Contract assertions, the noexcept operator, and deduced exception specifications

Timur Doumler

Document #: P3113R0
Date: 2024-02-01
Audience: SG21 (Contracts)

Contracts MVP – The Final Boss

Every C++ expression is:

- **either potentially-throwing**
- **or not potentially-throwing**

[except.spec]

6 An expression *E* is *potentially-throwing* if

(6.1) — *E* is a function call whose *postfix-expression* has a function type, or a pointer-to-function type, with a potentially-throwing exception specification, or

(6.2) — *E* implicitly invokes a function (such as an overloaded operator, an allocation function in a *new-expression*, a constructor for a function argument, or a destructor if *E* is a full-expression) that has a potentially-throwing exception specification, or

(6.3) — *E* is a *throw-expression* ([expr.throw]), or

(6.4) — *E* is a `dynamic_cast` expression that casts to a reference type and requires a runtime check ([expr.dynamic.cast]), or

(6.5) — *E* is a `typeid` expression applied to a (possibly parenthesized) built-in unary `*` operator applied to a pointer to a polymorphic class type ([expr.typeid]), or

(6.6) — any of the immediate subexpressions of *E* is potentially-throwing.

Every C++ expression is:

- either potentially-throwing
- or not potentially-throwing

It matters in two situations:

- result of `noexcept(expr)`
- whether defaulted special member functions are `noexcept` (exception specification is deduced by the compiler)

Are contract assertions potentially-throwing?

Are contract assertions potentially-throwing?

It doesn't matter for pre and post:

```
noexcept(pre(f()))      // ill-formed (pre/post are not expressions)

struct X
{
  X() pre(f()) = default;  // ill-formed (consensus in Kona)
}
```

Are contract assertions potentially-throwing?

It matters for contract_assert:

```
noexcept(contract_assert(false));  // true or false?
```

Are contract assertions potentially-throwing?

It matters for contract_assert:

```
noexcept(contract_assert(false));  // true or false?

noexcept((contract_assert(x.a()), x.b())); // true or false?
```

Are contract assertions potentially-throwing?

It matters for contract_assert:

```
noexcept(contract_assert(false));  // true or false?

noexcept((contract_assert(x.a()), x.b())); // true or false?

class B {
  int i = (contract_assert(true), 17);      // default member initialiser
  B(int j = (contract_assert(true), 34));  // default argument
};
class D : B {};   // noexcept(D{}) true or false ?
```

Fact: `contract_assert(x)` can throw an exception.

Fact: `contract_assert(x)` can throw an exception.

```
#include <contracts>
using namespace std::contracts;

handle_contract_violation(const contract_violation&) {
  throw 666;
}

int main() {
  contract_assert(false); // this statement throws an exception
}
```

Design principle: "Concepts do not see Contracts" (P2932)

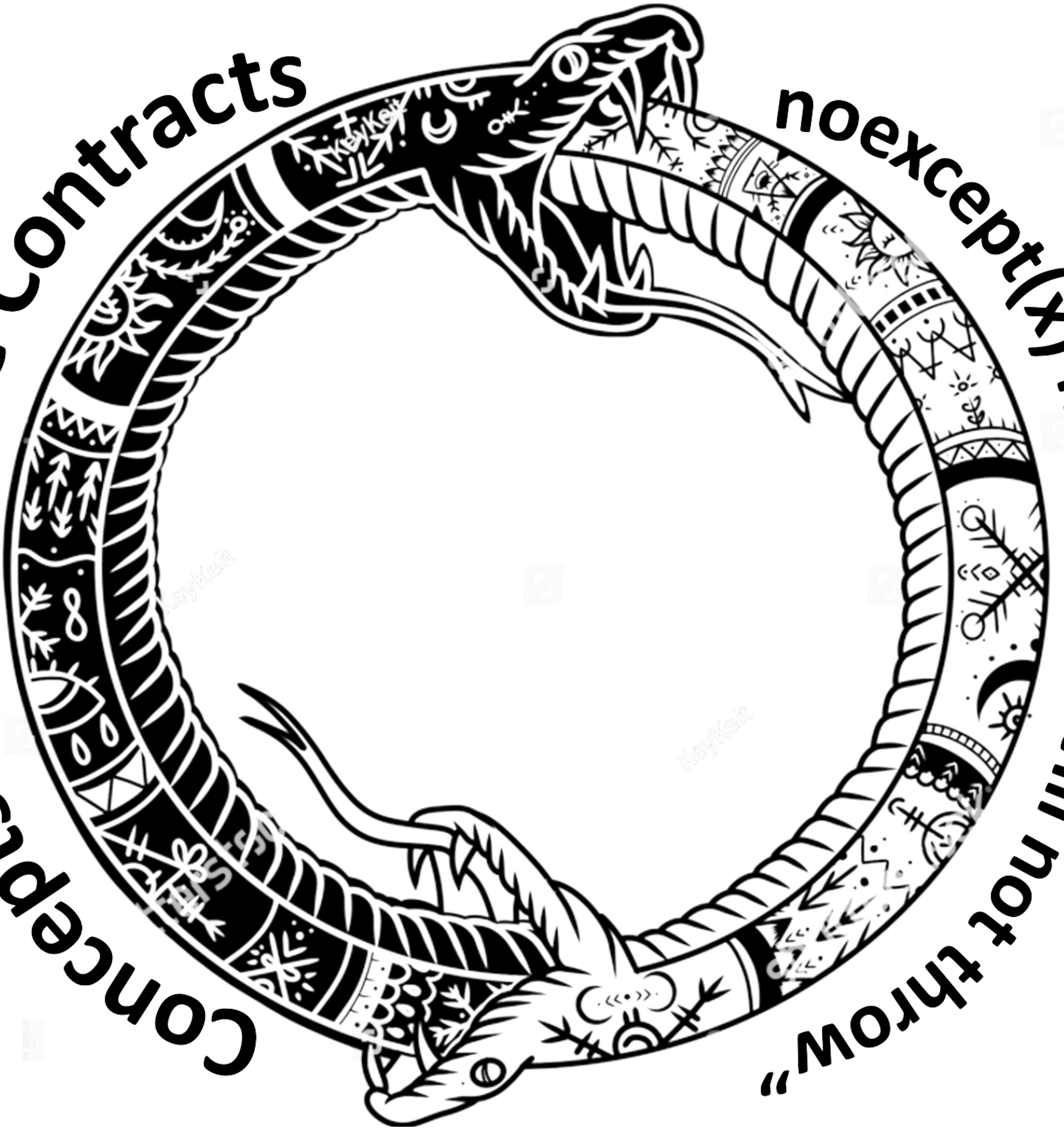
Adding a contract annotation to an existing program must <u>never</u> alter the compile-time semantics of the program:

- **Whether a concept or constraint is satisfied**
- **SFINAE**
- **Overload resolution**
- **which branch is taken by `if constexpr`**
- **the result of operator `noexcept`**
- **...**

Design principle: "Concepts do not see Contracts" (P2932)

Adding a contract annotation to an existing program must <u>never</u> alter the compile-time semantics of the program:

- **Whether a concept or constraint is satisfied**
- **SFINAE**
- **Overload resolution**
- **which branch is taken by `if constexpr`**
- **the result of operator `noexcept`**
- **...**



Concepts do not see Contracts

noexcept(x) means “x will not throw”

Options

1. **Make `contract_assert(x)` potentially-throwing (P2969R0, option 3.1)**

```
noexcept(contract_assert(false));  // -> false

noexcept((contract_assert(x.a()), x.b())); // -> false

class B {
  int i = (contract_assert(true), 17);     // default member initialiser
  B(int j = (contract_assert(true), 34));  // default argument
};
class D : B {};   // noexcept(D{}) -> false
```

Options

2. **Make `contract_assert(x)` not potentially-throwing**
   ~ "operator `noexcept` assumes no contract violations happen"
   (P2969R0, option 3.2)

```
noexcept(contract_assert(false));  // -> true

noexcept((contract_assert(x.a()), x.b())); // -> true

class B {
  int i = (contract_assert(true), 17);      // default member initialiser
  B(int j = (contract_assert(true), 34));  // default argument
};
class D : B {};   // noexcept(D{}) -> true
```

Options

3. **When determining if a set of expressions is potentially-throwing, CCAs are not considered. If there are no non-CCA expressions the query is ill-formed. (P2932R2, proposal 7A)**

```
noexcept(contract_assert(false));  // -> ill-formed, like noexcept()

noexcept((contract_assert(x.a()), x.b())); // -> true

class B {
  int i = (contract_assert(true), 17);     // default member initialiser
  B(int j = (contract_assert(true), 34));  // default argument
};
class D : B {};   // noexcept(D{}) -> true
```

Options

4. **Allow both options, via an extra annotation (P2969R0, option 3.3)**

```
int f(int i) pre (i > 0);          // potentially-throwing contract check
int g(int i) pre noexcept (i > 0); // non-throwing contract check
```

Options

4. **Allow both options, via an extra annotation (P2969R0, option 3.3)**

```
int f(int i) pre (i > 0);          // potentially-throwing contract check
int g(int i) pre noexcept (i > 0); // non-throwing contract check
```

→ not proposed; exact syntax and semantics unclear, no paper, default case still violates Concepts do not see Contracts

Options

5. **Allow erroneously thrown exceptions to escape deduced non-throwing exception specifications (P2969R0, option 3.4)**

Options

5. **Allow erroneously thrown exceptions to escape deduced non-throwing exception specifications (P2969R0, option 3.4)**

   → not proposed; we have SG21 consensus to not do this:

```
Poll, 2023-05-18
Throwing an exception from a contract violation handler shall invoke the usual
exception semantics: stack unwinding occurs, and if a `noexcept` barrier is
encountered during unwinding, std::terminate is called, as proposed in P2811.

SF F N A SA
10 7 2 0 0

Result: Consensus
```

Options

6. **`contract_assert` is neither potentially-throwing nor not potentially-throwing. Any use of `contract_assert` in a situation where this must be determined is ill-formed. (P2969R0, option 3.5; P2832R2, proposal 7B)**

Options

6. `contract_assert` is neither potentially-throwing nor not potentially-throwing. Any use of `contract_assert` in a situation where this must be determined is ill-formed. (P2969R0, option 3.5; P2832R2, proposal 7B)
    a. Make `contract_assert` a statement, not an expression
    b. Make it ill-formed if a `contract_assert` appears as a subexpression of the operand of `noexcept` or while deducing an exception specification
    c. Make it ill-formed if a `contract_assert` appears as a subexpression of the operand of `noexcept` or while deducing an exception specification, and no other subexpression is potentially-throwing

Options

6. **`contract_assert` is neither potentially-throwing nor not potentially-throwing. Any use of `contract_assert` in a situation where this must be determined is ill-formed. (P2969R0, option 3.5; P2832R2, proposal 7B)**
   a. **Make `contract_assert` a statement, not an expression**
   b. ~~Make it ill-formed if a `contract_assert` appears as a subexpression of the operand of `noexcept` or while deducing an exception specification~~ **→ not proposed**
   c. **Make it ill-formed if a `contract_assert` appears as a subexpression of the operand of `noexcept` or while deducing an exception specification, and no other subexpression is potentially-throwing**

Options

6a. Make `contract_assert` a statement, not an expression

```
noexcept(contract_assert(false));          // -> ill-formed
noexcept((contract_assert(x.a()), x.b())); // -> ill-formed
class B {
  int i = (contract_assert(true), 17);     // -> ill-formed
  B(int j = (contract_assert(true), 34));  // -> ill-formed
};
```

Options

6c. Make it ill-formed if a `contract_assert` appears as a subexpression of the operand of `noexcept` or while deducing an exception specification, and no other subexpression is potentially-throwing

```
noexcept(contract_assert(false));                // -> ill-formed
noexcept((contract_assert(false), true));        // -> ill-formed
noexcept((contract_assert(false), throw 666)); // -> OK, returns false
```

Options

7. **Address the issue via coding guidelines or diagnostics**
   - **with `contract_assert` potentially-throwing or not potentially-throwing**
   - **with diagnostics being normative, recommended practice, or QoI**

Options

7. Address the issue via coding guidelines or diagnostics
   - with `contract_assert` potentially-throwing or not potentially-throwing
   - with diagnostics being normative, recommended practice, or QoI

→ not proposed; not really a solution as we still need to define the normative behaviour

Options

8. **Make `contract_assert(x)` not potentially-throwing and the contract-violation handler always `noexcept` (P2969R0, option 3.7: "Remove support for throwing contract-violation handlers").**

Viable options – Overview

1. Make `contract_assert(x)` potentially-throwing
2. Make `contract_assert(x)` not potentially-throwing
3. When determining if a set of expressions is potentially-throwing, `contract_assert` is not considered; if there are no expressions other than `contract_assert`, the query is ill-formed

6a. Make `contract_assert` a statement rather than an expression

6c. `contract_assert` is neither potentially-throwing nor not potentially-throwing; if a `contract_assert` appears as a subexpression of the operand of `noexcept` or while deducing an exception specification, and no other subexpression is potentially-throwing, the program is ill-formed.

8. Make `contract_assert(x)` not potentially-throwing and the contract-violation handler always `noexcept` (= remove throwing violation handlers)

Instead of talking about solutions, let's talk about the underlying design goals and principles!

The Swan, The Pike, and The Crab – Fable by Ivan Krylov, 1814

Desiderata for this problem:

- **Maximises teachability**
- **Minimises chance of standardising something suboptimal**
- **Concepts do not see Contracts (~ adding a contract assertion cannot silently switch behaviour of surrounding code)**
- **Maximises consistency with existing language**
- **Minimises cognitive dissonance with current understanding that noexcept(x) means "x will not throw"**
- **Minimises making code ill-formed when adding Contracts to it**
- **Minimises interaction between Contracts and exception handling (makes them orthogonal)**
- **Minimises ability to write useless code**
- **Maximises backward-compatible evolution of the language**
- **Does not inject new code paths into existing code**
- **Maximises compatibility with code bases that compile with exceptions turned off or have coding guidelines against using exceptions**
- **Does not disenfranchise important use cases**
- **Allows effective negative testing**
- **Allows recovery (non-terminating non-continuing violation handling)**

Desiderata for this problem:

- Maximises teachability
- Minimises chance of standardising something suboptimal
- **Concepts do not see Contracts (~ adding a contract assertion cannot silently switch behaviour of surrounding code)**
- Maximises consistency with existing language
- **Minimises cognitive dissonance with current understanding that noexcept(x) means "x will not throw"**
- **Minimises making code ill-formed when adding Contracts to it**
- Minimises interaction between Contracts and exception handling (makes them orthogonal)
- Minimises ability to write useless code
- Maximises backward-compatible evolution of the language
- Does not inject new code paths into existing code
- Maximises compatibility with code bases that compile with exceptions turned off or have coding guidelines against using exceptions
- Does not disenfranchise important use cases
- Allows effective negative testing
- **Allows recovery (non-terminating non-continuing violation handling)**

Desiderata for this problem:

- Maximises teachability
- Minimises chance of standardising something suboptimal
- **Concepts do not see Contracts (~ adding a contract assertion cannot silently switch behaviour of surrounding code)**
- Maximises consistency with existing language
- **Minimises cognitive dissonance with current understanding that noexcept(x) means "x will not throw"**
- **Minimises making code ill-formed when adding Contracts to it**
- Minimises interaction between Contracts and exception handling (makes them orthogonal)
- Minimises ability to write useless code
- Maximises backward-compatible evolution of the la
- s into existing
- e bases
- ned off or
- using
- ex
- Does not disenfranchise important use cases
- Allows effective negative testing
- **Allows recovery (non-terminating non-continuing violation handling)**

These are the four properties which were referred to with words like "this is imperative", "people won't use Contracts", "I will vote against Contracts", "over my dead body", etc.

	1. contract_assert is potentially-throwing	2. contract_assert is not potentially-throwing	3. contract_assert is not considered when determining exception spec	6a. Make contract_assert a statement, not an expression	6c. Determining exception spec of contract_assert is ill-formed	8. Remove support for throwing contract-violation handlers
Concepts do not see Contracts	❌	✅	✅	✅	✅	✅
noexcept(x) means "x will not throw"	✅	❌	❌	✅	⚠️	✅
Adding Contracts cannot make client code ill-formed	✅	✅	✅	✅	❌	✅
Allows recovery (non-terminating non-continuing violation handling)	✅	⚠️	⚠️	✅	✅	❌

	1. contract_assert is potentially-throwing	2. contract_assert is not potentially-throwing	3. contract_assert is not considered when determining exception spec	6a. Make contract_assert a statement, not an expression	6c. Determining exception spec of contract_assert is ill-formed	8. Remove support for throwing contract-violation handlers
Concepts do not see Contracts	✗	✓	✓	✓	✓	✓
noexcept(x) means "x will not throw"	✓	✗	✗	✓	⚠	✓
Adding Contracts cannot make client code ill-formed	✓	✓	✓	✓	[illegible]	✓
Allows recovery (non-terminating non-continuing violation handling)	✓	⚠	⚠	✓	[illegible]	✗

Unlike options 2 and 3, this does not subvert the meaning of `noexcept(x)`, but it creates a new category of expressions for which `noexcept(x)` is ill-formed

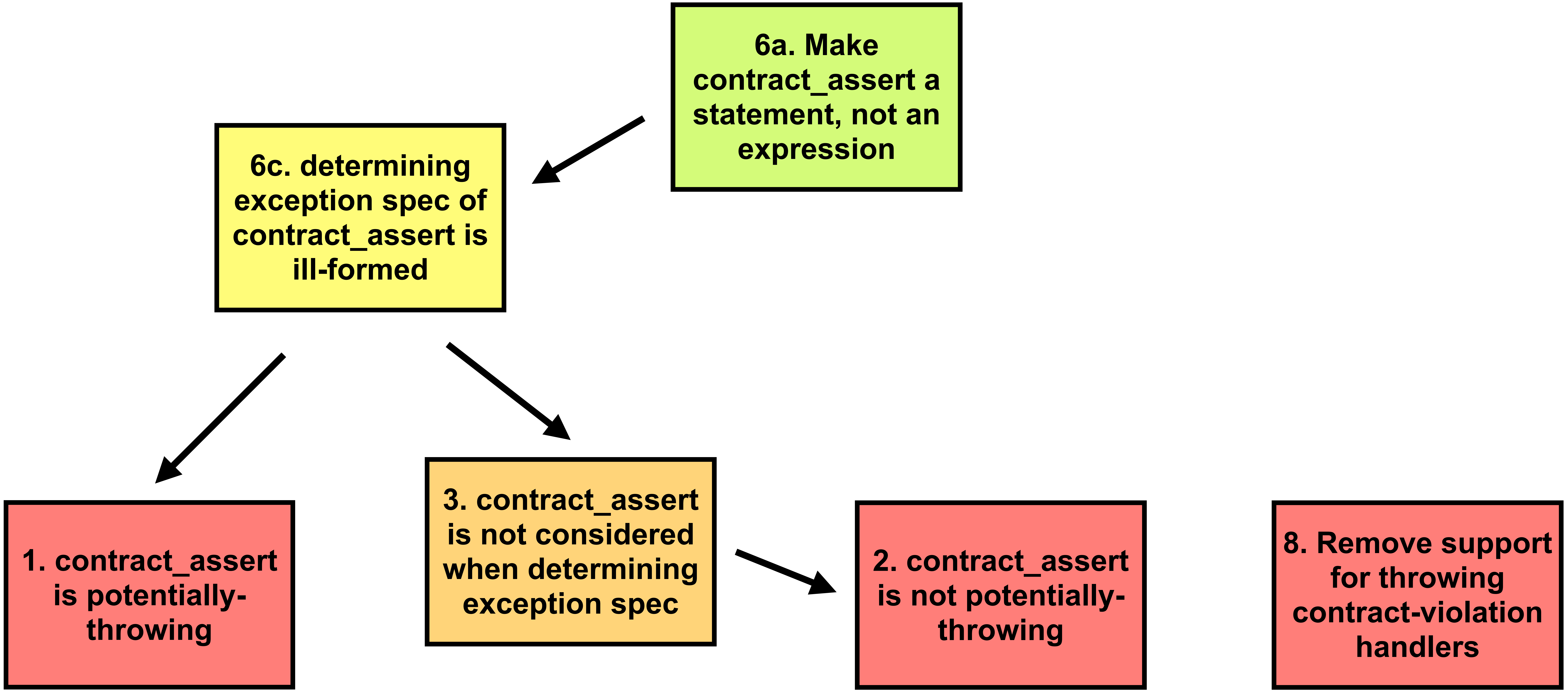
	1. contract_assert is potentially-throwing	2. contract_assert is not potentially-throwing	3. contract_assert is not considered when determining exception spec	6a. Make contract_assert a statement, not an expression	6c. Determining exception spec of contract_assert is ill-formed	8. Remove support for throwing contract-violation handlers
Concepts do not see Contracts	❌			✅	✅	✅
noexcept(x) means "x will not throw"	✅			✅	⚠️	✅
Adding Contracts cannot make client code ill-formed	✅			✅	❌	✅
Allows recovery (non-terminating non-continuing violation handling)	✅	⚠️	⚠️	✅	✅	❌

Treating contract_assert as not potentially-throwing lands you in the `noexcept(true)` branch of algorithms such as `push_back`; throwing an exception in such a place is likely to lead to UB, reducing the usefulness of a throwing contract-violation handler.

Desiderata for this problem:

- Maximises teachability
- Minimises chance of standardising something suboptimal
- Concepts do not see Contracts (~ adding a contract assertion cannot silently switch behaviour of surrounding code)
- Maximises consistency with existing language
- Minimises cognitive dissonance with current understanding that noexcept(x) means "x will not throw"
- Minimises making code ill-formed when adding Contracts to it
- Minimises interaction between Contracts and exception handling (makes them orthogonal)
- Minimises ability to write useless code
- **Maximises backward-compatible evolution of the language**
- Does not inject new code paths into existing code
- Maximises compatibility with code bases that compile with exceptions turned off or have coding guidelines against using exceptions
- Does not disenfranchise important use cases
- Allows effective negative testing
- Allows recovery (non-terminating non-continuing violation handling)

Possible language evolution paths


6a. Make contract_assert a statement, not an expression
6c. determining exception spec of contract_assert is ill-formed
1. contract_assert is potentially-throwing
3. contract_assert is not considered when determining exception spec
2. contract_assert is not potentially-throwing
8. Remove support for throwing contract-violation handlers