JOACHIM FAULHABER

Boost.Alabaster
A Law Based Tester

LECTURE HELD AT THE
4TH INTERNATIONAL CONFERENCE
ON BOOST C++ LIBRARIES
BOOSTCON 2010

Version 1.0.0
2010-04-30

A Short Definition

Law Based Testing is automated testing of properties or laws that are assumed to be valid for a program.

A Short Historie

- A test tool developed along with the ITL-library of interval containers
- A prototype called LaBatea: Law Based Test Automaton
- An application exploiting polymorphic tuples based on typelists, inspired by Andrej Alexandrescu 2001: Modern c++ Design,
- Currently available in non-boost quality, as a prototype:
  Sandbox: https://svn.boost.org/svn/boost/sandbox/itl/boost/validate/
- And as part of the extended ITL releases 3.2.x
  Boost-Vault: itl_plus_3_2_0.zip
  Sourceforge: http://sourceforge.net/projects/itl/

Experiences and Outlook

- As a test tool for the ITL-library, it turned out that law based testing …
- … was *very* effective for refactoring and unit tests
- … fostered abstraction and quality in the development process
- … contained challenging abstraction tasks as a library
- … attracted interest in the boost community
- … has the potential for a future boost library in a currently underdeveloped field.
- The name Boost.Alabaster is the label for that future project.

Today

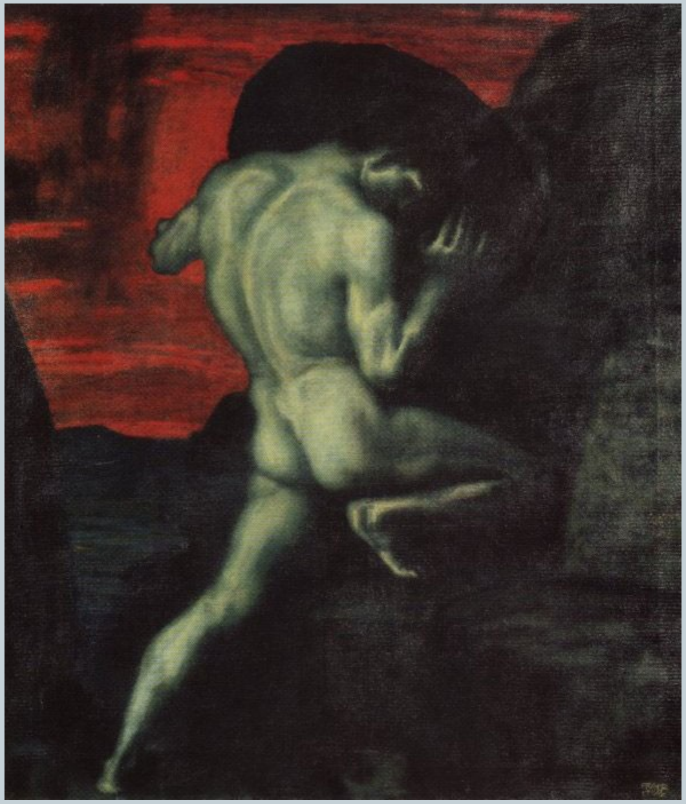
- Today I will introduce the current prototype
- that is not yet boost compliant but Loki biased
- For the most part I will show *what I have*
- … in order to make the ideas of law based testing clear

- Then I will outline some aspects of a redesigned and boost-quality library Boost.Alabaster as a project to go about.

Motivation

Testing in *(not so)* ancient times

- … is a Sisyphean Task
- … is limited by the frame of knowledge of the tester
- … is inherently ineffective and leads to frustration

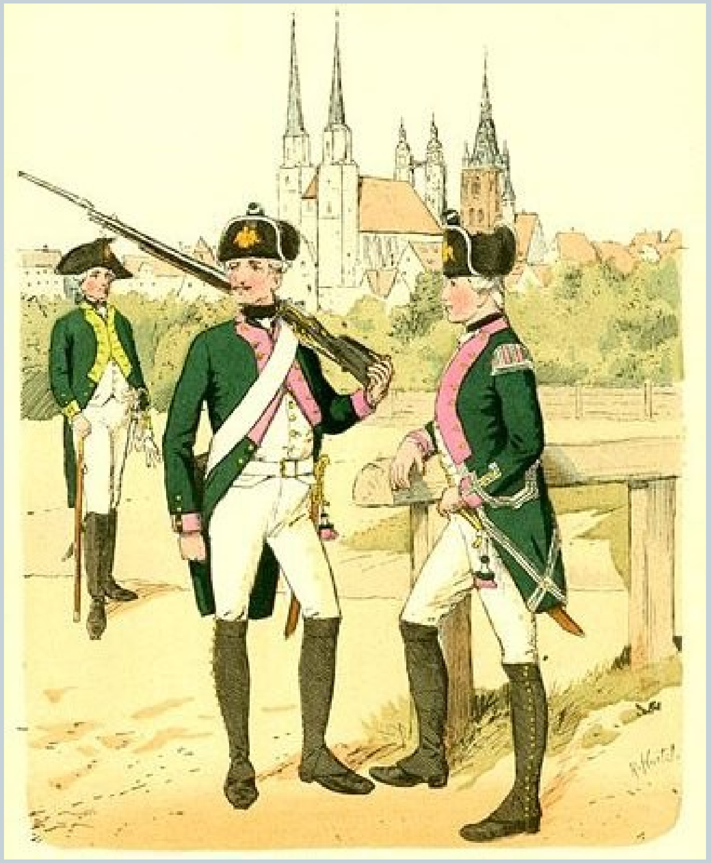


Sisyphus by Franz von Stuck, 1920 Source: WikiMedia Commons

Motivation

Testing in *(more or less)* modern times

- … is automated.
- Testing tools allow for unit tests
- Developers use those, writing tests first and maintaining them vigorously.
- … which costs time and discipline.
- This works as effective as other new years resolutions like "exercising more" , "eating less" or "file my tax return ASAP"



Prussian Fusiliers Source: WikiMedia Commons

Motivation

Let’s face it

- While programming is frequently experienced as an interesting, creative and self motivating task,
- testing remains an unloved duty
- that tends to be postponed to the end of the development process.

- Moreover there is a natural tendency to chase the bug at locations where it is not hiding.

Benefits

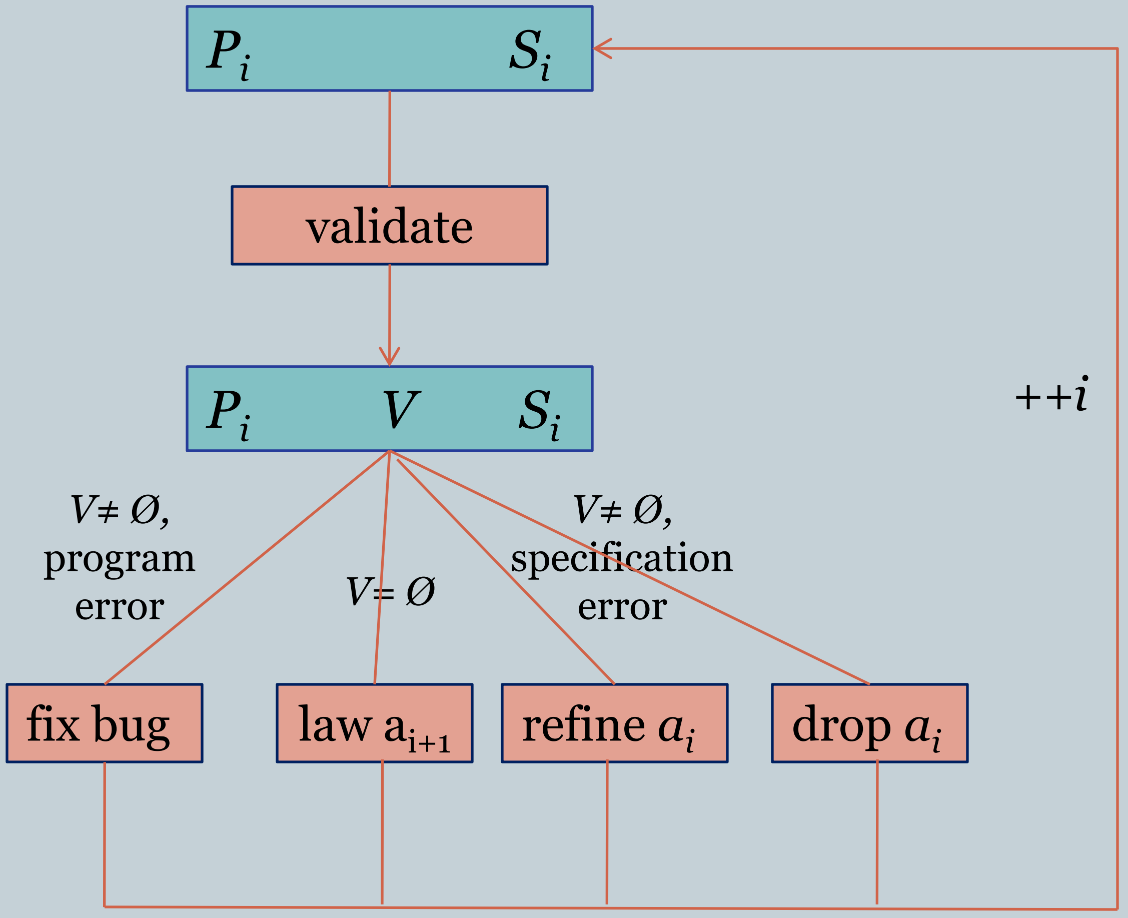
Law based testing gives direct access to the *hidden* realm of errors that we *are not aware of*.

- … an aha experience about the program or it's properties
- … providing a minimal counter example
- … which makes testing extremely efficient
- … and also fun

Benefits

- Each time a law is *falsified* by means of a minimal counter example …
  - we detect a program error
  - or a specification error
- … so we can either
  - fix a bug in the program
  - refine a law
  - abandon a law, because it is not valid

Program Evolution



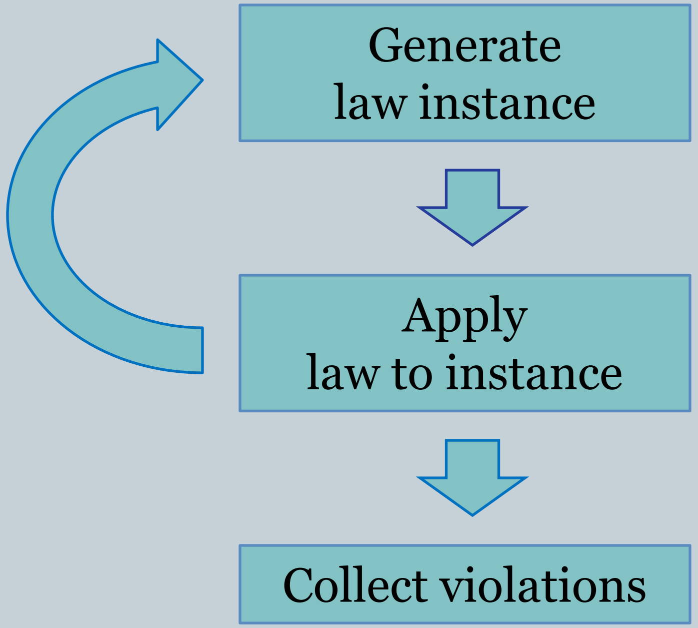
i	Development cycle
P_i	Program for i
S_i	Specification for i: the set of laws.
V	A set of violations (counter-examples) ordered by simplicity
a_i	A law to be validated

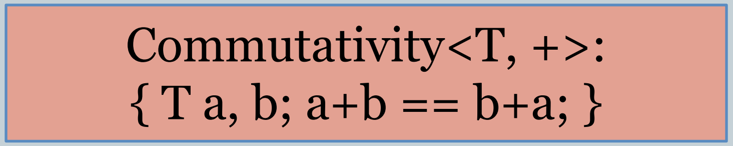
Most important Aspect

Law based testing transforms testing into development.

Design: A Law Based Testing Machine

- A simple testing algorithm.





Design: Law & Law Instance

- A Law in Alabaster is a boolean c++ function: *bool h(T_1, …, T_n);*
- that depends on variables *(x_1, …, x_n)* of types *(T_1, …, T_n)* .
- A law instance is the law's function together with an instance of variables *(h, (x_1, …, x_n))* .

Design: Laws

- A basic `Law` class template

```
template<class SubType,        // For static polymorphism (CRTP)
         class InputTypes>     // Typelist for (T1, ..., Tn)
class Law
{
public:
  typedef typename Loki::tuple<InputTypes>  input_tuple;
  bool holds();                // Function h
private:
  input_tuple  _input_tuple;   // Variables (x1, ..., xn)
};
```

Design: Laws

- Adding features for debugging

```
template<class SubType,     // For static polymorphism (CRTP)
         class InputTypes, // Typelist for $(T_1, ..., T_n)$
         class OutputTypes>// Types for intermediate and
                           // final results $(U_1, ..., U_m)$.
class Law
{
public:
  typedef typename Loki::tuple<InputTypes>  input_tuple;
  typedef typename Loki::tuple<OutputTypes> output_tuple;
  bool holds();          // Function $h$
  bool debug_holds(); // Verbose variant of $h$
private:
  input_tuple  _input_tuple;  // Variables $(x_1, ..., x_n)$
  output_tuple _output_tuple; // Output-Variables $(y_1, ..., y_n)$
};                       // for intermediate and final results.
```

Design: Laws

- Interface

```
template<class SubType, class InputTypes, class OutputTypes>
class Law
{
public:
  typedef typename Loki::tuple<InputTypes>  input_tuple;
  typedef typename Loki::tuple<OutputTypes> output_tuple;
  bool holds();                      // Function h
  bool debug_holds();                // Verbose variant of h
  void setInstance(const input_tuple& inVars);
  void getInstance(input_tuple& inVars,output_tuple& outVars)const;
  size_t size()const;                // Size of the law instance as base
  bool operator < (const Law& rhs)const;//for a simplicity ordering
  std::string name()const;           // Descriptive functions for a
  std::string formula()const;        // readable report on violations
  std::string typeString()const;// or counter-examples.
private: . . .
};
```

Design: Generators

- A generator g
- generates instance variables $(x_1, \ldots, x_n)$
- of types $(T_1, \ldots, T_n)$ for a given Law.

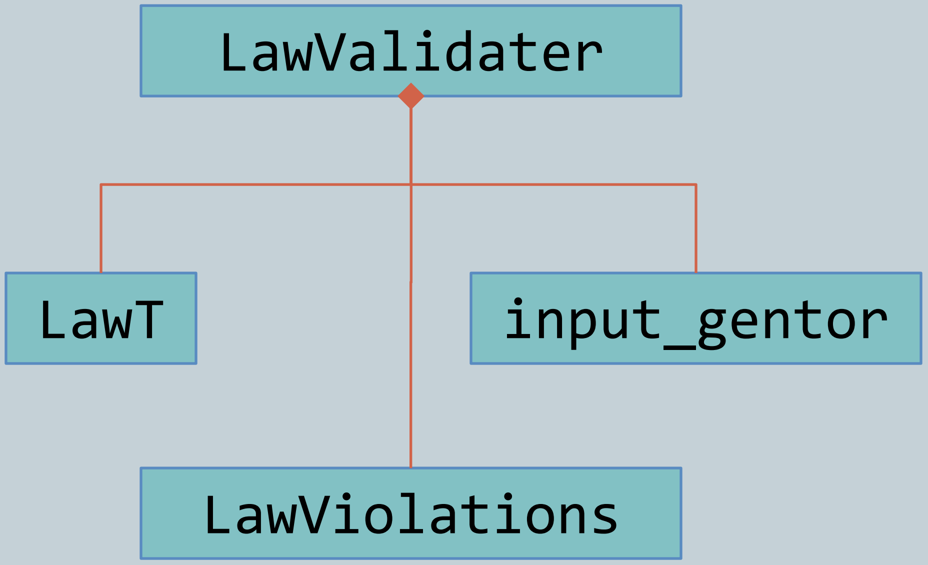
$$g<\mathrm{T}<T_1, \ldots , T_n>> \to \mathrm{T}<g<T_1>, \ldots , g<T_n>>$$

where g is a generator class template,
T denotes a typelist.

```
//Code
typedef typename
  MapUnaryTemplate<GentorT, input_types>::Result
  gentor_types;
```

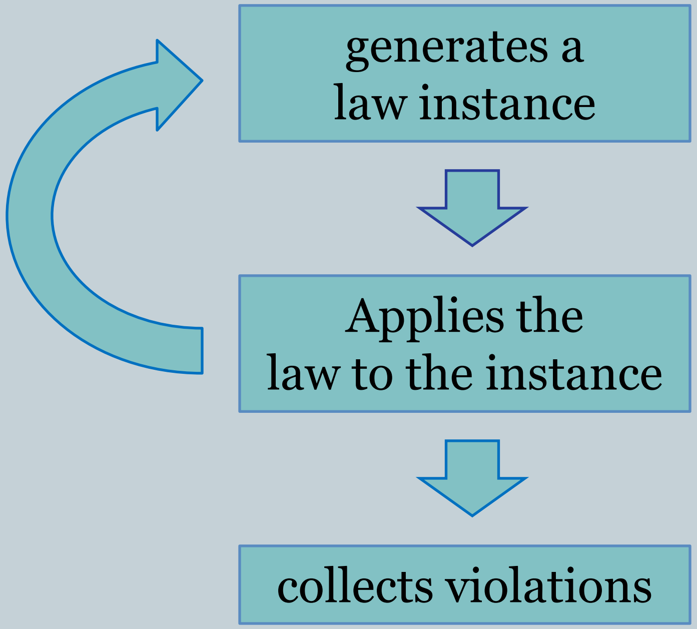
Design: Law Validator

- A law validator holds a law to be validated,
- a generator for the law's instance variables,
- and a container to collect law violations.



Implementation: Law Validator

- A law validator implements a function `run()` that …



Implementation: Law Validator

```
template <class LawT, template<class>class GentorT>
class LawValidater : public LawValidaterI
{                            // ^Abstract base class
public:
  void init();
  void run();                // The test machine
  void addViolations(LawViolations<LawT>& collector);

  typedef typename LawT::input_types input_types;
  typedef typename // This generates the generator type
    MapUnaryTemplate<GentorT, input_types>::Result gentor_types;
  typedef typename Loki::tuple<gentor_types> input_gentor;

private:
  LawT                _law;         // The Law
  input_gentor        _gentor;      // Generator for law instances
  LawViolations<LawT> _lawViolations;// Collector for counter-examples
};                                  // ordered by simplicity
```

Implementation: Validation Loop

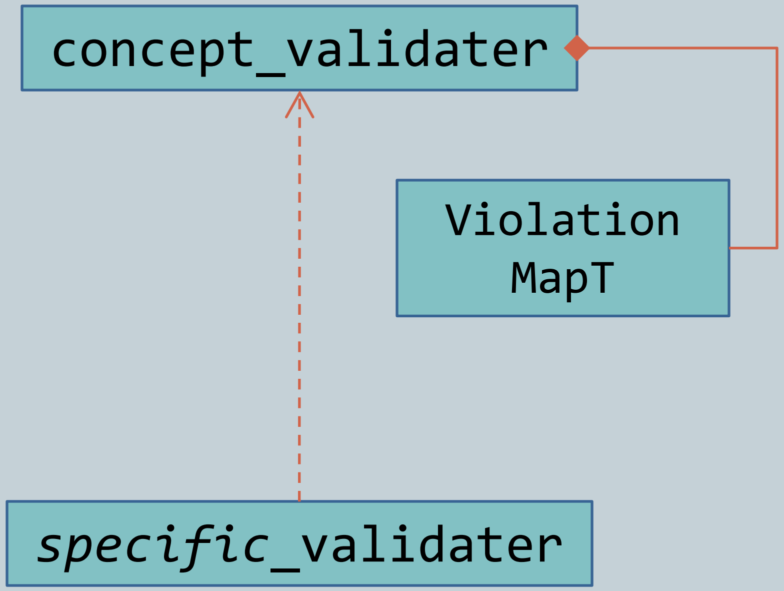
```
template <class LawT, template<class>class GentorT>
void LawValidater<LawT, GentorT>::run()
{
  input_tuple values; // Instance variables (x1, ..., xn)

  for(int idx=0; idx<_trials_count; idx++)
  {                                  // Generate law instance
    _gentor.template map_template<GentorT,SomeValue>(values);
    _law.setInstance(values);

    if(!_law.holds())                // Apply law to instance
      _lawViolations.insert(_law); // Collect violations
  }
}
```

Example: Commutativity

- Example `boostcon_law_validater.cpp` shows how to validate a single law like Commutativity for different types.
- Examples of basic laws for (abelian) monoids can be found in in file `validate\laws\monoid.hpp`.

Design: Concept Validator

- Composing law validators *inevitably* leads to the creation of concept validators.
- A concept validator validates the set of laws that are assumed to be valid for a concept: The semantic constraints of the concept.
- To develop a concept validator means to develop a concept.



Design: Concept Validator

- A concept validator calls law validators or other concept validaters (partial validators) to perform a validation.
- A concept validator defines a probability distribution for it's partial validator to control the relative frequencies of tests.

```
class concept_validater
{
public:
  virtual LawValidaterI* chooseValidater()=0;
  void validate() { _validater = chooseValidater(); ... _validater->run(); ... }
  ...
private:
  LawValidaterI*  _validater;
  ViolationMapT   _violations;
};
```

Example: Monoid Validator

```
template <class Type>
class monoid_validater : public concept_validater
{
public:
  enum Laws {associativity, neutrality, Laws_size};

  void setProfile(){ // Probability distribution for the choice of laws:
    _lawChoice.setSize(Laws_size);
    _lawChoice[associativity] = 50;
    _lawChoice[neutrality]    = 50;
    _lawChoice.init();
  };

  LawValidaterI* chooseValidater(){
    switch(_lawChoice.some()){
    case associativity: return new LawValidater<InplaceAssociativity<Type> >;
    case neutrality:    return new LawValidater<InplaceNeutrality   <Type> >;
    }
  }

private:
  ChoiceT _lawChoice;
};
```

Example: Composing Validaters

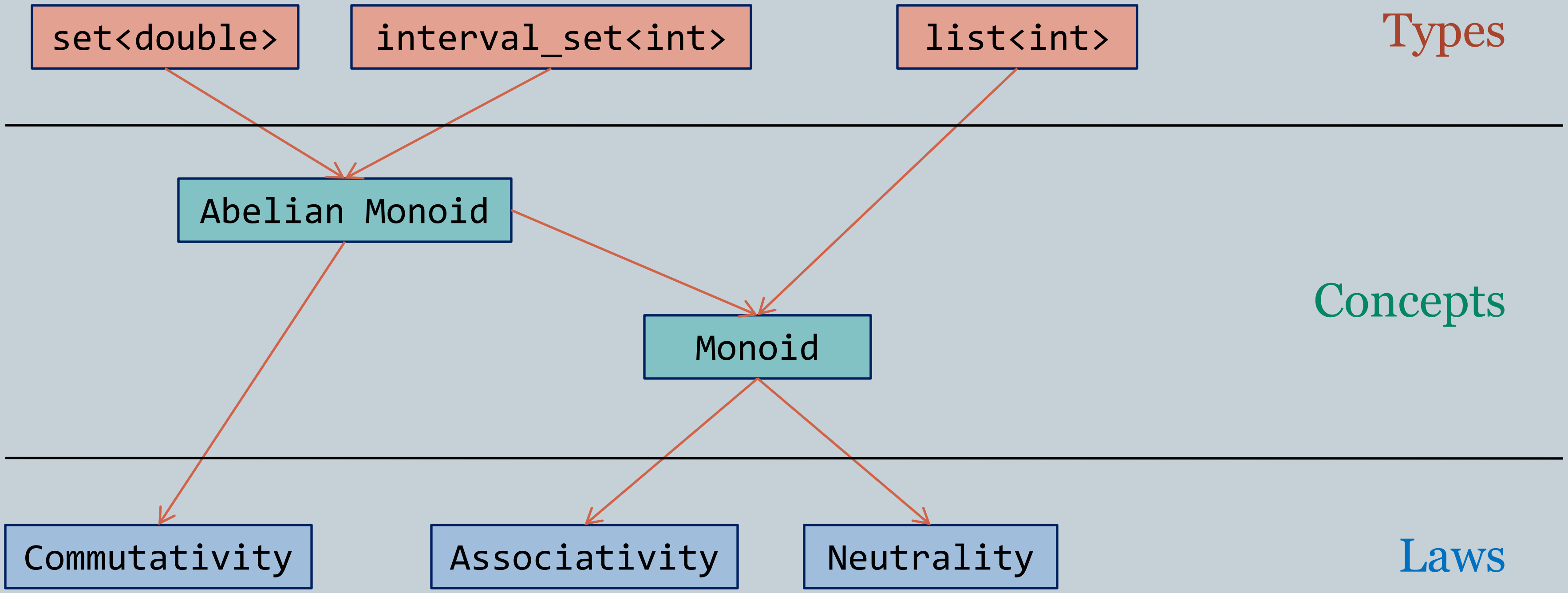
```
template <class Type>
class abelian_monoid_validater : public concept_validater
{
public:
  enum Laws {monoid_laws, commutativity, Laws_size};

  void setProfile(){ // Probability distribution for the choice of laws:
    _lawChoice.setSize(Laws_size);
    _lawChoice[monoid_laws]   = 66;
    _lawChoice[commutativity] = 34;
    _lawChoice.init();
  };

  LawValidaterI* chooseValidater(){
    switch(_lawChoice.some()){
    case monoid_laws:   return _monoid_validater.chooseValidater();
    case commutativity: return new LawValidater<InplaceCommutativity<Type> >;
    }
  }

private:
  ChoiceT                 _lawChoice;
  monoid_validater<Type> _monoid_validater;
};
```

Generating Type Tests

- Using concept validators we can write top level test drivers,
- that call concept validators for various instantiations of a generic class template to be tested.
- Random choices are used again, to grant that all combinations occur with a certain likelihood.

Example: abelian_monoids

- boostcon_abelian_monoids.cpp tests if a type is model of the concept AbelianMonoid.



Developing Laws

- Laws are abstractions that can be used across concepts.
- Laws developed for one library could be used to validate others, they are reusable.
- Abstraction on laws may lead to more general laws.

Some General Laws

- Comparing two implementations of a function
  $\forall\, S\ x;\ f(x) = g(x)$

$$\begin{tikzcd} S \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] \arrow[r, phantom, "="] & T \end{tikzcd}$$

```
template <class S, class T,
          template<class,class>class Function_f,
          template<class,class>class Function_g,
          template<class>class Equality = itl::std_equal>
class FunctionEquality : public Law<. . .>
```

Some General Laws

- Comparing binary operations $.\circ. : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ for two different implementations S and E of a concept $\mathcal{C}$,
- where $f : S \to E$ is a function that transforms S into E
- $\forall\, S\ x,y;\ f(x \circ y) = f(x) \circ f(y)$

$$\begin{tikzcd} \mathcal{C} \arrow[r] & \mathit{Set} \end{tikzcd}$$

$$\begin{tikzcd} S \times S \arrow[r, "\circ"] \arrow[d, "f"'] \arrow[dr, phantom, "="] & S \arrow[d, "f"] \\ E \times E \arrow[r, "\circ"'] & E \end{tikzcd}$$

$$\begin{tikzcd} \texttt{interval set}^2 \arrow[r, "\circ"] \arrow[d, "\texttt{atomize}"'] \arrow[dr, phantom, "="] & \texttt{interval set} \arrow[d, "\texttt{atomize}"] \\ \texttt{itl::set}^2 \arrow[r, "\circ"'] & \texttt{itl::set} \end{tikzcd}$$

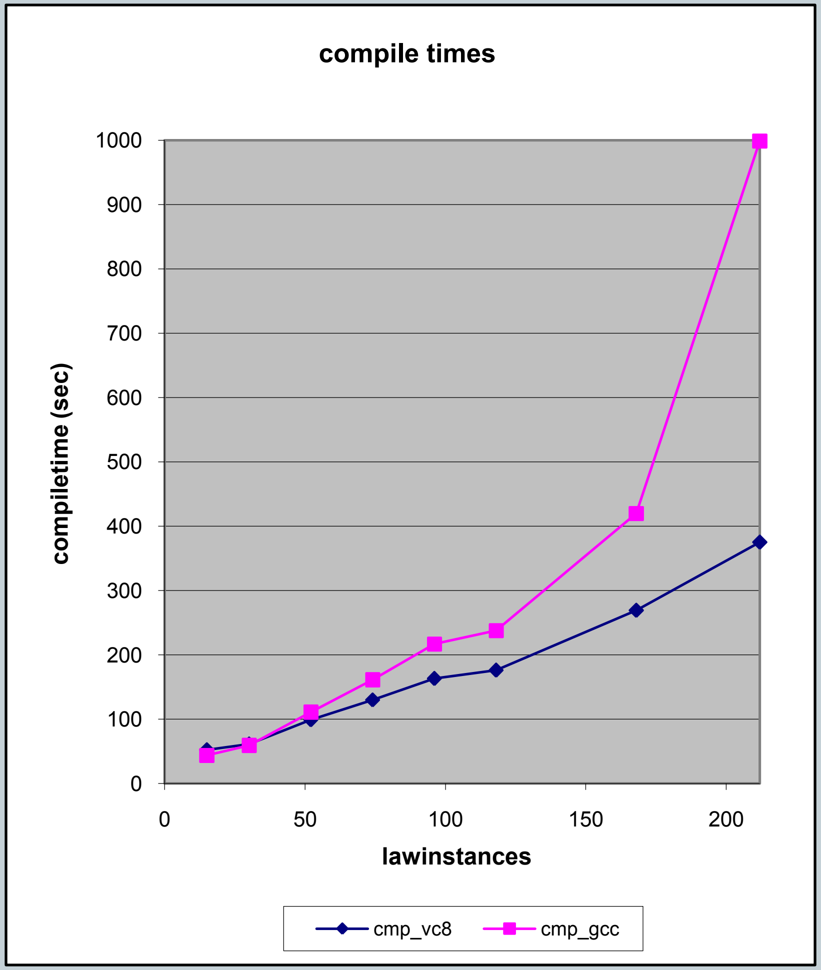
$$\begin{tikzcd} \texttt{interval impl} \arrow[d, "f\texttt{=atomize}"] \\ \texttt{element impl} \end{tikzcd}$$

Practical Aspects

- Law based testing can provide very good code coverage, *depending* on the specified laws.
- Undefined behavior will be detected quickly, but without counter-example service.
- Resource leaks, that might be overlooked otherwise, will be detected.
- Stress tests can easily be created using the Calibrator.
- Law based testing can be combined with other unit testing methods e. g. Boost.Test.

Problems and Limitations

- Compile time performance.
- Template induced code bloat.
- Some compile time measures from December 2008:
- Compilers have improved by now but compile time is still an obstacle.
- Law based testing is no verification! Only advanced testing.



Challenges for Alabaster

- A generator library.
  - Could be of value for all kinds of mocking tools and Monte Carlo studies.
  - Generating specific subtypes.
- A law or specification library.
  - Structuring and developing the field of computable laws.
- Profiling.
- Theorem proving.

Summary

Law based testing …

- transforms testing into development.
- is inherently motivating and can be fun.
- results in extremely solid test suites.
- always produces some abstract insights about a program
- … allowing for more durable design decisions.
- allows to check, if an implementation of a type is model of a concept on the semantic level.
- is probably most adequate to generic library development.

Thanks to

THE BOOSTCON 2010 ORGANIZERS
THE BOOST COMMUNITY

Version 1.0.0
2010-04-30