Doc No: N2136=06-0206

Bool_set: multi-valued logic (revision 1)

Hervé Brönnimann[*] Guillaume Melquiond[†] Sylvain Pion[‡]

2006-11-01

Contents

I History of changes to this document

Since initial version (N2046=06-0116) :

— Removed overload of the short-circuiting `operator||` and `operator&&`.

II Motivation and Scope

Multi-valued logic is a natural extension of two-valued binary logic. There are several variants, with ternary logic (encoding true, false, and a third value representing maybe) being one of the most fundamental,

[*]CIS, Polytechnic University, Six Metrotech, Brooklyn, NY 11201, USA. `hbr@poly.edu`
[†]École Normale Supérieure de Lyon, 46 allée d'Italie, 69364 Lyon cedex 07, France. `guillaume.melquiond@ens-lyon.fr`
[‡]INRIA, BP 93, 06902 Sophia Antipolis cedex, France. `Sylvain.Pion@sophia.inria.fr`

explored since the 1400s by William of Occam, and in the 1920s by Łukasiewicz [1]. It is used in the design of ternary circuits, which have the promise of using less power and manipulating larger numbers than binary circuits (such circuits have been patented and proposed long ago [7], and a whole ternary computer been realized in the 1950s by the Soviets [2]). It is used in microprocessor chips for embedded systems (microwaves, etc.).

In the context of the C++ standard library, we have recently had a need for multi-valued logic in our interval arithmetic proposal [3] to represent the return value of an interval comparison (with empty or overlapping intervals). There are other cases in applications which could use multi-valued logic, with a third value representing an indeterminate (several examples come to mind: static analysis to encode if two pointers can possibly be aliased - yes, no, or maybe; should a window be refreshed - yes, no, or partially; initialization of boolean variables - true, false, intentionally nothing, or uninitialized; etc.). The SQL language also provides a three valued logic whose semantic is covered by this proposal.

One may also envision using indeterminates when working with floating point numbers and not-a-number (NaN). For instance, it could be quite elegant to use a functor returning an indeterminate when comparing a number with a nan, and could make the treatment of NaNs much more uniform. In fact, it would probably make even more sense to have floating-point predicates with NaNs return the empty `bool_set`, and have comparisons of infinities return an indeterminate. See the example section for an illustration.

Why standardize it?

- Because the functionality is beneficial for other parts of the standard library (notably the proposed interval arithmetic).
- Because there is a strong history of this kind of multi-valued logic, and the potential user base is large (note that Boost.Tribool is cited several times in “who’s using Boost?”).

III Impact on the Standard

What does it depend on, and what depends on it? Is it a pure extension, or does it require changes to standard components? Can it be implemented using today’s compilers, or does it require language features that will only be available as part of C++0x?

It is a pure extension to the standard library.

IV Design Decisions

Why did you choose the specific design that you did? What alternatives did you consider, and what are the tradeoffs? What are the consequences of your choice, for users and implementers? What decisions are left up to implementers? If there are any similar libraries in use, how do their design decisions compare to yours?

Design overview:

We largely follow the design that we had for `interval<bool>` [3] and the one that is used in Boost.tribool [5]. We have added the empty set, because it is useful for intervals and makes it mathematically complete and consistent. Note that, however, the empty set will never be created by the other values under the logical operations, so *it is possible to use `bool_set` exactly like a `tribool` as far as boolean operations are concerned*. In fact, the restriction to the two boolean values is also stable under logical operators, so by the same token it is possible to use `bool_set` exactly like a `bool` but there is little benefit to that.

Alternatives and trade-offs:

As for the representation of a multi-valued logic, there are not many alternatives. Using a floating-point value in [0,1] allows for various degrees of indetermination and can always be done without any addition to the language, but does not allow the logic operators. We specifically needed a type that extends bool and has the same operators.

The most sticky point to decide was to allow an implicit conversion to `bool`. Explicit conversions are not yet part of the language so that was not an option. An alternative would have been to provide a member function `to_bool()`. We considered it and decided against it. See the rationale.

Decisions left to implementers:

The internal representation of a `bool_set` is completely unspecified and left to implementers.

Comparison with existing libraries:

This proposal is similar to Boost.Tribool [5]. Similar functionality can also be found in the `Uncertain<bool>` class of CGAL [8].

Rationale

— **Why introduce `bool_set`?**
The choice was made in the interval proposal [3] to introduce `bool_set`, which is similar in spirit to Boost.Tribool, except that it throws an exception for invalid conversions to `bool` (à la `dynamic_cast`), and also supports the empty value.
Note that the conversion to `bool` *should* throw for an empty or indeterminate `bool_set` (see fourth item below), but it is possible for the user to test for these cases by hand and avoid exceptions altogether.
It is primarily used for the result of uncertain comparisons of types where comparisons can return a trivalent state (e.g., intervals can be compared or overlap), to avoid systematically throwing an exception when the comparison returns a result other than `true` or `false`.

— **Why introduce `bool_set` as a separate proposal from `interval<T>`?**
It can be useful independent of any numerical context. There are many uses for 3-valued boolean logic, motivating the Boost.Tribool library. Comparing values in a partial order is one of the main motivations, along with encoding a degree of uncertainty, both of which extend beyond the context of intervals.

— **Why not just three states as in Boost.Tribool**?
In addition to indeterminate, usage with `interval<T>` requires an empty state. Moreover, there is likely no penalty for having a fourth state in current systems, since the possibilities for space savings by having three instead of four are almost null. If the empty state is not needed, there is no penalty for not using it, and it will not be created by the other boolean operations. In other words, the algebra consisting of true, false, and indeterminate states is stable.

— **Why a conversion to `bool` and why does it throw**?
`bool_set` is intended to be used in the same context as a `bool`. Thus it makes sense to have some way to cast a `bool_set` into a `bool`. The big question is what to do for conversion. We decided to allow an implicit conversion to `bool`, which throws an exception. In this way, `bool_set` can be used wherever a `bool` is used and the user can expect the same result except if an exception is raised, which she then has to handle.
Alternatives: Boost.Tribool has a direct conversion to `bool`, which does not throw (and maps indeterminate to false) and removes the possibility of the tribool to be cast into an int (since only two levels of implicit conversions are allowed).

Another alternative is not to allow a conversion to `bool` and let the user add an explicit conversion (e.g., in tests, `equals(..., true)`) everywhere. The benefits are that the code will not compile when `bool` is changed into `bool_set` and will trigger a code review. The problem is that it may not be possible for the programmer to modify the code if it belongs to a library (e.g. when `bool_set` is returned by an interval comparison in an external library—hence unmodifiable— function templated by a number type which can be a floating point type or an interval).

— **Why constructor** `bool_set(bool)` **is implicit and not explicit?**
There is no danger in the conversion, and we must enable the use of the keywords `true` and `false` as valid `bool_set` values, where the explicit conversion would be heavy.

— **Why both** `bool_set::is_indeterminate()` **and** `is_indeterminate(bool_set)`?
To allow `x.is_indeterminate()` as well as `is_indeterminate(x < y)` (the code `(x < y).is_-indeterminate()` might look very strange to a novice). Furthermore, the latter expression may return a `bool_set` or a `bool` depending on the types of `x` and `y`, precluding a member function call whereas the free function will work in both cases. Thus, the non-member function is also useful for writing template code. Note that it is not forbidden (and even encouraged) to provide `inline bool has_true(bool x) { return x; }` for optimization (it is legal by the "as-if" rule as long as the semantic is the same as a conversion to `bool_set`).

— **Why** `bool_set::is_emptyset()` **and not** `bool_set::empty()`?
For uniformity and also to emphasize the fact that `bool_set` is not a container but still a set.

— **Why so many functions for testing the value of a** `bool_set`?
We feel that there should be a single function call for each situation (both for efficiency and for readability). It is possible to implement `x.equals(false)` by `x == false` but this involves a conversion to `bool` which may raise an exception, whereas `equals` will not.

— **Why set default constructed** `bool_set` **to** `false`?
By compatibility with `bool` when zero-initialized. If `bool_set` is intended to be used with a default constructed meaning 'nothing known', then it's not clear whether it should be initialized to indeterminate or empty instead. We think the performance gain of leaving the default constructed value undefined is not worth it.

— **Why no identity semantic for** `std::operator==` **on** `bool_set`?
There are two reasonnable definitions of `operator==`, the logical one returning a `bool_set` (as a powerset extension of `operator==(bool,bool)` and the identity (expressed by `equals`). We decided to provide the former, for consistency with the semantic of all other operations on `bool_-set` which correspond to functions over `bool`. Identity testing is offered as a free and a member functions `equals`.

— **Why no** `operator<` **defined for** `bool_set`?
We believe that the `operator<` on bool is not used much, so there is very little need to provide such an operator. However, if we decided to do so, it should have the same kind of semantic as `operator==`.

— **Why no** `operator&&` **and** `operator||` **in addition to** `operator&` **and** `operator|`?
To preserve the short-circuiting idiom of `operator&&` and `operator||` which cannot be defined directly for UDTs. (note : we changed position since initial version of this document)

— **Why provide I/O**?
I/O is already provided with the (throwing) conversion to `bool`. We provide an I/O which reads and writes indeterminates and emptysets as well as booleans. This I/O does not throw. In alpha mode, a user can easily provide his/her own facet to overrule the default names or even to throw an exception for non-boolean values in alpha mode, if desired. Another behavior could be to throw for non-booleans in numerical (non-alphabetical) mode. We have chosen to always allow I/O in numerical mode.

— **Why choose 2 and 3 for the I/O of indeterminate and emptyset**?
There is no canonical value in addition to 0 and 1 for `false` and `true`, although yet many libraries use the value 2 for indeterminates (other choices are -1, 0 and 1 for false, indeterminate and true, but this is incompatible with `bool`).

V Proposed Text for the Standard

In Chapter 20, General utilities library.

In 20.2/1, add :

```
// 20.2.3, bool_set:
class bool_set;

// 20.2.3.2 bool_set values:
bool contains(bool_set, bool_set);
bool equals(bool_set, bool_set);
bool is_emptyset(bool_set);
bool is_indeterminate(bool_set);
bool is_singleton(bool_set);
bool certainly(bool_set);
bool possibly(bool_set);

// 20.2.3.3 bool_set set operations:
bool_set set_union(bool, bool_set);
bool_set set_union(bool_set, bool);
bool_set set_union(bool_set, bool_set);

bool_set set_intersection(bool, bool_set);
bool_set set_intersection(bool_set, bool);
bool_set set_intersection(bool_set, bool_set);

bool_set set_complement(bool_set);

// 20.2.3.4 bool_set logical operators:
bool_set operator!(bool_set);

bool_set operator^(bool, bool_set);
bool_set operator^(bool_set, bool);
bool_set operator^(bool_set, bool_set);

bool_set operator|(bool, bool_set);
bool_set operator|(bool_set, bool);
bool_set operator|(bool_set, bool_set);

bool_set operator&(bool, bool_set);
bool_set operator&(bool_set, bool);
bool_set operator&(bool_set, bool_set);

// 20.2.3.5 bool_set relational operators:
bool_set operator==(bool, bool_set);
bool_set operator==(bool_set, bool);
bool_set operator==(bool_set, bool_set);

bool_set operator!=(bool, bool_set);
bool_set operator!=(bool_set, bool);
bool_set operator!=(bool_set, bool_set);
```

20.2.3 Boolean set [lib.bool_set]

1 The type `bool_set` represents the power set of the Boolean set. An object of type `bool_set` represents therefore a set of boolean values and is thus one of the four distinct values `∅` (empty), `{false}`, `{true}`

^	empty	false	indet.	true
empty	empty	empty	empty	empty
false	empty	false	indet.	true
indet.	empty	indet.	indet.	indet.
true	empty	true	indet.	false

|	empty	false	indet.	true
empty	empty	empty	empty	empty
false	empty	false	indet.	true
indet.	empty	indet.	indet.	true
true	empty	true	true	true

&	empty	false	indet.	true
empty	empty	empty	empty	empty
false	empty	false	false	false
indet.	empty	false	indet.	indet.
true	empty	false	indet.	true

Table 1: The semantics of `operator^`, `operator|` and `operator&`.

==	empty	false	indet.	true
empty	empty	empty	empty	empty
false	empty	true	indet.	false
indet.	empty	indet.	indet.	indet.
true	empty	false	indet.	true

Table 2: The semantics of `operator==`.

and `{false,true}`. The singletons are identified with their boolean value. The value `{false,true}` is used to represent an unknown boolean and is named an *indeterminate*.

2 `bool_set` shares the interface of `bool` as much as possible, with the general semantic of functions being naturally extended to the operations on sets. The semantics of the logical `bool_set` operations are given in Table 1.

3 `bool_set` also supports some query functions as well as set operations. Equality as set is offered by the `equals` function, while the equality operator follows the extended semantic by returning a `bool_set`, see Table 2.

4 None of the member functions throws, nor do value and operation functions, except for the conversion to `bool`.

```
namespace std {

struct bool_set
{
  bool_set();
  bool_set(bool t);

  bool contains(bool_set b) const;
  bool equals(bool_set b) const;
  bool is_emptyset() const;
  bool is_indeterminate() const;
  bool is_singleton() const;

  operator bool() const;

  static bool_set emptyset();
  static bool_set indeterminate();
};

} // of namespace std
```

`bool_set` constructors [lib.bool_set.ctors]

```
bool_set();
```

5 **Effects:** Constructs a false `bool_set`.
6 **Postcondition:** `equals(false) == true`.

```
bool_set(bool b);
```

7 **Effects:** Constructs a boolean-valued `bool_set` equal to `b`.

`bool_set` member functions [lib.bool_set.members]

```
bool contains(bool_set b) const;
```

8 **Returns:** `true` iff `*this` contains the set of boolean values held by `b`.

```
bool equals(bool_set b) const;
```

9 **Returns:** `true` iff `*this` and `b` contain the same state.

```
bool is_emptyset() const;
```

10 **Returns:** `true` iff `*this` is the empty set.

```
bool is_indeterminate() const;
```

11 **Returns:** `true` iff `*this` is the indeterminate set.

```
bool is_singleton() const;
```

12 **Returns:** `true` iff `*this` contains only `true` or only `false`.

`bool_set` conversion to `bool` [lib.bool_set.conversion]

```
operator bool() const;
```

13 **Effects:** Returns `b` if `this->equals(b)`, with `b` being a boolean value.
14 **Throws:** `std::bad_cast()` otherwise.

`bool_set` set operations: [lib.bool_set.set.operations]

```
bool_set set_union(bool_set lhs, bool_set rhs);
bool_set set_union(bool_set lhs, bool rhs);
bool_set set_union(bool lhs, bool_set rhs);
```

15 **Returns:** the union of `lhs` and `rhs` when viewed as a set of `bool`s.

```
bool_set set_intersection(bool_set lhs, bool_set rhs);
bool_set set_intersection(bool_set lhs, bool rhs);
bool_set set_intersection(bool lhs, bool_set rhs);
```

16 **Returns:** the intersection of `lhs` and `rhs` when viewed as a set of `bool`s.

```
bool_set set_complement(bool_set x);
```

17 **Returns:** the complement of `x` when viewed as a set of `bool`s.

`bool_set` values [lib.bool_set.free]

```
bool contains(bool_set a, bool_set b);
```

18 **Returns:** `a.contains(b)`.

```
bool equals(bool_set a, bool_set b);
```

19 **Returns:** `a.equals(b)`.

```
bool is_emptyset(bool_set a);
```

20 **Returns:** `a.is_emptyset()`.

```
bool is_indeterminate(bool_set a);
```

21 **Returns:** `a.is_indeterminate()`.

```
bool is_singleton(bool_set a);
```

22 **Returns:** `a.is_singleton()`.

```
bool certainly(bool_set a);
```

23 **Returns:** `!a.contains(false)`.

```
bool possibly(bool_set a);
```

24 **Returns:** `a.contains(true)`.

`bool_set` logical operators [lib.bool_set.operators]

```
bool_set operator!(bool_set x);
```

25 **Returns:** `x` if `x` is empty or indeterminate, and `bool_set(!bool(x))` otherwise.

```
bool_set operator^(bool_set lhs, bool_set rhs);
bool_set operator^(bool_set lhs, bool rhs);
bool_set operator^(bool lhs, bool_set rhs);
```

26 **Returns:** an empty `bool_set` if any of `lhs` or `rhs` is empty, an indeterminate `bool_set` if any of `lhs` or `rhs` is indeterminate, else `bool(lhs) ^ bool(rhs)`. See Table 1.

```
bool_set operator|(bool_set lhs, bool_set rhs);
bool_set operator|(bool_set lhs, bool rhs);
bool_set operator|(bool lhs, bool_set rhs);
```

27 **Returns:** an empty `bool_set` if any of `lhs` or `rhs` is empty, else `true` if any of `lhs` or `rhs` is true, `false` if both `lhs` or `rhs` are false, and an indeterminate `bool_set` otherwise. See Table 1.

```
bool_set operator&(bool_set lhs, bool_set rhs);
bool_set operator&(bool_set lhs, bool rhs);
bool_set operator&(bool lhs, bool_set rhs);
```

28 **Returns:** an empty `bool_set` if any of `lhs` or `rhs` is empty, else `false` if any of `lhs` or `rhs` is false, `true` if both `lhs` or `rhs` are true, and an indeterminate `bool_set` otherwise. See Table 1.

`bool_set` relational operators [lib.bool_set.rel]

```
bool_set operator==(bool_set lhs, bool_set rhs);
bool_set operator==(bool_set lhs, bool rhs);
bool_set operator==(bool lhs, bool_set rhs);
```

29 **Returns:** an empty `bool_set` if any `lhs` or `rhs` is empty, else an indeterminate `bool_set` if any of `lhs` or `rhs` is indeterminate, and `bool(lhs) == bool(rhs)` otherwise. See Table 2.

```
bool_set operator!=(bool_set lhs, bool_set rhs);
bool_set operator!=(bool_set lhs, bool rhs);
bool_set operator!=(bool lhs, bool_set rhs);
```

30 **Returns:** `!( lhs == rhs )`

`bool_set` static value operations [lib.bool_set.static.values]

```
static bool_set emptyset();
```

31 **Returns:** an empty `bool_set`.

```
static bool_set indeterminate();
```

32 **Returns:** an indeterminate `bool_set`.

22.2.2 The numeric category [lib.category.numeric]

In Section 22.2.2.1, synopsis, add after the corresponding `bool&` overload:

```
iter_type get(iter_type in, iter_type end , ios_base&,
              ios_base::iostate& err, bool_set& v) const;
// ... and later, in: // virtual
virtual iter_type do_get(iter_type, iter_type, ios_base&,
              ios_base::iostate& err, bool_set& v) const;
```

Likewise, add in Section 22.2.2.1.1, immediately after the corresponding `bool&` overload:

```
iter_type get(iter_type in, iter_type end, ios_base& str,
              ios_base::iostate& err, bool_set& val) const;
```

And at the end of Section 22.2.2.2.1:

```
virtual iter_type do_get(iter_type in, iter_type end, ios_base& str,
                         ios_base::iostate& err, bool_set& val ) const;
```

10 **Effects:** If `(str.flags() & ios_base::boolalpha) == 0` then input proceeds as it would for a `bool` except that if a value is being stored into `val`, the value is determined according to the following: If the value to be stored is 0 then `false` is stored. If the value is 2 then `bool_set::indeterminate()` is stored. If the value is 1 then `true` is stored. If the value is 3 then `bool_set::emptyset()` is stored. Otherwise `err |= ios_base::failbit` is performed and no value is stored.

11 Otherwise target sequences are determined "as if" by calling the members `falsename()`, `truename()`, `emptysetname()`, and `indeterminatename()` of the facet obtained by `use_facet<numpunct<charT> >(str.getloc())`. Successive characters in the range `[in, end)` (see 23.1.1) are obtained and matched against corresponding positions in the target sequences in the same fashion and with the same behavior.

12 **Returns:** `in`

In Section 22.2.2.2, synopsis, add after the corresponding `bool` overload:

```
iter_type put(iter_type s, ios_base& f, char_type fill, bool_set v) const;
// ... and later, in: // virtual
virtual iter_type do_put(iter_type, ios_base&, char_type fill,
                         bool_set v) const;
```

Likewise, add in Section 22.2.2.2.1, immediately after the corresponding `bool` overload:

```
iter_type put(iter_type out, ios_base& str, char_type fill,
              bool_set val) const;
```

And at the end of Section 22.2.2.2.1:

```
virtual iter_type do_put(iter_type, ios_base&, char_type fill,
                         bool_set v) const;
```

7 **Returns:** If `(str.flags() & ios_base::boolalpha) == 0` returns `do_put(out , str , fill , is_singleton(val) ? bool(val) : (is_indeterminate(val) ? 2 : 3))`, otherwise obtains a string `s` as if by

```
string_type s = val.equals(true)  ? use_facet<ctype<charT> >(loc).truename() :
                val.equals(false) ? use_facet<ctype<charT> >(loc).falsename() :
       is_indeterminate(val) ? use_facet<ctype<charT> >(loc).indeterminatename()
      /* is_emptyset(val) */ : use_facet<ctype<charT> >(loc).emptysetname();
```

and then inserts each character `c` of `s` into out via `*out++ = c` and returns `out`.

22.2.3.1.1 numpunct members [lib.facet.numpunct.members]

Change paragraph:

```
string_type truename() const;
string_type falsename() const;
string_type indeterminatename() const;
string_type emptysetname() const;
```

4 **Returns:** `do_truename()`, `do_falsename()`, `do_indeterminatename()` or `do_emptysetname()`, respectively.

22.2.3.1.2 numpunct virtual functions [lib.facet.numpunct.virtuals]

Change paragraph:

```
string_type do_truename() const;
string_type do_falsename() const;
string_type do_indeterminatename() const;
string_type do_emptysetname() const;
```

5 **Returns:** A string representing the name of the boolean value `true` or `false`, or the `bool_set` values `bool_set::indeterminate()` or `bool_set::emptyset()` respectively.

6 In the base class implementation these names are `"true"`, `"false"`, `"indeterminate"`, and `"emptyset"`, or `L"true"`, `L"false"`, `L"indeterminate"`, and `L"emptyset"`.

V Examples of use

V.1 Interval arithmetic

In interval arithmetic, an interval $x = [\underline{x}, \overline{x}]$ represents the set of numbers $\{t : \underline{x} \leq t \leq \overline{x}\}$. We use the functions $\inf(x) = \underline{x}$ and $\sup(x) = \overline{x}$. Interval comparisons may be done in a number of ways.

One comparison scheme is the ‘possibly’ or ‘certainly’ comparison, with the semantic $\exists u \in x, v \in y, u < v$ for possibly, and $\forall u \in x, v \in y, u < v$ for certainly. Both can be handled uniformly by a single comparison operator returning a `bool_set`. Such an interval comparison `x<y` returns an empty `bool_set` if either x or y is empty, `true` if $\sup(x) < \inf(y)$, `false` if $\inf(y) \leq \sup(x)$, and an indeterminate otherwise. With this comparison, `certainly_lt(x,y)` is simply implemented as `equals(x<y, true)` and `possibly_-lt(x,y)` as `contains(x<y, true)`. Many other comparison schemes can be implemented in terms of this `operator<` with various negations, combinations, and parameter swapping. This is the original motivation behind having a type `bool_set` as return type of interval comparisons.

V.2 Ternary logic

This example is taken from Semple, a static analysis tool developed at Rensselaer Polytechnic Institute [6]. Semple uses `boost::tribool`, which is `bool_set` without the empty state, in several places. The `AbstractInterpreter` class uses 3-state boolean values to represent the values of boolean variables and to report the results of evaluating conditionals. Among the functions returning a `boost::tribool` are: `semple::analysis::pointer::TrivialPointsTo::null(Pointer p)` which returns a `tribool` encoding whether the pointer is always null, never null, or may be null. Another one in the same class is `semple::analysis::pointer::TrivialPointsTo::aliasing(Pointer p, Pointer q)` which determines the aliasing relationship between two pointers: if either of the pointer is unknown, then they may alias, otherwise, the pointers are either equivalent (and therefore must alias) or distinct (and therefore must not alias).

V.3 Extended floating-point comparisons

Suppose one desires a floating point comparison that takes into account exceptional values such as infinities, NaNs, or signed zeroes, where these concepts are used to represent a set of numbers (those larger—for infinities—or smaller—for signed zeroes—in magnitude than representable numbers, and empty sets for NaNs). Following a constructivist point of view that 0 is never defined because we do not have infinite time to verify all the decimals, we treat 0 as $0+$. For instance, one may desire that $+\infty == +\infty$ or $0- == 0$ return an indeterminate, rather than `true`, rather than a single number. With this semantic, one may code:

```
template <class FloatingPointType>
struct extended_less {
  typedef FloatingPointType first_argument_type;
  typedef FloatingPointType second_argument_type;
  typedef bool_set result_type;
  result_type operator() (first_argument_type, second_argument_type) const;
};
```

which obeys the following semantic where x and y are positive, finite, representable numbers of type `FloatingPointType`:

`extended_less(a,b)`	$b =$nan	$b = -\infty$	$b = -x$	$b = 0-$	$b = 0$	$b = y$	$b = +\infty$
$a =$nan	empty	empty	empty	empty	empty	empty	empty
$a = -\infty$	empty	indet.	`true`	`true`	`true`	`true`	`true`
$a = -x$	empty	`false`	indet.	`true`	`true`	`true`	`true`
$a = 0-$	empty	`false`	`false`	indet.	indet.	`true`	`true`
$a = 0$	empty	`false`	`false`	indet.	indet.	`true`	`true`
$a = y$	empty	`false`	`false`	`false`	`false`	indet.	`true`
$a = +\infty$	empty	`false`	`false`	`false`	`false`	`false`	indet.

Using this operator, one may write robust code (in the sense of the constructive theory of the reals) with floating point values, by handling special cases when a decision comes to an empty or indeterminates. For instance, one may trigger higher precision evaluation of the numbers, until all comparisons come out as boolean.

V.4 Multi-valued boolean algorithms

As `bool_set` is the powerset of bool, this is the natural result when applying a predicate to every element of a set. The following algorithm returns true if each element of the set satisfies the predicate, false if none of the elements satisfies it, and indeterminate if some elements satisfy it while others do not.

```
template<typename InputIterator, typename Predicate>
bool_set check_if(InputIterator first, InputIterator last, Predicate pred) {
  bool_set result = bool_set::emptyset();
  for (; first != last; ++first) {
    result = set_union(result, pred(*first));
    if (result.is_indeterminate()) break;
  }
  return result;
}
```

The same feature could be achieved by using `count_if` STL algorithm and then comparing its result with zero and with the size of the set. But the program would be a bit awkward and not as efficient since it would require applying the predicate to all the elements, even when the result is already known to be indeterminate.

V.5 Introducing non-determinism into C++ via `bool_set`

One intriguing possibility to introduce non-determinism using `bool_set` is to define a conversion from a `bool_set b` to `bool` returns a `bool` if `b.is_singleton()`, raises an exception if `b.is_emptyset()`, and which forks in case of indeterminate and returns true in one instantiation of the program and false in the other. Hence all execution paths with indeterminates would run in parallel and all possible execution paths would be eventually followed. This could be quite interesting for some applications (e.g. symbolic computations where the sign of a symbolic variable could not be formally determined at runtime, etc.)

One may even envision a language extension where a conversion in the context of a branch `if (bool_-set x) then A; else B;` would execute A if `x.equals(true)`, B if `x.equals(false)`, and neither or both if `x` is empty or indeterminate. Unlike the situation in the previous paragraph which can already be simulated in current C++, it is not clear how to simulate such a four-branch non-determinism. But this raises intriguing possibilities. This could require either to have `bool_set` as a builtin type and special treatment of `if` statements, or the more general possibility to "overload" the behavior of `if`.

VI Acknowledgements

We are grateful to the Boost community for its support, and the deep peer review of the Boost.Interval library, together with the reliable computing community, all of whose comments led to this proposal. We also would like to thank Douglas Gregor as author of the Boost.Tribool library.

References

[1] Anonymous. Ternary Logic - Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Ternary_logic`

[2] Anonymous. Setun - - Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Setun`

[3] H. Brönnimann, G. Melquiond, and S. Pion. A Proposal to add Interval Arithmetic to the C++ Standard Library. JTC1/SC22/WG21 - The C++ Standards Committee, Doc No: N1843=05-0103. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1843.pdf`

[4] H. Brönnimann, G. Melquiond, and S. Pion. The Boost interval arithmetic library. `http://www.boost.org/libs/numeric/interval/doc/interval.htm`

[5] Douglas Gregor. Boost.Tribool. `http://boost.org/doc/html/tribool.html`

[6] Douglas Gregor. Semple Analysis Engine Documentation. `http://www.cs.rpi.edu/~gregod/Semple/main.html`

[7] Various authors. Several U.S. patents of interest:
#3,129,340 (04/14/64): Logical and Memory Circuits Utilizing Tri-Level Signals.
#3,176,154 (03/30/65): Three State Memory Device.
#3,207,922 (09/21/65): Three Level Inverter and Latch Circuits.
#3,660,678 (05/02/72): Basic Ternary Logic Circuits.
#3,671,763 (06/20/72): Ternary Latches.
#3,671,764 (06/20/72): Auto-Reset Latches.
#3,909,634 (09/30/75): Three State Latch.

[8] CGAL. *Computational Geometry Algorithms Library*. `http://www.cgal.org/`