Ruby trunk - Bug #3589

Converting Bignums to Float for equality checks is wrong

07/21/2010 04:48 AM - taw (Tomasz Wegrzanowski)

Status:	Closed		
Priority:	Normal		
Assignee:	akr (Akira Tanaka)		
Target version:	2.0.0		
ruby -v:	ruby 1.9.1p429 (2010-07-02 revision 28523) [i386-darwin9]	**Backport:**	2.3: UNKNOWN, 2.4: UNKNOWN, 2.5: UNKNOWN

Description

=begin
In all versions of Ruby, when comparing Bignums with Floats, Bignum get converted to Floats first. This naturally results in wrong results, as this conversion is lossy. Not only will some unequal number be reported as equal, transitivity of equality gets broken:

```
big    = 10**20
bigger = big+1
flt    = big.to_f
[big == bigger, big == flt, bigger == flt] #=> [false, true, true]
```

Ruby is so close to getting equality right, it would be a shame not to get it totally right. And it's not Float's fault - IEEE 754 defines correct results of all Float operations to the last bit, and all rounding and comparisons between Floats happens with what is mathematically equivalent to infinite precision.

= Solution 1 =

Now I could be missing something, but it seems to be that it's all as simple as:

- perform equality / comparison check like now with bignum.to_f and float
- if they're not equal so far, direction of inequality is correct
- compare bignum with float.to_i

As both Float <=> Float, and Bignum <=> Bignum are exact to the last bit (ignoring issues like -0.0 etc. - they're not relevant here), this means roundtrip conversion is identity, and there can exist no other Float that would be equal to this particular Bignum, and no other Bignum that would be equal to this particular Float. It also seems to me that they'd need to be mathematically equal for that unless I miss something big.

The first check ensures all fractional bits are correct (that is if flt has any it will fail as convertion of integer to float is guaranteed not to generate any). The second check ensures that all integer bits are correct (that is every bit exceeding limit of float representation is 0, as correct float to integer conversion is guaranteed not to geterate any there)

This leads where we want:
[bigger.to_f == flt, bigger == flt.to_i]
=> [true, false]

Some pictures:

- Huge BigNum XXXXXXXXXXXXXXXYYYYYYYYYYY.00000000000000
- Huge Float XXXXXXXXXXXXXXX00000000000.00000000000000
- Small BigNum XXXXXXX.000000000000000
- Small Float XXXXXXX.YYYYYYYYY000000

0s are bits that cannot be represented, Xs bits represented by both, Ys bits represented in only one. Someone should double check with all the rounding etc. but it seems to that it's impossible for Ys to exist in both, if they're nearly equal.

= Solution 2 =

A completely different solution would be converting BigNum to Float with hardware (it's already universally supported, even if standard C tends to ignore it) in two modes - round up and round down (starting from low bits) - they will be the same or differ by 1ulp - so its impossible for the compared float to be between them. If bignum.to_f_up > flt, it's >. If bignum.to_f_down < flt, it's <. If they're all three equal, they're really ==.
=end

Associated revisions

Revision 94735e9e - 07/16/2012 09:08 AM - akr (Akira Tanaka)

- bignum.c (rb_big_float_cmp): compare an integer and float precisely. [ruby-core:31376] [Bug #3589] reported by Tomasz Wegrzanowski.

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@36404 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision 36404 - 07/16/2012 09:08 AM - akr (Akira Tanaka)

- bignum.c (rb_big_float_cmp): compare an integer and float precisely. [ruby-core:31376] [Bug #3589] reported by Tomasz Wegrzanowski.

Revision 36404 - 07/16/2012 09:08 AM - akr (Akira Tanaka)

- bignum.c (rb_big_float_cmp): compare an integer and float precisely. [ruby-core:31376] [Bug #3589] reported by Tomasz Wegrzanowski.

Revision 36404 - 07/16/2012 09:08 AM - akr (Akira Tanaka)

- bignum.c (rb_big_float_cmp): compare an integer and float precisely. [ruby-core:31376] [Bug #3589] reported by Tomasz Wegrzanowski.

Revision 36404 - 07/16/2012 09:08 AM - akr (Akira Tanaka)

- bignum.c (rb_big_float_cmp): compare an integer and float precisely. [ruby-core:31376] [Bug #3589] reported by Tomasz Wegrzanowski.

Revision 36404 - 07/16/2012 09:08 AM - akr (Akira Tanaka)

- bignum.c (rb_big_float_cmp): compare an integer and float precisely. [ruby-core:31376] [Bug #3589] reported by Tomasz Wegrzanowski.

History

#1 - 07/21/2010 11:33 AM - shyouhei (Shyouhei Urabe)

- Status changed from Open to Rejected

```
=begin
irb(main):001:0> x = 10.0 ** 20
=> 1.0e+20
irb(main):002:0> y = x + 1
=> 1.0e+20
irb(main):003:0> y == x
=> true
irb(main):004:0>

Welcome to the Real world.  It is a Float's "fault".  Do not blame us.
=end
```

#2 - 07/21/2010 02:44 PM - shyouhei (Shyouhei Urabe)

=begin
For a record: Akira Tanaka kindly told me that this behaviour was intentionally introduced since revision r1800, to make
100000000000000000000000 == 100000000000000000000000.0

http://svn.ruby-lang.org/cgi-bin/viewvc.cgi/trunk/bignum.c?r1=1800&r2=1799&pathrev=1800
=end

#3 - 07/21/2010 10:52 PM - taw (Tomasz Wegrzanowski)

=begin
This isn't how Floats and other numbers work. No numerical type can represent every real number, and many operations are implicitly followed by rounding to nearest representable value. But once rounded a number corresponds to a precise mathematical value - and comparisons never need to involve any further rounding.

Integer 10 means exactly 10, not everything that would end up as 10 if rounded. 10 == 10.2 #=> false

And r1800 is wrong. 100000000000000000000000.to_f is a precise number 9999999999999991611392,
and it doesn't equal 100000000000000000000000 any more than 10.2.to_i equals 10.2.
=end

#4 - 07/22/2010 12:55 AM - marcandre (Marc-Andre Lafortune)

- Category set to core

=begin
Hi,

On Wed, Jul 21, 2010 at 9:52 AM, Tomasz Wegrzanowski redmine@ruby-lang.org wrote:

> And r1800 is wrong. 100000000000000000000000.to_f is a precise number 9999999999999991611392,
> and it doesn't equal 100000000000000000000000 any more than 10.2.to_i equals 10.2.

You should think of floats as a range of values. For example, the float 10.2 is a range of value and the mathematical value 10.2 is simply the one that has the smallest decimal form. That range of value does not contain 10, so in Ruby 10 != 10.2

One the other hand, the float 100000000000000000000000.to_f is a range that contains both the integer 9999999999999991611392 and the integer 100000000000000000000000. The fact that the binary representation of 100000000000000000000000.to_f corresponds to 9999999999999991611392 is not important here; one the conversion to a float is done, there is no way to know which exact number was supposed to be represented, if any.

Please refer to previous discussions about floats, for instance: http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-core/25662
--
Marc-André
=end

#5 - 07/22/2010 03:01 AM - taw (Tomasz Wegrzanowski)

=begin

You should think of floats as a range of values.

This is a very common misunderstanding about floats, but it
is exactly the opposite of how IEEE 754 specifies floats to work.
They are exact numbers, and every basic floating point operation
acts as if it:

- converted arguments to infinitely long numbers
- performed operation on that
- rounded this infinitely long result the way you specified, also returning information if this caused any loss of precision.

If you treated floats as (value +- half ulp), then it would only be
consistent to treat integers as (value +- half ulp), that is 10
equaling everything in (9.5..10.5) range. That still wouldn't make
comparisons sensible if two numbers compared had different ulps,
like when comparing integers with floats.

Reading this would be a good start:
http://docs.sun.com/source/806-3568/ncg_goldberg.html
=end

#6 - 07/22/2010 01:16 PM - shyouhei (Shyouhei Urabe)

- Status changed from Rejected to Assigned

- Assignee set to matz (Yukihiro Matsumoto)

=begin

Integer 10 means exactly 10, not everything that would end up as 10 if rounded. 10 == 10.2 #=> false

but 10 == 10.0. 100000000000000000000000 should == 100000000000000000000000.0

You know, if you think it should not, you have to persuade us.

Assigning this to matz because it turned out to be a design matter.
=end

#7 - 07/25/2010 12:25 PM - taw (Tomasz Wegrzanowski)

=begin

> Integer 10 means exactly 10, not everything that would end up as 10 if rounded. 10 == 10.2 #=> false
> but 10 == 10.0. 100000000000000000000000 should == 100000000000000000000000.0
> You know, if you think it should not, you have to persuade us.

Assigning this to matz because it turned out to be a design matter.

To me 100000000000000000000000.0 means 100000000000000000000000.to_f,
or "floating point number closest to 100000000000000000000000".
This doesn't have to be exactly 100000000000000000000000.

Some arguments follow.

== Argument from other languages ==

This treatment of float equality seem to be unique to Ruby.
Compare with Python:

```
print(100000000000000000000000 == 100000000000000000000000.0) #=> False
print( 999999999999999991611392 == 100000000000000000000000.0) #=> True
print(Fraction(1,3) == 1.0/3.0) #=> False
print(Fraction(1,4) == 1.0/4.0) #=> True
```

With Perl it's more complicated, as Perl's standard number type
switches between native int, native float, and decimal the way Ruby switches
between fixint and bigint - so 1000000000000000000.0 in Perl is
decimal, not float and so exact.

Still, it doesn't follow "equal if converts".
This is the same regardless of $x being decimal or float internally.

```
my $x = 1000000000000000000;
my $a = Math::BigInt->new('1000000000000000000');
my $b = Math::BigInt->new('1000000000000000001');

print($x == $a->numify() ? "equal" : "not"); #=> equal
print($x == $b->numify() ? "equal" : "not"); #=> equal
print($x == $a ? "equal" : "not"); #=> equal
print($x == $b ? "equal" : "not"); #=> not
```

A counterexample to this would be C, which is quite explicit that
operations involving different numeric types involve implicit conversion.
It doesn't have bignums, but comparing float vs int or
double vs long long will convert and lose precision before comparison.
It also loses precision this way when comparing integers of
different signedness etc. - this mess is a good example of what
we shouldn't do ;-)

== Argument from sort ==

What would you guess this code to print?

```
# Build array of bignums, add extra floats
big = 10**50
values  = (1..10).map{|x| big + x}
values += (1..10).map{ big.to_f }
# Make sure it's sorted
values.shuffle!
values.sort!
# Just cleanup for printing
values.reject!{|x| x.is_a?(Float)}
values.map!{|x| x - big}
p values
```

If you guessed [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
you'd be wrong most of the time.

sort relies on transitivity of <=>, so if <=>
says big.to_f equals both big+1 and big+7,

it naturally assumes big+1 <=> big+7 will also be 0.

It's not a bug in sort - it's how <=> works for everything
except floats.

== Argument from mathematics ==
Ruby has a few equality-like relations - ==, eql?, equal?.
They differ, but they're all (almost) mathematical equivalence relations.
For all three, these can be expected:

- a == a
- a == b iff b == a
- if a == b and b == c then a == c

I can think of one good exception to a == a -
with things like float NaNs and sql nulls - but these really
are more techniques for handling errors than real values.

I really cannot think of a single case where it would
make sense to make equality either non-symmetric,
or non-transitive.

Similar reasoning applies to <=> - normally it defines
partial order between objects:

- a >= b and b >= a only if a == b

- if a >= b and b >= c then a >= c

Transitivity of equality, and transitivity of partial ordering
seem to be violated only in one case in Ruby - for comparisons
between floats and other numeric types. (and this causes sort
to break).

Can you think of any other type that does something like that?

== Argument from rationals ==
When you have mixed type operations, and one type is bigger,
the most obvious thing to do is simply converting argument
of smaller type to bigger type.

For example you can define pretty much every Rational vs Integer
operation as:

```
class Rational
def something(x)
x = x.to_r if x.is_a?(Integer)
...
```

And you can do this for Complex operations
and non-complex arguments etc.

You never do it the other way around -
converting to smaller type.
This would be obviously wrong:

```
class Integer
def something(x)
x = x.to_i if x.is_a?(Rational)
...
```

So why, if Rational can represent every floating
point value (except nans/infinities/negative zero that is),
do rational vs float operations downconvert to float,
instead of upconverting to rational?

I can think of no other such case anywhere.

With bignum vs float, neither is strictly wider than other,
but both can be represented as rationals.

This doesn't mean I want bignum + float to return rationals,
downconversion before returning is perfectly fine.
I just want it to be pretty much equivalent to this:
class Integer

```
def +(x)
if x.is_a?(Float) and x.finite?
return (self.to_r + x.to_r).to_f
...

== Argument from IEEE 754 ==
This .to_r/.to_f above might be puzzling, but look at this.
All basic floating point operations are defined by IEEE 754
standard as mathematically equivalent to this:
def +(x)
return (self.to_real + x.to_real).to_f(rounding_mode)
end

Where .to_real means conversion to actual mathematical
real number with potentially infinite precision, all extra
bits being 0s.

Of course it's not implemented like that - but the result
is guaranteed to be exactly the same as if it was.

== Other inaccuracies ==
I'm not really terribly bothered by that, only by equality
and <=>, but a lot of operations involving floats and other
types downconvert to float too early and lose precision.

puts(Rational(15,11)*11.0 == 15.0) #=> false
puts((Rational(15,11) * 11.0.to_r).to_f == 15.0) # => true
puts(100000000000000000000000 - 100000000000000000000000.0) #=> 0
puts((100000000000000000000000 - 100000000000000000000000.0.to_i).to_f) #=> 8388608.0

Such precision loss never happens for anything that involves
only floats, or only non-floats. And it really wouldn't be
that difficult to avoid it if we cared, but if even I don't,
I doubt others will.
=end
```

#8 - 06/26/2011 01:42 PM - akr (Akira Tanaka)

- Project changed from Ruby to Ruby trunk

- Category changed from core to core

#9 - 07/14/2012 02:26 PM - ko1 (Koichi Sasada)

- Description updated

- Assignee changed from matz (Yukihiro Matsumoto) to akr (Akira Tanaka)

- Target version set to 2.0.0

#10 - 07/14/2012 02:28 PM - akr (Akira Tanaka)

I think Bignum <=> Integer (and Integer <=> Bignum) can be implemented specially to compare them precisely.

#11 - 07/16/2012 06:09 PM - akr (Akira Tanaka)

- Status changed from Assigned to Closed

- % Done changed from 0 to 100

This issue was solved with changeset r36404.
Tomasz, thank you for reporting this issue.
Your contribution to Ruby is greatly appreciated.
May Ruby be with you.

- bignum.c (rb_big_float_cmp): compare an integer and float precisely. [Bug #3589] reported by Tomasz Wegrzanowski.