CoCoALib - Design #859

Twin-float: comparisons and equality test

28 Mar 2016 21:18 - John Abbott

Status:	Closed	**Start date:**	28 Mar 2016
Priority:	Normal	**Due date:**	
Assignee:	John Abbott	**% Done:**	100%
Category:	Tidying	**Estimated time:**	7.70 hours
Target version:	CoCoALib-0.99550 spring 2017	**Spent time:**	7.95 hours

Description

I have some philosophical questions about comparisons between twin-floats.

For the equality test there are 3 possible outcomes: true, false and InsuffPrec.

For instance, is it reasonable that there could be twin-floats X and Y such that X == Y gives InsuffPrec but that X >= Y gives true?

Related issues:

Related to CoCoALib - Bug #858: floor for TwinFloat can produce ERR::SERIOUS	**Closed**	**26 Mar 2016**
Related to CoCoALib - Feature #896: myIsEqual, myCmp: direct comparisons betw...	**In Progress**	**25 Jun 2016**

History

#1 - 28 Mar 2016 21:19 - John Abbott

- Related to Bug #858: floor for TwinFloat can produce ERR::SERIOUS added

#2 - 28 Mar 2016 21:25 - John Abbott

- Status changed from New to In Progress

- % Done changed from 0 to 10

Consider the following code excerpt:

```
X = [twin-float];
BigInt Y = floor(X);
Y <= X; // could throw???
```

Assuming the first two lines do not throw InsuffPrec, may the last line throw?

JAA's instinct says it should not throw, but the current impl can throw (e.g. if X is 1+eps where eps is of the same size as the random variation added to the secondary component).

#3 - 28 Mar 2016 21:39 - John Abbott

- Assignee set to John Abbott

The critical step is comparing two twin-floats for equality. If the two values are quite different then it is easy to say which is larger. If the equality test says true then the outcome of the other inequality relations is clear.

The interesting case arises when the values are "quite similar". Exactly what conditions are required for two twin-floats to be considered equal? And

when are they definitely unequal? In all other cases the comparison is unclear (*e.g.* must throw InsuffPrec when doing an equality test).

The basic idea behind the current impl is that each twin-float value corresponds to two nested (real) intervals: the **outer interval**, and the **inner interval**. The inner interval is much narrower than the outer interval (ratio of widths is at least 1000), and is centrally placed in the outer interval. The idea of the heuristic is that the true value very likely lies in the inner interval, and "certainly" lies in the outer interval (this is *heuristic certainty*, not the mathematical certainty of interval arithmetic).

The current equality test says "unequal" of the outer intervals are disjoint, "equal" if the inner intervals are not disjoint, and otherwise "uncertain". Furthermore, all comparisons conduct an equality test and throw "InsuffPrec" if the outcome was "uncertain".

#4 - 28 Mar 2016 22:08 - John Abbott

If we want to keep the behaviour of the comparison functions unchanged (esp. regarding when to throw InsuffPrec) then the only way I can see to avoid the possibility of Y <= X throwing is to make the conversion of the integer Y into a twin-float produce a value whose corresponding intervals are of zero width; I say this because X could be a value whose outer interval reaches to within a whisker of the integer Y, so converting Y to a twin float would create a value whose outer interval intersects that of X.

Allowing "exact" twin-floats would probably not be impossible, but it would definitely complicate arithmetic operations which would need to know when to introduce "small random perturbations".

At the moment I do not much like this idea; it seems to run contrary to the "simple heuristic" of twin-floats.

#5 - 28 Mar 2016 22:25 - John Abbott

- % Done changed from 10 to 20

A simpler idea would be to modify when comparisons throw InsuffPrec.

The model I have in mind is a fixed (non-zero) twin-float X, and a "sliding" twin-float value Y; we want to consider what happens as the value of Y becomes ever closer to that of X. We can probably just fix X = 1 without any loss of generality. We may also assume that Y tends to X from the right (*i.e.* Y is "larger than" X).

The definition of the comparison functions is to enable a "smooth transition" from "unequal" to "equal" via the intermediate "uncertain".

We must also consider cases where the outer intervals of X and Y are similarly wide, and where one is far wider than the other.

Variant A

We say that X and Y are definitely unequal if the primary component of X lies outside the outer interval of Y **and** the primary component of Y lies outside the outer interval of X; a single test suffices once we know which has the wider outer interval.

This would resolve the problem about floor(X) <= X throwing unexpectedly unless X has an unusually narrow outer interval (narrower than that of a newly created twin-float from an integer!)

Variant B

We could say that X and Y are definitely unequal if the outer interval of X is disjoint from the inner interval of Y **and** the outer interval of Y is disjoint from the inner interval of X; a single test suffices once we know which has the wider outer interval.

This is *only very slightly different* from **Variant A** since the inner interval is tiny compared to the outer interval. It does not fully resolve the issue of floor(X) <= X throwing unexpectedly, but presumably makes unexpected behaviour very rare (which could be a nightmare to debug!).

Variant C

Some other degree of overlap between the outer intervals could be allowed (but I'd need a justification for the chosen amount of allowed overlap).

NOTE I have just revised the variants to make them obviously symmetric.

#6 - 28 Mar 2016 22:45 - John Abbott

Variant A is notionally quite simple; I think I prefer it to the other possibilities. I can try implementing it (inside a #ifdef maybe), and see if any surprises come out.

One possible surprise of allowing some overlap of outer intervals is that it may be possible to have two twin-float values which test as different but which print out the same -- if I recall well, printing tries to print as many digits as are certain.

Similarly the new MantissaAndExponent2 function may possibly yield the same value for two "unequal" twin-floats -- this must be tested!

#7 - 28 Mar 2016 22:59 - John Abbott

I am still unsure whether it is reasonable for a test such as Y <= X to give true when Y = X throws InsuffPrec.

I believe the implementation should be easy: for the weak inequalities, apply first an equality test; if that produces true then return true, otherwise simply compare the primary components (regardless of whether the equality test gave false or uncertain).

What does it mean if X == Y throws InsuffPrec? The values must be so close (compared to the intrinsic precision of the less precise value) that we cannot be sure if they are equal or not. Could we nevertheless be sure that one value is at least as great as the other?

Consider this scenario: X has a very narrow outer interval, while Y has a wide one. Suppose that outer(X) is contained inside outer(Y). If outer(X) is disjoint from inner(Y), is that convincing enough to let me say to which side of Y the value X lies?

#8 - 28 Mar 2016 23:07 - John Abbott

Perhaps I could implement both approaches and let the user choose. If so, should it be a compile-time switch or a run-time switch; if run-time, can the user change it freely during the computation?

Having a switch certainly fits in while the "software laboratory" philosophy, but it might make results hard to reproduce if someone "plays carelessly" with the setting... well, perhaps that's asking for trouble anyway!

#9 - 29 Mar 2016 16:46 - John Abbott

- % Done changed from 20 to 30

The idea of changing operator>= and operator<= is trickier than I had originally thought (or hoped) because they are defined in ring.C based on the "universal comparison" member function myCmp which is expected to return -1,0,+1 depending on the comparison.

The change I was thinking of involved a "comparison function" (let's call it **myCmp5**) with 5 possible outcomes:

- +2 definitely greater
- +1 uncertain whether equal, if not equal then greater
- 0 definitely equal
- -1 uncertain whether equal, if not equal then less than
- -2 definitely less than

The classical myCmp could easily be adapted to be compatible with myCmp5 (essentially by multiplying its output by 2).

op= would throw if myCmp5 returns +1 or -1, otherwise true if myCmp5 gives 0, otherwise false.
op>= is a little more complicated: returns true if myCmp5 gives non-neg result, throws if myCmp5 gives -1, otherwise returns false.
op> returns true if myCmp5 gives +2, false if myCmp5 <= 0, and throws if myCmp gave +1.

So technically it is not that hard to adapt the comparison operators for RingElem as suggested; however, I'm not too happy about having to change all the myCmp functions.

NOTE There is also a myIsEqual mem fn which is used by op= (rather than calling myCmp which exists only for ordered rings, and which is hopefully compatible with myIsEqual!).

#10 - 30 Mar 2016 15:20 - John Abbott

- % Done changed from 30 to 50

With the impl I have in mind for op>= and op<= via myCmp5, it is not possible to have values X and Y such that the equality test X==Y does not give true and yet both X>=Y and X<=Y give true. Since X==Y does not give true the result of myCmp5 is not zero, so exactly one of op<= and op>= will give false.

In RingTwinFloat if X==Y does not give true (*i.e.* throws or gives false) then the primary component of X is clearly either greater than or less than the primary component of Y; the result of this last comparison dictates the value which myCmp5 gives.

#11 - 30 Mar 2016 15:31 - John Abbott

One unfortunate aspect of the mem fn myCmp5 is who throws InsuffPrec when a comparison operator needs to throw?

By its design myCmp5 never throws. The comparison operators are defined in ring.C and act based on the value furnished by myCmp (planned to become myCmp5).

Either RingTwinFloatImpl::myCmp5 needs an extra argument to say which comparison operator called it (so that it knows when to throw), **or** the impls of the comparison operators in ring.C need to test the return value of myCmp5, and if it is "troublesome" then they must somehow cause InsuffPrec to be thrown (but this seems to reveal too much detail about the internals of RingTwinFloatImpl to the context of ring.C).

Another approach is to create new member fns in each ring, one for each comparison operator. There could be a *default* impl which throws NotOrdDom if the ring is not ordered, and otherwise calls myCmp and returns accordingly. The impl for RingTwinFloat would then be different, and could handle *locally* the special needs of twin-floats.

I note that there is also a cmp fn for RingElem; what should this become? Should its interface be updated to allow 5 possible return values?

#12 - 30 Mar 2016 18:31 - John Abbott

- Status changed from In Progress to Resolved

- % Done changed from 50 to 80

- Estimated time set to 7.70 h

I have added a new "boolean" data member saying whether **myIsEqualNZIgnoreSign** should allow the outer intervals to overlap or not (*i.e.* **Variant (A)**). At the moment the setting is fixed at compile time, and there is no member fn to change the setting (but it should be a trivial matter to add such an operation).

All tests pass with either setting, so I have checked in the code.

I have not yet updated the doc for RingTwinFloat.

#13 - 25 Jun 2016 14:11 - John Abbott

- Status changed from Resolved to Closed

- % Done changed from 80 to 100

I have updated the documentation for RingTwinFloat.

In practice the current impl of myCmp deals first with the obvious cases, then calls myEqualNZIgnoreSign. If that fn returns uncertain3 then myCmp throws InsuffPrec. This means that if any comparison between two twin-float values X and Y throws, then **all** comparisons will throw.

The question about floor(X) <= X in comment 2 is valid. The problem arises because converting an integer to a twin-float does discard some information (an exact value has become approximate), and the comparison floor(X) <= X does implicitly convert the integer floor(X) to a twin-float.

If we implement member fns for comparisons between MachineInt, BigInt, BigRat and a ring-elem then the problem should go away. Is it worth it? I have opened issue #896 to discuss this.

I shall close this issue since its "spirit" has now been transferred to #896.

#14 - 25 Jun 2016 14:11 - John Abbott

- Related to Feature #896: myIsEqual, myCmp: direct comparisons between RingElem and MachineInt, BigInt and BigRat? added