A generic datatype traversal library for C++

Dan Marsden

May 8, 2008

Outline

Outline

Scrap your boilerplate

Context

- Manipulating complex data structures in many languages is dull repetitive work

Scrap your boilerplate

Context

- Manipulating complex data structures in many languages is dull repetitive work
- There is a lot of literature in the Haskell community concerned with simplifying work with complex data structures

Scrap your boilerplate

Context

- Manipulating complex data structures in many languages is dull repetitive work
- There is a lot of literature in the Haskell community concerned with simplifying work with complex data structures
- The current state of the art is C++ is is masses of deeply nested iteration - either loops or algorithm calls

Motivation

```
for(int_vvv::iterator it=v.begin(), vvv_end=v.end();
  it != vvv_end; ++it)
  for(int_vv::iterator jt=it->begin(), vv_end=it->end();
    jt != vv_end; ++jt)
    for(int_v::iterator kt=jt->begin(), v_end=jt->end();
      kt != v_end; ++kt)
        *kt += increase;
```

Motivation

```
for(int_vvv::iterator it=v.begin(), vvv_end=v.end();
  it != vvv_end; ++it)
  for(int_vv::iterator jt=it->begin(), vv_end=it->end();
    jt != vv_end; ++jt)
    for(int_v::iterator kt=jt->begin(), v_end=jt->end();
      kt != v_end; ++kt)
        *kt += increase;

  traversal::full(
    v,
    traversal::restrict<void(int&)>(_1 += increase));
```

Why is this relevant to Boostcon?

- Experience translating theory from another language
- The library touches a lot of existing parts of Boost
- Hopefully the library is interesting in its own right
- I'm always after feedback!

Outline

Structure like types

```
data Struct = St Int String Float
```

Details

- Read this Int *and* String *and* Float
- Other more complex data types are built similarly

Enumerations

```haskell
data Colors = Red | Blue | Yellow
```

Details

- Read this as Red *or* Blue *or* Yellow
- Another fundamental way of building up types

Natural numbers

data Nat = Zero | Succ Nat

Natural numbers

data Nat = Zero | Succ Nat

Zero

Natural numbers

data Nat = Zero | Succ Nat

Zero

Succ Zero

Natural numbers

data Nat = Zero | Succ Nat

Zero

Succ Zero

Succ (Succ Zero)

Natural numbers

data Nat = Zero | Succ Nat

Zero

Succ Zero

Succ (Succ Zero)

Details

- Recursive type definition
- Yet another fundamental building block

Cons lists

data **List** a = Nil | Cons a **List**

Cons lists

data List a = Nil | Cons a **List**

Nil

Cons lists

```
data List a = Nil | Cons a List
Nil
Cons 101 Nil
```

Cons lists

data List a = Nil | Cons a **List**

Nil

Cons 101 Nil

Cons 101 (Cons 202 (Cons 303 Nil))

Cons lists

data List a = Nil | Cons a **List**

Nil

Cons 101 Nil

Cons 101 (Cons 202 (Cons 303 Nil))

Details

- Polymorphic type
- Our final variation of interest

List syntactic sugar

```haskell
-- Syntactic sugar for cons
x:xs
-- Syntactic sugar for the empty list
[]
-- Syntactic sugar for a list
[1,2,3,4,5]
-- Syntactic sugar for lists with elements of type a
[a]
```

Functions in Haskell

```
length :: [a] -> Int
length []     = 0
length (x:xs) = 1 + length xs
```

Details

- Functions defined on patterns
- Multiple definitions can be provided for different patterns
- Recursion is commonplace

Pointless definitions

```
foldr f e [] = e
foldr f e (x:xs) = f x (foldr f e xs)
```

Pointless definitions

```
foldr f e [] = e
foldr f e (x:xs) = f x (foldr f e xs)

foldr f e [1,2,3,4]
f 1 (f 2 (f 3 (f 4 e)))
```

Pointless definitions

```
foldr f e [] = e
foldr f e (x:xs) = f x (foldr f e xs)

foldr f e [1,2,3,4]
f 1 (f 2 (f 3 (f 4 e)))

sum = foldr (+) 0
```

Some common functions

```
foldl f e [] = e
foldl f e (x:xs) = foldl f (f e x) xs
```

Some common functions

```haskell
foldl f e [] = e
foldl f e (x:xs) = foldl f (f e x) xs

foldl f e [1,2,3,4]
(f (f (f (f e 1) 2) 3) 4)
```

Some common functions

```
foldl f e [] = e
foldl f e (x:xs) = foldl f (f e x) xs

foldl f e [1,2,3,4]
(f (f (f (f e 1) 2) 3) 4)

map f = foldr (\x xs -> f x : xs) []
map f [1,2,3,4]
[f 1, f 2, f 3, f 4]
```

Outline

Boost Fusion library

C++

```
fusion::vector<int, float, std::string> v(
  1, 1.1, hello);

std::pair<int, float> p(1, 1.1);

struct example {
  std::string name;
  int number;
};
```

Boost Fusion library

C++

```
fusion::vector<int, float, std::string> v(
  1, 1.1, hello);

std::pair<int, float> p(1, 1.1);

struct example {
  std::string name;
  int number;
};
```

Haskell

```
data Struct = St Int String Float
```

Boost Variant library

C++

```
boost::variant<int, std::string> = 101;
boost::variant<int, std::string> = "hello"
```

- Stores 1 and only 1 of a fixed list of types
- Cannot be empty

Boost Variant library

C++

```
boost::variant<int, std::string> = 101;
boost::variant<int, std::string> = "hello"
```

- Stores 1 and only 1 of a fixed list of types
- Cannot be empty

Haskell

```
data Either a b = Left a | Right b

Left 101
Right "hello"
```

Boost Optional library

C++

```
boost::optional<int> full = 101;
boost::optional<int> empty;
```

- Optionally contains a value of a specified type
- Otherwise its empty!
- Like boost::variant<int,nothing_type>

Boost Optional library

C++

```
boost::optional<int> full = 101;
boost::optional<int> empty;
```

- Optionally contains a value of a specified type
- Otherwise its empty!
- Like boost :: variant <int, nothing_type>

Haskell

```
data Maybe a = Just a | Nothing

Just 101
Nothing
```

Boost Smart Pointer library

C++

```
boost::shared_ptr<int> full = new int(101);
boost::shared_ptr<int> empty;
```

- Optionally holds a value
- Can be empty
- Resembles Boost.Optional, and therefore Boost.Variant
- Pointers and other smart pointers are similar

Brief aside

Observations

- Boost has a library unifying all tuple types
- Boost has (at least) 3 libraries covering variant types
- Built in pointers and `auto_ptr` provide more variants
- Unifying these variants under 1 framework seems natural

Standard containers

Details

- Obvious building blocks in C++
- What do we do with associative containers?
- They're not built up recursively like their Haskell equivalents

Function object related libraries

Function objects in Boost

- Boost.Bind
- Boost.MemFn
- Boost.Function
- Boost.Lambda + Spirit.Phoenix
- Boost.ResultOf

Outline

The original inspiration

Scrap your boilerplate

Scrap Your Boilerplate: A Practical Design for Generic Programming - Lämmel and Peyton Jones

- Automates the boilerplate code required to traverse complex data structures
- Allows users to concentrate on the interesting functionality in their application

Our example data structure

```
data Company = C [Dept]
data Dept = D Name Manager [SubUnit]
data SubUnit = PU Employee | DU Dept
data Employee = E Person Salary
data Person = P Name Address
data Salary = S Float
type Manager = Employee
type Name = String
type Address = String
```

Pay rise time

```
increase k (C ds) = C (map (incD k) ds)
```

Pay rise time

```
increase k (C ds) = C (map (incD k) ds)

incD k (D nm mgr us) =
  D nm (incE k mgr) (map (incU k) us)
```

Pay rise time

```haskell
increase k (C ds) = C (map (incD k) ds)

incD k (D nm mgr us) =
  D nm (incE k mgr) (map (incU k) us)

incU (PU e) = PU (incE k e)
incU (DU d) = DU (incD k d)
```

Pay rise time

```
increase k (C ds) = C (map (incD k) ds)

incD k (D nm mgr us) =
  D nm (incE k mgr) (map (incU k) us)

incU (PU e) = PU (incE k e)
incU (DU d) = DU (incD k d)

incE k (E p s) = E p (incS k s)
incS k (S s) = S (s * (k + 1))
```

Pay rise time

```
1 increase k (C ds) = C (map (incD k) ds)

2 incD k (D nm mgr us) =
3   D nm (incE k mgr) (map (incU k) us)

4 incU (PU e) = PU (incE k e)
5 incU (DU d) = DU (incD k d)

6 incE k (E p s) = E p (incS k s)
7 incS k (S s) = S (s * (k + 1))
```

Boilerplate

- Repetitive boilerplate is everywhere
- The only interesting code is on line 7

Making it easier to give pay rises

```
increase k = everywhere (mkT (incS k))
```

Details

- It's certainly shorter
- The style is more declarative
- How do we build something like the everywhere function?
- What does mkT do?

The one level map trick

Outline

Apply a function to each *immediate* sub element of a given type

The one level map trick

Outline

Apply a function to each *immediate* sub element of a given type

```
gmapT f (E per sal) = E (f per) (f sal)
```

The one level map trick

Outline

Apply a function to each *immediate* sub element of a given type

```
gmapT f (E per sal) = E (f per) (f sal)

gmapT f Nil = Nil
gmapT f (Cons x xs) = Cons (f x) (f xs)
```

Applying the 1 level map trick

Outline

Use the one level map to build a function that applies *itself* as the mapping function

Applying the 1 level map trick

Outline

Use the one level map to build a function that applies *itself* as the mapping function

```
everywhere f x = f (gmapT (everywhere f) x)
```

Applying the 1 level map trick

Outline

Use the one level map to build a function that applies *itself* as the mapping function

```
everywhere f x = f (gmapT (everywhere f) x)

everywhere' f x = gmapT (everywhere' f) (f x)
```

Another horizontal mapping

Outline

A mapping that applies a function to each *immediate* sub element, and returns a list of results of a *fixed* type

Another horizontal mapping

Outline

A mapping that applies a function to each *immediate* sub element, and returns a list of results of a *fixed* type

```
gmapQ f (E per sal) = [f per, f sal]
```

Another horizontal mapping

Outline

A mapping that applies a function to each *immediate* sub element, and returns a list of results of a *fixed* type

```
gmapQ f (E per sal) = [f per, f sal]

gmapQ f Nil = []
gmapQ f (Cons x xs) = [f x, f xs]
```

Building type unifying recursive traversals

Outline

Combine our new traversal that converts to a *fixed* type with a function to reduce the lists to a single value

Building type unifying recursive traversals

Outline

Combine our new traversal that converts to a *fixed* type with a function to reduce the lists to a single value

```
everything k f x =
  foldl k (f x) (gmapQ (everything k f) x)
```

Generalizing the horizontal traversals

Generalization

- gmapT - Transformations
- gmapQ - Queries
- gmapM - Monadic traversals

```
gfoldl k z (E p s) = (z E `k` p) `k` s
```

Generalizing the horizontal traversals

Generalization

- gmapT - Transformations
- gmapQ - Queries
- gmapM - Monadic traversals

```
gfoldl k z (E p s) = (z E `k` p) `k` s

gfoldl k z Nil = z Nil
gfoldl k z (Cons x xs) = (z Cons `k` x) `k` xs
```

More ideas for the Haskell community

Scrap your boilerplate systematically

A Generic Recursion Toolbox for Haskell Or: Scrap Your Boilerplate Systematically - Ren and Erwig

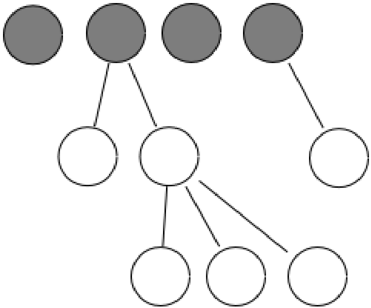
- An extension of the original Scrap Your Boilerplate work
- Provides a detailed analysis of the domain of recursive traversal functions
- A richer collection of high level building blocks to simplify use
- Still extensible for complex schemes, but aims to provide more functionality out of the box

The different aspects of recursive traversal

Description

- Transformers and accumulators - Type preservation versus unification
- Horizontal traversal direction
- Vertical traversal direction
- Early horizontal traversal termination
- Early vertical traversal termination
- Fixed point traversals - For later versions of our library

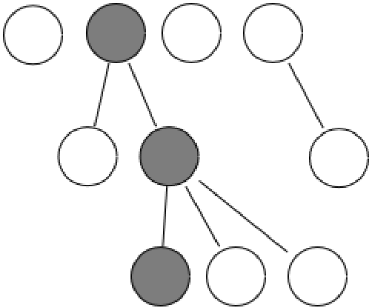
Horizontal Traversal



Details

- Key building block of the library
- Do not recurse into substructure directly
- 2 major variants one and all

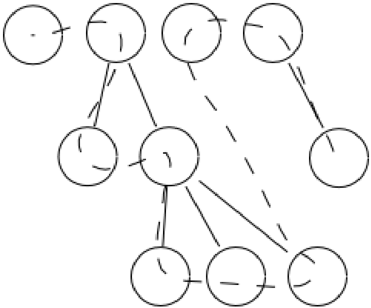
Vertical Traversal



Details

- Concept for understanding traversals
- No concrete representation in the library

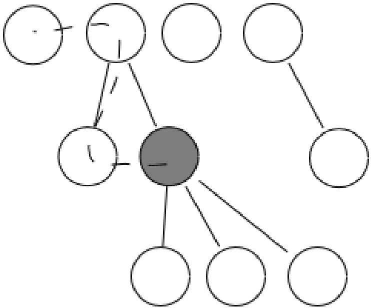
Full Traversal



Details

- Horizontal traversals cannot terminate early
- Vertical traversals cannot terminate early

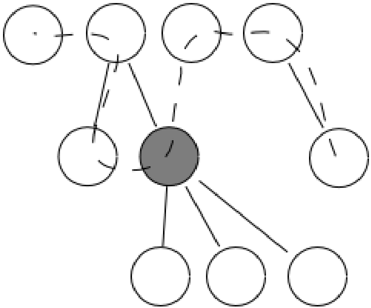
Once Traversal



Details

- Horizontal traversals terminate early
- Vertical traversals terminate early

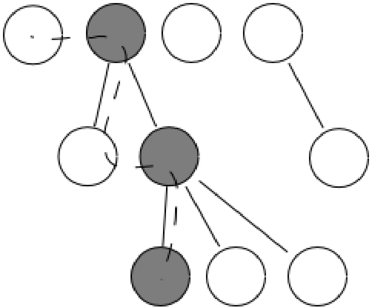
Stop Traversal



Details

- Horizontal traversals cannot terminate early
- Vertical traversals terminate early

Spine Traversal



Details

- Horizontal traversals terminate early
- Vertical traversals cannot terminate early

Outline

Haskell versus C++

Haskell

- Type inference
- Purity - No side effects
- Lazy evaluation

C++

- Explicit typing
- Side effects!
- Eager evaluation

Example - Old school C++ programmer

```
1 for(int_vvv::iterator it=v.begin(), vvv_end=v.end();
2   it != vvv_end; ++it)
3   for(int_vv::iterator jt=it->begin(), vv_end=it->end();
4     jt != vv_end; ++jt)
5     for(int_v::iterator kt=jt->begin(), v_end=jt->end();
6       kt != v_end; ++kt)
7       *kt += increase;
```

Description

- Very repetitive
- The only interesting code is on line 7!

Example - Trendy modern C++ programmer

```
1 std::for_each(v.begin(), v.end(),
2   ll::for_each(bind(call_begin(), _1), bind(call_end(),
3     protect(
4       ll::for_each(
5         bind(call_begin(), _1), bind(call_end(), _1),
6         protect(
7           _1 += increase)))));
```

Description

- Still very repetitive
- Requires knowledge of 'advanced' features of Boost.Lambda
- The only interesting code is still on line 7!

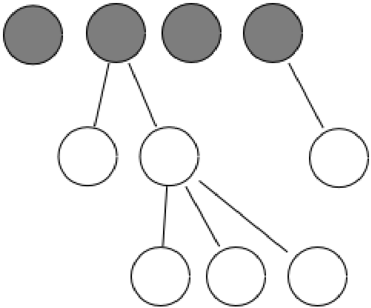
Using the current Traversal library

```
traversal::full(
  v,
  traversal::restrict<void(int&)>(_1 += increase));
```

Description

- Shorter!
- Hopefully clearer with a bit of experience
- Independent of the data structure to traverse

Horizontal Traversal in C++



Details

- Key building block of the C++ library
- Early termination makes things fiddly
- 4 variants in our library one, all, reverse_one, reverse_all

One traversal

```
bool result = one(my_object, func);
```

Detail

- Applies func to every element of my_object
- Stops the traversal if func returns true
- Returns the last result of func

Standard Container Horizontal Traversal

```
template<typename It, typename UnaryFunc>
bool iter_one(It first, It last, UnaryFunc func)
{
    for(;first != last; ++first)
        if(func(*first))
            return true;
    return false;
}
```

Fusion Container Horizontal Traversal

```
template<
  typename T0, typename T1,
  typename UnaryFunc>
bool pair_one(
  std::pair<T0, T1>& pr, UnaryFunc func)
{
  return func(pr.first) || func(pr.second);
}
```

The one level map trick in C++

```
template<typename Func>
struct full_impl {
  explicit full_impl(Func f)
    : f(f) {}

  template<typename T>
  bool operator()(T& t) const {
    f(t);
    all(t, *this);
    return false;
  }

  Func f;
};
```

C++ weakly typed basic building blocks

```
typedef int salary;
typedef int age;
```

C++ strongly typed basic building blocks

```
template<typename Tag>
struct Id {
    std::string id;
};

struct name_tag;
struct address_tag;

typedef Id<name_tag> surname_type;
typedef Id<address_tag> address_type;
```

C++ People are our business

```
struct person {
    surname_type surname;
    address_type address;
    int age;
};

struct employee {
    person detail;
    salary pay;
};
```

C++ Organizational structure

```
struct dept;

typedef boost::variant<dept, employee>
  unit;

struct dept {
    std::string name;
    std::vector<unit> units;
};

typedef std::vector<dept> company;
```

A simple example

```
traversal::full(
  my_company,
  traversal::restrict<void(surname_type&)>(
    std::cout << _1 << std::endl));
```

Full traversal

4 variations, full, bu_full, reverse_full, bu_reverse_full

Pay rise time again!

```
traversal::full(
  my_company,
  traversal::restrict<void(salary&)>(
    _1 += increase));
```

Rises

- We've raised everybodies salary

Pay rise time again!

```
traversal::full(
  my_company,
  traversal::restrict<void(salary&)>(
    _1 += increase));
```

Rises

- We've raised everybodies salary
- We've just raised their ages by the same amount!

A more selective pay rise

```
struct raise_salary {
  explicit raise_salary(int raise)
  : raise_(raise) {}

  int raise_;

  template<typename Parent, typename Child>
  void operator()(Child&, Parent&) const {}

  void operator()(salary& s, employee& e) const {
    s += raise_;
  }
};
```

Applying our more selective pay rise

```
traversal::full_parent(
  my_company,
  raise_salary(101))
```

Parent Traversals

- Our pay rise is applied correctly
- All traversals come with an _parent variant
- No access above the immediate parent yet
- You can get the same effect just by picking something up the hierarchy

How to add up salaries

```
struct sum_salaries {
  template<typename Child, typename Parent>
  int operator()(int state, Child&, Parent&) const {
    return state;
  }

  int operator()(
    int state, salary_type& salary,
    employee& e) const {
    state += salary;
    return state
  }
};
```

How much did that cost us?

```
traversal::tu_full_parent(
  my_company, 0,
  sum_salaries())
```

How much did that cost us?

```
traversal::tu_full_parent(
  my_company, 0,
  sum_salaries())

traversal::tu_full_parent(
  my_company, 0,
  traversal::tu_restrict<
    int(int&, salary_type&, employee&)>(
      _1 + _2))
```

Querying into a data structure

Specification

- Return a pointer to a named department
- Return a null pointer if no such department is found

Querying into a data structure

Specification

- Return a pointer to a named department
- Return a null pointer if no such department is found

Approach

- We cannot use a full traversal, we want to terminate early
- Our function object is going to need to indicate when to stop

Search object: default implementation

```
template<typename T>
std::pair<Dept*, bool>
operator()(
  std::pair<Dept*, bool> const& state,
  T& t) const
{
  return state;
}
```

Search object: interesting implementation

```
std::pair<Dept*, bool>
operator()(
  std::pair<Dept*, bool> const& state,
  Dept& dept) const
{
  if(dept.name == name)
    return std::make_pair(&dept, true);
  else
    return state;
}
```

Performing the search

```
traversal::tu_once(
  my_company,
  static_cast<Dept*>(0),
  find_dept("HR"))
```

Summary

- tu_once terminates early unlike the full traversals
- The static_cast is currently necessary to get the typing right
- The function object knows the right return type, so maybe we can do better

Searching variant number 2

```
std::pair<Dept*, bool>
hr_finder(
  std::pair<Dept*, bool> const& state,
  Dept& dept) const
{
  if(dept.name == "HR")
      return std::make_pair(&dept, true);
  else
      return state;
}

traversal::tu_once(
  my_company,
  static_cast<Dept*>(0),
  traversal::tu_extend(hr_finder))
```

Searching variant number 3

```
struct finder {
  typedef std::pair<Dept*, bool> result_type;
  typedef std::pair<Dept*, bool>& first_argument_type;
  typedef Dept& second_argument_type;

  explicit finder(std::string const& name);

  result_type
  operator()(
    first_argument_type state,
    second_argument_type dept) const;

  std::string name;
};
```

Having our cake and eating it!

```
traversal::tu_once(
  my_company,
  static_cast<Dept*>(0),
  traversal::tu_extend(finder("Finance")))
```

Summary

- We've now separated out the skipping logic
- We don't need to hardwire the search department

Pattern matching

```
traversal::full(
  data,
  traversal::match<
    pattern::vector<pattern::_> >(action))
```

Pattern matching

- More general patterns than just concrete types
- Still fully statically determined
- Inspired by Proto
- Structure shy programming!

Pattern details

```
template<typename T>
struct lit {
    template<typename U>
    struct apply
        : is_same<T, U> {};
};
```

Details

- A pattern is an MPL Metafunction Class
- The pattern builders are templates for building Metafunction Classes
- Interoperation with the MPL is easy
- No runtime pattern matching

The pattern based function object

```
template<typename T>
typename boost::enable_if<
  typename mpl::apply<
    MetaFuncClass, T>::type, bool>::type
operator()(T& t) const
{
  return f_(t);
}
```

Pattern matching

- Matching uses enable_if for overload selection
- Nicely orthogonal to the traversal code

Customizing horizontal traversal

```
traversal::full(
  v, func,
  traversal::unstructured<std::vector<int> >(
    traversal::all));
```

Details

- All traversals take an optional horizontal traversal argument
- The default traversal :: all implementation calls the provided overloads
- Combinators such as traversal :: unstructured can be used to customize behaviour for particular types or families of types

Future work

Stuff to do

- Improved pattern matching capabilities
- Easy implementation of custom traversals
- Broader combinator support for common use cases
- Loads of usability / interoperation enhancements

Future work

Stuff to do

- Improved pattern matching capabilities
- Easy implementation of custom traversals
- Broader combinator support for common use cases
- Loads of usability / interoperation enhancements

Other ideas

- Fission library for variants to match Fusions tuple support
- A container / algorithm library based around its algebraic properties