Joe Duffy's Blog

Founder/CEO Pulumi • Cloud, languages, and developer tools guy • Eat, sleep, code, repeat

Popular:
Pulumi
Midori
Software Leadership

February 7, 2016

The Error Model

Midori was written in an ahead-of-time compiled, type-safe language based on C#. Aside from our microkernel, the whole system was written in it, including drivers, the domain kernel, and all user code. I've hinted at a few things along the way and now it's time to address them head-on. The entire language is a huge space to cover and will take a series of posts. First up? The Error Model. The way errors are communicated and dealt with is fundamental to any language, especially one used to write a reliable operating system. Like many other things we did in Midori, a "whole system" approach was necessary to getting it right, taking several iterations over several years. I regularly hear from old teammates, however, that this is the thing they miss most about programming in Midori. It's right up there for me too. So, without further ado, let's start.

Introduction

The basic question an Error Model seeks to answer is: how do "errors" get communicated to programmers and users of the system? Pretty simple, no? So it seems.

One of the biggest challenges in answering this question turns out to be defining what an error actually *is*. Most languages lump bugs and recoverable errors into the same category, and use the same facilities to deal with them. A `null` dereference or out-of-bounds array access is treated the same way as a network connectivity problem or parsing error. This consistency may seem nice at first glance, but it has deep-rooted issues. In particular, it is misleading and frequently leads to unreliable code.

Our overall solution was to offer a two-pronged error model. On one hand, you had fail-fast – we called it *abandonment* – for programming bugs. And on the other hand, you had statically checked exceptions for recoverable errors. The two were very different, both in programming model and the mechanics behind them. Abandonment unapologetically tore down the entire process in an instant, refusing to run any user code while doing so. (Remember, a typical Midori program had many small, lightweight processes.) Exceptions, of course, facilitated recovery, but had deep type system support to aid checking and verification.

This journey was long and winding. To tell the tale, I've broken this post into six major areas:

In hindsight, certain outcomes seem obvious. Especially given modern systems languages like Go and Rust. But some outcomes surprised us. I'll cut to the chase wherever I can but I'll give ample back-story along the way. We tried out plenty of things that didn't work, and I suspect that's even more interesting than where we ended up when the dust settled.

Ambitions and Learnings

Let's start by examining our architectural principles, requirements, and learnings from existing systems.

Principles

As we set out on this journey, we called out several requirements of a good Error Model:

- **Usable.** It must be easy for developers to do the "right" thing in the face of error, almost as if by accident. A friend and colleague famously called this falling into the The Pit of Success.

The model should not impose excessive ceremony in order to write idiomatic code. Ideally it is cognitively familiar to our target audience.

- **Reliable.** The Error Model is the foundation of the entire system's reliability. We were building an operating system, after all, so reliability was paramount. You might even have accused us as obsessively pursuing extreme levels of it. Our mantra guiding much of the programming model development was "*correct by construction*."
- **Performant.** The common case needs to be extremely fast. That means as close to zero overhead as possible for success paths. Any added costs for failure paths must be entirely "pay-for-play." And unlike many modern systems that are willing to overly penalize error paths, we had several performance-critical components for which this wasn't acceptable, so errors had to be reasonably fast too.
- **Concurrent.** Our entire system was distributed and highly concurrent. This raises concerns that are usually afterthoughts in other Error Models. They needed to be front-and-center in ours.
- **Diagnosable.** Debugging failures, either interactively or after-the-fact, needs to be productive and easy.
- **Composable.** At the core, the Error Model is a programming language feature, sitting at the center of a developer's expression of code. As such, it had to provide familiar orthogonality and composability with other features of the system. Integrating separately authored components had to be natural, reliable, and predictable.

It's a bold claim, however I do think what we ended up with succeeded across all dimensions.

Learnings

Existing Error Models didn't meet the above requirements for us. At least not fully. If one did well on a dimension, it'd do poorly at another. For instance, error codes can have good reliability, but many programmers find them error prone to use; further, it's easy to do the wrong thing – like forget to check one – which clearly violates the "pit of success" requirement.

Given the extreme level of reliability we sought, it's little surprise we were dissatisfied with most models.

If you're optimizing for ease-of-use over reliability, as you might in a scripting language, your conclusions will differ significantly. Languages like Java and C# struggle because they are right at the crossroads of scenarios – sometimes being used for systems, sometimes being used for applications – but overall their Error Models were very unsuitable for our needs.

Finally, also recall that this story began in the mid-2000s timeframe, before Go, Rust, and Swift were available for our consideration. These three languages have done some great things with Error Models since then.

Error Codes

Error codes are arguably the simplest Error Model possible. The idea is very basic and doesn't even require language or runtime support. A function just returns a value, usually an integer, to indicate success or failure:

```
int foo() {
    // <try something here>
    if (failed) {
        return 1;
    }
    return 0;
}
```

This is the typical pattern, where a return of `0` means success and non-zero means failure. A caller must check it:

```
int err = foo();
if (err) {
    // Error!  Deal with it.
}
```

Most systems offer constants representing the set of error codes rather than magic numbers. There may or may not be functions you can use to get extra information about the most recent error (like `errno` in standard C and `GetLastError` in Win32). A return code really isn't anything special in the language – it's just a return value.

C has long used error codes. As a result, most C-based ecosystems do. More low-level systems code has been written using the return code discipline than any other. Linux does, as do countless mission-critical and realtime systems. So it's fair to say they have an impressive track record going for them!

On Windows, `HRESULT`s are equivalent. An `HRESULT` is just an integer "handle" and there are a bunch of constants and macros in `winerror.h` like `S_OK`, `E_FAULT`, and `SUCCEEDED()`, that are used to create and check values. The most important code in Windows is written using a return code discipline. No exceptions are to be found in the kernel. At least not intentionally.

In environments with manual memory management, deallocating memory on error is uniquely difficult. Return codes can make this (more) tolerable. C++ has more automatic ways of doing this using RAII, but unless you buy into the C++ model whole hog – which a fair number of systems programmers don't – then there's no good way to incrementally use RAII in your C programs.

More recently, Go has chosen error codes. Although Go's approach is similar to C's, it has been modernized with much nicer syntax and libraries.

Many functional languages use return codes disguised in monads and named things like `Option<T>`, `Maybe<T>`, or `Error<T>`, which, when coupled with a dataflow-style of programming and pattern matching, feel far more natural. This approach removes several major drawbacks to return codes that we're about to discuss, especially compared to C. Rust has largely adopted this model but has dome some exciting things with it for systems programmers.

Despite their simplicity, return codes do come with some baggage; in summary:

- Performance can suffer.
- Programming model usability can be poor.
- The biggie: You can accidentally forget to check for errors.

Let's discuss each one, in order, with examples from the languages cited above.

Performance

Error codes fail the test of "zero overhead for common cases; pay for play for uncommon cases":

1. There is calling convention impact. You now have *two* values to return (for non-`void` returning functions): the actual return value and the possible error. This burns more registers and/or stack space, making calls less efficient. Inlining can of course help to recover this for the subset of calls that can be inlined.

2. There are branches injected into callsites anywhere a callee can fail. I call costs like this "peanut butter," because the checks are smeared across the code, making it difficult to measure the impact directly. In Midori we were able to experiment and measure, and confirm that yes, indeed, the cost here is nontrivial. There is also a secondary effect which is, because functions contain more branches, there is more risk of confusing the optimizer.

This might be surprising to some people, since undoubtedly everyone has heard that "exceptions are slow." It turns out that they don't have to be. And, when done right, they get error handling code and data off hot paths which increases I-cache and TLB performance, compared to the overheads above, which obviously decreases them.

Many high performance systems have been built using return codes, so you might think I'm nitpicking. As with many things we did, an easy criticism is that we took too extreme an approach. But the baggage gets worse.

Forgetting to Check Them

It's easy to forget to check a return code. For example, consider a function:

```
int foo() { ... }
```

Now at the callsite, what if we silently ignore the returned value entirely, and just keep going?

```
foo();
// Keep going -- but foo might have failed!
```

At this point, you've masked a potentially critical error in your program. This is easily the most vexing and damaging problem with error codes. As I will show later, option types help to address this for functional languages. But in C-based languages, and even Go with its modern syntax, this is a real issue.

This problem isn't theoretical. I've encountered numerous bugs caused by ignoring return codes and I'm sure you have too. Indeed, in the development of this very Error Model, my team encountered some fascinating ones. For example, when we ported Microsoft's Speech Server to Midori, we found that 80% of Taiwan Chinese (`zh-tw`) requests were failing. Not failing in a way the developers immediately saw, however; instead, clients would get a gibberish response. At first, we thought it was our fault. But then we discovered a silently swallowed `HRESULT` in the original code. Once we got it over to Midori, the bug was thrown into our faces, found, and fixed immediately after porting. This experience certainly informed our opinion about error codes.

It's surprising to me that Go made unused `import`s an error, and yet missed this far more critical one. So close!

It's true you can add a static analysis checker, or maybe an "unused return value" warning as most commercial C++ compilers do. But once you've missed the opportunity to add it to the core of the language, as a requirement, none of those techniques will reach critical mass due to complaints about noisy analysis.

For what it's worth, forgetting to use return values in our language was a compile time error. You had to explicitly ignore them; early on we used an API for this, but eventually devised language syntax – the equivalent of `>/dev/null`:

```
ignore foo();
```

We didn't use error codes, however the inability to accidentally ignore a return value was important for the overall reliability of the system. How many times have you debugged a problem only to find that the root cause was a return value you forgot to use? There have even been security exploits where this was the root cause. Letting developers say `ignore` wasn't bulletproof, of course, as they could still do the wrong thing. But it was at least explicit and auditable.

Programming Model Usability

In C-based languages with error codes, you end up writing lots of hand-crafted `if` checks everywhere after function calls. This can be especially tedious if many of your functions fail which, in C programs where allocation failures are also communicated with return codes, is frequently the case. It's also clumsy to return multiple values.

A warning: this complaint is subjective. In many ways, the usability of return codes is actually elegant. You reuse very simple primitives – integers, returns, and `if` branches – that are used in myriad other situations. In my humble opinion, errors are an important enough aspect of programming that the language should be helping you out.

Go has a nice syntactic shortcut to make the standard return code checking *slightly* more pleasant:

```
if err := foo(); err != nil {
    // Deal with the error.
}
```

Notice that we've invoked `foo` and checked whether the error is non-`nil` in one line. Pretty neat.

The usability problems don't stop there, however.

It's common that many errors in a given function should share some recovery or remediation logic. Many C programmers use labels and `goto`s to structure such code. For example:

```
int error;

// ...

error = step1();
if (error) {
    goto Error;
}

// ...

error = step2();
if (error) {
    goto Error;
}

// ...

// Normal function exit.
return 0;

// ...
Error:
// Do something about `error`.
return error;
```

Needless to say, this is the kind of code only a mother could love.

In languages like D, C#, and Java, you have `finally` blocks to encode this "before scope exits" pattern more directly. Similarly, Microsoft's proprietary extensions to C++ offer `__finally`, even if you're not fully buying into RAII and exceptions. And D provides `scope` and Go offers `defer`. All of these help to eradicate the `goto Error` pattern.

Next, imagine my function wants to return a real value *and* the possibility of an error? We've burned the return slot already so there are two obvious possibilities:

1. We can use the return slot for one of the two values (commonly the error), and another slot – like a pointer parameter – for the other of the two (commonly the real value). This is the common approach in C.

2. We can return a data structure that carries the possibility of both in its very structure. As we will see, this is common in functional languages. But in a language like C, or Go even, that lacks parametric polymorphism, you lose typing information about the returned value, so this is less common to see. C++ of course adds templates, so in principle it could do this, however because it adds exceptions, the ecosystem around return codes is lacking.

In support of the performance claims above, imagine what both of these do to your program's resulting assembly code.

Returning Values "On The Side"

An example of the first approach in C looks like this:

```
int foo(int* out) {
    // <try something here>
    if (failed) {
        return 1;
    }
    *out = 42;
    return 0;
}
```

The real value has to be returned "on the side," making callsites clumsy:

```
int value;
int ret = foo(&value);
if (ret) {
    // Error!  Deal with it.
}
else {
    // Use value...
}
```

In addition to being clumsy, this pattern perturbs your compiler's definite assignment analysis which impairs your ability to get good warnings about things like using uninitialized values.

Go also takes aim at this problem with nicer syntax, thanks to multi-valued returns:

```
func foo() (int, error) {
    if failed {
        return 0, errors.New("Bad things happened")
    }
    return 42, nil
}
```

And callsites are much cleaner as a result. Combined with the earlier feature of single-line `if` checking for errors – a subtle twist, since at first glance the `value` return wouldn't be in scope, but it is – this gets a touch nicer:

```
if value, err := foo(); err != nil {
    // Error!  Deal with it.
} else {
    // Use value ...
}
```

Notice that this also helps to remind you to check the error. It's not bulletproof, however, because functions can return an error and nothing else, at which point forgetting to check it is just as easy as it is in C.

As I mentioned above, some would argue against me on the usability point. Especially Go's designers, I suspect. A big appeal to Go using error codes is as a rebellion against the overly complex languages in today's landscape. We have lost a lot of what makes C so elegant – that you can usually look at any line of code and guess what machine code it translates into. I won't argue against these points. In fact, I vastly prefer Go's model over both unchecked exceptions and Java's incarnation of checked exceptions. Even as I write this post, having written lots of Go lately, I look at Go's simplicity and wonder, did we go too far with all the `try` and `requires` and so on that you'll see shortly? I'm not sure. Go's error model tends to be one of the most divisive aspect of the language; it's probably largely because you can't be sloppy with errors as in most languages, however programmers really did enjoy writing code in Midori's. In the end, it's hard to compare them. I'm convinced both can be used to write reliable code.

Return Values in Data Structures

Functional languages address many of the usability challenges by packaging up the possibility of *either* a value *or* an error, into a single data structure. Because you're forced to pick apart the error from the value if you want to do anything useful with the value at the callsite – which, thanks to a dataflow style of programming, you probably will – it's easy to avoid the killer problem of forgetting to check for errors.

For an example of a modern take on this, check out Scala's `Option` type. The unfortunate news is that some languages, like those in the ML family and even Scala (thanks to its JVM heritage), mix this elegant model with the world of unchecked exceptions. This taints the elegance of the monadic data structure approach.

Haskell does something even cooler and gives the illusion of exception handling while still using error values and local control flow:

> *There is an old dispute between C++ programmers on whether exceptions or error return codes are the right way. Niklas Wirth considered exceptions to be the reincarnation of GOTO and thus omitted them in his languages. Haskell solves this problem a diplomatic way: Functions return error codes, but the handling of error codes does not uglify the code.*

The trick here is to support all the familiar `throw` and `catch` patterns, but using monads rather than control flow.

Although Rust also uses error codes it is also in the style of the functional error types. For example, imagine we are writing a function named `bar` in Go: we'd like to call `foo`, and then simply propagate the error to our caller if it fails:

```
func bar() error {
    if value, err := foo(); err != nil {
        return err
    } else {
        // Use value ...
    }
}
```

The longhand version in Rust isn't any more concise. It might, however, send C programmers reeling with its foreign pattern matching syntax (a real concern but not a dealbreaker). Anyone comfortable with functional programming, however, probably won't even blink, and this approach certainly serves as a reminder to deal with your errors:

```
fn bar() -> Result<(), Error> {
    match foo() {
        Ok(value) => /* Use value ... */,
        Err(err) => return Err(err)
    }
}
```

But it gets better. Rust has a `try!` macro that reduces boilerplate like the most recent example to a single expression:

```
fn bar() -> Result<(), Error> {
    let value = try!(foo);
    // Use value ...
}
```

This leads us to a beautiful sweet spot. It does suffer from the performance problems I mentioned earlier, but does very well on all other dimensions. It alone is an incomplete picture – for that, we need to cover fail-fast (a.k.a. abandonment) – but as we will see, it's far better than any other exception-based model in widespread use today.

Exceptions

The history of exceptions is fascinating. During this journey I spent countless hours retracing the industry's steps. That includes reading some of the original papers – like Goodenough's 1975 classic, Exception Handling: Issues and a Proposed Notation – in addition to looking at the approaches of several languages: Ada, Eiffel, Modula-2 and 3, ML, and, most inspirationally, CLU. Many papers do a better job than I can summarizing the long and arduous journey, so I won't do that here. Instead, I'll focus on what works and what doesn't work for building reliable systems.

Reliability is the most important of our requirements above when developing the Error Model. If you can't react appropriately to failures, your system, by definition, won't be very reliable. Operating systems generally speaking need to be reliable. Sadly, the most commonplace model – unchecked exceptions – is the worst you can do in this dimension.

For these reasons, most reliable systems use return codes instead of exceptions. They make it possible to locally reason about and decide how best to react to error conditions. But I'm getting ahead of myself. Let's dig in.

Unchecked Exceptions

A quick recap. In an unchecked exceptions model, you `throw` and `catch` exceptions, without it being part of the type system or a function's signature. For example:

```
// Foo throws an unhandled exception:
void Foo() {
    throw new Exception(...);
}

// Bar calls Foo, and handles that exception:
void Bar() {
    try {
        Foo();
    }
    catch (Exception e) {
        // Handle the error.
    }
}

// Baz also calls Foo, but does not handle that exception:
void Baz() {
    Foo(); // Let the error escape to our callers.
}
```

In this model, any function call – and sometimes any *statement* – can throw an exception, transferring control non-locally somewhere else. Where? Who knows. There are no annotations or type system artifacts to guide your analysis. As a result, it's difficult for anyone to reason about a program's state at the time of the throw, the state changes that occur while that exception is propagated up the call stack – and possibly across threads in a concurrent program – and the resulting state by the time it gets caught or goes unhandled.

It's of course possible to try. Doing so requires reading API documentation, doing manual audits of the code, leaning heavily on code reviews, and a healthy dose of luck. The language isn't helping you out one bit here. Because failures are rare, this tends not to be as utterly disastrous as it sounds. My conclusion is that's why many people in the industry think unchecked exceptions are "good enough." They stay out of your way for the common success paths and, because most people don't write robust error handling code in non-systems programs, throwing an exception *usually* gets you out of a pickle fast. Catching and then proceeding often works too. No harm, no foul. Statistically speaking, programs "work."

Maybe statistical correctness is okay for scripting languages, but for the lowest levels of an operating system, or any mission critical application or service, this isn't an appropriate solution. I hope this isn't controversial.

.NET makes a bad situation even worse due to *asynchronous exceptions*. C++ has so-called "asynchronous exceptions" too: these are failures that are triggered by hardware faults, like access violations. It gets really nasty in .NET, however. An arbitrary thread can inject a failure at nearly any point in your code. Even between the RHS and LHS of an assignment! As a result, things that look atomic in source code aren't. I wrote about this 10 years ago and the challenges still exist, although the risk has lessened as .NET generally learned that thread aborts are problematic. The new CoreCLR even lacks AppDomains, and the new ASP.NET Core 1.0 stack certainly doesn't use thread aborts like it used to. But the APIs are still there.

There's a famous interview with Anders Hejlsberg, C#'s chief designer, called The Trouble with Checked Exceptions. From a systems programmer's perspective, much of it leaves you scratching your head. No statement affirms that the target customer for C# was the rapid application developer more than this:

> *Bill Venners: But aren't you breaking their code in that case anyway, even in a language without checked exceptions? If the new version of foo is going to throw a new exception that clients should think about handling, isn't their code broken just by the fact that they didn't expect that exception when they wrote the code?*
>
> *Anders Hejlsberg : No, because in a lot of cases, people don't care. They're not going to handle any of these exceptions. There's a bottom level exception handler around their message loop. That handler is just going to bring up a dialog that says what went wrong and continue. The programmers protect their code by writing try finally's everywhere, so they'll back out correctly if an exception occurs, but they're not actually interested in handling the exceptions.*

This reminds me of `On Error Resume Next` in Visual Basic, and the way Windows Forms automatically caught and swallowed errors thrown by the application, and attempted to proceed. I'm not blaming Anders for his viewpoint here; heck, for C#'s wild popularity, I'm convinced it was the right call given the climate at the time. But this sure isn't the way to write operating system code.

C++ at least *tried* to offer something better than unchecked exceptions with its `throw` exception specifications. Unfortunately, the feature relied on dynamic enforcement which sounded its death knell instantaneously.

If I write a function `void f() throw(SomeError)`, the body of `f` is still free to invoke functions that throw things other than `SomeError`. Similarly, if I state that `f` throws no exceptions, using `void f() throw()`, it's still possible to invoke things that throw. To implement the stated contract, therefore, the compiler and runtime must ensure that, should this happen, `std::unexpected` is called to rip the process down in response.

I'm not the only person to recognize this design was a mistake. Indeed, `throw` is now deprecated. A detailed WG21 paper, Deprecating Exception Specifications, describes how C++ ended up here, and has this to offer in its opening statement:

> *Exception specifications have proven close to worthless in practice, while adding a measurable overhead to programs.*

The authors list three reasons for deprecating `throw`. Two of the three reasons were a result of the dynamic choice: runtime checking (and its associated opaque failure mode) and runtime performance overheads. The third reason, lack of composition in generic code, could have been dealt with using a proper type system (admittedly at an expense).

But the worst part is that the cure relies on yet another dynamically enforced construct – the `noexcept` specifier – which, in my opinion, is just as bad as the disease.

"Exception safety" is a commonly discussed practice in the C++ community. This approach neatly classifies how functions are intended to behave from a caller's perspective with respect to failure, state transitions, and memory management. A function falls into one of four kinds: *no-throw* means forward progress is guaranteed and no exceptions will emerge; *strong safety* means that state transitions happen atomically and a failure will not leave behind partially committed state or broken invariants; *basic safety* means that, though a function might partially commit state changes, invariants will not be broken and leaks are prevented; and finally, *no safety* means anything's possible. This taxonomy is quite helpful and I encourage anyone to be intentional and rigorous about error behavior, either using this approach or something similar. Even if you're using error codes. The problem is, it's essentially impossible to follow these guidelines in a system using unchecked exceptions, except for leaf node data structures that call a small and easily auditable set of other functions. Just think about it: to guarantee strong safety everywhere, you would need to consider the possibility of *all function calls throwing*, and safeguard the surrounding code accordingly. That either means programming defensively, trusting another function's documented English prose (that isn't being checked by a computer), getting lucky and only calling `noexcept` functions, or just hoping for the best. Thanks to RAII, the leak-freedom aspect of basic safety is easier to attain – and pretty common these days thanks to smart pointers – but even broken invariants are tricky to prevent. The article Exception Handling: A False Sense of Security sums this up well.

For C++, the real solution is easy to predict, and rather straightforward: for robust systems programs, don't use exceptions. That's the approach Embedded C++ takes, in addition to numerous realtime and mission critical guidelines for C++, including NASA's Jet Propulsion Laboratory's. C++ on Mars sure ain't using exceptions anytime soon.

So if you can safely avoid exceptions and stick to C-like return codes in C++, what's the beef?

The entire C++ ecosystem uses exceptions. To obey the above guidance, you must avoid significant parts of the language and, it turns out, significant chunks of the library ecosystem. Want to use the Standard Template Library? Too bad, it uses exceptions. Want to use Boost? Too bad, it uses exceptions. Your allocator likely throws `bad_alloc`. And so on. This even causes insanity like people creating forks of existing libraries that eradicates exceptions. The Windows kernel, for instance, has its own fork of the STL that doesn't use exceptions. This bifurcation of the ecosystem is neither pleasant nor practical to sustain.

This mess puts us in a bad spot. Especially because many languages use unchecked exceptions. It's clear that they are ill-suited for writing low-level, reliable systems code. (I'm sure I will make a few C++ enemies by saying this so bluntly.) After writing code in Midori for years, it brings me tears to go back and write code that uses unchecked exceptions; even simply code reviewing is torture. But "thankfully" we have checked exceptions from Java to learn and borrow from … Right?

Checked Exceptions

Ah, checked exceptions. The rag doll that nearly every Java programmer, and every person who's observed Java from an arm's length distance, likes to beat on. Unfairly so, in my opinion, when you compare it to the unchecked exceptions mess.

In Java, you know *mostly* what a method might throw, because a method must say so:

```
void foo() throws FooException, BarException {
    ...
}
```

Now a caller knows that invoking `foo` could result in either `FooException` or `BarException` being thrown. At callsites a programmer must now decide: 1) propagate thrown exceptions as-is, 2) catch and deal with them, or 3) somehow transform the type of exception being thrown (possibly even "forgetting" the type altogether). For instance:

```
// 1) Propagate exceptions as-is:
void bar() throws FooException, BarException {
    foo();
}

// 2) Catch and deal with them:
void bar() {
    try {
        foo();
    }
    catch (FooException e) {
        // Deal with the FooException error conditions.
    }
    catch (BarException e) {
        // Deal with the BarException error conditions.
    }
}

// 3) Transform the exception types being thrown:
void bar() throws Exception {
    foo();
}
```

This is getting much closer to something we can use. But it fails on a few accounts:

1. Exceptions are used to communicate unrecoverable bugs, like null dereferences, divide-by-zero, etc.
2. You don't actually know *everything* that might be thrown, thanks to our little friend `RuntimeException`. Because Java uses exceptions for all error conditions – even bugs, per above – the designers realized people would go mad with all those exception specifications. And so they introduced a kind of exception that is unchecked. That is, a method can throw it without declaring it, and so callers can invoke it seamlessly.
3. Although signatures declare exception types, there is no indication at callsites what calls might throw.
4. People hate them.

That last one is interesting, and I shall return to it later when describing the approach Midori took. In summary, peoples' distaste for checked exceptions in Java is largely derived from, or at least significantly reinforced by, the other three bullets above. The resulting model seems to be the worst of both worlds. It doesn't help you to write bulletproof code and it's hard to use. You end up writing down a lot of gibberish in your code for little perceived benefit. And versioning your interfaces is a pain in the ass. As we'll see later, we can do better.

That versioning point is worth a ponder. If you stick to a single kind of `throw`, then the versioning problem is no worse than error codes. Either a function fails or it doesn't. It's true that if you design version 1 of your API to have no failure mode, and then want to add failures in version 2, you're screwed. As you should be, in my opinion. An API's failure mode is a critical part of its design and contract with callers. Just as you wouldn't change the return type of an API silently without callers needing to know, you shouldn't change its failure mode in a semantically meaningful way. More on this controversial point later on.

CLU has an interesting approach, as described in this crooked and wobbly scan of a 1979 paper by Barbara Liskov, Exception Handling in CLU. Notice that they focus a lot on "linguistics"; in other words, they wanted a language that people would love. The need to check and repropagate all errors at callsites felt a lot more like return values, yet the programming model had that richer and slightly declarative feel of what we now know as exceptions. And most importantly, `signal`s (their name for `throw`) were checked. There were also convenient ways to terminate the program should an unexpected `signal` occur.

Universal Problems with Exceptions

Most exception systems get a few major things wrong, regardless of whether they are checked or unchecked.

First, throwing an exception is usually ridiculously expensive. This is almost always due to the gathering of a stack trace. In managed systems, gathering a stack trace also requires groveling metadata, to create strings of function symbol names. If the error is caught and handled, however, you don't even need that information at runtime! Diagnostics are better implemented in the logging and diagnostics infrastructure, not the exception system itself. The concerns are orthogonal. Although, to really nail the diagnostics requirement above, *something* needs to be able to recover stack traces; never underestimate the power of `printf` debugging and how important stack traces are to it.

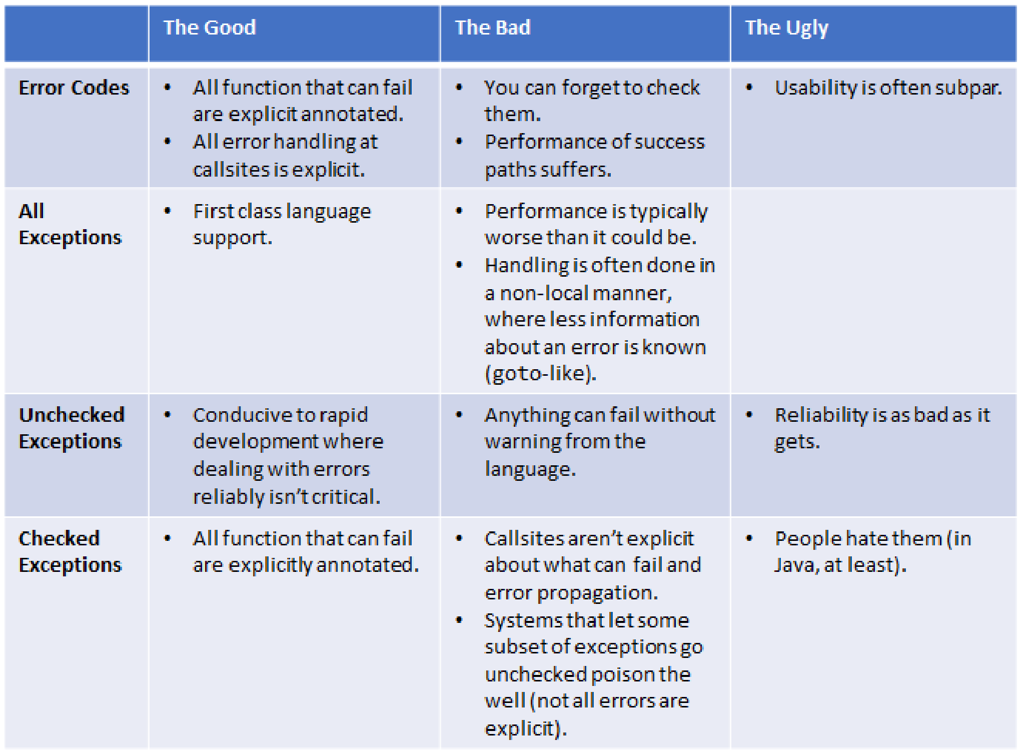
Next, exceptions can significantly impair code quality. I touched on this topic in my last post, and there are good papers on the topic in the context of C++. Not having static type system information makes it hard to model control flow in the compiler, which leads to overly conservative optimizers.

Another thing most exception systems get wrong is encouraging too coarse a granularity of handling errors. Proponents of return codes love that error handling is localized to a specific function call. (I do too!) In exception handling systems, it's all too easy to slap a coarse-grained `try`/`catch` block around some huge hunk of code, without carefully reacting to individual failures. That produces brittle code that's almost certainly wrong; if not today, then after the inevitable refactoring that will occur down the road. A lot of this has to do with having the right syntaxes.

Finally, control flow for `throw`s is usually invisible. Even with Java, where you annotate method signatures, it's not possible to audit a body of code and see precisely where exceptions come from. Silent control flow is just as bad as `goto`, or `setjmp`/`longjmp`, and makes writing reliable code very difficult.

Where Are We?

Before moving on, let's recap where we are:



	The Good	The Bad	The Ugly
Error Codes	• All function that can fail are explicit annotated. • All error handling at callsites is explicit.	• You can forget to check them. • Performance of success paths suffers.	• Usability is often subpar.
All Exceptions	• First class language support.	• Performance is typically worse than it could be. • Handling is often done in a non-local manner, where less information about an error is known (goto-like).	
Unchecked Exceptions	• Conducive to rapid development where dealing with errors reliably isn't critical.	• Anything can fail without warning from the language.	• Reliability is as bad as it gets.
Checked Exceptions	• All function that can fail are explicitly annotated.	• Callsites aren't explicit about what can fail and error propagation. • Systems that let some subset of exceptions go unchecked poison the well (not all errors are explicit).	• People hate them (in Java, at least).

Wouldn't it be great if we could take all of The Goods and leave out The Bads and The Uglies?

This alone would be a great step forward. But it's insufficient. This leads me to our first big "ah-hah" moment that shaped everything to come. For a significant class of error, *none* of these approaches are appropriate!

Bugs Aren't Recoverable Errors!

A critical distinction we made early on is the difference between recoverable errors and bugs:

- A *recoverable error* is usually the result of programmatic data validation. Some code has examined the state of the world and deemed the situation unacceptable for progress. Maybe it's some markup text being parsed, user input from a website, or a transient network connection failure. In these cases, programs are expected to recover. The developer who wrote this code must think about what to do in the event of failure because it will happen in well-constructed programs no matter what you do. The response might be to communicate the situation to an end-user, retry, or abandon the operation entirely, however it is a *predictable* and, frequently, *planned* situation, despite being called an "error."

- A *bug* is a kind of error the programmer didn't expect. Inputs weren't validated correctly, logic was written wrong, or any host of problems have arisen. Such problems often aren't even detected promptly; it takes a while until "secondary effects" are observed indirectly, at which point significant damage to the program's state might have occurred. Because the developer didn't expect this to happen, all bets are off. All data structures reachable by this code are now suspect. And because these problems aren't necessarily detected promptly, in fact, a whole lot more is suspect. Depending on the isolation guarantees of your language, perhaps the entire process is tainted.

This distinction is paramount. Surprisingly, most systems don't make one, at least not in a principled way! As we saw above, Java, C#, and dynamic languages just use exceptions for everything; and C and Go use return codes. C++ uses a mixture depending on the audience, but the usual story is a project picks a single one and uses it everywhere. You usually don't hear of languages suggesting *two* different techniques for error handling, however.

Given that bugs are inherently not recoverable, we made no attempt to try. All bugs detected at runtime caused something called *abandonment*, which was Midori's term for something otherwise known as "fail-fast".

Each of the above systems offers abandonment-like mechanisms. C# has `Environment.FailFast`; C++ has `std::terminate`; Go has `panic`; Rust has `panic!`; and so on. Each rips down the surrounding context abruptly and promptly. The scope of this context depends on the system – for example, C# and C++ terminate the process, Go the current Goroutine, and Rust the current thread, optionally with a panic handler attached to salvage the process.

Although we did use abandonment in a more disciplined and ubiquitous way than is common, we certainly weren't the first to recognize this pattern. This Haskell essay, articulates this distinction quite well:

> *I was involved in the development of a library that was written in C++. One of the developers told me that the developers are divided into the ones who like exceptions and the other ones who prefer return codes. As it seem to me, the friends of return codes won. However, I got the impression that **they debated the wrong point: Exceptions and return codes are equally expressive**, they should however not be used to describe errors. Actually the return codes contained definitions like `ARRAY_INDEX_OUT_OF_RANGE`. But I wondered: How shall my function react, when it gets this return code from a subroutine? Shall it send a mail to its programmer? It could return this code to its caller in turn, but it will also not know how to cope with it. Even worse, since I cannot make assumptions about the implementation of a function, I have to expect an `ARRAY_INDEX_OUT_OF_RANGE` from every subroutine. My conclusion is that `ARRAY_INDEX_OUT_OF_RANGE` is a (programming) error. **It cannot be handled or fixed at runtime, it can only be fixed by its developer. Thus there should be no according return code, but instead there should be asserts.***

Abandoning fine grained mutable shared memory scopes is suspect – like Goroutines or threads or whatever – unless your system somehow makes guarantees about the scope of the potential damage done. However, it's great that these mechanisms are there for us to use! It means using an abandonment discipline in these languages is indeed possible.

There are architectural elements necessary for this approach to succeed at scale, however. I'm sure you're thinking "If I tossed the entire process each time I had a null dereference in my C# program, I'd have some pretty pissed off customers"; and, similarly, "That wouldn't be reliable at all!" Reliability, it turns out, might not be what you think.

Reliability, Fault-Tolerance, and Isolation

Before we get any further, we need to state a central belief: ~~Shi~~ Failure Happens.

To Build a Reliable System

Common wisdom is that you build a reliable system by systematically guaranteeing that failure can never happen. Intuitively, that makes a lot of sense. There's one problem: in the limit, it's impossible. If you can spend millions of dollars on this property alone – like many mission critical, realtime systems do – then you can make a significant dent. And perhaps use a language like SPARK (a set of contract-based extensions to Ada) to formally prove the correctness of each line written. However, experience shows that even this approach is not foolproof.

Rather than fighting this fact of life, we embraced it. Obviously you try to eliminate failures where possible. The error model must make them transparent and easy to deal with. But more importantly, you architect your system so that the whole remains functional even when individual pieces fail, and then teach your system to recover those failing pieces gracefully. This is well known in distributed systems. So why is it novel?

At the center of it all, an operating system is just a distributed network of cooperating processes, much like a distributed cluster of microservices or the Internet itself. The main differences include things like latency; what levels of trust you can establish and how easily; and various assumptions about locations, identity, etc. But failure in highly asynchronous, distributed, and I/O intensive systems is just bound to happen. My impression is that, largely because of the continued success of monolithic kernels, the world at large hasn't yet made the leap to “operating system as a distributed system” insight. Once you do, however, a lot of design principles become apparent.

As with most distributed systems, our architecture assumed process failure was inevitable. We went to great length to defend against cascading failures, journal regularly, and to enable restartability of programs and services.

You build things differently when you go in assuming this.

In particular, isolation is critical. Midori’s process model encouraged lightweight fine-grained isolation. As a result, programs and what would ordinarily be “threads” in modern operating systems were independent isolated entities. Safeguarding against failure of one such connection is far easier than when sharing mutable state in an address space.

Isolation also encourages simplicity. Butler Lampson’s classic Hints on Computer System Design explores this topic. And I always loved this quote from Hoare:

> *The unavoidable price of reliability is simplicity. (C. Hoare).*

By keeping programs broken into smaller pieces, each of which can fail or succeed on its own, the state machines within them stay simpler. As a result, recovering from failure is easier. In our language, the points of possible failure were explicit, further helping to keep those internal state machines correct, and pointing out those connections with the messier outside world. In this world, the price of individual failure is not nearly as dire. I can’t over-emphasize this point. None of the language features I describe later would have worked so well without this architectural foundation of cheap and ever-present isolation.

Erlang has been very successful at building this property into the language in a fundamental way. It, like Midori, leverages lightweight processes connected by message passing, and encourages fault-tolerant architectures. A common pattern is the “supervisor,” where some processes are responsible for watching and, in the event of failure, restarting other processes. This article does a terrific job articulating this philosophy – “let it crash” – and recommended techniques for architecting reliable Erlang programs in practice.

The key thing, then, is not preventing failure per se, but rather knowing how and when to deal with it.

Once you’ve established this architecture, you beat the hell out of it to make sure it works. For us, this meant week-long stress runs, where processes would come and go, some due to failures, to ensure the system as a whole kept making good forward progress. This reminds me of systems like Netflix’s Chaos Monkey which just randomly kills entire machines in your cluster to ensure the service as a whole stays healthy.

I expect more of the world to adopt this philosophy as the shift to more distributed computing happens. In a cluster of microservices, for example, the failure of a single container is often handled seamlessly by the enclosing cluster management software (Kubernetes, Amazon EC2 Container Service, Docker Swarm, etc). As a result, what I describe in this post is possibly helpful for writing more reliable Java, Node.js/JavaScript, Python, and even Ruby services. The unfortunate news is you’re likely going to be fighting your languages to get there. A lot of code in your process is going to work real damn hard to keep limping along when something goes awry.

Abandonment

Even when processes are cheap and isolated and easy to recreate, it’s still reasonable to think that abandoning an entire process in the face of a bug is an overreaction. Let me try to convince you otherwise.

Proceeding in the face of a bug is dangerous when you’re trying to build a robust system. If a programmer didn’t expect a given situation that’s arisen, who knows whether the code will do the right thing anymore. Critical data structures may have been left behind in an incorrect state. As an extreme (and possibly slightly silly) example, a routine that is meant to round your numbers *down* for banking purposes might start rounding them *up*.

And you might be tempted to whittle down the granularity of abandonment to something smaller than a process. But that's tricky. To take an example, imagine a thread in your process encounters a bug, and fails. This bug might have been triggered by some state stored in a static variable. Even though some other thread might *appear* to have been unaffected by the conditions leading to failure, you cannot make this conclusion. Unless some property of your system – isolation in your language, isolation of the object root-sets exposed to independent threads, or something else – it's safest to assume that anything other than tossing the entire address space out the window is risky and unreliable.

Thanks to the lightweight nature of Midori processes, abandoning a process was more like abandoning a single thread in a classical system than a whole process. But our isolation model let us do this reliably.

I'll admit the scoping topic is a slippery slope. Maybe all the data in the world has become corrupt, so how do you know that tossing the process is even enough?! There is an important distinction here. Process state is transient by design. In a well designed system it can be thrown away and recreated on a whim. It's true that a bug can corrupt persistent state, but then you have a bigger problem on your hands – a problem that must be dealt with differently.

For some background, we can look to fault-tolerant systems design. Abandonment (fail-fast) is already a common technique in that realm, and we can apply much of what we know about these systems to ordinary programs and processes. Perhaps the most important technique is regularly journaling and checkpointing precious persistent state. Jim Gray's 1985 paper, Why Do Computers Stop and What Can Be Done About It?, describes this concept nicely. As programs continue moving to the cloud, and become aggressively decomposed into smaller independent services, this clear separation of transient and persistent state is even more important. As a result of these shifts in how software is written, abandonment is far more achievable in modern architectures than it once was. Indeed, abandonment can help you avoid data corruption, because bugs detected before the next checkpoint prevent bad state from ever escaping.

Bugs in Midori's kernel were handled differently. A bug in the microkernel, for instance, is an entirely different beast than a bug in a user-mode process. The scope of possible damage was greater, and the safest response was to abandon an entire "domain" (address space). Thankfully, most of what you'd think of being classic "kernel" functionality – the scheduler, memory manager, filesystem, networking stack, and even device drivers – was run instead in isolated processes in user-mode where failures could be contained in the usual ways described above.

Bugs: Abandonment, Assertions, and Contracts

A number of kinds of bugs in Midori might trigger abandonment:

- An incorrect cast.
- An attempt to dereference a `null` pointer.
- An attempt to access an array outside of its bounds.
- Divide-by-zero.
- An unintended mathematical over/underflow.
- Out-of-memory.
- Stack overflow.
- Explicit abandonment.
- Contract failures.
- Assertion failures.

Our fundamental belief was that each is a condition the program cannot recover from. Let's discuss each one.

Plain Old Bugs

Some of these situations are unquestionably indicative of a program bug.

An incorrect cast, attempt to dereference `null`, array out-of-bounds access, or divide-by-zero are clearly problems with the program's logic, in that it attempted an undeniably illegal operation. As we will see later, there are ways out (e.g., perhaps you want NaN-style propagation for DbZ). But by default we assume it's a bug.

Most programmers were willing to accept this without question. And dealing with them as bugs this way brought abandonment to the inner development loop where bugs during development could be found and fixed fast. Abandonment really did help to make people more productive at writing code. This was a surprise to me at first, but it makes sense.

Some of these situations, on the other hand, are subjective. We had to make a decision about the default behavior, often with controversy, and sometimes offer programmatic control.

Arithmetic Over/Underflow

Saying an unintended arithmetic over/underflow represents a bug is certainly a contentious stance. In an unsafe system, however, such things frequently lead to security vulnerabilities. I encourage you to review the National Vulnerability Database to see the sheer number of these.

In fact, the Windows TrueType Font parser, which we ported to Midori (with *gains* in performance), has suffered over a dozen of them in the past few years alone. (Parsers tend to be farms for security holes like this.)

This has given rise to packages like SafeInt, which essentially moves you away from your native language's arithmetic operations, in favor of checked library ones.

Most of these exploits are of course also coupled with an access to unsafe memory. You could reasonably argue therefore that overflows are innocuous in a safe language and therefore should be permitted. It's pretty clear, however, based on the security experience, that a program often does the wrong thing in the face of an unintended over/underflow. Simply put, developers frequently overlook the possibility, and the program proceeds to do unplanned things. That's the definition of a bug which is precisely what abandonment is meant to catch. The final nail in the coffin on this one is that philisophically, when there was any question about correctness, we tended to err on the side of explicit intent.

Hence, all unannotated over/underflows were considered bugs and led to abandonment. This was similar to compiling C# with the `/checked` switch, except that our compiler aggressively optimized redundant checks away. (Since few people ever think to throw this switch in C#, the code-generators don't do nearly as aggressive a job in removing the inserted checks.) Thanks to this language and compiler co-development, the result was far better than what most C++ compilers will produce in the face of SafeInt arithmetic. Also as with C#, the `unchecked` scoping construct could be used where over/underflow was intended.

Although the initial reactions from most C# and C++ developers I've spoken to about this idea are negative about it, our experience was that 9 times out of 10, this approach helped to avoid a bug in the program. That remaining 1 time was usually an abandonment sometime late in one of our 72 hour stress runs – in which we battered the entire system with browsers and multimedia players and anything else we could do to torture the system – when some harmless counter overflowed. I always found it amusing that we spent time fixing these instead of the classical way products mature through the stress program, which is to say deadlocks and race conditions. Between you and me, I'll take the overflow abandonments!

Out-of-Memory and Stack Overflow

Out-of-memory (OOM) is complicated. It always is. And our stance here was certainly contentious also.

In environments where memory is manually managed, error code-style of checking is the most common approach:

```
X* x = (X*)malloc(...);
if (!x) {
    // Handle allocation failure.
}
```

This has one subtle benefit: allocations are painful, require thought, and therefore programs that use this technique are often more frugal and deliberate with the way they use memory. But it has a huge downside: it's error prone and leads to huge amounts of frequently untested code-paths. And when code-paths are untested, they usually don't work.

Developers in general do a terrible job making their software work properly right at the edge of resource exhaustion. In my experience with Windows and the .NET Framework, this is where egregious mistakes get made. And it leads to ridiculously complex programming models, like .NET's so-called Constrained Execution Regions. A program limping along, unable to allocate even tiny amounts of memory, can quickly become the enemy of reliability. Chris Brumme's wondrous Reliability post describes this and related challenges in all its gory glory.

Parts of our system were of course "hardened" in a sense, like the lowest levels of the kernel, where abandonment's scope would be necessarily wider than a single process. But we kept this to as little code as possible.

For the rest? Yes, you guessed it: abandonment. Nice and simple.

It was surprising how much of this we got away with. I attribute most of this to the isolation model. In fact, we could *intentionally* let a process suffer OOM, and ensuing abandonment, as a result of resource management policy, and still remain confident that stability and recovery were built in to the overall architecture.

It was possible to opt-in to recoverable failure for individual allocations if you really wanted. This was not common in the slightest, however the mechanisms to support it were there. Perhaps the best motivating example is this: imagine your program wants to allocate a buffer of 1MB in size. This situation is different than your ordinary run-of-the-mill sub-1KB object allocation. A developer may very well be prepared to think and explicitly deal with the fact that a contiguous block of 1MB in size might not be available, and deal with it accordingly. For example:

```
var bb = try new byte[1024*1024] else catch;
if (bb.Failed) {
    // Handle allocation failure.
}
```

Stack overflow is a simple extension of this same philosophy. Stack is just a memory-backed resource. In fact, thanks to our asynchronous linked stacks model, running out of stack was physically identical to running out of heap memory, so the consistency in how it was dealt with was hardly surprising to developers. Many systems treat stack overflow this way these days.

Assertions

An assertion was a manual check in the code that some condition held true, triggering abandonment if it did not. As with most systems, we had both debug-only and release code assertions, however unlike most other systems, we had more release ones than debug. In fact, our code was peppered liberally with assertions. Most methods had multiple.

This kept with the philosophy that it's better to find a bug at runtime than to proceed in the face of one. And, of course, our backend compiler was taught how to optimize them aggressively as with everything else. This level of assertion density is similar to what guidelines for highly reliable systems suggest. For example, from NASA's paper, The Power of Ten -Rules for Developing Safety Critical Code:

> *Rule: The assertion density of the code should average to a minimum of two assertions per function. Assertions are used to check for anomalous conditions that should never happen in real-life executions. Assertions must always be side-effect free and should be defined as Boolean tests.*
>
> *Rationale: Statistics for industrial coding efforts indicate that unit tests often find at least one defect per 10 to 100 lines of code written. The odds of intercepting defects increase with assertion density. Use of assertions is often also recommended as part of strong defensive coding strategy.*

To indicate an assertion, you simply called `Debug.Assert` or `Release.Assert`:

```
void Foo() {
    Debug.Assert(something); // Debug-only assert.
    Release.Assert(something); // Always-checked assert.
}
```

We also implemented functionality akin to `__FILE__` and `__LINE__` macros like in C++, in addition to `__EXPR__` for the text of the predicate expression, so that abandonments due to failed assertions contained useful information.

In the early days, we used different "levels" of assertions than these. We had three levels, `Contract.Strong.Assert`, `Contract.Assert`, and `Contract.Weak.Assert`. The strong level meant "always checked," the middle one meant "it's up to the compiler," and the weak one meant "only checked in debug mode." I made the controversial decision to move away from this model. In fact, I'm pretty sure 49.99% of the team absolutely hated my choice of terminology (`Debug.Assert` and `Release.Assert`), but I always liked them because it's pretty unambiguous what they do. The problem with the old taxonomy was that nobody ever knew exactly when the assertions would be checked; confusion in this area is simply not acceptable, in my opinion, given how important good assertion discipline is to the reliability of one's program.

As we moved contracts to the language (more on that soon), we tried making `assert` a keyword too. However, we eventually switched back to using APIs. The primary reason was that assertions were *not* part of an API's signature like contracts are; and given that assertions could easily be implemented as a library, it wasn't clear what we gained from having them in the language. Furthermore, policies like "checked in debug" versus "checked in release" simply didn't feel like they belonged in a programming language. I'll admit, years later, I'm still on the fence about this.

Contracts

Contracts were *the* central mechanism for catching bugs in Midori. Despite us beginning with Singularity, which used Sing#, a variant of Spec#, we quickly moved away to vanilla C# and had to rediscover what we wanted. We ultimately ended up in a very different place after living with the model for years.

All contracts and assertions were proven side-effect free thanks to our language's understanding of immutability and side-effects. This was perhaps the biggest area of language innovation, so I'll be sure to write a post about it soon.

As with other areas, we were inspired and influenced by many other systems. Spec# is the obvious one. Eiffel was hugely influential especially as there are many published case studies to learn from. Research efforts like Ada-based SPARK and proposals for realtime and embedded systems too. Going deeper into the theoretical rabbit's hole, programming logic like Hoare's axiomatic semantics provide the foundation for all of it. For me, however, the most philosophical inspiration came from CLU's, and later Argus's, overall approach to error handling.

Preconditions and Postconditions

The most basic form of contract is a method precondition. This states what conditions must hold for the method to be dispatched. This is most often used to validate arguments. Sometimes it's used to validate the state of the target object, however this was generally frowned upon, since modality is a tough thing for programmers to reason about. A precondition is essentially a guarantee the caller provides to the callee.

In our final model, a precondition was stated using the `requires` keyword:

```
void Register(string name)
    requires !string.IsEmpty(name) {
```

```
    // Proceed, knowing the string isn't empty.
}
```

A slightly less common form of contract is a method postcondition. This states what conditions hold *after* the method has been dispatched. This is a guarantee the callee provides to the caller.

In our final model, a postcondition was stated using the `ensures` keyword:

```
void Clear()
    ensures Count == 0 {
    // Proceed; the caller can be guaranteed the Count is 0 when we return.
}
```

It was also possible to mention the return value in the postcondition, through the special name `return`. Old values – such as necessary for mentioning an input in a post-condition – could be captured through `old(..)`; for example:

```
int AddOne(int value)
    ensures return == old(value)+1 {
    ...
}
```

Of course, pre- and postconditions could be mixed. For example, from our ring buffer in the Midori kernel:

```
public bool PublishPosition()
    requires RemainingSize == 0
    ensures UnpublishedSize == 0 {
    ...
}
```

This method could safely execute its body knowing that `RemainingSize` is `0` and callers could safely execute after the return knowing that `UnpublishedSize` is also `0`.

If any of these contracts are found to be false at runtime, abandonment occurs.

This is an area where we differ from other efforts. Contracts have recently became popular as an expression of program logics used in advanced proof techniques. Such tools prove truths or falsities about stated contracts, often using global analysis. We took a simpler approach. By default, contracts are checked at runtime. If a compiler could prove truth or falsehood at compile-time, it was free to elide runtime checks or issue a compile-time error, respectively.

Modern compilers have constraint-based analyses that do a good job at this, like the range analysis I mentioned in my last post. These propagate facts and use them to optimize code already. This includes eliminating redundant checks: either explicitly encoded in contracts, or in normal program logic. And they are trained to perform these analyses in reasonable amounts of time, lest programmers switch to a different, faster compiler. The theorem proving techniques simply did not scale for our needs; our core system module took over a day to analyze using the best in breed theorem proving analysis framework!

Furthermore, the contracts a method declared were part of its signature. This meant they would automatically show up in documentation, IDE tooltips, and more. A contract was as important as a method's return and argument types. Contracts really were just an extension of the type system, using arbitrary logic in the language to control the shape of exchange types. As a result, all the usual subtyping requirements applied to them. And, of course, this facilitated modular local analysis which could be done in seconds using standard optimizing compiler techniques.

90-something% of the typical uses of exceptions in .NET and Java became preconditions. All of the `ArgumentNullException`, `ArgumentOutOfRangeException`, and related types and, more importantly, the manual checks and `throw`s were gone. Methods are often peppered with these checks in C# today; there are thousands of these in .NET's CoreFX repo alone. For example, here is `System.IO.TextReader`'s `Read` method:

```
/// <summary>
/// ...
/// </summary>
/// <exception cref="ArgumentNullException">Thrown if buffer is null.</excep
/// <exception cref="ArgumentOutOfRangeException">Thrown if index is less th
/// <exception cref="ArgumentOutOfRangeException">Thrown if count is less th
/// <exception cref="ArgumentException">Thrown if index and count are outsi
public virtual int Read(char[] buffer, int index, int count) {
    if (buffer == null) {
        throw new ArgumentNullException("buffer");
    }
    if (index < 0) {
        throw new ArgumentOutOfRangeException("index");
    }
    if (count < 0) {
        throw new ArgumentOutOfRangeException("count");
    }
    if (buffer.Length - index < count) {
        throw new ArgumentException();
    }
    ...
}
```

This is broken for a number of reasons. It's laboriously verbose, of course. All that ceremony! But we have to go way out of our way to document the exceptions when developers really ought not to ever catch them. Instead, they should find the bug during development and fix it. All this exception nonsense encourages very bad behavior.

If we use Midori-style contracts, on the other hand, this collapses to:

```
/// <summary>
/// ...
/// </summary>
public virtual int Read(char[] buffer, int index, int count)
    requires buffer != null
    requires index >= 0
    requires count >= 0
    requires buffer.Length - index >= count {
    ...
}
```

There are a few appealing things about this. First, it's more concise. More importantly, however, it self-describes the contract of the API in a way that documents itself and is easy to understand by callers. Rather than requiring programmers to express the error condition in English, the actual expressions are available for callers to read, and tools to understand and leverage. And it uses abandonment to communicate failure.

I should also mention we had plenty of contracts helpers to help developers write common preconditions. The above explicit range checking is very messy and easy to get wrong. Instead, we could have written:

```
public virtual int Read(char[] buffer, int index, int count)
    requires buffer != null
    requires Range.IsValid(index, count, buffer.Length) {
    ...
}
```

And, totally aside from the conversation at hand, coupled with two advanced features – arrays as slices and non-null types – we could have reduced the code to the following, while preserving the same guarantees:

```
public virtual int Read(char[] buffer) {
    ...
}
```

But I'm jumping way ahead …

Humble Beginnings

Although we landed on the obvious syntax that is very Eiffel- and Spec#-like – coming full circle – as I mentioned earlier, we really didn't want to change the language at the outset. So we actually began with a simple API approach:

```
public bool PublishPosition() {
    Contract.Requires(RemainingSize == 0);
    Contract.Ensures(UnpublishedSize == 0);
    ...
}
```

There are a number of problems with this approach, as the .NET Code Contracts effort discovered the hard way.

First, contracts written this way are part of the API's *implementation*, whereas we want them to be part of the *signature*. This might seem like a theoretical concern but it is far from being theoretical. We want the resulting program to contain built-in metadata so tools like IDEs and debuggers can display the contracts at callsites. And we want tools to be in a position to auto-generate documentation from the contracts. Burying them in the implementation doesn't work unless you somehow disassemble the method to extract them later on (which is a hack).

This also makes it tough to integrate with a backend compiler which we found was necessary for good performance.

Second, you might have noticed an issue with the call to `Contract.Ensures`. Since `Ensures` is meant to hold on all exit paths of the function, how would we implement this purely as an API? The answer is, you can't. One approach is rewriting the resulting MSIL, after the language compiler emitted it, but that's messy as all heck. At this point, you begin to wonder, why not simply acknowledge that this is a language expressivity and semantics issue, and add syntax?

Another area of perpetual struggle for us was whether contracts are conditional or not. In many classical systems, you'd check contracts in debug builds, but not the fully optimized ones. For a long time, we had the same three levels for contracts that we did assertions mentioned earlier:

- Weak, indicated by `Contract.Weak.*`, meaning debug-only.
- Normal, indicated simply by `Contract.*`, leaving it as an implementation decision when to check them.
- Strong, indicated by `Contract.Strong.*`, meaning always checked.

I'll admit, I initially found this to be an elegant solution. Unfortunately, over time we found that there was constant confusion about whether "normal" contracts were on in debug, release, or all of the above (and so people misused weak and strong accordingly). Anyway, when we began integrating this scheme into the language and backend compiler toolchain, we ran into substantial issues and had to backpedal a little bit.

First, if you simply translated `Contract.Weak.Requires` to `weak requires` and `Contract.Strong.Requires` to `strong requires`, in my opinion, you end up with a fairly clunky and specialized language syntax, with more policy than made me comfortable. It immediately calls out for parameterization and substitutability of the `weak`/`strong` policies.

Next, this approach introduces a sort of new mode of conditional compilation that, to me, felt awkward. In other words, if you want a debug-only check, you can already say something like:

```
#if DEBUG
    requires X
#endif
```

Finally – and this was the nail in the coffin for me – contracts were supposed to be part of an API's signature. What does it even *mean* to have a conditional contract? How is a tool supposed to reason about it? Generate different documentation for debug builds than release builds? Moreover, as soon as you do this, you lose a critical guarantee, which is that code doesn't run if its preconditions aren't met.

As a result, we nuked the entire conditional compilation scheme.

We ended up with a single kind of contract: one that was part of an API's signature and checked all the time. If a compiler could prove the contract was satisfied at compile-time – something we spent considerable energy on – it was free to elide the check altogether. But code was guaranteed it would never execute if its preconditions weren't satisfied. For cases where you wanted conditional checks, you always had the assertion system (described above).

I felt better about this bet when we deployed the new model and found that lots of people had been misusing the "weak" and "strong" notions above out of confusion. Forcing developers to make the decision led to healthier code.

Future Directions

A number of areas of development were at varying stages of maturity when our project wound down.

Invariants

We experimented **a lot** with invariants. Anytime we spoke to someone versed in design-by-contract, they were borderline appalled that we didn't have them from day one. To be honest, our design did include them from the outset. But we never quite got around to finishing the implementation and deploying it. This was partly just due to engineering bandwidth, but also because some difficult questions remained. And honestly the team was almost always satisfied with the combination of pre- and post-conditions plus assertions. I suspect that in the fullness of time we'd have added invariants for completeness, but to this day some questions remain for me. I'd need to see it in action for a while.

The approach we had designed was where an `invariant` becomes a member of its enclosing type; for example:

```
public class List<T> {
    private T[] array;
    private int count;
    private invariant index >= 0 && index < array.Length;
...
}
```

Notice that the `invariant` is marked `private`. An invariant's accessibility modifier controlled which members the invariant was required to hold for. For example, a `public invariant` only had to hold at the entry and exit of functions with `public` accessibility; this allowed for the common pattern of `private` functions temporarily violating invariants, so long as `public` entrypoints preserved them. Of course, as in the above example, a class was free to declare a `private invariant` too, which was required to hold at all function entries and exits.

I actually quite liked this design, and I think it would have worked. The primary concern we all had was the silent introduction of checks all over the place. To this day, that bit still makes me nervous. For example, in the `List<T>` example, you'd have the `index >= 0 && index < array.Length` check at the beginning and end of *every single function* of the type. Now, our compiler eventually got very good at recognizing and coalescing redundant contract checks; and there were ample cases where the presence of a contract actually made code quality *better*. However, in the extreme example given above, I'm sure there would have been a performance penalty. That would have put pressure on us changing the policy for when invariants are checked, which would have possibly complicated the overall contracts model.

I really wish we had more time to explore invariants more deeply. I don't think the team sorely missed not having them – certainly I didn't hear much complaining about their absence (probably because the team was so performance conscious) – but I do think invariants would have been a nice icing to put on the contracts cake.

Advanced Type Systems

I always liked to say that contracts begin where the type system leaves off. A type system allows you to encode attributes of variables using types. A type limits the expected range values that a variable might hold. A contract similarly checks the range of values that a variable holds. The difference? Types are proven at compile-time through rigorous and composable inductive rules that are moderately inexpensive to check local to a function, usually, but not always, aided by developer-authored annotations. Contracts are proven at compile-time *where possible* and at runtime otherwise, and as a result, permit far less rigorous specification using arbitrary logic encoded in the language itself.

Types are preferable, because they are *guaranteed* to be compile-time checked; and *guaranteed* to be fast to check. The assurances given to the developer are strong and the overall developer productivity of using them is better.

Limitations in a type system are inevitable, however; a type system needs to leave *some* wiggle room, otherwise it quickly grows unwieldly and unusable and, in the extreme, devolves into bi-value bits and bytes. On the other hand, I was always disappointed by two specific areas of wiggle room that required the use of contracts:

1. Nullability.
2. Numeric ranges.

Approximately 90% of our contracts fell into these two buckets. As a result, we seriously explored more sophisticated type systems to classify the nullability and ranges of variables using the type system instead of contracts.

To make it concrete, this was the difference between this code which uses contracts:

```
public virtual int Read(char[] buffer, int index, int count)
    requires buffer != null
    requires index >= 0
    requires count >= 0
    requires buffer.Length - index < count {
    ...
}
```

And this code which didn't need to, and yet carried all the same guarantees, checked statically at compile-time:

```
public virtual int Read(char[] buffer) {
    ...
}
```

Placing these properties in the type system significantly lessens the burden of checking for error conditions. Lets say that for any given 1 producer of state there are 10 consumers. Rather than having each of those 10 defend themselves against error conditions, we can push the responsibility back onto that 1 producer, and either require a single assertion that coerces the type, or even better, that the value is stored into the right type in the first place.

Non-Null Types

The first one's really tough: guaranteeing statically that variables do not take on the `null` value. This is what Tony Hoare has famously called his "billion dollar mistake". Fixing this for good is a righteous goal for any language and I'm happy to see newer language designers tackling this problem head-on.

Many areas of the language fight you every step of the way on this one. Generics, zero-initialization, constructors, and more. Retrofitting non-null into an existing language is tough!

The Type System

In a nutshell, non-nullability boiled down to some simple type system rules:

1. All unadorned types `T` were non-null by default.
2. Any type could be modified with a `?`, as in `T?`, to mark it nullable.
3. `null` is an illegal value for variables of non-null types.
4. `T` implicitly converts to `T?`. In a sense, `T` is a subtype of `T?` (although not entirely true).
5. Operators exist to convert a `T?` to a `T`, with runtime checks that abandoned on `null`.

Most of this is probably "obvious" in the sense that there aren't many choices. The name of the game is systematically ensuring all avenues of `null` are known to the type system. In particular, no `null` can ever "sneakily" become the value of a non-null `T` type; this meant addressing zero-initialization, perhaps the hardest problem of all.

The Syntax

Syntactically, we offered a few ways to accomplish #5, converting from `T?` to `T`. Of course, we discouraged this, and preferred you to stay in "non-null" space as long as possible. But sometimes it's simply not possible. Multi-step initialization happens from time to time – especially with collections data structures – and had to be supported.

Imagine for a moment we have a map:

```
Map<int, Customer> customers = ...;
```

This tells us three things by construction:

1. The `Map` itself is not null.
2. The `int` keys inside of it will not be `null`.
3. The `Customer` values inside of it will also not be null.

Let's now say that the indexer actually returns `null` to indicate the key was missing:

```
public TValue? this[TKey key] {
    get { ... }
}
```

Now we need some way of checking at callsites whether the lookup succeeded. We debated many syntaxes.

The easiest we landed on was a guarded check:

```
Customer? customer = customers[id];
if (customer != null) {
    // In here, `customer` is of non-null type `Customer`.
}
```

I'll admit, I was always on the fence about the "magical" type coercions. It annoyed me that it was hard to figure out what went wrong when it failed. For example, it didn't work if you compared `c` to a variable that held the `null` value, only the literal `null`. But the syntax was easy to remember and usually did the right thing.

These checks dynamically branch to a different piece of logic if the value is indeed `null`. Often you'd want to simply assert that the value is non-null and abandon otherwise. There was an explicit type-assertion operator to do that:

```
Customer? maybeCustomer = customers[id];
Customer customer = notnull(maybeCustomer);
```

The `notnull` operator turned any expression of type `T?` into an expression of type `T`.

Generics

Generics are hard, because there are multiple levels of nullability to consider. Consider:

```
class C {
    public T M<T>();
    public T? N<T>();
}

var a = C.M<object>();
var b = C.M<object?>();
var c = C.N<object>();
var d = C.N<object?>();
```

The basic question is, what are the types of `a`, `b`, `c`, and `d`?

I think we made this one harder initially than we needed to largely because C#'s existing nullable is a pretty odd duck and we got distracted trying to mimic it too much. The good news is we finally found our way, but it took a while.

To illustrate what I mean, let's go back to the example. There are two camps:

- **The .NET camp:** `a` is `object`; `b`, `c`, and `d` are `object?`.
- **The functional language camp:** `a` is `object`; `b` and `c` are `object?`; `d` is `object??`.

In other words, the .NET camp thinks you should collapse any sequence of 1 or more `?`s into a single `?`. The functional language camp – who understands the elegance of mathematical composition – eschews the magic and lets the world be as it is. We eventually realized that the .NET route is incredibly complex, and requires runtime support.

The functional language route does bend your mind slightly at first. For example, the map example from earlier:

```
Map<int, Customer?> customers = ...;
Customer?? customer = customers[id];
if (customer != null) {
    // Notice, `customer` is still `Customer?` in here, and could still be
}
```

In this model, you need to peel off one layer of `?` at a time. But honestly, when you stop to think about it, that makes sense. It's more transparent and reflects precisely what's going on under here. Best not to fight it.

There's also the question of implementation. The easiest implementation is to expand `T?` into some "wrapper type," like `Maybe<T>`, and then inject the appropriate wrap and unwrap operations. Indeed, that's a reasonable mental model for how the implementation works. There are two reasons this simple model doesn't work, however.

First, for reference type `T`, `T?` must not carry a wasteful extra bit; a pointer's runtime representation can carry `null` as a value already, and for a systems language, we'd like to exploit this fact and store `T?` as efficiently as `T`. This can be done fairly easily by specializing the generic instantiation. But this does mean that non-null can no longer simply be a front-end trick. It requires back-end compiler support.

(Note that this trick is not so easy to extend to `T??`!)

Second, Midori supported safe covariant arrays, thanks to our mutability annotations. If `T` and `T?` have a different physical representation, however, then converting `T[]` to `T?[]` is a non-transforming operation. This was a minor blemish, particularly since covariant arrays become far less useful once you plug the safety holes they already have.

Anyway, we eventually burned the ships on .NET `Nullable<T>` and went with the more composable multi-`?` design.

Zero-Initialization

Zero-initialization is a real pain in the butt. To tame it meant:

- All non-null fields of a class must be initialized at construction time.
- All arrays of non-null elements must be fully initialized at construction time.

But it gets worse. In .NET, value types are implicitly zero-initialized. The initial rule was therefore:

- All fields of a struct must be nullable.

But that stunk. It infected the whole system with nullable types immediately. My hypothesis was that nullability only truly works if nullable is the uncommon (say 20%) case. This would have destroyed that in an instant.

So we went down the path of eliminating automatic zero-initialization semantics. This was quite a large change. (C# 6 went down the path of allowing structs to provide their own zero-arguments constructors and eventually had to back it out due to the sheer impact this had on the ecosystem.) It could have been made to work but veered pretty far off course, and raised some other problems that we probably got too distracted with. If I could do it all over again, I'd just eliminate the value vs. reference type distinction altogether in C#. The rationale for that'll become clearer in an upcoming post on battling the garbage collector.

The Fate of Non-Null Types

We had a solid design, and several prototypes, but never deployed this one across the entire operating system. The reason why was tied up in our desired level of C# compatibility. To be fair, I waffled on this one quite a bit, and I suppose it was ultimately my decision. In the early days of Midori, we wanted "cognitive familiarity." In the later days of the project, we actually considered whether all of the features could be done as "add on" extensions to C#. It was that later mindset that prevented us from doing non-null types in earnest. My belief to this day is that additive annotations just won't work; Spec# tried this with `!` and the polarity always felt inverted. Non-null needs to be the default for this to have the impact we desired.

One of my biggest regrets is that we waited so long on non-null types. We only explored it in earnest once contracts were a known quantity, and we noticed the thousands of `requires x != null`s all over the place. It would have been complex and expensive, however this would have been a particularly killer combination if we nuked the value type distinction at the same time. Live and learn!

If we shipped our language as a standalone thing, different from C# proper, I'm convinced this would have made the cut.

Range Types

We had a design for adding range types to C#, but it always remained one step beyond my complexity limit.

The basic idea is that any numeric type can be given a lower and upper bound type parameter. For example, say you had an integer that could only hold the numbers 0 through 1,000,000, exclusively. It could be stated as `int<0..1000000>`. Of course, this points out that you probably should be using a `uint` instead and the compiler would warn you. In fact, the full set of numbers could be conceptually represented as ranges in this way:

```
typedef byte number<0..256>;
typedef sbyte number<-128..128>;
typedef short number<-32768..32768>;
typedef ushort number<0..65536>;
typedef int number<-2147483648..2147483648>;
typedef uint number<0..4294967295>;
// And so on ...
```

The really "cool" – but scary complicated – part is to then use dependent types to permit symbolic range parameters. For example, say I have an array and want to pass an index whose range is guaranteed to be in-bounds. Normally I'd write:

```
T Get(T[] array, int index)
        requires index >= 0 && index < array.Length {
    return array[index];
}
```

Or maybe I'd use a `uint` to eliminate the first half of the check:

```
T Get(T[] array, uint index)
        index < array.Length {
    return array[index];
}
```

Given range types, I can instead associate the upper bound of the number's range with the array length directly:

```
T Get(T[] array, number<0, array.Length> index) {
    return array[index];
}
```

Of course, there's no guarantee the compiler will eliminate the bounds check, if you somehow trip up its alias analysis. But we would hope that it does no worse a job with these types than with normal contracts checks. And admittedly this approach is a more direct encoding of information in the type system.

Anyway, I still chalk this one up to a cool idea, but one that's still in the realm of "nice to have but not critical."

The "not critical" aspect is especially true thanks to slices being first class in the type system. I'd say 66% or more of the situations where range checks were used would have been better written using slices. I think mainly people were still getting used to having them and so they'd write the standard C# thing rather than just using a slice. I'll cover slices in an upcoming post, but they removed the need for writing range checks altogether in most code.

Recoverable Errors: Type-Directed Exceptions

Abandonment isn't the only story, of course. There are still plenty of legitimate situations where an error the programmer can reasonably recover from occurs. Examples include:

- File I/O.
- Network I/O.
- Parsing data (e.g., a compiler parser).
- Validating user data (e.g., a web form submission).

In each of these cases, you usually don't want to trigger abandonment upon encountering a problem. Instead, the program expects it to occur from time to time, and needs to deal with it by doing something reasonable. Often by communicating it to someone: the user typing into a webpage, the administrator of the system, the developer using a tool, etc. Of course, abandonment is one method call away if that's the most appropriate action to take, but it's often too drastic for these situations. And, especially for IO, it runs the risk of making the system very brittle. Imagine if the program you're using decided to wink out of existence every time your network connection dropped a packet!

Enter Exceptions

We used exceptions for recoverable errors. Not the unchecked kind, and not quite the Java checked kind, either.

First thing's first: although Midori had exceptions, a method that wasn't annotated as `throws` could never throw one. Never ever ever. There were no sneaky `RuntimeException`s like in Java, for instance. We didn't need them anyway, because the same situations Java used runtime exceptions for were instead using abandonment in Midori.

This led to a magical property of the result system. 90-something% of the functions in our system could not throw exceptions! By default, in fact, they could not. This was a stark contrast to systems like C++ where you must go out of your way to abstain from exceptions and state that fact using `noexcept`. APIs could still fail due to abandonment, of course, but only when callers fail meet the stated contract, similar to passing an argument of the wrong type.

Our choice of exceptions was controversial at the outset. We had a mixture of imperative, procedural, object oriented, and functional language perspective on the team. The C programmers wanted to use error codes and were worried we would recreate the Java, or worse, C# design. The functional perspective would be to use dataflow for all errors, but exceptions were very control-flow-oriented. In the end, I think what we chose was a nice compromise between all of the available recoverable error models available to us. As we'll see later, we did offer a mechanism for treating errors as first class values for that rare case where a more dataflow style of programming was what the developer wanted.

Most importantly, however, we wrote a lot of code in this model, and it worked very well for us. Even the functional language guys came around eventually. As did the C programmers, thanks to some cues we took from return codes.

Language and Type System

At some point, I made a controversial observation and decision. Just as you wouldn't change a function's return type with the expectation of zero compatibility impact, you should not be changing a function's exception type with such an expectation. *In other words, an exception, as with error codes, is just a different kind of return value!*

This has been one of the parroted arguments against checked exceptions. My answer may sound trite, but it's simple: too bad. You're in a statically typed programming language, and the dynamic nature of exceptions is precisely the reason they suck. We sought to address these very problems, so therefore we embraced it, embellished strong typing, and never looked back. This alone helped to bridge the gap between error codes and exceptions.

Exceptions thrown by a function became part of its signature, just as parameters and return values are. Remember, due to the rare nature of exceptions compared to abandonment, this wasn't as painful as you might think. And a lot of intuitive properties flowed naturally from this decision.

The first thing is the Liskov substitution principle. In order to avoid the mess that C++ found itself in, all "checking" has to happen statically, at compile time. As a result, all of those performance problems mentioned in the WG21 paper were not problems for us. This type system must be bulletproof, however, with no backdoors to defeat it. Because we needed to address those performance challenges by depending on `throws` annotations in our optimizing compiler, type safety hinged on this property.

We tried many many different syntaxes. Before we committed to changing the language, we did everything with C# attributes and static analysis. The user experience wasn't very good and it's hard to do a real type system that way. Furthermore, it felt too bolted on. We experimented with approaches from the Redhawk project – what eventually became .NET Native and CoreRT – however, that approach also didn't leverage the language and relied instead on static analysis, though it shares many similar principles with our final solution.

The basic gist of the final syntax was to simply state a method `throws` as a single bit:

```
void Foo() throws {
    ...
}
```

(For many years, we actually put the `throws` at the beginning of the method, but that read wrong.)

At this point, the issue of substitutability is quite simple. A `throws` function cannot take the place of a non- `throws` function (illegal strengthening). A non-`throws` function, on the other hand, can take the place of a `throws` function (legal weakening). This obviously impacts virtual overrides, interface implementation, and lambdas.

Of course, we did the expected co- and contravariance substitution bells and whistles. For example, if `Foo` were virtual and you overrode it but didn't throw exceptions, you didn't need to state the `throws` contract. Anybody invoking such a function virtually, of course, couldn't leverage this but direct calls could.

For example, this is legal:

```
class Base {
    public virtual void Foo() throws {...}
}

class Derived : Base {
    // My particular implementation doesn't need to throw:
    public override void Foo() {...}
}
```

and callers of `Derived` could leverage the lack of `throws`; whereas this is wholly illegal:

```
class Base {
    public virtual void Foo () {...}
}

class Derived : Base {
    public override void Foo() throws {...}
}
```

Encouraging a single failure mode was quite liberating. A vast amount of the complexity that comes with Java's checked exceptions evaporated immediately. If you look at most APIs that fail, they have a single failure mode anyway (once all bug failure modes are done with abandonment): IO failed, parsing failed, etc. And many recovery actions a developer tends to write don't actually depend on the specifics of *what exactly* failed when, say, doing an IO. (Some do, and for those, the keeper pattern is often the better answer; more on this topic shortly.) Most of the information in modern exceptions are *not* actually there for programmatic use; instead, they are for diagnostics.

We stuck with just this "single failure mode" for 2-3 years. Eventually I made the controversial decision to support multiple failure modes. It wasn't common, but the request popped up reasonably often from teammates, and the scenarios seemed legitimate and useful. It did come at the expense of type system complexity, but only in all the usual subtyping ways. And more sophisticated scenarios – like aborts (more on that later) – required that we do this.

The syntax looked like this:

```
int Foo() throws FooException, BarException {
    ...
}
```

In a sense, then, the single `throws` was a shortcut for `throws Exception`.

It was very easy to "forget" the extra detail if you didn't care. For example, perhaps you wanted to bind a lambda to the above `Foo` API, but didn't want callers to care about `FooException` or `BarException`. That lambda must be marked `throws`, of course, but no more detail was necessary. This turned out to be a very common pattern: An internal system would use typed exceptions like this for internal control flow and error handling, but translate all of them into just plain `throws` at the public boundary of the API, where the extra detail wasn't required.

All of this extra typing added great power to recoverable errors. But if contracts outnumbered exceptions by 10:1, then simple `throws` exceptional methods outnumbered multi-failure-mode ones by another 10:1.

At this point, you may be wondering, what differentiated this from Java's checked exceptions?

1. The fact that the lion's share of errors were expressed using abandonment meant most APIs didn't throw.
2. The fact that we encouraged a single mode of failure simplified the entire system greatly. Moreover, we made it easy to go from the world of multiple modes, to just a single and back again.

The rich type system support around weakening and strengthening also helped, as did something else we did to that helped bridge the gap between return codes and exceptions, improved code maintainability, and more …

Easily Auditable Callsites

At this point in the story, we still haven't achieved the full explicit syntax of error codes. The declarations of functions say whether they can fail (good), but callers of those functions still inherit silent control flow (bad).

This brings about something I always loved about our exceptions model. A callsite needs to say `try`:

```
int value = try Foo();
```

This invokes the function `Foo`, propagates its error if one occurs, and assigns the return value to `value` otherwise.

This has a wonderful property: all control flow remains explicit in the program. You can think of `try` as a kind of conditional `return` (or conditional `throw` if you prefer). I *freaking loved* how much easier this made code reviewing error logic! For example, imagine a long function with a few `try`s inside of it; having the explicit annotation made the points of failure, and therefore control flow, as easy to pick out as `return` statements:

```
void doSomething() throws {
    blah();
    var x = blah_blah(blah());
    var y = try blah(); // <-- ah, hah! something that can fail!
    blahdiblahdiblahdiblahdi();
    blahblahblahblah(try blahblah()); // <-- another one!
    and_so_on(...);
}
```

If you have syntax highlighting in your editor, so the `try`s are bold and blue, it's even better.

This delivered many of the strong benefits of return codes, but without all the baggage.

(Both Rust and Swift now support a similar syntax. I have to admit I'm sad we didn't ship this to the general public years ago. Their implementations are very different, however consider this a huge vote of confidence in their syntax.)

Of course, if you are `try`ing a function that throws like this, there are two possibilities:

- The exception escapes the calling function.
- There is a surrounding `try`/`catch` block that handles the error.

In the first case, you are required to declare that your function `throws` too. It is up to you whether to propagate strong typing information should the callee declare it, or simply leverage the single `throws` bit, of course.

In the second case, we of course understood all the typing information. As a result, if you tried to catch something that wasn't declared as being thrown, we could give you an error about dead code. This was yet another controversial departure from classical exceptions systems. It always bugged me that `catch (FooException)` is essentially hiding a dynamic type test. Would you silently permit someone to call an API that returns just `object` and automatically assign that returned value to a typed variable? Hell no! So we didn't let you do that with exceptions either.

Here too CLU influenced us. Liskov talks about this in A History of CLU:

> *CLU's mechanism is unusual in its treatment of unhandled exceptions. Most mechanisms pass these through: if the caller does not handle an exception raised by a called procedure, the exception is propagated to its caller, and so on. We rejected this approach because it did not fit our ideas about modular program construction. We wanted to be able to call a procedure knowing just its specification, not its implementation. However, if exceptions are propagated automatically, a procedure may raise an exception not described in its specification.*

Although we discouraged wide `try` blocks, this was conceptually a shortcut for propagating an error code. To see what I mean, consider what you'd do in a system with error codes. In Go, you might say the following:

```
if err := doSomething(); err != nil {
    return err
}
```

In our system, you say:

```
try doSomething();
```

But we used exceptions, you might say! It's completely different! Sure, the runtime systems differ. But from a language "semantics" perspective, they are isomorphic. We encouraged people to think in terms of error codes and not the exceptions they knew and loved. This might seem funny: Why not just use return codes, you might wonder? In an upcoming section, I will describe the true isomorphism of the situation to try to convince you of our choice.

Syntactic Sugar

We also offered some syntactic sugar for dealing with errors. The `try`/`catch` block scoping construct is a bit verbose, especially if you're following our intended best practices of handling errors as locally as possible. It also still retains a bit of that unfortunate `goto` feel for some, especially if you are thinking in terms of return codes. That gave way to a type we called `Result<T>`, which was simply *either* a `T` value *or* an `Exception`.

This essentially bridged from the world of control-flow to the world of dataflow, for scenarios in which the latter was more natural. Both certainly had their place, although most developers preferred the familiar control flow syntax.

To illustrate common usage, imagine you want to log all errors that occur, before repropagating the exception. Though this is a common pattern, using `try`/`catch` blocks feels a little too control flow heavy for my taste:

```
int v;
try {
    v = try Foo();
    // Maybe some more stuff...
}
catch (Exception e) {
    Log(e);
    rethrow;
}
// Use the value `v`...
```

The "maybe some more stuff" bit entices you to squeeze more than you should into the `try` block. Compare this to using `Result<T>`, leading to a more return-code feel and more convenient local handling:

```
Result<int> value = try Foo() else catch;
if (value.IsFailure) {
    Log(value.Exception);
    throw value.Exception;
}
// Use the value `value.Value`...
```

The `try ... else` construct also permitted you to substitute your own value instead, or even trigger abandonment, in response to failure:

```
int value1 = try Foo() else 42;
int value2 = try Foo() else Release.Fail();
```

We also supported NaN-style propagation of dataflow errors by lifting access to `T`s members out of the `Result<T>`. For example, let's say I have two `Result<int>`s and want to add them together. I can do so:

```
Result<int> x = ...;
Result<int> y = ...;
Result<int> z = x + y;
```

Notice that third line, where we added the two `Result<int>`s together, yielding a – that's right – third `Result<T>`. This is the NaN-style dataflow propagation, similar to C#'s new `.?` feature.

This approach blends what I found to be an elegant mixture of exceptions, return codes, and dataflow error propagation.

Implementation

The model I just described doesn't have to be implemented with exceptions. It's abstract enough to be reasonably implemented using either exceptions or return codes. This isn't theoretical. We actually tried it. And this is what led us to choose exceptions instead of return codes for performance reasons.

To illustrate how the return code implementation might work, imagine some simple transformations:

```
int foo() throws {
    if (...p...) {
        throw new Exception();
    }
    return 42;
}
```

becomes:

```
Result<int> foo() {
    if (...p...) {
        return new Result<int>(new Exception());
    }
    return new Result<int>(42);
}
```

And code like this:

```
int x = try foo();
```

becomes something more like this:

```
int x;
Result<int> tmp = foo();
if (tmp.Failed) {
    throw tmp.Exception;
}
x = tmp.Value;
```

An optimizing compiler can represent this more efficiently, eliminating excessive copying. Especially with inlining.

If you try to model `try`/`catch`/`finally` this same way, probably using `goto`, you'll quickly see why compilers have a hard time optimizing in the presence of unchecked exceptions. All those hidden control flow edges!

Either way, this exercise very vividly demonstrates the drawbacks of return codes. All that goop – which is meant to be rarely needed (assuming, of course, that failure is rare) – is on hot paths, mucking with your program's golden path performance. This violates one of our most important principles.

I described the results of our dual mode experiment in my last post. In summary, the exceptions approach was 7% smaller and 4% faster as a geomean across our key benchmarks, thanks to a few things:

- No calling convention impact.
- No peanut butter associated with wrapping return values and caller branching.
- All throwing functions were known in the type system, enabling more flexible code motion.

- All throwing functions were known in the type system, giving us novel EH optimizations, like turning try/finally blocks into straightline code when the try could not throw.

There were other aspects of exceptions that helped with performance. I already mentioned that we didn't grovel the callstack gathering up metadata as most exceptions systems do. We left diagnostics to our diagnostics subsystem. Another common pattern that helped, however, was to cache exceptions as frozen objects, so that each `throw` didn't require an allocation:

```
const Exception retryLayout = new Exception();
...
throw retryLayout;
```

For systems with high rates of throwing and catching – as in our parser, FRP UI framework, and other areas – this was important to good performance. And this demonstrates why we couldn't simply take "exceptions are slow" as a given.

Patterns

A number of useful patterns came up that we embellished in our language and libraries.

Concurrency

Back in 2007, I wrote this note about concurrency and exceptions. I wrote it mainly from the perspective of parallel, shared memory computations, however similar challenges exist in all concurrent orchestration patterns. The basic issue is that the way exceptions are implemented assumes single, sequential stacks, with single failure modes. In a concurrent system, you have many stacks and many failure modes, where 0, 1, or many may happen "at once."

A simple improvement that Midori made was simply ensuring all `Exception`-related infrastructure handled cases with multiple inner errors. At least then a programmer wasn't forced to decide to toss away 1/N'th of the failure information, as most exceptions systems encourage today. More than that, however, our scheduling and stack crawling infrastructure fundamentally knew about cactus-style stacks, thanks to our asynchronous model, and what to do with them.

At first, we didn't support exceptions across asynchronous boundaries. Eventually, however, we extended the ability to declare `throws`, along with optional typed exceptions clauses, across asynchronous process boundaries. This brought a rich, typed programming model to the asynchronous actors programming model and felt like a natural extension. This borrowed a page from CLU's successor, Argus.

Our diagnostics infrastructure embellished this to give developers debugging experiences with full-blown cross-process causality in their stack views. Not only are stacks cactuses in a highly concurrent system, but they are often smeared across process message passing boundaries. Being able to debug the system this way was a big time-saver.

Aborts

Sometimes a subsystem needs to "get the hell out of Dodge." Abandonment is an option, but only in response to bugs. And of course nobody in the process can stop it in its tracks. What if we want to back out the callstack to some point, know that no-one on the stack is going to stop us, but then recover and keep going within the same process?

Exceptions were close to what we wanted here. But unfortunately, code on the stack can catch an in-flight exception, thereby effectively suppressing the abort. We wanted something unsuppressable.

Enter aborts. We invented aborts mainly in support of our UI framework which used Functional Reactive Programming (FRP), although the pattern came up in a few spots. As an FRP recalculation was happening, it's possible that events would enter the system, or new discoveries got made, that invalidate the current recalculation. If that happened – typically deep within some calculation whose stack was an interleaving of user and system code – the FRP engine needed to quickly get back to top of its stack where it could safely begin a recalculation. Thanks to all of that user code on the stack being functionally pure, aborting it mid-stream was easy. No errant side-effects would be left behind. And all engine code that was traversed was audited and hardened thoroughly, thanks to typed exceptions, to ensure invariants were maintained.

The abort design borrows a page from the capability playbook. First, we introduce a base type called `AbortException`. It may be used directly or subclassed. One of these is special: nobody can catch-and-ignore it. The exception is reraised automatically at the end of any catch block that attempts to catch it. We say that such exceptions are *undeniable*.

But someone's got to catch an abort. The whole idea is to exit a context, not tear down the entire process a la abandonment. And here's where capabilities enter the picture. Here's the basic shape of `AbortException`:

```
public immutable class AbortException : Exception {
    public AbortException(immutable object token);
    public void Reset(immutable object token);
    // Other uninteresting members omitted...
}
```

Notice that, at the time of construction, an immutable `token` is provided; in order to suppress the throw, `Reset` is called, and a matching `token` must be provided. If the `token` doesn't match, abandonment occurs. The idea is that the throwing and intended catching parties of an abort are usually the same, or at least in cahoots with one another, such that sharing the `token` securely with one another is easy to do. This is a great example of objects as unforgeable capabilities in action.

And yes, an arbitrary piece of code on the stack can trigger an abandonment, but such code could already do that by simply dereferencing `null`. This technique prohibits executing in the aborting context when it might not have been ready for it.

Other frameworks have similar patterns. The .NET Framework has `ThreadAbortException` which is also undeniable unless you invoke `Thread.ResetAbort`; sadly, because it isn't capability-based, a clumsy combination of security annotations and hosting APIs are required to stop unintended swallowing of aborts. More often, this goes unchecked.

Thanks to exceptions being immutable, and the `token` above being immutable, a common pattern was to cache these guys in static variables and use singletons. For example:

```
class MyComponent {
    const object abortToken = new object();
    const AbortException abortException = new AbortException(abortToken);

    void Abort() throws AbortException {
        throw abortException;
    }

    void TopOfTheStack() {
        while (true) {
            // Do something that calls deep into some callstacks;
            // deep down it might Abort, which we catch and reset:
            let result = try ... else catch<AbortException>;
            if (result.IsFailed) {
                result.Exception.Reset(abortToken);
            }
        }
    }
}
```

This pattern made aborts very efficient. An average FRP recalculation aborted multiple times. Remember, FRP was the backbone of all UI in the system, so the slowness often attributed to exceptions was clearly not acceptable. Even allocating an exception object would have been unfortunate, due to the ensuing GC pressure.

Opt-in "Try" APIs

I mentioned a number of operations that abandoned upon failure. That included allocating memory, performing arithmetic operations that overflowed or divided-by-zero, etc. In a few of these instances, a fraction of the uses are appropriate for dynamic error propagation and recovery, rather than abandonment. Even if abandonment is better in the common case.

This turned out to be a pattern. Not terribly common, but it came up. As a result, we had a whole set of arithmetic APIs that used a dataflow-style of propagation should overflow, NaN, or any number of things happen.

I also already mentioned a concrete instance of this earlier, which is the ability to `try new` an allocation, when OOM yields a recoverable error rather than abandonment. This was super uncommon, but could crop up if you wanted to, say, allocate a large buffer for some multimedia operation.

Keepers

The last pattern I'll cover is called *the keeper pattern*.

In a lot of ways, the way recoverable exceptions are handled is "inside out." A bunch of code is called, passing arguments down the callstack, until finally some code is reached that deems that state unacceptable. In the exceptions model, control flow is then propagated back up the callstack, unwinding it, until some code is found that handles the error. At that point if the operation is to be retried, the sequence of calls must be reissued, etc.

An alternative pattern is to use a keeper. The keeper is an object that understands how to recover from errors "in situ," so that the callstack needn't be unwound. Instead, the code that would have otherwise thrown an exception consults the keeper, who instructs the code how to proceed. A nice aspect of keepers is that often, when done as a configured capability, surrounding code doesn't even need to know they exist – unlike exceptions which, in our system, had to be declared as part of the type system. Another aspect of keepers is that they are simple and cheap.

Keepers in Midori could be used for prompt operations, but more often spanned asynchronous boundaries.

The canonical example of a keeper is one guarding filesystem operations. Accessing files and directories on a file system typically has failure modes such as:

- Invalid path specification.
- File not found.
- Directory not found.
- File in use.
- Insufficient privileges.
- Media full.
- Media write-protected.

One option is to annotate each filesystem API with a `throws` clause for each. Or, like Java, to create an `IOException` hierarchy with each of these as subclasses. An alternative is to use a keeper. This ensures the overall application doesn't need to know or care about IO errors, permitting recovery logic to be centralized. Such a keeper interface might look like this:

```
async interface IFileSystemKeeper {
    async string InvalidPathSpecification(string path) throws;
    async string FileNotFound(string path) throws;
    async string DirectoryNotFound(string path) throws;
    async string FileInUse(string path) throws;
    async Credentials InsufficientPrivileges(Credentials creds, string path
    async string MediaFull(string path) throws;
    async string MediaWriteProtected(string path) throws;
}
```

The idea is that, in each case, the relevant inputs are provided to the keeper when failure occurs. The keeper is then permitted to perform an operation, possibly asynchronous, to recover. In many cases, the keeper can optionally return updated arguments for the operation. For example, `InsufficientPrivileges` could return alternative `Credentials` to use. (Maybe the program prompted the user and she switched to an account with write access.) In each case shown, the keeper could throw an exception if it didn't want to handle the error, although this part of the pattern was optional.

Finally, I should note that Windows's Structured Exception Handling (SEH) system supports "continuable" exceptions which are conceptually attempting to achieve this same thing. They let some code decide how to restart the faulting computation. Unfortunately, they're done using ambient handlers on the callstack, rather than first class objects in the language, and so are far less elegant – and significantly more error prone – than the keepers pattern.

Future Directions: Effect Typing

Most people asked us about whether having `async` and `throws` as type system attributes bifurcated the entire universe of libraries. The answer was "No, not really." But it sure was painful in highly polymorphic library code.

The most jarring example was combinators like map, filter, sort, etc. In those cases, you often have arbitrary functions and want the `async` and `throws` attributes of those functions to "flow through" transparently.

The design we had to solve this was to let you parameterize over effects. For instance, here is a universal mapping function, `Map`, that propagates the `async` or `throws` effect of its `func` parameter:

```
U[] Map<T, U, effect E>(T[] ts, Func<T, U, E> func) E {
    U[] us = new U[ts.Length];
    for (int i = 0; i < ts.Length; i++) {
        us[i] = effect(E) func(ts[i]);
    }
    return us;
}
```

Notice here that we've got an ordinary generic type, `E`, except that its declaration is prefixed by the keyword `effect`. We then use `E` symbolically in place of the effects list of the `Map` signature, in addition to using it in the "propagate" position via `effect(E)` when invoking `func`. It's a pretty trivial exercise in substitution, replacing `E` with `throws` and `effect(E)` with `try`, to see the logical transformation.

A legal invocation might be:

```
int[] xs = ...;
string[] ys = try Map<int, string, throws>(xs, x => ...);
```

Notice here that the `throws` flows through, so that we can pass a callback that throws exceptions.

As a total aside, we discussed taking this further, and allowing programmers to declare arbitrary effects. I've hypothesized about such a type system previously. We were concerned, however, that this sort of higher order programming might be gratuitously clever and hard to understand, no matter how powerful. The simple model above probably would've been a sweet spot and I think we'd have done it given a few more months.

Retrospective and Conclusions

We've reached the end of this particular journey. As I said at the outset, a relatively predictable and tame outcome. But I hope all that background helped to take you through the evolution as we sorted through the landscape of errors.

In summary, the final model featured:

- An architecture that assumed fine-grained isolation and recoverability from failure.
- Distinguishing between bugs and recoverable errors.
- Using contracts, assertions, and, in general, abandonment for all bugs.
- Using a slimmed down checked exceptions model for recoverable errors, with a rich type system and language syntax.
- Adopting some limited aspects of return codes – like local checking – that improved reliability.

And, though this was a multi-year journey, there were areas of improvement we were actively working on right up until our project's untimely demise. I classified them differently because we didn't have enough experience using them to claim success. I would have hoped we'd have tidied up most of them and shipped them if we ever got that far. In particular, I'd have liked to put this one into the final model category:

- Leveraging non-null types by default to eliminate a large class of nullability annotations.

Abandonment, and the degree to which we used it, was in my opinion our biggest and most successful bet with the Error Model. We found bugs early and often, where they are easiest to diagnose and fix. Abandonment-based errors outnumbered recoverable errors by a ratio approaching 10:1, making checked exceptions rare and tolerable to the developer.

Although we never had a chance to ship this, we have since brought some of these lessons learned to other settings.

During the Microsoft Edge browser rewrite from Internet Explorer, for example, we adopted abandonment in a few areas. The key one, applied by a Midori engineer, was OOM. The old code would attempt to limp along as I described earlier and almost always did the wrong thing. My understanding is that abandonment has found numerous lurking bugs, as was our experience regularly in Midori when porting existing codebases. The great thing too is that abandonment is more of an architectural discipline that can be adopted in existing code-bases ranging in programming languages.

The architectural foundation of fine-grained isolation is critical, however many systems have an informal notion of this architecture. A reason why OOM abandonment works well in a browser is that most browsers devote separate processes to individual tabs already. Browsers mimic operating systems in many ways and here too we see this playing out.

More recently, we've been exploring proposals to bring some of this discipline – including contracts – to C++. There are also concrete proposals to bring some of these features to C# too. We are actively iterating on a proposal that would bring some non-null checking to C#. I have to admit, I wish all of those proposals the best, however nothing will be as bulletproof as an entire stack written in the same error discipline. And remember, the entire isolation and concurrency model is essential for abandonment at scale.

I am hopeful that continued sharing of knowledge will lead to even more wide-scale adoption some of these ideas.

And, of course, I've mentioned that Go, Rust, and Swift have given the world some very good systems-appropriate error models in the meantime. I might have some minor nits here and there, but the reality is that they're worlds beyond what we had in the industry at the time we began the Midori journey. It's a good time to be a systems programmer!

Next time I'll talk more about the language. Specifically, we'll see how Midori was able to tame the garbage collector using a magical elixir of architecture, language support, and libraries. I hope to see you again soon!

Tweet