A New Taxonomy for Contracts

Document: P3977R0
Date: January 22, 2026
Audience: WG21
Reply to: Oliver J. Rosten (oliver.rosten@gmail.com)

Abstract

Historically, contracts have been categorized according to whether they are wide or narrow. A wide contract does not specify any undefined behaviour, whereas a narrow contract does. This provided and continues to provide a useful way of reasoning about behaviour. However, there are circumstances in which this categorization may be counterintuitive or even inapplicable, as has been brought into sharp focus by the current discussions around non-ignorable contracts. A richer classification is therefore proposed, providing amongst other things, a rigorous understanding of what it means for a function to have a precondition (suggestive of a narrow contract) that is always enforced (suggestive of a wide contract).

CONTENTS

I. INTRODUCTION

How do we reason about what a function does? Notwithstanding the axiom that naming is hard, ideally a clue will come from its signature. Two examples, which will feature prominently in this paper, nicely summarize their intent[1] :

```
void abort();
float sqrt(float);
```

However, despite its simplicity, the latter immediately presents us with a particular problem. Assuming we are working with the real numbers, `sqrt` is a map from the semi-positive reals to the semi-positive reals. But here mathematical purity collides with the reality of programming: the domain of the mathematical[2] *sqrt* is not the same as the domain of `float sqrt(float)`. Quite apart from the limited precision of floating-point, a `float` can of course be negative and it may also include NaN. So how do we reason about the behaviour of this function in totality?

One option is to look at the implementation and try to divine what it does. However, not only may this be hard to do in practice, it also presupposes that the implementation does not have any bugs. Furthermore, without additional information, there will be no way to tell which implemented behaviours are set in stone and which are not. For example, if we deduce that an implementation of `sqrt` always returns -1 for negative input, is this guaranteed by the function's author to hold for perpetuity, or are they within their rights to change it to -2, without giving notice?

Indeed, it is worth pointing out that it is entirely conforming for `std::vector::operator[]` and `std::vector::at` to have exactly the same implementation. The difference is that, for the latter, the implementation *must* throw `std::out_of_range` if the index is out of range, whereas, for the former, this would be a (dubious) implementation choice that could be changed. The C++ standard is wise to this: `std::vector::operator[]` is not `noexcept` yet 'throws nothing'. The latter is a statement about the behaviour when called in-contract. Out of contract, the behaviour is not defined by the standard, and implementers can do whatever they want, including throw.

The best hope to understand what a function is supposed to do is to appeal to its documented specification, known as the plain-language contract. This comes with the tacit assumption that not merely has the specification been documented but documented correctly. For the cases of well-named functions, it is arguably true that the name itself suffices to state at least part of the plain-language contract. Indeed, for both standard library functions `std::abort` and `std::sqrt` it is clear what they primarily do. However, in both cases there is more to the story. Nevertheless, we are on the right track: a central tenet of this paper is the need to formalize what

[1] We use e.g. `float sqrt(float)` to talk about some hypothetical function; anything belonging to the C++ standard library will be qualified with `std`.

[2] Throughout this paper, when we talk of a function without further qualification, we mean in the programming sense.

we mean by the primary behaviour of a function, as will be done in section IV.

A crucial aspect of this is that a necessary *but not sufficient* condition for primary behaviour to be invoked is that all preconditions are satisfied. This distinguishes primary behaviour from essential behaviour as defined in [P2861R0, P2899R1]: essential behaviour follows from a function being invoked in-contract. In other words, satisfying all preconditions is both necessary *and* sufficient to elicit essential behaviour.

To understand why it is useful to define primary behaviour differently from essential behaviour, return to the case of what `float sqrt(float)` does for negative inputs. Whatever it turns out to do, this is not the primary purpose of the function, which is to compute actual square roots. If the behaviour for negative inputs is *specified*—say to return NaN—then this is part of the essential behaviour, according to the definition of [P2861R0, P2899R1]. Being able to isolate the primary behaviour of a function, which may only be part of the essential behaviour, will turn out to be extremely useful, hence the new terminology.

Up until now, the classification of contracts has been binary: they are either wide or narrow. To quote [N3248]:

> A wide contract for a function or operation does not specify any undefined behavior. Such a contract has no preconditions: A function with a wide contract places no additional constraints on its arguments, on any object state, nor on any external global state...
>
> A narrow contract is a contract which is not wide. Narrow contracts for a functions or operations result in undefined behavior when called in a manner that violates the documented contract.

Note that, in this context, undefined behaviour means behaviour specified by the plain-language contract to be undefined. This could be explicit "this function's behaviour for negative inputs is undefined". However, tacitly, we may need to accept that there are situations where the plain-language contract only implicitly specifies undefined behaviour. Consider encountering a function, `float sqrt(float)`, with no documentation. Given that this function has a commonly understood meaning due to its mathematical roots, it is reasonably to assume, for pragmatic purposes, that if it is fed a finite non-negative number it will compute the square root (to some precision). What is does for negative numbers, NaN or infinity is anyone's guess and so, realistically, this behaviour should be taken as undefined behaviour according to the (incomplete) plain-language contract.

It is important to emphasise that this does not imply that the implementation of the function is necessarily badly behaved, in the sense of summoning Nasal Demons or, more prosaically, dereferencing a null pointer. Undefined behaviour in this context is no more or less than a statement that the behaviour is not defined by the plain-language contract. However, this begs the question as to how to talk about situations where a function with no behaviour left unspecified by the plain-language contract accidentally dereferences a null pointer.

In the definitions given by [N3248], quoted above, 'specified' serves to distinguish behaviour not defined by the plain-language contract from undefined behaviour that incidentally arises due to a buggy implementation. It is almost always the case in this paper that we talk of undefined behaviour purely in the context of the plain-language contract. Another way to reason about such specified undefined behaviour is that the implementor is free to choose this behaviour however they see fit and, moreover, are perfectly free to change it: clients cannot rely upon behaviours which are not specified.

Historically, the wide/narrow classification has proven very helpful, since whether or not a function is specified to have undefined behaviour is often a consideration of particular importance. However, there are cases where this classification is either counterintuitive or inapplicable.

To illustrate this, let us first a consider an implementation of `float sqrt(float)` for which the plain-language contract guarantees that it will abort the program[3] if fed a negative number or NaN. There is no specified undefined behaviour: therefore the contract is wide. However, this may be surprising to some. Calling this function with a negative number is almost certainly a bug, and will likely cause the program to crash. Generally, we talk of violating preconditions as being indicative of a bug, but there are no preconditions here. This suggests that it would be useful to be able to sub-categorize wide contracts for situations where it is desirable to emphasise certain guarantees or lack thereof. There is a qualitative difference between a wide contract which specifies termination in atypical situations, versus the wide contract for something like addition of unsigned integers, for which there are no surprises (at least for anyone familiar with a clock).

A scenario that is more fundamentally problematic for the current classification is where we have a precondition which is unconditionally hardened.[4] This is particularly topical, following the recent publication of [P3911R1]. Assuming a future labels syntax along the lines of [P3400R2] consider:

```
float sqrt(float)
  pre <quick_enforce> (x >= 0);
```

[3] This is actually not as straightforward to guarantee as it sounds, since `std::abort` may not actually abort the program if a strange signal handler is installed. This will be discussed later in the paper but distracts from the main point of this example.

[4] The notion of hardened preconditions was introduced into the C++ standard by [P3471R2]. We discuss in section II why it is useful to further talk of unconditional hardening.

This function has a precondition that is always quick-enforced; as such, there is no undefined behaviour.[5] Therefore, according to the first part of the definition of a wide contract [N3248] 'A wide contract for a function or operation does not specify any undefined behavior', we conclude that this function has a wide contract. However, the definition of a wide contract goes on to say that 'Such a contract has no preconditions'. Since we evidently do have a precondition we seem to have a contradiction here; is the contract wide or not?!

We can go even further. Consider the following entertaining—if utterly perverse—declaration:

```
void abort()
  pre <quick_enforce> (false);
```

To emphasise: this paper is absolutely not advocating that people ever write code like this. Indeed, quite the opposite! But to discourage such things requires that we name them. The first question to ask is whether this function has a wide contract or a narrow one. It is apparent that it has a precondition that is always violated. Surely such a function must have the narrowest of contracts. And yet, the guaranteed enforcement of the precondition implements the function's desired behaviour—to abort the program—without any undefined behaviour. So does this function have a wide contract?! The premise of this paper is that, previously, we have not had the right words to describe situations such as this and others: the existing classification of contracts is incomplete.

The essence to resolving conundrums such as this revolves around classifying the *secondary* behaviour of a function. If the primary behaviour of a function is, roughly speaking, what it is supposed to do, then the secondary behaviour is what it does when invoked in a way that does not elicit the primary behaviour. For example, the secondary behaviour of `float sqrt(float)` is whatever it does when called with a negative number (and perhaps NaN).

As mentioned above, satisfaction of all preconditions is necessary but not sufficient to elicit the primary behaviour of a function. Therefore, it follows that violating at least one precondition is sufficient *but not necessary* to elicit the secondary behaviour. Again, consider a version of `sqrt` which is guaranteed to return NaN for negative inputs. This behaviour is specified, and so feeding the function a negative number does not violate any preconditions. However, the behaviour is secondary.

With the notion of secondary behaviour established, at least informally, let us call a contract faithful[6] if the *secondary behaviour* is guaranteed to

1. Return control to the caller (in a general sense, so throwing an exception is included);
2. Produce a behaviour distinct from the primary behaviour.

Note that this places no restrictions whatsoever on the primary behaviour. If the primary behaviour is such that control is not returned to the caller, then so be it. Immediately, we see that a notion of faithfulness gives us a tool to start talking about the cases listed above:

1. A version of `float sqrt(float)` guaranteed to abort for negative inputs has secondary behaviour which does not return control to the caller and so is not faithful.
2. A version of `float sqrt(float)` with a precondition has secondary behaviour which does not guarantee return control to the caller and so is not faithful.
3. A vanilla version of `void abort()` has a faithful contract since its primary behaviour is respected and there is no secondary behaviour. To emphasise: the fact that the primary behaviour of `abort` does not return to the caller in no way impacts whether the contract is considered faithful. The faithfulness or otherwise follows solely from the secondary behaviour.
4. The pathological (in the mathematical sense)

   ```
   void abort()
     pre <quick_enforce> (false);
   ```

   does not have any primary behaviour since its precondition is always violated. Therefore, the secondary behaviour is guaranteed not to return control to the caller and so the contract is not faithful.

As we will see, 'faithful' is not actually a fundamental category which we propose, though it is used to describe the union of two of them. The precise details are given in section V but here we provide an overview, as shown in table I, together with a visual representation of how to reason about them in figure 1. Wide contracts are sub-divided into 4 categories. Two of these combine to form faithful contracts. Of the remaining two

1. 'Lossy' is defined to include the case of constrained but not fully specified behaviour—i.e. unspecified behaviour;
2. 'Disconnecting' covers cases where secondary behaviour does not guarantee return of control to the caller.

Unconditionally hardened contracts leverage existing terminology (hardening) to name cases which, as alluded to above, cannot be classified as other wide or narrow according to the definition of [N3248]. It is interesting to note that disconnecting contracts and unconditionally hardened contracts have considerable similarity; the

[5] Glossing over the issue of what, if anything, the plain-language contract has to say; we will return to this in section III.

[6] This is loosely inspired by the mathematical notion of a faithful representation.

main difference is that, in the disconnecting case, secondary behaviour not guaranteeing return of control to the caller is part of the essential behaviour (the contract is wide), whereas in the unconditionally hardened case this behaviour is specified through guarantees if preconditions are violated.

Narrow contracts primarily retain their previous meaning in the sense that there is specified undefined behaviour, and then there is a final classification to deal with edge cases such as the perverse implementation of `abort` given above.

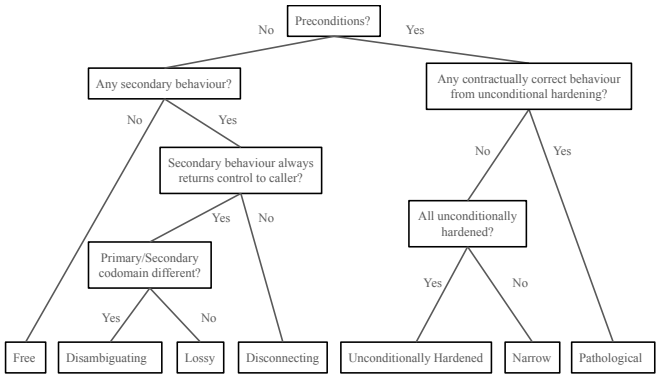

FIG. 1. How to reason about categorization in the proposed taxonomy.

We conclude the introduction with an important point about how to reason about the category to which a particular plain-language contract should be assigned. The deciding consideration is the guarantees that are given: anything that is not strictly guaranteed cannot be assumed and so cannot be used to justify reassigning to a 'wider' category. Ultimately, this comes down to what is specified. Going back to the previous example of `vector::operator[]`: just because a particular implementation *happens* to perform a bounds check does not mean that this function magically has a wide contract. It is narrow as a consequence of the plain-language contract, independently of the chosen implementation.

II. INCORRECT PROGRAMS

One of the things which, in my opinion, makes contracts so difficult to understand deeply is that we are forced to reason about incorrect programs, which can be very counterintuitive. In this section, we describe terminology to aid us.

First, we need to define delineate different meanings of undefined behaviour. To crystalize the issue, suppose that someone remarks "This function has undefined behaviour". What might they mean? There are three options:

1. The function's *specification* states that, for certain inputs, the behaviour is not defined. Therefore, no constraints are placed upon the implementer for handling such inputs. Colloquially: "This function has undefined behaviour because its specification says it does".

2. The function's specification is insufficiently complete for clients to be confident that the behaviour for particular inputs in well-defined. Colloquially: "I am not confident that this function is specified well enough to be sure that all of its behaviour is well-defined".

3. The implementer of the function made a mistake and so the body of the function *accidentally* exhibits undefined behaviour, contrary to the specification. Colloquially, "This function exhibits undefined behaviour because its implementer made a mistake".

The classification of contracts is purely a matter of specification, and so it is only the first two cases that are of relevance to this paper. Importantly, invoking a function in a way which elicits behaviour specified—explicitly or implicitly—as undefined is a programming error. Since C++26, the standard [N5014] has another way to categorize programming errors: erroneous behaviour [P2795R5]. Unlike, say, dereferencing a null pointer erroneous behaviour is well-defined. However, it is "always a consequence of incorrect program code". This leads us to define:

Definition 2.1. Contractually Incorrect Behaviour *is any behaviour which, according to the plain-language contract, can only be elicited as the result of incorrect program code.*

Example 2.1.a. *A version of* `float sqrt(float)` *with a plain-language contract stating that the behaviour is undefined for negative arguments specifies contractually incorrect behaviour.*

Example 2.1.b. *A version of* `float sqrt(float)` *with a plain-language contract stating that it returns an erroneous value for negative arguments specifies contractually incorrect behaviour.*

Example 2.1.c. *A version of* `float sqrt(float)` *with a plain-language contract stating that it is a bug to call it with negative values and that the program will abort specifies contractually incorrect behaviour.*

The last two examples are subtle, since the contractually incorrect behaviour is not associated with undefined behaviour. Note further that, in the last example, the mention of buggy behaviour in the plain-language contract is crucial. If this were to be removed leaving just "the function will attempt to abort the program if is called with a negative argument", then there ceases to be contractually incorrect behaviour: this behaviour is elevated to essential behaviour of the function and the

wide						
faithful						
free	disambiguating	**lossy**	disconnecting	**Unconditionally hardened**	**narrow**	pathological

TABLE I. The various categories of contract (new and old) and their relationships. The terms in bold are the ones expected to cover the majority of practical cases. Wide contracts are subdivided into four categories, the union of two of which are called faithful. The non-wide categories include the remaining three cases, two of which—unconditionally hardened and pathological—have no analogue in the wide/narrow scheme of [N3248].

contract is wide. As mentioned above, this may be counterintuitive which is part of the motivation for the taxonomy proposed by this paper: we call a wide contract of this type disconnecting.

In the real world, we may not have the luxury of a plain-language contract that is written down with sufficient precision (or even at all). Therefore, pragmatically, it may be necessary to apply some judgement as to what constitutes contractually incorrect behaviour.

Example 2.1.d. *A version of* `float sqrt(float)` *with no plain-language contract beyond what can reasonably be inferred from its name implicitly specifies contractually incorrect behaviour: while it is reasonable to surmise its behaviour for finite, non-negative values, beyond this there is not enough information to be able to rely on what it does.*

The degree to which plain-language contracts are incomplete will hopefully reduce as people adopt C++26, as some aspects of the plain-language contract may now be expressed in code, as part of a function's signature [P2900R14]. With this in mind:

Definition 2.2. *A* **precondition** *of a function is a predicate which, if violated, results in contractually incorrect behaviour. A necessary and sufficient condition for a correctly implemented function to exhibit contractually incorrect behaviour is for at least one precondition to be violated.*

This definition is slightly different to the definition in the C++ Standard [N5014]:

> *Preconditions*: conditions that the function assumes to hold whenever it is called; violation of any preconditions results in undefined behavior.

Part of the reason for the difference is that our definition of contractually incorrect behaviour includes both undefined and erroneous behaviour. Furthermore, just a erroneous behaviour is well-defined, we also want to capture the case of preconditions which, if violated, lead to a specific form of well defined behaviour:

Definition 2.3. *An unconditionally hardened (a.k.a. non-ignorable) precondition is a precondition for which violation is guaranteed to be handled in a well-defined way that attempts to end program execution.*

The C++ standard (taken as [N5014], with the delta in [P3878R1] applied) does not define an *unconditionally* hardened precondition. Rather, a hardened precondition is defined similarly to a precondition, but with the following removed: 'violation of any preconditions results in undefined behavior'. Thus, according to the standard, while violating a precondition definitely results in undefined behaviour, violating a hardened precondition may or may not. How can that be? That point is that a hardened precondition has guaranteed behaviour only in a *hardened implementation*. If the implementation is not hardened—as may be desirable for clients in certain domains—violating a hardened precondition is no different from violating a non-hardened precondition.

For the purposes of this paper, the strictness, or otherwise, of guarantees that are provided *in code* are of particular importance. Thus, if we imagine something like

```
float sqrt(float x)
  pre <hardened> (x >= 0);
```

this expresses that, in *some* situations, precondition violation will unequivocally lead to well-defined behaviour but this is not guaranteed in all situations. The contract for this function is therefore narrow: the existing classification works just fine. The situation which is problematic for the existing classification is where the precondition is guaranteed to lead to well-defined behaviour in all situations: put a different way, the precondition is unconditionally hardened.

This section started with the definition of contractually incorrect behaviour and ends with that of contractually correct behaviour:

Definition 2.4. Contractually correct *behaviour is any behaviour which is not contractually incorrect.*

While the definition is straightforward, there is an interesting nuance, since the definition permits behaviours about which the plain-language contract is ambivalent. As eloquently pointed out in [P2861R0], while a contract defines the behaviours that a function must model, it may be unopinionated about additional, implementation-defined behaviours. To paraphrase an example, none of the specifications given in this paper for a hypothetical `sqrt` function rule out an implementation from *additionally* dumping the answer to `std::cout`. The quality of such an implementation may leave rather a lot to be desired, but it is conforming.

III. IMPLEMENTATION FREEDOM & THE ABI

We now return to the issue glossed over when first discussing hardened preconditions. Consider a function with a narrow contract i.e. with behaviour that the plain-language contract specifies as undefined. An implementer of such a function is free to do (almost) whatever they want in the situation that leads to the contractually incorrect behaviour. As mentioned in the introduction, it is entirely conforming for `std::vector::operator[]` and `std::vector::at` to have exactly the same implementation: the implementer of `std::vector` is at liberty to throw an exception if `operator[]` is called out of contract (their clients may not like this, but that's a different matter).

With this in mind, consider some function with a narrow contract that is declared in a header but defined out of line. In its first incarnation, let us suppose the header contains this:

```
// Precondition: x >= 0
void foo(int x);
```

If, for narrow contracts, an implementer is free to do whatever they want when it comes to the contractually incorrect behaviour, can they therefore not do this?

```
void foo(int x)
    pre <quick_enforce> (x >= 0);
```

The problem here is that, by design, `quick_enforce` cannot be ignored. Therefore, as pointed out by Lisa Lippincott, it becomes part of the ABI and removing it constitutes an ABI break. One way to see this is to consider optimization. From the signature alone—i.e. without any knowledge of the body of the function (which we take to be in the cpp)—the caller's compiler has enough information to do things such as eliding checks:

```
foo(y);
if(y >= 0) {
  ...
}
```

The condition is redundant due to guaranteed enforcement of `foo`'s precondition and so may be optimized away. If the precondition on `foo` is now removed, but the calling code is not recompiled, then the absence of the check could prove to be damaging.[7]

This is fundamentally different from any other implementation decision for the contractually incorrect behaviour of `foo`. For example, if the implementer had decided to throw an exception in the body of the function and then reverts this decision, clients do not experience an ABI break since, in this example, the change is 'hidden' in the cpp. Similarly, P2900 Contracts [P2900R14] are designed so that an implementer can introduce

```
void foo(int x)
    pre (x >= 0);
```

without fear of ABI breakage. This is down to the fact that P2900 contracts may be ignored: clients of `foo` cannot assume that the precondition is enforced by the compiler which consumes the definition of `foo` and so cannot optimize on that premise. The ignore semantic is very much a feature, and not a bug, of P2900.

Therefore, we conclude that there is an important restriction on the implementation freedom for contractually incorrect behaviour: for functions with a narrow contract, implementers may not elevate the treatment of contractually incorrect behaviour to an unconditionally hardened precondition. Put another way, a function with nothing but unconditionally hardened contracts does not have a narrow contract. Of course, since it has preconditions is does not have a wide contract either which speaks to the premise of this paper and the need for a richer taxonomy.

To conclude this section, we note that optimization is not the only source of ABI issues when it comes to unconditional hardening. Consider the act of calling a function. When a precondition is present, there is the question of where the corresponding code is actually emitted during compilation: caller side, callee side or both. P2900 is ambivalent; combined with ignorability, this gives implementers considerable freedom within the space of schemes that yield a stable ABI.

To illustrate this, imagine an application built in C++23 which utilizes a shared library. Now suppose that the library has C++26 contracts added and the library alone is rebuilt. Can the application—which has not been rebuilt—continue to correctly consume the updated library binary? The answer is yes. Even if the library has been built using a terminating semantic, the application—which here plays the role of the caller—is under no obligation to insert caller-side checks. Which is just as well since in this example the application is not rebuilt!

The situation with unconditional hardening is more difficult. If checking is done caller-side then inserting unconditionally hardened preconditions is an ABI break, since the application in our example cannot possibly fulfil its responsibilities without being rebuilt. This implies that unconditionally hardened preconditions should be emitted callee side (this conclusion can also be reached by considering indirect calls [P3912R0]). However, this creates an interesting tension. Suppose that a function, `bar`, in the shared library is updated to use P2900 contracts:

```
void bar(int x, int y)
     pre (y/x > 0);
```

[7] If the precondition is never actually violated, then we may get away with this. However, preconditions in code can be thought of as a mechanism to build up statistical confidence of correctness [P3578R1]. Therefore, even if something appears to be working perfectly and has never previously had its preconditions violated, it may just be a matter of waiting long enough.

And let us be optimistic: the entire system, library, application and all is rebuilt with a terminating semantic. It is entirely reasonable if code for this precondition is emitted *caller*-side. But now suppose that an unconditionally hardened precondition is added to the library code

```
void bar(int x, int y)
    pre <enforce> (x > 0)
    pre (y/x > 0);
```

This is an ABI break! When the library is rebuilt, the first precondition must—according to the above logic—be emitted callee-side. To preserve the requirement that predicates are evaluated in lexical order, the second precondition must also be emitted callee-side. However, if the application is not rebuilt this time, then it is stuck with the caller-side check `y/x > 0` which will incorrectly be evaluated before the nascent, unconditionally hardened precondition.

To emphasise: this is not a problem for P2900 contracts. If checks are emitted caller-side, `pre (x > 0)` is added, and only the library is rebuilt, this check will simply not be emitted. But that is conforming with the P2900 model. This analysis reinforces the conclusion above that unconditionally hardened contracts are fundamentally different from narrow contracts.

IV. PRIMARY AND SECONDARY BEHAVIOUR

The goal of this section is to formalize the notions of primary and secondary behaviours sketched out in the introduction. To develop a rigorous framework for describing the behaviour of functions, we start with the standard definitions of a mathematical function:

Definition 4.1. *A mathematical function from a set X to a set Y assigns to each element of X exactly one element of Y. The set X is called the domain of the function and the set Y is called the codomain of the function.*

It will also be useful to consider the more general case of a partial function, which acts on a subset of X:

Definition 4.2. *A mathematical partial function f from a set X to a set Y is a function from a subset S of X (possibly the whole X itself) to Y.*

Every function is a map from a subset of its inputs to its effects. This subset of inputs may not comprise all of a function's arguments. For example, an 'out' parameter is identified with the codomain of a function, not the domain. Some arguments may modulate the effect of the function, rather than forming part of the domain. An example is the comparison object used for `sort` and related algorithms. Furthermore, the domain of a function may be carried by things other than the arguments, for example `class` state (both static and non-static) and global or thread-local variables.

Similarly, a function's behaviour may be modulated by things other than its parameters. For example, although `std::abort` does not take any arguments the documentation says that it:

> Causes abnormal program termination unless `SIGABRT` is being caught by a signal handler passed to `std::signal` and the handler does not return.

In other words, the behaviour of `std::abort` may be modulated by state which exists outside of the function itself. Similarly, the behaviour of `std::sqrt` may implicitly depend on the floating-point environment. Finally, the behaviour of a true random number generator may depend on some state determined by a physical device.

To make progress, recall the key observation was made in the introduction: a function's domain in the strict mathematical sense may differ from the practical domain in code and, therefore, it makes sense to talk of the primary behaviour of a function. Intuitively, the primary behaviour captures the primary intent of the function.

Definition 4.3. *The* **primary behaviour** *of a function, f, from a set X to a set Y corresponds to the partial function from a subset of X to Y that represents the primary purpose of f. A necessary condition but not sufficient condition for executing primary behaviour is that all preconditions are satisfied.*

Example 4.3.a. *The primary behaviour of* `void std::abort()` *is to abort the program unless* `SIGABRT` *is being caught by a signal handler passed to* `std::signal` *and the handler does not return.*

Example 4.3.b. *The primary behaviour of*

```
void abort()
  pre <quick_enforce> (false);
```

is the empty set, since a necessary condition for behaviour to be primary is that all preconditions are satisfied.

Example 4.3.c. *The primary behaviour of* `float std::sqrt(float)` *is to compute the square-root of* ± 0 *and the positive floats, excluding NaN. If adhering to IEEE 754, the result must be correctly rounded.*

It is well worth pointing out that there is a degree of choice here, particularly regarding NaN. For functions like `sqrt`, which approximate a mathematical ideal, the latter provides a strong guide to what the primary behaviour should comprise. Ultimately, however, we are interested in programming and not mathematics, per se, and so the final say resides with the plain-language contract. It is generally a good idea for the latter to be informed by mathematics, but it should not mindlessly defer. Thus it would be legitimate to consider the computation of the square root of NaN to be part of the primary behaviour; however for this paper we will not do so. This

choice does not affect any of the paper's essential logic, but rather just the examples, for which it is important to establish a convention and apply it consistently.

Interestingly, in some situations it is necessary for aspects of the primary behaviour to be unspecified.

Example 4.3.d. *The primary behaviour of* `std::sort` *is to sort the elements according to the predicate. However the ordering of elements which compare equal according to the predicate is not specified.*

Indeed, the typical methodologies for implementing `std::sort`—quick sort and heap sort—are not stable (though insertion sort, which may be used for very small ranges, is). On the other hand, merge sort is an efficient sorting algorithm that is stable[8] and is the workhorse for `std::stable_sort`, the primary behaviour of which is fully specified.

From the definition of primary behaviour, it is tempting to immediately define secondary behaviour as its complement. However, this would be a mistake.

The problem with defining secondary behaviour as the complement of primary behaviour is that it may include behaviours about which the plain-language contract is ambivalent, as discussed at the end of section II. Quite simply, that is not what we want from this particular definition.

To arrive at the desired definition simply involves introducing some relevant terminology first. Just as for mathematical functions, we talk of the domain and the codomain, we may generalize these to the current context:

Definition 4.4. *The* **primary domain** *of a function is the subset of the domain that elicits the primary behaviour.*

Example 4.4.a. *The primary domain of* `void std::abort()` *is a* `void` *argument.*

Example 4.4.b. *The primary domain of*

```
void abort()
  pre <quick_enforce> (false);
```

is the empty set.

Example 4.4.c. *The primary domain of* `float std::sqrt(float)` *is* ± 0 *and the positive floats, excluding NaN.*

From this, the definition of the secondary domain naturally follows and provides us with the tool we need to define secondary behaviour:

Definition 4.5. *The* **secondary domain** *of a function is the complement of the primary domain: i.e. the subset of the domain not within the primary domain.*

Example 4.5.a. *The secondary domain of* `void std::abort()` *is the empty set.*

Example 4.5.b. *The secondary domain of*

```
void abort()
  pre <quick_enforce> (false);
```

is a `void` *argument.*

Example 4.5.c. *The secondary domain of* `float std::sqrt(float)` *is the negative floats, (taken to exclude* -0.0*), and NaN. Note: a discussed above, it is debatable whether NaN is in the primary or secondary domain. However, from the perspective of the underlying mathematics, NaN is merely an artefact of how we approximate the real numbers on a computer and so belongs in the secondary domain.*

Now we are in a position to define secondary behaviour:

Definition 4.6. *The* **secondary behaviour** *of a function is the behaviour elicited by elements of the secondary domain.*

To emphasise again what was stated in the introduction: invocation of secondary behaviour is necessary but in general not sufficient to result in the execution of contractually incorrect behaviour.

Example 4.6.a. *The secondary behaviour of* `void std::abort()` *is the empty set.*

Example 4.6.b. *The secondary behaviour of*

```
void abort()
  pre <quick_enforce> (false);
```

is to abort the program.

Example 4.6.c. *The secondary behaviour of* `float std::sqrt(float)` *is the behaviour for negative inputs (taken to exclude* -0.0*), and NaN.*

According to our definitions, it is set in stone that all preconditions must be satisfied to elicit primary behaviour. But within this, as emphasised above, there is some flexibility over what is considered primary versus secondary behaviour, with the plain-language contract having the final say. We have already discussed this for the case of NaN in the context of `float std::sqrt(float)`. Now we consider a further example.

Example 4.6.d. *Consider a version of* `float sqrt(float)` *which, for negative values, computes the magnitude of the actual imaginary result. This could be thought of as a 'best effort' for the secondary behaviour. Alternatively, one could recognize this function as, say,* `float sqrt_of_abs(float)`*, for which the primary behaviour is elicited by both positive and*

[8] Albeit with other drawbacks—there are no magic unicorns.

negative numbers. Choosing between these characterizations is ultimately a matter of taste, decided by the plain-language contract. However, in this case, it seems most reasonable to recognize that the primary behaviour has been extended.

Finally, we may talk of the effects of a function. Abstractly, these behaviours belong to a set known as the codomain: a function maps elements of its domain into its codomain. An important and subtle point is that the *image* of a function may be distinct from the codomain. In particular, the image is a subset of the codomain, which may correspond to the entire codomain but, equally, may not. Not only is this nuance highly relevant to our considerations, but it has a beautiful interpretation in terms of postconditions.

Consider the following:

```
float sqrt(float x)
  pre (x >= 0)
  post(r : r >= 0);
```

This declaration specifies the codomain as the non-negative `float`s (including -0.0). However not all of these floats can actually be reached by the function! Most obviously, any finite float bigger than the square root of

```
constexpr auto flt_max
    = numeric_limits<float>::max();
```

is unreachable. But even for numbers less than this, due to finite precision effects, an implementation may not generate every float in the reduced range. The point is that the image of the function—i.e. the set of numbers actually generated—is, in this case, not the same as the codomain. Indeed, different implementations of this function may even have slightly different images.

We can go further. Suppose we introduce two mathematical square root functions. For one the codomain is specified to be the real numbers and, for the other, it is specified to be just the non-negative reals. Regardless, both functions have the same domain and the same image. So are they the same function or not? It turns out that, strictly, they are not due to their differing specifications. Interestingly, the C++ contracts model guides us to the right answer. Imagine that we have a program containing the following two declarations:

```
float sqrt(float x)
  pre (x >= 0)
  post(r : r >= 0);

float sqrt(float x)
  pre (x >= 0)
  post(r : (r >= -flt_max) && (r <= flt_max));
```

Such a program is ill-formed[9]: these functions are not the same, by dint of their differing codomains, even if their implementations—and hence images—are identical.[10]

One additional point worth making is that considering functions with different codomains as different, even if the images are the same, is very natural when it comes to composition. Take the following declarations of a function which takes the square:

```
float square(float x)
  post(r : r >= 0);
```

versus

```
float square(float x)
  post(r : (r >= -flt_max) && (r <= flt_max));
```

Now consider the composition `sqrt(square(x))`. If we choose the first of the above declarations of `square` then this has the nice property that its postcondition matches the precondition of `sqrt` into which its output is fed. These functions cleanly compose, which is useful information for a human reader, a compiler, a static analyser, an optimizer and, one must tacitly presume, a sufficiently enlightened LLM. On the other hand, if we choose the second of the above declarations of `square` then we must accept that our code is not specified in a way which allows us, for example, to elide the precondition check on `sqrt`.

One of the lessons of this is that there may be benefits to constraining (but not over-constraining) the codomain. There is nothing incorrect, from the perspective of either programming or mathematics, in specifying the codomain of `square` to be the real numbers. There are, however, benefits from specifying it to be the non-negative numbers. On the other hand, trying to constrain it to the actual image of a particular implementation is likely unhelpful.

To conclude the discussion on the importance of distinguishing the codomain and image we give a somewhat different example by considering `std::sort`. To make the discussion simpler, let us restrict our attention to sorting a range holding `pair{2, 1}, pair{1, 1}, pair{1, 7}`, where the predicate is sensitive only to the first element of the pair. Since `std::sort` is not a stable sort, the codomain is the set of all sets of elements which are ordered with respect to the first element:

```
{ { pair{1, 1}, pair{1, 7}, pair{2, 1} },
  { pair{1, 7}, pair{1, 1}, pair{2, 1} } }
```

However, a specific *implementation* will map the input into one of these elements of the codomain: the image and the codomain are not the same.

With all of this in mind, we now tie together our notion of primary behaviour with the codomain of a function:

[9] Although no diagnostic required in the sad situation that these declarations are in different translation units.

[10] From this mathematically oriented discussion, it seems very natural to make pre/post conditions part of the type. However, there are very good reasons not to do this, from a programming perspective [P2899R1, P2900R14]

Definition 4.7. *The* **primary codomain** *of a function,* `f`*, is the set of effects into which the contractually specified observable effects of* `f` *operating on the elements of its primary domain are constrained to fall.*

Example 4.7.a. *The primary codomain of* `void std::abort()` *is the program aborting unless* `SIGABRT` *is being caught by a signal handler passed to* `std::signal` *and the handler does not return.*

Example 4.7.b. *The primary codomain of*

```
void abort()
  pre <quick_enforce> (false);
```

is the empty set.

Example 4.7.c. *The primary codomain of* `float std::sqrt(float)` *is the positive floats and* ± 0.0.

In direct correspondence with the primary behaviour being induced by the elements of the primary domain we have:

Definition 4.8. *The* **secondary codomain** *of a function,* `f`*, is the set of effects into which the contractually specified observable effects of* `f` *operating on the elements of its secondary domain are constrained to fall.*

Example 4.8.a. *The secondary codomain of* `void std::abort()` *is the empty set.*

Example 4.8.b. *The secondary codomain of*

```
void abort()
  pre <quick_enforce> (false);
```

is the the program aborting.

Example 4.8.c. *Consider a version of* `float sqrt(float)` *guaranteed to produce an unspecified negative number for numbers less than zero. In this case, the secondary codomain is the negative floats.*

Example 4.8.d. *The secondary codomain of* `float std::sqrt(float)` *is implementation-defined though, if IEEE 754 is supported, it is the union of*

1. *NaN (returned if the argument is NaN);*
2. *The pair* $\{NaN,$ `FE_INVALID`$\}$ *which is produced by a negative value (with* -0.0 *excluded).*

We conclude this section with an example where programming considerations override the underlying mathematical ideal. Consider $\sin 10^{100}$. Given that the period of sin is 2π, any computed value for such extreme numbers is devoid of physical meaning: the gap between adjacent `double`s at this scale is about 20 orders of magnitude bigger than the ratio of the size of the observable universe to the Planck scale! What, then, is a reasonable plain-language contract for `float sin(float)`? It depends on what one is hoping to achieve, as illustrated by the following (non-exhaustive) pair of examples.

Example 4.8.e. *For a version of* `float sin(float)` *for which cross-platform reproducibility is the goal, then*

1. *The answer is prescribed down to the last bit, regardless of how much physical sense it makes;*
2. *The primary domain is* `[-flt_max, flt_max]`*;*
3. *The secondary domain comprises* $\pm\infty$ *and NaN;*
4. *The primary codomain is the range* $[-1, 1]$*;*
5. *The secondary codomain is NaN.*

Example 4.8.f. *For a version of* `float sin(float)` *for which the raw speed is the goal and the philosophy is 'garbage in garbage out', it may be acceptable to define some large number,* L*, such that*

1. *The primary domain is* $[-L, L]$*,*
2. *The secondary domain is NaN and all values, including* $\pm\infty$*, outside the primary domain;*
3. *The primary codomain is the range* $[-1, 1]$*;*
4. *The secondary codomain is the same as the secondary domain.*

Thus, in this realization of `float sin(float)`*, only arguments within a given range are considered sensible, and comprise the primary domain; anything outside of this is simply returned directly. Mathematical purity has deferred to practical coding realities.*

V. A TAXONOMY FOR CONTRACTS

The key to categorizing contracts resides with the *secondary* behaviour. The first category is where there is no secondary behaviour, which is a special case of a wide contract.

Definition 5.1. *A contract is* **free** *if the secondary domain is the empty set.*

Example 5.1.a. `std::abort` *has a free contract.*

Example 5.1.b. *Unsigned integer addition has a free contract.*

Example 5.1.c. *A function which does nothing and returns nothing, viz.* `void foo() {}` *has a free contract. The primary domain is a* `void` *argument and the secondary domain is the empty set.*

The remaining cases all deal with functions that have some secondary behaviour. The next is the natural generalization of a free contract: the secondary behaviour always return control to the caller in a way that primary and secondary behaviour are distinguishable purely from the effects of the function. By control returning to the caller, we mean in the most general sense. Thus, while it may be via an actual `return` it could also be

via an exception or, conceivably, a `longjmp`. And again, to emphasise: control returning to the caller is only being considered in the context of the secondary behaviour. Whether or not the primary behaviour returns control to the caller is not relevant to our considerations.

One way to achieve distinguishability is to throw an exception (of a particular type) only for secondary behaviour. Alternatively, in some situations a `std::expected`, or equivalent, could be used. Another possibility would be a that a special return value, such as NaN, could uniquely identify secondary behaviour. Or it could be that secondary behaviour is disambiguated by a flag being set in some global/thread-local state. However it is done, the point is that the caller can, in principle, distinguish primary from secondary behaviour purely by looking at the effects of the function, without having to reference the state with which it was called.

Definition 5.2. *A contract is* **disambiguating** *if and only if neither the primary nor secondary domains are empty and, for each element of the secondary domain, invocation:*

1. *Returns control to the caller;*
2. *Produces a behaviour not within the primary codomain.*

Example 5.2.a. *If IEEE 754 is supported then* `float std::sqrt(float)` *has a disambiguating contract since the secondary domain maps to NaN, which is not part of the primary codomain. (Given the establishment of our earlier convention that NaN is not part of the primary domain.)*

As mentioned in the introduction, it is helpful to combine the previous two definitions:

Definition 5.3. *A contract is* **faithful** *if and only if it is either* **free** *or* **disambiguating**.

The next contract definition relaxes the requirement that no elements of the secondary domain map to the primary domain. Amongst other things, this case captures secondary behaviour which is constrained but not fully specified.

Definition 5.4. *A contract is* **lossy** *if and only if neither the primary nor secondary domains are empty and*

1. *For each element of the secondary domain, invocation returns control to the caller;*
2. *It is not guaranteed that the primary and secondary codomains are disjoint.*

Example 5.4.a. *A version of* `float sqrt(float)` *which, for negative numbers, is specified to return zero is lossy, since zero is within the primary codomain.*

Example 5.4.b. *A version of* `float sqrt(float)` *which, for negative numbers, is specified to return an unspecified number is lossy, since it is not guaranteed that this number is not within the primary codomain.*

Suppose, in the previous example, that a particular implementation happens to return -1 for the square root of a negative number. Since -1 is not in the primary codomain, does this mean that the function has a disambiguating contract? The answer is a resounding no! As discussed in the introduction, it all comes down to contractual guarantees. There is no guarantee that the function returns -1 for the square root of a negative number and, according to the contract, the implementer is perfectly within their rights to change this behaviour without due notice. When classifying, one must always consider the 'worst case'.

The next case covers wide contracts which may have the counterintuitive effect of unexpectedly ending the program. However, as noted already, `std::abort` could be prevented from actually aborting the program if we were to install an appropriate signal handler. To avoid aborting, the signal handler must not return and so could, for example, loop forever printing out something unhelpful. Therefore, this category of contract will cover cases where the function attempts to end the program, with the tacit acknowledgement that this may not succeed. We could make contracts like this their own category, but it seems reasonable to bundle in the edge case where no attempt is made to end the program but something like an infinite loop is encountered nevertheless.

Definition 5.5. *A contract is* **disconnecting** *if and only if neither the primary nor secondary domains are empty, and for at least one element of the secondary domain*

1. *A call is made that attempts to end the program; or*
2. *Program execution continues indefinitely without return control to the caller*

Example 5.5.a. *A version of* `float sqrt(float)` *which, for negative numbers, is specified to call* `std::abort` *has a disconnecting contract.*

Example 5.5.b. *A version of* `float sqrt(float)` *which, for negative numbers, is specified to invoke an infinite loop that repeatedly prints out* `Bad Human` *has a disconnecting contract.*

Example 5.5.c. *A version of* `float sqrt(float)` *which, for negative numbers, is specified to call* `std::breakpoint_if_debugging` *has a disconnecting contract.*

This classification of the last example is a consequence of the point made at the end of the introduction. When classifying contracts, one must reason in terms of the behaviour which is strictly guaranteed. While it may be the case that there is no debugger present, and `std::breakpoint_if_debugging` immediately returns, it is possible that a debugger is both present and never returns control. Since, in this example, `std::breakpoint_if_debugging` is part of the secondary behaviour, there are no strict guarantees that the

secondary behaviour always returns control to the caller. Therefore, this contract is disconnecting.

The remaining three categories all have preconditions. In terms of arriving at definitions, it is actually helpful to start with the most obscure.

Definition 5.6. *A contract is* **pathological** *if and only if a non-empty subset of the contractually correct behaviour is elicited by the violation of an unconditionally hardened precondition.*

Example 5.6.a. *If, as the name suggests, the contractually correct behaviour of* `abort` *is to attempt to abort the program then the highly dubious declaration mentioned in the introduction has a pathological contract:*

```
void abort()
      pre <quick_enforce> (false);
```

Note that since, according to our definitions, a condition for behaviour to be primary is that all preconditions are satisfied, a pathological contract effectively causes the demotion of primary behaviour to secondary behaviour.

The next category similarly cannot be classified using wide/narrow since it occupies the interesting niche of both having preconditions but no specified undefined (or erroneous) behaviour.

Definition 5.7. *A contract is* **unconditionally hardened** *if and only if it is not pathological and has at least one precondition, such that all preconditions are unconditionally hardened.*

Example 5.7.a. *A version of* `float sqrt(float x)` *which has a precondition that* `x >= 0` *which is guaranteed to be enforced, in code, has an unconditionally hardened precondition. Note that this cannot be done with [P2900R14] since contract semantics are not specified in code. However, along the lines of [P3400R2] one could envisage:*

```
float sqrt(float x)
    pre <enforce> (x >= 0);
```

Next, we move to narrow contracts which are, in spirit, the same as in [N3248] .

Definition 5.8. *A contract is* **narrow** *if and only if it is not pathological and has at least one precondition, at least one of which is not unconditionally hardened.*

Example 5.8.a. *Precisely as in [N3248], the following function has a narrow contract:*

```
float sqrt(float x)
    pre  (x >= 0);
```

Note that while the semantic may be chosen to be `enforce` *or* `quick_enforce`*, this is not specified in code. Insofar as the declaration of the function goes, there is no guarantee that violating the precondition does not ultimately lead to the execution of specified undefined behaviour.*

Example 5.8.b. *Consider everybody's favourite mathematical function*

```
float std::frexp(float num, int* exp);
```

This has a plain-language precondition that the pointer, `exp`*, must be derefenceable. Therefore, the function has a narrow contract. Note that the precondition is not fully expressible in C++ since, while a null-check is trivial to do, there is no mechanism to check that the pointer is not dangling.*

VI. CONCLUSION

By adopting [P2900R14], C++26 has a contracts facility that provides both immediate utility in a range of scenarios, and a sound foundation for future evolution. It is hard not to be excited by the tantalizing prospect of providing a mechanism to mitigate swathes of undefined behaviour from C++, as described in the remarkable [P3100R4]. Indeed, this would bootstrap the safety of contract assertions themselves: the protector would protect itself, providing a plausible realization of some of the ideas expressed in [P3285R0]. As C++ contracts evolve, and as they gain adoption within the wider ecosystem, it is essential that we have the right vocabulary with which to discuss them.

This is well-illustrated by the current debates around non-ignorable contracts [P3911R1]. It is my contention that the existing terminology is insufficient to describe what non-ignorable contracts express since they have aspects that are simultaneously indicative of a wide and narrow contract. The resolution proposed by this paper is that they are in fact neither: with a richer classification they can be appropriately named and, most importantly, not confused with existing concepts.

Personally, my biggest concern about the adoption of contracts within the broader C++ community is their misapplication due to misunderstandings of their purpose. To go back to one of the examples at the start of this paper: a version of `sqrt` that aborts for negative input. Should this behaviour be lifted to a precondition, or maybe even an unconditionally hardened precondition? Answering this question requires an analysis of the plain-language contract—should it even exist—and nuanced understanding of what the various different option actually express.

It is hoped that this paper plugs a gap in the literature, providing terminology based on a rigorous, elementary approach that will help people to reason about questions such as this.

ACKNOWLEDGMENTS

I would like to thank Mark Hoemmen, Alisdair Meredith, Timur Doumler, Josh Berne, Matthew Taylor, An-

thony Williams and Philip Craig for helpful discussions/comments on the draft. Particular thanks to Andrzej Krzemieński for encouraging me to write this paper.

REFERENCES

[P2861R0] John Lakos, The Lakos Rule

[P2899R1] Joshua Berne, Timur Doumler, Rostislav Khlebnikov and Andrzej Krzemieński Contracts for C++ — Rationale

[N3248] Alisdair Meredith and John Lakos, `noexcept` Prevents Library Validation

[P3471R2] Konstantin Varlamov and Louis Dionne Standard Library Hardening

[P3911R1] Darius Neațu, Andrei Alexandrescu, Lucian Radu Teodorescu, Radu Nichita and Herb Sutter RO 2-056 6.11.2 [basic.contract.eval] Make Contracts Reliably Non-Ignorable

[P3400R2] Joshua Berne, Controlling Contract-Assertion Properties

[P2795R5] Thomas Köppe, Erroneous behaviour for uninitialized reads

[P2900R14] Joshua Berne, Timur Doumler and Andrzej Krzemieński Contracts for C++

[P3578R1] Ryan McDougall, What is "Safety"?

[P3912R0] Timur Doumler, Joshua Berne, Gašper Ažman, Oliver Rosten, Lisa Lippincott and Peter Bindels Design considerations for always-enforced contract assertion

[N5014] Ed. Thomas Köppe Working Draft Programming Languages — C++

[P3878R1] Ville Voutilainen, Johnathan Wakely, John Spicer and Stephan T. Lavavej Standard library hardening should not use the 'observe' semantic

[P3100R4] Timur Doumler and Joshua Berne A framework for systematically addressing undefined behaviour in the C++ Standard

[P3285R0] Gabriel dos Reis Contracts: Protecting The Protector