Javascript Bignum Extensions

Version 2018-06-16

Author: Fabrice Bellard

Table of Contents

1 Introduction 1

2 Operator overloading 2

2.1 Introduction 2
2.2 Builtin Object changes 3
2.2.1 `Symbol` constructor 3

3 The BigInt Mode 4

3.1 Introduction 4
3.2 Changes that introduce incompatibilities with Javascript 4
3.2.1 Standard mode 4
3.2.2 Bigint mode 4
3.3 Operators 5
3.3.1 Arithmetic operators 5
3.3.2 Logical operators 5
3.3.3 Relational operators 6
3.4 Number literals 6
3.5 Builtin Object changes 6
3.5.1 `BigInt` function 6
3.5.2 `BigInt.prototype` 7
3.5.3 `Number` constructor 7
3.5.4 `Number.prototype` 7
3.5.5 `Math` object 7

4 Arbitrarily large floating point numbers 8

4.1 Introduction 8
4.2 Floating point rounding 8
4.3 Operators 8
4.4 BigFloat literals 8
4.5 Builtin Object changes 9
4.5.1 `BigFloat` function 9
4.5.2 `BigFloat.prototype` 10
4.5.3 `BigFloatEnv` constructor 10
4.5.4 `Math` object 12

5 Math mode 13

5.1 Introduction 13
5.2 Builtin Object changes 13
5.2.1 `Symbol` constructor 13
5.3 Remaining issues 13

1 Introduction

The Bignum extensions add the following features to the Javascript language while being 100% backward compatible:

- Overloading of the standard operators to support new types such as complex numbers, fractions or matrixes.
- Bigint mode where arbitrarily large integers are available by default (no `n` suffix is necessary as in the TC39 BigInt proposal[1]).
- Arbitrarily large floating point numbers (`BigFloat`) in base 2 using the IEEE 754 semantics.
- Optional `math` mode which modifies the semantics of the division, modulo and power operator. The division and power operator return a fraction with integer operands and the modulo operator is defined as the Euclidian remainder.

The extensions are independent from each other except the `math` mode which relies on the bigint mode and the operator overloading.

[1] `https://tc39.github.io/proposal-bigint/`

2 Operator overloading

2.1 Introduction

If the operands of an operator have at least one object type, a custom operator method is searched before doing the legacy Javascript `ToNumber` conversion.

For unary operators, the custom function is looked up in the object and has the following name:

`unary +`	`Symbol.operatorPlus`
`unary -`	`Symbol.operatorNeg`
`++`	`Symbol.operatorInc`
`--`	`Symbol.operatorDec`
`~`	`Symbol.operatorNot`

For binary operators:

- If both operands have the same constructor function, then the operator is looked up in the constructor.
- Otherwise, the property `Symbol.operatorOrder` is looked up in both constructors and converted to `Int32`. The operator is then looked in the constructor with the larger `Symbol.operatorOrder` value. A `TypeError` is raised if both constructors have the same `Symbol.operatorOrder` value.

The operator is looked up with the following name:

`+`	`Symbol.operatorAdd`
`-`	`Symbol.operatorSub`
`*`	`Symbol.operatorMul`
`/`	`Symbol.operatorDiv`
`%`	`Symbol.operatorMod`
`% (math mode)`	`Symbol.operatorMathMod`
`**`	`Symbol.operatorPow`
`|`	`Symbol.operatorOr`
`^`	`Symbol.operatorXor`
`&`	`Symbol.operatorAnd`
`<<`	`Symbol.operatorShl`
`>>`	`Symbol.operatorShr`
`<`	`Symbol.operatorCmpLT`
`>`	`Symbol.operatorCmpLT`, operands swapped
`<=`	`Symbol.operatorCmpLE`
`>=`	`Symbol.operatorCmpLE`, operands swapped
`==, !=`	`Symbol.operatorCmpEQ`

The return value of `Symbol.operatorCmpLT`, `Symbol.operatorCmpLE` and `Symbol.operatorCmpEQ` is converted to `Boolean`.

2.2 Builtin Object changes

2.2.1 `Symbol` constructor

The following global symbols are added for the operator overloading:

```
operatorOrder
operatorAdd
operatorSub
operatorMul
operatorDiv
operatorMod
operatorPow
operatorShl
operatorShr
operatorAnd
operatorOr
operatorXor
operatorCmpLT
operatorCmpLE
operatorCmpEQ
operatorPlus
operatorNeg
operatorNot
operatorInc
operatorDec
```

3 The BigInt Mode

3.1 Introduction

The bigint mode is enabled with the `"use bigint"` directive. It propagates the same way as the strict mode. In bigint mode, all integers are considered as `bigint` (arbitrarily large integer, similar to the TC39 BigInt proposal[1]) instead of `number` (floating point number). In order to be able to exchange data between standard and bigint modes, numbers are internally represented as 3 different types:

- Small integer (SmallInt): 32 bit integer[2].
- Big integer (BigInt): arbitrarily large integer.
- Floating point number (Float).

In standard mode, the semantics of each operation is modified so that when it returns a `number`, it is either of SmallInt or Float. But the difference between SmallInt and Float is not observable in standard mode.

In bigint mode, each operation behaves differently whether its operands are integer or float. The difference between SmallInt and BigInt is not observable (i.e. they are both integers).

The following table summarizes the observable types:

Internal type	**Observable type (standard mode)**	**Observable type (bigint mode)**
SmallInt	number	bigint
BigInt	bigint	bigint
Float	number	number

3.2 Changes that introduce incompatibilities with Javascript

3.2.1 Standard mode

There is no incompatibility with Javascript.

3.2.2 Bigint mode

The following changes are visible:

- Integer and Float are different types. Constants are typed. For example: `typeof 1.0 === "number"` and `typeof 1 === "bigint"`. Another consequence is that `1.0 === 1` is false.
- The range of integers is unlimited. In standard mode: `2**53 + 1 === 2**53`. This is no longer true with the bignum extensions.
- Binary bitwise operators do not truncate to 32 bits i.e. `0x800000000 | 1 === 0x800000001` while it gives `1` in standard mode.
- Bitwise shift operators do not truncate to 32 bits and do not mask the shift count with `0x1f` i.e. `1 << 32 === 4294967296` while it gives `1` in standard mode. However, the `>>>` operator (unsigned right shift) which is useless with bignums keeps its standard mode behavior[3].
- Operators with integer operands never return the minus zero floating point value as result. Hence `Object.is(0, -0) === true`. Use `-0.0` to create a minus zero floating point value.

[1] `https://tc39.github.io/proposal-bigint/`

[2] Could be extended to 53 bits without changing the principle.

[3] The unsigned right right operator could be removed in bigint mode.

- The `ToPrimitive` abstract operation is called with the `"integer"` preferred type when an integer is required (e.g. for bitwise binary or shift operations).
- The prototype of integers is no longer `Number.prototype`. Instead `Object.getPrototypeOf(1) === BigInt.prototype`. The prototype of floats remains Number.prototype.
- If the TC39 BigInt proposal is supported, there is no observable difference between integers and `bigint`s.

3.3 Operators

3.3.1 Arithmetic operators

The operands are converted to number values as in normal Javascript. Then the general case is that an Integer is returned if both operands are Integer. Otherwise, a float is returned.

The `+` operator also accepts strings as input and behaves like standard Javascript in this case.

The binary operator `%` returns the truncated remainder of the division. When the result is an Integer type, a dividend of zero yields a RangeError exception.

The binary operator `%` in math mode returns the Euclidian remainder of the division i.e. it is always positive.

The binary operator `/` returns a float.

The binary operator `/` in math mode returns a float if one of the operands is float. Otherwise, `BigInt[Symbol.operatorDiv]` is invoked.

The returned type of `a ** b` is Float if a or b are Float. If a and b are integers:

- $b < 0$ returns a Float in bigint mode. In math mode, `BigInt[Symbol.operatorPow]` is invoked.
- $b >= 0$ returns an integer.

The unary `-` and unary `+` return the same type as their operand. They performs no floating point rounding when the result is a float.

The unary operators `++` and `--` return the same type as their operand.

In standard mode:

If the operator returns an Integer and that the result fits a SmallInt, it is converted to SmallInt. Otherwise, the Integer is converted to a Float.

In bigint mode:

If the operator returns an Integer and that the result fits a SmallInt, it is converted to SmallInt. Otherwise it is a BigInt.

3.3.2 Logical operators

In standard mode:

The operands have their standard behavior. If the result fits a SmallInt it is converted to a SmallInt. Otherwise it is a Float.

In bigint mode:

The operands are converted to integer values. The floating point values are converted to integer by rounding them to zero.

The logical operators are defined assuming the integers are represented in two complement notation.

For `<<` and `<<`, the shift can be positive or negative. So `a << b` is defined as $\lfloor a/2^{-b} \rfloor$ and `a >> b` is defined as $\lfloor a/2^{b} \rfloor$.

The operator `>>>` is supported for backward compatibility and behaves the same way as Javascript i.e. implicit conversion to `Uint32`.

If the result fits a SmallInt it is converted to a SmallInt. Otherwise it is a BigInt.

3.3.3 Relational operators

The relational operators `<`, `<=`, `>`, `>=`, ==, != work as expected with integers and floating point numbers (e.g. `1.0 == 1` is true).

The strict equality operators === and !== have the usual Javascript semantics. In particular, different types never equal, so `1.0 === 1` is false.

3.4 Number literals

Number literals in bigint mode have a slightly different behavior than in standard Javascript:

1. A number literal without a decimal point or an exponent is considered as an Integer. Otherwise it is a Float.
2. Hexadecimal, octal or binary floating point literals are accepted with a decimal point or an exponent. The exponent is specified with the `p` letter assuming a base 2. The same convention is used by C99. Example: `0x1p3` is the same as `8.0`.

3.5 Builtin Object changes

3.5.1 `BigInt` function

The `BigInt` function cannot be invoked as a constructor. When invoked as a function, it converts its first parameter to an integer. When a floating point number is given as parameter, it is truncated to an integer with infinite precision.

`BigInt` properties:

`asIntN(bits, a)`
: Set $b = a \pmod{2^{bits}}$. Return b if $b < 2^{bits-1}$ otherwise $b - 2^{bits}$.

`asUintN(bits, a)`
: Return $a \pmod{2^{bits}}$.

`tdiv(a, b)`
: Return $trunc(a/b)$. `b = 0` raises a RangeError exception.

`fdiv(a, b)`
: Return $\lfloor a/b \rfloor$. `b = 0` raises a RangeError exception.

`cdiv(a, b)`
: Return $\lceil a/b \rceil$. `b = 0` raises a RangeError exception.

`ediv(a, b)`
: Return $sgn(b)\lfloor a/|b| \rfloor$ (Euclidian division). `b = 0` raises a RangeError exception.

`tdivrem(a, b)`
`fdivrem(a, b)`
`cdivrem(a, b)`
`edivrem(a, b)`
: Return an array of two elements. The first element is the quotient, the second is the remainder. The same rounding is done as the corresponding division operation.

`sqrt(a)` Return $\lfloor \sqrt(a) \rfloor$. A RangeError exception is raised if $a < 0$.

`sqrtrem(a)`
: Return an array of two elements. The first element is $\lfloor \sqrt{a} \rfloor$. The second element is $a - \lfloor \sqrt{a} \rfloor^2$. A RangeError exception is raised if $a < 0$.

`floorLog2(a)`
: Return -1 if $a \leq 0$ otherwise return $\lfloor \log 2(a) \rfloor$.

`ctz(a)`
: Return the number of trailing zeros in the two's complement binary representation of a. Return -1 if $a = 0$.

3.5.2 `BigInt.prototype`

It is a normal object.

3.5.3 `Number` constructor

The number constructor returns its argument rounded to a Float using the global floating point environement. In bigint mode, the Number constructor returns a Float. In standard mode, it returns a SmallInt if the value fits it, otherwise a Float.

3.5.4 `Number.prototype`

The following properties are modified:

`toString(radix)`
: In bigint mode, integers are converted to the specified radix with infinite precision.

`toPrecision(p)`
`toFixed(p)`
`toExponential(p)`
: In bigint mode, integers are accepted and converted to string with infinite precision.

`parseInt(string, radix)`
: In bigint mode, an integer is returned and the conversion is done with infinite precision.

3.5.5 `Math` object

The following properties are modified:

`abs(x)`
: Absolute value. Return an integer if `x` is an Integer. Otherwise return a Float. No rounding is performed.

`min(a, b)`

`max(a, b)`
: No rounding is performed. The returned type is the same one as the minimum (resp. maximum) value.

4 Arbitrarily large floating point numbers

4.1 Introduction

This extension adds the `BigFloat` primitive type. The `BigFloat` type represents floating point numbers are in base 2 with the IEEE 754 semantics. A floating point number is represented as a sign, mantissa and exponent. The special values `NaN`, `+/-Infinity`, `+0` and `-0` are supported. The mantissa and exponent can have any bit length with an implementation specific minimum and maximum.

4.2 Floating point rounding

Each floating point operation operates with infinite precision and then rounds the result according to the specified floating point environment (`BigFloatEnv` object). The status flags of the environment are also set according to the result of the operation.

If no floating point environment is provided, the global floating point environment is used.

The rounding mode of the global floating point environment is always `RNDN` ("round to nearest with ties to even")[1]. The status flags of the global environment cannot be read[2]. The precision of the global environment is `BigFloatEnv.prec`. The number of exponent bits of the global environment is `BigFloatEnv.expBits`. If `BigFloatEnv.expBits` is strictly smaller than the maximum allowed number of exponent bits (`BigFloatEnv.expBitsMax`), then the global environment subnormal flag is set to `true`. Otherwise it is set to `false`;

For example, `prec = 53` and `expBits = 11` give exactly the same precision as the IEEE 754 64 bit floating point type. It is the default floating point precision.

The global floating point environment can only be modified temporarily when calling a function (see `BigFloatEnv.setPrec`). Hence a function can change the global floating point environment for its callees but not for its caller.

4.3 Operators

The builtin operators are extended so that a BigFloat is returned if at least one operand is a BigFloat. The computations are always done with infinite precision and rounded according to the global floating point environment.

`typeof` applied on a `BigFloat` returns `bigfloat`.

BigFloat can be compared with all the other numeric types and the result follows the expected mathematical relations.

However, since BigFloat and Number are different types they are never equal when using the strict comparison operators (e.g. `0.0 === 0.0l` is false).

4.4 BigFloat literals

BigFloat literals are floating point numbers with a trailing `l` suffix. BigFloat literals have an infinite precision. They are rounded according to the global floating point environment when they are evaluated.[3]

[1] The rationale is that the rounding mode changes must always be explicit.

[2] The rationale is to avoid side effects for the built-in operators.

[3] Base 10 floating point literals cannot usually be exactly represented as base 2 floating point number. In order to ensure that the literal is represented accurately with the current precision, it must be evaluated at runtime.

4.5 Builtin Object changes

4.5.1 `BigFloat` function

The `BigFloat` function cannot be invoked as a constructor. When invoked as a function: the parameter is converted to a primitive type. If the result is a numeric type, it is converted to BigFloat without rounding. If the result is a string, it is converted to BigFloat using the precision of the global floating point environment.

`BigFloat` properties:

`LN2`

`PI` Getter. Return the value of the corresponding mathematical constant rounded to nearest, ties to even with the current global precision. The constant values are cached for small precisions.

`MIN_VALUE`
`MAX_VALUE`
`EPSILON` Getter. Return the minimum, maximum and epsilon `BigFloat` values (same definition as the corresponding `Number` constants).

`fpRound(a[, e])`
Round the floating point number `a` according to the floating point environment `e` or the global environment if `e` is undefined.

`parseFloat(a[, radix[, e]])`
Parse the string `a` as a floating point number in radix `radix`. The radix is 0 (default) or from 2 to 36. The radix 0 means radix 10 unless there is a hexadecimal or binary prefix. The result is rounded according to the floating point environment `e` or the global environment if `e` is undefined.

`add(a, b[, e])`
`sub(a, b[, e])`
`mul(a, b[, e])`
`div(a, b[, e])`
Perform the specified floating point operation and round the floating point number `a` according to the floating point environment `e` or the global environment if `e` is undefined. If `e` is specified, the floating point status flags are updated.

`floor(x[, e])`
`ceil(x[, e])`
`round(x[, e])`
`trunc(x[, e])`
Round to integer. A rounded `BigFloat` is returned. `e` is an optional floating point environment.

`fmod(x, y[, e])`
`remainder(x, y[, e])`
Floating point remainder. The quotient is truncated to zero (fmod) or to the nearest integer with ties to even (remainder). `e` is an optional floating point environment.

`sqrt(x[, e])`
Square root. Return a rounded floating point number. `e` is an optional floating point environment.

`sin(x[, e])`
`cos(x[, e])`
`tan(x[, e])`
`asin(x[, e])`
`acos(x[, e])`
`atan(x[, e])`
`atan2(x, y[, e])`
`exp(x[, e])`
`log(x[, e])`
`pow(x, y[, e])`
: Transcendental operations. Return a rounded floating point number. `e` is an optional floating point environment.

4.5.2 `BigFloat.prototype`

The following properties are modified:

`toString(radix)`
: For floating point numbers:
  - If the radix is a power of two, the conversion is done with infinite precision.
  - Otherwise, the number is rounded to nearest with ties to even using the global precision. It is then converted to string using the minimum number of digits so that its conversion back to a floating point using the global precision and round to nearest gives the same number.

`toPrecision(p[, rnd_mode])`
`toFixed(p[, rnd_mode])`
`toExponential(p[, rnd_mode])`
: Same semantics as the corresponding `Number` functions with BigFloats. There is no limit on the accepted precision `p`. The rounding mode can be optionally specified. It is set by default to `BigFloatEnv.RNDNA`.

4.5.3 `BigFloatEnv` constructor

The `BigFloatEnv([p, [,rndMode]]` constructor cannot be invoked as a function. The floating point environment contains:

- the mantissa precision in bits
- the exponent size in bits assuming an IEEE 754 representation;
- the subnormal flag (if true, subnormal floating point numbers can be generated by the floating point operations).
- the rounding mode
- the floating point status. The status flags can only be set by the floating point operations. They can be reset with `BigFloatEnv.prototype.clearStatus()` or with the various status flag setters.

`new BigFloatEnv([p, [,rndMode]]` creates a new floating point environment. The status flags are reset. If no parameter is given the precision, exponent bits and subnormal flags are copied from the global floating point environment. Otherwise, the precision is set to `p`, the number of exponent bits is set to `expBitsMax` and the subnormal flags is set to `false`. If `rndMode` is `undefined`, the rounding mode is set to `RNDN`.

`BigFloatEnv` properties:

`prec`
: Getter. Return the precision in bits of the global floating point environment. The initial value is `53`.

`expBits` Getter. Return the exponent size in bits of the global floating point environment assuming an IEEE 754 representation. If `expBits < expBitsMax`, then subnormal numbers are supported. The initial value is `11`.

`setPrec(f, p[, e])`
Set the precision of the global floating point environment to `p` and the exponent size to `e` then call the function `f`. Then the Float precision and exponent size are reset to their precious value and the return value of `f` is returned (or an exception is raised if `f` raised an exception). If `e` is `undefined` it is set to `BigFloatEnv.expBitsMax`. `p` must be >= 53 and `e` must be >= 11 so that the global precision is at least equivalent to the IEEE 754 64 bit doubles.

`precMin` Read-only integer. Return the minimum allowed precision. Must be at least 2.

`precMax` Read-only integer. Return the maximum allowed precision. Must be at least 53.

`expBitsMin`
Read-only integer. Return the minimum allowed exponent size in bits. Must be at least 3.

`expBitsMax`
Read-only integer. Return the maximum allowed exponent size in bits. Must be at least 11.

`RNDN` Read-only integer. Round to nearest, with ties to even rounding mode.

`RNDZ` Read-only integer. Round to zero rounding mode.

`RNDD` Read-only integer. Round to -Infinity rounding mode.

`RNDU` Read-only integer. Round to +Infinity rounding mode.

`RNDNA` Read-only integer. Round to nearest, with ties away from zero rounding mode.

`RNDNU` Read-only integer. Round to nearest, with ties to +Infinity rounding mode.

`RNDF`[4] Read-only integer. Faithful rounding mode. The result is non-deterministicly rounded to -Infinity or +Infinity. This rounding mode usually gives a faster and deterministic running time for the floating point operations.

`BigFloatEnv.prototype` properties:

`prec` Getter and setter (Integer). Return or set the precision in bits.

`expBits` Getter and setter (Integer). Return or set the exponent size in bits assuming an IEEE 754 representation.

`rndMode` Getter and setter (Integer). Return or set the rounding mode.

`subnormal`
Getter and setter (Boolean). subnormal flag. It is false when `expBits = expBitsMax`.

`clearStatus()`
Clear the status flags.

`invalidOperation`
`divideByZero`
`overflow`

`underflow`
`inexact` Getter and setter (Boolean). Status flags.

[4] Could be removed in case a deterministic behvior for floating point operations is required.

4.5.4 `Math` object

The following properties are modified:

`abs(x)` Absolute value. If `x` is a BigFloat, its absolute value is returned as a BigFloat. No rounding is performed.

`min(a, b)`

`max(a, b)` The returned type is the same one as the minimum (resp. maximum) value, so `BigFloat` values are accepted. When a `BigFloat` is returned, no rounding is performed.

5 Math mode

5.1 Introduction

A new *math mode* is enabled with the `"use math"` directive. `"use bigint"` is implied in math mode. With this mode, writing mathematical expressions is more intuitive, exact results (e.g. fractions) can be computed for all operators and floating point literals have the `BigFloat` type by default.

It propagates the same way as the *strict mode*. In this mode:

- The `^` operator is a similar to the power operator (`**`).
- The power operator (both `^` and `**`) grammar is modified so that `-2^2` is allowed and yields `-4`.
- The logical xor operator is still available with the `^^` operator.
- The division operator invokes `BigInt[Symbol.operatorDiv]` in case both operands are integers.
- The power operator invokes `BigInt[Symbol.operatorPow]` in case both operands are integers and the exponent is strictly negative.
- The modulo operator returns the Euclidian remainder (always positive) instead of the truncated remainder.
- Floating point literals are `BigFloat` by default (i.e. a `l` suffix is implied).

5.2 Builtin Object changes

5.2.1 `Symbol` constructor

The following global symbol is added for the operator overloading:

`operatorMathMod`

5.3 Remaining issues

1. A new floating point literal suffix could be added for `Number` literals.