MODERN GENERIC PROGRAMMING

Paul Fultz II

MOTIVATION

```
template<class T>
void increment(T& x)
{
    ++x;
}

template<class T>
void twice(T& x)
{
    increment(x);
    increment(x);
}
```

MOTIVATION

```
foo f;
twice(f);
```

MOTIVATION

```
reqs.cpp:9:5: error: cannot increment value of type 'foo'
    ++x;
    ^ ~
reqs.cpp:15:5: note: in instantiation of function template specialization 'increment
    increment(x);
    ^
reqs.cpp:28:5: note: in instantiation of function template specialization 'twice<foo
    twice(f);
    ^
```

TYPE REQUIREMENTS

- Specify type requirements(or concepts)
  - Set of valid expressions that can be performed on a type or types
  - Provide documentation on these requirements
  - Check these type requirements with the compiler

USING BOOST.CONCEPTCHECK

```
template<class T>
struct Incrementable
{
    BOOST_CONCEPT_USAGE(Incrementable)
    {
        x++;
        ++x;
    }
    T x;
};
```

USING BOOST.CONCEPTCHECK

```
template<class T>
void increment(T& x)
{
    ++x;
}

template<class T>
void twice(T& x)
{
    BOOST_CONCEPT_ASSERT((Incrementable<T>));
    increment(x);
    increment(x);
}
```

USING BOOST.CONCEPTCHECK

```
reqs_check.cpp:13:10: error: cannot increment value of type 'foo'
        x++;
        ~^
/usr/local/include/boost/concept/usage.hpp:16:43: note: in instantiation of member f
    ~usage_requirements() { ((Model*)0)->~Model(); }
                                          ^
/usr/local/include/boost/concept/detail/general.hpp:38:42: note: in instantiation of
    static void failed() { ((Model*)0)->~Model(); }
                                         ^
reqs_check.cpp:11:5: note: in instantiation of member function 'boost::concepts::req
      ***********boost::concepts::usage_requirements<Incrementable<foo> >::********
    BOOST_CONCEPT_USAGE(Incrementable)
    ^
/usr/local/include/boost/concept/usage.hpp:29:7: note: expanded from macro 'BOOST_CO
      BOOST_CONCEPT_ASSERT((boost::concepts::usage_requirements<model>)); \
      ^
/usr/local/include/boost/concept/assert.hpp:43:5: note: expanded from macro 'BOOST_C
    BOOST_CONCEPT_ASSERT_FN(void(*)ModelInParens)
    ^
/usr/local/include/boost/concept/detail/general.hpp:78:51: note: expanded from macro
    &::boost::concepts::requirement_<ModelFnPtr>::failed>    \
                                                  ^
reqs_check.cpp:44:5: note: in instantiation of function template specialization 'twi
    twice(f);
    ^
reqs_check.cpp:14:9: error: cannot increment value of type 'foo'
        ++x;
        ^ ~
reqs_check.cpp:24:5: error: cannot increment value of type 'foo'
    ++x;
    ^ ~
reqs_check.cpp:31:5: note: in instantiation of function template specialization 'inc
    increment(x);
    ^
reqs_check.cpp:44:5: note: in instantiation of function template specialization 'twi
    twice(f);
    ^
```

LIMITATIONS OF BOOST.CONCEPTCHECK

- No overloading
- Doesn't reduce the number of errors

USING CONCEPTSLITE

```
template<class T>
concept bool Incrementable()
{
    return requires(T&& x)
    {
        x++;
        ++x;
    };
}
```

USING CONCEPTSLITE

```
template<class T>
void increment(T& x)
{
    ++x;
}

template<class T> requires Incrementable<T>()
void twice(T& x)
{
    increment(x);
    increment(x);
}
```

USING CONCEPTSLITE

```
req-concepts.cpp:37:12: error: cannot call function ‘void twice(T&) [with T = foo]’
     twice(f);
            ^
req-concepts.cpp:22:6: note:   constraints not satisfied
 void twice(T& x)
      ^
req-concepts.cpp:22:6: note:   concept ‘Incrementable<foo>()’ was not satisfied
```

USING TICK

- Create type traits to check type requirements
- Naming:
  - Concept: `DefaultConstructible`
  - Type trait: `std::is_default_constructible`

USING TICK

```
TICK_TRAIT(is_incrementable)
{
    template<class T>
    auto require(T&& x) -> valid<
        decltype(x++),
        decltype(++x)
    >;
};
```

USING TICK

```
template<class T>
void increment(T& x)
{
    ++x;
}

template<class T, TICK_REQUIRES(is_incrementable<T>())>
void twice(T& x)
{
    increment(x);
    increment(x);
}
```

USING TICK

```
reqs_tick.cpp:43:5: error: no matching function for call to 'twice'
    twice(f);
    ^~~~
reqs_tick.cpp:27:19: note: candidate template ignored: disabled by 'enable_if' [with
template<class T, TICK_REQUIRES(is_incrementable<T>())>
                  ^
```

REFINEMENTS

```
TICK_TRAIT(is_incrementable, std::is_default_constructible<_>)
{
    template<class T>
    auto require(T&& x) -> valid<
        decltype(x++),
        decltype(++x)
    >;
};
```

REFINEMENTS

```
TICK_TRAIT(is_equality_comparable,
    std::is_default_constructible<_1>,
    std::is_default_constructible<_2>)
{
    template<class T, class U>
    auto require(T&& x, U&& y) -> valid<
        decltype(x == y),
        decltype(x != y)
    >;
};
```

CHECK RETURNS

```
TICK_TRAIT(is_equality_comparable)
{
    template<class T, class U>
    auto require(T&& x, U&& y) -> valid<
        decltype(returns<bool>(x == y)),
        decltype(returns<bool>(x != y))
    >;
};
```

CHECK RETURNS

```
TICK_TRAIT(is_equality_comparable)
{
    template<class T, class U>
    auto require(T&& x, U&& y) -> valid<
        decltype(returns<std::is_fundamental<_>>(x == y)),
        decltype(returns<std::is_fundamental<_>>(x != y))
    >;
};
```

CHECKING FOR NESTED TYPES AND TEMPLATE

```
TICK_TRAIT(is_metafunction_class)
{
    template<class T>
    auto require(const T& x) -> valid<
        has_type<typename T::type>,
        has_template<T::template apply>
    >;
};
```

CHECKING REQUIREMENTS

```
template<class T>
void increment(T& x)
{
    ++x;
}

template<class T, TICK_REQUIRES(is_incrementable<T>())>
void twice(T& x)
{
    increment(x);
    increment(x);
}
```

CHECKING REQUIREMENTS

```
template<class T>
struct foo
{
    T x;

    TICK_MEMBER_REQUIRES(is_incrementable<T>())
    void up()
    {
        x++;
    }
};
```

CHECKING REQUIREMENTS

```
auto increment = [](auto& x, TICK_PARAM_REQUIRES(is_incrementable<decltype(x)>()))
{
    x++;
};
```

CHECKING REQUIREMENTS

```
auto increment = [](auto& x, TICK_PARAM_REQUIRES(trait<is_incrementable>(x)))
{
    x++;
};
```

CONCEPTS LITE VS TICK

- Interoperability with type traits

```
TICK_TRAIT(is_fusable, std::is_copy_constructible<_>)
{
    template<class T>
    auto require(T&& x) -> valid<
        decltype(returns<is_sequence<_>>(x.as_fusion_sequence()))
    >;
};
```

CONCEPTS LITE VS TICK

- Specialization: Types opt-in to a concept implicitly, specialization allows types to opt-out explicitly
  - Important with overloading
  - Unconstrained templates

CONCEPTS LITE VS TICK

- Dependent typing

```
template<class Tuple>
auto filter_numbers(const Tuple& t)
{
    return simple_filter(t, [](auto x)
    {
        return is_integral<decltype(x)>() or is_floating_point<decltype(x)>();
    });
}
```

CONCEPTS LITE VS TICK

- First class citizen

```
auto increment = [](auto& x, TICK_PARAM_REQUIRES(trait<is_incrementable>(x)))
{
    x++;
};
```

CONCEPTS LITE VS TICK

- Overloading
  - Subsuming vs tag dispatching

OVERLOADING

SIMPLE EXAMPLE OF ADVANCE

- `std::advance` advances an iterator several steps
  - Forward iterators are done in O(n)
  - Random access iterators are done in O(1)

TYPE REQUIREMENTS

```
template<class T>
concept bool Incrementable()
{
    return requires(T&& x)
    {
        { ++x } -> std::add_lvalue_reference_t<T>;
        { x++ } -> T;
    };
}
template<class T>
concept bool Decrementable()
{
    return Incrementable<T>() && requires(T&& x)
    {
        { --x } -> std::add_lvalue_reference_t<T>;
        { x-- } -> T;
    };
}
template<class T, class Number>
concept bool Advanceable()
{
    return Decrementable<T>() && requires(T&& x, Number n)
    {
        { x += n } -> std::add_lvalue_reference_t<T>;
    };
}
```

IMPLEMENTATION

```
template<class Iterator> requires Advanceable<Iterator, int>()
void advance(Iterator& it, int n)
{
    it += n;
}

template<class Iterator> requires Decrementable<Iterator>()
void advance(Iterator& it, int n)
{
    if (n > 0) while (n--) ++it;
    else
    {
        n *= -1;
        while (n--) --it;
    }
}

template<class Iterator> requires Incrementable<Iterator>()
void advance(Iterator& it, int n)
{
    while (n--) ++it;
}
```

USING TICK

```
TICK_TRAIT(is_incrementable)
{
    template<class T>
    auto require(T&& x) -> valid<
        decltype(returns<T>(x++)),
        decltype(returns<std::add_lvalue_reference_t<T>>(++x))
    >;
};
TICK_TRAIT(is_decrementable, is_incrementable<_>)
{
    template<class T>
    auto require(T&& x) -> valid<
        decltype(returns<T>(x--)),
        decltype(returns<std::add_lvalue_reference_t<T>>(--x))
    >;
};
TICK_TRAIT(is_advanceable, is_decrementable<_>)
{
    template<class T, class Number>
    auto require(T&& x, Number n) -> valid<
        decltype(returns<std::add_lvalue_reference_t<T>>(x += n))
    >;
};
```

USING TICK

```
template<class Iterator>
void advance_impl(Iterator& it, int n, tick::tag<is_advanceable>)
{
    it += n;
}

template<class Iterator>
void advance_impl(Iterator& it, int n, tick::tag<is_decrementable>)
{
    if (n > 0) while (n--) ++it;
    else
    {
        n *= -1;
        while (n--) --it;
    }
}

template<class Iterator>
void advance_impl(Iterator& it, int n, tick::tag<is_incrementable>)
{
    while (n--) ++it;
}

template<class Iterator, TICK_REQUIRES(is_incrementable<Iterator>())>
void advance(Iterator& it, int n)
{
    advance_impl(it, n, tick::most_refined<is_advanceable<Iterator, int>>());
}
```

INTRODUCING FIT

- C++ function utility library

FUNCTION OBJECTS

```
struct sum_f
{
    template<class T, class U>
    auto operator()(T x, U y) const
    {
        return x + y;
    }
};

FIT_STATIC_FUNCTION(sum) = sum_f();
```

```
auto three = sum(1, 2);
```

LAMBDAS

```
FIT_STATIC_LAMBDA_FUNCTION(sum) = [](auto x, auto y)
{
    return x + y;
};
```

ADAPTORS

- Decorate function with new capabilities

PIPABLE

```
auto three = 1 | sum(2);
```

```
FIT_STATIC_FUNCTION(sum) = fit::pipable(sum_f());
```

PIPABLE

```
FIT_STATIC_LAMBDA_FUNCTION(sum) = fit::pipable([](auto x, auto y)
{
    return x + y;
});
```

CONDITIONAL OVERLOADING

- Calls the first viable function in the overload set

CONDITIONAL OVERLOADING

```
template<class Iterator>
void advance(Iterator& it, int n) if (is_advanceable<Iterator, int>())
{
    it += n;
}
else if (is_decrementable<Iterator>())
{
    if (n > 0) while (n--) ++it;
    else
    {
        n *= -1;
        while (n--) --it;
    }
}
else if (is_incrementable<Iterator>())
{
    while (n--) ++it;
}
```

CONDITIONAL OVERLOADING

```
FIT_STATIC_LAMBDA_FUNCTION(advance) = fit::conditional(
    [](auto& it, int n, TICK_PARAM_REQUIRES(tick::trait<is_advanceable>(it, n)))
    {
        it += n;
    },
    [](auto& it, int n, TICK_PARAM_REQUIRES(tick::trait<is_decrementable>(it)))
    {
        if (n > 0) while (n--) ++it;
        else
        {
            n *= -1;
            while (n--) --it;
        }
    },
    [](auto& it, int n, TICK_PARAM_REQUIRES(tick::trait<is_incrementable>(it)))
    {
        while (n--) ++it;
    }
);
```

RECURSIVE PRINT

- A generic print function to recursively output values from:
  - Ranges
  - Fusion sequences
  - Variant
  - Streamable

TYPE REQUIREMENTS

```
template<class Stream, class T>
concept bool Streamable()
{
    return requires(Stream&& s, T&& x)
    {
        s << x;
    };
}
template<class T>
concept bool Iterator()
{
    return std::is_copy_constructible<T>::value &&
        std::is_copy_assignable<T>::value &&
        std::is_destructible<T>::value &&
        requires(T x)
        {
            *x;
            { ++x } -> T&;
        };
}
template<class T>
concept bool Range()
{
    return requires(T&& x)
    {
        { adl::adl_begin(x) } -> Iterator;
        { adl::adl_end(x) } -> Iterator;
    };
}
```

CONCEPTS LITE

```
void print(const std::string& x)
{
    std::cout << x << std::endl;
}

template<class R>
requires Range<R>()
void print(const R& r);

template<class Sequence>
requires boost::fusion::traits::is_sequence<Sequence>::value and
        not Range<Sequence>()
void print(const Sequence& s);

template<class... Ts>
void print(const boost::variant<Ts...>& v);

template<class T>
requires Streamable<std::ostream, T>() and
        not boost::fusion::traits::is_sequence<T>::value and
        not Range<T>()
void print(const T& x);


template<class R>
requires Range<R>()
void print(const R& r)
{
    for(const auto& x:r) print(x);
}

template<class Sequence>
requires boost::fusion::traits::is_sequence<Sequence>::value and
        not Range<Sequence>()
void print(const Sequence& s)
{
    boost::fusion::for_each(s, [](const auto& x)
    {
        print(x);
    });
}

template<class... Ts>
void print(const boost::variant<Ts...>& v)
{
    boost::apply_visitor(fit::result<void>([](const auto& x)
    {
        print(x);
    }), v);
}
```

```
template<class T> requires Streamable<std::ostream, T>() and
        not boost::fusion::traits::is_sequence<T>::value and
        not Range<T>()
void print(const T& x)
{
    std::cout << x << std::endl;
}
```

USING TICK

```
TICK_TRAIT(is_streamable)
{
    template<class Stream, class T>
    auto require(Stream&& s, T&& x) -> valid<
        decltype(s << x)
    >;
};

TICK_TRAIT(is_iterator,
    std::is_copy_constructible<_>,
    std::is_copy_assignable<_>,
    std::is_destructible<_>)
{
    template<class T>
    auto require(T x) -> valid<
        decltype(*x),
        decltype(returns<T&>(++x))
    >;
};

TICK_TRAIT(is_range)
{
    template<class T>
    auto require(T&& x) -> valid<
        decltype(returns<is_iterator<_>>(adl::adl_begin(x))),
        decltype(returns<is_iterator<_>>(adl::adl_end(x)))
    >;
};
```

USING FIT

```
FIT_STATIC_LAMBDA_FUNCTION(print) = fit::fix(fit::conditional(
    [](auto, const std::string& x)
    {
        std::cout << x << std::endl;
    },
    [](auto self, const auto& range,
        TICK_PARAM_REQUIRES(trait<is_range>(range)))
    {
        for(const auto& x:range) self(x);
    },
    [](auto self, const auto& sequence,
        TICK_PARAM_REQUIRES(trait<boost::fusion::traits::is_sequence>(sequence)))
    {
        boost::fusion::for_each(sequence, self);
    },
    [](auto self, const auto& variant,
        TICK_PARAM_REQUIRES(trait<is_variant>(variant)))
    {
        boost::apply_visitor(fit::result<void>(self), variant);
    },
    [](auto, const auto& x,
        TICK_PARAM_REQUIRES(trait<is_streamable>(std::cout, x)))
    {
        std::cout << x << std::endl;
    }
));
```

EMBEDDED TYPE REQUIREMENTS

```
FIT_STATIC_LAMBDA_FUNCTION(find_iterator) = fit::conditional(
    [](const auto& r, const auto& x) -> decltype(find(r, x))
    {
        return find(r, x);
    },
    [](const std::string& s, const auto& x)
    {
        auto index = s.find(x);
        if (index == std::string::npos) return s.end();
        else return s.begin() + index;
    },
    [](const auto& r, const auto& x) -> decltype(r.find(x))
    {
        return r.find(x);
    },
    [](const auto& r, const auto& x)
    {
        using std::begin;
        using std::end;
        return std::find(begin(r), end(r), x);
    }
);
```

ERRORS

ERROR FROM ADVANCE

```
overloading-1.cpp:227:5: error: no matching function for call to object of type 'con
      overloading-1.cpp:183:5>, <lambda at overloading-1.cpp:192:5> > >'
    advance(foo(), 1);
    ^~~~~~~
../../../github/Fit/fit/function.h:68:10: note: candidate template ignored: substitu
      fit::conditional_adaptor<<lambda at overloading-1.cpp:179:5>, <lambda at overl
    auto operator()(Ts&&... xs) const FIT_RETURNS
         ^
```

USING REVEAL ADAPTOR

```
overloading-1.cpp:227:5: error: no matching function for call to object of type 'con
      overloading-1.cpp:179:5>, <lambda at overloading-1.cpp:183:5>, <lambda at over
    advance(foo(), 1);
    ^~~~~~~
../../../github/Fit/fit/reveal.h:117:20: note: candidate template ignored: substitut
      '<lambda at overloading-1.cpp:179:5>'
    constexpr auto operator()(Ts&&... xs) ->
                   ^
../../../github/Fit/fit/reveal.h:117:20: note: candidate template ignored: substitut
      '<lambda at overloading-1.cpp:183:5>'
    constexpr auto operator()(Ts&&... xs) ->
                   ^
../../../github/Fit/fit/reveal.h:117:20: note: candidate template ignored: substitut
      '<lambda at overloading-1.cpp:192:5>'
    constexpr auto operator()(Ts&&... xs) ->
                   ^
../../../github/Fit/fit/function.h:67:10: note: candidate template ignored: substitu
      fit::conditional_adaptor<<lambda at overloading-1.cpp:179:5>, <lambda at overl
    auto operator()(Ts&&... xs) const FIT_RETURNS
         ^
```

IMPROVEMENTS FROM CLANG

```
overloading-1.cpp:227:5: error: no matching function for call to object of type 'con
      (lambda at overloading-1.cpp:183:5), (lambda at overloading-1.cpp:192:5)> > >'
    advance(foo(), 1);
    ^~~~~~
overloading-1.cpp:179:25: note: candidate template ignored: disabled by 'enable_if'
    [](auto& it, int n, TICK_PARAM_REQUIRES(tick::trait<is_advanceable>(it, n)))
                        ^
/home/paul/github/Tick/tick/requires.h:62:5: note: expanded from macro 'TICK_PARAM_R
    (tick::detail::param_extract<decltype(__VA_ARGS__)>::value), \
    ^
overloading-1.cpp:183:25: note: candidate template ignored: disabled by 'enable_if'
    [](auto& it, int n, TICK_PARAM_REQUIRES(tick::trait<is_decrementable>(it)))
                        ^
/home/paul/github/Tick/tick/requires.h:62:5: note: expanded from macro 'TICK_PARAM_R
    (tick::detail::param_extract<decltype(__VA_ARGS__)>::value), \
    ^
overloading-1.cpp:192:25: note: candidate template ignored: disabled by 'enable_if'
    [](auto& it, int n, TICK_PARAM_REQUIRES(tick::trait<is_incrementable>(it)))
                        ^
/home/paul/github/Tick/tick/requires.h:62:5: note: expanded from macro 'TICK_PARAM_R
    (tick::detail::param_extract<decltype(__VA_ARGS__)>::value), \
    ^
overloading-1.cpp:192:25: note: candidate template ignored: disabled by 'enable_if'
/home/paul/github/Tick/tick/requires.h:62:5: note: expanded from macro 'TICK_PARAM_R
    (tick::detail::param_extract<decltype(__VA_ARGS__)>::value), \
    ^
```

GETTING MORE DETAIL

- Why did my class not meet the type requirements?

```
TICK_TRAIT_CHECK(is_advanceable<foo>);
```

GETTING MORE DETAIL

- Lack of backtrace for substitution failures
- Use a macro to define additional version that is not in a non-deduced context
- Compiler report back more information
  - Build a tree where each node is substitution failure from overload resolution
  - Report back only the leafs in the tree

MAKING THE COMPILER EVEN SMARTER

- Parse the boolean expression in `enable_if`
- For each trait that is false report back the "leaf" failures for each specialization tried

LANGUAGE FEATURE

- No macros
- Multi-phase checking
  - Analyze template definitions to ensure it matches the type requirements
  - Template-based type requirements(such as checking for template member functions)
- Concept mapping

LIBRARY SUPPORT

- Supported and tested on clang 3.4-3.7, gcc 4.6-4.9, and Visual Studio 2015:
- https://github.com/pfultz2/Tick
- https://github.com/pfultz2/Fit

QUESTIONS