Javascript Bignum Extensions

Version 2020-01-11

Author: Fabrice Bellard

Table of Contents

1 Introduction

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

- Operator overloading with a dispatch logic inspired from the proposal available at `https://github.com/tc39/proposal-operator-overloading/`.
- Arbitrarily large floating point numbers (`BigFloat`) in base 2 using the IEEE 754 semantics.
- Arbitrarily large floating point numbers (`BigDecimal`) in base 10 based on the proposal available at `https://github.com/littledan/proposal-bigdecimal`.
- `math` mode: arbitrarily large integers and floating point numbers are available by default. The integer division and power can be overloaded for example to return a fraction. The modulo operator (`%`) is defined as the Euclidian remainder. `^` is an alias to the power operator (`**`). `^^` is used as the exclusive or operator.

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

2 Operator overloading

Operator overloading is inspired from the proposal available at `https://github.com/tc39/proposal-operator-overloading/`. It implements the same dispatch logic but finds the operator sets by looking at the `Symbol.operatorSet` property in the objects. The changes were done in order to simplify the implementation.

More precisely, the following modifications were made:

- `with operators from` is not supported. Operator overloading is always enabled.
- The dispatch is not based on a static `[[OperatorSet]]` field in all instances. Instead, a dynamic lookup the of the `Symbol.operatorSet` property is done. This property is typically added in the prototype of each object.
- `Operators.create(...dictionaries)` is used to create a new OperatorSet object. The `Operators` function is supported as an helper to be closer to the TC39 proposal.
- `[]` cannot be overloaded.
- In math mode, the BigInt division and power operators can be overloaded with `Operators.updateBigIntOperators(dictionary)`.

3 BigInt extensions

A few properties are added to the BigInt object:

`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 $a = 0$.

4 BigFloat

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. The default precision is `prec = 113` and `expBits = 15` (IEEE 754 128 bit floating point).

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.

`isFinite(a)`
Return true if `a` is a finite bigfloat.

`isNaN(a)` Return true if `a` is a NaN bigfloat.

`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)`

`ceil(x)`

`round(x)`

`trunc(x)` Round to an integer. No additional rounding is performed.

`abs(x)` Return the absolute value of x. No additional rounding is performed.

`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:

`valueOf()`
: Return the bigfloat primitive value corresponding to `this`.

`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.

  The exponent letter is `e` for base 10, `p` for bases 2, 8, 16 with a binary exponent and `@` for the other bases.

`toPrecision(p, rnd_mode = BigFloatEnv.RNDNA, radix = 10)`
`toFixed(p, rnd_mode = BigFloatEnv.RNDNA, radix = 10)`
`toExponential(p, rnd_mode = BigFloatEnv.RNDNA, radix = 10)`
: Same semantics as the corresponding `Number` functions with BigFloats. There is no limit on the accepted precision `p`. The rounding mode and radix can be optionally specified. The radix must be between 2 and 36.

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 `113`.

`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 `15`.

`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`.

`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 113.

`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 15.

`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.

`RNDA` Read-only integer. Round away from zero rounding mode.

`RNDF`[4] Read-only integer. Faithful rounding mode. The result is non-deterministically 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.

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

`invalidOperation`
`divideByZero`
`overflow`

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

5 BigDecimal

This extension adds the `BigDecimal` primitive type. The `BigDecimal` type represents floating point numbers in base 10. It is inspired from the proposal available at `https://github.com/littledan/proposal-bigdecimal`.

The `BigDecimal` floating point numbers are always normalized and finite. There is no concept of `-0`, `Infinity` or `NaN`. By default, all the computations are done with infinite precision.

5.1 Operators

The following builtin operators support BigDecimal:

`+`

`-`

`*` Both operands must be BigDecimal. The result is computed with infinite precision.

`%` Both operands must be BigDecimal. The result is computed with infinite precision. A range error is throws in case of division by zero.

`/` Both operands must be BigDecimal. A range error is throws in case of division by zero or if the result cannot be represented with infinite precision (use `BigDecimal.div` to specify the rounding).

`**` Both operands must be BigDecimal. The exponent must be a positive integer. The result is computed with infinite precision.

`===` When one of the operand is a BigDecimal, return true if both operands are a BigDecimal and if they are equal.

`==`

`!=`

`<=`

`>=`

`<`

`>`

Numerical comparison. When one of the operand is not a BigDecimal, it is converted to BigDecimal by using ToString(). Hence comparisons between Number and BigDecimal do not use the exact mathematical value of the Number value.

5.2 BigDecimal literals

BigDecimal literals are decimal floating point numbers with a trailing `m` suffix.

5.3 Builtin Object changes

5.3.1 The `BigDecimal` function.

It returns `0m` if no parameter is provided. Otherwise the first parameter is converted to a bigdecimal by using ToString(). Hence Number value are not converted to their exact numerical value as BigDecimal.

5.3.2 Properties of the `BigDecimal` object

`add(a, b[, e])`
`sub(a, b[, e])`
`mul(a, b[, e])`
`div(a, b[, e])`
`mod(a, b[, e])`
`sqrt(a, e)`
`round(a, e)`

Perform the specified floating point operation and round the floating point result according to the rounding object `e`. If the rounding object is not present, the operation is executed with infinite precision.

For `div`, a `RangeError` exception is thrown in case of division by zero or if the result cannot be represented with infinite precision if no rounding object is present.

For `sqrt`, a range error is thrown if `a` is less than zero.

The rounding object must contain the following properties: `roundingMode` is a string specifying the rounding mode (`"floor"`, `"ceiling"`, `"down"`, `"up"`, `"half-even"`, `"half-up"`). Either `maximumSignificantDigits` or `maximumFractionDigits` must be present to specify respectively the number of significant digits (must be >= 1) or the number of digits after the decimal point (must be >= 0).

5.3.3 Properties of the `BigDecimal.prototype` object

`valueOf()`

Return the bigdecimal primitive value corresponding to `this`.

`toString()`

Convert `this` to a string with infinite precision in base 10.

`toPrecision(p, rnd_mode = "half-up")`
`toFixed(p, rnd_mode = "half-up")`
`toExponential(p, rnd_mode = "half-up")`

Convert the BigDecimal `this` to string with the specified precision `p`. There is no limit on the accepted precision `p`. The rounding mode can be optionally specified. `toPrecision` outputs either in decimal fixed notation or in decimal exponential notation with a `p` digits of precision. `toExponential` outputs in decimal exponential notation with `p` digits after the decimal point. `toFixed` outputs in decimal notation with `p` digits after the decimal point.

6 Math mode

A new *math mode* is enabled with the `"use math"` directive. It propagates the same way as the *strict mode*. It is designed so that arbitrarily large integers and floating point numbers are available by default. In order to minimize the number of changes in the Javascript semantics, integers are represented either as Number or BigInt depending on their magnitude. Floating point numbers are always represented as BigFloat.

The following changes are made to the Javascript semantics:

- Floating point literals (i.e. number with a decimal point or an exponent) are `BigFloat` by default (i.e. a `l` suffix is implied). Hence `typeof 1.0 === "bigfloat"`.
- Integer literals (i.e. numbers without a decimal point or an exponent) with or without the `n` suffix are `BigInt` if their value cannot be represented as a safe integer. A safe integer is defined as a integer whose absolute value is smaller or equal to `2**53-1`. Hence `typeof 1 === "number "`, `typeof 1n === "number"` but `typeof 9007199254740992 === "bigint"` .
- All the bigint builtin operators and functions are modified so that their result is returned as a Number if it is a safe integer. Otherwise the result stays a BigInt.
- The builtin operators are modified so that they return an exact result (which can be a BigInt) if their operands are safe integers. Operands between Number and BigInt are accepted provided the Number operand is a safe integer. The integer power with a negative exponent returns a BigFloat as result. The integer division returns a BigFloat as result.
- The `^` operator is an alias 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 integer division operator can be overloaded by modifying the corresponding operator in `BigInt.prototype.[[OperatorSet]]`.
- The integer power operator with a non zero negative exponent can be overloaded by modifying the corresponding operator in `BigInt.prototype.[[OperatorSet]]`.
- The modulo operator (`%`) returns the Euclidian remainder (always positive) instead of the truncated remainder.