



Joachim Faulhaber

An Introduction to the Interval Template Library

Lecture
held at the Boost Library Conference 2009

2009-05-08

- Background and Motivation
- Design
- Examples
- Semantics
- Implementation
- Future Works
- Availability

- Interval containers simplified the implementation of date and time related tasks
 - Decomposing “*histories*” of attributed events into segments with constant attributes.
 - Working with time grids, e.g. a grid of months.
 - Aggregations of values associated to date or time intervals.
- ... that occurred frequently in programs like
 - Billing modules
 - Therapy scheduling programs
 - Hospital and controlling statistics

- Background is the date time problem domain ...
- ... but the scope of the **Itl** as a generic library is more general:

*an **interval_set** is a **set**
that is implemented as a set of intervals*

*an **interval_map** is a **map**
that is implemented as a map of interval value pairs*

- There are two aspects in the design of interval containers
- Conceptual aspect

```
interval_set<int> mySet;  
mySet.insert(42);  
bool has_answer = mySet.contains(42);
```

- On the conceptual aspect an interval_set can be used just as a set of elements
- except for . . .
- . . . *iteration over elements*
- consider interval_set<double> or interval_set<string>
- Iterative Aspect
 - **Iteration** is always done over *intervals*

• Addability and Subtractability

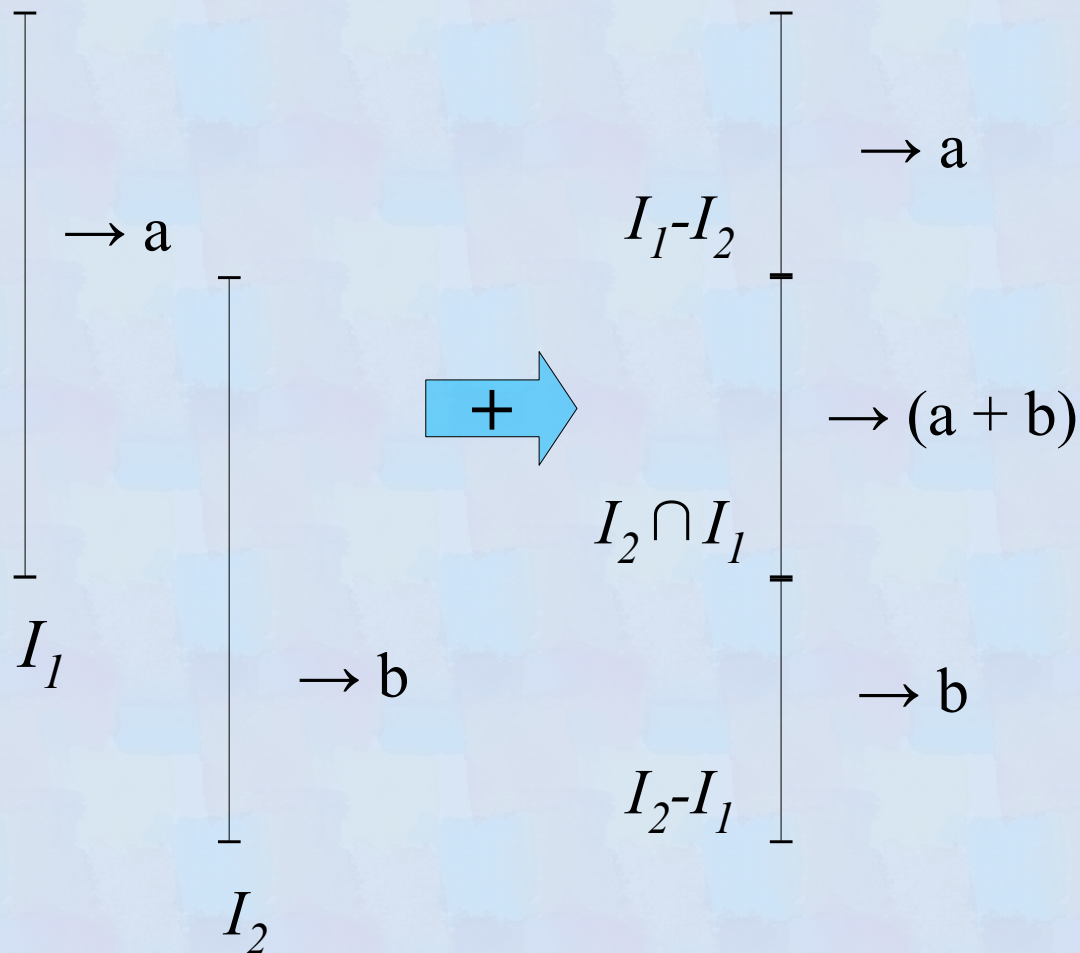
- All of itl's (interval) containers are *Addable* and *Subtractable*
- They implement **operators** $+=$, $+$, $-=$ and $-$

	$+=$	$-=$
sets	set union	set difference
maps	?	?

• A possible implementation for maps

- Propagate addition/subtraction to the associated values
- . . . or aggregate on overlap
- . . . or aggregate on collision

Aggregate on overlap



- Decompositional effect on Intervals
- Accumulative effect on associated values

Aggregate on overlap, a minimal example

```
typedef itl::set<string> guests;
interval_map<time, guests> party;

party += make_pair(
    interval<time>::rightopen(20:00, 22:00), guests("Mary"));

party += make_pair(
    interval<time>::rightopen(21:00, 23:00), guests("Harry"));

// party now contains
[20:00, 21:00) -> {"Mary"}
[21:00, 22:00) -> {"Harry", "Mary"} //guest sets aggregated
[22:00, 23:00) -> {"Harry"}
```


• The Itl's class templates

Granularity	Style	Sets	Maps
interval		<code>interval</code>	
	joining	<code>interval_set</code>	<code>interval_map</code>
	separating	<code>separate_interval_set</code>	
	splitting	<code>split_interval_set</code>	<code>split_interval_map</code>
element		<code>set</code>	<code>map</code>

Interval Combining Styles: *Joining*

- Intervals are joined on overlap or on touch
- ... *for maps*, if associated values are equal
- Keeps interval_maps and sets in a minimal form

interval_set

```

{ [1      3)
+   [2      4)
+       [4 5)

= { [1      4)
}

= { [1      5) }
```

interval_map

```

{ [1      3) ->1
+   [2      4) ->1
+       [4 5) ->1

={ [1 2) [2 3) [3 4)
   ->1  ->2  ->1
}

={ [1 2) [2 3) [3      5)
   ->1  ->2      ->1
}
```


Interval Combining Styles: *Splitting*

- Intervals are split on overlap and kept separate on touch
- All interval borders are preserved (insertion memory)

split_interval_set

```

{ [1      3)      }
+   [2      4)
+   [4  5)

= { [1  2) [2  3) [3  4)      }
= { [1  2) [2  3) [3  4) [4  5) }

```

split_interval_map

```

{ [1      3) ->1      }
+   [2      4) ->1
+   [4  5) ->1

= { [1  2) [2  3) [3  4)      }
   ->1   ->2   ->1
= { [1  2) [2  3) [3  4) [4  5) }
   ->1   ->2   ->1   ->1

```

Interval Combining Styles: *Separating*

- Intervals are joined on overlap but kept separate on touch
- Preserves borders that are never crossed (preserves a hidden grid).

separate_interval_set

$$\begin{aligned}
 & \{ [1 \quad 3) \\
 + & \quad [2 \quad 4) \\
 + & \quad \quad [4 \quad 5) \\
 = & \{ [1 \quad 4) \\
 = & \{ [1 \quad 4) [4 \quad 5) \}
 \end{aligned}$$

● A few instances of intervals (interval.cpp)

```
interval<int> int_interval = interval<int>::closed(3,7);

interval<double> sqrt_interval
    = interval<double>::rightopen(1/sqrt(2.0), sqrt(2.0));

interval<std::string> city_interval
    = interval<std::string>::leftopen("Barcelona", "Boston");

interval<boost::ptime> time_interval
    = interval<boost::ptime>::open(
        time_from_string("2008-05-20 19:30"),
        time_from_string("2008-05-20 23:00")
    );
```

● A way to iterate over months and weeks (month_and_week_grid.cpp)

```
#include <boost/itl/gregorian.hpp> //boost::gregorian plus adapter code
#include <boost/itl/split_interval_set.hpp>

// A split_interval_set of gregorian dates as date_grid.
typedef split_interval_set<boost::gregorian::date> date_grid;

// Compute a date_grid of months using boost::gregorian.
date_grid month_grid(const interval<date>& scope)
{
    date_grid month_grid;
    // Compute a date_grid of months using boost::gregorian.
    . . .
    return month_grid;
}

// Compute a date_grid of weeks using boost::gregorian.
date_grid week_grid(const interval<date>& scope)
{
    date_grid week_grid;
    // Compute a date_grid of weeks using boost::gregorian.
    . . .
    return week_grid;
}
```


● A way to iterate over months and weeks

```
void month_and_time_grid()
{
    date someday = day_clock::local_day();
    date thenday = someday + months(2);
    interval<date> scope = interval<date>::rightopen(someday, thenday);

    // An intersection of the month and week grids ...
    date_grid month_and_week_grid
        = month_grid(scope) & week_grid(scope);

    // ... allows to iterate months and weeks. Whenever a month
    // or a week changes there is a new interval.
    for(date_grid::iterator it = month_and_week_grid.begin();
        it != month_and_week_grid.end(); it++)
    {
        . . .
    }

    // We can also intersect the grid into an interval_map to make
    // shure that all intervals are within months and week bounds.
    interval_map<boost::gregorian::date, some_type> accrual;
    compute_some_result(accrual, scope);
    accrual &= month_and_week_grid;
}
```

• Aggregating with interval_maps

- Computing averages via implementing **operator +=**
(partys_guest_average.cpp)

```
class counted_sum
{
public:
    counted_sum() : _sum(0), _count(0) {}
    counted_sum(int sum) : _sum(sum), _count(1) {}

    int sum() const {return _sum;}
    int count() const {return _count;}
    double average() const
    { return _count==0 ? 0.0 : _sum/static_cast<double>(_count); }

    counted_sum& operator += (const counted_sum& right)
    { _sum += right.sum(); _count += right.count(); return *this; }

private:
    int _sum;
    int _count;
};

bool operator == (const counted_sum& left, const counted_sum& right)
{ return left.sum()==right.sum() && left.count()==right.count(); }
```


Aggregating with interval_maps

Computing averages via implementing **operator +=**

```
void partys_height_average()
{
    interval_map<ptime, counted_sum> height_sums;

    height_sums += (
        make_pair(
            interval<ptime>::rightopen(
                time_from_string("2008-05-20 19:30"),
                time_from_string("2008-05-20 23:00")),
            counted_sum(165)) // Mary is 1,65 m tall.
    );

    // Add height of more pary guests . . .

    interval_map<ptime, counted_sum>::iterator height_sum_ =
        height_sums.begin();
    while(height_sum_ != height_sums.end())
    {
        interval<ptime> when = height_sum_->first;
        double height_average = (*height_sum_++).second.average();

        cout << "[" << when.first() << " - " << when.upper() << "]"
             << ": " << height_average << " cm" << endl;
    }
}
```

- Interval containers allow to express a variety of date and time operations in an easy way.
 - Example `man_power.cpp` ...
 - Subtract weekends and holidays from an `interval_set`
`worktime -= weekends(scope)`
`worktime -= german_reunification_day`
 - Intersect an `interval_map` with an `interval_set`
`claudias_working_hours &= worktime`
 - Subtract an `interval_set` from an `interval_map`
`claudias_working_hours -= claudias_absense_times`
 - Adding `interval_maps`
`interval_map<date, int> manpower;`
`manpower += claudias_working_hours;`
`manpower += bodos_working_hours;`

Interval_maps can also be intersected Example `user_groups.cpp`

```
typedef boost::itl::set<string> MemberSetT;
typedef interval_map<date, MemberSetT> MembershipT;

void user_groups()
{
    . . .

    MembershipT med_users;
    // Compute membership of medical staff
    med_users += make_pair(member_interval_1, MemberSetT("Dr.Jekyll"));
    med_users += . . .

    MembershipT admin_users;
    // Compute membership of administration staff
    med_users += make_pair(member_interval_2, MemberSetT("Mr.Hyde"));
    . . .

    MembershipT all_users    = med_users + admin_users;

    MembershipT super_users = med_users & admin_users;
    . . .

}
```

- The semantics of *itl sets* is based on a concept `itl::Set`
 - `itl::set`, `interval_set`, `split_interval_set` and `separate_interval_set` are models of concept `itl::Set`

```
// Abstract part
empty set:      Set::Set()
subset relation: bool Set::contained_in(const Set& s2) const
equality:       bool is_element_equal(const Set& s1, const Set& s2)
set union:      Set& operator += (Set& s1, const Set& s2)
                Set operator + (const Set& s1, const Set& s2)
set difference: Set& operator -= (Set& s1, const Set& s2)
                Set operator - (const Set& s1, const Set& s2)
set intersection: Set& operator &= (Set& s1, const Set& s2)
                Set operator & (const Set& s1, const Set& s2)

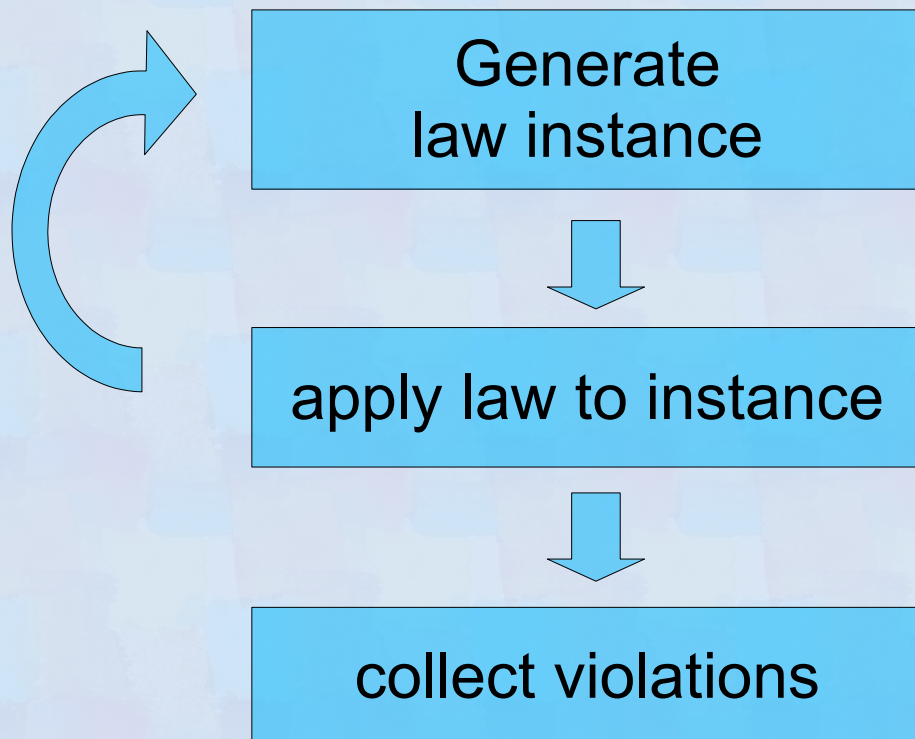
// Part related to sequential ordering
sorting order:  bool operator < (const Set& s1, const Set& s2)
lexicographical equality:
                bool operator == (const Set& s1, const Set& s2)
```


- The semantics of *itl maps* is based on a concept `itl::Map`
 - `itl::map`, `interval_map` and `split_interval_map` are models of concept `itl::Map`

```
// Abstract part
empty map:      Map::Map()
submap relation: bool Map::contained_in(const Map& m2) const
equality:       bool is_element_equal(const Map& m1, const Map& m2)
map union:      Map& operator += (Map& m1, const Map& m2)
                Map& operator +  (const Map& m1, const Map& m2)
map difference: Map& operator -= (Map& m1, const Map& m2)
                Map& operator -  (const Map& m1, const Map& m2)
map intersection: Map& operator &= (Map& m1, const Map& m2)
                Map& operator &  (const Map& m1, const Map& m2)

// Part related to sequential ordering
sorting order:  bool operator <  (const Map& m1, const Map& m2)
lexicographical equality:
                bool operator == (const Map& m1, const Map& m2)
```

- Defining semantics of itl concepts via sets of laws
 - aka c++0x axioms
- Checking law sets via automatic testing:
 - A **Law Based Test Automaton** **LaBatea**



Commutativity<T a, U b, +>:
 $a + b = b + a;$

• Lexicographical Ordering and Equality

- For all itl containers `operator <` implements a ***strict weak ordering***.
- The ***induced equivalence*** of this ordering is ***lexicographical equality*** which is implemented as `operator ==`
- This is in line with the semantics of `SortedAssociativeContainers`

• Subset Ordering and Element Equality

- For all itl containers function `contained_in` implements a *partial ordering*.
- The *induced equivalence* of this ordering is *equality of elements* which is implemented as function `is_element_equal`.

- itl::Sets
- **All** itl sets implement a **Set Algebra**, which is to say satisfy a “*classical*” set of laws . . .
 - . . . using `is_element_equal` as equality
 - Associativity, Neutrality, Commutativity (for `+` and `&`)
 - Distributivity, DeMorgan, Symmetric Difference
- **Most of** the itl sets satisfy the classical set of laws even if . . .
 - . . . lexicographical equality: `operator ==` is used
 - The differences reflect proper inequalities in sequence that occur for `separate_interval_set` and `split_interval_set`.

- Concepts induction / concept transition
 - The semantics of itl Maps appears to be *determined* by the *codomain type* of the map
 - Itl Maps are *mapping* the semantics of the *codomain type* on themselves.

is model of example

Map<D, Monoid>	Monoid	interval_map<int, string>
Map<D, CommutMonoid>	CommutMonoid	interval_map<int, unsigned>
Map<D, AbelianGroup>	AbelianGroup	interval_map<int, int>
Map<D, Set>	Set	interval_map<int, set<int>>

- Itl containers are implemented simply based on `std::set` and `std::map`
 - Basic operations like *adding* and *subtracting* intervals have a *best case complexity of $O(\lg n)$* , if the added or subtracted intervals are *relatively small*.
 - Worst case complexity of *adding* or *subtracting* intervals *for `interval_set`* is $O(n)$.
 - For all other interval containers *adding* or *subtracting* intervals has a *worst case performance of $O(n \lg(n))$* .
 - There is a *potential* for optimization . . .

- A **segment_tree** implementation: A balanced tree, where . . .
 - an interval represents a perfectly balanced subtree
 - large intervals are rotated towards the root
- First results
 - much better worst case performance $O(n)$ instead of $O(n \lg(n))$
 - but slower for best case due to heavier bookkeeping and recursive algorithms.

- Completing and optimizing the segment_tree implementation of interval containers
- Implementing interval_maps of sets more efficiently
- Revision of features of the extended itl (itl_plus.zip)
 - **Decomposition of histories**: k histories h_k with attribute types A_1, \dots, A_k are “*decomposed*” to a product history of tuples of attribute sets:
$$(h_1\langle T, A_1 \rangle, \dots, h_k\langle T, A_k \rangle) \rightarrow h\langle T, (\text{set}\langle A_1 \rangle, \dots, \text{set}\langle A_k \rangle) \rangle$$
 - **Cubes** (generalized crosstables): Applying *aggregate on collision* to **maps of tuple value pairs** in order to organize hierarchical data and their aggregates.

- Itl project on **sourceforge** (version 2.0.1)
<http://sourceforge.net/projects/itl>
- Latest version on **boost vault/Containers** (3.0.0)
<http://www.boostpro.com/vault/> → containers
 - itl.zip : Core itl in preparation for boost
 - itl_plus.zip : Extended itl including product histories, cubes and automatic validation (LaBatea).
- **Online documentation** at
<http://www.herold-faulhaber.de/>
 - Doxygen generated docs for (version 2.0.1)
<http://www.herold-faulhaber.de/itl/>
 - Latest boost style documentation (version 3.0.0)
http://www.herold-faulhaber.de/boost_itl/doc/libs/itl/doc/html/

■ Boost sandbox

<https://svn.boost.org/svn/boost/sandbox/itl/>

- Core itl: Interval containers preparing for boost

<https://svn.boost.org/svn/boost/sandbox/itl/boost/itl/>

<https://svn.boost.org/svn/boost/sandbox/itl/libs/itl/>

- Extended itl_xl: “histories” and cubes

https://svn.boost.org/svn/boost/sandbox/itl/boost/itl_xt/


https://svn.boost.org/svn/boost/sandbox/itl/libs/itl_xt/

- Validator LaBatea: Currently only vc8 or newer

<https://svn.boost.org/svn/boost/sandbox/itl/boost/validate/>

<https://svn.boost.org/svn/boost/sandbox/itl/libs/validate/>

■ jo.faulhaber@gmail.com



Joachim Faulhaber

An Introduction to the Interval Template Library

Lecture
held at the Boost Library Conference 2009

2009-05-08

Copyright © Joachim Faulhaber 2009
Distributed under Boost Software Licence 1.0

Slide Design by Chih-Hao Tsai
<http://www.chtsai.org>

Lecture Outline

- Background and Motivation
- Design
- Examples
- Semantics
- Implementation
- Future Works
- Availability

Background and Motivation

- Interval containers simplified the implementation of date and time related tasks
 - Decomposing “*histories*” of attributed events into segments with constant attributes.
 - Working with time grids, e.g. a grid of months.
 - Aggregations of values associated to date or time intervals.
- ... that occurred frequently in programs like
 - Billing modules
 - Therapy scheduling programs
 - Hospital and controlling statistics

- Background is the date time problem domain ...
- ... but the scope of the **ItI** as a generic library is more general:

an **interval_set** is a **set**
that is implemented as a set of intervals

an **interval_map** is a **map**
that is implemented as a map of interval value pairs

- There are two aspects in the design of interval containers
- Conceptual aspect

```
interval_set<int> mySet;  
mySet.insert(42);  
bool has_answer = mySet.contains(42);
```

- On the conceptual aspect an interval_set can be used just as a set of elements
- except for . . .
- . . . **iteration** over *elements*
- consider interval_set<double> or interval_set<string>
- Iterative Aspect
 - **Iteration** is always done over *intervals*

- Addability and Subtractability

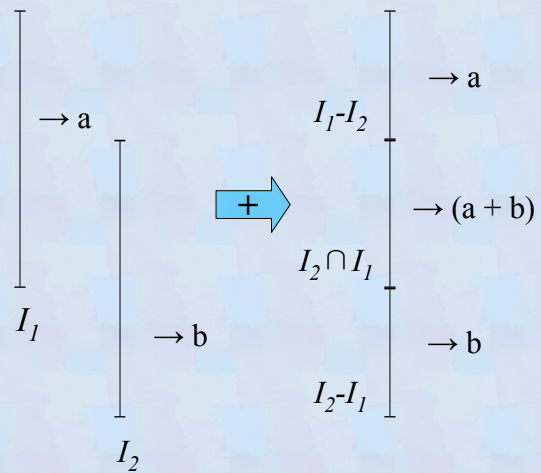
- All of itl's (interval) containers are *Addable* and *Subtractable*
- They implement **operators** `+=`, `+`, `-=` and `-`

	<code>+=</code>	<code>-=</code>
sets	set union	set difference
maps	?	?

- A possible implementation for maps

- Propagate addition/subtraction to the associated values
- . . . or aggregate on overlap
- . . . or aggregate on collision

● Aggregate on overlap



- Decompositional effect on Intervals
- Accumulative effect on associated values

● Aggregate on overlap, a minimal example

```
typedef itl::set<string> guests;
interval_map<time, guests> party;

party += make_pair(
    interval<time>::rightopen(20:00, 22:00), guests("Mary"));

party += make_pair(
    interval<time>::rightopen(21:00, 23:00), guests("Harry"));

// party now contains
[20:00, 21:00)->{"Mary"}
[21:00, 22:00)->{"Harry", "Mary"} //guest sets aggregated
[22:00, 23:00)->{"Harry"}
```

- The Itl's class templates

Granularity	Style	Sets	Maps
interval		<code>interval</code>	
	joining	<code>interval_set</code>	<code>interval_map</code>
	separating	<code>separate_interval_set</code>	
	splitting	<code>split_interval_set</code>	<code>split_interval_map</code>
element		<code>set</code>	<code>map</code>

● Interval Combining Styles: *Joining*

- Intervals are joined on overlap or on touch
- ... *for maps*, if associated values are equal
- Keeps interval_maps and sets in a minimal form

interval_set

```
{ [1      3)      ]
+   [2      4)
+           [4 5)
= { [1      4)      ]
= { [1      5) }
```

interval_map

```
{ [1      3) ->1      ]
+   [2      4) ->1
+           [4 5) ->1
= { [1 2) [2 3) [3 4)      ]
    ->1  ->2  ->1
= { [1 2) [2 3) [3      5)      ]
    ->1  ->2      ->1
```

● Interval Combining Styles: *Splitting*

- Intervals are split on overlap and kept separate on touch
- All interval borders are preserved (insertion memory)

`split_interval_set`

```
{ [1      3)      ]
+   [2      4)
+           [4 5)

= { [1 2) [2 3) [3 4)      ]
= { [1 2) [2 3) [3 4) [4 5) }
```

`split_interval_map`

```
{ [1      3) ->1      ]
+   [2      4) ->1
+           [4 5) ->1

= { [1 2) [2 3) [3 4)      ]
   ->1 ->2 ->1
= { [1 2) [2 3) [3 4) [4 5) }
   ->1 ->2 ->1 ->1
```


- Interval Combining Styles: *Separating*

- Intervals are joined on overlap but kept separate on touch
- Preserves borders that are never crossed (preserves a hidden grid).

```
separate_interval_set
```

```
{ [1      3)      }
+      [2      4)
+              [4 5)
= { [1      4)      }
= { [1      4) [4 5) }
```

● A few instances of intervals (interval.cpp)

```
interval<int> int_interval = interval<int>::closed(3,7);

interval<double> sqrt_interval
    = interval<double>::rightopen(1/sqrt(2.0), sqrt(2.0));

interval<std::string> city_interval
    = interval<std::string>::leftopen("Barcelona", "Boston");

interval<boost::ptime> time_interval
    = interval<boost::ptime>::open(
        time_from_string("2008-05-20 19:30"),
        time_from_string("2008-05-20 23:00")
    );
```


● A way to iterate over months and weeks (month_and_week_grid.cpp)

```
#include <boost/itl/gregorian.hpp> //boost::gregorian plus adapter code
#include <boost/itl/split_interval_set.hpp>

// A split_interval_set of gregorian dates as date_grid.
typedef split_interval_set<boost::gregorian::date> date_grid;

// Compute a date_grid of months using boost::gregorian.
date_grid month_grid(const interval<date>& scope)
{
    date_grid month_grid;
    // Compute a date_grid of months using boost::gregorian.
    . . .
    return month_grid;
}

// Compute a date_grid of weeks using boost::gregorian.
date_grid week_grid(const interval<date>& scope)
{
    date_grid week_grid;
    // Compute a date_grid of weeks using boost::gregorian.
    . . .
    return week_grid;
}
```

● A way to iterate over months and weeks

```
void month_and_time_grid()
{
    date someday = day_clock::local_day();
    date thenday = someday + months(2);
    interval<date> scope = interval<date>::rightopen(someday, thenday);

    // An intersection of the month and week grids ...
    date_grid month_and_week_grid
        = month_grid(scope) & week_grid(scope);

    // ... allows to iterate months and weeks. Whenever a month
    // or a week changes there is a new interval.
    for(date_grid::iterator it = month_and_week_grid.begin();
        it != month_and_week_grid.end(); it++)
    {
        . . .
    }

    // We can also intersect the grid into an interval_map to make
    // shure that all intervals are within months and week bounds.
    interval_map<boost::gregorian::date, some_type> accrual;
    compute_some_result(accrual, scope);
    accrual &= month_and_week_grid;
}
```

● Aggregating with interval_maps

- Computing averages via implementing **operator +=**
(partys_guest_average.cpp)

```
class counted_sum
{
public:
    counted_sum(): _sum(0), _count(0) {}
    counted_sum(int sum): _sum(sum), _count(1) {}

    int sum() const {return _sum;}
    int count() const {return _count;}
    double average() const
    { return _count==0 ? 0.0 : _sum/static_cast<double>(_count); }

    counted_sum& operator += (const counted_sum& right)
    { _sum += right.sum(); _count += right.count(); return *this; }

private:
    int _sum;
    int _count;
};

bool operator == (const counted_sum& left, const counted_sum& right)
{ return left.sum()==right.sum() && left.count()==right.count(); }
```


● Aggregating with interval_maps

■ Computing averages via implementing **operator +=**

```
void partys_height_average()
{
    interval_map<ptime, counted_sum> height_sums;

    height_sums += (
        make_pair(
            interval<ptime>::rightopen(
                time_from_string("2008-05-20 19:30"),
                time_from_string("2008-05-20 23:00")),
            counted_sum(165)) // Mary is 1,65 m tall.
    );

    // Add height of more pary guests . . .

    interval_map<ptime, counted_sum>::iterator height_sum_ =
        height_sums.begin();
    while(height_sum_ != height_sums.end())
    {
        interval<ptime> when = height_sum_>first;
        double height_average = (*height_sum_++).second.average();

        cout << "[" << when.first() << " - " << when.upper() << "]" << endl;
        cout << ": " << height_average << " cm" << endl;
    }
}
```

- Interval containers allow to express a variety of date and time operations in an easy way.
 - Example `man_power.cpp` ...
 - Subtract weekends and holidays from an `interval_set`
`worktime -= weekends(scope)`
`worktime -= german_reunification_day`
 - Intersect an `interval_map` with an `interval_set`
`claudias_working_hours &= worktime`
 - Subtract an `interval_set` from an `interval_map`
`claudias_working_hours -= claudias_absense_times`
 - Adding `interval_maps`
`interval_map<date,int> manpower;`
`manpower += claudias_working_hours;`
`manpower += bodos_working_hours;`

Interval_maps can also be intersected

Example `user_groups.cpp`

```
typedef boost::itl::set<string> MemberSetT;
typedef interval_map<date, MemberSetT> MembershipT;

void user_groups()
{
    . . .

    MembershipT med_users;
    // Compute membership of medical staff
    med_users += make_pair(member_interval_1, MemberSetT("Dr.Jekyll"));
    med_users += . . .

    MembershipT admin_users;
    // Compute membership of administration staff
    med_users += make_pair(member_interval_2, MemberSetT("Mr.Hyde"));
    . . .

    MembershipT all_users = med_users + admin_users;

    MembershipT super_users = med_users & admin_users;
    . . .
}
```


- The semantics of *itl sets* is based on a concept `itl::Set`
 - `itl::set`, `interval_set`, `split_interval_set` and `separate_interval_set` are models of concept `itl::Set`

```
// Abstract part
empty set:      Set::Set()
subset relation: bool Set::contained_in(const Set& s2) const
equality:       bool is_element_equal(const Set& s1, const Set& s2)
set union:      Set& operator += (Set& s1, const Set& s2)
                Set& operator +  (const Set& s1, const Set& s2)
set difference: Set& operator -= (Set& s1, const Set& s2)
                Set& operator -  (const Set& s1, const Set& s2)
set intersection: Set& operator &= (Set& s1, const Set& s2)
                Set& operator &  (const Set& s1, const Set& s2)

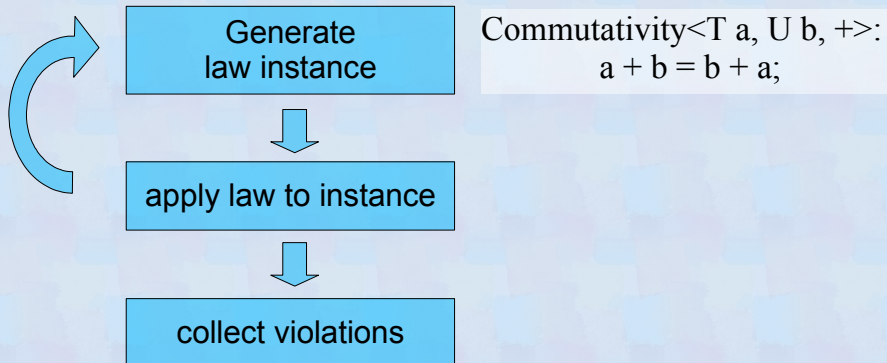
// Part related to sequential ordering
sorting order:  bool operator <  (const Set& s1, const Set& s2)
lexicographical equality:
                bool operator == (const Set& s1, const Set& s2)
```

- The semantics of *itl maps* is based on a concept `itl::Map`
 - `itl::map`, `interval_map` and `split_interval_map` are models of concept `itl::Map`

```
// Abstract part
empty map:      Map::Map()
submap relation: bool Map::contained_in(const Map& m2) const
equality:       bool is_element_equal(const Map& m1, const Map& m2)
map union:      Map& operator += (Map& m1, const Map& m2)
                Map& operator +  (const Map& m1, const Map& m2)
map difference: Map& operator -= (Map& m1, const Map& m2)
                Map& operator -  (const Map& m1, const Map& m2)
map intersection: Map& operator &= (Map& m1, const Map& m2)
                Map& operator &  (const Map& m1, const Map& m2)

// Part related to sequential ordering
sorting order:  bool operator <  (const Map& m1, const Map& m2)
lexicographical equality:
                bool operator == (const Map& m1, const Map& m2)
```

- Defining semantics of itl concepts via sets of laws
 - aka c++0x axioms
- Checking law sets via automatic testing:
 - A **Law Based Test Automaton** *LaBatea*



- Lexicographical Ordering and Equality

- For all itl containers `operator <` implements a **strict weak ordering**.
- The **induced equivalence** of this ordering is **lexicographical equality** which is implemented as `operator ==`
- This is in line with the semantics of SortedAssociativeContainers

- Subset Ordering and Element Equality
 - For all itl containers function `contained_in` implements a *partial ordering*.
 - The *induced equivalence* of this ordering is *equality of elements* which is implemented as function `is_element_equal`.

- itl::Sets
- All itl sets implement a **Set Algebra**, which is to say satisfy a “classical” set of laws . . .
 - . . . using `is_element_equal` as equality
 - Associativity, Neutrality, Commutativity (for `+` and `&`)
 - Distributivity, DeMorgan, Symmetric Difference
- Most of the itl sets satisfy the classical set of laws even if . . .
 - . . . lexicographical equality: `operator ==` is used
 - The differences reflect proper inequalities in sequence that occur for `separate_interval_set` and `split_interval_set`.

- Concepts induction / concept transition
 - The semantics of Itl Maps appears to be *determined* by the *codomain type* of the map
 - Itl Maps are *mapping* the semantics of the *codomain type on themselves*.

is model of example

Map<D, Monoid>	Monoid	interval_map<int, string>
Map<D, CommutMonoid>	CommutMonoid	interval_map<int, unsigned>
Map<D, AbelianGroup>	AbelianGroup	interval_map<int, int>
Map<D, Set>	Set	interval_map<int, set<int>>

- Itl containers are implemented simply based on `std::set` and `std::map`
 - Basic operations like *adding* and *subtracting* intervals have a *best case complexity of $O(\lg n)$* , if the added or subtracted intervals are *relatively small*.
 - Worst case complexity of *adding* or *subtracting* intervals *for `interval_set`* is $O(n)$.
 - For all other interval containers *adding* or *subtracting* intervals has a *worst case performance of $O(n \lg(n))$* .
 - There is a *potential* for optimization . . .

- A **segment_tree** implementation: A balanced tree, where . . .
 - an interval represents a perfectly balanced subtree
 - large intervals are rotated towards the root
- First results
 - much better worst case performance $O(n)$ instead of $O(n \lg(n))$
 - but slower for best case due to heavier bookkeeping and recursive algorithms.

- Completing and optimizing the segment_tree implementation of interval containers
- Implementing interval_maps of sets more efficiently
- Revision of features of the extended itl (itl_plus.zip)
 - **Decomposition of histories**: k histories h_k with attribute types A_1, \dots, A_k are “*decomposed*” to a product history of tuples of attribute sets:
 $(h_1 \langle T, A_1 \rangle, \dots, h_k \langle T, A_k \rangle) \rightarrow h \langle T, (\text{set} \langle A_1 \rangle, \dots, \text{set} \langle A_k \rangle) \rangle$
 - **Cubes** (generalized crosstables): Applying *aggregate on collision* to *maps of tuple value pairs* in order to organize hierarchical data and their aggregates.

- Itl project on **sourceforge** (version 2.0.1)
<http://sourceforge.net/projects/itl>
- Latest version on **boost vault/Containers** (3.0.0)
<http://www.boostpro.com/vault/> → containers
 - itl.zip : Core itl in preparation for boost
 - itl_plus.zip : Extended itl including product histories, cubes and automatic validation (LaBatea).
- **Online documentation** at
<http://www.herold-faulhaber.de/>
 - Doxygen generated docs for (version 2.0.1)
<http://www.herold-faulhaber.de/itl/>
 - Latest boost style documentation (version 3.0.0)
http://www.herold-faulhaber.de/boost_itl/doc/libs/itl/doc/html/

- Boost sandbox

<https://svn.boost.org/svn/boost/sandbox/itl/>

- Core itl: Interval containers preparing for boost
<https://svn.boost.org/svn/boost/sandbox/itl/boost/itl/>
<https://svn.boost.org/svn/boost/sandbox/itl/libs/itl/>
- Extended itl_xl: “histories” and cubes
https://svn.boost.org/svn/boost/sandbox/itl/boost/itl_xt/
https://svn.boost.org/svn/boost/sandbox/itl/libs/itl_xt/
- Validator LaBatea: Currently only vc8 or newer
<https://svn.boost.org/svn/boost/sandbox/itl/boost/validate/>
<https://svn.boost.org/svn/boost/sandbox/itl/libs/validate/>

● jo.faulhaber@gmail.com