JavaScript Semantic Typesystem

Dr. Stefan Bosse
Version 2018-08-23

Contents

1. JavaScript Core Types

The JavaScript core type systems consists only of six basic types and types classes: numbers (usually referred to real type numbers), Boolean values, strings (that are immutable), arrays (that can be objects, too), objects, and functions. That's all.

2. JavaScript Objects

2.1. Synopsis

```
new Object()
{p1:..,p2:..,..}
```

```
o.p
```

2.2. Description

Objects in JavaScript are a kind of two-faced. From one side, an object is an associative array (called hash in some languages). It stores key-value pairs. From the other side, objects are used for object-oriented programming, and that's a different story. In this section we start from the first side and then go on to the second one. An object consists of named properties and methods, i.e., an object is an aggregation of variables and functions. A pure record object only have property variables.

2.3. Creating objects

An empty object (you may also read as empty associative array) is created with one of two syntax forms:

```
o = new Object()
o = { } // the same
```

It stores any values by key which you can assign or delete using 'dot notation':

```
var obj = {} // create empty object (associative array)

obj.name = 'John' // add entry with key 'name' and value 'John'

// Now you have an associative array with single element
// key:'name', value: 'John'

alert(obj.name) // get value by key 'name'

delete obj.name // delete value by key 'name'

// Now we have an empty object once again
```

You can use square brackets instead of dot. The key is passed as a string:

```
var obj = {}

obj['name'] = 'John'

alert(obj['name'])

delete obj['name']
```

There's a significant difference between dot and square brackets. obj.prop returns the key named 'prop', obj[prop] returns the key named by value of prop.

```
var prop = 'name'

// returns same as obj['name'], which is same as obj.name
alert(obj[prop])
```

Literal syntax

You can also set values of multiple properties when creating an object, using a curly-bracketed list: { key1: value1, key2: value2, ... }. Here's an example.

Literal syntax is a more readable alternative to multiple assignments:

```
var menuSetup = {
  width: 300,
  height: 200,
  title: "Menu"
}

// same as:

var menuSetup = {}
menuSetup.width = 300
menuSetup.height = 200
menuSetup.title = 'Menu'
```

It is also possible to create nested objects:

```
var user = {
  name: "Rose",
  age: 25,
  size: {
    top: 90,
    middle: 60,
    bottom: 90
  }
}

alert( user.name ) // "Rose"
alert( user.size.top ) // 90
```

Non-existing properties, undefined

We can try to get any property from an object. There will be no error. But if the property does not exist, then undefined is returned:

```
var obj = {}

var value = obj.nonexistant

alert(value)
```

So it's easy to check whether a key exists in the object, just compare it against undefined:

```
if (obj.name !== undefined) { // strict(!) comparison
  alert(" I've got a name! ")
}
```

Checking if a key exists

A peculiar coder (I bet you are) might have a question. What if I assign a key to undefined explicitly? How to check if whether the object got such key?

```
var obj = { key: undefined }

alert( obj.key ) // undefined.. just if there were no key
```

Hopefully, there is a special "in" operator to check for keys. It?s syntax is "prop" in obj, like this:

```
var obj = { key: undefined }

alert("key" in obj) // true, key exists
alert("blabla" in obj) // false, no such key
```

In real life, usually null is used a ?no-value?, leaving undefined for something? truly undefined. Iterating over keys-values There is a special for..in syntax to list object properties:

```
for(key in obj) {
  ... obj[key] ...
}
```

The following example demonstrates it.

```
var menu = {
  width: 300,
  height: 200,
  title: "Menu"
};

for(var key in menu) {
  var val = menu[key];

  alert("Key: "+key+" value:"+val);
}
```

Note how it is possible to define a variable right inside for loop. Order of iteration In theory, the order of iteration over object properties is not guaranteed. In practice, there is a de-facto standard about it. IE < 9, Firefox, Safari always iterate in the order of definition. Opera, IE9, Chrome iterate in the order of definition for string keys. Numeric keys become sorted and go before string keys. Try the code below in different browsers.

```
var obj = {
  "name": "John",
  "1": 1,
  age: 25,
  "0": 0
}
obj[2] = 2 // add new numeric
obj.surname = 'Smith' // add new string

for(var key in obj) alert(key)
// 0, 1, 2, name, age, surname <- in Opera, IE9, Chrome
// name, 1, age, 0, 2, surname <- in IE<9, Firefox, Safari
```

Here ‘numeric keys’ are those which can be parsed as integers, so “1” is a numeric key. There is an issue about it in Chrome.

Object variables are references

A variable which is assigned to object actually keeps reference to it. That is, a variable stores kind-of pointer to real data. You can use the variable to change this data, this will affect all other references.

```
var user = { name: 'John' }; // user is reference to the object

var obj = user; // obj references same object

```

```
obj.name = 'Peter'; // change data in the object

alert(user.name); // now Peter
```

Same happens when you pass an object to function. The variable is a reference, not a value. Compare this:

```
function increment(val) {
  val++
}

val = 5
increment(val)

alert(val) // val is still 5
```

And this (now changing val in object):

```
var obj = { val: 5}

function increment(obj) {
  obj.val++
}
increment(obj)

alert(obj.val) // obj.val is now 6
```

The difference is because in first example variable val is changed, while in second example obj is not changed, but data which it references is modified instead.

Properties and methods

You can store anything in object. Not just simple values, but also functions.

```
var user = {
  name: "Guest",
  askName: function() {
    this.name = prompt("Your name?")
  },
  sayHi: function() {
    alert('Hi, my name is '+this.name)
  }
}
```

Calling methods

When you put a function into an object, you can call it as method:

```
var user = {
  name: "Guest",
  askName: function() {
    this.name = prompt("Your name?")
  },
  sayHi: function() {
    alert('Hi, my name is '+this.name)
  }
}

user.askName()
user.sayHi()
```

Note the this keyword inside askName and askName. When a function is called from the object, this becomes a reference to this object.

The constructor function, "new"

An object can be created literally, using obj = { ... } syntax. Another way of creating an object in JavaScript is to construct it by calling a function with new directive. A simple example:

```
function Animal(name) {
  this.name = name
  this.canWalk = true
}

var animal = new Animal("beastie")

alert(animal.name)
```

A function takes the following steps: Create this = {}. The function then runs and may change this, add properties, methods etc. The resulting this is returned. So, the function constructs an object by modifying this. The result in the example above:

```
animal = {
  name: "beastie",
  canWalk: true
}
```

Traditionally, all functions which are meant to create objects with new have uppercased first letter in the name.

3. JavaScript Semantic Types

3.1. Description

The JS Semantic Type System (JST) extends JS with type annotations. It provides an extension to the JS core type system, which only consists of a few basic types, functions, and objects. The JST notation cannot be processed by the JS run-time system and is therefore specified in a separate type interface file (with jsi extension) or embedded in JS comments. With an extended *esprima* parser that understands JST extensions it is possible to combine JST and JS code. The primary purpose of JST is API documentation, but the JST can be used for type checking and profiling using external tools or JS extensions.

Originally, there is no type signature for an JS object or function. Although there is JavaDoc, but it is limited to the JS type system. Commonly, a programming language covers only types that are immediately related to the language and run-time concepts, e.g., structures. But often there is a higher abstract type level composed of core types. One example is a sum type, supported only by a few languages directly. A sum type is composed of different types, i.e., structures or objects, that can be distinguished by pattern matching or type tags using names.

JST will not convert the dynamic typed JS language in a static typed language. It is intended to provide type constraints and hints, enabling software API specification and documentation, and to enable (lazy and tolerant) program checking.

3.2. Types and Type Composition

There are type definitions `type t = T` (equality) and type declarations of functions, objects, parameters, variables, `o : T` (type annotations). In JST, types can be composed, e.g., a number is array is typed by `number []`, an array of records is typed by `{} []`, and so on. Furthermore, types can be extended, e.g., a constructor function is typed by `constructor function`. Since JavaScript is a dynamic typed language (type assignments at run-time), type alternatives can be defined by `t1|t2|..` type composition.

Notation

```
@  : smybolic identifier, label, or wildcard place holder
$  : type macro
?  : optional,
.. : more (repition)
|  : alternative, type choice operator
{ }: structure object, set
```

A type macro is usually only a fragment of a type definition, e.g., `type t = { $a, .. , $b }` with `$a= x:number` and `$b = y:string`. The repetition `..` expresses more macro expansions `$a` can follow.

Core Types

boolean
  Boolean type {true,false}

number
  Floating point number

char
  Character (string of length 1)

string
  Text string

buffer
  Byte buffer

{}
  Any procedural structure object

object
  Any object-orientated object with methods

array, []
  Array

undefined
  The void type

*****
  Any type (polymorphic)

3.3. Modules

The current module is declared with the `module` statement. Imported (used) modules are declared with the `use` as statement, only importing the type interface, and using the `open` statement to import the code as well.

module $path/m$
use $path/m$ **as** M
open $path/m$

3.4. Type Annotation and Type-of

Any variable, function, function parameter, object, and array can be type annotated, i.e., a type casting that defines a type constraint. Additionally, the type annotation can be extended with a description text. Any function parameter, variable, or named expression can get a type annotation by using the `typeof` statement.

Definition 1. (*Type Annotation*)

$identifier\ :\ type\ [$ **is** $comment\ ]$
$\Leftrightarrow$
typeof $identifier\ =\ type;$

3.5. Function Type Signature

A function type signature consists of parameter type annotations and by specifying a return type of the function (if it is not a procedure).

Definition 2. (*Function Declaration with Type Signature*)

function $(\ p_1\ :\ typ_1,\ p_2\ :\ typ_2,\ ..\ ,\ @p_i,\ typ_j,\ ..)\ \rightarrow typ$

Definition 3. (*Function Definition with Type Annotation*)

function $(\ p_1\ :\ typ_1,\ p_2\ :\ typ_2,\ ..,\ p_i,\ ..)\ :\ typ\ \{$
  $body\ statements$
$\}$

Parameters in function declarations without a type specification must have a name preceded by the `@` symbol to distinguish from types. The return type can be preceded by an optional parameter name.

3.6. Record Types

In JavaScript, there is no distinction between pure procedural record (structure) types and object types with methods (prototypes). In JST, a record type represents an object that was created without a constructor function and do not have any methods. A record object consists of value attributes (although a value can be a function), only.

Sub-types of records can be declared by using type composition with a sub-set of attributes, i.e., `st {a,b,c,..}` will derive a sub-type of *st* with the attributes *a*, *b*, .., only.

In JST, each attribute can be assigned a type (core type, user type, or composed type). Attributes can be marked optional by `?` postfix annotation, or extendable (can be added at run-time) by + prefix annotation.

Record types can be defined anonymously in any type expression.

Definition 4. (*Record Type*)

type $st\ =\ \{$
  $e_1 : typ_1,$
  $e_2 : typ_2,$
  $..$
  $e_i,$
  $e_j : typ_j$ **is** $description,$
  $e_k? : typ_k,$
  $+\ e_n : typ_n,$
  $@name : typ_o,$
  $e_v\ =\ \epsilon[\![:\ typ_v\ ]\!]$
  $..$
$\}$

The object attribute names are e_x, their types typ_x, and ϵ is an expression (or value).

Attribute Flags

x?
Optional attribute

+x
Attribute that can be added at run-time

@x
Symbolic attribute name (wild-card place holder for any identifier)

..
: More attributes (with or without same type interface) follow

$x
: A macro substitution, commonly a choice list of different attribute-type elements.

x = ϵ $[\![$ **: typ** $]\!]$
: A record entry having a specific value (e.g., a type tag) with optional type annotation.

Examples

The following example shows a sketch of a type interface (a template). The types are not always fully specified leaving room for speculation and ambiguity and are primarily intended for an API declaration.

```
$pos = top:number | left:number | right:number | center:boolean
type info = { type='info', $pos?, .. , $geom?, .. ,
              label:string, value?:string, .. }
type widget = ..

type on = { click?:handler|string, onclick?:handler|string,
            check?:handler1, uncheck?:handler1,
            selected?:handlerd, change?:handler,
            show?:handlerp, hide?: handlerp }

$pageparam = next:string | prev:string | on: on {show,hide} |
             show:function | hide:function
$widget = @name : widget

typeof content = {
  pages : {
   main: { $widget, .. ,
           $pageparam, .. },
   @page2: { @widget, .. }, ..
  },
  static?: { @id:info }
}
```

3.7. Enumeration

An enumeration type is a sum type, but also declares or defines an additional object (a record type). In JS enumeration is performed usually by defining an object consisting of constant value properties. In JST, only the enumeration symbol names are listed. Basically, an enumeration type *e* denotes a set of symbols (sub-types) and an enumeration object *E* with properties (S_1,S_2,..). Hence, `enum` declares an object and defines a type! Beside constant symbols, symbol constructors with parameters can be defined. That is a seamless transition from enumerations to generic sum types. That means, an enumeration symbol can be a constant value (number or string, i.e., representing the tag) or a constructor function returning an object with a tag field (the symbol tag).

Definition 5. (*Enuemration Type*)

enum $E\ =\ \{$
  $S_1,\ //\ Symbol\ 1$
  $S_2,\ //\ Symbol\ 2$
  $S_3\ \{tag = "S_3", e_1, e_2, ..\}\ //\ Symbol\ Constructor$
  $S_4\ =\ \epsilon,\ //\ Concrete\ Symbol\ value\ assignment$
  $..$
$\}\ [\![:\ typ_e\ ]\!][\![(symtype\ ¦\ Tag)\ ]\!]$

typeof $E\ =\ e\ =\ S_1\,|\,S_2\,|\,..$
typeof $E\ =\ e\ =\ "E.S_1"\,|\,"E.S_2"\,|\,..$
function $S_3(e_1, e_2, ..)\ \rightarrow S_3$
$Anonymous\ Enumeration\ Type\ Set$
$\{\ S_1,\ S_2,\ ..\}$

Optionally, the symbol value type (*symtype*) and the enumeration tag (*ET*, string prefix) can be declared, too.

Examples

In the JS example below the enumeration symbols have the string type and the `E.` string prefix.

```
enum E = {S1,S2,..} : e (string)
⇔
var E = {
  S1:'E.S1',
  S2:'E.S2',
  ..
};
var x:e = E.S1;
```