hop: a language to design function-overload-sets

Tobias Loew

come and hop with me!

C++ now
2021
MAY 2-7
Aspen, Colorado, USA

outline

- prelude: homogeneous variadic functions
  - Blitz++ & a series of blogs on Fluent-C++
  - historic view / best practices and limitations
  - overload-resolution
  - what about concepts?
  - a library for homogeneous variadic functions
- hop: function-parameter building-blocks (started as: homogeneous parameters)
  - a grammar for function-parameters
  - defining overload sets
  - down into the rabbit hole: hop-examples and internals

homogeneous variadic functions (HVFs): what are we talking about

```
void logger("This", "is", "a", "log", "message");
double max(0.0, d1, d2, d3, d4, d5);
Array<double, 1> vec(1000);
Array<double, 2> matrix(6, 100);
Array<double, 3> cube(2, 3, 5);
```

- function with an arbitrary (non-zero) number of arguments, all of the **same type**

```
R f(T);   R f(T, T);   R f(T, T, T);   ...
```

- should also work for overloads of HVFs
  especially needed for constructors and function call `operator()`

Blitz++ & a series of blogs on Fluent-C++

- Blitz++: lin. algebra library, pioneering in expression templates, C++98
- Blitz++ `Array` class-template
  - 48 constructors, 85 `operator()` overloads -> essentially different:
    - 13 constructors
    - 3 `operator()` and 8 `operator() const`
    - several other homogenous variadic functions from 1 - 10 arguments
  - a lot of similar code for 1 to 10 homogenous args in whole library
- Fluent-C++: "How to Define a Variadic Number of Arguments of the Same Type" three parts published 01-02/2019
  - different approaches to HVFs: pros / cons

history of homogeneous variadic functions in C++

C++98: write overloads manually

```
R foo(A a1);
R foo(A a1, A a2);
R foo(A a1, A a2, A a3);
R foo(A a1, A a2, A a3, A a4);
R foo(A a1, A a2, A a3, A a4, A a5);
```

- pros
  - explicit: easy to write and understand
  - beginner friendly, no templates or macros required
- cons
  - only up to a given number of arguments
  - anti-pattern: archetype of DRY

C++98: Boost.Preprocessor

```
#define NUMBER_OF_ARGUMENTS_OF_F    10
#define GENERATE_OVERLOAD_OF_F(Z, N, _)           \
    R foo(BOOST_PP_ENUM_PARAMS(N, A a)) { /* ... */ }

BOOST_PP_REPEAT_FROM_TO(1, BOOST_PP_INC(NUMBER_OF_ARGUMENTS_OF_F),
    GENERATE_OVERLOAD_OF_F, _)
```

- pros
  - no repetition of code
  - number of arguments can easily be adjusted
- cons
  - declaration/definition is hidden in macros
  - hard to write, even harder to read, not to mention debugging
  - number of arguments is fixed a priori and limited by Boost.Preprocessor

C++11: std::initializer_list

```
R foo(std::initializer_list<A> as) {
    /*...*/
}
```

- pros
  - works for any number of arguments
- cons
  - requires additional braces when called: `{a1, ... , an}`
  - inflexible wrt. mutability
  - may also be called with empty list
  - just one argument (the `initializer_list`)

C++11: parameter pack

```
template <class T, class... Ts>
R foo(T t, Ts... ts) {
    /* ... */
}
```

- pros
  - works for any number of arguments
- cons
  - matches any input
  - invalid input causes
    - compilation errors: bad
    - logical errors: worse
    - runtime errors: havoc

C++11: parameter pack + SFINAE (in shorter C++14 style)

```
template <class... Ts,
    enable_if_t<
        (sizeof...(Ts) > 0) &&                  // at least one argument
        (std::is_convertible_v<Ts, A> && ...)  // all arguments convertible to A
    >* = nullptr >
R foo(Ts&&... ts) {
    /*...*/
}
```

- pros
  - works for any number of arguments
  - matches only valid input
- cons
  - none

C++20: parameter pack + requires

```
template <class... Ts>
    requires
        (sizeof...(Ts) > 0) &&                  // at least one argument
        (std::is_convertible_v<Ts, A> && ...)  // all arguments convertible to A
    R foo(Ts&&... ts) {
    /*...*/
}
```

- pros
  - works for any number of arguments
  - matches only valid input
- cons
  - none

the future of C++

P1219R2: homogeneous variadic function parameters

```
R foo(A a, A... as){ /* ... */ }
```

- pros
  - native language support for HVFs
  - matches only valid input
- cons
  - syntax conflicts with `varargs`: `,` before trailing `...` is optional, i.e. `R foo(int, ...)` is equivalent to `R foo(int...)`
  - current status of P1219R2: dead (Belfast 2019)

HVFs: current best practice

- parameter pack + SFINAE
- parameter pack + requires
- what about overload resolution and implicit conversion?

overload-resolution and implicit conversion

simple HVFs for any number of `int` and `float`

```
template <class... Ts>
    requires(is_convertible_v<Ts, int> && ...)
R foo(Ts&&... ts) { /* ... */ }
```

```
template <class... Ts>
    requires(is_convertible_v<Ts, float> && ...)
R foo(Ts&&... ts) { /* ... */ }
```

assuming both overloads of `foo` are visible, which one is called?

```
foo(1, 2, 3);     // (a)
foo(0.5f, -2.4f); // (b)
foo(1.5f, 3);     // (c)
```

this poll is presented to you by the hop-experts

Luna

Rolf

my naïve expectation

- ```
  foo(1, 2, 3);     // (a) calls int-overload
  ```
- ```
  foo(0.5f, -2.4f); // (b) calls float-overload
  ```
- ```
  foo(1.5f, 3);     // (c) error: ambiguous
  ```
- let's ask the compiler overload test @ godbolt.org/z/YbbY5ah7d
  ```

what just happened?
why are all 3 calls ambiguous?

analyzing the ambiguity

```cpp
template <class... Ts>
    requires(is_convertible_v<Ts, int> && ...)
R foo(Ts&&... ts) { /* ... */ }

template <class... Ts>
    requires(is_convertible_v<Ts, float> && ...)
R foo(Ts&&... ts) { /* ... */ }

foo(1, 2, 3);     // error: ambiguous
foo(0.5f, -2.4f); // error: ambiguous
foo(1.5f, 3);     // error: ambiguous
```

- all arguments are implicitly convertible to `int` / `float` (both overloads are viable)
- both overloads take arguments as forwarding reference `T&&` (perfect match)
- SFINAE-condition: test if overload is viable, it is *not* part of overload resolution
- wrt. overload resolution, both overloads are equivalent

overload-resolution ambiguity: can C++20 concepts help out?

- [over.match.best.general]: desired overload must be *more constrained*
  - requires concepts like: `Matches_T1_BetterThan_T2`
  - effectively: rebuilding overload-resolution with concepts
  - feasible for two types, gets messy for 3 or more
  - technical issue: fold expression over constraints is an atomic constraint

overload-resolution ambiguity: what have we reached?

HVFs: current best practice & overload resolution

- parameter pack + SFINAE / requires are the best approaches, but...
- overload-resolution is compromised
  if more than one is viable, then the call is ambiguous
- solution: the overloaded HVFs have to know of each other

merging overloaded HVFs: step-by-step guide

1. provide interface to specify the overloaded types
2. perform overload resolution
    - generate all possible overloads
    - use built-in overload-resolution to resolve call
3. report selected type to the user

- uses C++17 for simplicity
- uses Peter Dimov's **Boost.MP11**

1. provide interface to specify the overloaded types

- template parameter pack
- encapsulated in `mp_list` (Boost.MP11)
- on the call site

```
// list containing types for overload-resolution
using overloaded_types = mp_list<
    int,
    float
>;

// helper alias template
template<typename... Args>
using enabler = decltype(enable<overloaded_types, Args...>());

// "overloaded" HVF for int and float
template<typename... Args, enabler<Args...>* = nullptr >
void foo(Args&& ... args) { /* ... */ }
```

2. perform overload resolution

- for each type, generate all possible overloads
  - infinite number of overloads, but...
  - only a finite number viable: arity of the actual call
  - generate (the one) test-function for the type
- run built-in overload-resolution on all of them

library for overloading HVFs (26 LOC)

```
template <size_t Index, class Params>
struct single_function;

template <size_t Index, class... Params>
struct single_function<Index, mp_list<Params...>> {
    constexpr std::integral_constant<size_t, Index> test(Params...) const;
};

template <size_t arity, class Indices, class... Types>
struct _overloads;

template <size_t arity, size_t... Indices, class... Types>
struct _overloads<arity, std::index_sequence<Indices...>, Types...>
    : single_function<Indices, mp_repeat_c<mp_list<Types>, arity>>... {
    using single_function<Indices, mp_repeat_c<mp_list<Types>, arity>>::test...;
};

template <class Types, size_t arity>
struct overloads;

template <class... Types, size_t arity>
struct overloads<mp_list<Types...>, arity>
    : _overloads<arity, std::index_sequence_for<Types...>, Types...> {};

template<class Types, typename... Args>
constexpr decltype(overloads<Types, sizeof...(Args)>{}.test(std::declval<Args>()...)) enable();
```

analyzing the HVF-library: `enable` function

```
template<class Types, typename... Args>
constexpr
decltype(overloads<Types, sizeof...(Args)>{}.test(std::declval<Args>()...))
enable();
```

- invoked by caller with type list and actual arguments
- return type: executes overload-resolution by calling `test`
- returns `std::integral_constant<size_t, Index>` indicating then best match

analyzing the HVF-library: creating 'viable' type-lists

```
template <size_t arity, class Indices, class... Types>
struct _overloads;

template <size_t arity, size_t... Indices, class... Types>
struct _overloads<arity, std::index_sequence<Indices...>, Types...>
    : single_function<Indices, mp_repeat_c<mp_list<Types>, arity>>... {
    using single_function<Indices, mp_repeat_c<mp_list<Types>, arity>>::test...; // C++17 required
};

// required to create compile time index-sequence
template <class Types, size_t arity>
struct overloads;

template <class... Types, size_t arity>
struct overloads<mp_list<Types...>, arity>
    : _overloads<arity, std::index_sequence_for<Types...>, Types...> {};
```

```
template <size_t arity, class Indices, class... Types>
struct _overloads;

template <size_t arity, size_t... Indices, class... Types>
struct _overloads<arity, std::index_sequence<Indices...>, Types...>
    : single_function<Indices, mp_repeat_c<mp_list<Types>, arity>>... {
    using single_function<Indices, mp_repeat_c<mp_list<Types>, arity>>::test...;
};
```

- generate list of desired arity for each type

```
mp_repeat_c<mp_list<Type>, arity>> == mp_list<Type, ..., Type>
                                                \ arity-times /
```

- instantiate `single_function` with such a list
  for each 'overloaded' type using pack expansion

```
template <size_t arity, class Indices, class... Types>
struct _overloads;

template <size_t arity, size_t... Indices, class... Types>
struct _overloads<arity, std::index_sequence<Indices...>, Types...>
    : single_function<Indices, mp_repeat_c<mp_list<Types>, arity>>... {
    using single_function<Indices, mp_repeat_c<mp_list<Types>, arity>>::test...;
};
```

C++17 `using` -declaration with pack expansion to pull all `test` functions into scope

core of the library: generate the test function

```
template <size_t Index, class Params>
struct single_function;

template <size_t Index, class... Params>
struct single_function<Index, mp_list<Params...>> {
    constexpr std::integral_constant<size_t, Index> test(Params...) const;
};
```

- instantiated for each overloaded type
- remember: `sizeof...(Params)` is the arity of the call
- defines `test` with desired argument-types and arity
- technique also used in STL-implementations:
  to generate the converting constructors in `std::variant` for the type-alternatives

3. report selected type to the user

- in case of ambiguity: compilation error
- otherwise: return index of selected type
  - as compile-time constant
  - usable in template-dispatching or `if constexpr`

```cpp
// list containing types for overload-resolution
using overloaded_types = mp_list<int, float>;

// helper alias template (simplifies repetitions)
template<typename... Args>
using enabler = decltype(enable<overloaded_types, Args...>());

// "overloaded" function for int and float
template<typename... Args, enabler<Args...>*  = nullptr >
void foo(Args&& ... args) {
    constexpr auto selected_type_index = enabler<Args...>::value;
    if constexpr (selected_type_index == 0) {
        // "int"-overload invoked
        std::cout << "overload: (int, ...)" << std::endl;
    } else if constexpr (selected_type_index == 1) {
        // "float"-overload invoked
        std::cout << "overload: (float, ...)" << std::endl;
    }
}
```

HVF live demo @ godbolt.org/z/94nGrbaxd

questions?

hop

from HVFs to function-parameter building-blocks

hop: function-parameter building-blocks

required features

- create parameter-lists with type-generators
- create overload-resolution condition from overload-set
- within function implementation
  - identify active overload
  - accessing / forwarding arguments (esp. defaulted parameters)

hop: type-generators

- any cv-qualified C++ type
- repetition
- sequencing
- alternatives
- trailing / non-trailing defaulted parameter
- forwarding references with / without condition
- template argument deduction (per argument / global / mixed)
- adapt existing functions
- tag types / overloads for easier access

repetition

- base template for repetition: match `_min` to `_max` times

```
// repetition
template<class T, size_t _min, size_t _max = infinite>
struct repeat;
```

- syntactic sugar: specializations of `repeat`

```
template<class T>                               // parameter pack
using pack = repeat<T, 0, infinite>;

template<class T>                               // non-empty parameter pack
using non_empty_pack = repeat<T, 1, infinite>;

template<class T>                               // optional parameter (w/o default value)
using optional = repeat<T, 0, 1>;

template<class T, size_t _times>                // exactly n-times
using n_times = repeat<T, _times, _times>;

using eps = repeat<char, 0, 0>;                 // no parameter (epsilon)
```

sequencing

- matching type lists one after another

```
template<class... Ts>     // sequencing parameters
struct seq;
```

alternatives

- matching exactly one of the specified type lists

```
template<class... Ts>     // alternative types
struct alt;
```

defaulted parameters

- eats argument, if it fits the type, otherwise injects default-created argument

```
// C++-style defaulted parameter: only at end of the parameter list
template<class T, class _Init = impl::default_create<T>>
struct cpp_defaulted_param;

// general defaulted parameter: can appear anywhere in the parameter list
template<class T, class _Init = impl::default_create<T>>
struct general_defaulted_param;
```

forwarding references

- unrestricted perfect forwarding

```
struct fwd;
```

- conditional perfect forwarding (SFINAE)

```
// forward-reference guarded by meta-function
template<template<class> class _If>
struct fwd_if;

// forward-reference guarded by quoted meta-function
template<class _If>
struct fwd_if_q;
```

template argument deduction

- global: all instances deduce the same set of types

```
template<template<class...> class Patterns>
struct deduce;
```

- local: deduced instances are independent of each other

```
template<template<class...> class Patterns>
struct deduce_local;
```

- mixed: specify, which template arguments are matched globally

```
template<class GlobalDeductionBindings, template<class...> class Pattern>
struct deduce_mixed;
```

  - sequence of `vector<T, Alloc>` with same `T` but arbitrary `Alloc`
  - sequence of `map` with the same key-type

adapt existing functions

- adapting an existing function

```
template<auto funtion>
struct adapt;
```

- adapting overload-sets, templates, functions with defaulted parameters

```
template<class Adapter>
struct adapted;

// adapt overload-set 'qux'
    struct adapt_qux {
        template<class... Ts>
        static decltype(qux(std::declval<Ts>()...)) forward(Ts&&... ts) {
            return qux(std::forward<Ts>(ts)...);
        }
    };
```

auxiliary stuff

- tagging to refer by name instead by position
  - tagging overloads
  - tagging types - e.g. for accessing arguments from a pack
- SFINAE condition on the whole overload

hop-grammar for function-parameters

```
CppType ::=
    any (cv-qualified) C++ type

Type ::=
    CppType
    | tagged_ty<tag, Type>

ArgumentList ::=
    Argument
    | ArgumentList, Argument
```

```
Argument ::=
    Type
    | repeat<Argument, min, max>
    | seq<ArgumentList>
    | alt<ArgumentList>
    | cpp_defaulted_param<Type, init>
    | general_defaulted_param<Type, init>
    | fwd
    | fwd_if<condition>
    | deduce{_local|_mixed}<Pattern>
```

hop-grammar: examples

- at least one `int`

```
non_empty_pack<int>
```

- list of `double` followed by optional `options_t`

```
seq<pack<double>, cpp_defaulted_param<options_t>>
```

- optional `options_t` followed by a list of `double`

```
seq<general_defaulted_param<options_t>, pack<double>>
```

- name-value pairs

```
pack<seq<string, alt<bool, int, double, string>>>
```

- local type deduction: a list of `vector` of arbitrary value-types

```
template<class T>
using vector_alias = vector<T>const&;

pack<deduce_local<vector_alias>>
```

- mixed global/local type deduction: a pack of maps, all having the same key-type

```
// T1 is bound with global_deduction_binding
template<class T1, class T2>
using map_alias = map<T1, T2>const&;

// map index in pattern to tag-type:
// all deduced types with that index have to be the same!
using bindings = mp_list<global_deduction_binding<0, tag_map_key_type>>;

pack<deduce_mixed<bindings, map_alias>>
```

- only types fitting into a pointer

```
template<class T>
struct is_small : mp_bool<sizeof(remove_cvref_t<T>) <= sizeof(void*)> {};

pack<fwd_if<is_small>>
```

- matching all above as an overload set

```
using overloads = ol_list<
      ol< non_empty_pack<int>                                  >
    , ol< pack<double>, cpp_defaulted_param<options_t>         >
    , ol< general_defaulted_param<options_t>, pack<double>     >
    , ol< pack<seq<string, alt<bool, int, double, string>>>>   >
    , ol< pack<deduce_local<vector_alias>>                     >
    , ol< pack<deduce_mixed<bindings, map_alias>>              >
    , ol< pack<fwd_if<is_small>>                               >
>;
```

expanding the grammar: generate type-lists for an overload

- for HVFs
  - "repeat the type *arity*-times"

```
mp_repeat_c<mp_list<Types>, arity>
```

- combinatorial problem for hop
  - "generate all parameter-lists that accept *arity* arguments"

generate type-lists for an overload (cont.)

recursively traverse the overload

ensure progress (i.e. generate parameters) to avoid infinite recursion

1. for each sub-expression of the overload
    - analyse the min / max length of the producible type-lists
2. recursively traverse the overload
    - build types-lists $T_1, \ldots, T_n$ for all sub-expressions
    - generate cartesian product $T_1 \times \ldots \times T_n$ and concatenate tuples
    - prohibit infinite recursion
        - use min / max info to limit to types-lists that can match the call's arity
        - ensure progress: do not repeat type-lists

generate test-function from type-list

- for HVFs
  - unpack `mp_list` and expand as arguments of `test`

```
template <size_t Index, class... Params>
struct single_function<Index, mp_list<Params...>> {
    constexpr std::integral_constant<size_t, Index> test(Params...) const;
};
```

- additional tasks for hop
  - 'templated' arguments
    - forwarding references, local / global conditions, adapters
  - template argument deduction
  - don't forget about: removing the tag-templates

generate test-function from type-list (cont.)

```
template< typename... Args,
    std::enable_if_t<mp_invoke_q<_If, Args...>::value          &&
    deduction_helper<mp_list<Args...>, mp_list<TypeList...>>::value
    , int* > = nullptr >
constexpr result_t test(typename unpack<TypeList, Args>::type...) const;
```

- `mp_invoke_q<_If, Args...>` - checking global conditions against actual parameter types
- `deduction_helper` - global template argument deduction
- `unpack`
  - returns parameter type / generates type for forwarded parameters
  - checks local conditions
  - local template argument deduction
  - removes tag-types

test-function: template argument deduction (in an ideal world)

```
template<template<class...> class... Patterns>
struct deducer_t {

    template<template<class...> class... Patterns, class... T>
    static mp_list<std::true_type, mp_list<T...>> test(Patterns<T...>...);

    static mp_list<std::false_type, mp_list<>> test(...);
};
```

a pack of deduction patterns validated simultaneously against a pack of types - simple as that, but...

test-function: template argument deduction (in our universe)

```
template<template<class...> class... Patterns, class... T>
static mp_list<std::true_type, mp_list<T...>> test(Patterns<T...>...);
```

is generally invalid C++:

- `Patterns` may be an alias template
- instantiating an alias template with a pack is not allowed
- using Boost.Preprocessor to generate type-deduction `test` functions
- configurable through macro `HOP_MAX_DEDUCIBLE_TYPES`

generate test-function from type-list (reprise)

what else to do:

- keep a record of formal parameter types (the type-list)
- keep a record of skipped defaulted parameters
- create result-type from all that, i.a.
  - index of overload in overload-set
  - generated type-list, skipped defaulted parameters
  - locally / globally deduced types

hop live demo on godbolt

- ints & floats @ godbolt.org/z/esqc6GMsY
- reference-types @ godbolt.org/z/359Y34b9P
- tagging overloads @ godbolt.org/z/8oY5T7Wb6
- default arguments @ godbolt.org/z/aWPMdb6jG
- one entry point to rule them all? @ godbolt.org/z/dWG75MPeq
- template argument deduction @ godbolt.org/z/19ddvTfvW

hop statistics

- currently ~1800 LOC
- grammar: ~150 LOC
- template argument deduction: ~250 LOC
- test-function generator: ~300 LOC
- argument-access ~400 LOC
- overload generation: ~400 LOC
  in HVF-library it's just calling `mp_repeat`

summary

- homogenous variadic functions
  - analyzed the language support
  - overload-resolution problem for standalone implementations -> merge 'em
  - HVF library in 26 LOC
- hop
  - grammar for function-parameter building-blocks
  - generation of type-lists & overload resolution
  - tagging & accessing of overloads, types and arguments

hopping on

- type-list generation improvements by early exit
  - divisibility-checks for repeated lists
  - check convertibility of argument to type
- use `concepts` to create a type-system for the hop-internal

questions / remarks?

visit github.com/tobias-loew/hop

now, let's hop together!