CoCoALib - Design #1576

cmp for machine integers

08 Feb 2021 16:12 - John Abbott

Status:	In Progress	**Start date:**	08 Feb 2021
Priority:	Normal	**Due date:**	
Assignee:	John Abbott	**% Done:**	20%
Category:	Improving	**Estimated time:**	0.00 hour
Target version:	CoCoALib-0.99880	**Spent time:**	5.75 hours

Description

Redesign cmp for machine integers so that it works in all cases.

Note that C++20 already std::cmp_equal, std::cmp_less etc.
These wil eventially allow a simple impl of fully general cmp using templates.

Maybe make an interim version?
And remove cmp for MachineInt from BigIntOps?

Related issues:

Related to CoCoALib - Bug #1601: Compilation ambiguity	**Closed**	**16 Jun 2021**

History

#1 - 08 Feb 2021 16:13 - John Abbott

Also need to revise test-MachineInt2.C
Currently I have hacked it to work with the current "implementazione zoppicante".

#2 - 10 Feb 2021 21:46 - John Abbott

The impl of **IsInRange** should also be revised...

#3 - 10 Feb 2021 22:11 - John Abbott

- Status changed from New to In Progress

- % Done changed from 0 to 10

I have added 2 impls to **utils.H**; one adapted from "cppreference". Not sure how much I trust this template stuff.

Probably need to split utils.H: one change to it, and lots of files must be recompiled :-(

#4 - 10 Jun 2021 21:38 - John Abbott

- Assignee set to John Abbott

Calling the template fn cmp causes plenty of "fun" compilation problems. Not sure what to do.
The simplest is to use another name.... but is that the right approach?

#5 - 12 Jun 2021 14:21 - John Abbott

Not sure what to do.
Activating the template fn with name **cmp** causes overload resolution problems (seemingly even with cmp between two RingElem values???).
It may be possible to fix this by declaring cmp for RingElem (say) as a template specialization.. but why should I need to do this?

I tried changing the name to cmp_int, but then a compliation failure was triggered by an instantiation of the template LexCmp3 in VectorOps.H because it expected the comparison function to be called cmp.

What to do?

I did make some progress by declaring also cmp(const RingElem&, const RingElem&) in addition to the pre-existing cmp(ConstRefRingElem,...).
Not really sure why this made a difference.

Another tricky/tedious case is dealing with cmp(ConstRefRingElem, MachineInt) since the latter involves calling a ctor, which makes it a "disfavoured" match-candidate.

#6 - 12 Jun 2021 14:46 - John Abbott

There is a (hard-to-read) description of the C++ rules for overload resolution at https://en.cppreference.com/w/cpp/language/overload_resolution

#7 - 01 Sep 2021 14:13 - John Abbott

- Related to Bug #1601: Compilation ambiguity added

#8 - 01 Sep 2021 14:28 - John Abbott

I think I have now understood what is causing the trouble:
creating a ConstRefRingElem object from a const RingElem& is a **constructor call** (and not a cast-to-base-class, as I had believed).

I now have a version which compiles (just by adding some new fns to the header files).
I could supply implementations of these new functions, but that does not feel like the "right solution".

More detail: we have several different types for ring elems (which are actually smart pointers):

- **RingElem** is a unique "owning" pointer
- **RingElemAlias** is a unique "non-owning" pointer (*e.g.* for entries in a matrix)
- **ConstRefRingElem** is a typedef for const RingElemALias&

UMMMM: Actually RingElem **is** a derived class of RingElemAlias... puzzled again!

Comment: I am a little uneasy about the "kludge" which involves having both RingElem and RingElemAlias

#9 - 01 Sep 2021 14:51 - John Abbott

Here is an example program which fails to compile because the call to cmp in main refers to the template rather than the special-case function

```
#include <iostream>

template< class T, class U >
constexpr int cmp( T t, U u ) noexcept
{
  using UT = std::make_unsigned_t<T>;
  using UU = std::make_unsigned_t<U>;
  if /*constexpr*/ (std::is_signed<T>::value == std::is_signed<U>::value)
    return (t == u)?0:(t < u)?-1:1;
  else if /*constexpr*/ (std::is_signed<T>::value)
    return (t < 0) ? -1 : (UT(t) == u)? 0 : (UT(t) < u) ? -1 : 1;
  else
    return (u < 0) ? -1 : (t == UU(u))? 0 : (t < UU(u)) ? -1 : 1;
}

class BASE
{
public:
```

```
  BASE(): datum(0) {}
private:
  int datum;
};

class DERIV: public BASE
{
public:
  DERIV(): BASE(), datum2(1) {}
private:
  int datum2;
};

// special version of cmp for BASE objects
int cmp(const BASE& b1, const BASE& b2);

int main()
{
  DERIV a;
  DERIV b;

  std::cout << cmp(a,b) << std::endl; // why does template cmp beat special version?
}
```

Why is the template fn a better match???

#10 - 08 Sep 2021 15:50 - John Abbott

Here is a simpler failing example:

```
#include <iostream>

template< class T >
constexpr int func( T t ) noexcept
{
  using UT = std::make_unsigned_t<T>;
  return (t == 0);
}

class BASE
{
public:
  BASE(): datum(0) {}
  int datum;
```

```
};

class DERIV: public BASE
{
public:
  DERIV(): BASE(), datum2(1) {}
  int datum2;
};

// special version of func for BASE objects
int func(const BASE& b); // defn not needed here

int main()
{
  DERIV a;
  std::cout << func(a) << std::endl; // why does template func beat special version?
}
```

#11 - 08 Sep 2021 20:49 - John Abbott

I think I may have an ugly solution. The idea is to use yet another template fn:

```
template <typename T1, typename T2>
int cmp(const T1& x, const T2& y)
{
  if (numeric_limits<T1>::is_integer && numeric_limits<T2>::is_integer)
// EQUIV if (is_integral<T1>::value && is_integral<T2>::value)
     return cmp_for_machine_integers(x,y);
  return cmp_other_cases(x,y);
}
```

Then we must define all the comparison functions with the names cmp_for_machine_integer (the template fn already considered), or cmp_other_cases for all other cases.

I do not mind calling the original template fn cmp_for_machine_integer; but I am unhappy about having to use some name other than cmp for the other comparison fns.

NOTE we can use either **std::numeric_limits<T>::is_integer** or **std::is_integral<T>::value**; they appear to be completely equivalent. Which looks nicer? Mmmm???

#12 - 08 Sep 2021 21:13 - John Abbott

It seems there may be some nice solutions in C++20, but it is to early [Sept 2021] to adopt features from C++20:

- there is a new operator **<=>** which seems to do exactly what I want cmp to do
- templates may have "constraints", which might also obviate the problem here (I have not looked into constraints in detail)

The questions remains: what to do now, using C++14?

#13 - 13 Sep 2021 11:56 - John Abbott

- % Done changed from 10 to 20

Here is another approach which could be viewed as being "proper use of C++" (maybe):

- The only **cmp** function is the template function
- All comparisons between an object of class type and some other object are performed via **member functions**
- Possibly use of constexpr inside the template may remove any run-time overhead.

#14 - 21 Jan 2024 20:23 - John Abbott

- Target version changed from CoCoALib-0.99850 to CoCoALib-0.99880