Error Handling Rationale and Proposal

Contents

This paper surveys the error-handling world, analyzes various ideas which have been proposed or are in practice in other languages, and ultimately proposes an error-handling scheme for Swift together with import rules for our APIs.

Fundamentals

I need to establish some terminology first.

Kinds of propagation

I've heard people talk about **explicit vs. implicit propagation**. I'm not going to use those terms, because they're not helpful: there are at least three different things about error-handling that can be more or less explicit, and some of the other dimensions are equally important.

The most important dimensions of variation are:

- Whether the language allows functions to be designated as producing errors or not; such a language has **typed propagation**.
- Whether, in a language with typed propagation, the default rule is that a function can produce an error or that it can't; this is the language's **default propagation rule**.
- Whether, in a language with typed propagation, the language enforces this statically, so that a function which cannot produce an error cannot call a function which can without handling it; such a language has **statically-enforced typed propagation**. (A language could instead enforce this dynamically by automatically inserting code to assert if an error propagates out. C++ does this.)
- Whether the language requires all potential error sites to be identifiable as potential error sites; such a language has **marked propagation**.
- Whether propagation is done explicitly with the normal data-flow and control-flow tools of the language; such a language has **manual propagation**. In contrast, a language where control implicitly jumps from the original error site to the proper handler has **automatic propagation**.

Kinds of error

What is an error? There may be many different possible error conditions in a program, but they can be categorized into several kinds based on how programmers should be expected to react to them. Since the programmer is expected to react differently, and since the language is the tool of the programmer's reaction, it makes sense for each group to be treated differently in the language.

To be clear, in many cases the kind of error reflects a conscious decision by the author of the error-producing code, and different choices might be useful in different contexts. The example I'm going to use of a "simple domain error" could easily be instead treated as a "recoverable error" (if the author expected automatic propagation to be more useful than immediate recovery) or even a "logic failure" (if the author wished to prevent speculative use, e.g. if checking the precondition was very expensive).

In order of increasing severity and complexity:

Simple domain errors

A simple domain error is something like calling `String.toInt()` on a string that isn't an integer. The operation has an obvious precondition about its arguments, but it's useful to be able to pass other values to test whether they're okay. The client will often handle the error immediately.

Conditions like this are best modeled with an optional return value. They don't benefit from a more complex error-handling model, and using one would make common code unnecessarily awkward. For example, speculatively trying to parse a `String` as an integer in Java requires catching an exception, which is far more syntactically heavyweight (and inefficient without optimization).

Because Swift already has good support for optionals, these conditions do not need to be a focus of this proposal.

Recoverable errors

Recoverable errors include file-not-found, network timeouts, and similar conditions. The operation has a variety of possible error conditions. The client should be encouraged to recognize the possibility of the error condition and consider the right way to handle it. Often, this will be by aborting the current operation, cleaning up after itself if needed, and propagating the error to a point where it can more sensibly be handled, e.g. by reporting it to the user.

These are the error conditions that most of our APIs use `NSError` and `CFError` for today. Most libraries have some similar notion. This is the focus of this proposal.

Universal errors

The difference between universal errors and ordinary recoverable errors is less the kind of error condition and more the potential sources of the error in the language. An error is universal if it could arise from such a wealth of different circumstances that it becomes nearly impracticable for the programmer to directly deal with all the sources of the error.

Some conditions, if they are to be brought into the scope of error-handling, can only conceivably be dealt with as universal errors. These include:

- Asynchronous conditions, like the process receiving a `SIGINT`, or the current thread being cancelled by another thread. These conditions could, in principle, be delivered at an arbitrary instruction boundary, and handling them appropriately requires extraordinary care from the programmer, the compiler, and the runtime.
- Ubiquitous errors, like running out of memory or overflowing the stack, that essentially any operation can be assumed to potentially do.

But other kinds of error condition can essentially become universal errors with the introduction of abstraction. Reading the size of a collection, or reading a property of an object, is not an operation that a programmer would normally expect to produce an error. However, if the collection is actually backed by a database, the database query might fail. If the user must write code as if any opaque abstraction might produce an error, they are stuck in the world of universal errors.

Universal errors mandate certain language approaches. Typed propagation of universal errors is impossible, other than special cases which can guarantee to not produce errors. Marked propagation would provoke a user revolt. Propagation must be automatic, and the implementation must be "zero-cost", or as near to it as possible, because checking for an error after every single operation would be prohibitive.

For these reasons, in our APIs, universal error conditions are usually implemented using Objective-C exceptions, although not all uses of Objective-C exceptions fall in this category.

This combination of requirements means that all operations must be implicitly "unwindable" starting from almost any call site it makes. For the stability of the system, this unwinding process must restore any invariants that might have been temporarily violated; but the compiler cannot assist the programmer in this. The programmer must consciously recognize that an error is possible while an invariant is broken, and they must do this proactively --- that, or track it down when they inevitably forget. This requires thinking quite rigorously about one's code, both to foresee all the error sites and to recognize that an important invariant is in flux.

How much of a problem this poses depends quite a lot on the code being written. There are some styles of programming that make it pretty innocuous. For example, a highly functional program which conscientiously kept mutation and side-effects to its outermost loops would naturally have very few points where any invariants were in flux; propagating an error out of an arbitrary place within an operation would simply abandon all the work done up to that point. However, this happy state falls apart quite quickly the more that mutation and other side-effects come into play. Complex mutations cannot be trivially reversed. Packets cannot be unsent. And it would be quite amazing for us to assert that code shouldn't be written that way, understanding nothing else about it. As long as programmers do face these issues, the language has some responsibility to help them.

Therefore, in my judgment, promoting the use of universal errors is highly problematic. They undermine the easy comprehension of code, and they undermine the language's ability to help the programmer reason about errors. This design will instead focus on explicitly trackable errors of the sort that `NSError` is used for today on Apple platforms.

However, there are some important reasons not to rule out universal errors completely:

- They remain the only viable means of bringing certain error conditions into the error-handling model, as discussed above. Of these, most run into various objections; the most important remaining use case is "escaping", where an unexpected implementation of an API that was not designed to throw finds itself needing to.
- Objective-C and C++ exceptions are a legitimate interoperation problem on any conceivable platform Swift targets. Swift must have some sort of long-term answer for them.

These reasons don't override the problems with universal errors. It is inherently dangerous to implicitly volunteer functions for unwinding from an arbitrary point. We don't want to promote this model. However, it is certainly possible to write code that handles universal errors correctly; and pragmatically, unwinding through most code will generally just work. Swift could support a secondary, untyped propagation mechanism using "zero-cost" exceptions. Code can be written carefully to minimize the extent of implicit unwinding, e.g. by catching universal errors immediately after calling an "escaping" API and rethrowing them with normal typed propagation.

However, this work is outside of the scope of Swift 2.0. We can comfortably make this decision because doing so doesn't lock us out of implementing it in the future:

- We do not currently support propagating exceptions through Swift functions, so changing `catch` to catch them as well would not be a major compatibility break.
- With some admitted awkwardness, external exceptions can be reflected into an `Error` - like model automatically by the catch mechanism.
- In the meanwhile, developers who must handle an Objective-C exception can always do so by writing a stub in Objective-C to explicitly "bridge" the exception into an `NSError` out parameter. This isn't ideal, but it's acceptable.

Logic failures

The final category is logic failures, including out of bounds array accesses, forced unwrap of `nil` optionals, and other kinds of assertions. The programmer has made a mistake, and the failure should be handled by fixing the code, not by attempting to recover dynamically.

High-reliability systems may need some way to limp on even after an assertion failure. Tearing down the process can be viewed as a vector for a denial-of-service attack. However, an assertion failure might indicate that the process has been corrupted and is under attack, and limping on anyway may open the system up for other, more serious forms of security breach.

The correct handling of these error conditions is an open question and is not a focus of this proposal. Should we decide to make them recoverable, they will likely follow the same implementation mechanism as universal errors, if not necessarily the same language rules.

Analysis

Let's take a deeper look into the different dimensions of error-handling I laid out above.

Propagation methods

At a language level, there are two basic ways an error can be propagated from an error site to something handling it.

The first is that it can be done with the normal evaluation, data flow, and control flow processes of the language; let's call this **manual propagation**. Here's a good example of manual propagation using special return values in an imperative language, C:

```
struct object *read_object(void) {
```

```
  char buffer[1024];
  ssize_t numRead = read(0, buffer, sizeof(buffer));
  if (numRead < 0) return NULL;
  ...
}
```

Here's an example of manual propagation of an error value through out-parameters in another imperative language, Objective-C:

```
- (BOOL) readKeys: (NSArray<NSString*>**) strings error: (NSError**) err {
  while (1) {
    NSString *key;
    if ([self readKey: &key error: err]) {
      return TRUE;
    }
    ...
  }
  ...
}
```

Here's an example of manual propagation using an ADT in an impure functional language, SML; it's somewhat artificial because the SML library actually uses exceptions for this:

```
fun read_next_cmd () =
  case readline(stdin) of
    NONE => NONE
  | SOME line => if ...
```

All of these excerpts explicitly test for errors using the language's standard tools for data flow and then explicitly bypass the evaluation of the remainder of the function using the language's standard tools for control flow.

The other basic way to propagate errors is in some hidden, more intrinsic way not directly reflected in the ordinary control flow rules; let's call this **automatic propagation**. Here's a good example of automatic propagation using exceptions in an imperative language, Java:

```
String next = readline();
```

If `readline` encounters an error, it throws an exception; the language then terminates scopes until it dynamically reaches a `try` statement with a matching handler. Note the lack of any code at all implying that this might be happening.

The chief disadvantages of manual propagation are that it's tedious to write and requires a lot of repetitive boilerplate. This might sound superficial, but these are serious concerns. Tedium distracts programmers and makes them careless; careless error-handling code can be worse than useless. Repetitive boilerplate makes code less readable, hurting maintainability; it occupies the programmer's time, creating opportunity costs; it discourages handling errors *well* by making it burdensome to handle them *at all*; and it encourages shortcuts (such as extensive macro use) which may undermine other advantages and goals.

The chief disadvantage of automatic propagation is that it obscures the control flow of the code. I'll talk about this more in the next section.

Note that automatic propagation needn't be intrinsic in a language. The propagation is automatic if it doesn't correspond to visible constructs in the source. This effect can be duplicated as a library with any language facility that allows restructuring of code (e.g. with macros or other term-rewriting facilities) or overloading of basic syntax (e.g. Haskell mapping its `do` notation onto monads).

Note also that multiple propagation strategies may be "in play" for any particular program. For example, Java generally uses exceptions in its standard libraries, but some specific APIs might opt to instead return `null` on error for efficiency reasons. Objective-C provides a fairly full-featured exceptions model, but the standard APIs (with a few important exceptions) reserve them solely for unrecoverable errors, preferring manual propagation with `NSError` out-parameters instead. Haskell has a large number of core library functions which return `Maybe` values to indicate success or error, but it also offers at least two features resembling traditional, automatically-propagating exceptions (the `ErrorT` monad transform and exceptions in the `IO` monad).

So, while I'm going to talk as if languages implement a single propagation strategy, it should be understood that reality will always be more complex. It is literally impossible to prevent programmers from using manual propagation if they want to. Part of the proposal will discuss using multiple strategies at once.

Marked propagation

Closely related to the question of whether propagation is manual or automatic is whether it is marked or unmarked. Let's say that a language uses **marked propagation** if there is something *at the call site* which indicates that propagation is possible from that point.

To a certain extent, every language using manual propagation uses marked propagation, since the manual code to propagate the error approximately marks the call which generated the error. However, it is possible for the propagation logic to get separated from the call.

Marked propagation is at odds with one other major axis of language design: a language can't solely use marked propagation if it ever performs implicit operations that can produce errors. For example, a language that wanted out-of-memory conditions to be recoverable errors would have to consider everything that could allocate memory to a source of propagation; in a high-level language, that would include a large number of implicit operations. Such a language could not claim to use marked propagation.

The reason this all matters is because unmarked propagation is a pretty nasty thing to end up with; it makes it impossible to directly see what operations can produce errors, and therefore to directly understand the control flow of a function. This leaves you with two

options as a programmer:

- You can carefully consider the actual dynamic behavior of every function called by your function.
- You can carefully arrange your function so that there are no critical sections where a universal error can leave things in an unwanted state.

There are techniques for making the second more palatable. Chiefly, they involve never writing code that relies on normal control flow to maintain invariants and clean up after an operation; for example, always using constructors and destructors in C++ to manage resources. This is compulsory in C++ with exceptions enabled because of the possibility of implicit code that can throw, but it could theoretically be used in other languages. However, it still requires a fairly careful and rigorous style of programming.

It is possible to imagine a marked form of automatic propagation, where the propagation itself is implicit except that (local) origination points have to be explicitly marked. This is part of our proposal, and I'll discuss it below.

Typed propagation

The next major question is whether error propagation is explicitly tracked and limited by the language. That is, is there something explicitly *in the declaration of a function* that tells the programmer whether it can produce errors? Let's call this **typed propagation**.

Typed manual propagation

Whether propagation is typed is somewhat orthogonal to whether it's manual or marked, but there are some common patterns. The most dominant forms of manual propagation are all typed, since they pass the failure out of the callee, either as a direct result or in an out-parameter.

Here's another example of an out-parameter:

```
- (instancetype)initWithContentsOfURL:(NSURL *)url encoding:(NSStringEncoding)enc error:(NSError **)erro
```

Out-parameters have some nice advantages. First, they're a reliable source of marking; even if the actual propagation gets separated from the call, you can always detect a call that can generate errors as long as its out-parameter has a recognizable name. Second, some of the boilerplate can be shared, because you can use the same variable as an out-parameter multiple times; unfortunately, you can't use this to "cheat" and only check for an error once unless you have some conventional guarantee that later calls won't spuriously overwrite the variable.

A common alternative in functional languages is to return an `Either` type:

```
trait Writer {
  fn write_line(&mut self, s: &str) -> Result<(), IoError>;
}
```

This forces the caller to deal with the error if they want to use the result. This works well unless the call does not really have a meaningful result (as `write_line` does not); then it depends on whether language makes it easy to accidentally ignore results. It also tends to create a lot of awkward nesting:

```
fn parse_two_ints_and_add_them() {
  match parse_int() {
    Err e => Err e
    Ok x => match parse_int() {
      Err e => Err e
      Ok y => Ok (x + y)
    }
  }
}
```

Here, another level of nesting is required for every sequential computation that can fail. Overloaded evaluation syntax like Haskell's `do` notation would help with both of these problems, but only by switching to a kind of automatic propagation.

Manual propagation can be untyped if it occurs through a side channel. For example, consider an object which set a flag on itself when it encountered an error instead of directly returning it; or consider a variant of POSIX which expected you to separately check `errno` to see if a particular system call failed.

Typed automatic propagation

Languages with typed automatic propagation vary along several dimensions.

The default typing rule

The most important question is whether you opt in to producing errors or opt out of them. That is, is a function with no specific annotation able to produce errors or not?

The normal resilience guideline is that you want the lazier option to preserve more flexibility for the implementation. A function that can produce errors is definitely more flexible, since it can do more things. Contrariwise, changing a function that doesn't produce errors into a function that does clearly changes its contract in ways that callers need to respond to. Unfortunately, this has some unpleasant consequences:

- Marked propagation would become very burdensome. Every call would involve an annotation, either on the function (to say it cannot generate errors) or on the call site (to mark propagation). Users would likely rebel against this much bookkeeping.

- Most functions cannot generate recoverable errors in the way I've defined that. That is, ignoring sources of universal errors, most functions can be reasonably expected to not be able to produce errors. But if that's not the default state, that means that most functions would need annotations; again, that's a lot of tedious bookkeeping. It's also a lot of clutter in the API.
- Suppose that you notice that a function incorrectly lacks an annotation. You go to fix it, but you can't without annotating all of the functions it calls, ad infinitum; like `const` correctness in C++, the net effect is to punish conscientious users for trying to improve their code.
- A model which pretends that every function is a source of errors is likely to be overwhelming for humans. Programmers ought to think rigorously about their code, but expecting them to also make rigorous decisions about all the code their code touches is probably too much. Worse, without marked propagation, the compiler can't really help the programmer concentrate on the known-possible sources of error.
- The compiler's analysis for code generation has to assume that all sorts of things can produce errors when they really can't. This creates a lot of implicit propagation paths that are actually 100% dead, which imposes a serious code-size penalty.

The alternative is to say that, by default, functions are not being able to generate errors. This agrees with what I'm assuming is the most common case. In terms of resilience, it means expecting users to think more carefully about which functions can generate errors before publishing an API; but this is similar to how Swift already asks them to think carefully about types. Also, they'll have at least added the right set of annotations for their initial implementation. So I believe this is a reasonable alternative.

Enforcement

The next question is how to enforce the typing rules that prohibit automatic propagation. Should it be done statically or dynamically? That is, if a function claims to not generate errors, and it calls a function that generates errors without handling the error, should that be a compiler error or a runtime assertion?

The only real benefit of dynamic enforcement is that it makes it easier to use a function that's incorrectly marked as being able to produce errors. That's a real problem if all functions are assumed to produce errors by default, because the mistake could just be an error of omission. If, however, functions are assumed to not produce errors, then someone must have taken deliberate action that introduced the mistake. I feel like the vastly improved static type-checking is worth some annoyance in this case.

Meanwhile, dynamic enforcement undermines most of the benefits of typed propagation so completely that it's hardly worth considering. The only benefit that really remains is that the annotation serves as meaningful documentation. So for the rest of this paper, assume that typed propagation is statically enforced unless otherwise indicated.

Specificity

The last question is how specific the typing should be: should a function be able to state the specific classes of errors it produces, or should the annotation be merely boolean?

Experience with Java suggests that getting over-specific with exception types doesn't really work out for the best. It's useful to be able to recognize specific classes of error, but libraries generally want to reserve flexibility about the exact kind of error they produce, and so many errors just end up falling into broad buckets. Different libraries end up with their own library-specific general error classes, and exception lists end up just restating the library's own dependencies or wrapping the underlying errors in ways that lose critical information.

Tradeoffs of typed propagation

Typed propagation has a number of advantages and disadvantages, mostly independent of whether the propagation is automatic.

The chief advantage is that it is safer. It forces programmers to do *something* to handle or propagate errors. That comes with some downsides, which I'll talk about, but I see this as a fairly core static safety guarantee. This is especially important in an environment where shuttling operations between threads is common, since it calls out the common situation where an error needs to propagate back to the originating thread somehow.

Even if we're settled on using typed propagation, we should be aware of the disadvantages and investigate ways to ameliorate them:

- Any sort of polymorphism gets more complicated, especially higher-order functions. Functions which cannot generate errors are in principle subtypes of functions which can. But:
    - Composability suffers. A higher-order function must decide whether its function argument is allowed to generate errors. If not, the function may be significantly limiting its usability, or at least making itself much more difficult to use with error-generating functions. If so, passing a function that does not may require a conversion (an awkward explicit one if using manual propagation), and the result of the call will likely claim to be able to generate errors when, in fact, it cannot. This can be solved with overloads, but that's a lot of boilerplate and redundancy, especially for calls that take multiple functions (like the function composition operator).
    - If an implicit conversion is allowed, it may need to introduce thunks. In some cases, these thunks would be inlineable --- except that, actually, it is quite useful for code to be able to reverse this conversion and dynamically detect functions that cannot actually generate errors. For example, an algorithm might be able to avoid some unnecessary bookkeeping if it knows that its function argument never fails. This poses some representation challenges.
- It tends to promote decentralized error handling instead of letting errors propagate to a level that actually knows how to handle them.
    - Some programmers will always be tempted to incorrectly pepper their code with handlers that just swallow errors instead of correctly propagating them to the right place. This is often worse than useless; it would often be better if the error just propagated silently, because the result can be a system in an inconsistent state with no record of why. Good language and library facilities for propagating errors can help avoid this, especially when moving actions between

threads.
    - There are many situations where errors are not actually possible because the programmer has carefully restricted the input. For example, matching `/[0-9]{4}/` and then parsing the result as an integer. It needs to be convenient to do this in a context that cannot actually propagate errors, but the facility to do this needs to be carefully designed to discourage use for swallowing real errors. It might be sufficient if the facility does not actually swallow the error, but instead causes a real failure.
    - It is possible that the ease of higher-order programming in Swift might ameliorate many of these problems by letting users writing error-handling combinators. That is, in situations where a lazy Java programmer would find themselves writing a `try/catch` to swallow an exception, Swift would allow them to do something more correct with equal convenience.

One other minor advantage of marked, statically-enforced typed propagation: it's a boon for certain kinds of refactoring. Specifically, when a refactor makes an operation error-producing when it wasn't before, the absence of any those properties makes the refactor more treacherous and increases the odds of accidentally introducing a bug. If propagation is untyped, or the typing isn't statically enforced, the compiler isn't going to help you at all to find call sites which need to have error-checking code. Even with static typed propagation, if the propagation isn't marked specifically on the call site, the compiler won't warn you about calls made from contexts that can handle or implicitly propagate the error. But if all these things are true, the compiler will force you to look at all the existing call sites individually.

Error Types

There are many kinds of error. It's important to be able to recognize and respond to specific error causes programmatically. Swift should support easy pattern-matching for this.

But I've never really seen a point to coarser-grained categorization than that; for example, I'm not sure how you're supposed to react to an arbitrary, unknown IO error. And if there are useful error categories, they can probably be expressed with predicates instead of public subclasses. I think we start with a uni-type here and then challenge people to come up with reasons why they need anything more.

Implementation design

There are several different common strategies for implementing automatic error propagation. (Manual propagation doesn't need special attention in the implementation design.)

The implementation has two basic tasks common to most languages:

- Transferring control through scopes and functions to the appropriate handler for the error.
- Performing various semantic "clean up" tasks for the scopes that were abruptly terminated:
    - tearing down local variables, like C++ variables with destructors or strong/weak references in ARC-like languages;
    - releasing heap-allocated local variables, like captured variables in Swift or `__block` variables in ObjC;
    - executing scope-specific termination code, like C#'s `using` or Java/ObjC's `synchronized` statements; and
    - executing ad hoc cleanup blocks, like `finally` blocks in Java or `defer` actions in Swift.

Any particular call frame on the stack may have clean-ups or potential handlers or both; call these **interesting frames**.

Implicit manual propagation

One strategy is to implicitly produce code to check for errors and propagate them up the stack, imitating the code that the programmer would have written under manual propagation. For example, a function call could return an optional error in a special result register; the caller would check this register and, if appropriate, unwind the stack and return the same value.

Since propagation and unwinding are explicit in the generated code, this strategy hurts runtime performance along the non-error path more than the alternatives, and more code is required to do the explicitly unwinding. Branches involved in testing for errors are usually very easy to predict, so in hot code the direct performance impact is quite small, and the total impact is dominated by decreased code locality. Code can't always be hot, however.

These penalties are suffered even by uninteresting frames unless they appear in tail position. (An actual tail call isn't necessary; there just can't be anything that error propagation would skip.) And functions must do some added setup work before returning.

The upside is that the error path suffers no significant penalties beyond the code-size impact. The code-size impact can be significant, however: there is sometimes quite a lot of duplicate code needed for propagation along the error path.

This approach is therefore relatively even-handed about the error vs. the non-error path, although it requires some care in order to minimize code-size penalties for parallel error paths.

`setjmp` / `longjmp`

Another strategy to is to dynamically maintain a thread-local stack of interesting frames. A function with an interesting frame must save information about its context in a buffer, like `setjmp` would, and then register that buffer with the runtime. If the scope returns normally, the buffer is accordingly unregistered. Starting propagation involves restoring the context for the top of the interesting-frames stack; the place where execution returns is called the "landing pad".

The advantage of this is that uninteresting frames don't need to do any work; context restoration just skips over them implicitly. This is faster both for the error and non-error paths. It is also possible to optimize this strategy so that (unlike `setjmp`) the test for an error is implicitly elided: use a slightly different address for the landing pad, so that propagating errors directly restore to that location.

The downside is that saving the context and registering the frame are not free:

- Registering the frame requires an access to thread-local state, which on our platforms means a function call because we're not willing to commit to anything more specific in the ABI.
- Jumping across arbitrary frames invalidates the callee-save registers, so the registering frame must save them all eagerly. In calling conventions with many callee-save registers, this can be very expensive. However, this is only necessary when it's possible to resume normal execution from the landing pad: if the landing pad only has clean-ups and therefore always restarts propagation, those registers will have been saved and restored further out.
- Languages like C++, ObjC ARC, and Swift that have non-trivial clean-ups for many local variables tend to have many functions with interesting frames. This means both that the context-saving penalties are higher and that skipping uninteresting frames is a less valuable optimization.
- By the same token, functions in those languages often have many different clean-ups and/or handlers. For example, every new non-trivial variable might introduce a new clean-up. The function must either register a new landing pad for each clean-up (very expensive!) or track its current position in a way that a function-wide landing pad can figure out what scope it was in.

This approach can be hybridized with the unwinding approach below so that the interesting-frames stack abstractly describes the clean-ups in the frame instead of just restoring control somewhere and expecting the frame to figure it out. This can decrease the code size impact significantly for the common case of frames that just need to run some clean-ups before propagating the error further. It may even completely eliminate the need for a landing pad.

The ObjC/C++ exceptions system on iOS/ARM32 is kind of like that hybrid. Propagation and clean-up code is explicit in the function, but the registered context includes the "personality" information from the unwinding tables, which makes the decision whether to land at the landing pad at all. It also uses an optimized `setjmp` implementation that both avoids some context-saving and threads the branch as described above.

The ObjC exceptions system on pre-modern runtimes (e.g. on PPC and i386) uses the standard `setjmp` / `longjmp` functions. Every protected scope saves the context separately. This is all implemented in a very unsafe way that does not behave well in the presence of inlining.

Overall, this approach requires a lot of work in the non-error path of functions with interesting frames. Given that we expect functions with interesting frames to be very common in Swift, this is not an implementation approach we would consider in the abstract. However, it is the implementation approach for C++/ObjC exceptions on iOS/ARM32, so we need to at least interoperate with that.

Table-based unwinding

The final approach is side-table stack unwinding. This relies on being able to accurately figure out how to unwind through an arbitrary function on the system, given only the return address of a call it made and the stack pointer at that point.

On our system, this proceeds as follows. From an instruction pointer, the system unwinder looks up what linked image (executable or dylib) that function was loaded from. The linked image contains a special section, a table of unwind tables indexed by their offset within the linked image. Every non-leaf function should have an entry within this table, which provides sufficient information to unwind the function from an arbitrary call site.

This lookup process is quite expensive, especially since it has to repeat all the way up the stack until something actually handles the error. This makes the error path extremely slow. However, no explicit setup code is required along the non-error path, and so this approach is sometimes known as "zero-cost". That's something of a misnomer, because it does have several costs that can affect non-error performance. First, there's a small amount of load-time work required in order to resolve relocations to symbols used by the unwind tables. Second, the error path often requires code in the function, which can decrease code locality even if never executed. Third, the error path may use information that the non-error path would otherwise discard. And finally, the unwind tables themselves can be fairly large, although this is generally only a binary-size concern because they are carefully arranged to not need to be loaded off of disk unless an exception is thrown. But overall, "zero-cost" is close enough to correct.

To unwind a frame in this sense specifically means:

- Deciding whether the function handles the error.
- Cleaning up any interesting scopes that need to be broken down (either to get to the handler or to leave the function).
- If the function is being fully unwound, restoring any callee-save registers which the function might have changed.

This is language-specific, and so the table contains language-specific "personality" information, including a reference to a function to interpret it. This mechanism means that the unwinder is extremely flexible; not only can it support arbitrary languages, but it can support different language-specific unwinding table layouts for the same language.

Our current personality records for C++ and Objective-C contain just enough information to decide (1) whether an exception is handled by the frame and (2) if not, whether a clean-up is currently active. If either is true, it restores the context of a landing pad, which manually executes the clean-ups and enters the handler. This approach generally needs as much code in the function as implicit manual propagation would. However, we could optimize this for many common cases by causing clean-ups to be called automatically by the interpretation function. That is, instead of a landing pad that looks notionally like this:

```
void *exception = /*...*/;
SomeCXXType::~SomeCXXType(&foo);
objc_release(bar);
objc_release(baz);
_Unwind_Resume(exception);
```

The unwind table would have a record that looks notionally like this:

```
CALL_WITH_FRAME_ADDRESS(&SomeCXXType::~SomeCXXType, FRAME_OFFSET_OF(foo))
CALL_WITH_FRAME_VALUE(&objc_release, FRAME_OFFSET_OF(bar))
CALL_WITH_FRAME_VALUE(&objc_release, FRAME_OFFSET_OF(baz))
RESUME
```

And no code would actually be needed in the function. This would generally slow the error path down, because the interpretation function would have to interpret this mini-language, but it would move all the overhead out of the function and into the error table, where it would be more compact.

This is something that would also benefit C++ code.

Clean-up actions

Many languages have a built-in language tool for performing arbitrary clean-up when exiting a scope. This has two benefits. The first is that, even ignoring error propagation, it acts as a "scope guard" which ensures that the clean-up is done if the scope is exited early due to a `return`, `break`, or `continue` statement; otherwise, the programmer must carefully duplicate the clean-up in all such places. The second benefit is that it makes clean-up tractable in the face of automatic propagation, which creates so many implicit paths of control flow out of the scope that expecting the programmer to cover them all with explicit catch-and-rethrow blocks would be ridiculous.

There's an inherent tension in these language features between putting explicit clean-up code in the order it will be executed and putting it near the code it's cleaning up after. The former means that a top-to-bottom read of the code tells you what actions are being performed when; you don't have to worry about code implicitly intervening at the end of a scope. The latter makes it easy to verify at the point that a clean-up is needed that it will eventually happen; you don't need to scan down to the finally block and analyze what happens there.

`finally`

Java, Objective-C, and many other languages allow `try` statements to take a `finally` clause. The clause is an ordinary scope and may take arbitrary actions. The `finally` clause is performed when the preceding controlled scopes (including any `catch` clauses) are exited in any way: whether by falling off the end, directly branching or returning out, or throwing an exception.

`finally` is a rather awkward and verbose language feature. It separates the clean-up code from the operation that required it (although this has benefits, as discussed above). It adds a lot of braces and indentation, so edits that add new clean-ups can require a lot of code to be reformatted. When the same scope needs multiple clean-ups, the programmer must either put them in the same `finally` block (and thus create problems with clean-ups that might terminate the block early) or stack them up in separate blocks (which can really obscure the otherwise simple flow of code).

`defer`

Go provides a `defer` statement that just enqueues arbitrary code to be executed when the function exits. (More details of this appear in the survey of Go.)

This allows the defer action to be written near the code it "balances", allowing the reader to immediately see that the required clean-up will be done (but this has drawbacks, as discussed above). It's very compact, which is nice as most defer actions are short. It also allows multiple actions to pile up without adding awkward nesting. However, the function-exit semantics exacerbate the problem of searching for intervening clean-up actions, and they introduce semantic and performance problems with capturing the values of local variables.

Destructors

C++ allows types to define destructor functions, which are called when a function goes out of scope.

These are often used directly to clean up the ownership or other invariants on the type's value. For example, an owning-pointer type would free its value in its destructor, whereas a hash-table type would destroy its entries and free its buffer.

But they are also often used idiomatically just for the implicit destructor call, as a "scope guard" to ensure that something is done before the current operation completes. For an example close to my own heart, a compiler might use such a guard when parsing a local scope to ensure that new declarations are removed from the scope chains even if the function exits early due to a parse error. Unfortunately, since type destructors are C++'s only tool for this kind of clean-up, introducing ad-hoc clean-up code requires defining a new type every time.

The unique advantage of destructors compared to the options above is that destructors can be tied to temporary values created during the evaluation of an expression.

Generally, a clean-up action becomes necessary as the result of some "acquire" operation that occurs during an expression. `defer` and `finally` do not take effect until the next statement is reached, which creates an atomicity problem if code can be injected after the acquire. (For `finally`, this assumes that the acquire appears *before* the `try`. If instead the acquire appears *within* the `try`, there must be something which activates the clean-up, and that has the same atomicity problem.)

In contrast, if the acquire operation always creates a temporary with a destructor that does the clean-up, the language automatically guarantees this atomicity. This pattern is called "resource acquisition is initialization", or "RAII". Under RAII, all resources that require clean-up are carefully encapsulated within types with user-defined destructors, and the act of constructing an object of that type is exactly the act of acquiring the underlying resource.

Swift does not support user-defined destructors on value types, but it does support general RAII-like programming with class types

and `deinit` methods, although (at the moment) the user must take special care to keep the object alive, as Swift does not normally guarantee the destruction order of objects.

RAII is very convenient when there's a definable "resource" and somebody's already wrapped its acquisition APIs to return appropriately-destructed objects. For other tasks, where a reasonable programmer might balk at defining a new type and possibly wrapping an API for a single purpose, a more *ad hoc* approach may be warranted.

Survey

C

C doesn't really have a consensus error-handling scheme. There's a built-in unwinding mechanism in `setjmp` and `longjmp`, but it's disliked for a host of good reasons. The dominant idiom in practice is for a function to encode failure using some unreasonable value for its result, like a null pointer or a negative count. The bad value(s) are often function-specific, and sometimes even argument- or state-specific.

On the caller side, it is unfortunately idiomatic (in some codebases) to have a common label for propagating failure at the end of a function (hence `goto fail`); this is because there's no inherent language support for ensuring that necessary cleanup is done before propagating out of a scope.

C++

C++ has exceptions. Exceptions can have almost any type in the language. Propagation typing is tied only to declarations; an indirect function pointer is generally assumed to be able to throw. Propagation typing used to allow functions to be specific about the kinds of exceptions they could throw (`throws (std::exception)`), but this is deprecated in favor of just indicating whether a function can throw (`noexcept(false)`).

C++ aspires to making out-of-memory a recoverable condition, and so allocation can throw. Therefore, it is essentially compulsory for the language to assume that constructors might throw. Since constructors are called pervasively and implicitly, it makes sense for the default rule to be that all functions can throw. Since many error sites are implicit, there is little choice but to use automatic unmarked propagation. The only reasonable way to clean up after a scope in such a world is to allow the compiler to do it automatically. C++ programmers therefore rely idiomatically on a pattern of shifting all scope cleanup into the destructors of local variables; sometimes such local values are created solely to set up a cleanup action in this way.

Different error sites occur with a different set of cleanups active, and there are a large number of such sites. In fact, prior to C++11, compilers were forced to assume by default that destructor calls could throw, so cleanups actually created more error sites. This all adds up to a significant code-size penalty for exceptions, even in projects which don't directly use them and which have no interest in recovering from out-of-memory conditions. For this reason, many C++ projects explicitly disable exceptions and rely on other error propagation mechanisms, on which there is no widespread consensus.

Objective-C

Objective-C has a first-class exceptions mechanism which is similar in feature set to Java's: `@throw` / `@try` / `@catch` / `@finally`. Exception values must be instances of an Objective-C class. The language does a small amount of implicit frame cleanup during exception propagation: locks held by `@synchronized` are released, stack copies of `__block` variables are torn down, and ARC `__weak` variables are destroyed. However, the language does not release object pointers held in local variables, even (by default) under ARC.

Objective-C exceptions used to be implemented with `setjmp`, `longjmp`, and thread-local state managed by a runtime, but the only surviving platform we support which does that is i386, and all others now use a "zero-cost" implementation that interoperates with C++ exceptions.

Objective-C exceptions are *mostly* only used for unrecoverable conditions, akin to what I called "failures" above. There are a few major exceptions to this rule, where APIs do use exceptions to report errors.

Instead, Objective-C mostly relies on manual propagation, predominantly using out-parameters of type `NSError**`. Whether the call failed is usually *not* indicated by whether a non-`nil` error was written into this parameter; calls are permitted both to succeed and write an error object into the parameter (which should be ignored) and to report an error without creating an actual error object. Instead, whether the call failed is reported in the formal return value. The most common convention is for a false `BOOL` result or null object result to mean an error, but ingenious programmers have come up with many other conventions, and there do exist APIs where a null object result is valid.

CF APIs, meanwhile, have their own magnificent set of somewhat inconsistent conventions.

Therefore, we can expect that incrementally improving CF / Objective-C interoperation is going to be a long and remarkably painful process.

Java

Java has a first-class exceptions mechanism with unmarked automatic propagation: `throw` / `try` / `catch` / `finally`. Exception values must be instances of something inheriting from `Throwable`. Propagation is generally typed with static enforcement, with the default being that a call cannot throw exceptions *except* for subclasses of `Error` and `RuntimeException`. The original intent was that these classes would be used for catastrophic runtime errors (`Error`) and programming mistakes caught by the runtime (`RuntimeException`), both of which we would classify as unrecoverable failures in our scheme; essentially, Java attempts to

promote a fully statically-enforced model where truly catastrophic problems can still be handled when necessary. Unfortunately, these motivations don't seem to have been communicated very well to developers, and the result is kind of a mess.

Java allows methods to be very specific about the kinds of exception they throw. In my experience, exceptions tend to fall into two categories:

- There are some very specific exception kinds that callers know to look for and handle on specific operations. Generally these are obvious, predictable error conditions, like a host name not resolving, or like a string not being formatted correctly.
- There are also a lot of very vague, black-box exception kinds that can't really be usefully responded to. For example, if a method throws `IOException`, there's really nothing a caller can do except propagate it and abort the current operation.

So specific typing is useful if you can exhaustively handle a small number of specific failures. As soon as the exception list includes any kind of black box type, it might as well be a completely open set.

C#

C#'s model is almost exactly like Java's except that it is untyped: all methods are assumed to be able to throw. For this reason, it also has a simpler type hierarchy, where all exceptions just inherit from `Exception`.

The rest of the hierarchy doesn't really make any sense to me. Many things inherit directly from `Exception`, but many other things inherit from a subclass called `SystemException`. `SystemException` doesn't seem to be any sort of logical grouping: it includes all the runtime-assertion exceptions, but it also includes every exception that's thrown anywhere in the core library, including XML and IO exceptions.

C# also has a `using` statement, which is useful for binding something over a precise scope and then automatically disposing it on all paths. It's just built on top of `try` / `finally`.

Haskell

Haskell provides three different common error-propagation mechanisms.

The first is that, like many other functional languages, it supports manual propagation with a `Maybe` type. A function can return `Nothing` to indicate that it couldn't produce a more useful result. This is the most common failure method for functions in the functional subset of the library.

The `IO` monad also provides true exceptions with unmarked automatic propagation. These exceptions can only be handled as an `IO` action, but are otherwise untyped: there is no way to indicate whether an `IO` action can or cannot throw. Exceptions can be thrown either as an `IO` action or as an ordinary lazy functional computation; in the latter case, the exception is only thrown if the computation is evaluated for some reason.

The `ErrorT` monad transform provides typed automatic propagation. In an amusing twist, since the only native computation of `ErrorT` is `throwError`, and the reason to write a computation specifically in `ErrorT` is if it's throwing, and every other computation must be explicitly lifted into the monad, `ErrorT` effectively uses marked propagation by omission, since everything that *can't* throw is explicitly marked with a `lift`:

```
prettyPrintShiftJIS :: ShiftJISString -> ErrorT TranscodeError IO ()
prettyPrintShiftJIS str = do
  lift $ putChar '"'     -- lift turns an IO computation into an ErrorT computation
  case transcodeShiftJISToUTF8 str of
    Left error -> throwError error
    Right value -> lift $ putEscapedString value
  lift $ putChar '"'
```

Rust

Rust distinguishes between *failures* and *panics*.

A panic is an assertion, designed for what I called logic failures; there's no way to recover from one, it just immediately crashes.

A failure is just when a function doesn't produce the value you might expect, which Rust encourages you to express with either `Option<T>` (for simple cases, like what I described as simple domain errors) or `Result<T>` (which is effectively the same, except carrying an error). In either case, it's typed manual propagation, although Rust does at least offer a standard macro which wraps the common pattern-match-and-return pattern for `Result<T>`.

The error type in Rust is a very simple protocol, much like this proposal suggests.

Go

Go uses an error result, conventionally returned as the final result of functions that can fail. The caller is expected to manually check whether this is nil; thus, Go uses typed manual propagation.

The error type in Go is an interface named `error`, with one method that returns a string description of the error.

Go has a `defer` statement:

```
defer foo(x, y)
```

The argument has to be a call (possibly a method call, possibly a call to a closure that you made specifically to immediately call). All the operands are evaluated immediately and captured in a deferred action. Immediately after the function exits (through whatever

means), all the deferred actions are executed in LIFO order. Yes, this is tied to function exit, not scope exit, so you can have a dynamic number of deferred actions as a sort of implicit undo stack. Overall, it's a nice if somewhat quirky way to do ad-hoc cleanup actions.

It is also a key part of a second, funky kind of error propagation, which is essentially untyped automatic propagation. If you call `panic` --- and certain builtin operations like array accesses behave like they do --- it immediately unwinds the stack, running deferred actions as it goes. If a function's deferred action calls `recover`, the panic stops, the rest of the deferred actions for the function are called, and the function returns. A deferred action can write to the named results, allowing a function to turn a panic error into a normal, final-result error. It's conventional to not panic over API boundaries unless you really mean it; recoverable errors are supposed to be done with out-results.

Scripting languages

Scripting languages generally all use (untyped, obviously) automatic exception propagation, probably because it would be quite error-prone to do manual propagation in an untyped language. They pretty much all fit into the standard C++/Java/C# style of `throw` / `try` / `catch`. Ruby uses different keywords for it, though.

I feel like Python uses exceptions a lot more than most other scripting languages do, though.

Proposal

Automatic propagation

Swift should use automatic propagation of errors, rather than relying on the programmer to manually check for them and return out. It's just a lot less boilerplate for common error handling tasks. This introduces an implicit control flow problem, but we can ameliorate that with marked propagation; see below.

There's no compelling reason to deviate from the `throw` / `catch` legacy here. There are other options, like `raise` / `handle`. In theory, switching would somewhat dissociate Swift from the legacy of exceptions; people coming from other languages have a lot of assumptions about exceptions which don't necessarily apply to Swift. However, our error model is similar enough to the standard exception model that people are inevitably going to make the connection; there's no getting around the need to explain what we're trying to do. So using different keywords just seems petty.

Therefore, Swift should provide a `throw` expression. It requires an operand of type `Error` and formally yields an arbitrary type. Its dynamic behavior is to transfer control to the innermost enclosing `catch` clause which is satisfied by the operand. A quick example:

```
if timeElapsed() > timeThreshold { throw HomeworkError.Overworked }
```

A `catch` clause includes a pattern that matches an error. We want to repurpose the `try` keyword for marked propagation, which it seems to fit far better, so `catch` clauses will instead be attached to a generalized `do` statement:

```
do {
  ...

} catch HomeworkError.Overworked {
  // a conditionally-executed catch clause

} catch _ {
  // a catch-all clause
}
```

Swift should also provide some tools for doing manual propagation. We should have a standard Rust-like `Result<T>` enum in the library, as well as a rich set of tools, e.g.:

- A function to evaluate an error-producing closure and capture the result as a `Result<T>`.
- A function to unpack a `Result<T>` by either returning its value or propagating the error in the current context.
- A futures library that traffics in `Result<T>` when applicable.
- An overload of `dispatch_sync` which takes an error-producing closure and propagates an error in the current context.
- etc.

Typed propagation

Swift should use statically-enforced typed propagation. By default, functions should not be able to throw. A call to a function which can throw within a context that is not allowed to throw should be rejected by the compiler.

Function types should indicate whether the function throws; this needs to be tracked even for first-class function values. Functions which do not throw are subtypes of functions that throw.

This would be written with a `throws` clause on the function declaration or type:

```
// This function is not permitted to throw.
 func foo() -> Int {
   // Therefore this is a semantic error.
   return try stream.readInt()
 }

 // This function is permitted to throw.
```

```
func bar() throws -> Int {
  return try stream.readInt()
}

// 'throws' is written before the arrow to give a sensible and
// consistent grammar for function types and implicit () result types.
func baz() throws {
  if let byte = try stream.getOOB() where byte == PROTO_RESET {
    reset()
  }
}

// 'throws' appears in a consistent position in function types.
func fred(_ callback: (UInt8) throws -> ()) throws {
   while true {
     let code = try stream.readByte()
     if code == OPER_CLOSE { return }
     try callback(code)
   }
}

// It only applies to the innermost function for curried functions;
// this function has type:
//   (Int) -> (Int) throws -> Int
func jerry(_ i: Int)(j: Int) throws -> Int {
  // It's not an error to use 'throws' on a function that can't throw.
  return i + j
}
```

The reason to use a keyword here is that it's much nicer for function declarations, which generally outnumber function types by at least an order of magnitude. A punctuation mark would be easily lost or mistaken amidst all the other punctuation in a function declaration, especially if the punctuation mark were something like `!` that can validly appear at the end of a parameter type. It makes sense for the keyword to appear close to the return type, as it's essentially a part of the result and a programmer should be able to see both parts in the same glance. The keyword appears before the arrow for the simple reason that the arrow is optional (along with the rest of the return type) in function and initializer declarations; having the keyword appear in slightly different places based on the presence of a return type would be silly and would make adding a non-void return type feel awkward. The keyword itself should be descriptive, and it's particularly nice for it to be a form of the verb used by the throwing expression, conjugated as if performed by the function itself. Thus, `throw` becomes `throws`; if we used `raise` instead, this would be `raises`, which I personally find unappealing for reasons I'm not sure I can put a name to.

It shouldn't be possible to overload functions solely based on whether the functions throw. That is, this is not legal:

```
func foo() { ... } // called in contexts that cannot throw
func foo() throws { ... } // called in contexts that can throw
```

It is valuable to be able to overload higher-order functions based on whether an argument function throws; it is easy to imagine algorithms that can be implemented more efficiently if they do not need to worry about exceptions. (We do not, however, particularly want to encourage a pattern of duplicating. This is straightforward if the primary type-checking pass is able to reliably decide whether a function value can throw.)

Typed propagation checking can generally be performed in a secondary pass over a type-checked function body: if a function is not permitted to throw, walk its body and verify that there are no `throw` expressions or calls to functions that can `throw`. If all throwing calls must be marked, this can be done prior to type-checking to decide syntactically whether a function can apparently throw; of course, the later pass is still necessary, but the ability to do this dramatically simplifies the implementation of the type-checker, as discussed below. Certain type-system features may need to be curtailed in order to make this implementation possible for schedule reasons. (It's important to understand that this is *not* the motivation for marked propagation. It's just a convenient consequence that marked propagation makes this implementation possible.)

Reliably deciding whether a function value can throw is easy for higher-order uses of declared functions. The problem, as usual, is anonymous functions. We don't want to require closures to be explicitly typed as throwing or non-throwing, but the fully-accurate inference algorithm requires a type-checked function body, and we can't always type-check an anonymous function independently of its enclosing context. Therefore, we will rely on being able to do a pass prior to type-checking to syntactically infer whether a closure throws, then making a second pass after type-checking to verify the correctness of that inference. This may break certain kinds of reasonable code, but the multi-pass approach should let us heuristically unbreak targeted cases.

Typed propagation has implications for all kinds of polymorphism:

Higher-order polymorphism

We should make it easy to write higher-order functions that behave polymorphically w.r.t. whether their arguments throw. This can be done in a fairly simple way: a function can declare that it throws if any of a set of named arguments do. As an example (using strawman syntax):

```
func map<T, U>(_ array: [T], fn: T throws -> U) throwsIf(fn) -> [U] {
  ...
}
```

There's no need for a more complex logical operator than disjunction. You can construct really strange code where a function throws

only if one of its arguments doesn't, but it'd be contrived, and it's hard to imagine how they could be type-checked without a vastly more sophisticated approach. Similarly, you can construct situations where whether a function can throw is value-dependent on some other argument, like a "should I throw an exception" flag, but it's hard to imagine such cases being at all important to get right in the language. This schema is perfectly sufficient to express normal higher-order stuff.

In fact, while the strawman syntax above allows the function to be specific about exactly which argument functions cause the callee to throw, that's already overkill in the overwhelmingly likely case of a function that throws if any of its argument functions throw (and there's probably only one). So it would probably be better to just have a single `rethrows` annotation, with vague plans to allow it to be parameterized in the future if necessary.

This sort of propagation-checking would be a straightforward extension of the general propagation checker. The normal checker sees that a function isn't allowed to propagate out and looks for propagation points. The conditional checker sees that a function has a conditional propagation clause and looks for propagation points, assuming that the listed functions don't throw (including when looking at any conditional propagation clauses). The parameter would have to be a `let`.

We probably do need to get higher-order polymorphism right in the first release, because we will need it for the short-circuiting operators.

Generic polymorphism

It would be useful to be able to parameterize protocols, and protocol conformances, on whether the operations produce errors. Lacking this feature means that protocol authors must decide to either conservatively allow throwing conformances, and thus force all generic code using the protocol to deal with probably-spurious errors, or aggressively forbid them, and thus forbid conformances by types whose operations naturally throw.

There are several different ways we could approach this problem, and after some investigation I feel confident that they're workable. Unfortunately, they are clearly out-of-scope for the first release. For now, the standard library should provide protocols that cannot throw, even though this limits some potential conformances. (It's worth noting that such conformances generally aren't legal today, since they'd need to return an error result somehow.)

A future direction for both generic and higher-order polymorphism is to consider error propagation to be one of many possible effects in a general, user-extensible effect tracking system. This would allow the type system to check that certain specific operations are only allowed in specific contexts: for example, that a blocking operation is only allowed in a blocking context.

Error type

The Swift standard library will provide `Error`, a protocol with a very small interface (which is not described in this proposal). The standard pattern should be to define the conformance of an `enum` to the type:

```
enum HomeworkError : Error {
  case Overworked
  case Impossible
  case EatenByCat(Cat)
  case StopStressingMeWithYourRules
}
```

The `enum` provides a namespace of errors, a list of possible errors within that namespace, and optional values to attach to each option.

For now, the list of errors in a domain will be fixed, but permitting future extension is just ordinary enum resilience, and the standard techniques for that will work fine in the future.

Note that this corresponds very cleanly to the `NSError` model of an error domain, an error code, and optional user data. We expect to import system error domains as enums that follow this approach and implement `Error`. `NSError` and `CFError` themselves will also conform to `Error`.

The physical representation (still being nailed down) will make it efficient to embed an `NSError` as an `Error` and vice-versa. It should be possible to turn an arbitrary Swift `enum` that conforms to `Error` into an `NSError` by using the qualified type name as the domain key, the enumerator as the error code, and turning the payload into user data.

It's acceptable to allocate memory whenever an error is needed, but our representation should not inhibit the optimizer from forwarding a `throw` directly to a `catch` and removing the intermediate error object.

Marked propagation

Swift should use marked propagation: there should be some lightweight bit of syntax decorating anything that is known be able to throw (other than a `throw` expression itself, of course).

Our proposed syntax is to repurpose `try` as something that can be wrapped around an arbitrary expression:

```
// This try applies to readBool().
if try stream.readBool() {

  // This try applies to both of these calls.
  let x = try stream.readInt() + stream.readInt()

  // This is a semantic error; it needs a try.
  var y = stream.readFloat()
```

```
  // This is okay; the try covers the entire statement.
  try y += stream.readFloat()
}
```

Developers can "scope" the `try` very tightly by writing it within parentheses or on a specific argument or list element:

```
// Semantic error: the try only covers the parenthesized expression.
let x = (try stream.readInt()) + stream.readInt()

// The try applies to the first array element.  Of course, the
// developer could cover the entire array by writing the try outside.
let array = [ try foo(), bar(), baz() ]
```

Some developers may wish to do this to make the specific throwing calls very clear. Other developers may be content with knowing that something within a statement can throw.

We also briefly considered the possibility of putting the marker into the call arguments clause, e.g.:

```
parser.readKeys(&strings, try)
```

This works as long as the only throwing calls are written syntactically as calls; this covers calls to free functions, methods, and initializers. However, it effectively requires Swift to forbid operators and property and subscript accessors from throwing, which may not be a reasonable limitation, especially for operators. It is also somewhat unnatural, and it forces users to mark every single call site instead of allowing them to mark everything within a statement at once.

Autoclosures pose a problem for marking. For the most part, we want to pretend that the expression of an autoclosure is being evaluated in the enclosing context; we don't want to have to mark both a call within the autoclosure and the call to the function taking the autoclosure! We should teach the type-checking pass to recognize this pattern: a call to a function that `throwsIf` an autoclosure argument does.

There's a similar problem with functions that are supposed to feel like statements. We want you to be able to write:

```
autoreleasepool {
  let string = parseString(try)
  ...
}
```

without marking the call to `autoreleasepool`, because this undermines the ability to write functions that feel like statements. However, there are other important differences between these trailing-closure uses and true built-in statements, such as the behavior of `return`, `break`, and `continue`. An attribute which marks the function as being statement-like would be a necessary step towards addressing both problems. Doing this reliably in closures would be challenging, however.

Asserting markers

Typed propagation is a hypothesis-checking mechanism and so suffers from the standard problem of false positives. (Basic soundness eliminates false negatives, of course: the compiler is supposed to force programmers to deal with *every* source of error.) In this case, a false positive means a situation where an API is declared to throw but an error is actually dynamically impossible.

For example, a function to load an image from a URL would usually be designed to produce an error if the image didn't exist, the connection failed, the file data was malformed, or any of a hundred other problems arose. The programmer should be expected to deal with that error in general. But a programmer might reasonably use the same API to load an image completely under their control, e.g. from their program's private resources. We shouldn't make it too syntactically inconvenient to "turn off" error-checking for such calls.

One important point is that we don't want to make it too easy to *ignore* errors. Ignored errors usually lead to a terrible debugging experience, even if the error is logged with a meaningful stack trace; the full context of the failure is lost and can be difficult to reproduce. Ignored errors also have a way of compounding, where an error that's "harmlessly" ignored at one layer of abstraction causes another error elsewhere; and of course the second error can be ignored, etc., but only by making the program harder and harder to understand and debug, leaving behind log files that are increasingly jammed with the detritus of a hundred ignored errors. And finally, ignoring errors creates a number of type-safety and security problems by encouraging programs to blunder onwards with meaningless data and broken invariants.

Instead, we just want to make it (comparatively) easy to turn a static problem into a dynamic one, much as assertions and the ! operator do. Of course, this needs to be an explicit operation, because otherwise we would completely lose typed propagation; and it should be call-specific, so that the programmer has to make an informed decision about individual operations. But we already have an explicit, call-site-specific annotation: the `try` operator. So the obvious solution is to allow a variant of `try` that asserts that an error is not thrown out of its operand; and the obvious choice there within our existing design language is to use the universal "be careful, this is unsafe" marker by making the keyword `try!`.

It's reasonable to ask whether `try!` is actually *too* easy to write, given that this is, after all, an unsafe operation. One quick rejoinder is that it's no worse than the ordinary `!` operator in that sense. Like `!`, it's something that a cautious programmer might want to investigate closer, and you can easily imagine codebases that expect uses of it to always be explained in comments. But more importantly, just like `!` it's only *statically* unsafe, and it will reliably fail when the programmer is wrong. Therefore, while you can easily imagine (and demonstrate) incautious programmers flailing around with it to appease the type-checker, that's not actually a tenable position for the overall program: eventually the programmer will have to learn how to use the feature, or else their program simply won't run.

Furthermore, while `try!` does somewhat undermine error-safety in the hands of a careless programmer, it's still better to promote

this kind of unsafety than to implicitly promote the alternative. A careless programmer isn't going to write good error handling just because we don't give them this feature. Instead, they'll write out a `do/catch` block, and the natural pressure there will be to silently swallow the error --- after all, that takes less boilerplate than asserting or logging.

In a future release, when we add support for universal errors, we'll need to reconsider the behavior of `try!`. One possibility is that `try!` should simply start propagating its operand as a universal error; this would allow emergency recovery. Alternatively, we may want `try!` to assert that even universal errors aren't thrown out of it; this would provide a more consistent language model between the two kinds of errors. But we don't need to think too hard about this yet.

Other syntax

Clean-up actions

Swift should provide a statement for cleaning up with an *ad hoc* action.

Overall, I think it is better to use a Go-style `defer` than a Java-style `try ... finally`. While this makes the exact order of execution more obscure, it does make it obvious that the clean-up *will* be executed without any further analysis, which is something that readers will usually be interested in.

Unlike Go, I think this should be tied to scope-exit, not to function-exit. This makes it very easy to know the set of `defer` actions that will be executed when a scope exits: it's all the `defer` statement in exactly that scope. In contrast, in Go you have to understand the dynamic history of the function's execution. This also eliminates some semantic and performance oddities relating to variable capture, since the `defer` action occurs with everything still in scope. One downside is that it's not as good for "transactional" idioms which push an undo action for everything they do, but that style has composition problems across function boundaries anyway.

I think `defer` is a reasonable name for this, although we might also consider `finally`. I'll use `defer` in the rest of this proposal.

`defer` may be followed by an arbitrary statement. The compiler should reject an action that might terminate early, whether by throwing or with `return`, `break`, or `continue`.

Examples:

```
if exists(filename) {
  let file = open(filename, O_READ)
  defer close(file)

  while let line = try file.readline() {
    ...
  }

  // close occurs here, at the end of the formal scope.
}
```

We should consider providing a convenient way to mark that a `defer` action should only be taken if an error is thrown. This is a convenient shorthand for controlling the action with a flag that's only set to true at the end of an operation. The flag approach is often more useful, since it allows the action to be taken for *any* early exit, e.g. a `return`, not just for error propagation.

`using`

Swift should consider providing a `using` statement which acquires a resource, holds it for a fixed period of time, optionally binds it to a name, and then releases it whenever the controlled statement exits.

`using` has many similarities to `defer`. It does not subsume `defer`, which is useful for many ad-hoc and tokenless clean-ups. But it is convenient for the common pattern of a type-directed clean-up.

We do not expect this feature to be necessary in the first release.

C and Objective-C Interoperation

It's of paramount importance that Swift's error model interact as cleanly with Objective-C APIs as we can make it.

In general, we want to try to import APIs that produce errors as throwing; if this fails, we'll import the API as an ordinary non-throwing function. This is a safe approach only under the assumption that importing the function as throwing will require significant changes to the call. That is, if a developer writes code assuming that an API will be imported as throwing, but in fact Swift fails to import the API that way, it's important that the code doesn't compile.

Fortunately, this is true for the common pattern of an error out-parameter: if Swift cannot import the function as throwing, it will leave the out-parameter in place, and the compiler will complain if the developer fails to pass an error argument. However, it is possible to imagine APIs where the "meat" of the error is returned in a different way; consider a POSIX API that simply sets `errno`. Great care would need to be taken when such an API is only partially imported as throwing.

Let's wade into the details.

Error types

`NSError` and `CFError` should implement the `Error` protocol. It should be possible to turn an arbitrary Swift `enum` that conforms to `Error` into an `NSError` by using the qualified type name as the domain key, the enumerator as the error code, and turning the payload into user data.

Recognizing system enums as error domains is a matter of annotation. Most likely, Swift will just special-case a few common domains in the first release.

Objective-C method error patterns

The most common error pattern in ObjC by far is for a method to have an autoreleased `NSError**` out-parameter. We don't currently propose automatically importing anything as `throws` when it lacks such a parameter.

If any APIs take an `NSError**` and *don't* intend for it to be an error out-parameter, they will almost certainly need it to be marked.

Detecting an error

Many of these methods have some sort of significant result which is used for testing whether an error occurred:

- The most common pattern is a `BOOL` result, where a false value means an error occurred. This seems unambiguous.

  Swift should import these methods as if they'd returned `Void`.

- Also common is a pointer result, where a `nil` result usually means an error occurred.

  I've been told that there are some exceptions to this rule, where a `nil` result is valid and the caller is apparently meant to check for a non-`nil` error. I haven't been able to find any such APIs in Cocoa, though; the claimed APIs I've been referred to do have nullable results, but returned via out-parameters with a *BOOL* formal result. So it seems to be a sound policy decision for Objective-C that `nil` results are errors by default. CF might be a different story, though.

  When a `nil` result implies that an error has occurred, Swift should import the method as returning a non-optional result.

- A few CF APIs return `void`. As far as I can tell, for all of these, the caller is expected to check for a non-`nil` error.

For other sentinel cases, we can consider adding a new clang attribute to indicate to the compiler what the sentinel is:

- There are several APIs returning `NSInteger` or `NSUInteger`. At least some of these return 0 on error, but that doesn't seem like a reasonable general assumption.
- `AVFoundation` provides a couple methods returning `AVKeyValueStatus`. These produce an error if the API returned `AVKeyValueStatusFailed`, which, interestingly enough, is not the zero value.

The clang attribute would specify how to test the return value for an error. For example:

```
+ (NSInteger)writePropertyList:(id)plist
                      toStream:(NSOutputStream *)stream
                        format:(NSPropertyListFormat)format
                       options:(NSPropertyListWriteOptions)opt
                         error:(out NSError **)error
  NS_ERROR_RESULT(0)

- (AVKeyValueStatus)statusOfValueForKey:(NSString *)key
                                  error:(NSError **)
  NS_ERROR_RESULT(AVKeyValueStatusFailed);
```

We should also provide a Clang attribute which specifies that the correct way to test for an error is to check the out-parameter. Both of these attributes could potentially be used by the static analyzer, not just Swift. (For example, they could try to detect an invalid error check.)

A constant value would be sufficient for the cases I've seen, but if the argument has to generalized to a simple expression, that's still feasible.

The error parameter

The obvious import rule for Objective-C methods with `NSError**` out-parameters is to simply mark them `throws` and remove the selector clause corresponding to the out-parameter. That is, a method like this one from `NSAttributedString`:

```
- (NSData *)dataFromRange:(NSRange)range
       documentAttributes:(NSDictionary *)dict
                    error:(NSError **)error;
```

would be imported as:

```
func dataFromRange(_ range: NSRange,
                   documentAttributes dict: NSDictionary) throws -> NSData
```

However, applying this rule haphazardly causes problems for Objective-C interoperation, because multiple methods can be imported the same way. The model is far more comprehensible to both compiler and programmer if the original Objective-C declaration can be unambiguously reconstructed from a Swift declaration.

There are two sources of this ambiguity:

- The error parameter could have appeared at an arbitrary position in the selector; that is, both `foo:bar:error:` and `foo:error:bar:` would appear as `foo:bar:` after import.
- The error parameter could have had an arbitrary selector chunk; that is, both `foo:error:` and `foo:withError:` would appear as `foo:` after import.

To allow reconstruction, then, we should only apply the rule when the error parameter is the last parameter and the corresponding

selector is either `error:` or the first chunk. Empirically, this seems to do the right thing for all but two sets of APIs in the public API:

- The `ISyncSessionDriverDelegate` category on `NSObject` declares half-a-dozen methods like this:

  ```
  - (BOOL)sessionDriver:(ISyncSessionDriver *)sender
          didRegisterClientAndReturnError:(NSError **)outError;
  ```

  Fortunately, these delegate methods were all deprecated in Lion, and Swift currently doesn't even import deprecated methods.

- `NSFileCoordinator` has half a dozen methods where the `error:` clause is second-to-last, followed by a block argument. These methods are not deprecated as far as I know.

Of course, user code could also fail to follow this rule.

I think it's acceptable for Swift to just not import these methods as `throws`, leaving the original error parameter in place exactly as if they didn't follow an intelligible pattern in the header.

This translation rule would import methods like this one from `NSDocument`:

```
- (NSDocument *)duplicateAndReturnError:(NSError **)outError;
```

like so:

```
func duplicateAndReturnError() throws -> NSDocument
```

Leaving the `AndReturnError` bit around feels unfortunate to me, but I don't see what we could do without losing the ability to automatically reconstruct the Objective-C signature. This pattern is common but hardly universal; consider this method from `NSManagedObject`:

```
- (BOOL)validateForDelete:(NSError **)error;
```

This would be imported as:

```
func validateForDelete() throws
```

This seems like a really nice import.

CoreFoundation functions

CF APIs use `CFErrorRef` pretty reliably, but there are two problems.

First, we're not as confident about the memory management rules for the error object. Is it always returned at +1?

Second, I'm not as confident about how to detect that an error has occurred:

- There are a lot of functions that return `Boolean` or `bool`. It's likely that these functions consistently use the same convention as Objective-C: false means error.
- Similarly, there are many functions that return an object reference. Again, we'd need a policy on whether to treat `nil` results as errors.
- There are a handful of APIs that return a `CFIndex`, all with apparently the same rule that a zero value means an error. (These are serialization APIs, so writing nothing seems like a reasonable error.) But just like Objective-C, that does not seem like a reasonable default assumption.
- `ColorSyncProfile` has several related functions that return `float`! These are both apparently meant to be checked by testing whether the error result was filled in.

There are also some APIs that do not use `CFErrorRef`. For example, most of the `CVDisplayLink` APIs in CoreVideo returns their own `CVReturn` enumeration, many with more than one error value. Obviously, these will not be imported as throwing unless CoreVideo writes an overlay.

Other C APIs

In principle, we could import POSIX functions into Swift as throwing functions, filling in the error from `errno`. It's nearly impossible to imagine doing this with an automatic import rule, however; much more likely, we'd need to wrap them all in an overlay.

Implementation design

Error propagation for the kinds of explicit, typed errors that I've been focusing on should be handled by implicit manual propagation. It would be good to bias the implementation somewhat towards the non-error path, perhaps by moving error paths to the ends of functions and so on, and perhaps even by processing cleanups with an interpretive approach instead of directly inlining that code, but we should not bias so heavily as to seriously compromise performance. In other words, we should not use table-based unwinding.

Error propagation for universal errors should be handled by table-based unwinding. `catch` handlers can catch both, mapping unwind exceptions to `Error` values as necessary. With a carefully-designed interpretation function aimed to solve the specific needs of Swift, we can avoid most of the code-size impact by shifting it to the unwind tables, which needn't ever be loaded in the common case.