed recursively, until atomic expressions are evaluated (into numbers, booleans or strings) and primitive operators or functions are invoked (for example `(v >= 7)`).

When primitive operators or functions are invoked - we can obtain errors at runtime because the value which is passed to the operator does not fit the type of the primitive operator.

For example, in most languages, the evaluation of `(7 >= "a")` or `(8 + "b")` would return an error at runtime because the operators `>=` and `+` do not know how to operate on a mixture of numbers and strings. This will be the case in Scheme for example as we will see in Chapter 2.

JavaScript primitives do not fail.

In JavaScript, the language designers took a different approach: they made the primitive operators extremely flexible and robust. So that evaluations of strange expressions do not trigger a runtime error - but instead do either automatic conversions or return special values indicating an impossible value.

Typing Errors at Runtime

```
"a" > 2          // => false  
2 + "ab"        // => "2ab"  
"a" * 2         // => NaN  
"a" && true      // => true
```

This is a dubious decision - as such automatic handling of unexpected variations is most often a sign of poorly written code and produces *surprising* results.

Accessors to Compound Values in JavaScript do not Fail.

Another decision of the JavaScript designers was to make access to compound values “robust” against runtime errors as well:

- Accessing an index out of bounds in an array returns the special value **undefined**
- Accessing an **undefined** key in a map returns the special value undefined as well.

Typing Errors at Runtime

```
let arr = [1];  
arr[4]; // => undefined  
let map = { a: 1, b: 2 };  
map.c; // => undefined
```

Undefined Variables in JavaScript do Fail.

Variables access, however, **can fail**: trying to access an undefined variable raises an error at runtime. When variable expressions are evaluated, it may turn out that the variable is not defined in the current state of the program at the time its value is looked up.

Typing Errors at Runtime

```
let b = 2;  
c; // => ReferenceError: c is not defined
```

Similarly, trying to access a key from an `undefined` value triggers an error at runtime:

`e.k; // => ReferenceError: e is not defined`

Now that we have seen which errors can be triggered when executing a JavaScript program - and which *surprising* results may be returned when we combine unexpected value types in primitive operators invocations - we return to our question:

- Can we avoid such errors and surprises?
- What are the benefits of declaring the types of variables?

Type safety is the extent to which a programming language discourages or prevents type errors. A type error is erroneous or undesirable program behaviour caused by a discrepancy between differing data types for the program's constants, variables, and methods (functions), e.g., treating an integer (int) as a floating-point number (float).

Type enforcement can be either:

- **static**, catching potential errors at compile time
- **dynamic**, associating type information with values at run-time and consulting them as needed to detect imminent errors
- a combination of both

In the context of static (compile-time) type systems, type safety usually involves a guarantee that the eventual value of any expression will be a legitimate member of that expression's static type.

We will develop the notion of type safety incrementally, after we discuss what are types.

A data type is a classification of data which indicates what the programmer intends to do with the data. Data types are defined along 2 aspects:

- Types correspond to sets of values - for example, the type **Boolean** is the set of values $\{true, false\}$; the type **Number** is the (possibly infinite) set of numeric values.

- Types define the operations that can be done on data values in the set. For example, values of type **Boolean** can be computed using logical operators such as **and**, **or**; **Number** values can be manipulated with operators **+**, ***** and compared with operators such as **<**, **>**.

Types also determine how the interpreter reads and writes the values, and how it stores the values in memory.

When discussing types, it is important to distinguish the **type of a value** and the **type of a variable**:

- **Values** always have a type - that is, they belong to a specific set of values. The number **3** belongs to the set of **number** values; the value **true** belongs to the set of **boolean** values.

- **Variables** are parts of the programming language expressions. They are bound to values as the expressions are computed at runtime. When a programmer declares that a *variable has a type* - the meaning is that we express the intent of the programmer that this variable can **only be bound to values of this type** during the lifecycle of the variable. It states a constraint on the variable usage.

In most typed languages, this constraint can be checked at compile time: this is an extremely strong result. It means that the compiler can at compile time guarantee that all possible executions of the program, with all possible input values will satisfy the stated constraint. This static verification (that is, a verification performed at compile-time - without knowing the value of the variables) is called **type checking**.

As mentioned, a value always has a type - this simply states that a value belongs to a set of values. For example, the value **1** belongs to the set of values **number** - which means that the value **1** has type **number**.

In most cases, a value will have more than one type: consider for example the case of *sub-types* such as **Integer** and **Number**. **Integer** as a set of values is a proper subset of **Number** as a set of values. This means in particular that a value like **6** belongs both to the type **Integer** and to the type **Number**.

A good type system helps programmers not only declare the intended types of variables to obtain type safety, it also helps the programmer **design the range of values** the program will process and produce. This is a constructive process which helps structure the set of all possible values in meaningful ways and also document the domain upon which programs operate.

As we write more complex programs over more complex data structures - we will feel more strongly the benefit of describing precise data types - which will also guide the structure of the code we write to operate over them.

Since types denote set of values, we can define relations among types that are similar to set relations:

Types and Set Relations

- Type T1 can be a subset of type T2: this means any value of type T1 is also a value of type T2. We also say that T1 is a subtype of T2.
- Types T1 and T2 can be disjoint - meaning there are no values that are both of type T1 and T2. For example, **number** and **boolean** are disjoint.

- One can define a universal type - which is the type of all possible values. (This type exists in TypeScript and is called **any**).
- Some types denote finite sets (**boolean** has 2 values), others are infinite (**numbers**, **strings**).

One can construct new types on the basis of existing types:

- The union of T_1 and T_2 contains all the values of T_1 and T_2 together.
- The intersection of T_1 and T_2 .
- The cartesian product of T_1 and T_2 would be the type that contains pairs of values (v_1, v_2) such that v_1 is of type T_1 and v_2 is of type T_2 .

This last construction opens the door to compound data types - that is, types which contains values that have sub-components.

We will see later that programmers can define new types (user-defined types) on the basis of primitive types (types provided by the language) by using set theory operations like Union, Intersection or Cartesian Product.

Different languages offer various levels of introspection (often called reflection) to enable the analysis of the type of values at runtime, or at interpretation/compile time.

In JavaScript (the underlying language into which TypeScript is translated), variables are not typed, values have a type (which is encoded in the binary representation of the values in memory) and the **typeof** primitive operator can be used to inspect the type of values at runtime. Primitives in JavaScript use this introspection mechanism to decide how to operate on each combination of value types they receive as parameters.

In contrast, in Java and C/C++, primitive values (non-object values such as integers, booleans, characters, pointers and references) cannot be inspected at runtime: values of different types (for example a char, a boolean or an integer) are encoded in binary format in RAM without any distinctive sign that could be used at runtime to determine that the value belongs to different types.

Types with TypeScript

We now describe the basic data types manipulated in TypeScript.

TypeScript is a language built on top of JavaScript: it adds optional type declarations to variables (which do not exist in JavaScript). The TypeScript compiler translates TypeScript into JavaScript. During the translation, the TypeScript compiler performs type checking - which we will explain later.

Simple values can have the following primitive data types in TypeScript:

- Boolean: `true`, `false`
- Number: floating point numbers; `6`, `6.1`, `-4`, `0xf0a`
- String: immutable sequences of characters, `"this is a string"`

Atomic Primitive Value Types

```
console.log(typeof 6)    // => number  
console.log(typeof true) // => boolean  
console.log(typeof "a")  // => string
```

There are two special value types that are not useful on their own - but will play an important role when we start building more complex types:

- **null**: this is a type that contains only the special value `null`
- **undefined**: this is a type that contains only the special value `undefined`

`undefined` plays a special role in the lifecycle of variables. We will return to this when we discuss the types of variables (as opposed to the type of values which are we now reviewing).

Atomic Primitive Value Types

```
console.log(typeof null)      // => object  
console.log(typeof undefined) // => undefined
```

In contrast to atomic values, one can also define *compound values* - which are values that are built from multiple parts. There are two basic compound types in TypeScript:

- Arrays – `[1, 2, 3]`
- Maps – `{ name: "Ben", age: 31 }`

Any value that is not an atomic value is a *compound value* - which is called in JavaScript an **object** value.

Compound Value Types

```
console.log(typeof [1, 2, 3])    // => object  
console.log(typeof {a: 1, b: 2}) // => object
```

Notice that the primitive **typeof** operator is limited in its capability to distinguish the types of values, even if they are very different. We will see later that TypeScript improves on this operator tremendously.

More specific reflection operators allow finer distinction in the types of values, for example:

Compound Value Types

```
console.log([1, 2, 3] instanceof Array)
// => true
console.log({a: 1, b: 2} instanceof Array)
// => false
```

Compound values can be entered as constant literals (as above) or can be constructed by invoking the appropriate constructors and mutators (functions which incrementally change a value). For example:

Compound Value Types

```
Array(1, 2, 3) // => [1, 2, 3]
```

```
let map = {};  
map.a = 1;  
map.b = 2;  
console.log(map); // => { a: 1, b: 2 }
```

Compound values can be “put apart” by using getters - for arrays, using indexes which refer to the positions of the items within the array, and for maps using key values. In addition, the `slice()` method of arrays can return a range of values from within the array:

Compound Value Getters

```
let arr = ["a", "b", "c", "d", "e"];  
console.log(arr[0]);  
// => "a"  
console.log(arr.slice(1));  
// => ["b", "c", "d", "e"]  
console.log(arr.slice(1, 4));  
// => ["b", "c", "d"]
```

Maps are collections of pairs (key, value).

Map getters use the key to access the value.

Alternatively, the dot notation `m.k` can be used when the key is a string without spaces.

Compound Value Getters

```
let m = { a: 1, b: 2 };  
console.log(m["a"]); // => 1  
console.log(m.a);    // => 1
```

The method `Object.keys(x)` returns the list of the keys which are defined for any compound values. It operates both on arrays (the keys are the indexes) and on maps.

Compound Value Getters

```
let arr = ["a", "b", "c"];  
let map = { a: 1, b: 2 };  
console.log(Object.keys(arr));  
// => ["0", "1", "2"]  
console.log(Object.keys(map));  
// => ["a", "b"]
```

Alternatively, one can use the generalized **for** loops over arrays and maps:

Compound Value Getters

```
for (let k in map) {  
  console.log(`${k} has value ${map[k]}`);  
}
```

```
for (let k in arr) {  
  console.log(`${k} has value ${arr[k]}`);  
}
```

```
/* a has value 1  
   b has value 2  
   0 has value a  
   1 has value b  
   2 has value c */
```

Variable Types in TypeScript: Gradual Typing

In *static languages* like C++ and Java, variables must be typed - that is, when a variable is defined, its type must be declared by the programmer.

In *dynamically typed languages* like JavaScript and Scheme, variables are not typed. But when a variable is bound to a value, we can inspect its type at runtime (because values are always typed).

Variable Types in TypeScript: Gradual Typing

TypeScript extends JavaScript and introduces optional variable types. TypeScript is compiled into JavaScript - and at compilation time, type checking is performed. There is no additional type checking happening at runtime after compilation has completed.

Typing annotations can be introduced gradually as code matures.

Variable Types in TypeScript: Gradual Typing

Typing a variable means that the programmer declares how she intends to use the variable - which values it can be bound to. In TypeScript, this is performed by adding a type annotation to variable declarations:

```
let n: number = 6;  
let s: string = "hello";  
let b: boolean = true;
```

In general, programming languages provide ways to specify values as part of expressions.

Literal expressions are the syntactic representation of values that appear as part of expressions.

Examples of literal expressions in TypeScript:

1

"hello"

[1, 2, 3]

{ a: 1, b: 2 }

[{ c: 3 }, { d: 4 }]

Distinguish between **literal expressions** and the **values** they represent:

- **Literal expressions** are part of the syntax of a programming language; they are evaluated.
- **Values** are the result of the evaluation; they are part of the semantics of the language.