Javascript Bignum Extension

Version 0.1

Author: Fabrice Bellard

Table of Contents

1 Introduction

The aim of the BigNum extension is to integrate arbitrary precision integers and floating point numbers in the language. It is not backward compatible with standard Javascript but tries to minimize the changes in the language.

The BigNum extension replaces the `Number` Javascript type with the `Integer` and `Float` types. `Integer` represents arbitrary large integers. `Float` represents floating point numbers in base 2 with arbitrary precision using the IEEE 754 semantics.

The BigNum extension allows the overloading of the standard operators to support new types such as complex numbers, fractions or matrixes.

It also provides functions to specify the rounding of the floating operations and to access the IEEE 754 status flags.

2 Changes that introduce incompatibilities with Javascript

- The range of integers is unlimited. In standard javascript: `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 Javascript.
- 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 Javascript. However, the `>>>` operator (unsigned right shift) which is useless with bignums keeps its standard Javascript behavior.
- 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.
- Division or modulo with a zero integer dividend raises an exception i.e. `1/0` throws a `RangeError` exception. However, division by the floating point zero returns NaN as in standard Javascript: `1/0.0 === NaN`. The same holds for integers elevated to negative integer powers i.e. `0**-1` throws a `RangeError` exception.
- 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 or floats is no longer `Number.prototype`. Instead `Object.getPrototypeOf(1) === Integer.prototype` and `Object.getPrototypeOf(1.0) === Float.prototype`.

3 New Types

- `Integer` is an arbitrary large integer. The NaN, Infinity or -0 IEEE 754 values cannot be represented as integers.
- `Float` is an arbitrary large floating point number in base 2 with the IEEE 754 semantics. It 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.

`typeof` still returns `"number"` for integer or floating point values for backward compatibility.

4 Floating point rounding

Each floating point operation operates with infinite precision and then rounds the result according to the specified floating point environment (`FloatEnv` 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 `FloatEnv.prec`. The number of exponent bits of the global environment is `FloatEnv.expBits`. If `FloatEnv.expBits` is strictly smaller than the maximum allowed number of exponent bits (`FloatEnv.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 `FloatEnv.setPrec`). Hence a function can change the global floating point environment for its callees but not for its caller.

[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.

5 Math mode

A new *math mode* is enabled with the `"use math"` directive. It propagates the same way as the *strict mode*. In this mode:

- The `^` operator is a similar to the power operator (`**`), except that `a ^ b` invokes `Integer.operator_math_pow` if a and b are integers and $|a| \geq 2$ and $b < 0$.
- The logical xor operator is still available with the `^^` operator.
- The division operator invokes `Integer.operator_math_div` in case both operands are integers.
- The modulo operator returns the Euclidian remainder (always positive) instead of the truncated remainder.

6 Operators

6.1 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.

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, `Integer.operator_math_div` is invoked (and returns a fraction for example).

When the result is an Integer type, a dividend of zero yields a RangeError exception.

The returned type of `a ** b` or `a ^ b` (math mode) is Float if a or b are Float. If a and b are integers:

- $a = 0$ and $b < 0$ generates a RangeError exception
- $|a| \geq 2$ and $b < 0$ returns a Float in normal mode. In math mode, `Integer.operator_math_pow` is invoked (and returns a fraction for example)
- otherwise an integer is returned.

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.

6.2 Logical operators

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`.

6.3 Relational operators

The relational operators `<`, `<=`, `>`, `>=`, ==, != work as expected with integers and floating point numbers.

The relational operators === and !== have the usual Javascript semantics. They accept Float and Integer. A Float and an Integer are equal if they represent the same value (e.g. `1.0 === 1`).

6.4 Operator overloading

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 +	operator_plus
unary -	operator_neg
++	operator_inc
--	operator_dec
~	operator_not

For binary operators:

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

The operator is looked up with the following name:

+	operator_add
-	operator_sub
*	operator_mul
/	operator_div
/ (math mode)	operator_math_div
%	operator_mod
% (math mode)	operator_math_mod
^ (math mode)	operator_math_pow
**	operator_pow
|	operator_or
^	operator_xor
&	operator_and
<<	operator_shl
>>	operator_shr
<	operator_cmp_lt
>	operator_cmp_lt, operands swapped
<=	operator_cmp_le
>=	operator_cmp_le, operands swapped
==, !=	operator_cmp_eq

The return value of `operator_cmp_lt`, `operator_cmp_le` and `operator_cmp_eq` is converted to `Boolean`.

7 Number literals

Number literals have a slightly different behavior than standard Javascript:

1. A number literal without a decimal point or an exponent is considered as an Integer. Otherwise it is a Float.
2. Floating point literals have an infinite precision. They are rounded according to the global floating point environment when they are evaluated.[1]
3. 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`.

[1] 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.

8 Builtin Objects changes

8.1 `Integer` function

The `Integer` 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.

`Integer` properties:

`isInteger(a)`
: Return true if a is of the Integer type. Hence `Integer.isInteger(1.0)` returns false because `1.0` is a floating point number.

`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 e such as $a = m2^e$ with m an odd integer. Return 0 if $a = 0$.

8.2 `Integer.prototype`

It is a normal object. Its prototype is `Number.prototype`.

8.3 `Float` function

The `Float` function cannot be invoked as a constructor. When invoked as a function, it converts its first parameter to a floating point number. When a floating point number is given as parameter, it is returned without rounding. When an integer is given as parameter, it is returned as a floating point number without rounding.

`Float` 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.

`isFloat(a)`
Return true if a is of the Float type. Hence `Float.isFloat(1.0)` returns true.

`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.

8.4 `Float.prototype`

It is a normal object. Its prototype is `Number.prototype`.

8.5 `FloatEnv` constructor

The `FloatEnv([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 `FloatEnv.prototype.clearStatus()` or with the various status flag setters.

`new FloatEnv([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`.

`FloatEnv` 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 `FloatEnv.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 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.

`RNDF` 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.

`FloatEnv.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.

8.6 `Number.prototype`

The following properties are modified:

`toString(radix)`
: Integers are converted in the specified radix with infinite precision.

  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 global number of exponent bits or subnormal flag are not taken into account during the conversion.

8.7 `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.

`floor(x[, e])`
`ceil(x[, e])`
`round(x[, e])`
`trunc(x[, e])`
Round to integer. When `x` is an Integer, an integer is returned. Otherwise a rounded `Float` 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.

9 Remaining issues

1. New functions (e.g. `Math.div` and `Math.mod`) could be added to be able to call the normal division and modulo operators when in math mode.
2. Full Javascript compatibility would be possible by adding a specific bignum mode, similar to the strict mode. In this mode, the result of operators would be implicitely typed as integer or floating point. For example, a integer result less than $2^{53} - 1$ would be typed as integer. The downside is that it significantly complicates the semantics of the language.