Like in Haskell:
Final Tagless and eDSL on concepts

Alexander Granin
graninas@gmail.com, Twitter: @graninas

C++ Russia 2020 online

struct Presentation

{

Haskell & C++: What they have in common?

Intro to concepts

Final Tagless eDSLs on concepts

Advanced eDSL design with concepts

Testing

Functional concepts (pun intended)

Conclusion

};

Alexander Granin

Passionate Developer

Haskell Consultant

Software Architect

Experienced Developer

Haskell

PureScript

C++

C#

Python

Alexander Granin

Passionate Developer

Haskell Consultant

Software Architect

Experienced Developer

Haskell

PureScript

C++

C#

Python

Speaker

C++ Russia

C++ Siberia

f(by)

Functional Conf

FPure

FPConf

FProg Spb

LambdaNsk

Dev2Dev

Alexander Granin

Passionate Developer

Haskell Consultant

Software Architect

Experienced Developer

Haskell

PureScript

C++

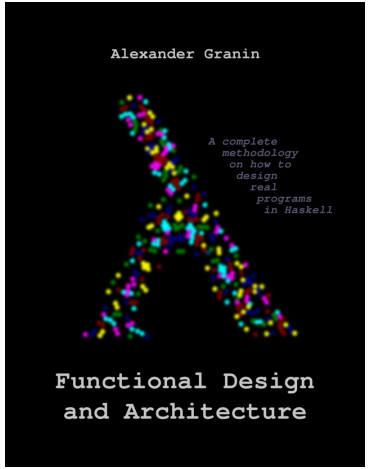
C#

Python

Author

The Book

“Functional Design and Architecture”



Speaker

C++ Russia

C++ Siberia

f(by)

Functional Conf

FPure

FPConf

FProg Spb

LambdaNsk

Dev2Dev

Alexander Granin

Haskell & C++:
What they have in common?

The Present and the Future of Functional Programming in C++

C++ Siberia 2019, Keynote

Monadic Parsers in C++

C++ Russia 2019

Functional Approach to Software Transactional Memory in C++

C++ Russia 2018

Functional “Life”: Parallel Cellular Automata and Comonads in C++

C++ Russia 2016

Functional Microscope: Lenses in C++

C++ Siberia 2015

Functional and Declarative Design in C++

C++ User Group 2014

	C++	Haskell	Scala	C#	Java
Real purity	templates, constexpr	Non-IO code	not really	?	?
IO constraint	-	inherent, IO monad	external, ZIO, scalaz, cats	?	?
Functional pipelining	ranges	composition, monads	composition, monads	Linq	Streams
Type constraints	concepts	type classes	traits, type classes*	?	?
Algebraic Data Types	std::variant, struct	ADTs	objects, case classes	?	?
Pattern matching	weak	excellent	excellent	good	?
Monads support	-	do-notation	for-comprehension	-	-

Intro to concepts

Talks on concepts

Intro to C++20's Concepts

Hendrik Niemeyer, Cpp Usergroup Dresden, 2020

Concepts in 60: Everything you need to know and nothing you don't

Andrew Sutton, CppCon, 2018

Concepts in C++ (-fconcepts) vs type classes in Haskell

Pavel Filonov, ruHaskell, 2018

Move only C++ design

Ivan Čukić, C++ Russia, 2019

Concepts are compile-time predicates on templates.
Concepts may express requirements (constraints) on different things.

- Require a method to have a particular signature
- Require a type to be defined
- Require a type to be convertible to something
- Require a constexpr expression to be true
- Require a template to be valid
- Require a logical clause on constraints (and, or)
- Require a code to be compilable

Concepts support in compilers

```
template<typename T>
concept Hashable = requires(T x) {
    { std::hash<T>{}(x) } -> std::convertible_to<std::size_t>;
};
```

Standard concepts (“Concepts Lite”) P0734R0	gcc 10* -std=c++2a	clang 10** -std=c++20	VS 2019 16.3 /std:c++latest /std:c++20
Old concepts N4377	gcc 6+ -fconcepts	?	?

Standard concepts library

Core language concepts

```
std::same_as<A, B>
std::convertible_to<From, To>
std::integral<T>
std::signed_integral<T>
std::floating_point<T>
std::unsigned_integral<T>
```

Comparison concepts

```
std::equality_comparable<T>
```

Object concepts

```
std::movable<T>
std::copyable<T>
```

Algorithm concepts

```
std::sortable<T>
std::mergeable<T>
std::permutable<T>
```

Range concepts

```
std::ranges::range<T>
std::ranges::view<T>
```

Custom concept

```
template<typename T>
concept Addable = requires (T x) { x + x; };
```

Custom concept

```
template<typename T>
concept Addable = requires (T x) { x + x; };

template<typename T> requires Addable<T>
T add(T a, T b) {
    return a + b;
}
```

Custom concept

```
template<typename T>
concept Addable = requires (T x) { x + x; };

template<typename T> requires Addable<T>
T add(T a, T b) {
    return a + b;
}

struct Point {
    int x, y;
    Point operator+(Point lhs, const Point& rhs) {
        lhs.x += rhs.x;
        lhs.y += rhs.y;
        return lhs;
    }
};

static_assert (Addable<Point>);
```

Final Tagless* eDSLs with concepts

** Final Tagless == final encoding without runtime tags*

eDSL definition: Logger interface

```haskell
class Logger m where
    logMessage :: String -> m ()
```

eDSL definition: Logger interface

```haskell
class Logger m where
    logMessage :: String -> m ()
```

(object-based)

```cpp
template<class M>
concept Logger = requires(M m, string msg) {
   { m.log_message(msg) } -> std::same_as<void>;
};
```

eDSL definition: Logger interface

```haskell
class Logger m where
    logMessage :: String -> m ()
```

(object-based)

```cpp
template<class M>
concept Logger = requires(M m, string msg) {
   { m.log_message(msg) } -> std::same_as<void>;
};
```

(function-based)

```cpp
template<class M>
concept Logger = requires(M m, string msg) {
   { log_message(m, msg) } -> std::same_as<void>;
};
```

eDSL definition: Random interface

```haskell
class Integral t => Random where
    getRandomInt :: t -> t -> m ()
```

(object-based)

```cpp
template<class M, typename T>
concept Random =
  std::integral<T>
  && requires(M m, T from, T to) {

  { m.get_random_int(from, to) } -> std::same_as<T>;

};
```

Integral type class & integral concept

```haskell
class (Real a, Enum a) => Integral a
where
  quot      :: a -> a -> a
  rem       :: a -> a -> a
  div       :: a -> a -> a
  quotRem   :: a -> a -> (a, a)
  divMod    :: a -> a -> (a, a)
  toInteger :: a -> Integer
```

Integral numbers, supporting integer division.

```cpp
template<class T>
concept integral = std::is_integral_v<T>;

template<class T> struct is_integral;
```

true, if T is an integral type type.

bool, char, char8_t, char16_t, char32_t, wchar_t, short, int, long, long long, including any signed, unsigned, and cv-qualified variants.

eDSL usage: “Generate and say” scenario

```
generateAndSay :: (Logger m, Random m Int) => m ()
generateAndSay = do
    die <- getRandomInt 1 6
    logMessage ("You got: " ++ show die)
```

(object-based)

```
template <typename M>
void generate_and_say(M& m)
   requires Logger<M> && Random<M, int> {

   int die = m.get_random_int(1, 6);
   m.log_message("You got: " + to_string(die));

}
```

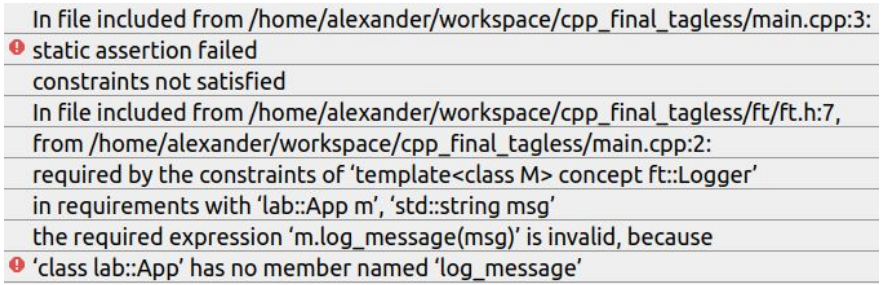
eDSL implementation

```
struct App {
    void log_message(const string& msg) const {
        cout << msg;
    }

    int get_random_int(int from, int to) const {
        return std::experimental::randint(from, to);
    }
};


static_assert (Logger<App>);
static_assert (Random<App, int>);
```

Constraints violation error



Static assertion failed

constraints not satisfied

required by the constraints of template<class M> concept ft::Logger

in requirements with ‘lab::App m’, ‘std::string msg’

the required expression ‘m.log_message(msg)’ is invalid, because

‘class lab::App’ has no member named ‘log_message’

Running the scenario

```
template <typename M>
void generate_and_say(M& m)
   requires Logger<M> && Random<M, int> {

   int die = m.get_random_int(1, 6);
   m.log_message("You got: " + to_string(die));
}


int main() {
   App app;
   generate_and_say<App>(app);          // You got: 2
   generate_and_say<App>(app);          // You got: 5
}
```

Advanced eDSL design with concepts

Key-value database raw interface

```
enum class DBError {
   ConnectionError
};

template <typename V>
using DBResult = expected<DBError, V>;
```

Key-value database raw interface

```
enum class DBError {
   ConnectionError
};

template <typename V>
using DBResult = expected<DBError, V>;


template<class M>
concept KV_RawWrite = requires(M m, string k, string v) {
   { m.raw_write(k, v) } -> std::same_as<DBResult<Unit>>;
};

template<class M>
concept KV_RawRead = requires(M m, string k) {
   { m.raw_read(k) } -> std::same_as<DBResult<optional<string>>>;
};
```

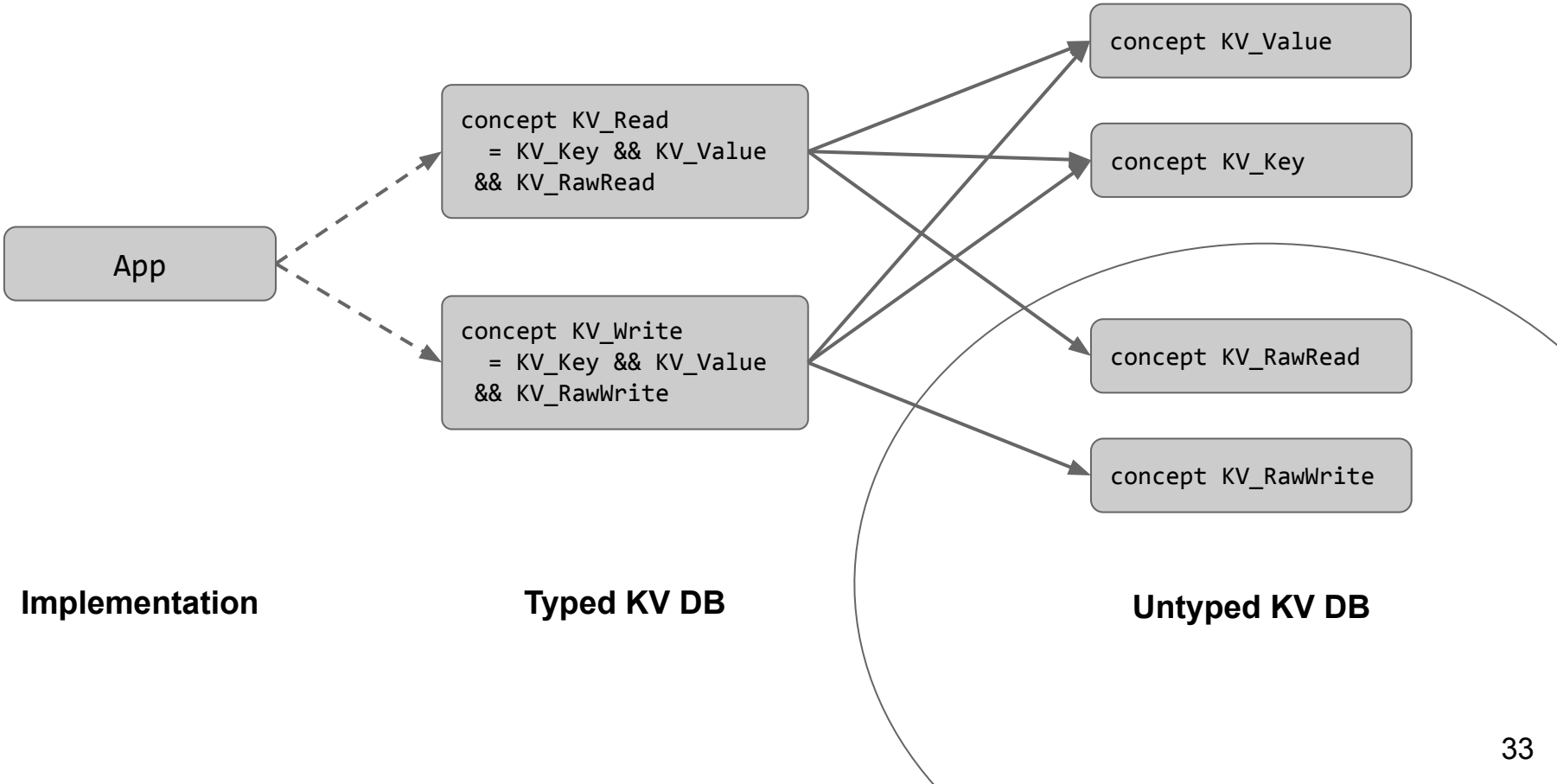
Key-value database raw implementation

```
struct App {
    DBResult<Unit> raw_write(const string& k, const string& v) {
        auto redis = Redis("tcp://127.0.0.1:6379");       // TODO: try block
        redis.set(k, v)
        return Unit{};
    }

    DBResult<optional<string>> raw_read(const string& k) {
        auto redis = Redis("tcp://127.0.0.1:6379");       // TODO: try block
        optional<string> val = redis.get(k);
        return val;
     }
}
```


String -> String -> String -> String
Types are a lie

Typed key-value database interface



Encoding and decoding keys and values

```
template<class M, typename K>
concept KV_Key = requires(M, K k) {
   { M::encode_key(k) } -> std::same_as<string>;
};


template<class M, typename V>
concept KV_Value = requires(M, V v, string vStr) {
   { M::encode_value(v)    } -> std::same_as<string>;
   { M::decode_value(vStr) } -> std::same_as<optional<V>>;
};
```

Typed key-value database interface

```
template<class M, typename K, typename V>
DBResult<Unit> kv_write(M& m, const K& k, const V& v) {
   return m.raw_write(M::encode_key(k), M::encode_value(v));
}


template<class M, typename K, typename V>
concept KV_Write =
   KV_Key<M, K> && KV_Value<M, V>
   && KV_RawWrite<M>
   && requires(M m, K k, V v) {

   { kv_write<M, K, V>(m, k, v) } -> std::same_as<DBResult<Unit>>;

};
```

Typed key-value database implementation

```
struct App {
   DBResult<Unit> raw_write(const string& k, const string& v) {..}
   DBResult<optional<string>> raw_read(const string& k) {..}

   static string encode_key(int k) {..}
   static string encode_value(???) {..}
   static optional<???> decode_value(const string& raw_val) {..}
};
```

Typed key-value database implementation

```
struct GameInfo {
   Point playerPos;
};

struct App {
   DBResult<Unit> raw_write(const string& k, const string& v) {..}
   DBResult<optional<string>> raw_read(const string& k) {..}

   static string encode_key(int k) {..}
   static string encode_value(GameInfo gi) {..}
   static optional<GameInfo> decode_value(const string& raw_val) {..}
};
```

Typed key-value database implementation

```
struct GameInfo {
   Point playerPos;
};

struct App {
   DBResult<Unit> raw_write(const string& k, const string& v) {..}
   DBResult<optional<string>> raw_read(const string& k) {..}

   static string encode_key(int k) {..}
   static string encode_value(GameInfo gi) {..}
   static optional<GameInfo> decode_value(const string& raw_val) {..}
};

static_assert (KV_Key<App, int>);
static_assert (KV_Value<App, GameInfo>);
static_assert (KV_Write<App, int, GameInfo>);
static_assert (KV_Read<App, int, GameInfo>);
```

Testing

Test runtime

```
struct TestRuntime {
   map<string, string> kvdb_stub;
   list<string> logs;
   list<int> random_gen_mocks;
};
```

Test implementation

```
struct TestApp {
   TestRuntime runtime;

   void log_message(const string& msg) { runtime.logs.push_back(msg); }
```

Test implementation

```
struct TestApp {
    TestRuntime runtime;

    void log_message(const string& msg) { runtime.logs.push_back(msg); }

    DBResult<Unit> raw_write(const string& k, const string& v) {
        runtime.kvdb_stub[k] = v;
        return Unit{};
    }
;
```

Test implementation

```
struct TestApp {
    TestRuntime runtime;

    void log_message(const string& msg) { runtime.logs.push_back(msg); }

    DBResult<Unit> raw_write(const string& k, const string& v) {
        runtime.kvdb_stub[k] = v;
        return Unit{};
    }

    DBResult<optional<string>> raw_read(const string& k) {
        if (runtime.kvdb_stub.contains(k))
            return optional<string>(runtime.kvdb_stub[k]);
        return Unit{};
    }
```

Test implementation

```
struct TestApp {
   TestRuntime runtime;

   void log_message(const string& msg) { runtime.logs.push_back(msg); }

   DBResult<Unit> raw_write(const string& k, const string& v) {
       runtime.kvdb_stub[k] = v;
       return Unit{};
   }

   DBResult<optional<string>> raw_read(const string& k) {
       if (runtime.kvdb_stub.contains(k))
           return optional<string>(runtime.kvdb_stub[k]);
       return Unit{};
   }

   void get_random_int(int, int) { return get_next_mock(runtime.random_gen_mocks); }

};
```

Test evaluation

```
// arrange

TestRuntime testRt;
TestApp app { testRt };

testRt.random_gen_mocks = {3, 5};

// act

generate_and_say<App>(app);
generate_and_say<App>(app);

// assert

assert_eq(testRt.logs[0], "You got: 3");
assert_eq(testRt.logs[1], "You got: 5");
assert_eq(testRt.random_gen_mocks.empty(), true);
assert_eq(testRt.kvdb_stub.empty(), true);
```

```
template <typename M>
void generate_and_say(M& m)
   requires Logger<M> && Random<M, int> {
   int die = m.get_random_int(1, 6);
   m.log_message("You got: " + to_string(die));
}
```

Application architecture

Interfaces
(concepts)

Random | KV DB | State

Logger | Other effects | IO

Scenarios
(template functions)

generateAndSay

doSomethingElse

Implementation
(classes, objects)

App | TestApp

Application architecture (“3-layer cake”)

Interfaces

(~~concepts~~ type classes)

Random | KV DB | State

Logger | Other effects | IO

Scenarios

(~~template~~ monadic functions)

generateAndSay

doSomethingElse

Implementation

(~~classes, objects~~ custom monad)

App

TestApp

Functional concepts (pun intended)

Boost::hana

Description:

Hana is a header-only library for C++ metaprogramming suited for computations on both types and values.

In reality:

Hana is a header-only library for functional programming using core functional concepts derived from Haskell.

Functor

```
fmap :: (A -> B) -> F<A> -> F<B>
```

Boost::hana::Functor

```
fmap :: (A -> B) -> F<A> -> F<B>
```

```
template <typename F>
 struct Functor
    : hana::integral_constant<bool,
         !is_default<transform_impl<typename tag_of<F>::type>>::value ||
         !is_default<adjust_if_impl<typename tag_of<F>::type>>::value
    >
{ };
```

Concept-based Functor

```
fmap :: (A -> B) -> F<A> -> F<B>
```

```
template <typename F>
 struct Functor
    : hana::integral_constant<bool,
         !is_default<transform_impl<typename tag_of<F>::type>>::value ||
         !is_default<adjust_if_impl<typename tag_of<F>::type>>::value
    >
{ };
```

```
template <template<class> class F, typename A, typename B>
concept Functor = requires(F<A> ma, std::function<B(A)> f) {
   { fmap(f, ma) } -> std::same_as<F<B>>;
};
```

Functor sample: optional

```
struct Nothing { };

template <typename T>
using optional = std::variant<Nothing, T>;
```

Functor sample: optional

```
struct Nothing { };

template <typename T>
using optional = std::variant<Nothing, T>;


template <typename A, typename B>
optional<B> fmap(const std::function<B(A)>& f, const optional<A>& ma) {
   if (ma.index() == 0) {
       return optional<B>(Nothing{});
   }
   return optional<B>(f(std::get<1>(ma)));
}
```

Checking for optional to be a Functor

```
template <typename A, typename B>
struct FOptHelper {
   static_assert(Functor<optional, A, B>);
};


FOptHelper<int, int> intOptHelper;
```

Optional usage

```
optional<int>    maybeInt = optional(10);
```

Optional usage

```
optional<int>    maybeInt = optional(10);
optional<string> maybeStr = fmap(intToStr, maybeInt);
```

```
function<string(int)> intToStr
        = [](int val) { return to_string(val); };
```

Optional usage

```
optional<int>    maybeInt = optional(10);
optional<string> maybeStr = fmap(intToStr, maybeInt);
optional<size_t> maybeLen = fmap(strLen, maybeStr);
```

```
function<string(int)> intToStr
        = [](int val) { return to_string(val); };

function<size_t(string)> strLen
        = [](const string& s) { return s.length(); };
```

Optional usage

```
optional<int>    maybeInt = optional(10);
optional<string> maybeStr = fmap(intToStr, maybeInt);
optional<size_t> maybeLen = fmap(strLen, maybeStr);


if (maybeLen.index() == 0) {
    cout << "Chain is broken.";
}
else {
    cout << "Digits count: " << std::get<1>(maybeLen);
}


function<string(int)> intToStr
        = [](int val) { return to_string(val); };

function<size_t(string)> strLen
        = [](const string& s) { return s.length(); };
```

Optional usage

```
In file included from /home/alexander/workspace/cpp_final_tagless/ft/prelude.h:9,
                 from /home/alexander/workspace/cpp_final_tagless/ft/ft.h:4,
                 from /home/alexander/workspace/cpp_final_tagless/main.cpp:2:
/usr/include/c++/10/variant: In substitution of ‘template<class ... _Types> template<class _Tp, class> using
__accepted_type = std::variant<_Types>::__to_type<__accepted_index<_Tp> > [with _Tp = _Tp&&;
<template-parameter-2-2> = typename std::enable_if<std::variant<ft::Unit, T>::__not_self<_Tp&&>,
void>::type; _Types = {ft::Unit, T}]’:
/home/alexander/workspace/cpp_final_tagless/main.cpp:29:49:   required from here
/usr/include/c++/10/variant:1332:36: internal compiler error: Segmentation fault
1332 |  using __accepted_type = __to_type<__accepted_index<_Tp>>;
     |                          ^~~~~~~~~~~~~~~~~~~~
Please submit a full bug report,
with preprocessed source if appropriate.
See <file:///usr/share/doc/gcc-10/README.Bugs> for instructions.
```

Foldable

```
foldr :: (A -> B -> B) -> B -> F A -> B
```

Boost::hana::Foldable

```
foldr :: (A -> B -> B) -> B -> F A -> B
```

```
template <typename T>
struct Foldable
    : hana::integral_constant<bool,
        !is_default<fold_left_impl<typename tag_of<T>::type>>::value ||
        !is_default<unpack_impl<typename tag_of<T>::type>>::value
    >
{ };
```

Concept-based Foldable

```
foldr :: (A -> B -> B) -> B -> F A -> B
```

```
template <typename T>
struct Foldable
    : hana::integral_constant<bool,
        !is_default<fold_left_impl<typename tag_of<T>::type>>::value ||
        !is_default<unpack_impl<typename tag_of<T>::type>>::value
    >
{ };
```

```
template <template<class> class F, typename A, typename B>
concept Foldable = requires(F<A> ma, std::function<B(A, B)> f, B val0) {
   { foldr(f, val0, ma) } -> std::same_as<B>;
};
```

Foldable vector usage

```
template <typename A, typename B>
B foldr(const std::function<B(A, B)>& f, B b, const vector<A>& v) {
    for (int val : v) {
        b = f(val, b);
    }
    return b;
}
```

Foldable vector usage

```
template <typename A, typename B>
B foldr(const std::function<B(A, B)>& f, B b, const vector<A>& v) {
   for (int val : v) {
       b = f(val, b);
   }
   return b;
}

function<int(int, int)> add = [](int a, int b) { return a + b; };
function<int(int, int)> mul = [](int a, int b) { return a * b; };

vector<int> ints = {1, 2, 3, 4};

int sum     = foldr(add, 0, ints);
int product = foldr(mul, 1, ints);

cout << endl << sum;          // 10
cout << endl << product;      // 24
```

Conclusion

- Concepts resemble the same idea as type classes in Haskell
- More powerful metaprogramming
- Great flexibility
- Better developer experience
- Better error messages
- Concepts can replace a lot of template boilerplate
- Concepts will change metaprogramming in C++ forever
- The practices are not well-understood yet
- Learn Haskell, and you'll get more insights on Modern C++!

GitHub List: Functional Programming in C++

https://github.com/graninas/cpp_functional_programming

Books	Papers	QA	Showcase projects
Articles	Talks	Libraries	And more...

C++ and Haskell: friends forever!

Alexander Granin

graninas@gmail.com

graninas

graninas_channel

StrangeMooder

granin.alexander