Catsfoot

0.1

Generated by Doxygen 1.7.3

Tue Oct 18 2011 00:23:33

Contents

Chapter 1

Introduction

Contents

Catsfoot is a C++ library providing:

- concept checking,
- concept-based overloadind,
- and generating tests automatically from concepts.

In short, it is intended to provide testing utilities for C++ template libraries.

Catsfoot is developed `Bergen Language Design Laboratory`.

1.1 Concept checking

That part of the library is inspired by `Boost Concept Check Library (BCCL)` and now rejected `concept extension for C++ standard`.

Unlike BCCL, Catsfoot is capable of not failing when concept requirements are not met, and instead use the concept as a static predicate.

C++2011 already provides several predicates in `<type_traits>` (c.f. section 20.7.2). Some other are provided by the library, mostly inspired by `has_member`, but also using interestring properties of `decltype` from C++2011. With this, we are capable for instance of testing the call-ability of functions or operators.

1.2 Concept-based overloading

Since our concepts are testable like predicates, we can then use similar constructs to Boost's `enable_if` to be able to overload function tempaltes based on both static and dynamic properties of our types.

1.3 Axiom testing

The rejected concept proposal provided syntax for axioms. It showed to be useful for automatic testing. Axioms were similar to functions, so making Catsfoot able to treat functions as axioms made the trick.

As in `QuickCheck`, axioms are functions which parameters represent universally quantified variables. Since C++2011 introduced variadic template parameters, it was possible to write test driver "parsing" those parameters and feed the axiom with data from a generator.

Like QuickCheck, Catsfoot provides generator combinators.

The most interesting example of data generator is based on a list of functions, methods and operators which a random generator will use to generate random instances. If the set of functions covers all methods, constructors and friend functions of a class, in theory, it is possible to cover all possible generate-able instances (if we had infinite time and memory).

1.4 Relations to OOP unit testing

It is possible to combine unit testing with axiom testing. However, you should note that several OOP testing techniques becomes deprecated with concept-based testing.

Fixtures are generally not needed since data set generator will generate them. Probably, it is a bad sign if you have fixtures. Your only fixture should be your data generator.

Mock implementations can be easily made through concept-based programming. Better, you can check the mock implementation itself with the concept it is supposed to implement. Concepts do not require virtual methods nor inheritance. For that case, instead of modifying the original implementation to mock. Which is good, because after all, the implementation you need to mock is probably not your code. You just need first to define its specification as a concept, and then modify your code to accept the implementation to choose as a template parameter.

1.5 Catsfoot can integrate your test suite

Catsfoot is just a library. It leaves you the choice to use any testing framework. Most of the time, you probably want to write small programs making the test. In that case, function `main` will just basically configure the data set generators and test the different

concepts. If the environment of your test suite requires a different approach, it will certainly be possible for you use Catsfoot anyway.

Chapter 2

User manual

Contents

An online version of this manual is available on http://catsfoot.sourceforge.net/manual.html.

2.1 Reference documentation

- Getting started
- Concepts-only tutorial
- Axioms-only tutorial
- Tutorial

2.2 API documentation

2.3 Package documentation

- README
- INSTALL
- NEWS
- COPYING
- AUTHORS

2.4 Getting started

2.4.1 Requirements

- A compiler supporting a majority of C++2011.
  - GCC >= 4.5.1
  - Known **not** to work (yet):
    - Visual Studio 2010 and before (e.g. no support of variadic template parameters)
    - Xcode 4 and before (old version of GCC)
    - Clang (has probably enough features, but does not support any C++2011 standard library)
    - ICC 11.1.072 and before
    - GCC 4.4.* and before (e.g. std::declval is missing from the library)
- A standard C++ library supporting a majority of C++2011.
  - GCC's libstdc++
  - Known **not** to work (yet):
    - Clang's libc++
  - Untested:
    - STLport

2.4.2 Installation

This describes the installation for development versions of Catsfoot. The run-time shared library contains only testing tools and does not need to be shipped with the software.

Nightly build versions are available from page Download.

2.4.2.1 From package

There are nightly build packages for

- Fedora 14 x86 and x86_64
- Fedora rawhide x86 and x86_64
- Ubuntu 10.04 x86 and x86_64
- Ubuntu 10.10 x86 and x86_64
- Debian sid x86 and x86_64

Those can be installed with `rpm` or `dpkg` depending on the distribution.

Gentoo users can install it with:

```
# emerge -uv layman
# sed -i "/^overlays *:/a http://vfrd-overlay.googlecode.com/svn/trunk/Documentation/vfrd-overlay.xml" /
# layman -a vfrd
# echo "dev-libs/catsfoot ~$(portageq envvar ARCH)" >>"${EPREFIX}/etc/portage/package.keywords"
# emerge -u dev-libs/catsfoot
```

2.4.2.2 From source

Catsfoot uses Autoconf and Automake, so the installation is pretty standard.

```
1 $ xz -dc /path/to/catsfoot-0.1.tar.gz | tar xf -
2 $ ./catsfoot-0.1/configure --prefix="/path/where/to/install"
3 $ make install
```

2.4.2.3 From the repository

We discourage using the version of the repository as some versions might not have passed the test suite. Use a nightly build source package instead.

2.4.3 Using Catsfoot in your build system

Catsfoot installs two `pkg-config` package descriptions:

- `catsfoot` is used only for supporting concepts. It is used for shipped binaries of your project. In this way it does not depend on any run-time library, but still provides static concept checking.
- `catsfoot-rt` is used for your test suite. It links to Catsfoot's runtime library.

2.4.3.1 Automake & Autoconf

In your configure.ac file, you should have a line such as:

```
PKG_CHECK_MODULES([CF_TESTING], [catsfoot-rt])
PKG_CHECK_MODULES([CF_RELEASED], [catsfoot])
```

Then in your Makefile.am file, should have for example (for "check" program "mytest"):

```
mytest_CXXFLAGS=$(CF_TESTING_CFLAGS)
mytest_LDFLAGS=$(CF_TESTING_LIBS)
```

The manual page of `pkg-config` contains more information.

Do not forget to call "configure" with "CXX='g++ -std=c++0x'", or make your configure script to add "-std=c++0x" automatically.

2.4.3.2 Scons

Scons wiki page `PkgConfig` explains how to use pkg-config in your project.

2.4.3.3 By hand

For example:

```
g++-4.6.0 -std=c++0x `pkg-config --cflags --libs catsfoot-rt` yourtest.cc
```

2.4.4 My first test program

Lets write a concept for a set. We can insert an element inside the set, and we can test whether an element has been inserted.

```
1  # include <catsfoot.hh>
2
3  // We define some tools we will need in our concept.
4  // - has_member_element_type<T> checks for member type
       "T::element_type"
5  DEF_TYPE_MEMBER_PREDICATE(element_type);
6  // - member_has(cv-qualifier T&, Args...) is an alias to
7  //    "T::has(Args...) cv-qualifier"
8  DEF_MEMBER_WRAPPER(has);
9  // - member_insert(cv-qualifier T&, Args...) is an alias to
10 //    "T::insert(Args...) cv-qualifier"
11 DEF_MEMBER_WRAPPER(insert);
```

```
12
13 namespace cf = catsfoot;
14
15 // We define a concept "set" for a type we call "Set"
16 template <typename Set>
17 struct set: public cf::concept {
18   // We define "element_type" to be an alias to Set::element_type if
19   // it ever exists. Using has_member_element_type makes sure that
20   // we will not get any strange error message in case
       Set::element_type
21   // was not a type, or was not declared.
22   // This alias makes the concept more concise, but is not necessary.
23   typedef typename has_member_element_type<Set>::member_type
       element_type;
24
25   // is_callable<C> is a predicate. We declare some aliases to the
26   // predicate calling members Set::has(element_type) const, and
27   // Set::insert(element_type). Since the predicates will be
28   // reused several times, it helps us to make the concept definition
29   // more concise. But it is not necessary.
30   typedef cf::is_callable<member_has(const Set&, element_type)> has;
31   typedef cf::is_callable<member_insert(Set&, element_type)> insert;
32
33   // We declare our static requirements
34   typedef cf::concept_list<
35     // Set::element_type must exist and define a type.
36     has_member_element_type<Set>,
37     // Set::has(element_type) const must be callable
38     has,
39     // Its result should be implicitly convertible to bool.
40     std::is_convertible<typename has::result_type, bool>,
41     // Set::insert(element_type) must be callable
42     insert,
43     // We require "operator==(element_type, element_type)" to be
         defined
44     cf::equality<element_type>
45     > requirements;
46
47   // For all element e and f, with a reference to set s, this axiom
       should
48   // hold.
49   static void insertion(const element_type& e, const element_type& f,
       Set& s) {
50     // Was f already inside?
51     bool res = s.has(f);
52     s.insert(e);
53     if (e != f)
54       // if e is not equal to f, then insertion of e should insert nor
55       // remove f.
56       axiom_assert(res == s.has(f));
57
58     if (e == f)
59       // if e and f are equal, then f should be in the set.
60       axiom_assert(true == s.has(f));
61   }
62
63   // Axioms are usually just static members. We need to declare them as
64   // axioms for the test driver to use them.
65   AXIOMS(insertion);
66 };
```

Note that this concept is does not specify everything from the commonly used set implementations. For example, according to the axioms, a type where `has` is always true for any element and were `insert` does not perform any operation, will model concept `set`.

Let's now define a very simple implementation of set. You can note that we can reuse any set type as long as it implements at least concept `set`. The type will actually implement more specific algebra.

```
1  template <typename T>
2  struct slow_set {
3  public:
4    // Required by the concept.
5    typedef T element_type;
6
7    // Although this is not required by the concept, we provide
8    // some constructors.
9    slow_set() = default;
10   slow_set(const slow_set&) = default;
11   slow_set(std::initializer_list<T>);
12
13   // Required by the concept
14   void insert(const T&);
15   bool has(const T&) const;
16
17 private:
18   std::vector<T> values;
19
20   // Because:
21   // - we do not resize the vector, we do not need default
       constructible
22   // - we do not use move semantic, we can require copy
23   // - we do not replace, we do not need assignment
24   typedef
25   cf::class_assert_concept<
26     cf::concept_list<cf::is_constructible<T(const T&)>,
27                      cf::equality<T> >
28     >
29   requirements;
30 };
31
32 template <typename T>
33 void slow_set<T>::insert(const T& t) {
34   for (auto i : values) {
35     if (i == t)
36       return;
37   }
38   values.push_back(t);
39 }
40
41 template <typename T>
42 bool slow_set<T>::has(const T& t) const {
43   for (auto i : values) {
44     std::cerr << t << " == " << i << std::endl;
45     if (i == t)
46       return true;
47   }
48   return false;
49 }
50
51 template <typename T>
52 slow_set<T>::slow_set(std::initializer_list<T> l) {
```

```
53    for (auto i : l) {
54      this->insert(i);
55    }
56  }
```

We now want to claim for any T, slow_set<T> is a set. We can claim this because we made sure that slow_set<T> was not instantiated with a incompatible type through instantiation of slow_set<T>::requirements. The claim is done through type traits.

```
1   namespace catsfoot {
2     template <typename T>
3     struct verified<set<slow_set<T> > >
4       : public std::true_type {
5     };
6   }
```

To test this claim we now write a test program. We just make a small program that defines the data set we want to use and call the test driver for the concepts we want to check. We can define very complex random term generators, but for now, we will just give a list of sample data.

```
1   #include <limits>
2
3   int main() {
4     using cf::list_data_generator;
5     using cf::choose;
6
7     // To get better error messages, we should first statically check
        that
8     // we are testing a valid claim.
9     cf::assert_concept(set<slow_set<int> >{});
10
11    auto generator =
12      choose
13      // The generator can generate either:
14      // - sets of type slow_set<int>
15      (list_data_generator<slow_set<int> >(
16        {slow_set<int>{}, slow_set<int>({0, 1}),
17         slow_set<int>({0, 0}), slow_set<int>({1, 0})},
18      // - or integers
19       list_data_generator<int>({0, 1, 2, 3, -1, -2,
20                   std::numeric_limits<int>::min(),
21                   std::numeric_limits<int>::max()}));
22
23    // We now test claim "set<slow_set<int> >" against generator.
24    bool ret = cf::test_all<set<slow_set<int> > >(generator);
25
26    // In our case, the implementations of the set was complete, so
27    // we want to verify that we did cover all conditions in the axioms.
28    ret = ret && cf::check_unverified();
29
30    return ret?0:1;
31  }
```

This is the simplified picture of what testing looks like. The tutorials will cover each points in details.

2.5 Concepts-only tutorial

In this tutorial, we introduce the concept based checking only for people not interested in testing utilities from Catsfoot. We expect that you are familiar with C++ templates.

Concepts in this case are useful for two things:

- giving better error messages to the user of a template library.
- selecting optimized version of overload functions based run-time behavior.

2.5.1 Concept requirements

Concepts describe a set of requirements, usually representing the existense of members (types, methods, or static methods), the existence of compatible overloaded functions, and the relations between types (convertible, castable, inheriting, same...). For instance it is possible to check that a type `T` has a non-`const` method `get` that return a type convertible to an instance of type `U`.

These requirements can be represented by those two predicates:

```
1 is_callable<member_get(T&)>
```

And:

```
1 is_convertible<typename is_callable<member_get(T&)>::result_type,
2                U>
```

`member_get` is a functor produced with macro DEF_MEMBER_WRAPPER(get). But it is possible to write it by hand.

For instance:

```
1 struct member_get {
2   template <typename T, typename... U,
3             typename Ret =
4             decltype(std::declval<T>()
5                      .get(std::declval<U>()...))>
6   Ret operator()(T&& t, U&&... u...) const {
7     return std::forward<T>(t).get(std::forward<U>(u)...);
8   }
9 };
```

It is not important to make the functor that generic. However, it is very important that the return type of the operator parenthesis is dependant on the existence of the method using `decltype`. The `decltype` expression needs then to be dependent to a template parameter from the operator parenthesis (and not the functor).

If you fail to provide such a functor. Errors will not be properly detected.

2.5.2 Specifying a concept

Now we can write our first concept. If axioms do not interest you, you will probably most of the time want to use automatic concepts. However introducing overloading based on concepts, we will use then non-automatic concept.

Requirements are represented throught member type `requirements` of the concept. For instance.

```
1  template <typename T, typename U>
2  struct has_get: public auto_concept {
3  private:
4    //easier with an alias to the predicate
5    typedef is_callable<member_get(T&)> call;
6
7  public:
8    typedef concept_list<
9      call,
10     is_convertible<typename call::result_type, U>
11   > requirements;
12 };
```

If the concept has several requirements, like here, we just use class template `concept_list`.

When asserting a concept, the compiler will report only the missing predicates, not the missing concepts. Predicates should be seen as the lowest level of requirements.

If we have a function requiring concept `has_get`, we can easily check against the concept.

```
1  template <typename SimpleStream>
2  int something_useless(SimpleStream& s) {
3    assert_concept(has_get<SimpleStream, int>{});
4    int res = 0;
5    for (unsigned i = 0; i < 8; ++i) {
6      res ^= int(s.get());
7    }
8    return res;
9  }
```

If for example, the result of `get` was not convertible to `int`, then we will get kind of error message:

```
/usr/include/catsfoot/concept/concept_tools.hh: At global scope:
/usr/include/catsfoot/concept/concept_tools.hh: In instantiation of
  'catsfoot::details::class_assert_concept<std::
  is_convertible<return_type, int>, false, false, false>':
[...]
/usr/include/catsfoot/concept/concept_tools.hh:125:7: error: static
  assertion failed: "Missing requirement"
```

If a class template instead of a function has requirements, it is possible to check

```
1  template <typename SimpleStream>
2  struct some_class {
3    //[...]
4    private:
5      class_axiom_assert<has_get<SimpleStream, int> > check;
6  };
```

2.5.3 Concept-based overloading

If we wanted to have a default implementation for types not implementing `has_get`, we could write such functions instead:

```
template <typename SimpleStream, ENABLE_IF(has_get<SimpleStream, int>)>
int something_useless(SimpleStream&);

template <typename NotAStream, ENABLE_IF_NOT(has_get<NotAStream, int>),
          typename = void> // work-around to make signatures different
int something_useless(NotAStream&);
```

In this case, the first function would be used if and only if `SimpleStream` implemented `has_get`. The second would be used otherwise.

It is possible to do the same for classes:

```
template <typename SimpleStream, bool = IF(has_get<SimpleStream, int>)>
struct some_class {
  // [...]
};

template <typename SimpleStream>
struct some_class<SimpleStream, false> {
  // [...]
};
```

You have to note that we have used here only automatic concepts. A signature does not differentiate two types. Sometimes two different types with the same signature might need to have different version of an overloaded algorithm. For instance sorting an ordered container should not do anything whereas it should run a sorting algorithm in all the other containers. There would be however not much difference on the signature of the types. In this case we can use concepts.

```
template <typename C>
struct sorted_container: concept {
  typedef container<C> requirements;
};
```

Since `sorted_container` is not automatic, no type by default implements it.

We can then populate the concept by marking them as "verified" through type traits.

```
namespace catsfoot {
  template <typename T>
  struct verified<sorted_container<std::set<T> > >:
    public std::true_type {};
}
```

Now we can write our two differen versions:

```
template <typename C,
          ENABLE_IF(container<C>),
          ENABLE_IF_NOT(sorted_container<C>)>
void sort(C&) {
  // [...]
}

template <typename C,
          ENABLE_IF(sorted_container<C>)>
void sort(C&) {}
```

2.6 Axioms-only tutorial

In this tutorial, we introduce automatic testing utilities from Catsfoot for people who are not interested into concepts, or are just not familiar enough with C++ templates.

2.6.1 Axioms

Let's say we have a definition of integer exponentiation.

```
1  unsigned long
2  pow(unsigned long b,
3      unsigned short e) {
4    if (e == 0)
5      return 1;
6    auto rec = pow(b, e/2);
7    return ((e%2)?b:1)*rec*rec;
8  }
```

Axioms are functions, if we wanted to test Femat's theorem on native unsigned integers with the definition of , we would write some code like that:

```
1  void
2  fermat(unsigned long x, unsigned long y, unsigned long z,
3         unsigned short n) {
4    if ((n > 2) && (x > 0) && (y > 0))
5      axiom_assert(pow(x, n) + pow(y, n) != pow(z, n));
6  }
```

The assertion is probably actually not true in case of overflow. We will try to invistigate if it fails, and how.

An axiom is a C++ function containing assertions, eventually conditional. Nevertheless, the user is free to write any kind of code.

2.6.2 Generators

We want to generate data for the axiom. We need two types of values: `unsigned long` and `unsigned short`. One way is to give lists of values.

```
1  bool
2  simple_test() {
3    auto generator =
4      list_data_generator<unsigned long, unsigned short>(
5        {1ul, 2ul, 3ul, 4ul, 5ul, 6ul},
6        {3ul, 4ul, 5ul, 6ul});
7
8    if (!test(generator, fermat, "fermat"))
9      return false;
10
11   return true;
12 }
```

The test itself will print information in the error output. It will return true on success. It is then possible build a test suite and directly call the function.

The test will pass here. It does because it is not intensively testing the axiom. If we used random numbers, we actually discover that for some special values, an overflow will appear to put zeroes in the factors of the exponentiation. Thus, the return value is zero.

We can write a test using random. And we will find that some values break the axiom.

```
1  bool
2  random_test(std::mt19937& engine) {
3    auto generator =
4      term_generator_builder<unsigned long, unsigned short>(10u)
5      (engine,
6       std::function<unsigned long()>
7       ([&engine] () {
8         return std::uniform_int_distribution<unsigned long>()(engine);
9       }),
10      std::function<unsigned short()>
11      ([&engine] () {
12        return std::uniform_int_distribution<unsigned short>()(engine);
13      }));
```

The output will give us some information like:

```
file.cc:14: Axiom void fermat(long unsigned int, long unsigned int, long unsigned int, short

Expression was: pow(x, n) + pow(y, n) != pow(z, n)

Values were:
 * 12775342532353695410
 * 12775342532353695410
 * 8564872738844034446
 * 23302
```

Random generation of integer is not interesting as for the tutorial. So the example is not explained in detail. Basically we give two functions generating two different types.

2.6.2.1 Term generators

We can now start with a new example. We will use an >example coming from Haskell's Quickcheck. The example is a function taking a stream as input and returns a list of the first five characters which are in range from 'a' to 'e'.

```
1  std::string getList(std::istream& input) {
2    std::string ret = "";
3    unsigned count = 0;
4    char c;
5    while (input >> c) {
6      if ((c >= 'a') && (c <= 'e')) {
7        ret.push_back(c);
8        if (++count >= 5)
9          return ret;
10     }
11   }
12   return ret;
13 }
```

We can now write an axiom verifying that the list is always as small as 5 characters. We can also check that all characters are actually from the valid range. We can make several assertions in the same axiom.

```
1  void axiom(std::istream& s) {
2    std::string str = getList(s);
3    axiom_assert(str.length() <= 5);
4    for (auto c : str) {
5      axiom_assert((c >= 'a') && (c <= 'e'));
6    }
7  }
```

Now we can start to write our test. Generating random strings works well with term generation as strings with concatenation form a free monoid, i.e. for two string which have just been generated, concatenation will generate a string that was never generated. We need however to provide 1-letter strings as atoms. Since our algorithm care more about letters between `'a'` and `'e'`, we will influence the distribution.

So we start to write a string generator, we want to generate a kilo of them. We also need to proved pseudo-random engine.

```
1  bool test_getList(std::mt19937& engine) {
2    auto string_generator =
3      term_generator_builder<std::string>{1024u}
4    (engine,
```

Now we provide the first operation, the one that gives 1-letter strings. We try to generate more strings with the letter between `'a'` and `'e'`.

```
1      std::function<std::string()>
2      ([&engine] () {
3        char c;
4        if (std::uniform_int_distribution<int>(0,2)(engine))
5          c = std::uniform_int_distribution<char>('a', 'e')(engine);
6        else
7          c = std::uniform_int_distribution<char>()(engine);
8        return std::string(1u, c);
9      }),
```

Now we can provide the concatenation.

```
1      std::function<std::string(const std::string&,
2                                const std::string&)>
3      ([] (const std::string& a,
4           const std::string& b) { return a + b; }));
```

`string_generator` generates strings. However we need streams. We can make another generator and reuse easily `string_generator`.

There are few annoying things with generating streams. First, we need to generate `std::istream` which is abstract, which means not copyable, not move-able. The generator will detect it is not possible to store them. It is OK, we can make an operation that returns references of generator `std::istringstream`. The latter is normally move-able. It is not the case yet with GCC up to 4.6.0 unfortunately. So instead we will deal with pointers. `std::unique_ptr` can be used as it is movable, and it will be just enough to store the streams somewhere in the memory.

We start defining our generator reusing the previous generator.

```
1    auto generator =
2      term_generator_builder<std::string,
3                             std::unique_ptr<std::istringstream>,
4                             std::istream>{1024u}
```

```
5         (engine,
6          pick<std::string>(string_generator),
```

Now we provide a way to construct string streams.

```
1          std::function<std::unique_ptr<std::istringstream>(const
               std::string&)>
2          ([] (const std::string& s) ->
               std::unique_ptr<std::istringstream>
3          { return std::unique_ptr<std::istringstream>(new
               std::istringstream(s)); }
4         ),
```

Finally we provide a conversion from string streams to abstract streams.

```
1          std::function<std::istream&(std::unique_ptr<std::istringstream>&
               i)>
2          ([] (std::unique_ptr<std::istringstream>& i) -> std::istream&
3          { return *i; }));
```

This generator will unfortunately will never generate the empty string. But since our axiom has only we on parameter, we will just test it manually.

```
1    std::istringstream in{std::string{}};
2    if (!catch_errors(axiom, in))
3      return false;
```

Now we can run the test.

```
1    if (!test(generator, axiom, "axiom"))
2      return false;
3
4    return true;
5  }
```

2.7 Tutorial

2.7.1 Introduction

2.7.2 Writing concepts

To illustrate on how to write a concept, we start by showing how to write a simple specification of monoid. A monoid is a set with a binary operator. The set is under the operator. The operator is associative. There is also an identity element in the set.

```
1  template <typename T, typename Op, typename Id>
2  struct monoid: public concept {
3    typedef concept_list<
4      is_callable<Op(T, T)>,
5      is_callable<Id()>,
6      std::is_convertible<typename is_callable<Op(T, T)>::result_type,
           T>,
7      std::is_convertible<typename is_callable<Id()>::result_type, T>
8      > requirements;
9
10   static void associativity(const Op& op, const T& a,
11                             const T& b, const T& c) {
```

```
12      axiom_assert(op(a, op(b, c)) == op(op(a, b), c));
13    }
14
15    static void identity(const Op& op, const T& a, const Id& id) {
16      axiom_assert((op(id(), a) == a) && (op(a, id()) == a));
17    }
18
19    AXIOMS(associativity, identity);
20  };
```

Our concept is declared as a class inheriting from concept. It is used to make the difference from automatic concepts (which inherit from auto_concept and also from predicates (which do not inherit from any of those two special types).

We declare or requirements with a member type `requirements`. If we have several requirements we can pack them as parameters of template concept_list.

A requirement can be a concept, an automatic concept, or a predicate. In this example all requirements are predicates.

We require `Op` to be callable with two `T`s, while `Id` with no parameter. We also require that the result of those operations can be implicitly converted (no explicit cast) to `T`.

Then we have axioms describing the run-time behavior. Parameters are some kind of universal quantifier, any value of the given type should keep the axiom to hold.

Then to be able to automatize testing we describe what are the axioms we want to test by using macro AXIOMS.

Here we defined the carrier set and the two operators are declared as type parameters. The point of concepts is to be generic. We want to allow as much signature morphisms as possible. Of course we could have set the operators. But we can do it afterward, and reuse the most generic concept. For instance, we can define `monoid_plus` as such:

```
1  template <typename T>
2  struct monoid_plus: concept {
3    typedef
4      monoid<T, op_plus,
5             wrapped_constructor<T()> >
6      requirements;
7  };
```

For people familiar with C++, `op_plus` and `wrapped_constructor<T()>` are functor types. That is, they have an operator "parenthesis" to implement the operation. It is usually better to represent operations through a type rather than a function pointer, as the latter would not represent overloaded operations.

It is possible to write your own functor. Nevertheless you need to be aware of some pitfalls. op_plus is defined as:

```
1  struct op_plus {
2    template <typename T, typename U,
3              typename Ret = decltype(std::declval<T>() +
                   std::declval<U>())>
4    Ret operator()(T&& t, U&& u) const {
5      return std::forward<T>(t) + std::forward<U>(u);
6    }
7  };
```

Using `std::plus<T>` instead would have not given the same result for static concept checking matters. First `std::plus<T>` forces the arguments to be converted to `T` and the return type as well. op_plus does not. Second, predicate is_callable<std::plus<T>(T, T)> would be always true, no matter if an operator has been defined. However using the operator will lead to an error message and would not be caught before by concept checking. Predicate is_callable<op_plus(T, T)> on the other hand, will be true if only if an operator has been defined. The reason is that if failing to detect the return type through `decltype()` will just make the operator un-accessible. Virtually, the operator will not be defined. In that way, op_plus is a real alias to operator plus. Finally, op_plus represents the whole overloaded operator plus, whereas `std::plus<T>` only represents one version of the operator.

The same is done with the constructor. wrapped_constructor is defined as follow:

```
1  template <typename T, bool = is_constructible<T>::value>
2  struct wrapped_constructor;
3
4  template <typename T, typename... Args>
5  struct wrapped_constructor<T(Args...), true> {
6    T operator()(Args... args...) const {
7      return T(std::forward<Args>(args)...);
8    }
9  };
10
11 template <typename T, typename... Args>
12 struct wrapped_constructor<T(Args...), false> {
13 };
```

As you see, the template will select between an implementation with or without operator parenthesis depending on the

Catsfoot provides wrappers for operators, and also macros for generating wrappers for methods or overloaded functions. The user is invited to use them.

As a side note, we could say that nothing says that the default constructor gives us the neutral value, for example on native types like `int`. Well this is something good to test.

Since our concept `monoid` is not automatic, then we want to tell whether it holds on some set of types. This is a form of contract that the developer signs where he certifies that the axioms will hold. Of course, this can be wrong, but testing is here to determine that.

Automatic concepts do not need such contract. However, we do not have specified run-time behavior in automatic concepts (no axiom). Some overloaded function might need run-time behavior information to select optimize version.

The only thing these contracts are useful for is with run-time-behavior-based overloading. For example imagine we have an implementation of sum of a set of values in parallel playing on the fact that a monoid is associative, to be able to reorder priorities. The signature of such a function would be:

```
1  template <typename T,
2            ENABLE_IF(monoid_plus<T>)>
3  T sum(const std::vector<T>& v);
```

Of course, if our T is not a carrier set for a monoid, then we do not want to use this version of the function `sum` but a more classical that would use the priority represented

by the set. The only way for selecting the right function is to sign contracts asserting that implementations follow the run-time specifications of the concepts.

Even if your point is not to use this overloading feature, you will be required to sign the contracts, otherwise the static concept checking will fail. This is to make sure that other people will be able to use your concepts.

Those contracts are "signed" with type traits verified. Its parameter should be a concept instantiation (eventually partial). For example we can tell that integer along with operator plus and the default constructor (which might actually be wrong depending on the compiler) forms a `monoid`.

```
1 namespace catsfoot {
2   template <>
3   struct verified<monoid<int, op_plus, wrapped_constructor<int()> > >
4     : public std::true_type
5   {};
6 }
```

Now we can automatize signing those contracts using partial specialization. For instance, we want to say that for all verified `monoid` with the plus operator and the default constructor, then concept `monoid_plus` on the same type is verified.

```
1 namespace catsfoot {
2   template <typename T>
3   struct verified<monoid_plus<T> >
4     : public verified<monoid<T, op_plus, wrapped_constructor<int()> > >
5   {};
6 }
```

Type traits verified does not trigger any testing, but only allows it. Writing tests is still the responsibility of the user.

2.7.3 Automatic concepts and predicates

Automatic concepts and predicates have the same role, however they are defined in a different ways. They also behave differently when asserting compile-time requirements. Requiring an automatic concept when some of its requirements are missing will give error messages on specific requirements, whereas requirement a false predicate will result as a simple error message that this predicate is false. When composing several requirements it is better to write an automatic concept, as the user will get a more verbose error output.

A predicate is type with a constant member `value` convertible to a `bool`.

For example, we can write a predicate as such:

```
1 template <typename T>
2 struct is_lvalue_reference: public std::false_type {
3 };
4
5 template <typename T>
6 struct is_lvalue_reference<T&>: public std::true_type {
7 };
```

An automatic concept is basically like a concept. However there is no axiom. Any axiom would never be called by the test driver.x

```
template <typename T, typename Stream>
struct is_printable: public auto_concept {
  typedef concept_list<
    is_callable<op_lsh(Stream&, T)>,
    std::is_convertible<typename is_callable<op_lsh(Stream&, T)>
                        ::result_type, Stream&>
  > requirements;
};
```

2.7.4 Functions

2.7.4.1 Overloading function with requirement

Let's say we want to write a function `foo` that takes 3 arguments of any type as long as there is an operator +, that the type has a default constructor and that the type with operator+ and the default constructor forms a monoid. The following example shows an example of such a function.

```
template <typename T,
          typename NonRefT = typename std::decay<T>::type,
          ENABLE_IF(monoid<NonRefT, op_plus,
                           wrapped_constructor<NonRefT()> >)>
NonRefT foo(T&& a, T&& b, T&& c) {
  // (a * b);
  NonRefT ret = std::forward<T>(a) +
    std::forward<T>(b) + std::forward<T>(c);
  return ret;
}
```

Note that this selection is done with the last template parameter. Macro `ENABLE_IF` verifies like `IF` that the compile-time part of the concept holds. However it enables the version of the function instead of returning a Boolean.

It is possible to get errors if the number of parameters is the same. In this case it is possible to add parameters as `typename = void` in the parameter list.

2.7.4.2 Checking function definitions

If we ever un-commented the line using the multiplication operator, the compiler would not see it. It would actually compile if we gave a type `T` which has an operator $*$. We want to verify this.

A requirement for `some_concept<T, T>` is more specific than for `some_concept<T, U>`, then the archetype for some_concept<T, T> will need to be more specific. Any function will probably end up in some unique requirement.

Checking that a function has the right requirement is then still a hassle and uses a classic way of writing archetypes.

```
namespace foo_check {
  struct T {
    T() = default;
    T(const T&) = default;
    ~T() = default;
  };
```

```
7     T operator+(const T&, const T&) {
8       return T();
9     }
10    bool operator==(const T&, const T&) {
11      return true;
12    }
13  }
14
15  namespace catsfoot {
16    template <>
17    struct verified<monoid< ::foo_check::T, op_plus,
18                            wrapped_constructor< ::foo_check::T()> > >
19      : public std::true_type {
20    };
```

2.7.4.3 Static assertion of concepts

Sometimes we do not want to overload for different requirements. We just want to require a concept for any call. Using the previous method will just end up in the compiler claiming it did not find the function for the corresponding parameters. If there is no overloading we want instead to what are the requirements missing.

```
1  template <typename T,
2            typename NonRefT = typename std::decay<T>::type>
3  NonRefT foo(T&& a, T&& b, T&& c) {
4    assert_concept(monoid<NonRefT, op_plus,
5                          wrapped_constructor<NonRefT()> >());
6
7    NonRefT ret = std::forward<T>(a) +
8      std::forward<T>(b) + std::forward<T>(c);
9    return ret;
10 }
```

In this code we call assert_concept which will provide the right error message if the requirement is not satisfied.

2.7.5 Requirements on class templates

Instantiating class_assert_concept will have the same effect as calling assert_concept. To make sure that the assertion is instantiated in the same time as the class, then it is possible to either inherit from the assertion class or to use it as type of a dummy member. For example:

```
1  template <typename T>
2  struct Foo
3    : public class_assert_concept<monoid<T, op_plus,
4                                         wrapped_constructor<T()> > >
5  {};
```

2.7.5.1 Specialization and requirements

There is no elegant way to use a similar tool as `ENABLE_IF` for overloaded function templates as the number of parameters has to be fixed. Fortunately, it is possible, if we

know all the possible specializations when writing the general class template, to use a list of Boolean parameters.

```
template <typename T,
          bool specialize =
          IF(monoid<T, op_plus, wrapped_constructor<int()> >)>
struct Bar {
};

template <typename T>
struct Bar<T, true> {
};
```

2.7.6 Testing

Testing is about calling axioms which are just simple function. There is nothing complex in this. However tools are provided to call automatically the axioms. Those tools will need data set generators to provide input data to the axioms.

Note that the test programs still need to be written. Catsfoot is only a library. This way it allows Catsfoot to be used in any testing environment. Most common environments will expect you to write programs (with a function `main` on each). This is what you have to do. However, the only things you need to do in your testing functions are:

- to define data generators;
- to call test drivers with concepts (or just axioms) in combination with generators.

2.7.6.1 Writing axioms carefully

Each axiom is a function whose parameters are universally quantified variables. Any possible generated value of the type can be given to the axiom, and it will still hold.

The following axiom:

```
static void associativity(const Op& op, const T& a,
                          const T& b, const T& c) {
  axiom_assert(op(a, op(b, c)) == op(op(a, b), c));
}
```

Would translate into:

$$\forall op : OP. \forall a : T. \forall b : T. \forall c : T. op(a, op(b,c)) = op(op(a,b),c)$$

It is important to profit of the universal quantifiers from the axioms and let the data generator finds the values to be tested. It is tempting in some axioms to have local variables and generate random values locally. However the concept is decoupled from the implementations.

For a stack for example `s` is the same as `pop(push(s), some_value)`. On the implementation side, there might be a difference between the objects even though the equality operator claims they are the same. For instance, one might have more memory allocated than the other. Thus it is not enough to generate stacks only from "push" and

the initial (empty) stack. We could even go further, and use a concept for a stack onto a list. A list behaves as a stack. It does even with values that have been initialized in a list style (insertion in the middle for example). And it better have to, otherwise you would need encapsulation rather than using templates.

Since knowing how to generate terms needs the knowledge of the concrete type, it is not possible to write good axioms generating terms locally.

For example, do not write:

```
1 static void erasure(AssociativeContainer c, SizeType i) {
2  Iterator it = begin(c)+(i%size(c));
3  axiom_assert(size(erase(c, it)) == size(c) - 1);
4 }
```

But rather:

```
1 static void erasure(AssociativeContainer c, Iterator i) {
2   if ((i == find(c, i)) && (i != end(c)))
3     axiom_assert(size(erase(c, i)) == size(c) - 1);
4 }
```

First, `find(c,i)` is a precondition for `erase(c,i)`, and it has to show in the axiom. Second, we where generating ourselves iterators which is a bad thing. There are lots of other ways in a std::set for example an iterator can be made.

2.7.6.1.1 Universality of operators

It is important to use universal quantifier on operations. Some operators could have a state (for example some tables for optimizations) and it has to be tested.

Other point is that `int (*)(int, int)` is a valid type for the operator on a monoid for instance, but not all the pointers of function of this type behave as a monoid operation. You want to forbid the user to instantiate the concept with this kind of type. For that reason you need an universal quantifier on the operation.

2.7.6.2 Data generation

A data generator has an operation `Set get<T>()` which returns a data set for. The return type is a container of values of type `T&`. One could define its own generator in such a way:

```
1 struct my_int_generator {
2   std::vector<int> v;
3   my_int_generator(): v{1, 2, 3} {}
4   template <typename T, ENABLE_IF(is_same<T, int>)>
5   const std::vector<int>& get() {
6     return v;
7   }
8 };
```

2.7.6.2.1 List

It is possible to give a list of values given as initializer lists.

```
auto mygenerator = list_data_generator<int, float>
    ({-1, 0, 1, 2, 3,
        std::numeric_limits<int>::min(),
        std::numeric_limits<int>::max()},
     {.5, 42.,
        std::numeric_limits<float>::quiet_NaN(),
        std::numeric_limits<float>::denorm_min(),
        std::numeric_limits<float>::infinity()}));
```

2.7.6.2.2 Random terms

It is easy to generate random values for simple types. It get however more complex to generate them for a type which signature (all operations available for this type) is big. To be able to generate all kinds of values, the generator has to randomly call functions. For example building a random list is not only about inserting random elements. It is also erasing some.

Also several types have to be build along side. For example, lists should be generated in the same time as iterators and values. Especially if we want to activate conditional axioms like described in section Writing axioms carefully.

Random term generation is generic. The only thing that changes is the signature (the set of operations we can call). To generate those values.

Since some functions have preconditions, we wrap those functions. At the end we have to define the signature as a list of functions. Those functions must take parameters from the set of types we generate (it can be references), and it must return a type from the set of types.

If we want to generate list of integer we could write such a generator:

```
  auto int_list_generator =
    cxx_axioms::term_generator_builder<std::list<int>,
                                       std::list<int>::iterator,
                                       int>()
      (engine,
       std::function<int()>([&engine] () {
           return std::uniform_int_distribution<int>()(engine);
         }),
       constructor<std::list<int>()>(),
       disamb<const int&>()(&std::list<int>::push_back),
       disamb<const int&>()(&std::list<int>::push_front),
       std::function<int(const std::list<int>&)>
       ([] (const std::list<int>& in) {
         if (!in.empty())
           return int(in.front());
         return 0;
       }),
       std::function<int(const std::list<int>&)>
       ([] (const std::list<int>& in) {
         if (!in.empty())
           return int(in.back());
         return 0;
       }),
       disamb<>()(&std::list<int>::begin),
       disamb<>()(&std::list<int>::end),
       std::function<std::list<int>
                     (std::list<int>)>([] (std::list<int> in) {
```

```
28                          if (!in.empty())
29                            in.pop_back();
30                          return in;
31                        }),
32        std::function<std::list<int>
33                      (std::list<int>)>([] (std::list<int> in) {
34              if (!in.empty())
35                in.pop_front();
36              return in;
37                        })
38        );
```

2.7.6.2.3 Default constructor

Since operators are usually described as types, and since most of the time these operators will be using wrappers that do not have any state, and just have a default constructor, then it is convenient to just give a generator that returns the default constructor value.

```
1  default_generator mygenerator;
```

2.7.6.2.4 Union with left-priority

It is possible to use a combination of generators. Function choose will build a generate that choose the left-most generator that can generate the requested type.

For example if we want to generate integer from a set, and any other type from the default constructor, we will build such a generator:

```
1  auto mygenerator =
2    choose
3      (list_data_generator<int>
4        ({-1, 0, 1, 2, 3,
5          std::numeric_limits<int>::min(),
6          std::numeric_limits<int>::max()}),
7       default_generator{});
```

2.7.6.3 Calling test drivers

Now data generators are defined, it is time to call test drivers. There are two test drivers:

- test: tests an axiom (represented by a function).
- test_all: retrieves all axioms for a concepts (including inherited axioms from requirements).

2.7.6.3.1 On simple axioms

We can test axioms individually:

```
bool res =
  test(mygenerator,
       monoid<int, op_plus, wrapped_constructor<int()> >
         ::associativity,
       "monoid's associativity");
```

We can even test any function if it behaves as axioms.

2.7.6.3.2 On concepts

It is not very practical to test axioms one by one. Usually the user should prefer to test all the axioms required by a concept.

```
bool res =
  test_all(mygenerator,
           monoid<int, op_plus, wrapped_constructor<int()> >{});
```

2.7.6.3.3 Coverage

It is possible that some conditions are never met. For example, in the following axiom:

```
if ((i == find(c, i)) && (i != end(c)))
  axiom_assert(size(erase(c, i)) == size(c) - 1);
```

If iterator `i` is never found inside container `c`, the axiom is never be triggered. To be able to check it, we can run at the end of the program a function that will verify the coverage of conditions.

```
res &&= check_unverified();
```

It will report any axioms never covered, and return false if any were found.

Note, it might not always be wished to cover all the axioms. Sometimes conditions might be static:

```
if (std::atomic<T>::is_lock_free())
  axiom_assert(...);
```

The behavior of `std::atomic` is dependent to the architecture. This dependence is represented by member `is_lock_free`. So in this case we would like to have different axioms. But this condition is "static". Coverage checking will probably report this axiom, in the case where the condition is false.

There is an ugly work-around: use of block delimiters around conditional axioms can disable coverage checking. This is due to the definition of axiom_assert.

```
if (std::atomic<T>::is_lock_free()) {
  axiom_assert(...);
}
```

2.7.6.4 Test error messages and IDE

Error messages output by the library quite standard and should be already understood by any IDE. However, if you use "parallel-tests" in Automake which outputs is redi-

rected you need to tell your IDE that the log file is a file of error messages. With Emacs you can insert a mode selection as first line of the output of your test program:

```
1 std::cout << "-*- mode: compilation -*-" << std::endl;
```

GNU has a documentation page for Compilation mode.

2.8 README

Catsfoot is a C++ library providing:

- concept checking,
- concept-based overloadind,
- and generating tests automatically from concepts.

In short, it is intended to provide testing utilities for C++ template libraries.

Catsfoot is developed Bergen Language Design Laboratory.

More information is available on http://catsfoot.sf.net/

2.9 INSTALL

```
Installation Instructions
*************************

Copyright (C) 1994, 1995, 1996, 1999, 2000, 2001, 2002, 2004, 2005,
2006, 2007, 2008, 2009 Free Software Foundation, Inc.

   Copying and distribution of this file, with or without modification,
are permitted in any medium without royalty provided the copyright
notice and this notice are preserved.  This file is offered as-is,
without warranty of any kind.

Basic Installation
==================

   Briefly, the shell commands `./configure; make; make install' should
configure, build, and install this package.  The following
more-detailed instructions are generic; see the `README' file for
instructions specific to this package.  Some packages provide this
`INSTALL' file but do not implement all of the features documented
below.  The lack of an optional feature in a given package is not
necessarily a bug.  More recommendations for GNU packages can be found
in *note Makefile Conventions: (standards)Makefile Conventions.

   The `configure' shell script attempts to guess correct values for
various system-dependent variables used during compilation.  It uses
those values to create a `Makefile' in each directory of the package.
It may also create one or more `.h' files containing system-dependent
definitions.  Finally, it creates a shell script `config.status' that
you can run in the future to recreate the current configuration, and a
file `config.log' containing compiler output (useful mainly for
```

debugging 'configure').

It can also use an optional file (typically called 'config.cache' and enabled with '--cache-file=config.cache' or simply '-C') that saves the results of its tests to speed up reconfiguring. Caching is disabled by default to prevent problems with accidental use of stale cache files.

If you need to do unusual things to compile the package, please try to figure out how 'configure' could check whether to do them, and mail diffs or instructions to the address given in the 'README' so they can be considered for the next release. If you are using the cache, and at some point 'config.cache' contains results you don't want to keep, you may remove or edit it.

The file 'configure.ac' (or 'configure.in') is used to create 'configure' by a program called 'autoconf'. You need 'configure.ac' if you want to change it or regenerate 'configure' using a newer version of 'autoconf'.

The simplest way to compile this package is:

1. 'cd' to the directory containing the package's source code and type './configure' to configure the package for your system.

   Running 'configure' might take a while. While running, it prints some messages telling which features it is checking for.

2. Type 'make' to compile the package.

3. Optionally, type 'make check' to run any self-tests that come with the package, generally using the just-built uninstalled binaries.

4. Type 'make install' to install the programs and any data files and documentation. When installing into a prefix owned by root, it is recommended that the package be configured and built as a regular user, and only the 'make install' phase executed with root privileges.

5. Optionally, type 'make installcheck' to repeat any self-tests, but this time using the binaries in their final installed location. This target does not install anything. Running this target as a regular user, particularly if the prior 'make install' required root privileges, verifies that the installation completed correctly.

6. You can remove the program binaries and object files from the source code directory by typing 'make clean'. To also remove the files that 'configure' created (so you can compile the package for a different kind of computer), type 'make distclean'. There is also a 'make maintainer-clean' target, but that is intended mainly for the package's developers. If you use it, you may have to get all sorts of other programs in order to regenerate files that came with the distribution.

7. Often, you can also type 'make uninstall' to remove the installed files again. In practice, not all packages have tested that uninstallation works correctly, even though it is required by the GNU Coding Standards.

8. Some packages, particularly those that use Automake, provide 'make distcheck', which can by used by developers to test that all other

```
     targets like ‘make install’ and ‘make uninstall’ work correctly.
     This target is generally not run by end users.

Compilers and Options
=====================

   Some systems require unusual options for compilation or linking that
the ‘configure’ script does not know about.  Run ‘./configure --help’
for details on some of the pertinent environment variables.

   You can give ‘configure’ initial values for configuration parameters
by setting variables in the command line or in the environment.  Here
is an example:

     ./configure CC=c99 CFLAGS=-g LIBS=-lposix

   *Note Defining Variables::, for more details.

Compiling For Multiple Architectures
====================================

   You can compile the package for more than one kind of computer at the
same time, by placing the object files for each architecture in their
own directory.  To do this, you can use GNU ‘make’.  ‘cd’ to the
directory where you want the object files and executables to go and run
the ‘configure’ script.  ‘configure’ automatically checks for the
source code in the directory that ‘configure’ is in and in ‘..’.  This
is known as a "VPATH" build.

   With a non-GNU ‘make’, it is safer to compile the package for one
architecture at a time in the source code directory.  After you have
installed the package for one architecture, use ‘make distclean’ before
reconfiguring for another architecture.

   On MacOS X 10.5 and later systems, you can create libraries and
executables that work on multiple system types--known as "fat" or
"universal" binaries--by specifying multiple ‘-arch’ options to the
compiler but only a single ‘-arch’ option to the preprocessor.  Like
this:

     ./configure CC="gcc -arch i386 -arch x86_64 -arch ppc -arch ppc64" \
                 CXX="g++ -arch i386 -arch x86_64 -arch ppc -arch ppc64" \
                 CPP="gcc -E" CXXCPP="g++ -E"

   This is not guaranteed to produce working output in all cases, you
may have to build one architecture at a time and combine the results
using the ‘lipo’ tool if you have problems.

Installation Names
==================

   By default, ‘make install’ installs the package’s commands under
‘/usr/local/bin’, include files under ‘/usr/local/include’, etc.  You
can specify an installation prefix other than ‘/usr/local’ by giving
‘configure’ the option ‘--prefix=PREFIX’, where PREFIX must be an
absolute file name.

   You can specify separate installation prefixes for
architecture-specific files and architecture-independent files.  If you
pass the option ‘--exec-prefix=PREFIX’ to ‘configure’, the package uses
PREFIX as the prefix for installing programs and libraries.
Documentation and other data files still use the regular prefix.
```

```
   In addition, if you use an unusual directory layout you can give
options like '--bindir=DIR' to specify different values for particular
kinds of files.  Run 'configure --help' for a list of the directories
you can set and what kinds of files go in them.  In general, the
default for these options is expressed in terms of '${prefix}', so that
specifying just '--prefix' will affect all of the other directory
specifications that were not explicitly provided.

   The most portable way to affect installation locations is to pass the
correct locations to 'configure'; however, many packages provide one or
both of the following shortcuts of passing variable assignments to the
'make install' command line to change installation locations without
having to reconfigure or recompile.

   The first method involves providing an override variable for each
affected directory.  For example, 'make install
prefix=/alternate/directory' will choose an alternate location for all
directory configuration variables that were expressed in terms of
'${prefix}'.  Any directories that were specified during 'configure',
but not in terms of '${prefix}', must each be overridden at install
time for the entire installation to be relocated.  The approach of
makefile variable overrides for each directory variable is required by
the GNU Coding Standards, and ideally causes no recompilation.
However, some platforms have known limitations with the semantics of
shared libraries that end up requiring recompilation when using this
method, particularly noticeable in packages that use GNU Libtool.

   The second method involves providing the 'DESTDIR' variable.  For
example, 'make install DESTDIR=/alternate/directory' will prepend
'/alternate/directory' before all installation names.  The approach of
'DESTDIR' overrides is not required by the GNU Coding Standards, and
does not work on platforms that have drive letters.  On the other hand,
it does better at avoiding recompilation issues, and works well even
when some directory options were not specified in terms of '${prefix}'
at 'configure' time.

Optional Features
=================

   If the package supports it, you can cause programs to be installed
with an extra prefix or suffix on their names by giving 'configure' the
option '--program-prefix=PREFIX' or '--program-suffix=SUFFIX'.

   Some packages pay attention to '--enable-FEATURE' options to
'configure', where FEATURE indicates an optional part of the package.
They may also pay attention to '--with-PACKAGE' options, where PACKAGE
is something like 'gnu-as' or 'x' (for the X Window System).  The
'README' should mention any '--enable-' and '--with-' options that the
package recognizes.

   For packages that use the X Window System, 'configure' can usually
find the X include and library files automatically, but if it doesn't,
you can use the 'configure' options '--x-includes=DIR' and
'--x-libraries=DIR' to specify their locations.

   Some packages offer the ability to configure how verbose the
execution of 'make' will be.  For these packages, running './configure
--enable-silent-rules' sets the default to minimal output, which can be
overridden with 'make V=1'; while running './configure
--disable-silent-rules' sets the default to verbose, which can be
overridden with 'make V=0'.
```

```
Particular systems
==================

   On HP-UX, the default C compiler is not ANSI C compatible.  If GNU
CC is not installed, it is recommended to use the following options in
order to use an ANSI C compiler:

     ./configure CC="cc -Ae -D_XOPEN_SOURCE=500"

and if that doesn't work, install pre-built binaries of GCC for HP-UX.

   On OSF/1 a.k.a. Tru64, some versions of the default C compiler cannot
parse its '<wchar.h>' header file.  The option '-nodtk' can be used as
a workaround.  If GNU CC is not installed, it is therefore recommended
to try

     ./configure CC="cc"

and if that doesn't work, try

     ./configure CC="cc -nodtk"

   On Solaris, don't put '/usr/ucb' early in your 'PATH'.  This
directory contains several dysfunctional programs; working variants of
these programs are available in '/usr/bin'.  So, if you need '/usr/ucb'
in your 'PATH', put it _after_ '/usr/bin'.

   On Haiku, software installed for all users goes in '/boot/common',
not '/usr/local'.  It is recommended to use the following options:

     ./configure --prefix=/boot/common

Specifying the System Type
==========================

   There may be some features 'configure' cannot figure out
automatically, but needs to determine by the type of machine the package
will run on.  Usually, assuming the package is built to be run on the
_same_ architectures, 'configure' can figure that out, but if it prints
a message saying it cannot guess the machine type, give it the
'--build=TYPE' option.  TYPE can either be a short name for the system
type, such as 'sun4', or a canonical name which has the form:

     CPU-COMPANY-SYSTEM

where SYSTEM can have one of these forms:

     OS
     KERNEL-OS

   See the file 'config.sub' for the possible values of each field.  If
'config.sub' isn't included in this package, then this package doesn't
need to know the machine type.

   If you are _building_ compiler tools for cross-compiling, you should
use the option '--target=TYPE' to select the type of system they will
produce code for.

   If you want to _use_ a cross compiler, that generates code for a
platform different from the build platform, you should specify the
"host" platform (i.e., that on which the generated programs will
```

```
eventually be run) with '--host=TYPE'.

Sharing Defaults
================

   If you want to set default values for 'configure' scripts to share,
you can create a site shell script called 'config.site' that gives
default values for variables like 'CC', 'cache_file', and 'prefix'.
'configure' looks for 'PREFIX/share/config.site' if it exists, then
'PREFIX/etc/config.site' if it exists.  Or, you can set the
'CONFIG_SITE' environment variable to the location of the site script.
A warning: not all 'configure' scripts look for a site script.

Defining Variables
==================

   Variables not defined in a site shell script can be set in the
environment passed to 'configure'.  However, some packages may run
configure again during the build, and the customized values of these
variables may be lost.  In order to avoid this problem, you should set
them in the 'configure' command line, using 'VAR=value'.  For example:

     ./configure CC=/usr/local2/bin/gcc

causes the specified 'gcc' to be used as the C compiler (unless it is
overridden in the site shell script).

Unfortunately, this technique does not work for 'CONFIG_SHELL' due to
an Autoconf bug.  Until the bug is fixed you can use this workaround:

     CONFIG_SHELL=/bin/bash /bin/bash ./configure CONFIG_SHELL=/bin/bash

'configure' Invocation
======================

   'configure' recognizes the following options to control how it
operates.

'--help'
'-h'
     Print a summary of all of the options to 'configure', and exit.

'--help=short'
'--help=recursive'
     Print a summary of the options unique to this package's
     'configure', and exit.  The 'short' variant lists options used
     only in the top level, while the 'recursive' variant lists options
     also present in any nested packages.

'--version'
'-V'
     Print the version of Autoconf used to generate the 'configure'
     script, and exit.

'--cache-file=FILE'
     Enable the cache: use and save the results of the tests in FILE,
     traditionally 'config.cache'.  FILE defaults to '/dev/null' to
     disable caching.

'--config-cache'
'-C'
     Alias for '--cache-file=config.cache'.
```

```
‘--quiet’
‘--silent’
‘-q’
     Do not print messages saying which checks are being made.  To
     suppress all normal output, redirect it to ‘/dev/null’ (any error
     messages will still be shown).

‘--srcdir=DIR’
     Look for the package’s source code in directory DIR.  Usually
     ‘configure’ can determine that directory automatically.

‘--prefix=DIR’
     Use DIR as the installation prefix.  *note Installation Names::
     for more details, including other options available for fine-tuning
     the installation locations.

‘--no-create’
‘-n’
     Run the configure checks, but stop before creating any output
     files.

‘configure’ also accepts some other, not widely useful, options.  Run
‘configure --help’ for more details.
```

2.10 NEWS

2.10.1 0.1

Intitial release.

2.11 COPYING

```
   GNU LESSER GENERAL PUBLIC LICENSE
                       Version 3, 29 June 2007

 Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/>
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.


  This version of the GNU Lesser General Public License incorporates
the terms and conditions of version 3 of the GNU General Public
License, supplemented by the additional permissions listed below.

  0. Additional Definitions.

  As used herein, "this License" refers to version 3 of the GNU Lesser
General Public License, and the "GNU GPL" refers to version 3 of the GNU
General Public License.

  "The Library" refers to a covered work governed by this License,
other than an Application or a Combined Work as defined below.

  An "Application" is any work that makes use of an interface provided
```

```
by the Library, but which is not otherwise based on the Library.
Defining a subclass of a class defined by the Library is deemed a mode
of using an interface provided by the Library.

  A "Combined Work" is a work produced by combining or linking an
Application with the Library.  The particular version of the Library
with which the Combined Work was made is also called the "Linked
Version".

  The "Minimal Corresponding Source" for a Combined Work means the
Corresponding Source for the Combined Work, excluding any source code
for portions of the Combined Work that, considered in isolation, are
based on the Application, and not on the Linked Version.

  The "Corresponding Application Code" for a Combined Work means the
object code and/or source code for the Application, including any data
and utility programs needed for reproducing the Combined Work from the
Application, but excluding the System Libraries of the Combined Work.

  1. Exception to Section 3 of the GNU GPL.

  You may convey a covered work under sections 3 and 4 of this License
without being bound by section 3 of the GNU GPL.

  2. Conveying Modified Versions.

  If you modify a copy of the Library, and, in your modifications, a
facility refers to a function or data to be supplied by an Application
that uses the facility (other than as an argument passed when the
facility is invoked), then you may convey a copy of the modified
version:

   a) under this License, provided that you make a good faith effort to
   ensure that, in the event an Application does not supply the
   function or data, the facility still operates, and performs
   whatever part of its purpose remains meaningful, or

   b) under the GNU GPL, with none of the additional permissions of
   this License applicable to that copy.

  3. Object Code Incorporating Material from Library Header Files.

  The object code form of an Application may incorporate material from
a header file that is part of the Library.  You may convey such object
code under terms of your choice, provided that, if the incorporated
material is not limited to numerical parameters, data structure
layouts and accessors, or small macros, inline functions and templates
(ten or fewer lines in length), you do both of the following:

   a) Give prominent notice with each copy of the object code that the
   Library is used in it and that the Library and its use are
   covered by this License.

   b) Accompany the object code with a copy of the GNU GPL and this license
   document.

  4. Combined Works.

  You may convey a Combined Work under terms of your choice that,
taken together, effectively do not restrict modification of the
portions of the Library contained in the Combined Work and reverse
engineering for debugging such modifications, if you also do each of
```

```
the following:

   a) Give prominent notice with each copy of the Combined Work that
   the Library is used in it and that the Library and its use are
   covered by this License.

   b) Accompany the Combined Work with a copy of the GNU GPL and this license
   document.

   c) For a Combined Work that displays copyright notices during
   execution, include the copyright notice for the Library among
   these notices, as well as a reference directing the user to the
   copies of the GNU GPL and this license document.

   d) Do one of the following:

       0) Convey the Minimal Corresponding Source under the terms of this
       License, and the Corresponding Application Code in a form
       suitable for, and under terms that permit, the user to
       recombine or relink the Application with a modified version of
       the Linked Version to produce a modified Combined Work, in the
       manner specified by section 6 of the GNU GPL for conveying
       Corresponding Source.

       1) Use a suitable shared library mechanism for linking with the
       Library.  A suitable mechanism is one that (a) uses at run time
       a copy of the Library already present on the user's computer
       system, and (b) will operate properly with a modified version
       of the Library that is interface-compatible with the Linked
       Version.

   e) Provide Installation Information, but only if you would otherwise
   be required to provide such information under section 6 of the
   GNU GPL, and only to the extent that such information is
   necessary to install and execute a modified version of the
   Combined Work produced by recombining or relinking the
   Application with a modified version of the Linked Version. (If
   you use option 4d0, the Installation Information must accompany
   the Minimal Corresponding Source and Corresponding Application
   Code. If you use option 4d1, you must provide the Installation
   Information in the manner specified by section 6 of the GNU GPL
   for conveying Corresponding Source.)

  5. Combined Libraries.

  You may place library facilities that are a work based on the
Library side by side in a single library together with other library
facilities that are not Applications and are not covered by this
License, and convey such a combined library under terms of your
choice, if you do both of the following:

   a) Accompany the combined library with a copy of the same work based
   on the Library, uncombined with any other library facilities,
   conveyed under the terms of this License.

   b) Give prominent notice with the combined library that part of it
   is a work based on the Library, and explaining where to find the
   accompanying uncombined form of the same work.

  6. Revised Versions of the GNU Lesser General Public License.

  The Free Software Foundation may publish revised and/or new versions
```

2.12 AUTHORS

- Valentin David <valentin@ii.uib.no>

Chapter 3

Download

Stable releases are available on this page.

Unstable pre-releases might be available from the nightly tarball repository
if the build farm is being nice enough.

The last option is to use the svn repository https://catsfoot.svn.sourceforge.net/svnroot/catsfoot/t
. You will then need recent versions of Autoconf and Automake.

Chapter 4

Support

Contents

4.1 Mailing-lists

To subscribe to any of those mailing lists, just send

- catsfoot-bugs@lists.sourceforge.net
- catsfoot-svn@lists.sourceforge.net
- catsfoot-users@lists.sourceforge.net

4.2 Bug tracker

Please send bug reports on the sourceforge page (quick link).

Chapter 5

Publications

Contents

Anya Helene Bagge, Valentin David, and Magne Haveraaen. 2009. **The axioms strike back: testing with concepts and axioms in C++.** SIGPLAN Not. 45, 2 (October 2009), 15-24. DOI=10.1145/1837852.1621612.

Anya Helene Bagge, Valentin David, and Magne Haveraaen. 2011. **Testing with Axioms in C++ 2011: Testing with Axioms in C++ 2011**. Journal of Object Technology, Volume 10, (2011), pp. 10:1-32. DOI=10.5381/jot.2011.10.1.a10. Figures.

5.1 Figures

This is the exact source code used as figure in submitted article **Testing with Axioms in C++ 2011: Testing with Axioms in C++ 2011**. Each listing is delimited by a "\start" and an "\end" command.

```
1  //%% Use non-used C++ characters for block.
2  //\catcode `\}=9
3  //\catcode `\{=9
4  //\catcode `\$=2
5  //\catcode `\`=1
6  //\ignoreinput
7
8  // Checked with r49
9
10 #include <catsfoot_testing.hh>
11
12 namespace testerns {
13   using namespace catsfoot;
14
15 template <typename... T>
16 struct tester {
17 };
18 //\start `tester$
19 template <>
20 struct tester<> {
```

```
21    template <typename Generator, typename Fun, typename... Params>
22    static bool call(Generator, Fun f, Params... values) {
23      try {
24        f(values...);
25        return true;
26      } catch (axiom_failure af) {
27        return false;
28      }
29    }
30  };
31
32  template <typename T, typename... U>
33  struct tester<T, U...> {
34    template <typename Generator, typename Fun, typename... Params>
35    static bool call(Generator g, Fun f, Params... values) {
36      auto container = g.get(selector<T>());
37      for (auto i = container.begin();
38           i != container.end(); ++i) {
39        if (!tester<U...>::call(g, f, values..., *i))
40          return false;
41      }
42      return true;
43    }
44  };
45
46  template <typename Generator,
47            typename... T>
48  bool test(Generator g, void f(T...)) {
49    return tester<T...>::call(g, f);
50  }
51  //\end
52
53    void axiom(int a, int b, int c) {
54      if ((a == b) && (b == c))
55        axiom_assert(a == c);
56    }
57
58    void axiom_fail(int a, int b, int c) {
59      if ((a == b) && (b == c))
60        axiom_assert(a != c);
61    }
62
63    bool dotest() {
64      auto generator =
65        choose(list_data_generator<int>
66               ({0, 1, 2, -1, 41, 52}),
67               default_generator());
68      if ( ::testerns::test(generator, axiom_fail))
69        return false;
70      return ::testerns::test(generator, axiom);
71    }
72  }
73
74  namespace equivalencens {
75    using namespace catsfoot;
76
77  //\start `equivalence$
78  template <typename T, typename Rel>
79  struct equivalence: public concept {
80    typedef concept_list<
81      is_callable<Rel(T, T)>,
82      std::is_convertible<typename is_callable<Rel(T, T)>::result_type,
```

```
83                          bool>
84       > requirements;
85     static void reflexivity(const Rel& rel, const T& a) {
86       axiom_assert(rel(a,a));
87     }
88     static void symmetry(const Rel& rel, const T& a, const T& b) {
89       if (rel(a, b))
90         axiom_assert(rel(b, a));
91     }
92     static void transitivity(const Rel& rel,
93                              const T& a, const T& b, const T& c) {
94       if (rel(a, b) && rel(b, c))
95         axiom_assert(rel(a, c));
96     }
97     AXIOMS(reflexivity, symmetry, transitivity);
98   };
99   //\end
100
101  #ifdef SHOULDBE
102  //\start 'congruence$
103  template <typename Rel, typename Op, typename... Args>
104  struct congruence: public concept {
105    typedef concept_list<
106      equivalence<Args, Rel>...,
107      is_callable<Op(Args...)>,
108      equivalence<typename is_callable<Op(Args...)>::result_type,
109                  Rel>
110      > requirements;
111    static void congruence_axiom(const Args&... args1, const Args&...
         args2,
112                                 const Op& op, const Rel& rel) {
113      if (rel(args1, args2)...)
114        axiom_assert(rel(op(args1...), op(args2...)));
115    }
116    AXIOMS(congruence_axiom);
117  };
118  //\end
119  #else
120  template <typename Rel, typename Op, typename... Args>
121  struct congruence: public concept {
122    typedef concept_list<
123      equivalence<Args, Rel>...,
124      is_callable<Op(Args...)>,
125      equivalence<typename is_callable<Op(Args...)>::result_type,
126                  Rel>
127      > requirements;
128    static void congruence_axiom(const std::tuple<Args...>& args1,
129                                 const std::tuple<Args...>& args2,
130                                 const Op& op,
131                                 const Rel& rel) {
132      if (tuple_rel(rel,args1, args2))
133        axiom_assert(rel(call_with(op,args1), call_with(op,args2)));
134    }
135    AXIOMS(congruence_axiom);
136  };
137  #endif
138
139  }
140
141  # include <drivers/test_driver.hh>
142  # include <drivers/test_all_driver.hh>
143  # include <dataset/choose.hh>
```

```
144
145 namespace equivalencens {
146   bool f() {
147     auto generator =
148       choose(list_data_generator<int>
149              ({0, 1, 2, -1, 41, 52}),
150              default_generator());
151 return
152 //\start`testall$
153 test_all<congruence<op_eq, op_plus, int, int> >(generator);
154 //\end
155   }
156
157 }
158
159 namespace wrapped {
160 //\start`wrappingmethods$
161 struct foo_wrapped {
162   template<typename T, typename... Args,
163            typename Ret = decltype(std::declval<T>()
164                                    .foo(std::declval<Args>()...))>
165   Ret operator()(T&& object, Args&&... args) const {
166     return std::forward<T>(object).foo(std::forward<Args>(args)...);
167   }
168 };
169
170 struct foo_functionalized {
171   template<typename T, typename... Args,
172            typename = decltype(std::declval<T>()
173                                .foo(std::declval<Args>()...))>
174   T operator()(const T& object, Args&&... args) const {
175     T ret(object);
176     ret.foo(std::forward<Args>(args)...);
177     return ret;
178   }
179 };
180 //\end
181
182   struct A {
183     bool flag;
184     A(): flag(false) {}
185     A(const A& other): flag(other.flag) {
186     }
187
188     A* clone() {
189       return new A(*this);
190     }
191
192     void foo() {
193       flag = true;
194     }
195   };
196
197   bool test() {
198     A a;
199     A other = foo_functionalized()(a);
200     if (!(other.flag && !a.flag))
201       return false;
202     foo_wrapped()(a);
203     return other.flag && a.flag;
204   }
205
```

```
206 }
207
208 namespace wrapped2 {
209   template <typename T>
210   struct clone_ptr {
211   private:
212     T* ptr;
213   public:
214     ~clone_ptr() {
215       delete ptr;
216     }
217
218     template <typename... Args>
219     explicit
220     clone_ptr(Args&&... args): ptr(new T(std::forward<Args>(args)...))
          {}
221
222     explicit clone_ptr(T* ptr): ptr(ptr) {}
223
224     clone_ptr(const T& t): ptr(t->clone()) {}
225
226     template <typename U,
227               typename = typename
228               std::enable_if<std::is_base_of<T, U>::value>::type>
229     clone_ptr(const clone_ptr<T>& other):
230       ptr(other.ptr->clone()) {
231     }
232
233     clone_ptr(const clone_ptr& other):
234       ptr(other.ptr->clone()) {
235     }
236
237     template <typename U,
238               typename = typename
239               std::enable_if<std::is_base_of<T, U>::value>::type>
240     clone_ptr(clone_ptr<U>&& other):
241       ptr(other.ptr) {
242       other.ptr = NULL;
243     }
244
245     clone_ptr(clone_ptr&& other):
246       ptr(other.ptr) {
247       other.ptr = NULL;
248     }
249
250     T* get() {
251       return ptr;
252     }
253
254     const T* get() const {
255       return ptr;
256     }
257
258     T& operator*() {
259       return *get();
260     }
261
262     const T& operator*() const {
263       return *get();
264     }
265
266     T* operator->() {
```

```
267       return get();
268     }
269
270     const T* operator->() const {
271       return get();
272     }
273   };
274
275 //\start`inheritanceclone$
276 struct foo_functionalized {
277   template<typename T, typename... Args,
278            typename = decltype(std::declval<T>()
279                                .foo(std::declval<Args>()...))>
280   clone_ptr<T> operator()(clone_ptr<T> object, Args&&... args) const {
281     object->foo(std::forward<Args>(args)...);
282     return std::move(object);
283   }
284 };
285 //\end
286   struct A {
287     bool flag;
288     A(): flag(false) {}
289     A(const A& other): flag(other.flag) {
290     }
291
292     A* clone() {
293       return new A(*this);
294     }
295
296     void foo() {
297       flag = true;
298     }
299   };
300
301   bool test() {
302     clone_ptr<A> a;
303     clone_ptr<A> other = foo_functionalized()(a);
304     return other->flag && !a->flag;
305   }
306 }
307
308 namespace codeforannotation {
309   using namespace catsfoot;
310   template <typename T>
311   struct verified;
312
313   template <typename T>
314   struct container;
315
316   template <typename T>
317   struct plus_monoid;
318
319 //\start`codeforannotation$
320 template <typename T, typename Op, typename Id>
321 struct monoid: public concept {
322   typedef concept_list<
323     is_callable<Op(T, T)>,
324     is_callable<Id()>,
325     std::is_convertible<typename is_callable<Op(T, T)>
326                         ::result_type, T>,
327     std::is_convertible<typename is_callable<Id()>
328                         ::result_type, T>
```

```
329       > requirements;
330
331     static void associativity(const Op& op, const T& a,
332                               const T& b, const T& c) {
333       axiom_assert(op(a, op(b, c)) == op(op(a, b), c));
334     }
335
336     static void identity(const Op& op, const T& a, const Id& id) {
337       axiom_assert((op(id(), a) == a) && (op(a, id()) == a));
338     }
339
340     AXIOMS(associativity, identity);
341   };
342
343   // This predicate is a example of predicate coming from the
344   // standard library.
345   template <typename T>
346   struct is_lvalue_reference: public std::false_type {
347   };
348
349   template <typename T>
350   struct is_lvalue_reference<T&>: public std::true_type {
351   };
352
353
354   template <>
355   struct verified<monoid<int, op_plus, constant<int, 0> > >
356    : public std::true_type
357   {};
358
359   template <typename C,
360             ENABLE_IF(container<C>),
361             typename T = typename container<C>::element_type,
362             ENABLE_IF(plus_monoid<T>)>
363   T sum(C set);
364   //\end
365   }
366
367   #include <limits>
368   #include <dataset/dataset.hh>
369
370   bool g()
371   {
372     using namespace catsfoot;
373   //\start 'static_list$
374   auto generator =
375     choose
376     (list_data_generator<int>
377      ({-1, 0, 1, 2, 3,
378          std::numeric_limits<int>::min(),
379          std::numeric_limits<int>::max()}),
380      default_generator());
381   //\end
382    generator.get(selector<int>{});
383    return true;
384   }
385
386   #include <dataset/random_term_generator.hh>
387
388   #include <set>
389
390   bool h()
```

```
391 {
392   using namespace catsfoot;
393   std::mt19937 engine;
394
395
396 //\start 'randomterm$
397 auto int_set_generator =
398   term_generator_builder
399     <std::set<int>,                  // list of supported types
400       std::set<int>::iterator,
401       int>()
402   (engine,                           // random generator engine
403
404    // generate random integers
405    std::function<int()>([&engine] () {
406      return std::uniform_int_distribution<int>()(engine);
407    }),
408
409    // generate set: initial
410    constructor<std::set<int>()>(),
411
412    // generate set: insert
413    std::function<std::set<int>(std::set<int>, int)>
414    ([](std::set<int> in, int i) {
415        in.insert(i);
416        return std::move(in);
417      }),
418
419    // generate random iterator for a given set
420    std::function<std::set<int>::iterator(std::set<int>&)>
421    ([&engine] (std::set<int>& s) {
422       auto n = std::uniform_int_distribution<decltype(s.size())>
423         (0, s.size())(engine);
424       auto i = s.begin();
425       for (decltype(n) j = 0; j < n; ++j, ++i) ;
426       return i; // will point to random element in s
427    }));
428 //\end
429  int_set_generator.get(selector<int>{});
430
431 //\start 'powerset$
432 typedef std::set<int> elt;
433 auto power_set_generator =
434   term_generator_builder
435     <std::set<elt>,                  // list of supported types
436       std::set<elt>::iterator,
437       elt>()
438   (engine,                           // random generator engine
439
440    // generate random sets
441    pick<elt>(int_set_generator),
442
443    // generate set: initial
444    constructor<std::set<elt>()>(),
445
446    // generate set: insert
447    std::function<std::set<elt>(std::set<elt>, const elt&)>
448    ([](std::set<elt> in, const elt& i) {
449        in.insert(i);
450        return std::move(in);
451      }),
452
```

```
453     // generate random iterator for a given set
454     std::function<std::set<elt>::iterator(std::set<elt>&)>
455     ([&engine] (std::set<elt>& s) {
456        auto n = std::uniform_int_distribution<decltype(s.size())>
457          (0, s.size())(engine);
458        auto i = s.begin();
459        for (decltype(n) j = 0; j < n; ++j, ++i) ;
460        return i; // will point to random element in s
461      }));
462   //\end
463
464    power_set_generator.get(selector<std::set<int> >{});
465
466    return true;
467   }
468
469   template <typename AssociativeContainer, typename Iterator>
470   struct something {
471     //\start`erasure$
472   static void erasure(AssociativeContainer c, Iterator i) {
473     if ((i == find(c, i)) && (i != end(c)))
474       axiom_assert(size(erase(c, i)) == size(c) - 1);
475   }
476     //\end
477   };
478
479   namespace monoiddef {
480     using namespace catsfoot;
481
482   //\start`monoid$
483   template <typename T, typename Op, typename Id>
484   struct monoid: public concept {
485     typedef concept_list<
486       is_callable<Op(T, T)>,
487       is_callable<Id()>,
488       std::is_convertible<typename is_callable<Op(T, T)>
489                           ::result_type, T>,
490       std::is_convertible<typename is_callable<Id()>
491                           ::result_type, T>
492       > requirements;
493
494     static void associativity(const Op& op, const T& a,
495                               const T& b, const T& c) {
496       axiom_assert(op(a, op(b, c)) == op(op(a, b), c));
497     }
498
499     static void identity(const Op& op, const T& a, const Id& id) {
500       axiom_assert((op(id(), a) == a) && (op(a, id()) == a));
501     }
502
503     AXIOMS(associativity, identity);
504   };
505   //\end
506   }
507
508   //\start`predicate_dflt$
509   template <typename T>
510   struct is_lvalue_reference: public std::false_type {
511   };
512   //\end
513
514   //\start`predicate$
```

```
515 template <typename T>
516 struct is_lvalue_reference<T&>: public std::true_type {
517 };
518 //\end
519
520 namespace catsfoot {
521   using namespace monoiddef;
522 //\start`verified_monoid$
523 template <>
524 struct verified<monoid<int, op_plus, constant<int, 0> > >
525  : public std::true_type
526 {};
527 //\end
528 }
529
530 namespace monoiddef {
531   template <typename T>
532   struct verified {};
533 //\start`monoid_plus$
534 template <typename T>
535 struct plus_monoid: public concept {
536   typedef
537     monoid<T, op_plus, wrapped_constructor<T()> >
538     requirements;
539 };
540
541 template <typename T>
542 struct verified<monoid<T, op_plus, wrapped_constructor<T()> > >
543  : public verified<plus_monoid<T> >
544 {};
545 //\end
546 }
547
548 namespace monoiddef {
549   template <typename T>
550   struct container;
551 //\start`sum$
552 template <typename C,
553           ENABLE_IF(container<C>),
554           typename T = typename container<C>::element_type,
555           ENABLE_IF(plus_monoid<T>)>
556 T sum(C set);
557 //\end
558 }
559
560 namespace monoiddecomposed {
561   using namespace catsfoot;
562 //\start`monoidhead$
563 template <typename T, typename Op, typename Id>
564 struct monoid: public concept {
565 //\end
566 //\start`monoidreqs$
567   typedef concept_list<
568     // operations are callable with the given parameter types
569     is_callable<Op(T, T)>,
570     is_callable<Id()>,
571     // results are convertible to T
572     std::is_convertible<typename is_callable<Op(T, T)>
573                         ::result_type, T>,
574     std::is_convertible<typename is_callable<Id()>
575                         ::result_type, T>
576   > requirements;
```

```
577 //\end
578 //\start`monoidaxioms$
579   static void associativity(const Op& op, const T& a,
580                             const T& b, const T& c) {
581     axiom_assert(op(a, op(b, c)) == op(op(a, b), c));
582   }
583
584   static void identity(const Op& op, const T& a, const Id& id) {
585     axiom_assert((op(id(), a) == a) && (op(a, id()) == a));
586   }
587 //\end
588 //\start`monoidgetaxioms$
589   AXIOMS(associativity, identity);
590 };  // end of concept monoid
591 //\end
592
593 template <typename T>
594 struct verified;
595 //\start`monoidverified$
596 template <>
597 struct verified<monoid<int, op_plus, constant<int,0> > >
598  : public std::true_type
599 {};
600 //\end
601 }
602
603 namespace ringdef {
604   using namespace monoiddef;
605
606   template <typename T, typename Op, typename Minus,
607             typename Id>
608   struct group: public concept {
609     typedef concept_list<
610       monoid<T, Op, Id>
611       > requirements;
612     static void inverse(const T& a,
613                         const Id& id,
614                         const Minus& minus,
615                         const Op& op) {
616       axiom_assert(op(a, minus(a)) == id());
617       axiom_assert(op(minus(a), a) == id());
618     }
619
620     AXIOMS(inverse);
621   };
622
623   template <typename T, typename Op>
624   struct commutative: public concept {
625     typedef concept_list<
626     is_callable<Op(T, T)>,
627     std::is_convertible<typename is_callable<Op(T, T)>
628                         ::result_type, T>,
629       > requirements;
630
631     static void commutativity(const T& a, const T& b,
632                               const Op& op) {
633       axiom_assert(op(a, b) == op(b, a));
634     }
635
636     AXIOMS(commutativity);
637   };
638
```

```
639   template <typename T, typename MOp, typename AOp>
640   struct distributive {
641   typedef concept_list<
642     is_callable<MOp(T, T)>,
643     is_callable<AOp(T, T)>,
644     std::is_convertible<typename is_callable<MOp(T, T)>
645                         ::result_type, T>,
646     std::is_convertible<typename is_callable<AOp(T, T)>
647                         ::result_type, T>
648     > requirements;
649
650     static void distributivity(const T& a, const T& b, const T& c,
651                                const MOp& mop, const AOp& aop) {
652       axiom_assert(mop(a, aop(b, c)) == aop(mop(a, b), mop(a, c)));
653       axiom_assert(mop(aop(a, b), c) == aop(mop(a, c), mop(a, c)));
654     }
655
656     AXIOMS(distributivity);
657   };
658
659 //\start`ring$
660 template <typename T, typename MOp, typename AOp,
661           typename Minus, typename Zero, typename One>
662 struct ring: public concept {
663   typedef monoid<T, AOp, Zero> add_monoid;
664   typedef group<T, AOp, Minus, Zero> add_group;
665   typedef monoid<T, MOp, One> mul_monoid;
666
667   typedef concept_list<
668     mul_monoid,
669     add_group, // implies add_monoid
670     distributive<T, MOp, AOp>,
671     commutative<T, AOp>
672     > requirements;
673
674   // check that we also have add_monoid
675   class_assert_concept<add_monoid> check;
676 };
677 //\end
678
679   struct op_minus {
680     template <typename T,
681               typename Ret = decltype(-std::declval<T>())>
682     Ret operator()(T&& t) const {
683       return -std::forward<T>(t);
684     }
685   };
686 }
687
688 namespace ringdef {
689 bool test()
690 {
691   auto generator =
692     choose
693     (list_data_generator<int>
694      ({-1, 0, 1, 2, 3,
695          std::numeric_limits<int>::min(),
696          std::numeric_limits<int>::max()}),
697      default_generator());
698
699   return test_all<ring<int, op_times, op_plus, op_minus,
700                        constant<int, 0>, constant<int, 1> > >
```

```
701                     (generator);
702   }
703
704   }
705
706   namespace monoiddef {
707   bool test()
708   {
709     using catsfoot::test;
710
711     auto generator =
712       choose
713       (list_data_generator<int>
714        ({-1, 0, 1, 2, 3,
715             std::numeric_limits<int>::min(),
716             std::numeric_limits<int>::max()}),
717        default_generator());
718
719   //\start`monoiddef_test$
720     test(generator,
721          monoid<int, op_plus, constant<int, 0>>::associativity);
722   //\end
723
724     if (!test(generator,
725               monoid<int, op_plus, constant<int, 0>>::associativity,
726               "associativity"))
727       return false;
728
729     return
730   //\start`monoiddef_testall$
731     test_all<monoid<int, op_plus, constant<int, 0>>>(generator);
732   //\end
733   }
734   bool test_fail()
735   {
736     auto generator =
737       choose
738       (list_data_generator<int>
739        ({-1, 0, 1, 2, 3,
740             std::numeric_limits<int>::min(),
741             std::numeric_limits<int>::max()}),
742        default_generator());
743
744     return
745   //\start`monoiddef_testfail$
746     test_all<monoid<int, op_plus, constant<int, 1>>>(generator);
747   //\end
748   }
749
750   }
751
752   namespace hashdef {
753     using namespace catsfoot;
754   //\start`hash$
755   template <typename T, typename Hash>
756   struct hash: public concept {
757     typedef concept_list<
758       congruence<op_eq, Hash, T>
759       > requirements;
760   };
761   //\end
762     bool test() {
```

```
763       auto generator =
764         choose
765         (list_data_generator<std::string>
766          ({"aaa", "bbb", "ccc"}),
767          default_generator());
768       return test_all<hash<std::string, std::hash<std::string> > >
769         (generator);
770     }
771   }
772
773   namespace orderdef {
774     using namespace catsfoot;
775   //\start 'strict_weak_order$
776   template <typename T, typename Rel>
777   struct strict_weak_order: public concept {
778     typedef concept_list<
779       is_callable<Rel(T, T)>,
780       std::is_convertible<typename is_callable<Rel(T, T)>
781                           ::result_type, bool>,
782       > requirements;
783
784     static void irreflexivity(const T& a, const Rel& rel) {
785       axiom_assert(!rel(a, a));
786     }
787
788     static void asymmetry(const T& a, const T& b, const Rel& rel) {
789       if (a != b)
790         axiom_assert(!(rel(a,b) && rel(b,a)));
791     }
792     static void transitivity(const T& a, const T& b, const T& c,
793                              const Rel& rel) {
794       if ((rel(a, b) && rel(b, c)))
795         axiom_assert(rel(a,c));
796     }
797     static void transitivity_of_equivalence
798     (const T& a, const T&b, const T& c, const Rel& rel) {
799       if (rel(a, b))
800         axiom_assert(rel(a, c) || rel(c, b));
801     }
802
803     AXIOMS(irreflexivity,
804            asymmetry,
805            transitivity,
806            transitivity_of_equivalence)
807   };
808   //\end
809     bool test() {
810       auto generator =
811         choose
812         (list_data_generator<int>
813          ({-1, 0, 1, 2, 3,
814              std::numeric_limits<int>::min(),
815              std::numeric_limits<int>::max()}),
816          default_generator());
817       return test_all<strict_weak_order<int, op_lt> >
818         (generator);
819     }
820   }
821
822
823   namespace exception {
824     using namespace catsfoot;
```

```
825 //\start `exception$
826 template <typename T,
827           typename Fun, typename... Args,
828           ENABLE_IF(is_callable<Fun(Args...)>)>
829 bool throwing(Fun&& fun, Args&&... args) {
830   try {
831     std::forward<Fun>(fun)(std::forward<Args>(args)...);
832   } catch (T) {
833     return true;
834   }
835   return false;
836 }
837 //\end
838   struct Bidule {};
839   void f(int i) {
840     if (i == 0)
841       throw Bidule();
842   }
843
844   bool test() {
845     return throwing<Bidule>(f, 0) && !throwing<Bidule>(f, 1);
846   }
847 }
848
849 int main() {
850   bool ret = true;
851   ret = ret && equivalencens::f();
852   ret = ret && g();
853   ret = ret && h();
854   ret = ret && ringdef::test();
855   ret = ret && monoiddef::test();
856   ret = ret && !monoiddef::test_fail();
857   ret = ret && exception::test();
858   ret = ret && orderdef::test();
859   ret = ret && hashdef::test();
860   ret = ret && wrapped::test();
861   ret = ret && wrapped2::test();
862   ret = ret && testerns::dotest();
863   return !ret;
864 }
865
866 //\endignore
```

Chapter 6

Todo List

Class catsfoot::container< T, ValueType > Check the returns are iterators.

Member catsfoot::details::try_all_compare< T, std::function< Ret(Args...)> >::doit(Generator &, const std::functio
If == does not exist?

Chapter 7

Module Index

Contents

7.1 Modules

Here is a list of all modules:

Chapter 8

Namespace Index

Contents

8.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 9

Class Index

Contents

9.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 10

Class Index

Contents

10.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 11

Module Documentation

Contents

11.1 User type traits

Classes

- struct catsfoot::verified< T >

  User type traits for validating models.

11.2 Concepts

Classes

- struct catsfoot::equality< T, U >

  Check whether T is comparable to U by ==.

- struct catsfoot::printable< T, U >

  Check whether U is printable on a stream T.

- struct catsfoot::is_same< T, U >

  Check whether T and U are the same type.

- struct catsfoot::equivalence< T, Rel >

  Generic concept for equivalence relation.

- struct catsfoot::equivalence_eq< T >

  Concept for equivalence relation with operator==.

- struct catsfoot::congruence< Rel, Op, Args >

  Check whether an operation sees equality as congruence relation.

- struct catsfoot::congruence_eq< T, Args >

  Concept for congruence relation with operator==.

- struct catsfoot::container< T, ValueType >
- struct catsfoot::generator_for< T, Type >

  Whether T is a generator of Type.

11.3 Predicates

Classes

- struct catsfoot::eval< T >

  Predicate evaluating models and predicates.

- struct catsfoot::is_auto_concept< T >

  Checks if T is an auto concept model.

- struct catsfoot::is_concept< T >

  Checks if T is a concept model.

- struct catsfoot::is_callable< T(U...)>

  Tells whether T is callable with (U...)

- struct catsfoot::is_constructible< T(U...)>

  Tells whether T is constructible with {U...}.

11.4 Data generators

Classes

- struct catsfoot::default_generator

Generator that build default constructors.

- struct catsfoot::list_data_generator< T >

  Generator using lists of data provided by the user.

Functions

- template<typename... T>
  details::generator_choose< typename std::decay< T >::type...> catsfoot::choose (T &&...t)

  Combines generators.
  Picks the left-most generator capable or generating the requested data set.

- template<typename Generator >
  details::tuple_generator< typename std::decay< Generator >::type > catsfoot::tuple_gen (Generator &&g)

  Builds a tuple generator from element generators.

11.5 Type traits

Classes

- struct catsfoot::wrapped< T >

  Get the return type of the wrap function.

11.6 Macro definitions

Defines

- #define axiom_assert(expr)

  Throws an axiom failure if expr is false.

- #define ENABLE_IF(X...) typename = typename std::enable_if< ::catsfoot::eval< X >::value>::type

  Enable function definition if concept is fulfilled.

- #define ENABLE_IF_NOT(X...) typename = typename std::enable_if<!::catsfoot::eval< X >::value>::type

  Enable function definition if concept has no model.

- #define IF(X...) ::catsfoot::eval< X >::value

Tells whether a concept is fulfilled.

- #define AXIOMS(X...)

  Defines a list of axiom in a concept.

- #define DEF_MEMBER_WRAPPER(X...)

  Defines a wrapper for calling a member of a class.

- #define DEF_STATIC_MEMBER_WRAPPER(X...)

  Defines a wrapper for calling a static member of a class.

- #define DEF_FUNCTION_WRAPPER(X...)

  Defines a function wrapper for calling a function.

11.6.1 Define Documentation

11.6.1.1 #define axiom_assert(*expr*)

Value:

```
1 _catsfoot__test_axiom("<unknown>", \
2                                    "<unknown>", \
3                                    0, \
4                                    expr)
```

Throws an axiom failure if *expr* is false.

11.6.1.2 #define AXIOMS(*X...*)

Value:

```
1 static auto get_axioms() ->                                          \
2     decltype(::catsfoot::details::zip_vec_tuple                      \
3              (::catsfoot::details::split_identifiers(#X),            \
4               std::make_tuple(X))) {                                 \
5     return ::catsfoot::details::zip_vec_tuple                        \
6       (::catsfoot::details::split_identifiers(#X),                   \
7        std::make_tuple(X));                                          \
8   }
```

Defines a list of axiom in a concept.

11.6.1.3 #define DEF_FUNCTION_WRAPPER(*X...*)

Value:

```
1 struct function_##X {                                                \
2     template <typename... U,                                         \
3               typename Ret =                                         \
4               decltype(X(std::declval<U>()...))>                     \
5     Ret operator()(U&&... u...) const {                              \
```

```
6         return X(std::forward<U>(u)...);                                \
7       }                                                                 \
8     };
```

Defines a function wrapper for calling a function.

11.6.1.4 #define DEF_MEMBER_WRAPPER(X...)

Value:

```
1  struct member_##X {                                                    \
2      template <typename T, typename... U,                               \
3                typename Ret =                                           \
4                decltype(std::declval<T>()                              \
5                         .X(std::declval<U>()...))>                     \
6      Ret operator()(T&& t, U&&... u...) const {                         \
7        return std::forward<T>(t).X(std::forward<U>(u)...);              \
8      }                                                                  \
9    };
```

Defines a wrapper for calling a member of a class.

It can be useful for such a case:

```
1  DEF_MEMBER_WRAPPER(foo);
2  template <typename T, ENABLE_IF(callable<member_foo(T, int)>)>
3    [...]
4      t.foo(20);
```

11.6.1.5 #define DEF_STATIC_MEMBER_WRAPPER(X...)

Value:

```
1  template <typename T>                                                  \
2    struct static_member_##X {                                           \
3      template <typename... U,                                           \
4                typename Ret =                                           \
5                decltype(T::X(std::declval<U>()...))>                   \
6      Ret operator()(U&&... u...) const {                                \
7        return T::X(std::forward<U>(u)...);                              \
8      }                                                                  \
9    };
```

Defines a wrapper for calling a static member of a class.

It can be useful for such a case:

```
1  DEF_MEMBER_WRAPPER(foo);
2  template <typename T, ENABLE_IF(callable<static_member_foo<T>(int)>)>
3    [...]
4      T::foo(20);
```

11.6.1.6 #define ENABLE_IF(X...) typename = typename std::enable_if< ::catsfoot::eval< X >::value>::type

Enable function definition if concept is fulfilled.

It should be used in the parameter list a function template.

11.6.1.7 #define ENABLE_IF_NOT(*X...*) typename = typename std::enable_if<!::catsfoot::eval< X >::value>::type

Enable function definition if concept has no model.

It should be used in the parameter list a function template.

11.6.1.8 #define IF(*X...*) ::catsfoot::eval< X >::value

Tells whether a concept is fulfilled.

It makes a constant expression of type *bool*.

11.7 Operator wrappers

Classes

- struct catsfoot::op_eq
  Wraps operator==.
- struct catsfoot::op_neq
  Wraps operator!=.
- struct catsfoot::op_plus
  Wraps operator+.
- struct catsfoot::op_times
  Wraps operator∗.
- struct catsfoot::op_lsh
  Wraps operator<<.
- struct catsfoot::op_lt
  Wraps operator<.
- struct catsfoot::op_inc
  Wraps operator++.
- struct catsfoot::op_post_inc
  Wraps operator++(T&, int)
- struct catsfoot::op_star
  Wraps operator∗.
- struct catsfoot::wrapped_constructor< T >
  Wraps constructor T.

Chapter 12

Namespace Documentation

Contents

12.1 catsfoot Namespace Reference

Main namespace for Catsfoot.

Namespaces

- namespace details

  Implementation details for Catsfoot.

Classes

- struct axiom_failure

  Exception class for axiom failure.

- struct verified

  User type traits for validating models.

- struct concept_list

List representation of several concepts.

- struct eval

  Predicate evaluating models and predicates.

- struct class_assert_concept

  Checks that a concept is.

- struct equality

  Check whether T is comparable to U by ==.

- struct printable

  Check whether U is printable on a stream T.

- struct is_same

  Check whether T and U are the same type.

- struct equivalence

  Generic concept for equivalence relation.

- struct equivalence_eq

  Concept for equivalence relation with operator==.

- struct verified< equivalence_eq< T > >

  If == is an equivalence relation to T, then T is equivalence_eq.

- struct verified< equivalence< T, op_eq > >

  If T is equivalence_eq, then == is an equivalence relation to T.

- struct congruence

  Check whether an operation sees equality as congruence relation.

- struct congruence_eq

  Concept for congruence relation with operator==.

- struct verified< congruence_eq< T, Args...> >

  When T is a congruence for ==, then T is a congruence_eq.

- struct verified< congruence< op_eq, T, Args...> >

  When T is a congruence_eq, then T is a congruence for ==.

- struct concept

  Base marker for concepts.

- struct auto_concept

  Base marker for auto concepts.

- struct is_auto_concept

  Checks if T is an auto concept model.

- struct is_concept

  Checks if T is a concept model.

- struct product
- struct simple_product
- struct verified< product< Type, constructor_wrap< Type >, Projections...> >
- struct selector

  Constructible type used to select a type for overloaded functions.

- struct container
- struct generator_for

  Whether T is a generator of Type.

- struct default_generator

  Generator that build default constructors.

- struct list_data_generator

  Generator using lists of data provided by the user.

- struct pick_functor
- struct term_generator_builder

  Builds random term generators.

- struct undefined_return

  Return type used in case of error for is_callable.

- struct is_callable

  Default case: callable without parameters?

- struct is_callable< T(U...)>

  Tells whether T is callable with (U...)

- struct is_callable< void(U...)>

  Void is a special type that may induce extra error messages.

- struct is_constructible

  Undefined.

- struct is_constructible< T(U...)>

  Tells whether T is constructible with {U...}.

- struct is_constructible< void(U...)>

Void is a special type that may induce extra error messages.

- struct undefined_member_type

  Type used return by has_member_... in case member does not exist.

- struct build_comparer

  This is class is used to build an black box equality engine.

- struct constant

  Functor returning and integral constant.

- struct wrapped

  Get the return type of the wrap function.

- struct disamb

  Disambiguates an overloaded functions address.

- struct disamb_const

  Disambiguates an overloaded const member.

- struct constructor_wrap

  Wraps constructors.

- struct op_eq

  Wraps operator==.

- struct op_neq

  Wraps operator!=.

- struct op_plus

  Wraps operator+.

- struct op_times

  Wraps operator∗.

- struct op_lsh

  Wraps operator<<.

- struct op_lt

  Wraps operator<.

- struct op_inc

  Wraps operator++.

- struct op_post_inc

  Wraps operator++(T&, int)

- struct op_star

  Wraps operator∗.

- struct wrapped_constructor

  Wraps constructor T.

Functions

- EXTERN bool check_unverified (std::ostream &s=std::cerr)

  Tells whether some conditional axioms where never reached.

- template<typename T >
  void assert_concept (const T &)

  Assert that the concept T holds.

- template<typename Rel , typename... E, typename Index = std::integral_constant<size_t, 0>, typename = typename std::enable_if <(Index::value == sizeof...(E))>::type, typename = void>
  bool tuple_rel (Rel, const std::tuple< E...> &, const std::tuple< E...> &, Index=Index())

  Check that a relation holds for all elements on a pair of tuples.

- template<typename Rel , typename... E, typename Index = std::integral_constant<size_t, 0>, typename >
  bool tuple_rel (Rel rel, const std::tuple< E...> &a, const std::tuple< E...> &b, Index=Index())

  Check that a relation holds for all elements on a pair of tuples.

- template<typename... T>
  details::generator_choose< typename std::decay< T >::type...> choose (T &&...t)

  Combines generators.
  Picks the left-most generator capable or generating the requested data set.

- **DEF_MEMBER_WRAPPER** (begin)
- **DEF_MEMBER_WRAPPER** (end)
- **DEF_MEMBER_WRAPPER** (get)
- template<typename T , typename Generator >
  std::function< T()> **pick** (Generator &g)
- template<typename Generator >
  details::tuple_generator< typename std::decay< Generator >::type > tuple_gen (Generator &&g)

  Builds a tuple generator from element generators.

- template<typename Model , typename Generator , typename Stream = decltype(std::cerr)>
  bool test_all (Generator &g, Stream &s=std::cerr)

Tests all axioms of model Model and its sub-models.

- template<typename Generator , typename... T, typename Stream = decltype(std::cerr)>
  bool test (Generator &g, void f(T...), std::string name="<unknown>", Stream &s=std::cerr)

  This will call f with parameters from g.

- template<typename Fun , typename... Params>
  bool catch_errors (Fun f, Params &&...values)

  This will call f with the parameters and catch any axiom error.

- template<typename Op , typename Tuple >
  details::call_with_ret< Op, Tuple >::type call_with (Op &&op, Tuple &&args)

  Calls a function with arguments from a std::tuple.

- template<typename Ret , typename... Args>
  std::function< Ret(Args...)> wrap (Ret(*f)(Args...))

  Wraps a function.

- template<typename T >
  std::function< T > wrap (const std::function< T > &f)

  Forwards function.

- template<typename T >
  std::function< T > wrap (std::function< T > &&f)

  Forwards function.

- template<typename T >
  std::function< T > constructor ()

  Wraps a constructor.

- template<typename T >
  details::remove_side_effect_helper< typename std::decay< T >::type > **remove_side_effect** (T t)

Variables

- std::tuple< typename std::decay< typename is_callable< const Functions(const Parameter &)>::result_type >::type...> call_tuple (const std::tuple< Functions...> &functions, const Parameter &param)

  Calls a tuple of functions with an argument.

- std::function< Ret(T &, Args...)> wrap (Ret(T::*f)(Args...))

  Wraps a non-const member function.

12.1.1 Detailed Description

Main namespace for Catsfoot.

12.1.2 Function Documentation

12.1.2.1 template<typename T > void catsfoot::assert_concept (const T &)

Assert that the concept T holds.

This function does not do anything. It will however raise a static assertion in case T is not a valid concept.

12.1.2.2 template<typename Op , typename Tuple > details::call_with_ret<Op, Tuple>::type catsfoot::call_with (Op && *op,* Tuple && *args*)

Calls a function with arguments from a *std::tuple*.

Parameters

op	A function.
args	The tuple of arguments

Returns

the result of op(args...)

12.1.2.3 template<typename Fun , typename... Params> bool catsfoot::catch_errors (Fun *f,* Params &&... *values*)

This will call *f* with the parameters and catch any axiom error.

It is a shortcut for

```
1  try {
2    f(values...);
3  } catch (axiom_failure af) {
4    std::cerr << /*...*/;
5  }
```

12.1.2.4 template<typename T > std::function<T> catsfoot::constructor ( )

Wraps a constructor.

Template Parameters

T	Signature where return type is the class.

12.1.3 Variable Documentation

12.1.3.1 std::tuple<typename std::decay <typename is_callable <const Functions(const Parameter&)>::result_type>::type...> catsfoot::call_tuple(const std::tuple< Functions...> &functions, const Parameter &param)

Calls a tuple of functions with an argument.

Parameters

functions	The functions
param	The parameter for calling the function

Returns

the tuple of results

12.1.3.2 std::function< Ret(const T &, Args...)> catsfoot::wrap

Wraps a non-const member function.

Wraps a const member function.

We "functionalize" the method here.

12.2 catsfoot::details Namespace Reference

Implementation details for Catsfoot.

Classes

- struct eval

  Default case: predicate.

- struct eval< T, true, true, B >

  If T is a concept model.

- struct eval< T, true, false, true >

  If T is an auto-concept model.

- struct eval< concept_list< T...>, false, false, false >

  If the parameter is a list of requirements.

- struct class_assert_verified

  Check that T is a verified model.

- struct class_assert_concept

Concept checking: predicate.

- struct class_assert_concept< concept_list< F, T...>, A, B, C >

  Concept checking: recursion on a list of requirements.

- struct class_assert_concept< concept_list<>, A, B, C >

  Concept checking: empty list of requirements.

- struct class_assert_concept< T, true, B, true >

  Concept checking: T is a concept model.

- struct class_assert_concept< T, false, true, true >

  Concept checking: T is a auto-concept model.

- struct is_same
- struct is_same< T, T >
- struct has_requirements

  Check whether T has a member type `requirements`.

- struct static_binary_and

  Predicate true if all predicate parameters are true.

- struct static_binary_and< T, U, true >

  Predicate true if all predicate parameters are true.

- struct static_and< F, T...>

  Predicate true if all predicate parameters are true.

- struct static_and<>

  Predicate true if all predicate parameters are true.

- struct generator_choose
- struct generator_choose< T, Other...>
- struct position_impl

  Finds the first occurrence of a type in a list.

- struct position_impl< N, T, T, U...>

  Finds the first occurrence of a type in a list.

- struct position_impl< N, T, U, V...>

  Finds the first occurrence of a type in a list.

- struct position

  Finds the first occurrence of a type in a list.

- struct number_function_returns< T >

No function given.

- struct number_function_returns< T, std::function< T(Args...)>, Functions...>
  The first function returns a T.
- struct number_function_returns< T, std::function< const T &(Args...)>, Functions...>
  The first function returns a T.
- struct number_function_returns< T, std::function< T &(Args...)>, Functions...>
  The first function returns a T.
- struct number_function_returns< T, std::function< T &&(Args...)>, Functions...>
  The first function returns a T.
- struct number_function_returns< T, std::function< Ret(Args...)>, Functions...>
  The first function does not return a T.
- struct number_ground_terms< T >
  No function.
- struct number_ground_terms< T, std::function< T()>, Functions...>
  First function is ground term.
- struct number_ground_terms< T, std::function< const T &()>, Functions...>
  First function is ground term.
- struct number_ground_terms< T, std::function< T &()>, Functions...>
  First function is ground term.
- struct number_ground_terms< T, std::function< T &&()>, Functions...>
  First function is ground term.
- struct number_ground_terms< T, std::function< Ret(Args...)>, Functions...>
  First function is not ground term.
- struct tuple_generator_tool
- struct tuple_generator_tool< std::tuple< U...> & >
  Generates containers of tuples std::tuple<U...>
- struct tuple_generator_tool< const std::tuple< U...> & >
  Generates containers of tuples std::tuple<U...>
- struct tuple_generator_tool< std::tuple< U...> >

Generates containers of tuples std::tuple<U...>

- struct tuple_generator

  Generates tuple from generator.

- struct has_get_axiom

  Predicate testing that U has static member get_axioms()

- struct test_all

  Test axioms of a predicate (always true)

- struct test_all< T, true, false >

  Test axioms of a concept with no local axioms.

- struct test_all< T, true, true >

  Test axioms of a concept with local axioms.

- struct test_all< concept_list< T, U...>, false, B >

  Test axioms of a list of requirements.

- struct tester<>
- struct tester< T, U...>
- struct always_false

  Predicate always false but depending on type parameters.

- struct always_true

  Predicate always true but depending on type parameters.

- struct callable_real_map

  Traits to use when everything is OK.

- struct callable_bad_map

  Traits to use on error.

- struct callable_real_map< T(U...)>

  Traits defining the return type.

- struct callable_bad_map< T(U...)>

  Traits defining an undefined return type.

- struct is_constructible_work_around< false, T(U...)>

  If T is not a class, it cannot be built with several parameter.

- struct is_constructible_work_around< false, T()>

  If not a class, is it always default constructible (maybe)

- struct is_constructible_work_around< false, T(U)>

  If is explicitly convertible and not a class, then it is like constructible.

- struct is_constructible_work_around< true, T(U...)>

  If T is a class, then use the standard library.

- struct try_first

  Tag for prioritizing some overloaded functions.

- struct try_second

  Tag for prioritizing some overloaded functions.

- struct try_all_compare< T, std::function< Ret(Args...)> >
- struct compare< Generator, T >

  No left operation.

- struct compare< Generator, T, Op, Ops...>

  Compare using the first operator and recurse.

- struct compare_top

  Builds a black-box comparator.

- struct call_tuple_helper
- struct add_const_ref
- struct add_ref
- struct call_with_ret
- struct call_with_ret< Op, std::tuple< Args...> & >
- struct call_with_ret< Op, std::tuple< Args...> >
- struct call_with_ret< Op, const std::tuple< Args...> & >
- struct member_wrapper< Ret(T::*)(Args...)>

  Wraps a non-const member.

- struct member_wrapper< void(T::*)(Args...)>

  Wraps a non-const void returning member.

- struct member_wrapper< Ret(T::*)(Args...) const >

  Wraps a const member.

- struct return_of
- struct return_of< T(Args...)>
- struct remove_side_effect_helper
- struct wrapped_constructor< T(Args...), true >

  Wraps constructor T(Args...)

- struct wrapped_constructor< T(Args...), false >

  T(Args...) is not constructible.

Functions

- tmp tmp KbNL7z0sQt catsfoot src axioms axioms hh EXTERN void reached_function (std::string &&func, std::string &&file, unsigned line)

  Counts axiom as reached.

- EXTERN void register_function (std::string &&func, std::string &&file, unsigned line)

  Ensures the axiom as a counter.

- template<typename T , typename = typename T::requirements>
  std::true_type **has_requirements_helper** (try_first, T &&)
- template<typename T >
  std::false_type **has_requirements_helper** (try_second, T &&)
- template<typename T , typename... U, typename... V, typename = typename std::enable_if<sizeof...(V) == sizeof...(U)>::type>
  std::tuple< std::pair< T, U >...> zip_vec_tuple (const std::vector< T > &vec, const std::tuple< U...> &, V &&...v)

  Zip a vector of T with a tuple of U...

- template<typename T , typename... U, typename... V, typename = void, typename >
  std::tuple< std::pair< T, U >...> zip_vec_tuple (const std::vector< T > &v, const std::tuple< U...> &t, V &&...values)

  Zip a vector of T with a tuple of U...

- EXTERN std::vector< std::string > split_identifiers (const std::string &)

  Splits a comma separated list of C++ identifiers into a vector.

- template<typename U >
  std::false_type has_get_axiom_helper (try_second, U &&)

  Selected only when U has no static member get_axioms()

- template<typename U , typename = decltype(U::get_axioms())>
  std::true_type has_get_axiom_helper (try_first, U &&)

  Selected only when U has a static member get_axioms()

- template<typename Stream , typename T , ENABLE_IF(printable< Stream &, T >) >
  void print_if_printable (Stream &s, T &&t)

  Prints the value if printable.

- template<typename Stream , typename T , ENABLE_IF_NOT(printable< Stream &, T >) , typename = void>
  void print_if_printable (Stream &s, T &&)

  Prints type name of value if not printable.

- void display_values ()

  Prints a list of values.

- template<typename T , typename... U>
  void display_values (T &&t, U &&...u)

  Prints a list of values.

- template<typename T , typename... U>
  std::false_type is_callable_helper (try_second, T &&, U &&...)

  Selected when T is not callable with U...

- template<typename T , typename... U, typename = decltype(std::declval<T>()(std::declval<U>()...))>
  std::true_type is_callable_helper (try_first, T &&, U &&...)

  Selected when T is callable with U...

- template<typename Op , typename Tuple , typename... Given, typename = typename std::enable_-if<(sizeof...(Given) == call_with_ret<Op, Tuple>::size)>::type>
  call_with_ret< Op, Tuple >::type **call_with_it** (Op &&op, Tuple &&, Given &&...args)
- template<typename Op , typename Tuple , typename... OtherArgs, typename = typename std::enable_-if<(sizeof...(OtherArgs) != call_with_ret<Op, Tuple>::size)>::type, typename = void>
  call_with_ret< Op, Tuple >::type **call_with_it** (Op &&op, Tuple &&args, Other-Args &&...otherargs)
- template<typename T >
  std::string type_to_string ()

  Gives a string of the type T.

- template<typename T , typename = typename std::enable_if<!std::is_reference<T>::value>::type>
  T **copy_if_non_const** (T &&t)
- template<typename T >
  T **copy_if_non_const** (T &t)
- template<typename T >
  const T & **copy_if_non_const** (const T &t)

12.2.1 Detailed Description

Implementation details for Catsfoot.

12.2.2 Function Documentation

12.2.2.1 template<typename T > std::string catsfoot::details::type_to_string ( )

Gives a string of the type *T*.

Template Parameters

T	a type

Returns

A possibly demangled string representation of *T*

Chapter 13

Class Documentation

Contents

13.1 catsfoot::details::add_const_ref< T > Struct Template Reference

Public Types

- typedef const T & **type**

template<typename T> struct catsfoot::details::add_const_ref< T >

The documentation for this struct was generated from the following file:

- utils/call_with.hh

13.2 catsfoot::details::add_ref< T > Struct Template Reference

Public Types

- typedef T & **type**

template<typename T> struct catsfoot::details::add_ref< T >

The documentation for this struct was generated from the following file:

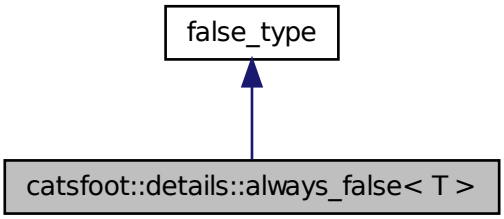
- utils/call_with.hh

13.3 catsfoot::details::always_false< T > Struct Template Reference

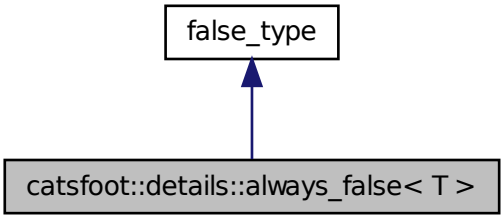
Predicate always false but depending on type parameters.

`#include <always_false.hh>`

Inheritance diagram for catsfoot::details::always_false< T >:



Collaboration diagram for catsfoot::details::always_false< T >:



13.3.1 Detailed Description

template<typename... T> struct catsfoot::details::always_false< T >

Predicate always false but depending on type parameters.

The documentation for this struct was generated from the following file:

- type_traits/always_false.hh

13.4 catsfoot::details::always_true< T > Struct Template Reference

Predicate always true but depending on type parameters.

`#include <always_false.hh>`

13.4.1 Detailed Description

template<typename... T> struct catsfoot::details::always_true< T >

Predicate always true but depending on type parameters.

The documentation for this struct was generated from the following file:

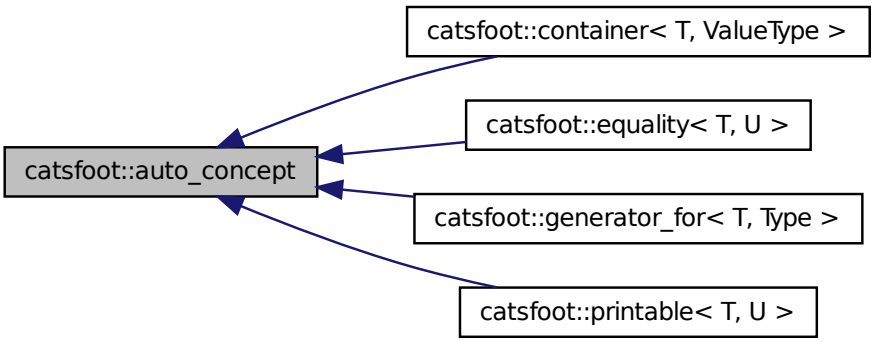
- type_traits/always_false.hh

13.5 catsfoot::auto_concept Struct Reference

Base marker for auto concepts.

`#include <is_concept.hh>`

Inheritance diagram for catsfoot::auto_concept:



13.5.1 Detailed Description

Base marker for auto concepts.

The documentation for this struct was generated from the following file:

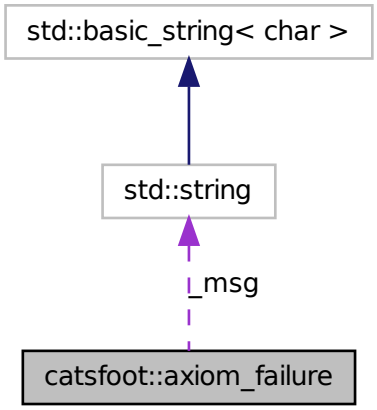
- concept/is_concept.hh

13.6 catsfoot::axiom_failure Struct Reference

Exception class for axiom failure.

`#include <axioms.hh>`

Collaboration diagram for catsfoot::axiom_failure:



Public Member Functions

- const std::string & msg () const

  Returns the message explaining the axiom failure.

- **axiom_failure** (const std::string &_msg)
- **axiom_failure** (std::string &&_msg)
- **axiom_failure** (std::string &&func, std::string &&file, unsigned line, std::string &&expr)
- axiom_failure (const axiom_failure &)

  Copy constructor.

- axiom_failure (axiom_failure &&other)

  Move constructor.

- axiom_failure & operator= (axiom_failure &&other)

  Swap.

- axiom_failure & operator= (const axiom_failure &other)

Assignment.

13.6.1 Detailed Description

Exception class for axiom failure.

The documentation for this struct was generated from the following file:

- axioms/axioms.hh

13.7 catsfoot::build_comparer< T > Struct Template Reference

This is class is used to build an black box equality engine.

`#include <black_box_equal.hh>`

Public Member Functions

- template<typename Generator , typename... Functions, typename G = typename std::decay<Generator>::type> details::compare_top< G, T, typename wrapped< Functions >::type...> **operator()** (Generator g, Functions &&...functions...) const

13.7.1 Detailed Description

template<typename T> struct catsfoot::build_comparer< T >

This is class is used to build an black box equality engine.

Template Parameters

T	is the type to provide equality

The documentation for this struct was generated from the following file:

- utils/black_box_equal.hh

13.8 catsfoot::details::call_tuple_helper< Parameter, Functions > Struct Template Reference

Public Types

- typedef std::tuple< typename std::decay< typename is_callable< const Functions(const Parameter &)>::result_type >::type...> **return_type**

Static Public Member Functions

- static return_type **call** (const std::tuple< Functions...> &, const Parameter &, typename is_callable< const Functions(const Parameter &)>::result_type...args)

- template<typename... Args>
  static return_type **call** (const std::tuple< Functions...> &functions, const Parameter &param, Args...args)

template<typename Parameter, typename... Functions> struct catsfoot::details::call_tuple_helper< Parameter, Functions >

The documentation for this struct was generated from the following file:

- utils/call_tuple.hh

13.9 catsfoot::details::call_with_ret< Op, Tuple > Struct Template Reference

template<typename Op, typename Tuple> struct catsfoot::details::call_with_ret< Op, Tuple >

The documentation for this struct was generated from the following file:

- utils/call_with.hh

13.10 catsfoot::details::call_with_ret< Op, const std::tuple< Args...> & > Struct Template Reference

Public Member Functions

- typedef **decltype** (std::declval< Op >()(std::declval< typename add_const_ref< Args >::type >()...)) type

Static Public Attributes

- static const size_t **size** = sizeof...(Args)

template<typename Op, typename... Args> struct catsfoot::details::call_with_ret< Op, const std::tuple< Args...> & >

The documentation for this struct was generated from the following file:

- utils/call_with.hh

13.11 catsfoot::details::call_with_ret< Op, std::tuple< Args...> & > Struct Template Reference

Public Member Functions

- typedef **decltype** (std::declval< Op >()(std::declval< typename add_ref< Args >::type >()...)) type

Static Public Attributes

- static const size_t **size** = sizeof...(Args)

template<typename Op, typename... Args> struct catsfoot::details::call_with_ret< Op, std::tuple< Args...> & >

The documentation for this struct was generated from the following file:

- utils/call_with.hh

13.12 catsfoot::details::call_with_ret< Op, std::tuple< Args...> > Struct Template Reference

Public Member Functions

- typedef **decltype** (std::declval< Op >()(std::declval< typename add_ref< Args >::type >()...)) type

Static Public Attributes

- static const size_t **size** = sizeof...(Args)

template<typename Op, typename... Args> struct catsfoot::details::call_with_ret< Op, std::tuple< Args...> >

The documentation for this struct was generated from the following file:

- utils/call_with.hh

13.13 catsfoot::details::callable_bad_map< T > Struct Template Reference

Traits to use on error.

`#include <is_callable.hh>`

13.13.1 Detailed Description

template<typename T> struct catsfoot::details::callable_bad_map< T >

Traits to use on error.

The documentation for this struct was generated from the following file:

- type_traits/is_callable.hh

13.14 catsfoot::details::callable_bad_map< T(U...)> Struct Template Reference

Traits defining an undefined return type.

`#include <is_callable.hh>`

Public Types

- typedef undefined_return< T(U...)> **result_type**

13.14.1 Detailed Description

template<typename T, typename... U> struct catsfoot::details::callable_bad_map< T(U...)>

Traits defining an undefined return type.

The documentation for this struct was generated from the following file:

- type_traits/is_callable.hh

13.15 catsfoot::details::callable_real_map< T > Struct Template Reference

Traits to use when everything is OK.

`#include <is_callable.hh>`

13.15.1 Detailed Description

template<typename T> struct catsfoot::details::callable_real_map< T >

Traits to use when everything is OK.

The documentation for this struct was generated from the following file:

- type_traits/is_callable.hh

13.16 catsfoot::details::callable_real_map< T(U...)> Struct Template Reference

Traits defining the return type.

`#include <is_callable.hh>`

Public Member Functions

- typedef **decltype** (std::declval< T >()(std::declval< U >()...)) result_type

13.16.1 Detailed Description

template<typename T, typename... U> struct catsfoot::details::callable_real_map< T(U...)>

Traits defining the return type.

The documentation for this struct was generated from the following file:

- type_traits/is_callable.hh

13.17 catsfoot::details::class_assert_concept< T, bool, bool, bool > Struct Template Reference

Concept checking: predicate.

`#include <concept_tools.hh>`

Public Member Functions

- **static_assert** (T::value,"Missing requirement")

13.17.1 Detailed Description

template<typename T, bool = is_concept<T>::value, bool = is_auto_concept<T>::value, bool = has_requirements<T>::value> struct catsfoot::details::class_assert_concept< T, bool, bool, bool >

Concept checking: predicate.

The documentation for this struct was generated from the following file:

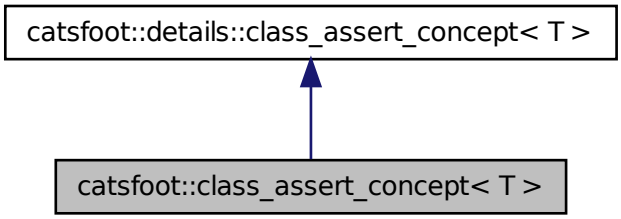
- concept/concept_tools.hh

13.18 catsfoot::class_assert_concept< T > Struct Template Reference

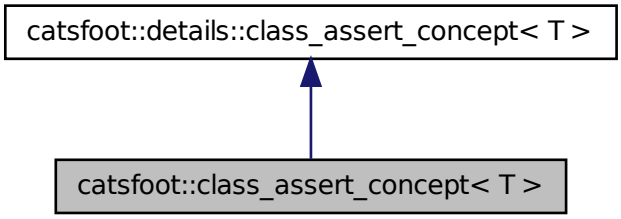
Checks that a concept is.

`#include <concept_tools.hh>`

Inheritance diagram for catsfoot::class_assert_concept< T >:



Collaboration diagram for catsfoot::class_assert_concept< T >:



13.18.1 Detailed Description

template<typename T> struct catsfoot::class_assert_concept< T >

Checks that a concept is. This type will raise a static assertion on when instantiating its concrete type if T is not a valid concept. It is intended to be used in a class template. For example:

```
1  template <typename T>
2  struct foo {
3    //...
4    class_assert_verified<some_concept<T> > check;
5  };
```

The documentation for this struct was generated from the following file:

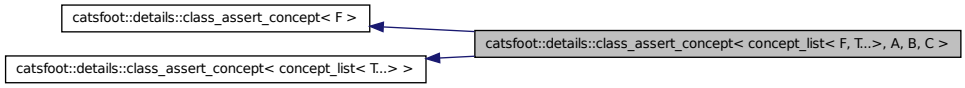
- concept/concept_tools.hh

13.19 catsfoot::details::class_assert_concept< concept_list< F, T...>, A, B, C > Struct Template Reference

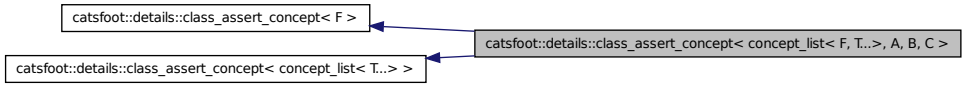
Concept checking: recursion on a list of requirements.

`#include <concept_tools.hh>`

Inheritance diagram for catsfoot::details::class_assert_concept< concept_list< F, T...>, A, B, C >:



Collaboration diagram for catsfoot::details::class_assert_concept< concept_list< F, T...>, A, B, C >:



13.19.1 Detailed Description

template<typename F, typename... T, bool A, bool B, bool C> struct catsfoot::details::class_assert_concept< concept_list< F, T...>, A, B, C >

Concept checking: recursion on a list of requirements.

The documentation for this struct was generated from the following file:

- concept/concept_tools.hh

13.20 catsfoot::details::class_assert_concept< concept_list<>, A, B, C > Struct Template Reference

Concept checking: empty list of requirements.

`#include <concept_tools.hh>`

13.20.1 Detailed Description

template<bool A, bool B, bool C> struct catsfoot::details::class_assert_concept< concept_list<>, A, B, C >

Concept checking: empty list of requirements.

The documentation for this struct was generated from the following file:

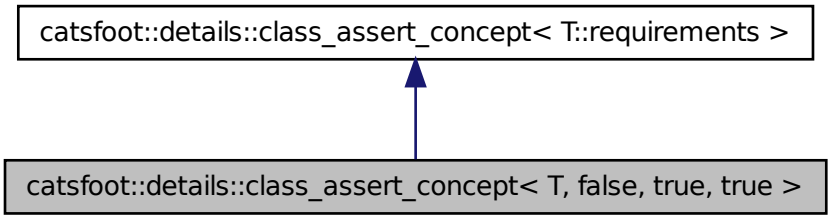
- concept/concept_tools.hh

13.21 catsfoot::details::class_assert_concept< T, false, true, true > Struct Template Reference

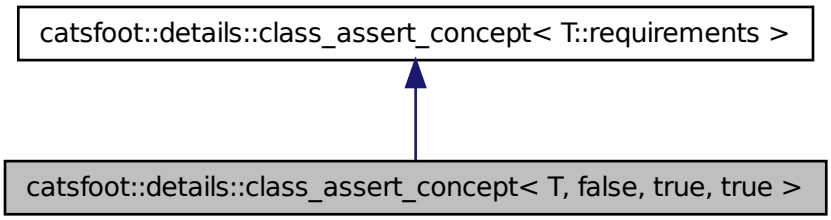
Concept checking: T is a auto-concept model.

`#include <concept_tools.hh>`

Inheritance diagram for catsfoot::details::class_assert_concept< T, false, true, true >:



Collaboration diagram for catsfoot::details::class_assert_concept< T, false, true, true >:



13.21.1 Detailed Description

template<typename T> struct catsfoot::details::class_assert_concept< T, false, true, true >

Concept checking: T is a auto-concept model.

The documentation for this struct was generated from the following file:

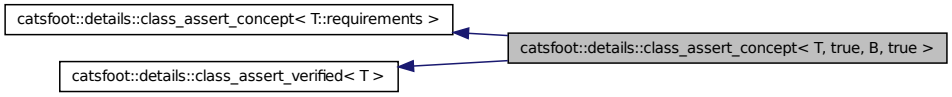
- concept/concept_tools.hh

13.22 catsfoot::details::class_assert_concept< T, true, B, true > Struct Template Reference

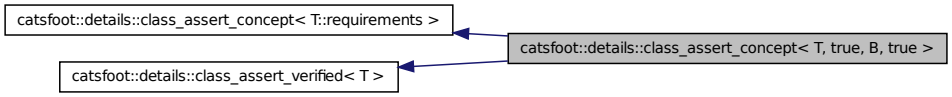
Concept checking: T is a concept model.

`#include <concept_tools.hh>`

Inheritance diagram for catsfoot::details::class_assert_concept< T, true, B, true >:



Collaboration diagram for catsfoot::details::class_assert_concept< T, true, B, true >:



13.22.1 Detailed Description

template<typename T, bool B> struct catsfoot::details::class_assert_concept< T, true, B, true >

Concept checking: T is a concept model.

The documentation for this struct was generated from the following file:

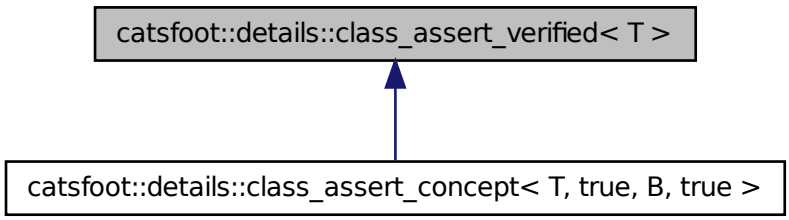
- concept/concept_tools.hh

13.23 catsfoot::details::class_assert_verified< T > Struct Template Reference

Check that T is a verified model.

`#include <concept_tools.hh>`

Inheritance diagram for catsfoot::details::class_assert_verified< T >:



Public Member Functions

- **static_assert** (verified< T >::value,"Concept was not verified")

13.23.1 Detailed Description

template<typename T> struct catsfoot::details::class_assert_verified< T >

Check that T is a verified model.

The documentation for this struct was generated from the following file:

- concept/concept_tools.hh

13.24 catsfoot::details::compare< Generator, T > Struct Template Reference

No left operation.

`#include <black_box_equal.hh>`

Public Member Functions

- bool **operator()** (Generator &, const T &, const T &) const

13.24.1 Detailed Description

template<typename Generator, typename T> struct catsfoot::details::compare< Generator, T >

No left operation.

The documentation for this struct was generated from the following file:

- utils/black_box_equal.hh

13.25 catsfoot::details::compare< Generator, T, Op, Ops...> Struct Template Reference

Compare using the first operator and recurse.

`#include <black_box_equal.hh>`

Public Member Functions

- **compare** (Op &&op, Ops &&...ops...)
- bool **operator()** (Generator &g, const T &a, const T &b) const

13.25.1 Detailed Description

template<typename Generator, typename T, typename Op, typename... Ops> struct catsfoot::details::compare< Generator, T, Op, Ops...>

Compare using the first operator and recurse.

The documentation for this struct was generated from the following file:

- utils/black_box_equal.hh

13.26 catsfoot::details::compare_top< Generator, T, Ops > Struct Template Reference

Builds a black-box comparator.

`#include <black_box_equal.hh>`

Public Member Functions

- **compare_top** (const Generator &g, Ops &&...ops...)
- **compare_top** (Generator &&g, Ops &&...ops...)
- bool **operator()** (const T &a, const T &b) const

13.26.1 Detailed Description

template<typename Generator, typename T, typename... Ops> struct catsfoot::details::compare_top< Generator, T, Ops >

Builds a black-box comparator.

The documentation for this struct was generated from the following file:

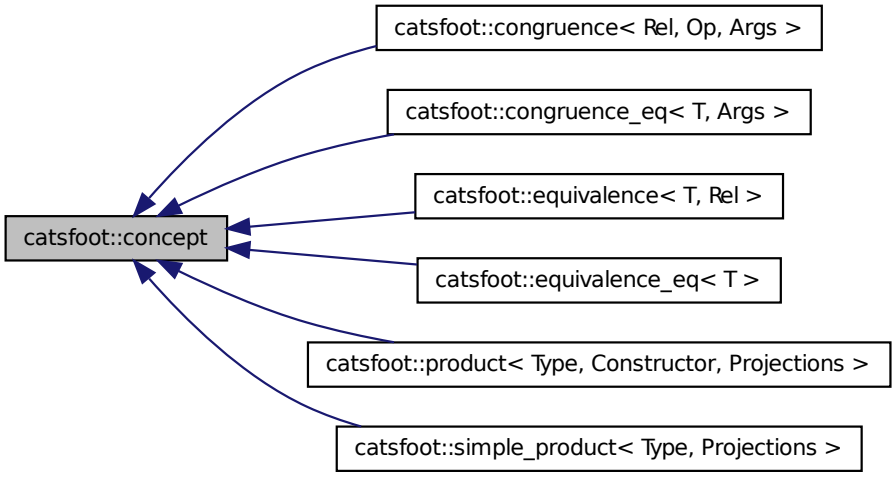
- utils/black_box_equal.hh

13.27 catsfoot::concept Struct Reference

Base marker for concepts.

`#include <is_concept.hh>`

Inheritance diagram for catsfoot::concept:



13.27.1 Detailed Description

Base marker for concepts.

The documentation for this struct was generated from the following file:

- concept/is_concept.hh

13.28 catsfoot::concept_list< T > Struct Template Reference

List representation of several concepts.

`#include <concept_tools.hh>`

13.28.1 Detailed Description

template<typename... T> struct catsfoot::concept_list< T >

List representation of several concepts.

The documentation for this struct was generated from the following file:

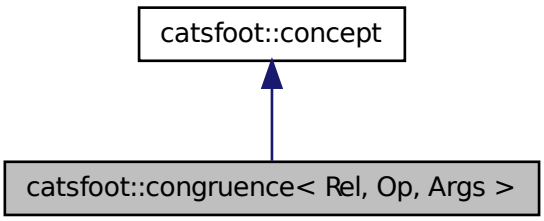
- concept/concept_tools.hh

13.29 catsfoot::congruence< Rel, Op, Args > Struct Template Reference

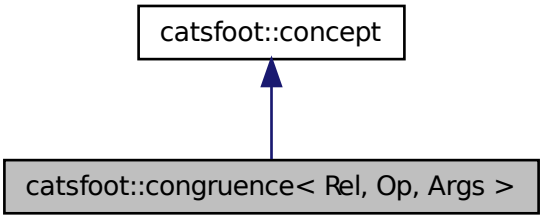
Check whether an operation sees equality as congruence relation.

`#include <congruence.hh>`

Inheritance diagram for catsfoot::congruence< Rel, Op, Args >:



Collaboration diagram for catsfoot::congruence< Rel, Op, Args >:



Public Types

- typedef concept_list< equivalence< Args, Rel >..., is_callable< Op(Args...)>, equivalence< typename is_callable< Op(Args...)>::result_type, Rel > > **requirements**

Public Member Functions

- **AXIOMS** (congruence_axiom)

Static Public Member Functions

- static void **congruence_axiom** (const std::tuple< Args...> &args1, const std::tuple< Args...> &args2, const Op &op, const Rel &rel)

13.29.1 Detailed Description

template<typename Rel, typename Op, typename... Args> struct catsfoot::congruence< Rel, Op, Args >

Check whether an operation sees equality as congruence relation.

The documentation for this struct was generated from the following file:

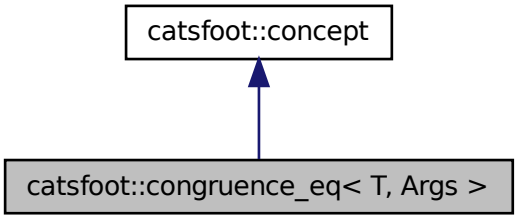
- concept/congruence.hh

13.30 catsfoot::congruence_eq< T, Args > Struct Template Reference

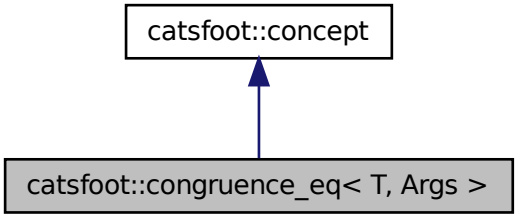
Concept for congruence relation with operator==.

`#include <congruence.hh>`

Inheritance diagram for catsfoot::congruence_eq< T, Args >:



Collaboration diagram for catsfoot::congruence_eq< T, Args >:



Public Types

- typedef concept_list< congruence< op_eq, T, Args...> > **requirements**

13.30.1 Detailed Description

template<typename T, typename... Args> struct catsfoot::congruence_eq< T, Args >

Concept for congruence relation with operator==.

The documentation for this struct was generated from the following file:

- concept/congruence.hh

13.31 catsfoot::constant< T, Value > Struct Template Reference

Functor returning and integral constant.

`#include <constant.hh>`

Public Member Functions

- T **operator()** () const

13.31.1 Detailed Description

template<typename T, T Value> struct catsfoot::constant< T, Value >

Functor returning and integral constant.

The documentation for this struct was generated from the following file:

- wrappers/constant.hh

13.32 catsfoot::constructor_wrap< T > Struct Template Reference

Wraps constructors.

`#include <function_wrappers.hh>`

Public Member Functions

- template<typename... Args>
  T **operator()** (Args &&...args) const

13.32.1 Detailed Description

template<typename T> struct catsfoot::constructor_wrap< T >

Wraps constructors.

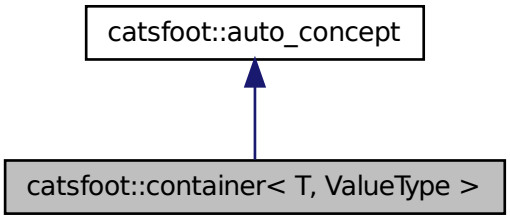
The documentation for this struct was generated from the following file:

- wrappers/function_wrappers.hh

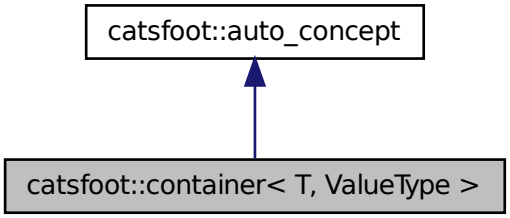
13.33 catsfoot::container< T, ValueType > Struct Template Reference

`#include <dataset.hh>`

Inheritance diagram for catsfoot::container< T, ValueType >:



Collaboration diagram for catsfoot::container< T, ValueType >:



Public Types

- typedef concept_list< is_callable< member_begin(T)>, is_callable< member_-end(T)> > **requirements**

13.33.1 Detailed Description

template<typename T, typename ValueType> struct catsfoot::container< T, ValueType >

Todo

Check the returns are iterators.

The documentation for this struct was generated from the following file:

- dataset/dataset.hh

13.34 catsfoot::default_generator Struct Reference

Generator that build default constructors.

`#include <dataset.hh>`

Public Member Functions

- template<typename Return , typename NonRefReturn = typename std::decay<Return>::type>
  std::list< NonRefReturn > **get** (selector< Return >) const

13.34.1 Detailed Description

Generator that build default constructors.

The documentation for this struct was generated from the following file:

- dataset/dataset.hh

13.35 catsfoot::disamb< Args > Struct Template Reference

Disambiguates an overloaded functions address.

`#include <function_wrappers.hh>`

Classes

- struct **fun**

Public Member Functions

- template<typename Ret , typename T >
  fun< Ret, T >::type **operator()** (Ret(T::*f)(Args...)) const
- template<typename Ret , typename T >
  fun< Ret, T >::fun_type **operator()** (Ret(*f)(Args...)) const

13.35.1 Detailed Description

template<typename... Args> struct catsfoot::disamb< Args >

Disambiguates an overloaded functions address.

Template Parameters

Args	The types of the arguments

The documentation for this struct was generated from the following file:

- wrappers/function_wrappers.hh

13.36 catsfoot::disamb_const< Args > Struct Template Reference

Disambiguates an overloaded const member.

`#include <function_wrappers.hh>`

Classes

- struct **fun**

Public Member Functions

- template<typename Ret , typename T >
  fun< Ret, T >::const_type **operator()** (Ret(T::*f)(Args...) const) const

13.36.1 Detailed Description

template<typename... Args> struct catsfoot::disamb_const< Args >

Disambiguates an overloaded const member.

Template Parameters

Args	The types of the arguments

The documentation for this struct was generated from the following file:

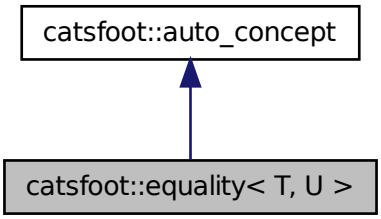
- wrappers/function_wrappers.hh

13.37 catsfoot::equality< T, U > Struct Template Reference

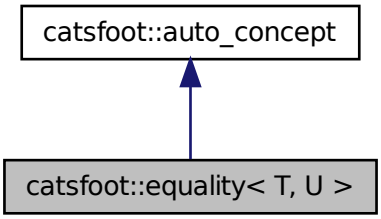
Check whether *T* is comparable to *U* by ==.

`#include <concepts.hh>`

Inheritance diagram for catsfoot::equality< T, U >:



Collaboration diagram for catsfoot::equality< T, U >:



Public Types

- typedef concept_list< is_callable< op_eq(T, U)>, std::is_convertible< typename is_callable< op_eq(T, U)>::result_type, bool > > **requirements**

13.37.1 Detailed Description

template<typename T, typename U = T> struct catsfoot::equality< T, U >

Check whether *T* is comparable to *U* by ==.

The documentation for this struct was generated from the following file:

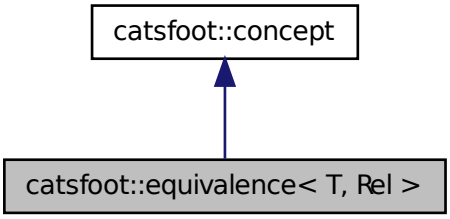
- concept/concepts.hh

13.38 catsfoot::equivalence< T, Rel > Struct Template Reference

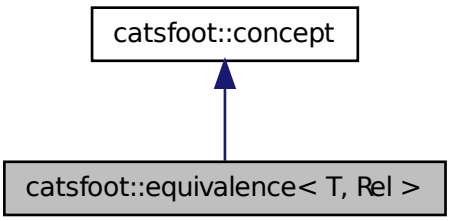
Generic concept for equivalence relation.

`#include <congruence.hh>`

Inheritance diagram for catsfoot::equivalence< T, Rel >:



Collaboration diagram for catsfoot::equivalence< T, Rel >:



Public Types

- typedef concept_list< is_callable< Rel(T, T)>, std::is_convertible< typename is_callable< Rel(T, T)>::result_type, bool > > **requirements**

Public Member Functions

- **AXIOMS** (reflexivity, symmetry, transitivity)

Static Public Member Functions

- static void **reflexivity** (const Rel &rel, const T &a)
- static void **symmetry** (const Rel &rel, const T &a, const T &b)
- static void **transitivity** (const Rel &rel, const T &a, const T &b, const T &c)

13.38.1 Detailed Description

template<typename T, typename Rel> struct catsfoot::equivalence< T, Rel >

Generic concept for equivalence relation.

The documentation for this struct was generated from the following file:

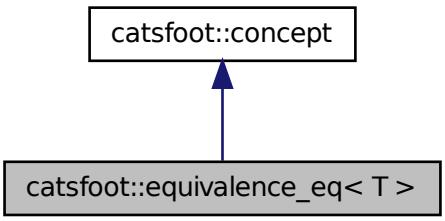
- concept/congruence.hh

13.39 catsfoot::equivalence_eq< T > Struct Template Reference

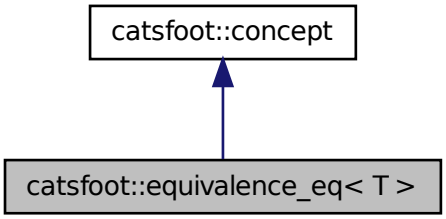
Concept for equivalence relation with operator==.

`#include <congruence.hh>`

Inheritance diagram for catsfoot::equivalence_eq< T >:



Collaboration diagram for catsfoot::equivalence_eq< T >:



Public Types

- typedef concept_list< equality< T >, equivalence< T, op_eq > > **requirements**

13.39.1 Detailed Description

template<typename T> struct catsfoot::equivalence_eq< T >

Concept for equivalence relation with operator==.

The documentation for this struct was generated from the following file:

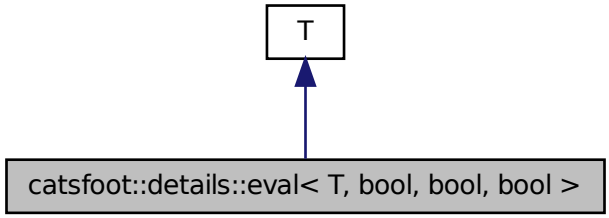
- concept/congruence.hh

13.40 catsfoot::details::eval< T, bool, bool, bool > Struct Template Reference

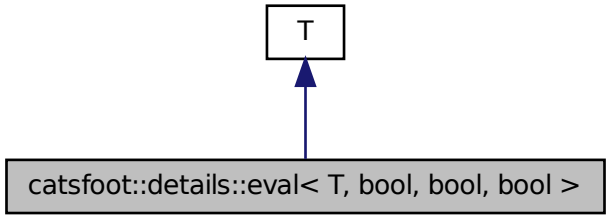
Default case: predicate.

`#include <concept_tools.hh>`

Inheritance diagram for catsfoot::details::eval< T, bool, bool, bool >:



Collaboration diagram for catsfoot::details::eval< T, bool, bool, bool >:



13.40.1 Detailed Description

template<typename T, bool = has_requirements<T>::value, bool = is_concept<T>::value, bool = is_auto_concept<T>::value> struct catsfoot::details::eval< T, bool, bool, bool >

Default case: predicate.

The documentation for this struct was generated from the following file:

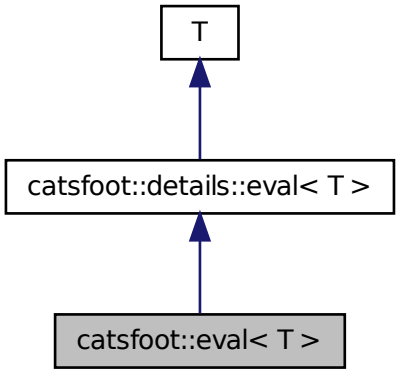
- concept/concept_tools.hh

13.41 catsfoot::eval< T > Struct Template Reference

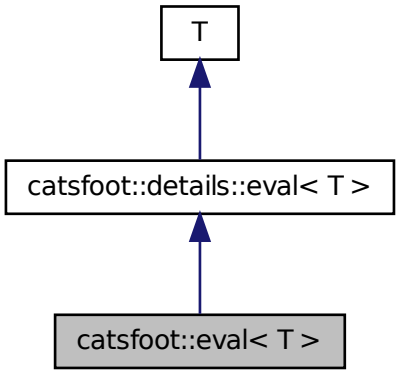
Predicate evaluating models and predicates.

`#include <concept_tools.hh>`

Inheritance diagram for catsfoot::eval< T >:



Collaboration diagram for catsfoot::eval< T >:



13.41.1 Detailed Description

template<typename T> struct catsfoot::eval< T >

Predicate evaluating models and predicates.

The documentation for this struct was generated from the following file:

- concept/concept_tools.hh

13.42 catsfoot::details::eval< concept_list< T...>, false, false, false > Struct Template Reference

If the parameter is a list of requirements.

`#include <concept_tools.hh>`

13.42.1 Detailed Description

template<typename... T> struct catsfoot::details::eval< concept_list< T...>, false, false, false >

If the parameter is a list of requirements.

The documentation for this struct was generated from the following file:

- concept/concept_tools.hh

13.43 catsfoot::details::eval< T, true, false, true > Struct Template Reference

If T is an auto-concept model.

`#include <concept_tools.hh>`

13.43.1 Detailed Description

template<typename T> struct catsfoot::details::eval< T, true, false, true >

If T is an auto-concept model.

The documentation for this struct was generated from the following file:

- concept/concept_tools.hh

13.44 catsfoot::details::eval< T, true, true, B > Struct Template Reference

If T is a concept model.

`#include <concept_tools.hh>`

13.44.1 Detailed Description

template<typename T, bool B> struct catsfoot::details::eval< T, true, true, B >

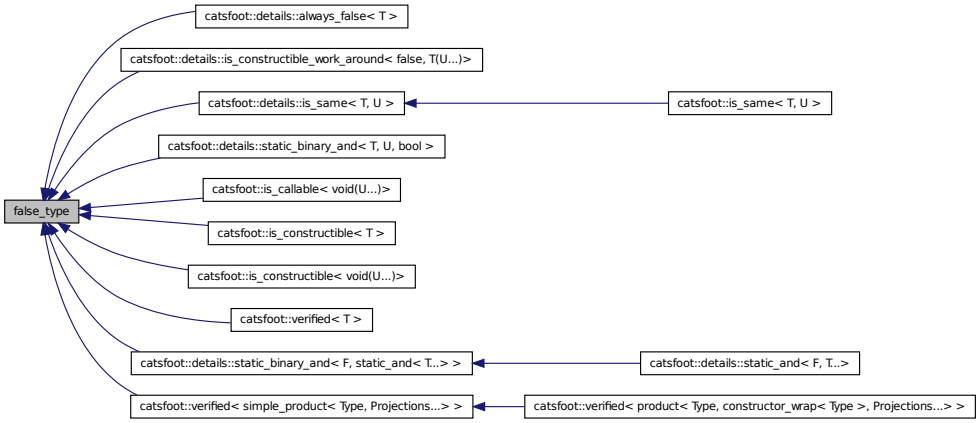
If T is a concept model.

The documentation for this struct was generated from the following file:

- concept/concept_tools.hh

13.45 false_type Class Reference

Inheritance diagram for false_type:

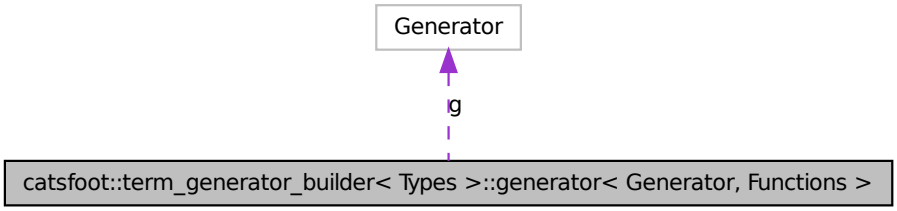


The documentation for this class was generated from the following file:

- concept/static_and.hh

13.46 catsfoot::term_generator_builder< Types >::generator< Generator, Functions > Struct Template Reference

Collaboration diagram for catsfoot::term_generator_builder< Types >::generator< Generator, Functions >:



Classes

- struct **random_container< Return, false >**

Public Member Functions

- template<typename... F>
  generator (size_t size, Generator &g, F &&...functions)

template<typename... Types>template<typename Generator, typename... Functions> struct catsfoot::term_generator_builder< Types >::generator< Generator, Functions >

The documentation for this struct was generated from the following file:

- dataset/random_term_generator.hh

13.47 catsfoot::details::generator_choose<> Struct Template Reference

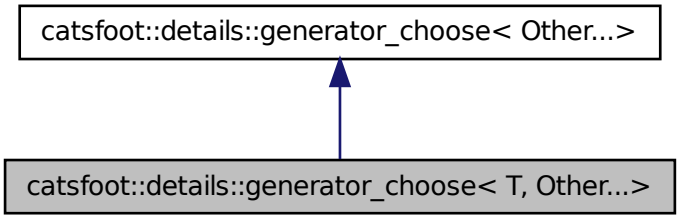
template<typename...> struct catsfoot::details::generator_choose<>

The documentation for this struct was generated from the following file:

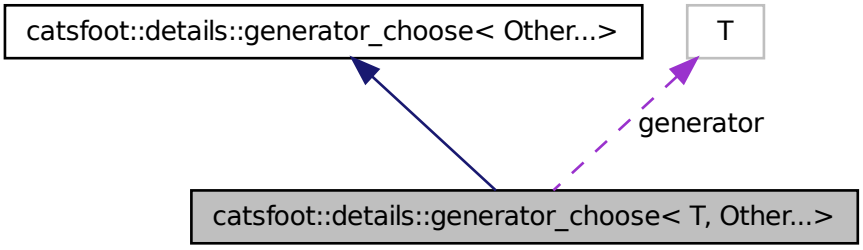
- dataset/choose.hh

13.48 catsfoot::details::generator_choose< T, Other...> Struct Template Reference

Inheritance diagram for catsfoot::details::generator_choose< T, Other...>:



Collaboration diagram for catsfoot::details::generator_choose< T, Other...>:



Public Member Functions

- **generator_choose** (T t, Other...other)
- template<typename Return , ENABLE_IF(generator_for< T, Return >) , typename Ret = typename is_callable<member_get(T&, selector<Return>)>::result_type>
  Ret **get** (selector< Return > s)
- template<typename Return , ENABLE_IF_NOT(generator_for< T, Return >) , ENABLE_IF(generator_for< super, Return >) , typename Ret = typename is_callable<member_get(super&, selector<Return>)>::result_-

type>
Ret **get** (selector< Return > s...)

template<typename T, typename... Other> struct catsfoot::details::generator_choose< T, Other...>

The documentation for this struct was generated from the following file:

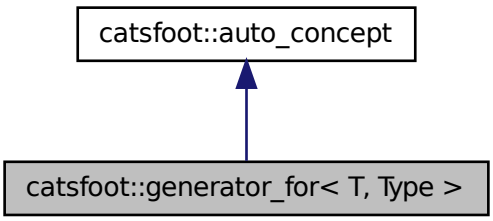
- dataset/choose.hh

13.49 catsfoot::generator_for< T, Type > Struct Template Reference

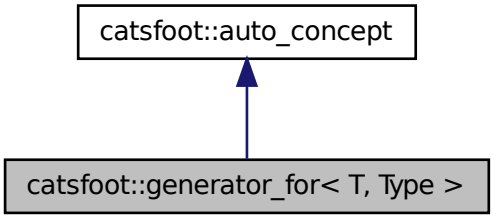
Whether T is a generator of Type.

`#include <dataset.hh>`

Inheritance diagram for catsfoot::generator_for< T, Type >:



Collaboration diagram for catsfoot::generator_for< T, Type >:



Public Types

- typedef concept_list< is_callable< member_get(T, selector< Type >)>, container< typename is_callable< member_get(T, selector< Type >)>::result_type, Type > > **requirements**

13.49.1 Detailed Description

template<typename T, typename Type> struct catsfoot::generator_for< T, Type >

Whether T is a generator of Type.

The documentation for this struct was generated from the following file:

- dataset/dataset.hh

13.50 catsfoot::details::has_get_axiom< U > Struct Template Reference

Predicate testing that U has static member get_axioms()

`#include <test_all_driver.hh>`

13.50.1 Detailed Description

template<typename U> struct catsfoot::details::has_get_axiom< U >

Predicate testing that U has static member get_axioms()

The documentation for this struct was generated from the following file:

- drivers/test_all_driver.hh

13.51 catsfoot::details::has_requirements< T > Struct Template Reference

Check whether T has a member type `requirements`.

`#include <has_requirements.hh>`

13.51.1 Detailed Description

template<typename T> struct catsfoot::details::has_requirements< T >

Check whether T has a member type `requirements`.

The documentation for this struct was generated from the following file:

- concept/has_requirements.hh

13.52 catsfoot::is_auto_concept< T > Struct Template Reference

Checks if T is an auto concept model.

`#include <is_concept.hh>`

13.52.1 Detailed Description

template<typename T> struct catsfoot::is_auto_concept< T >

Checks if T is an auto concept model.

The documentation for this struct was generated from the following file:

- concept/is_concept.hh

13.53 catsfoot::is_callable< T > Struct Template Reference

Default case: callable without parameters?

`#include <is_callable.hh>`

13.53.1 Detailed Description

template<typename T> struct catsfoot::is_callable< T >

Default case: callable without parameters?

The documentation for this struct was generated from the following file:

- type_traits/is_callable.hh

13.54 catsfoot::is_callable< T(U...)> Struct Template Reference

Tells whether T is callable with (U...)

`#include <is_callable.hh>`

Public Types

- typedef std::conditional< super::value, details::callable_real_map< T(U...)>, details::callable_bad_map< T(U...)> >::type::result_type **result_type**

Public Member Functions

- typedef **decltype** (details::is_callable_helper(details::try_first(), std::declval< T >(), std::declval< U >()...)) super

13.54.1 Detailed Description

template<typename T, typename... U> struct catsfoot::is_callable< T(U...)>

Tells whether T is callable with (U...)

The documentation for this struct was generated from the following file:

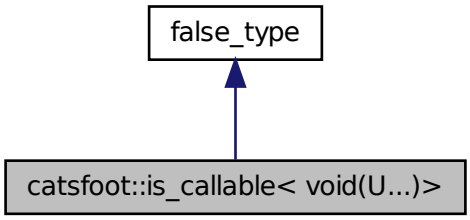
- type_traits/is_callable.hh

13.55 catsfoot::is_callable< void(U...)> Struct Template Reference

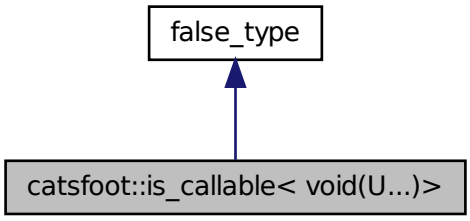
Void is a special type that may induce extra error messages.

`#include <is_callable.hh>`

Inheritance diagram for catsfoot::is_callable< void(U...)>:



Collaboration diagram for catsfoot::is_callable< void(U...)>:



Public Types

- typedef undefined_return< void(U...)> **result_type**

13.55.1 Detailed Description

template<typename... U> struct catsfoot::is_callable< void(U...)>

Void is a special type that may induce extra error messages.

The documentation for this struct was generated from the following file:

- type_traits/is_callable.hh

13.56 catsfoot::is_concept< T > Struct Template Reference

Checks if T is a concept model.

`#include <is_concept.hh>`

13.56.1 Detailed Description

template<typename T> struct catsfoot::is_concept< T >

Checks if T is a concept model.

The documentation for this struct was generated from the following file:

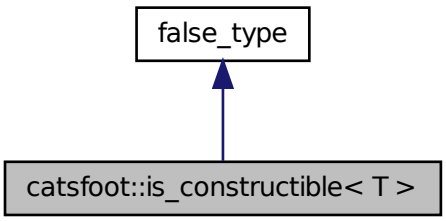
- concept/is_concept.hh

13.57 catsfoot::is_constructible< T > Struct Template Reference

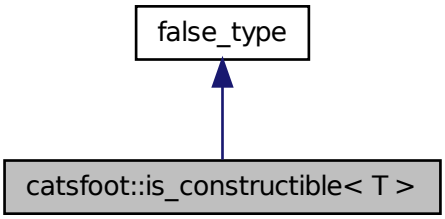
Undefined.

`#include <is_constructible.hh>`

Inheritance diagram for catsfoot::is_constructible< T >:



Collaboration diagram for catsfoot::is_constructible< T >:



Public Member Functions

- **static_assert** (details::always_false< T >::value,"Bad format of parameter")

13.57.1 Detailed Description

template<typename T> struct catsfoot::is_constructible< T >

Undefined.

The documentation for this struct was generated from the following file:

- type_traits/is_constructible.hh

13.58 catsfoot::is_constructible< T(U...)> Struct Template Reference

Tells whether T is constructible with {U...}.

`#include <is_constructible.hh>`

13.58.1 Detailed Description

template<typename T, typename... U> struct catsfoot::is_constructible< T(U...)>

Tells whether T is constructible with {U...}.

The documentation for this struct was generated from the following file:

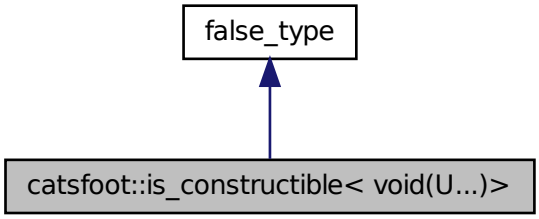
- type_traits/is_constructible.hh

13.59 catsfoot::is_constructible< void(U...)> Struct Template Reference

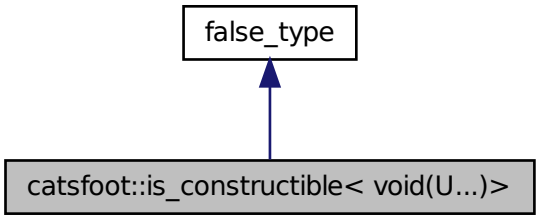
Void is a special type that may induce extra error messages.

`#include <is_constructible.hh>`

Inheritance diagram for catsfoot::is_constructible< void(U...)>:



Collaboration diagram for catsfoot::is_constructible< void(U...)>:



13.59.1 Detailed Description

template<typename... U> struct catsfoot::is_constructible< void(U...)>

Void is a special type that may induce extra error messages.

The documentation for this struct was generated from the following file:

- type_traits/is_constructible.hh

13.60 catsfoot::details::is_constructible_work_around< false, T()> Struct Template Reference

If not a class, is it always default constructible (maybe)

`#include <is_constructible.hh>`

13.60.1 Detailed Description

template<typename T> struct catsfoot::details::is_constructible_work_around< false, T()>

If not a class, is it always default constructible (maybe)

The documentation for this struct was generated from the following file:

- type_traits/is_constructible.hh

13.61 catsfoot::details::is_constructible_work_around< false, T(U)> Struct Template Reference

If is explicitly convertible and not a class, then it is like constructible.

`#include <is_constructible.hh>`

13.61.1 Detailed Description

template<typename T, typename U> struct catsfoot::details::is_constructible_work_around< false, T(U)>

If is explicitly convertible and not a class, then it is like constructible.

The documentation for this struct was generated from the following file:

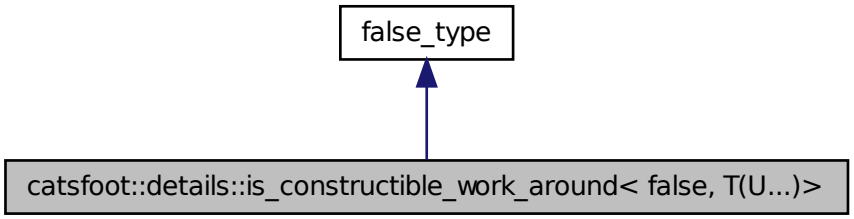
- type_traits/is_constructible.hh

13.62 catsfoot::details::is_constructible_work_around< false, T(U...)> Struct Template Reference

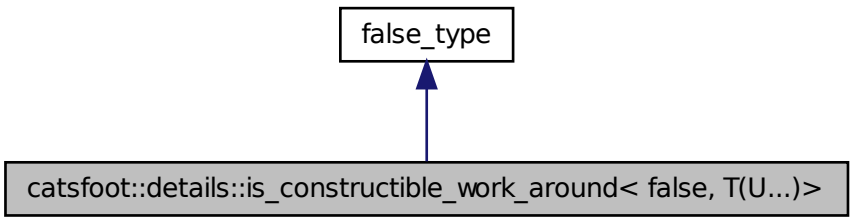
If T is not a class, it cannot be built with several parameter.

`#include <is_constructible.hh>`

Inheritance diagram for catsfoot::details::is_constructible_work_around< false, T(U...)>:



Collaboration diagram for catsfoot::details::is_constructible_work_around< false, T(U...)>:



13.62.1 Detailed Description

template<typename T, typename... U> struct catsfoot::details::is_constructible_work_around< false, T(U...)>

If T is not a class, it cannot be built with several parameter.

The documentation for this struct was generated from the following file:

- type_traits/is_constructible.hh

13.63 catsfoot::details::is_constructible_work_around< true, T(U...)> Struct Template Reference

If T is a class, then use the standard library.

`#include <is_constructible.hh>`

13.63.1 Detailed Description

template<typename T, typename... U> struct catsfoot::details::is_constructible_work_around< true, T(U...)>

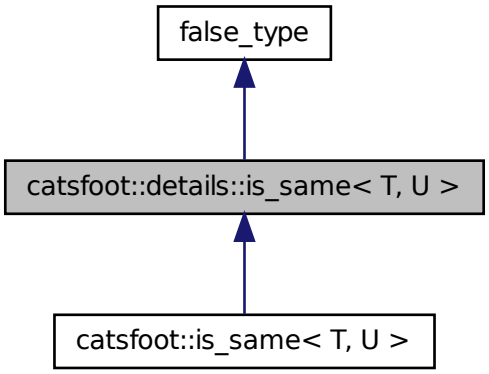
If T is a class, then use the standard library.

The documentation for this struct was generated from the following file:

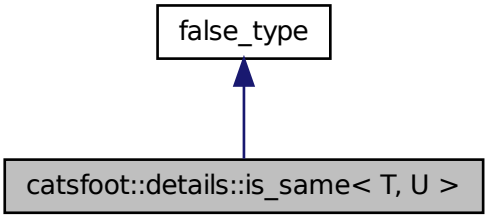
- type_traits/is_constructible.hh

13.64 catsfoot::details::is_same< T, U > Struct Template Reference

Inheritance diagram for catsfoot::details::is_same< T, U >:



Collaboration diagram for catsfoot::details::is_same< T, U >:



template<typename T, typename U> struct catsfoot::details::is_same< T, U >

The documentation for this struct was generated from the following file:

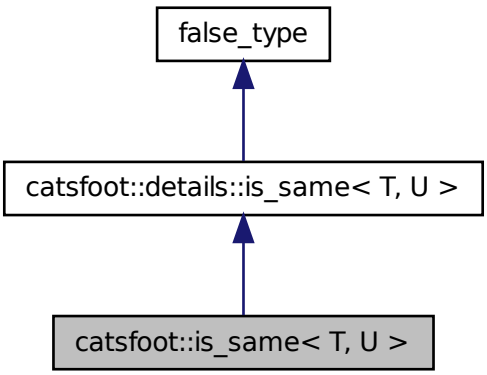
- concept/concepts.hh

13.65 catsfoot::is_same< T, U > Struct Template Reference

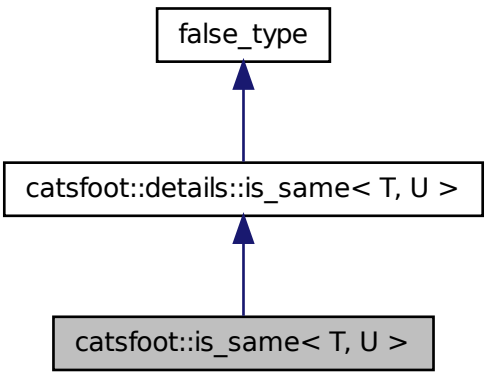
Check whether *T* and *U* are the same type.

`#include <concepts.hh>`

Inheritance diagram for catsfoot::is_same< T, U >:



Collaboration diagram for catsfoot::is_same< T, U >:



13.65.1 Detailed Description

template<typename T, typename U> struct catsfoot::is_same< T, U >

Check whether *T* and *U* are the same type.

The documentation for this struct was generated from the following file:

- concept/concepts.hh

13.66 catsfoot::details::is_same< T, T > Struct Template Reference

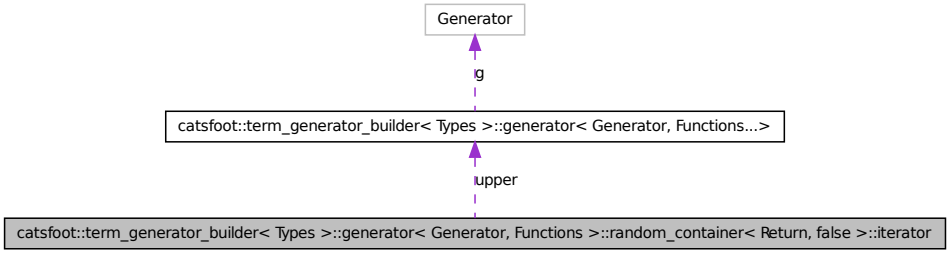
template<typename T> struct catsfoot::details::is_same< T, T >

The documentation for this struct was generated from the following file:

- concept/concepts.hh

13.67 catsfoot::term_generator_builder< Types >::generator< Generator, Functions >::random_container< Return, false >::iterator Struct Reference

Collaboration diagram for catsfoot::term_generator_builder< Types >::generator< Generator, Functions >::random_container< Return, false >::iterator:



Public Member Functions

- **iterator** (generator< Generator, Functions...> &upper, size_t pos=0u)
- **iterator** (const iterator &other)
- Return & **operator*** () const
- iterator & **operator++** ()

- iterator & **operator++** (int)
- bool **operator==** (const iterator &other) const
- bool **operator!=** (const iterator &other) const
- iterator & **operator=** (iterator &&other)
- iterator & **operator=** (const iterator &other)

template<typename... Types>template<typename Generator, typename... Functions>template<typename Return> struct catsfoot::term_generator_builder< Types >::generator< Generator, Functions >::random_container< Return, false >::iterator

The documentation for this struct was generated from the following file:

- dataset/random_term_generator.hh

13.68 catsfoot::list_data_generator< T > Struct Template Reference

Generator using lists of data provided by the user.

`#include <dataset.hh>`

Public Member Functions

- **list_data_generator** (std::initializer_list< T >...lists)
- **list_data_generator** (const list_data_generator &)
- template<typename Return , typename NonRefReturn = typename std::decay<Return>::type, typename pos = typename details::position<NonRefReturn, T...>::type>
  std::list< NonRefReturn > & **get** (selector< Return >)
- template<typename Return , typename NonRefReturn = typename std::decay<Return>::type, typename pos = typename details::position<NonRefReturn, T...>::type>
  const std::list< NonRefReturn > & **get** (selector< Return >) const

13.68.1 Detailed Description

template<typename... T> struct catsfoot::list_data_generator< T >

Generator using lists of data provided by the user.

The documentation for this struct was generated from the following file:

- dataset/dataset.hh

13.69 catsfoot::details::member_wrapper< Ret(T::*)(Args...) const > Struct Template Reference

Wraps a const member.

`#include <function_wrappers.hh>`

Public Member Functions

- **member_wrapper** (Ret(T::*f)(Args...) const)
- Ret **operator()** (const T &t, Args &&...args...) const

13.69.1 Detailed Description

template<typename T, typename Ret, typename... Args> struct catsfoot::details::member_wrapper< Ret(T::*)(Args...) const >

Wraps a const member.

The documentation for this struct was generated from the following file:

- wrappers/function_wrappers.hh

13.70 catsfoot::details::member_wrapper< Ret(T::*)(Args...)> Struct Template Reference

Wraps a non-const member.

`#include <function_wrappers.hh>`

Public Member Functions

- **member_wrapper** (Ret(T::*f)(Args...))
- Ret **operator()** (T &t, Args &&...args...) const

13.70.1 Detailed Description

template<typename T, typename Ret, typename... Args> struct catsfoot::details::member_wrapper< Ret(T::*)(Args...)>

Wraps a non-const member.

The documentation for this struct was generated from the following file:

- wrappers/function_wrappers.hh

13.71 catsfoot::details::member_wrapper< void(T::*)(Args...)> Struct Template Reference

Wraps a non-const void returning member.

`#include <function_wrappers.hh>`

Public Member Functions

- **member_wrapper** (void(T::*f)(Args...))
- T **operator()** (T t, Args &&...args...) const

13.71.1 Detailed Description

template<typename T, typename... Args> struct catsfoot::details::member_wrapper< void(T::*)(Args...)>

Wraps a non-const void returning member.

The documentation for this struct was generated from the following file:

- wrappers/function_wrappers.hh

13.72 catsfoot::details::number_function_returns< T > Struct Template Reference

No function given.

`#include <random_term_generator.hh>`

Static Public Attributes

- static const size_t **value** = 0u

13.72.1 Detailed Description

template<typename T> struct catsfoot::details::number_function_returns< T >

No function given.

The documentation for this struct was generated from the following file:

- dataset/random_term_generator.hh

13.73 catsfoot::details::number_function_returns< T, std::function< const T &(Args...)>, Functions...> Struct Template Reference

The first function returns a T.

`#include <random_term_generator.hh>`

Static Public Attributes

- static const size_t **value**

13.73.1 Detailed Description

template<typename T, typename... Args, typename... Functions> struct catsfoot::details::number_function_returns< T, std::function< const T &(Args...)>, Functions...>

The first function returns a T.

13.73.2 Member Data Documentation

13.73.2.1 template<typename T , typename... Args, typename... Functions> const size_t catsfoot::details::number_function_returns< T, std::function< const T &(Args...)>, Functions...>::value [static]

Initial value:

```
1
2         number_function_returns<T, Functions...>::value + 1
```

The documentation for this struct was generated from the following file:

- dataset/random_term_generator.hh

13.74 catsfoot::details::number_function_returns< T, std::function< Ret(Args...)>, Functions...> Struct Template Reference

The first function does not return a T.

`#include <random_term_generator.hh>`

Static Public Attributes

- static const size_t **value**

13.74.1 Detailed Description

template<typename T, typename Ret, typename... Args, typename... Functions> struct catsfoot::details::number_function_returns< T, std::function< Ret(Args...)>, Functions...>

The first function does not return a T.

13.74.2 Member Data Documentation

13.74.2.1 template<typename T , typename Ret , typename... Args, typename... Functions> const size_t catsfoot::details::number_function_returns< T, std::function< Ret(Args...)>, Functions...>::value [static]

Initial value:

```
1
2         number_function_returns<T, Functions...>::value
```

The documentation for this struct was generated from the following file:

- dataset/random_term_generator.hh

13.75 catsfoot::details::number_function_returns< T, std::function< T &&(Args...)>, Functions...> Struct Template Reference

The first function returns a T.

`#include <random_term_generator.hh>`

Static Public Attributes

- static const size_t **value**

13.75.1 Detailed Description

template<typename T, typename... Args, typename... Functions> struct catsfoot::details::number_function_returns< T, std::function< T &&(Args...)>, Functions...>

The first function returns a T.

13.75.2 Member Data Documentation

13.75.2.1 template<typename T , typename... Args, typename... Functions> const size_t catsfoot::details::number_function_returns< T, std::function< T &&(Args...)>, Functions...>::value [static]

Initial value:

```
1
2         number_function_returns<T, Functions...>::value + 1
```

The documentation for this struct was generated from the following file:

- dataset/random_term_generator.hh

13.76 catsfoot::details::number_function_returns< T, std::function< T &(Args...)>, Functions...> Struct Template Reference

The first function returns a T.

`#include <random_term_generator.hh>`

Static Public Attributes

- static const size_t **value**

13.76.1 Detailed Description

template<typename T, typename... Args, typename... Functions> struct catsfoot::details::number_function_returns< T, std::function< T &(Args...)>, Functions...>

The first function returns a T.

13.76.2 Member Data Documentation

13.76.2.1 template<typename T , typename... Args, typename... Functions> const size_t catsfoot::details::number_function_returns< T, std::function< T &(Args...)>, Functions...>::value [static]

Initial value:

```
1
2         number_function_returns<T, Functions...>::value + 1
```

The documentation for this struct was generated from the following file:

- dataset/random_term_generator.hh

13.77 catsfoot::details::number_function_returns< T, std::function< T(Args...)>, Functions...> Struct Template Reference

The first function returns a T.

`#include <random_term_generator.hh>`

Static Public Attributes

- static const size_t **value**

13.77.1 Detailed Description

template<typename T, typename... Args, typename... Functions> struct catsfoot::details::number_function_returns< T, std::function< T(Args...)>, Functions...>

The first function returns a T.

13.77.2 Member Data Documentation

13.77.2.1 template<typename T , typename... Args, typename... Functions> const size_t catsfoot::details::number_function_returns< T, std::function< T(Args...)>, Functions...>::value [static]

Initial value:

```
1
2         number_function_returns<T, Functions...>::value + 1
```

The documentation for this struct was generated from the following file:

- dataset/random_term_generator.hh

13.78 catsfoot::details::number_ground_terms< T > Struct Template Reference

No function.

`#include <random_term_generator.hh>`

Static Public Attributes

- static const size_t **value** = 0u

13.78.1 Detailed Description

template<typename T> struct catsfoot::details::number_ground_terms< T >

No function.

The documentation for this struct was generated from the following file:

- dataset/random_term_generator.hh

13.79 catsfoot::details::number_ground_terms< T, std::function< const T &()>, Functions...> Struct Template Reference

First function is ground term.

`#include <random_term_generator.hh>`

Static Public Attributes

- static const size_t **value**

13.79.1 Detailed Description

template<typename T, typename... Functions> struct catsfoot::details::number_ground_terms< T, std::function< const T &()>, Functions...>

First function is ground term.

13.79.2 Member Data Documentation

13.79.2.1 template<typename T , typename... Functions> const size_t catsfoot::details::number_ground_terms< T, std::function< const T &()>, Functions...>::value [static]

Initial value:

```
1
2         number_ground_terms<T, Functions...>::value + 1
```

The documentation for this struct was generated from the following file:

- dataset/random_term_generator.hh

13.80 catsfoot::details::number_ground_terms< T, std::function< Ret(Args...)>, Functions...> Struct Template Reference

First function is not ground term.

`#include <random_term_generator.hh>`

Static Public Attributes

- static const size_t **value**

13.80.1 Detailed Description

template<typename T, typename Ret, typename... Args, typename... Functions> struct catsfoot::details::number_ground_terms< T, std::function< Ret(Args...)>, Functions...>

First function is not ground term.

13.80.2 Member Data Documentation

13.80.2.1 template<typename T , typename Ret , typename... Args, typename... Functions> const size_t catsfoot::details::number_ground_terms< T, std::function< Ret(Args...)>, Functions...>::value [static]

Initial value:

```
1
2       number_ground_terms<T, Functions...>::value
```

The documentation for this struct was generated from the following file:

- dataset/random_term_generator.hh

13.81 catsfoot::details::number_ground_terms< T, std::function< T &&()>, Functions...> Struct Template Reference

First function is ground term.

`#include <random_term_generator.hh>`

Static Public Attributes

- static const size_t **value**

13.81.1 Detailed Description

template<typename T, typename... Functions> struct catsfoot::details::number_ground_terms< T, std::function< T &&()>, Functions...>

First function is ground term.

13.81.2 Member Data Documentation

13.81.2.1 template<typename T , typename... Functions> const size_t catsfoot::details::number_ground_terms< T, std::function< T &&()>, Functions...>::value [static]

Initial value:

```
1
2         number_ground_terms<T, Functions ... >::value + 1
```

The documentation for this struct was generated from the following file:

- dataset/random_term_generator.hh

13.82 catsfoot::details::number_ground_terms< T, std::function< T &()>, Functions...> Struct Template Reference

First function is ground term.

`#include <random_term_generator.hh>`

Static Public Attributes

- static const size_t **value**

13.82.1 Detailed Description

template<typename T, typename... Functions> struct catsfoot::details::number_ground_terms< T, std::function< T &()>, Functions...>

First function is ground term.

13.82.2 Member Data Documentation

13.82.2.1 template<typename T , typename... Functions> const size_t catsfoot::details::number_ground_terms< T, std::function< T &()>, Functions...>::value [static]

Initial value:

```
1
2         number_ground_terms<T, Functions...>::value + 1
```

The documentation for this struct was generated from the following file:

- dataset/random_term_generator.hh

13.83 catsfoot::details::number_ground_terms< T, std::function< T()>, Functions...> Struct Template Reference

First function is ground term.

`#include <random_term_generator.hh>`

Static Public Attributes

- static const size_t **value**

13.83.1 Detailed Description

template<typename T, typename... Functions> struct catsfoot::details::number_ground_terms< T, std::function< T()>, Functions...>

First function is ground term.

13.83.2 Member Data Documentation

13.83.2.1 template<typename T , typename... Functions> const size_t catsfoot::details::number_ground_terms< T, std::function< T()>, Functions...>::value [static]

Initial value:

```
1
2         number_ground_terms<T, Functions...>::value + 1
```

The documentation for this struct was generated from the following file:

- dataset/random_term_generator.hh

13.84 catsfoot::op_eq Struct Reference

Wraps *operator*==.

`#include <operators.hh>`

Public Member Functions

- template<typename T , typename U , typename Ret = decltype(std::declval<T>() == std::declval<U>())>
  Ret **operator()** (T &&t, U &&u) const

13.84.1 Detailed Description

Wraps *operator*==.

The documentation for this struct was generated from the following file:

- wrappers/operators.hh

13.85 catsfoot::op_inc Struct Reference

Wraps *operator*++.

`#include <operators.hh>`

Public Member Functions

- template<typename T , typename Ret = decltype(++std::declval<T>())>
  Ret **operator()** (T &&t) const

13.85.1 Detailed Description

Wraps *operator*++.

The documentation for this struct was generated from the following file:

- wrappers/operators.hh

13.86 catsfoot::op_lsh Struct Reference

Wraps *operator*<<.

`#include <operators.hh>`

Public Member Functions

- template<typename T , typename U , typename Ret = decltype(std::declval<T>() << std::declval<U>())>
  Ret **operator()** (T &&t, U &&u) const

13.86.1 Detailed Description

Wraps *operator<<*.

The documentation for this struct was generated from the following file:

- wrappers/operators.hh

13.87 catsfoot::op_lt Struct Reference

Wraps *operator<*.

`#include <operators.hh>`

Public Member Functions

- template<typename T , typename U , typename Ret = decltype(std::declval<T>() < std::declval<U>())>
  Ret **operator()** (T &&t, U &&u) const

13.87.1 Detailed Description

Wraps *operator<*.

The documentation for this struct was generated from the following file:

- wrappers/operators.hh

13.88 catsfoot::op_neq Struct Reference

Wraps *operator!=*.

`#include <operators.hh>`

Public Member Functions

- template<typename T , typename U , typename Ret = decltype(std::declval<T>() != std::declval<U>())>
  Ret **operator()** (T &&t, U &&u) const

13.88.1 Detailed Description

Wraps *operator!=*.

The documentation for this struct was generated from the following file:

- wrappers/operators.hh

13.89 catsfoot::op_plus Struct Reference

Wraps *operator+*.

`#include <operators.hh>`

Public Member Functions

- template<typename T , typename U , typename Ret = decltype(std::declval<T>() + std::declval<U>())>
  Ret **operator()** (T &&t, U &&u) const

13.89.1 Detailed Description

Wraps *operator+*.

The documentation for this struct was generated from the following file:

- wrappers/operators.hh

13.90 catsfoot::op_post_inc Struct Reference

Wraps *operator++(T&, int)*

`#include <operators.hh>`

Public Member Functions

- template<typename T , typename Ret = decltype(++std::declval<T>())>
  Ret **operator()** (T &&t) const

13.90.1 Detailed Description

Wraps *operator++(T&, int)*

The documentation for this struct was generated from the following file:

- wrappers/operators.hh

13.91 catsfoot::op_star Struct Reference

Wraps *operator*∗.

`#include <operators.hh>`

Public Member Functions

- template<typename T , typename Ret = decltype(∗std::declval<T>())>
  Ret **operator()** (T &&t) const

13.91.1 Detailed Description

Wraps *operator*∗.

The documentation for this struct was generated from the following file:

- wrappers/operators.hh

13.92 catsfoot::op_times Struct Reference

Wraps *operator*∗.

`#include <operators.hh>`

Public Member Functions

- template<typename T , typename U , typename Ret = decltype(std::declval<T>() ∗ std::declval<U>())>
  Ret **operator()** (T &&t, U &&u) const

13.92.1 Detailed Description

Wraps *operator*∗.

The documentation for this struct was generated from the following file:

- wrappers/operators.hh

13.93 catsfoot::pick_functor< T, Generator > Struct Template Reference

Public Member Functions

- **pick_functor** (Generator &g)

- **pick_functor** (const pick_functor &other)
- **pick_functor** (pick_functor &&other)
- pick_functor & **operator=** (pick_functor &&other)
- pick_functor & **operator=** (const pick_functor &other)
- T & **operator()** ()

template<typename T, typename Generator> struct catsfoot::pick_functor< T, Generator >

The documentation for this struct was generated from the following file:

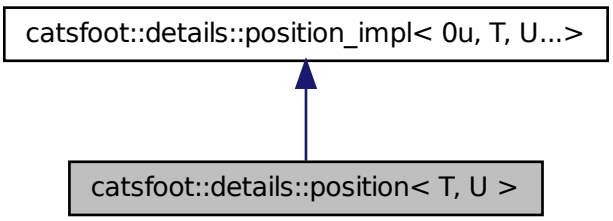
- dataset/pick.hh

13.94 catsfoot::details::position< T, U > Struct Template Reference

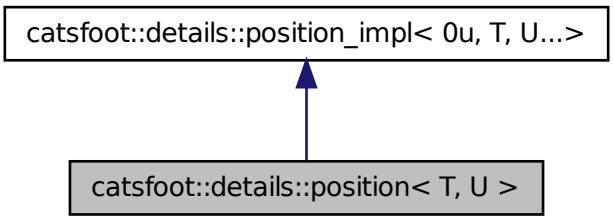
Finds the first occurrence of a type in a list.

`#include <position.hh>`

Inheritance diagram for catsfoot::details::position< T, U >:



Collaboration diagram for catsfoot::details::position< T, U >:



13.94.1 Detailed Description

template<typename T, typename... U> struct catsfoot::details::position< T, U >

Finds the first occurrence of a type in a list.

The documentation for this struct was generated from the following file:

- dataset/position.hh

13.95 catsfoot::details::position_impl< size_t, typename, > Struct Template Reference

Finds the first occurrence of a type in a list.

`#include <position.hh>`

13.95.1 Detailed Description

template<size_t, typename, typename...> struct catsfoot::details::position_impl< size_t, typename, >

Finds the first occurrence of a type in a list.

The documentation for this struct was generated from the following file:

- dataset/position.hh

13.96 catsfoot::details::position_impl< N, T, T, U...> Struct Template Reference

Finds the first occurrence of a type in a list.

`#include <position.hh>`

13.96.1 Detailed Description

template<size_t N, typename T, typename... U> struct catsfoot::details::position_impl< N, T, T, U...>

Finds the first occurrence of a type in a list.

The documentation for this struct was generated from the following file:

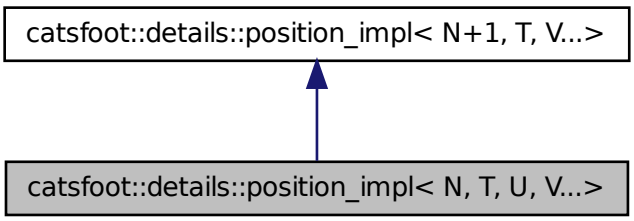
- dataset/position.hh

13.97 catsfoot::details::position_impl< N, T, U, V...> Struct Template Reference

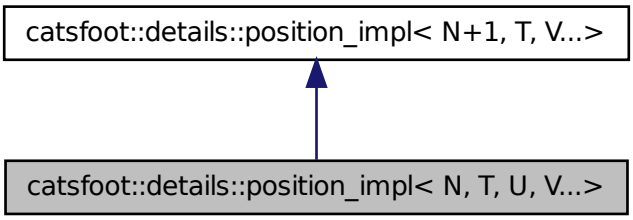
Finds the first occurrence of a type in a list.

`#include <position.hh>`

Inheritance diagram for catsfoot::details::position_impl< N, T, U, V...>:



Collaboration diagram for catsfoot::details::position_impl< N, T, U, V...>:



13.97.1 Detailed Description

template<size_t N, typename T, typename U, typename... V> struct catsfoot::details::position_impl< N, T, U, V...>

Finds the first occurrence of a type in a list.

The documentation for this struct was generated from the following file:

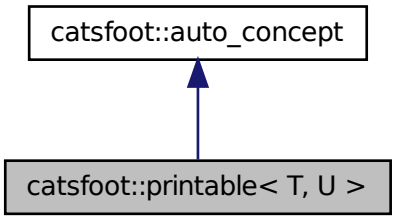
- dataset/position.hh

13.98 catsfoot::printable< T, U > Struct Template Reference

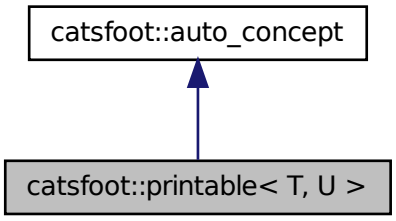
Check whether *U* is printable on a stream *T*.

`#include <concepts.hh>`

Inheritance diagram for catsfoot::printable< T, U >:



Collaboration diagram for catsfoot::printable< T, U >:



Public Types

- typedef concept_list< is_callable< op_lsh(T, U)> > **requirements**

13.98.1 Detailed Description

template<typename T, typename U> struct catsfoot::printable< T, U >

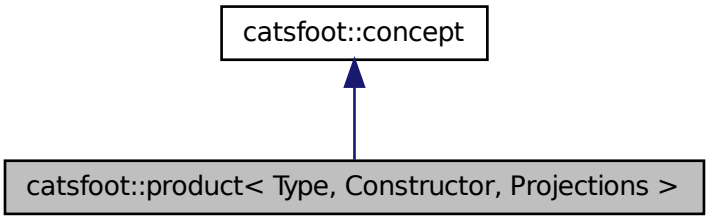
Check whether *U* is printable on a stream *T*.

The documentation for this struct was generated from the following file:

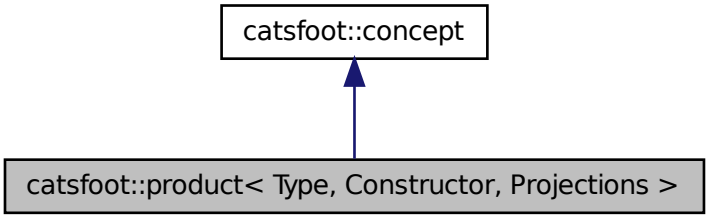
- concept/concepts.hh

13.99 catsfoot::product< Type, Constructor, Projections > Struct Template Reference

Inheritance diagram for catsfoot::product< Type, Constructor, Projections >:



Collaboration diagram for catsfoot::product< Type, Constructor, Projections >:



Public Types

- typedef concept_list< is_callable< Projections(const Type &)>..., is_callable< Constructor(typename is_callable< Projections(const Type &)>::result_type...)>, std::is_convertible< typename is_callable< Constructor(typename is_callable< Projections(const Type &)>::result_type...)>::result_type, Type >, equivalence_eq< Type >, equivalence_eq< typename is_callable< Projections(const Type &)>::result_type >... > **requirements**

Public Member Functions

- **AXIOMS** (projections, universality)

Static Public Member Functions

- static void **projections** (const std::tuple< typename is_callable< Projections(const Type &)>::result_type...> &comps, const std::tuple< Projections...> &p, const Constructor &constr)
- static void **universality** (const Type &t, const std::tuple< Projections...> &p, const Constructor &constr)

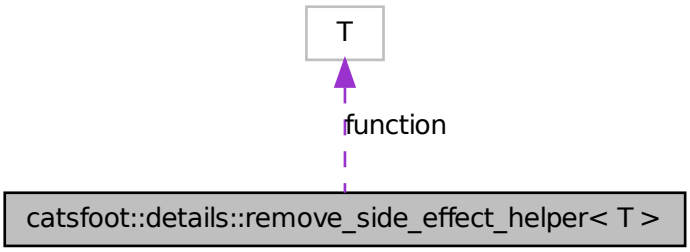
template<typename Type, typename Constructor, typename... Projections> struct catsfoot::product< Type, Constructor, Projections >

The documentation for this struct was generated from the following file:

- concept/product.hh

13.100 catsfoot::details::remove_side_effect_helper< T > Struct Template Reference

Collaboration diagram for catsfoot::details::remove_side_effect_helper< T >:



Public Member Functions

- template<typename U , typename = typename std::enable_if <std::is_same<typename std::decay<U>::type, T>::value> ::type>
  remove_side_effect_helper (U &&u)

- **remove_side_effect_helper** (const remove_side_effect_helper &other)
- **remove_side_effect_helper** (remove_side_effect_helper &&other)
- template<typename... Args, typename Ret = decltype(std::declval<T>()(std::declval<Args>()...))> Ret **operator()** (Args &&...args) const

template<typename T> struct catsfoot::details::remove_side_effect_helper< T >

The documentation for this struct was generated from the following file:

- wrappers/function_wrappers.hh

13.101 catsfoot::details::return_of< T > Struct Template Reference

template<typename T> struct catsfoot::details::return_of< T >

The documentation for this struct was generated from the following file:

- wrappers/function_wrappers.hh

13.102 catsfoot::details::return_of< T(Args...)> Struct Template Reference

Public Types

- typedef T **type**

template<typename T, typename... Args> struct catsfoot::details::return_of< T(Args...)>

The documentation for this struct was generated from the following file:

- wrappers/function_wrappers.hh

13.103 catsfoot::selector< T > Struct Template Reference

Constructible type used to select a type for overloaded functions.

`#include <dataset.hh>`

Public Types

- typedef T **type**

13.103.1 Detailed Description

template<typename T> struct catsfoot::selector< T >

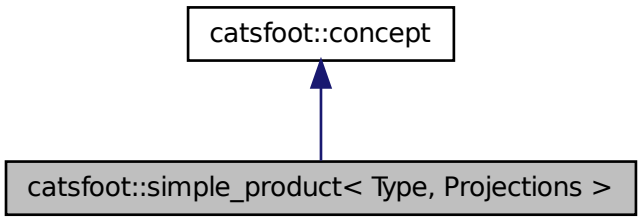
Constructible type used to select a type for overloaded functions.

The documentation for this struct was generated from the following file:

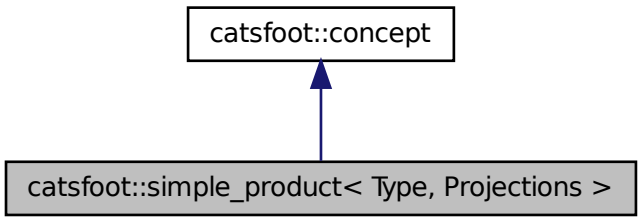
- dataset/dataset.hh

13.104 catsfoot::simple_product< Type, Projections > Struct Template Reference

Inheritance diagram for catsfoot::simple_product< Type, Projections >:



Collaboration diagram for catsfoot::simple_product< Type, Projections >:



Public Types

- typedef concept_list< is_constructible< Type(typename is_callable< Projections(const Type &)>::result_type...)>, product< Type, constructor_wrap< Type >, Projections...> > **requirements**

template<typename Type, typename... Projections> struct catsfoot::simple_product< Type, Projections >

The documentation for this struct was generated from the following file:

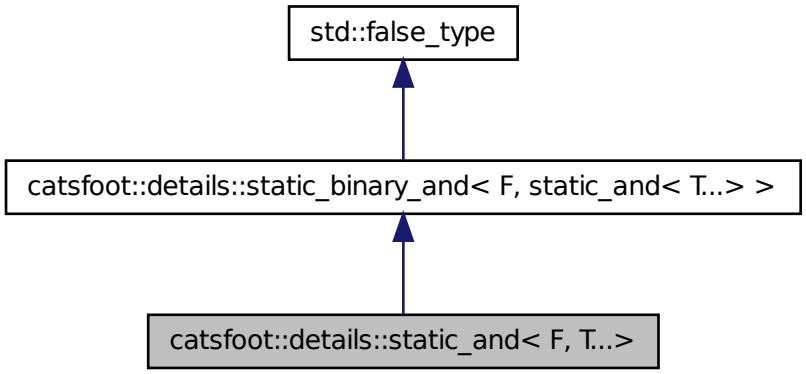
- concept/product.hh

13.105 catsfoot::details::static_and< F, T...> Struct Template Reference

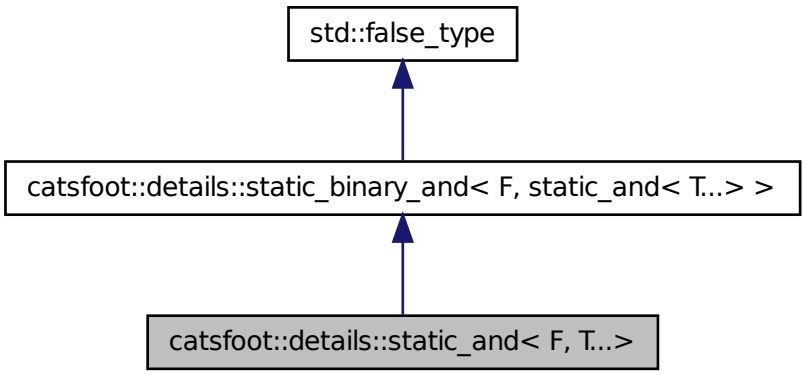
Predicate true if all predicate parameters are true.

`#include <static_and.hh>`

Inheritance diagram for catsfoot::details::static_and< F, T...>:



Collaboration diagram for catsfoot::details::static_and< F, T...>:



13.105.1 Detailed Description

template<typename F, typename... T> struct catsfoot::details::static_and< F, T...>

Predicate true if all predicate parameters are true.

The documentation for this struct was generated from the following file:

- concept/static_and.hh

13.106 catsfoot::details::static_and<> Struct Template Reference

Predicate true if all predicate parameters are true.

`#include <static_and.hh>`

13.106.1 Detailed Description

template<> struct catsfoot::details::static_and<>

Predicate true if all predicate parameters are true.

The documentation for this struct was generated from the following file:

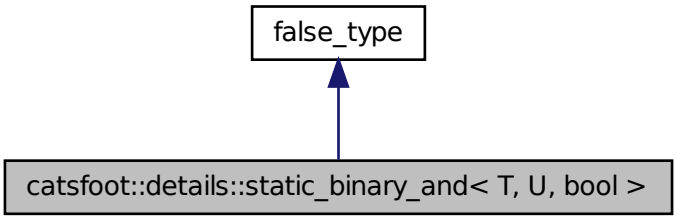
- concept/static_and.hh

13.107 catsfoot::details::static_binary_and< T, U, bool > Struct Template Reference

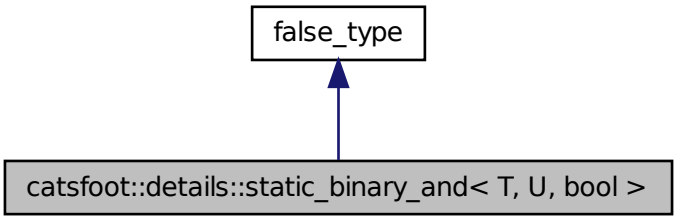
Predicate true if all predicate parameters are true.

`#include <static_and.hh>`

Inheritance diagram for catsfoot::details::static_binary_and< T, U, bool >:



Collaboration diagram for catsfoot::details::static_binary_and< T, U, bool >:



13.107.1 Detailed Description

template<typename T, typename U, bool = T::value> struct catsfoot::details::static_binary_and< T, U, bool >

Predicate true if all predicate parameters are true.

The documentation for this struct was generated from the following file:

- concept/static_and.hh

13.108 catsfoot::details::static_binary_and< T, U, true > Struct Template Reference

Predicate true if all predicate parameters are true.

`#include <static_and.hh>`

13.108.1 Detailed Description

template<typename T, typename U> struct catsfoot::details::static_binary_and< T, U, true >

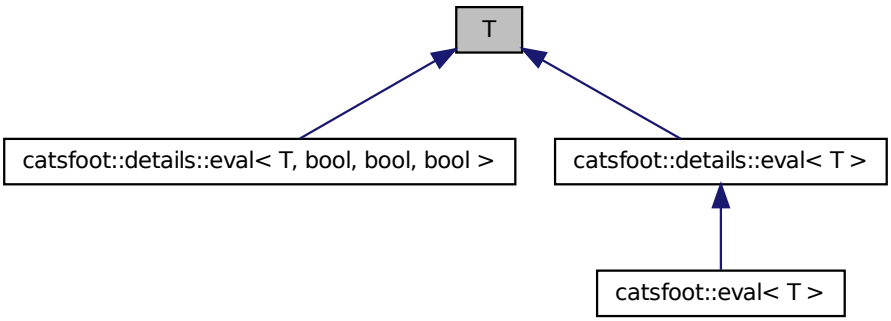
Predicate true if all predicate parameters are true.

The documentation for this struct was generated from the following file:

- concept/static_and.hh

13.109 T Class Reference

Inheritance diagram for T:



The documentation for this class was generated from the following file:

- concept/concept_tools.hh

13.110 catsfoot::term_generator_builder< Types > Struct Template Reference

Builds random term generators.

`#include <random_term_generator.hh>`

Classes

- struct generator

Public Member Functions

- **term_generator_builder** (size_t size=200u)
- template<typename Generator , typename... Functions>
  generator< Generator, typename wrapped< Functions >::type...> operator()
  (Generator &g, Functions &&...functions) const

  Build a random term generator.

13.110.1 Detailed Description

template<typename... Types> struct catsfoot::term_generator_builder< Types >

Builds random term generators.

Template Parameters

Types	The list of types it can generate

13.110.2 Member Function Documentation

13.110.2.1 template<typename... Types> template<typename Generator , typename... Functions> generator<Generator, typename wrapped<Functions>::type...> catsfoot::term_generator_builder< Types >::operator() (Generator & *g*, Functions &&... *functions*) const [inline]

Build a random term generator.

Parameters

g	A random generator engine.
functions	A list of functions.

The documentation for this struct was generated from the following file:

- dataset/random_term_generator.hh

13.111 catsfoot::details::test_all< T, bool, bool > Struct Template Reference

Test axioms of a predicate (always true)

`#include <test_all_driver.hh>`

Public Member Functions

- template<typename Generator , typename Stream = decltype(std::cerr)>
  bool **operator()** (Generator &, Stream &=std::cerr)

13.111.1 Detailed Description

template<typename T, bool = is_concept<T>::value, bool = has_get_axiom<T>::value> struct catsfoot::details::test_all< T, bool, bool >

Test axioms of a predicate (always true)

The documentation for this struct was generated from the following file:

- drivers/test_all_driver.hh

13.112 catsfoot::details::test_all< concept_list< T, U...>, false, B > Struct Template Reference

Test axioms of a list of requirements.

`#include <test_all_driver.hh>`

Public Member Functions

- template<typename Generator , typename Stream = decltype(std::cerr)>
  bool **operator()** (Generator &g, Stream &s=std::cerr)

13.112.1 Detailed Description

template<typename T, typename... U, bool B> struct catsfoot::details::test_all< concept_list< T, U...>, false, B >

Test axioms of a list of requirements.

The documentation for this struct was generated from the following file:

- drivers/test_all_driver.hh

13.113 catsfoot::details::test_all< T, true, false > Struct Template Reference

Test axioms of a concept with no local axioms.

`#include <test_all_driver.hh>`

Public Member Functions

- template<typename Generator , typename Stream = decltype(std::cerr)>
  bool **operator()** (Generator &g, Stream &s=std::cerr)

13.113.1 Detailed Description

template<typename T> struct catsfoot::details::test_all< T, true, false >

Test axioms of a concept with no local axioms.

The documentation for this struct was generated from the following file:

- drivers/test_all_driver.hh

13.114 catsfoot::details::test_all< T, true, true > Struct Template Reference

Test axioms of a concept with local axioms.

`#include <test_all_driver.hh>`

Public Member Functions

- template<typename Generator , typename... Axioms, typename Stream , typename Index = std::integral_-constant<size_t, 0>>
  bool test (Generator &g, const std::tuple< Axioms...> &axioms, Stream &s, Index=Index())

  Test axiom number "Index".

- template<typename Generator , typename... Axioms, typename Stream >
  bool test (Generator &, const std::tuple< Axioms...> &, Stream &, std::integral_-constant< size_t, sizeof...(Axioms)>)

  All axioms were tested.

- template<typename Generator , typename Stream = decltype(std::cerr)>
  bool **operator()** (Generator &g, Stream &s=std::cerr)

13.114.1 Detailed Description

template<typename T> struct catsfoot::details::test_all< T, true, true >

Test axioms of a concept with local axioms.

The documentation for this struct was generated from the following file:

- drivers/test_all_driver.hh

13.115 catsfoot::details::tester< T, U...> Struct Template Reference

Static Public Member Functions

- template<typename Generator , typename Fun , typename... Params, typename Stream >
  static bool **call_gen** (Stream &s, Generator &g, Fun f, Params &&...values)

template<typename T, typename... U> struct catsfoot::details::tester< T, U...>

The documentation for this struct was generated from the following file:

- drivers/test_driver.hh

13.116 catsfoot::details::tester<> Struct Template Reference

Static Public Member Functions

- template<typename Generator , typename Fun , typename... Params, typename Stream >
  static bool **call_gen_final** (Stream &s, Generator &, Fun f, Params &&...values)

- template<typename Generator , typename Fun , typename... Params, typename Stream >
  static bool **call_gen** (Stream &s, Generator &g, Fun f, Params &&...values)

template<> struct catsfoot::details::tester<>

The documentation for this struct was generated from the following file:

- drivers/test_driver.hh

13.117 catsfoot::details::try_all_compare< T, std::function< Ret(Args...)> > Struct Template Reference

Static Public Member Functions

- template<typename Generator , typename... OtherArgs>
  static bool **doit** (Generator &g, const selector< const T & > &, const std::function< Ret(Args...)> &f, const T &a, const T &b, const std::tuple< OtherArgs, OtherArgs > &...args...)
- template<typename Generator , typename U , typename... OtherArgs>
  static bool **doit** (Generator &g, const selector< U > &, const std::function< Ret(Args...)> &f, const T &a, const T &b, const std::tuple< OtherArgs, OtherArgs > &...args...)
- template<typename Generator , typename... OtherArgs>
  static bool **doit** (Generator &g, const std::function< Ret(Args...)> &f, const T &a, const T &b, const std::tuple< OtherArgs, OtherArgs > &...args...)
- template<typename Generator >
  static bool doit (Generator &, const std::function< Ret(Args...)> &f, const T &, const T &, const std::tuple< Args, Args > &...args...)

template<typename T, typename Ret, typename... Args> struct catsfoot::details::try_all_compare< T, std::function< Ret(Args...)> >

13.117.1 Member Function Documentation

13.117.1.1 template<typename T , typename Ret , typename... Args> template<typename Generator > static bool catsfoot::details::try_all_compare< T, std::function< Ret(Args...)> >::doit (Generator & , const std::function< Ret(Args...)> & *f*, const T & , const T & , const std::tuple< Args, Args > &... *args...*) [inline, static]

Todo

If == does not exist?

The documentation for this struct was generated from the following file:

- utils/black_box_equal.hh

13.118 catsfoot::details::try_first Struct Reference

Tag for prioritizing some overloaded functions.

`#include <try_first.hh>`

13.118.1 Detailed Description

Tag for prioritizing some overloaded functions.

The documentation for this struct was generated from the following file:

- type_traits/try_first.hh

13.119 catsfoot::details::try_second Struct Reference

Tag for prioritizing some overloaded functions.

`#include <try_first.hh>`

Public Member Functions

- **try_second** (const try_first &)

13.119.1 Detailed Description

Tag for prioritizing some overloaded functions.

The documentation for this struct was generated from the following file:

- type_traits/try_first.hh

13.120 catsfoot::details::tuple_generator< Generator > Struct Template Reference

Generates tuple from generator.

`#include <tuple_generator.hh>`

Public Member Functions

- **tuple_generator** (Generator &&g)
- **tuple_generator** (const Generator &g)
- **tuple_generator** (const tuple_generator &other)
- **tuple_generator** (tuple_generator &&other)
- template<typename U >
  decltype (tuple_generator_tool< U >()(std::declval< const Generator & >()))
  get(selector< U >)

13.120.1 Detailed Description

template<typename Generator> struct catsfoot::details::tuple_generator< Generator >

Generates tuple from generator.

The documentation for this struct was generated from the following file:

- dataset/tuple_generator.hh

13.121 catsfoot::details::tuple_generator_tool< U > Struct Template Reference

template<typename U> struct catsfoot::details::tuple_generator_tool< U >

The documentation for this struct was generated from the following file:

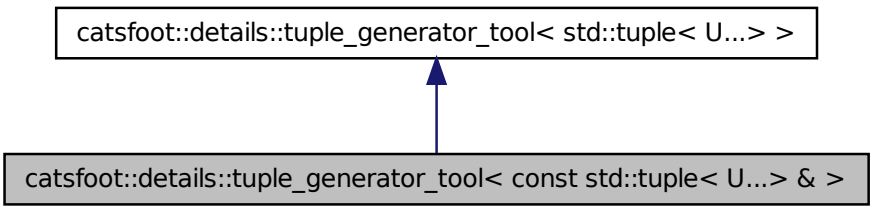
- dataset/tuple_generator.hh

13.122 catsfoot::details::tuple_generator_tool< const std::tuple< U...> & > Struct Template Reference

Generates containers of tuples std::tuple<U...>

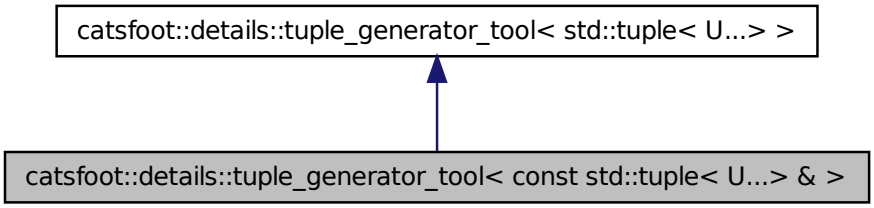
`#include <tuple_generator.hh>`

Inheritance diagram for catsfoot::details::tuple_generator_tool< const std::tuple< U...> & >:



Collaboration diagram for catsfoot::details::tuple_generator_tool< const std::tuple<

U...> & >:



13.122.1 Detailed Description

template<typename... U> struct catsfoot::details::tuple_generator_tool< const std::tuple< U...> & >

Generates containers of tuples std::tuple<U...>

The documentation for this struct was generated from the following file:

- dataset/tuple_generator.hh

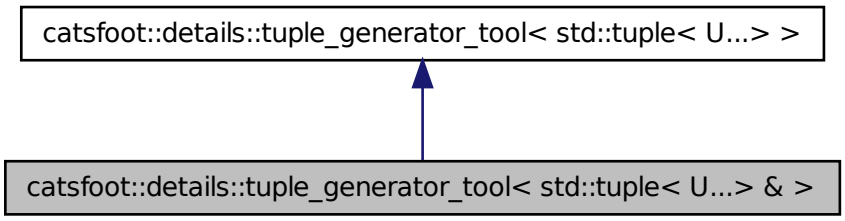
13.123 catsfoot::details::tuple_generator_tool< std::tuple< U...> & > Struct Template Reference

Generates containers of tuples std::tuple<U...>

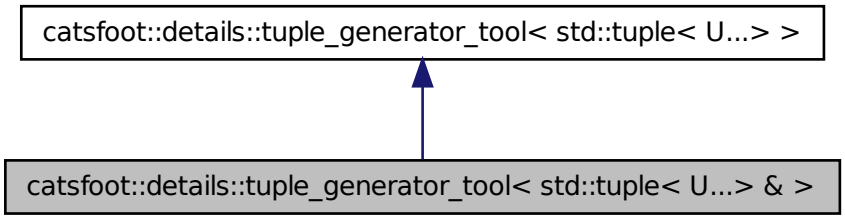
`#include <tuple_generator.hh>`

Inheritance diagram for catsfoot::details::tuple_generator_tool< std::tuple< U...> &

>:



Collaboration diagram for catsfoot::details::tuple_generator_tool< std::tuple< U...> & >:



13.123.1 Detailed Description

template<typename... U> struct catsfoot::details::tuple_generator_tool< std::tuple< U...> & >

Generates containers of tuples std::tuple<U...>

The documentation for this struct was generated from the following file:

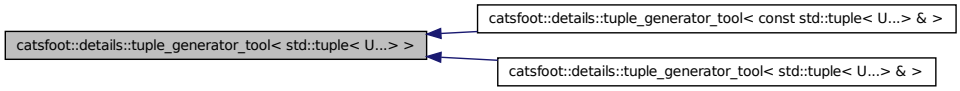
- dataset/tuple_generator.hh

13.124 catsfoot::details::tuple_generator_tool< std::tuple< U...> > Struct Template Reference

Generates containers of tuples std::tuple<U...>

`#include <tuple_generator.hh>`

Inheritance diagram for catsfoot::details::tuple_generator_tool< std::tuple< U...> >:



Public Member Functions

- template<typename... V, typename... Values, typename = void, typename = typename std::enable_if <(sizeof...(V) == sizeof...(Values))>::type>
  std::list< std::tuple< U...> > **make_list** (std::tuple< V...> &, Values...values...) const
- template<typename... V, typename... Values, typename = typename std::enable_if<(sizeof...(V) > sizeof...(Values)>
  ::type std::list< std::tuple< U...> > **make_list** (std::tuple< V...> &containers, Values...values) const
- template<typename Generator >
  std::list< std::tuple< U...> > **operator()** (Generator &g) const

13.124.1 Detailed Description

template<typename... U> struct catsfoot::details::tuple_generator_tool< std::tuple< U...> >

Generates containers of tuples std::tuple<U...>

The documentation for this struct was generated from the following file:

- dataset/tuple_generator.hh

13.125 catsfoot::undefined_member_type Struct Reference

Type used return by has_member_... in case member does not exist.

`#include <type_member.hh>`

13.125.1 Detailed Description

Type used return by has_member_... in case member does not exist.

The documentation for this struct was generated from the following file:

- type_traits/type_member.hh

13.126 catsfoot::undefined_return< T > Struct Template Reference

Return type used in case of error for is_callable.

`#include <is_callable.hh>`

13.126.1 Detailed Description

template<typename T> struct catsfoot::undefined_return< T >

Return type used in case of error for is_callable.

The documentation for this struct was generated from the following file:

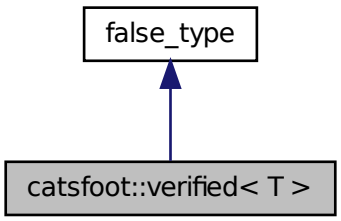
- type_traits/is_callable.hh

13.127 catsfoot::verified< T > Struct Template Reference

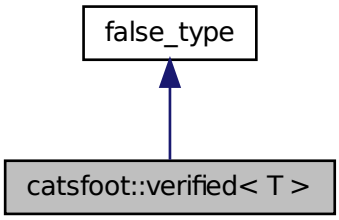
User type traits for validating models.

`#include <concept_tools.hh>`

Inheritance diagram for catsfoot::verified< T >:



Collaboration diagram for catsfoot::verified< T >:



13.127.1 Detailed Description

template<typename T> struct catsfoot::verified< T >

User type traits for validating models.

The documentation for this struct was generated from the following file:

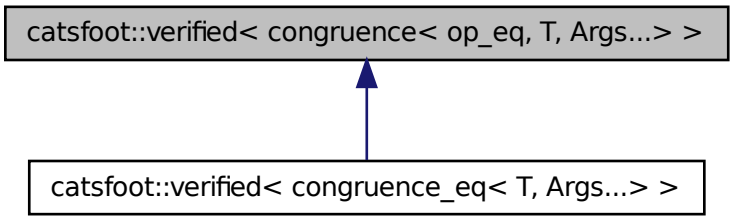
- concept/concept_tools.hh

13.128 catsfoot::verified< congruence< op_eq, T, Args...> > Struct Template Reference

When T is a congruence_eq, then T is a congruence for ==.

`#include <congruence.hh>`

Inheritance diagram for catsfoot::verified< congruence< op_eq, T, Args...> >:



13.128.1 Detailed Description

template<typename T, typename... Args> struct catsfoot::verified< congruence< op_eq, T, Args...> >

When T is a congruence_eq, then T is a congruence for ==.

The documentation for this struct was generated from the following file:

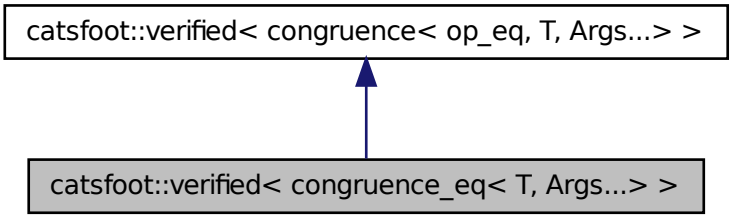
- concept/congruence.hh

13.129 catsfoot::verified< congruence_eq< T, Args...> > Struct Template Reference

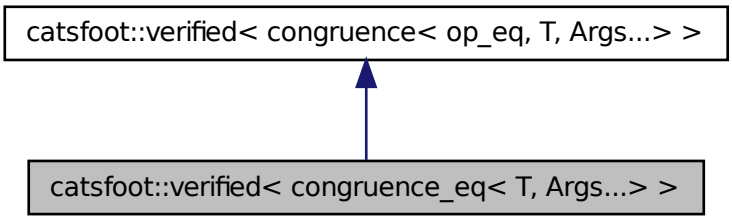
When T is a congruence for ==, then T is a congruence_eq.

`#include <congruence.hh>`

Inheritance diagram for catsfoot::verified< congruence_eq< T, Args...> >:



Collaboration diagram for catsfoot::verified< congruence_eq< T, Args...> >:



13.129.1 Detailed Description

template<typename T, typename... Args> struct catsfoot::verified< congruence_eq< T, Args...> >

When T is a congruence for ==, then T is a congruence_eq.

The documentation for this struct was generated from the following file:

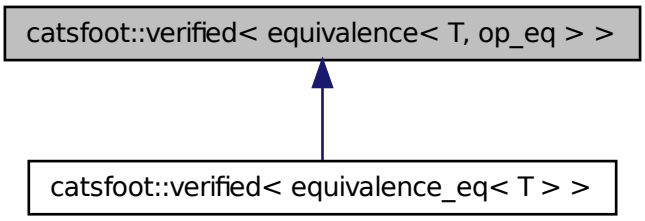
- concept/congruence.hh

13.130 catsfoot::verified< equivalence< T, op_eq > > Struct Template Reference

If T is equivalence_eq, then == is an equivalence relation to T.

`#include <congruence.hh>`

Inheritance diagram for catsfoot::verified< equivalence< T, op_eq > >:



13.130.1 Detailed Description

template<typename T> struct catsfoot::verified< equivalence< T, op_eq > >

If T is equivalence_eq, then == is an equivalence relation to T.

The documentation for this struct was generated from the following file:

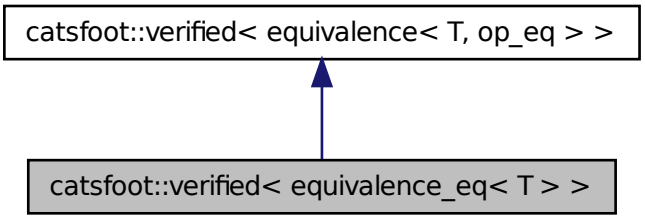
- concept/congruence.hh

13.131 catsfoot::verified< equivalence_eq< T > > Struct Template Reference

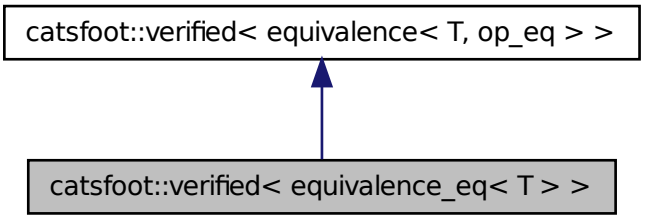
If == is an equivalence relation to T, then T is equivalence_eq.

`#include <congruence.hh>`

Inheritance diagram for catsfoot::verified< equivalence_eq< T > >:



Collaboration diagram for catsfoot::verified< equivalence_eq< T > >:



13.131.1 Detailed Description

template<typename T> struct catsfoot::verified< equivalence_eq< T > >

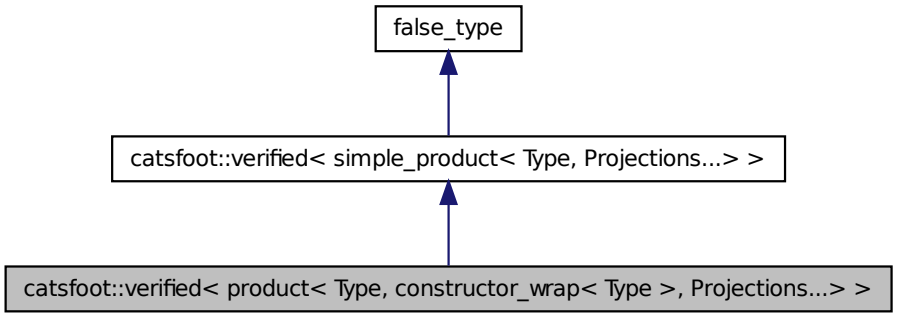
If == is an equivalence relation to T, then T is equivalence_eq.

The documentation for this struct was generated from the following file:

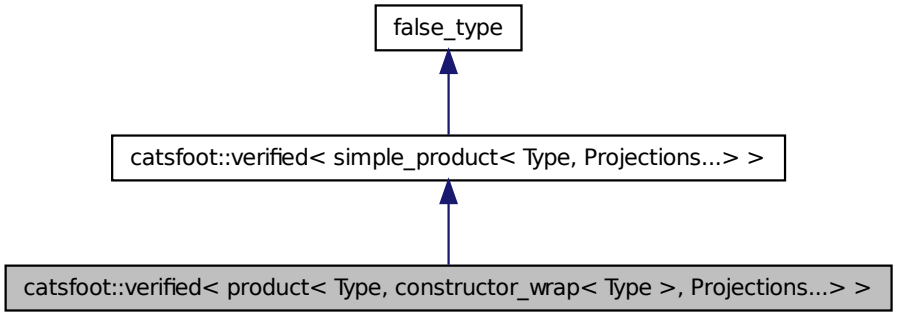
- concept/congruence.hh

13.132 catsfoot::verified< product< Type, constructor_wrap< Type >, Projections...> > Struct Template Reference

Inheritance diagram for catsfoot::verified< product< Type, constructor_wrap< Type >, Projections...> >:



Collaboration diagram for catsfoot::verified< product< Type, constructor_wrap< Type >, Projections...> >:



template<typename Type, typename... Projections> struct catsfoot::verified< product< Type, constructor_wrap< Type >, Projections...> >

The documentation for this struct was generated from the following file:

- concept/product.hh

13.133 catsfoot::wrapped< T > Struct Template Reference

Get the return type of the wrap function.

`#include <function_wrappers.hh>`

Public Member Functions

- typedef **decltype** (wrap(std::declval< T >())) type

13.133.1 Detailed Description

template<typename T> struct catsfoot::wrapped< T >

Get the return type of the wrap function.

The documentation for this struct was generated from the following file:

- wrappers/function_wrappers.hh

13.134 catsfoot::wrapped_constructor< T > Struct Template Reference

Wraps constructor T.

`#include <operators.hh>`

13.134.1 Detailed Description

template<typename T> struct catsfoot::wrapped_constructor< T >

Wraps constructor T.

The documentation for this struct was generated from the following file:

- wrappers/operators.hh

13.135 catsfoot::details::wrapped_constructor< T(Args...), false > Struct Template Reference

T(Args...) is not constructible.

`#include <operators.hh>`

13.135.1 Detailed Description

template<typename T, typename... Args> struct catsfoot::details::wrapped_constructor< T(Args...), false >

T(Args...) is not constructible.

The documentation for this struct was generated from the following file:

- wrappers/operators.hh

13.136 catsfoot::details::wrapped_constructor< T(Args...), true > Struct Template Reference

Wraps constructor T(Args...)

`#include <operators.hh>`

Public Member Functions

- T **operator()** (Args...args...) const

13.136.1 Detailed Description

template<typename T, typename... Args> struct catsfoot::details::wrapped_constructor< T(Args...), true >

Wraps constructor T(Args...)

The documentation for this struct was generated from the following file:

- wrappers/operators.hh

Index