P1521R0

Types as function names

Author: John Bandela

Audience: EWGI

Project: ISO/IEC JTC1/SC22/WG21 14882: Programming Language — C++

Reply-To: John Bandela (jbandela@gmail.com)

TLDR:

This paper proposes a new call syntax `type.(args)` where `type` is an arbitrary type.

Motivation

Currently we use identifiers to designate free and member functions. However, this is a source of limitations for C++. We are limited due to the need to maintain backwards compatibility, incomplete namespacing, and lack of meta-programmability for these names. For each of these limitations, there are features that we would like to have, but we do not.

1. Backwards compatibility limitations

- Universal function call syntax
- Extension methods

2. Incomplete namespacing

- Extension points

3. Lack of meta-programmability

- Overload sets
- Smart references/proxies
- Runtime polymorphism without inheritance
- Exception and non-exception based overloads
- Adding monadic bind to pre-existing monad types
- Making a monadic type automatically map/bind

All these issues can be addressed if we allow functions to be designated using types.

Definitions

Let *action_tag* be any type, including an incomplete type.

Proposed calling syntax

Anywhere where a function name is used to call a function or member function, `action_tag.` may be used instead. In addition, `t.action_tag.(args..)` is equivalent of `action_tag.(t,args..)`.

```
// Declaration
void free_function();
free_function();
// Definition of action tag type
struct foo;
foo.(); // calling an action tag
// Declaration
struct object{
  void member_function();
};
object o;
o.member_function();

// Declaration of action type tag.
struct bar;
// The following 2 calls are equivalent.
o.bar.();
bar.(o);
```

Proposed definition syntax

An action_tag may be declared and defined in the following manner:

```
// Declare action_tags.
struct foo;
struct bar;

// With specified return value
auto foo.(parameter) -> return_type;

// With deduced return value;
auto bar.(parameter){
    return 5;
}

// All function template syntax is supported
template<typename T>
```

```
auto bar.(T t){
    return 5;
}

// Template parameters can be the action_tag
template<typename ActionTag>
auto ActionTag.(object&){
    // Implement all action tags that take a object as a parameter
}

// Concepts supported
template<typename ActionTag>
requires my_concept<ActionTag>
auto ActionTag.(object&){
    // Implement all action tags that take a object as a parameter
}
```

Supporting types

is_action_tag_invocable

```
// iff ActionTag.(declval<Args>()...) is well-formed.
template<typename ActionTag, typename... Args>
struct is_action_tag_invocable :std::true_type{}
// otherwise
template<typename ActionTag, typename... Args>
struct is_action_tag_invocable :std::false_type{}

template<typename ActionTag, typename... Args>
inline constexpr bool is_action_tag_invocable_v =
is_action_type_invocable<ActionTag,Args...>::value;
```

action_tag_overload_set

A class that overloads `operator()` and forwards all arguments to an action tag

```
template<typename ActionTag>
struct action_tag_overload_set{
  template<typename... T> typename =
  requires is_action_tag_invocable_v<ActionTag,T...>
  decltype(auto) operator()(T&&... t) const{
    return ActionTag.(std::forward<T>(t)...);
  }
};
```

Applications

Overcoming Backwards Compatibility Limitations

Universal Function Call Syntax

While universal function call syntax would be very useful, there are backward compatibility issues. Using action tags because we do not have to worry about backward compatibility we can define that `t.foo.(args...)` and `foo.(t,args...)` are equivalent.

Extension methods

Along the same lines of what we are doing, we can easily add new methods to a type. The nice thing is that since we are using types, we can use namespacing to make sure they don't conflict with future methods. For example, wouldn't it be nice for int to have a `to_string` method?

```
namespace my_methods{
  struct to_string;
  auto to_string.(int i) -> std::string{
    return std::to_string(i);
  }
}
int i = 5;
i.my_methods::to_string.();
```

We could even extend this and provide a default to_string implementation.

```
template<typename T>
auto to_string.(const T& t) -> std::string{
  std::stringstream s;
  s << t;
  return s.str();
}
```

This would also be useful for ranges actions and views where instead of of forcing `operator|` into service with it's precedence quirks, we could actually use something like

```
v.action::sort.().action::unique.().view::filter.([](auto&){/*stuff*/});
```

Overcoming incomplete namespacing

Customization points

Whenever we define a free function that is called through ADL, we are essentially reserving that name across namespaces. An illustration is from https://quuxplusone.github.io/blog/2018/06/17/std-size/ . There was a library that defined size in it's own namespace, which compiled fine with `C++11`. However, `C++17` introduced `std::size` which then caused a conflict. With

action tags, this would not be an issue. For example, we could put extension points in a hypothetical namespace.

```
namespace std::extension_points{
  struct size;
  template<typename T>
  auto size.(T& t){
    return t.size();
  }
}
```

Which could then be called

```
std::vector v{1,2,3,...};
std::extension_points::size.(v);
```

in addition, because with action tags the namespace of the tag is taken into account for ADL lookup, we do not have to do the `using std::size/begin/end` dance.

```
// Not needed any more with action tags
using std::size;
size(v);
```

Overcoming lack of meta-programmability

The final limitation of using identifiers for functions is that we cannot use meta-programming except for the pre-preprocessor token pasting. There is nothing in C++ language outside the pre-processor that can manipulate a function name. The only thing you can do with a function name is to call it or get a function pointer (which you may not be able to do if it is overloaded or is a function template). The proposed feature would allow meta-programming and open up the following features.

Overload sets

When calling a generic function such as `transform` it is not convenient to pass in an overloaded function. We either have to specify the exact parameters, or wrap in a lambda. With this proposal, any action tag overload set can easily be passed to a generic overload using `action_tag_overload_set`. So if you have action tag `foo`, you can call `transform(v.begin(),b.end(), action_tag_overload_set<foo>{});`

Smart references and proxies

While you can easily define a smart pointer in C++ by overloading `operator->()`, there is no easy way to define a smart reference that will forward the member function calls. With this proposal, you can write smart reference that forwards action tags.

```
namespace dumb_reference {
  // Action tag which resets the dumb reference. Applies to the dumb reference itself.
  struct reset;
  template<typename T>
  class basic_dumb_reference{
    T* t_;
    public:

    template<typename ActionTag, typename Self, typename... Args>
    requires is_action_tag_invocable_v<ActionTag,Self,Args...>
      && std::is_same_v<std::remove_cvref_t<Self>,basic_dumb_reference<T>>
    friend decltype(auto) ActionTag.(Self&&
    self, Args&&... args){
      return ActionTag.(self.t_,std::forward<T>(t)...);
    }

    friend auto reset.(basic_dumb_reference<T>& t) -> void{
      t_ = &t;
    }
  };
}
int i = 0;
int j = 2;
dumb_reference::basic_dumb_reference<int> ref;
ref.dumb_reference::reset.(i);
ref.to_string.(); // "0"
ref.dumb_reference::reset.(j);
ref.to_string.(); // "2"
}
```

In addition to forwarding easily, action tags also solve the issue of how to differentiate the member functions of the smart reference itself. Because action tags are namespaced, it is obvious that the action tag reset applies to the reference itself and not to what is being stored. In addition, if we could extend this tecnhique to making for example debugging proxies doing such things as logging all action tags and parameters if we so desired. In fact, we could even use this to create a transparent remoting proxy which will convert action tag invocations into a remote procedure call.

Composition

One of the reasons inheritance is used over composition is the convenience of not having to manually forward member function calls. With action tags you can automatically forward all non-implemented action tags to the contained object.

```
struct object{
  contained_object contained_;
```

```
  template<typename ActionTag, typename... T>
  requires is_action_tag_invocable_v<
   ActionTag,decltype(declval<object>().contained_),T...>
  friend decltype(auto) ActionTag.(const object& self, T&&... t){
    return ActionTag.(self.contained_,std::forward<T>(t)...);
  }
};
```

Polymorphism without inheritance

You can use template metaprogramming to define a polymorphic_object that takes action tags signatures.

```
template<typename ReturnType, typename ActionTag, typename...
Parameters>
struct signature{};
template<typename... Signatures>
struct polymorphic_object{
  // Implementation unspecified.
};

// Action tags
struct foo;
struct bar;

using foo_and_barable =
polymorphic_object<signature<void,foo>,signature<void,bar,int>>;
std::vector<foo_and_barable> stuff;
for(auto& v: stuff){
  v.foo.();
  v.bar.(1);
}
```

Thus instead of basing polymporphism on inheritance patterns, it can be based on support for invoking an action tag with a set of parameters.

Exception and non-exception based overloads

Sometimes a class needs both a throwing and a non-throwing version of a member function. The solution adopted in `std::filesystem` and the networking ts, is to have 2 overloads of each member function.

```
class my_class{
  void operation1(int i, std::error_code& ec);
  void operation1(int i){
    std::error_code ec;
    operation1(i,ec);
    if(ec) throw std::system_error(ec);
```

```
  }
  // And so on for all supported operations
};
```

There is a lot of boilerplate. Using this proposal, we can automatically generate the throwing versions from the non-throwing versions.

```
#include <system_error>
struct operation1;
struct operation2;

class my class{
// stuff
public:
  // std::error_code based error handling
  friend auto operation1.(my_class& c, int i, std::error_code& ec)->void;

 // std::error_code based error handling
 friend auto operation2.(my_class& c, int i,int j, std::error_code& ec) -> void;

};

// Automatically generate the throwing counterparts, when an action
// tag is called without passing in an std::error_code.
template <typename ActionTag, typename... Args>
requires !std::disjunction_v<std::is_same<std::error_code,std:
:decay_t<Args>>...> && is_action_tag_invocable_v<ActionTag, my_class&,
Args..., std::error_code&>>
>
decltype(auto) ActionTag.(my_class& self,Args&&... args) {
  std::error_code ec;
  auto call = [&]() mutable {
    return ActionTag.(self,std::forward<Args>(args)..., ec);
  };

  using V = decltype(call());
  if constexpr (!std::is_same_v<void, V>) {
    V r = call();
    if (ec) throw std::system_error(ec);
    return r;
  }
  else {
    call();
    if (ec) throw std::system_error(ec);
  }
}
```

Adding monadic bind to existing monad types.

Let's say we have this tree node type.

```
// Action tags
struct get_parent;
struct get_child;

class node{
  node* parent;
  std::vector<node*> children;
public:
  friend auto get_parent.(const node& self)->node*{
    return parent;
  }
  friend auto get_child.(const node& self, int i)->node*{
    if(i >= 0 && i < children.size())
      return children[i];
    else
      return nullptr;
  }
};
// We can use like this
node n;
node* parent = n.get_parent.();
// Get the first child
node* child = n.get_child.(0);
```

However, if we want to get the grandparent, we have to write code like

```
node* parent = n.get_parent.();
node* grandparent = parent?parent->get_parent.():nullptr;
```

However, we observe that the pointer returned is a monad. We can use this to write a monadic bind.

```
template<typename ActionTag>
struct and_then;

template <typename ActionTag, typename T, typename... Args>
auto <and_then<ActionTag>>(T&& t, Args&&... args)
  ->decltype(ActionTag.(*std::forward<T>(t),std::forward<Args>(args)...)){
  if(t){
    return ActionTag.(*std::forward<T>(t),std::forward<Args>(args)...);
  }
  else{
    return nullptr;
  }
}
```

The we can do stuff like

```
node n;
auto great_grandparent =
n.get_parent.().and_then<get_parent>.().and_then<get_parent>.();
auto grand_child = n.get_child.(0).and_then<get_child>.(0);
```

This is much simpler, and less error prone than before. In addition, even if we change get_parent and get_child to return `std::shared_ptr<node>` instead of `node*` our code will still work.

Making a monadic type automatically map/bind

In fact, we could take this even further, and implement monadic binding and functor mapping at a type level. Using `std::optional` as an example:

```
template<typename T>
struct wrap_optional{
  using type = std::optional<T>;
};
template<typename T>
struct wrap_optional<std::optional<T>>{
  using type = std::optional<T>;
};

template<typename T>
using wrap_optional_t = typename wrap_optional<T>::type;

template<typename ActionTag, typename T, typename...Args>
auto ActionTag.(std::optional<T>& self, Args&&... args)
 -> wrap_optional_t<decltype(ActionTag.(*self,std::forward<Args>(args)...))>{
  if(!self){
    return std::nullopt;
  }
  return ActionTag.(*self,std::forward<Args>(args)...);
}
```

Then we could do the following

```
// Reads a optional<string> from istream
struct read_string;
auto read_string.(std::istream&) -> std::optional<std::string>;

// Converts from a string to optional<T>
template<typename T>
struct from_string;
template<typename T>
auto from_string<T>.(std::string_view s) -> std::optional<T>;
```

```
// Squares an integer
struct square;
auto square.(int i) -> int;

int main(){
  auto opt_squared =
    std::cin.read_string.()
            .from_string<int>.()
            .square.();
}
```

Alternative definition of action tag

```
auto foo.; // foo is an action tag.

template<int N>
auto get.; // action tag taking non-type template parameter.

template<typename T>
auto numeric_cast.; // action tag taking type template parameter

template<auto action_tag.>
auto and_then.; // action tag taking another action tag as a parmeter
```

Alternative calling syntax

The calling syntax would be unchanged.

Alternative definition syntax

The definition syntax would be unchanged.

Alternative Example: and_then

```
template <auto ActionTag., typename T, typename... Args>
auto and_then<ActionTag>.(T&& t, Args&&... args)
 -> decltype(ActionTag.(*std::forward<T>(t),std::forward<Args>(args)...)){
  if(t){
    return ActionTag.(*std::forward<T>(t),std::forward<Args>(args)...);
  }
  else{
    return nullptr;
  }
}
```

Thoughts on alternative syntax

While the alternative definition is interesting, I see the following downsides:

- We would need to introduce a new kind of declaration. We would also have to make sure that all the template type machinery also works with this
- Other than saving some characters, no specific advantages of this approach are obvious to me.

Given the above, I believe that sticking with actions tags as types makes the most sense.

Acknowledgements

Thanks to Arthur O'Dwyer, James Dennet, Richard Smith, Waldemar Horwat, Kyle Konrad, Nicol Bolas for providing valuable feedback.