Exhaustiveness Checking for Pattern Matching

Document #:	P2211R0
Date:	2020-11-16
Project:	Programming Language C++
Audience:	Evolution

Reply-to: Bruno Cardoso Lopes <bruno.cardoso@gmail.com>
Sergei Murzin <smurzin@bloomberg.net>
Michael Park <mcypark@gmail.com>
David Sankel <dsankel@bloomberg.net>
Dan Sarginson <dsarginson@bloomberg.net>
Bjarne Stroustrup <bjarne@stroustrup.com>

Contents

1 Abstract

With the planned introduction of pattern matching into C++, there's an opportunity to give it support for "exhaustiveness checking" which enables compile-time detection and diagnosis of several common pattern matching bugs. This paper describes the design for such a feature that intentionally caters to typical software engineering patterns and commonly understood semantics, as opposed to a pedantic interpretation of code. It is our hope that such a design will maximize this feature's utility as a bug prevention mechanism.

```
enum Color { Red, Green, Blue };
//...
Color c = /*...*/;
vec3 v = inspect(c) {                  // ERROR: Missing case 'Blue'
  case Red   => vec3(1.0, 0.0, 0.0);
  case Green => vec3(0.0, 1.0, 0.0);
};

vec3 v2 = inspect(c) {                 // OKAY
  case Red   => vec3(1.0, 0.0, 0.0);
  case Green => vec3(0.0, 1.0, 0.0);
  case Blue  => vec3(0.0, 0.0, 1.0);
};
```

2 Introduction

This paper describes a mechanism that allows for compilation errors for `inspect` expressions that that aren't exhaustive, i.e. none of the patterns match a particular value. Without such a feature, the best a compiler could do in such a situation is produce a warning and, if unheeded, undefined behavior or some other failure mode would result at runtime.

The "Examples" section of this paper builds up our proposed semantics using a series of code snippets. It is our intent that this section be easily understood by those already familiar with [P1371R2]. This section is a followed by a "Specification" section that includes a formal treatment of the semantics. The remaining sections consider some special cases and alternatives.

In all our examples, we utilize the pattern matching syntax specified in [P1371R2] for expressions. While it is known that several syntax changes will be made in the next revision of that paper, none of those changes are expected to impact exhaustiveness checking semantics.

We implemented a proof-of-concept exhaustiveness checker that implements the design described by this paper in [RefImpl]. It is an adaptation of the efficient algorithm described by Luc Maranget in [maranget_2007].

3 Examples

This section presents our proposed exhaustiveness checking semantics by building on a series of simple examples.

3.1 Exhaustive Patterns

Exhaustive patterns are patterns that match any possible value for a given type. The wildcard pattern (`__`) in the third arm of the `inspect` statement below is one such pattern, but the literal `2` in the second arm is not.

```
inspect(i) -> std::ostream& {
  1  => std::cout << "one";
  2  => std::cout << "two";
  __ => std::cout << "something else";
}
```

Binding patterns, such as `x` in the following example, are also exhaustive.

```
inspect(i) -> std::ostream& {
  1 => std::cout << "one";
  2 => std::cout << "two";
  x => std::cout << x;
}
```

Finally, structured binding patterns where every subpattern is exhaustive is also an exhaustive pattern as in the two arms of the `inspect` expression below. We'll delve into more detail on structured binding patterns later.

```
struct Point { int xCoordinate; int yCoordinate; }

struct Box { Point topLeft; int width; int height };

inspect(box) {
  [tl, w, h] => /*...*/;
  [[x,y], w, h] => /*...*/;
}
```

3.2 Fundamental Types

The following example results in a compilation error because ther isn't a pattern that matches the value `false`.

```
bool b = /*...*/;
char const * const str = inspect(b) { true => "true"; }; // ERROR: missing
                                                          // false pattern
```

This error can be fixed by adding an exhaustive pattern.

```
char const * const str = inspect(b) {
  true => "true";
  __   => "false";                    // OKAY, pattern exhaustive
};
```

`bool` happens to be a fundamental type that doesn't always require an arm with an exhaustive pattern. Providing a `false` pattern can also be used to fix the error.

```
bool b = /*...*/;
char const * const str = inspect(b) {
  true => "true";
  false => "false";                   // OKAY, all values are covered.
};
```

Other fundamental types with a small number of values (e.g. `std::nullptr_t`) behave in a similar way. Types populated with many values (e.g. `int`) require an exhaustive pattern arm.

3.3 `enum`

`enum` types present an interesting case. The example below is an error because the `Blue` enumerator isn't covered by one of the `inspect` arms.

```
enum Color { Red, Green, Blue };
//...
Color c = /*...*/;
vec3 v = inspect(c) {                  // ERROR: Missing case 'Blue'
  case Red   => vec3(1.0, 0.0, 0.0);
  case Green => vec3(0.0, 1.0, 0.0);
};
```

This can be fixed by adding a pattern for the missing `Blue` enumerator.

```
vec3 v = inspect(c) {                 // OKAY
  case Red   => vec3(1.0, 0.0, 0.0);
  case Green => vec3(0.0, 1.0, 0.0);
  case Blue  => vec3(0.0, 0.0, 1.0);
};
```

The astute reader will point out that `enum`s can have values different from their enumerators. Our `Color` type, for example, can take on the value `4`. While almost all `enum` types have this behavior defined, it is relatively rare that extra-enumerator values are considered "valid" for the type from an application semantic perspective. For this reason, we do not require an exhaustive pattern for `enum` types.

However, we still need to define what happens at runtime when an extra-enumerator value is inspected and no patterns match. Considering the various options available (e.g. undefined behavior or throwing an exception), a call to `std::terminate` seems the most palatable.

```
Color val_outside_enumerators = static_cast<Color>(3);

vec3 v = inspect(val_outside_enumerators) {     // 'std::terminate' at runtime
  case Red   => vec3(1.0, 0.0, 0.0);
  case Green => vec3(0.0, 1.0, 0.0);
  case Blue  => vec3(0.0, 0.0, 1.0);
};
```

Those desiring different behavior in this situation are free to add a wildcard arm with their desired behavior by throwing an exception, as in the snippet below, or calling something like `std::unreachable` ([P0627R3]) if undefined behavior is desired.

```
Color val_outside_enumerators = static_cast<Color>(3);

vec3 v = inspect(val_outside_enumerators) {     // Throw exception at runtime
  case Red   => vec3(1.0, 0.0, 0.0);
  case Green => vec3(0.0, 1.0, 0.0);
  case Blue  => vec3(0.0, 0.0, 1.0);
  __         => throw Up{};
};
```

3.4 Classes and `tuple`-like Types

Exhaustiveness checking for classes and `tuple`-like types is defined in terms of exhaustiveness checking of their underlying types. Note how in the example below, `flagsV1` is inspected with the combination of wildcards and literals.

```
struct FlagsV1 {
  bool firstFlag;
  bool secondFlag;
};
```

```
inspect(flagsV1) {
  [false, false] => /*...*/;
  [true , false] => /*...*/;
  [__   , true ] => /*...*/;
};
```

In the definition of `FlagsV2` below, a defaulted `operator==` is provided. This allows us to freely mix `constexpr` value patterns and structured binding patterns.

```
struct FlagsV2 {
  bool firstFlag;
  bool secondFlag;

  bool operator==(const FlagsV2&) const = default;
};
```

```
constexpr auto allFalse = FlagsV2{ .firstFlag=false,
                                   .secondFlag=false };
inspect(flagsV2) {
  case allFalse => /*...*/;
  [true, false] => /*...*/;
  [__  , true ] => /*...*/;
};
```

Note that this mixture of `constexpr` value matching and structured binding matching for exhaustiveness checking does not work for custom `operator==` implementations.

```
struct FlagsV3 {
  bool firstFlag;
  bool secondFlag;

  bool operator==(const FlagsV3& other) const {
    return firstFlag  == other.firstFlag &&
           secondFlag == other.secondFlag;
  };
};
```

```
constexpr auto allFalse = FlagsV3{ .firstFlag=false,
                                   .secondFlag=false };
inspect(flagsV3) {
  case allFalse => /*...*/;
  [true, false] => /*...*/;
  [__  , true ] => /*...*/;  // ERROR: {false, false} case not handled.
};
```

This behavior results from the difficulty (it's undecidable) of determining if an arbitrary `operator==` function behaves identically to the conjunction of equality of a class's fields.

3.5 `variant`-like Types

Exhaustiveness checking for `variant`-like types work in a similar way to `enum`s.

In the following code, we create a `Command` type alias whose values are either `FireBlasters` or `Move` objects. This could, for example, represent a command in an X-Wing simulator.

```
struct FireBlasters{
  int intensity;
```

```
  bool operator==(const FireBlasters&) const = default;
};

enum Direction{ Left, Right };

struct Move{
  Direction direction;
  bool operator==(const FireBlasters&) const = default;
};

using Command = std::variant<FireBlasters, Move>;
```

The following function converts `Command` objects into strings. It, however, has a bug in that moving right isn't covered by any of the `inspect` expressions arms. This is detected and will produce an error at compile time.

```
std::string cmdToStringV1(Command cmd) {
  return inspect(cmd) {
    <FireBlasters> [i] => std::format("Fire Blasters with power {}", i);
    <Move> [case Left] => std::string("Move Left");

    // ERROR: No coverage for '<Move> [Right]' value.
  };
}
```

Adding a "move right" case fixes the issue.

```
std::string cmdToStringV2(Command cmd) {
  return inspect(cmd) { // OK
    <FireBlasters> [i]  => std::format("Fire Blasters with power {}", i);
    <Move> [case Left]  => std::string("Move Left");
    <Move> [case Right] => std::string("Move Right");
  };
}
```

The exceptionally sharp-witted reader will note that the "valueless by exception" state isn't handled by the `inspect` above even though it compiles. That is correct. As with `enum`, `std::terminate` is called at runtime in this situation.

```
Command pathological = /*...*/; // Somehow put pathological in the
                                // 'valueless_by_exception' state.

auto s = cmdToStringV2(pathological); // 'std::terminate'
```

If there is desire to handle this case explicitly, one may use a wildcard.

```
std::string cmdToStringV3(Command cmd) {
  return inspect(cmd) {
    <FireBlasters> [i]  => std::format("Fire Blasters with power {}", i);
    <Move> [case Left]  => std::string("Move Left");
    <Move> [case Right] => std::string("Move Right");
    __ => std::string("Pathological Command");
  };
}

//...
auto s = cmdToStringV3(pathological); // Assign 's' to "Pathological Command"
```

Note that this is analogous to `enum` behavior.

3.6 Special Cases

While we've provided examples of the core machinery above, but there are several special cases that also need to be considered.

3.6.1 Class Hierarchy Matching

Consider a `CommandV2` data structure that has the same use case as `Command`, but is instead implemented with a class hierarchy.

```
struct CommandV2 {
  virtual ~Command() = default;
};

struct FireBlastersV2 : CommandV2 {
  int intensity;
};

struct MoveV2 : CommandV2 {
  Direction direction;
};
```

The first thing to note is that having patterns that match all the possible values will not satisfy the exhaustiveness checker.

```
std::string cmdToStringV5(CommandV2 cmd) {
  return inspect(cmd) {
    <FireBlastersV2> [i]  => std::format("Fire Blasters with power {}", i);
    <MoveV2> [case Left]  => std::string("Move Left");
    <MoveV2> [case Right] => std::string("Move Right");

    // ERROR, exhaustive pattern required
  };
}
```

Instead, an exhaustive pattern is required when doing this kind of downcasting.

```
std::string cmdToStringV5(CommandV2 cmd) {
  return inspect(cmd) { // OK
    <FireBlastersV2> [i]  => std::format("Fire Blasters with power {}", i);
    <MoveV2> [case Left]  => std::string("Move Left");
    <MoveV2> [case Right] => std::string("Move Right");
    __                    => std::string("Unknown");
  };
}
```

3.6.2 Pattern Guards

Pattern guards provide a convenient way to express runtime conditions for pattern arms. However, as the example below illustrates, arms with guards are essentially ignored when making a compile-time exhaustiveness checking determination.

```
int fib(int n) {
  return inspect(n) {          // ERROR: Patterns not exhaustive
    0             => 0;
```

```
    1             => 1;
    n if (n >= 1) => fib(n-1) + fib(n-2);
    n if (n < 0)  => throw std::invalid_argument("fib called with negative");
  };
}
```

`inspect` statements such as these can usually be rewritten to utilize exhaustive patterns as below.

```
int fib(int n) {
  return inspect(n) {          // OK
    0             => 0;
    1             => 1;
    n if (n >= 1) => fib(n-1) + fib(n-2);
    __            => throw std::invalid_argument("fib called with negative");
  };
}
```

3.6.3 Dereference Extractors

Consider the following definition of `BinaryTree`.

```
struct BinaryTree;
struct Node { int value; };
struct Branch {
  std::unique_ptr<BinaryTree> left;
  std::unique_ptr<BinaryTree> right;
};
struct BinaryTree : std::variant<Node, Branch> {
  using std::variant<Node, Branch>;
};
```

The following function attempts to determine whether its argument has a depth of at least two. It makes use of the conditional dereference extractor (`(*?)`) to reach within the `unique_ptr` values. Unfortunately, the runtime nature of the conditional dereference extractor prevents compile-time determination of exhaustiveness.

```
bool depthAtLeastTwo(const BinaryTree & t) {
  return inspect(t) {             // ERROR: Patterns not exhaustive
    <Node> __                                  => false;
    <Branch> [(*?) <Branch> __, __]            => true;
    <Branch> [__, (*?) <Branch> __]            => true;
    <Branch> [(*?) <Node> __, (*?) <Node> __] => false;
    <Branch> [nullptr, nullptr]                => false;
  };
}
```

This can be fixed by replacing the `<Branch> [nullptr, nullptr]` pattern with `<Branch> __` which is exhaustive for the branch value.

```
bool depthAtLeastTwo(const BinaryTree & t) {
  return inspect(t) {             // OK
    <Node> __ => false;
    <Branch> [(*?) <Branch> __, __]            => true;
    <Branch> [__, (*?) <Branch> __]            => true;
    <Branch> [(*?) <Node> __, (*?) <Node> __] => false;
    <Branch> __                                => false;
  };
}
```

The unconditional dereference extractor (`(*!)`) on the other hand, because it is unconditional, contributes directly to the exhaustiveness checking algorithm as in the following example. Note that there is no need for `nullptr` handling in this example since the unconditional dereference extractor implies that null is not valid input.

```
bool depthAtLeastTwo(const BinaryTree & t) {
  return inspect(t) {              // OK
    <Node> __ => false;
    <Branch> [(*!) <Branch> __, __]           => true;
    <Branch> [__, (*!) <Branch> __]           => true;
    <Branch> [(*!) <Node> __, (*!) <Node> __] => false;
  };
}
```

For reference, we provide our preferred implementation below.

```
bool depthAtLeastTwo(const BinaryTree & t) {
  assert(no_nulls(t));
  return inspect(t) {
    <Node>   __                               => false;
    <Branch> [(*!) <Node> __, (*!) <Node> __] => false;
    __                                        => true;
  };
}
```

3.6.4 General Extractors

Like conditional dereference extractors, general conditional extractors do not contribute to pattern matching exhaustiveness checking.

```
int val = inspect(str) {     //ERROR: Non-exhaustive
  (regex_pat<"(\\d+)"> ?) [digits] => std::atoi(digits);
  (regex_pat<".*"> ?)      __      => -1;
}
```

Exhaustive patterns need to be used in conjunction with these patterns.

```
int val = inspect(str) {     //OK
  (regex_pat<"(\\d+)"> ?) [digits] => std::atoi(digits);
  __                               => -1;
}
```

Similar to unconditional dereference extractors, general unconditional extractors *do* contribute to exhaustiveness checking.

4 Specification

This section provides a formal specification of our exhaustiveness checking semantics. Both Pointer-like types and extractor patterns are omitted and will be provided in a future revision of this document.

For an expression `e` of type `T`, any `inspect` expression,

```
inspect(e) {
  /case_1/ => code_1  // arm_1
  /case_2/ => code_2  // arm_2
        ⋮
  /case_n/ => code_n  // arm_n
}
```

, must, for every q-value (q_i) of `T`, include an case (case$_j$) that q-matches that q-value (q-match(q_i, case$_j$) = `true`). q-values are defined on a per-type basis and the q-match function is defined on a per-pattern basis.

4.1 *q*-match and guards

Arms with guards do not contribute to compile-time exhaustiveness checking due to their runtime semantics. Therefore, we have the following rule:

- q-match(v, `pat` *inspect-guard*) is `false` for every q-value v and pattern `pat`.

4.2 *q*-values for fundamental types

- `std::nullptr_t` is defined to have a single q-value, `nullptr`.
- `bool` is defined to have two q-values, `true`, and `false`.
- The remaining fundamental types are defined to each have a single q-value ϵ.

4.3 *q*-match for fundamental types

Three patterns apply to fundamental types: wildcards (*wildcard-pattern*), bindings (*binding-pattern*), and expressions (*expression-pattern*).

Wildcards and bindings, unsurprisingly, match any q-value.

- q-match(v, *wildcard-pattern*) is `true` for every q-value v
- q-match(v, *binding-pattern*) is `true` for every q-value v

Expression patterns q-match only when the expression evaluates to the particular q-value.

- q-match(v, *expression-pattern*) is `true` if the expression pattern evaluates to q-value v and `false` otherwise.

Note that because expression patterns cannot evaluate to ϵ, q-match(ϵ, *expression-pattern*) is always `false`.

4.4 `enum` types

Every `enum` type `E` is defined to have one q-value per enumerator e_i defined by `E`. q-match is defined the same as it is for fundamental types.

4.5 Classes

Classes without data members have a single q-value `{}` and we have the following rules:

- q-match(`{}`, *wildcard-pattern*) is `true`
- q-match(`{}`, *binding-pattern*) is `true`
- q-match(`{}`, *expression-pattern*) is `true`
- q-match(`{}`, `[]`) is `true`

Classes with data members have q-values based on their fields. These q-values are of the form `{` v_1, v_2, ..., v_n `}` where v_i ranges over the q-values of the ith data member of the class. The following q-match rules apply:

- q-match(`{` v_1, v_2, ..., v_n `}`, *wildcard-pattern*) is `true`
- q-match(`{` v_1, v_2, ..., v_n `}`, *binding-pattern*) is `true`
- q-match(`{` v_1, v_2, ..., v_n `}`, `[` pat$_1$, pat$_2$, ..., pat$_n$ `]`) is `true` if q-match(v_i, pat$_i$)=`true` for every i, and `false` otherwise.

Expression patterns q-match classes only if the class type is said to have *deep derived equality*.

- q-match(`{` v_1, v_2, ..., v_n `}`, *expression-pattern*) is `true` if `{` v_1, v_2, ..., v_n `}` has the same value as the *expression-pattern* and the class being matched has *deep derived equality*.

A class `C` has *deep derived equality* if the following conditions are met:

1. `C` has a defaulted `operator==`.
2. All of `C`'s fields are `std::nullptr_t`, `bool`, or are classes having *deep derived equality*.

Finally, classes that are polymorphic have the additional *q*-match rule:

- *q*-match(*v*, < *type* > *pattern*) is `false`

4.6 Tuple-like types

Tuple-like types are those that opt-in to structured binding syntax by specializing `std::tuple_size`, `std::get`, and `std::tuple_element`. Like classes, tuple-like types `T` have *q*-values of the form `{` v_1, v_2, ..., v_n `}`, but where v_i ranges over the *q*-values of `std::tuple_element<`*i*`, T>::type`.

The *q*-match rules are identical to those with classes except *q*-match always returns false for *expression-patterns*.

- *q*-match(`{` v_1, v_2, ..., v_n `}`, *expression-pattern*) is `false` if the class being matched is a tuple-like type.

4.7 Variant-like types

Variant-like types are those that opt-in to pattern matching syntax by specializing `std::variant_size`, `std::holds_alternative`, `std::get`, and `std::variant_alternative`. *q*-values of variant-like types `V` are of the form (*i*, *v*) where 0 <= *i* < `std::variant_size<V>::value` and *v* ranges over the *q*-values of `std::variant_alternative<`*i*`, V>::type`.

Our matching rules are as follows:

- *q*-match((*i*, *v*), *wildcard-pattern*) is `true`
- *q*-match((*i*, *v*), *binding-pattern*) is `true`
- *q*-match((*i*, *v*), *expression-pattern*) where *epat* is an *expression-pattern* is `true` if and only if
  1. the expression evaluates to a value *w* where `std::holds_alternative<`*i*`>(`*w*`) = true`,
  2. *q*-match(*v*, `std::get<`*i*`>(`*w*`)`) = `true`,
  3. the `std::holds_alternative<`*i*`>` specialization is `constexpr`, and
  4. the `std::get<`*i*`>` specialization is `constexpr`.
- *q*-match((*i*, *v*), < auto > *pat*) is `true` if and only if *q*-match(*v*, *pat*) is true.
- *q*-match((*i*, *v*), < *concept* > *pat*) is `true` if and only if `std::variant_alternative<i,V>::type` satisfies the concept and *q*-match(*v*, *pat*) is true.
- *q*-match((*i*, *v*), < *type* > *pat*) is `true` if and only if `std::variant_alternative<i,V>::type` is the same as *type* and *q*-match(*v*, *pat*) is true.
- *q*-match((*i*, *v*), < *constant-expression* > *pat*) is `true` if and only if the expression evaluates to *i* and *q*-match(*v*, *pat*) is true.

4.8 Any-like types

Any-like types are those that opt-in to pattern matching syntax by specializing the `any_cast` function template. All such types `A` have a single *q*-value ϵ with the following *q*-match rules.

- *q*-match(ϵ, *wildcard-pattern*) is `true`
- *q*-match(ϵ, *binding-pattern*) is `true`
- *q*-match(ϵ, *expression-pattern*) is `false`
- *q*-match(ϵ, < *type* > *pattern*) is `false`

5 Patterns for Invariant Checking

Consider this example program:

```
enum Color{ RED, GREEN, BLUE };

int main() {
  // Assuming an 'enumerators' reflection facility
  std::for_each(enumerators<Color>(), [](Color c) {
    std::cout << int(c) << " = "
      << inspect(c){ case RED   => "Red";
                     case GREEN => "Green";
                     case BLUE  => "Blue";
                   }
      << std::endl;
  });
  std::cout << "\nSelect a color: " << std::flush;
  Color c;
  std::cin >> c;

  inspect(c) -> std::ostream& {
    case RED   => std::cout << "(1,0,0)" << std::endl;
    case GREEN => std::cout << "(0,1,0)" << std::endl;
    __ => std::cerr << "Bad selection!" << std::endl;
  };
}
```

Note that the second inspect statement has a bug: the BLUE case isn't handled. Our exhaustiveness checking algorithm will not catch this case because of the presence of the wildcard arm.

It would be nice to annotate the wildcard arm to somehow indicate that it is an error handling arm and should not impact exhaustiveness checking.

One way to do this is to add a guard since the presence of a guard excludes an arm from exhaustiveness checking. The following code will produce a compilation error as desired, indicating the missing `BLUE` case:

```
// OPTION 1

inspect(c) -> std::ostream& {
  case RED   : std::cout << "(1,0,0)" << std::endl;
  case GREEN : std::cout << "(0,1,0)" << std::endl;
  __ if(true) : std::cerr << "Bad selection!" << std::endl;
};
```

While this works, the `if(true)` syntax doesn't capture the intent very well. If we want special syntax cheaply we could allow the condition in the guard to be empty:

```
// OPTION 2

inspect(c) -> std::ostream& {
  case RED   : std::cout << "(1,0,0)" << std::endl;
  case GREEN : std::cout << "(0,1,0)" << std::endl;
  __ if() : std::cerr << "Bad selection!" << std::endl;
};
```

Alternatively, we could use some kind of context-sensitive keyword (or annotation) to more directly indicate this is an exceptional case:

```
// OPTION 3

inspect(c) -> std::ostream& {
  case RED   : std::cout << "(1,0,0)" << std::endl;
  case GREEN : std::cout << "(0,1,0)" << std::endl;
  __ exceptional : std::cerr << "Bad selection!" << std::endl;
};
```

Our preference is option 1 for core pattern matching. It demonstrates that we do not need special syntax or additional complexity to handle this use case right now and it isn't clear that this use case will be a prevalent one. Options 2 and 3 could be added to the language later on if we see that the use case is more widespread.

6 Analysis of alternatives

Compilation errors due to inexhaustive patterns is not a new idea, although it is relatively rare. The Rust programming language [RustLang] is a modern example.

6.1 Use warnings instead of errors

Most languages with pattern matching depend on compiler-provided warnings to discover bugs due to lacking pattern coverage and, up until this point, this was our suggested approach. While developers could reap most of the benefits of this proposal through use of an "error on warn" flag to their compilers, the advantages would be seen primarily by large companies with uniform flag usage and advanced engineers who know enough to turn this on. Unfortunately, this leaves the developers who are most likely to introduce these bugs, beginners to either programming or C++, without adequate protection. By requiring exhaustiveness checking we enhance C++'s image as language that provides safe constructs while maintaining a low performance overhead.

6.2 Require pedantically exhaustive `enum` inspection

One design alternative we considered is to require inspection of `enum` types to include an exhaustive pattern. The benefit of that approach would be that the `inspect` construct matches more closely the precise language semantics of `enum`.

While that argument is compelling, the opportunity to detect at compile-time one of the most common bugs observed in practice (missing enumerators in a `switch`) is drastically more interesting. This feature alone is expected to free up, in aggregate, vast monetary and personnel resources that would otherwise be spent on issues resulting from these bugs.

7 Conclusion

In this paper we have presented a design for compile-time exhaustiveness checking for a C++ pattern matching feature. This included an example-based tutorial, a precise specification, and consideration of related topics. In our opinion, the benefits of such an enhancement significantly outweigh the drawbacks.

8 References

[maranget_2007] Luc Maranget. 2007. Warnings for pattern matching. *Journal of Functional Programming* 17, (2007), 387–421.
https://github.com/camio/exhaustiveness_checking

[P0627R3] Melissa Mears. 2018. Function to mark unreachable code.
https://wg21.link/p0627r3

[P1371R2] Sergei Murzin, Michael Park, David Sankel, Dan Sarginson. 2020. Pattern Matching.
https://wg21.link/p1371r2

[RefImpl] David Sankel. Exhaustiveness Checking Reference Implementation.
https://github.com/camio/exhaustiveness_checking

[RustLang] Rust Website.
https://www.rust-lang.org/