Error Handling Approaches in Programming Languages

Joey Aldrin Rees-Hill

Oberlin College

Follow this and additional works at: https://digitalcommons.oberlin.edu/honors

Part of the Computer Sciences Commons

Repository Citation
https://digitalcommons.oberlin.edu/honors/855

This Thesis - Open Access is brought to you for free and open access by the Student Work at Digital Commons at Oberlin. It has been accepted for inclusion in Honors Papers by an authorized administrator of Digital Commons at Oberlin. For more information, please contact megan.mitchell@oberlin.edu.
Error Handling Approaches in Programming Languages

JOEY REES-HILL, Oberlin College, USA

Error handling is a part of nearly every computer program, but it is rarely the main focus of a program’s developers. Nevertheless, correct error handling is important because it can enable a program to recover from abnormal circumstances and continue to function and serve its purpose. Programming languages offer a variety of tools and approaches for programs to detect and handle errors. I investigated the different approaches to error handling in contemporary programming languages.

I found three general paradigms of error handling approaches. One paradigm was Special Return Value, in which certain return values of a function signify that an error occurred. Another paradigm was Try-Catch, where exception objects can be raised at some point in code and intercepted and handled by guards further up in the call stack. The final paradigm was Type System Approaches, in which languages used their type system to encode possible failures and ensure proper handling of these states. Besides the three main paradigms, I found several smaller or auxiliary features, such as Defer, Checked Exceptions, and Automatically Closing Resources.

In addition, I compared an error handling feature originally created as an academic proposal (Compensations) with actual implementations of a similar feature in modern languages, finding that the actual implementation is more straightforward at the cost of being somewhat less powerful.

1 INTRODUCTION

Most of the time, when we think about using a programming language to write a program that accomplishes some task, we focus on the “go-right” case, where program execution continues as we expect it to. But almost every sufficiently complicated program contains operations that might fail, and specifies how to handle these failure cases. Different programming languages take different approaches when it comes to detecting and responding to failure conditions. This could have a significant impact on the correctness and resiliency of complicated programs with intricate error handling.

In this paper, the term “error” refers to any erroneous runtime condition which a program might want to detect and recover from. Examples of possible errors include division by zero, failure to open a file, and trying to use a \texttt{null} value as if it is non-\texttt{null}. The term “exception” is used to refer to a language feature in some programming languages following the Try-Catch pattern (section 2.2.2).

Error handling is a significant part of every programming language because it shapes how programs will behave in the presence of abnormal or incorrect conditions, which is important in many applications. But error handling is not something that most programmers will spend very much time thinking about, unless something goes particularly wrong. Thus, it’s important that programming languages lead programmers towards correct and robust error handling.

2 LANGUAGE SURVEY

I performed a survey of error handling approaches in a sample of programming languages. The aim of this survey was to understand the approaches that are in use today. I found several approaches and features that were common across multiple languages, or were otherwise interesting.
Table 1. Languages surveyed

<table>
<thead>
<tr>
<th>Language</th>
<th>Age</th>
<th>Paradigm</th>
<th>Type system</th>
<th>Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>31</td>
<td>Imp., OOP, func.</td>
<td>dynamic</td>
<td>scripting, data science, web apps</td>
</tr>
<tr>
<td>Ruby</td>
<td>27</td>
<td>Imp., OOP, func.</td>
<td>dynamic</td>
<td>scripting, web apps</td>
</tr>
<tr>
<td>JavaScript</td>
<td>26</td>
<td>Imperative</td>
<td>dynamic</td>
<td>web front- and back-end</td>
</tr>
<tr>
<td>TypeScript</td>
<td>9</td>
<td>Imperative</td>
<td>static</td>
<td>web front- and back-end</td>
</tr>
<tr>
<td>Go</td>
<td>12</td>
<td>Imp., OOP</td>
<td>static</td>
<td>web infrastructure</td>
</tr>
<tr>
<td>D</td>
<td>20</td>
<td>Imp., OOP, func.</td>
<td>static</td>
<td>systems programming</td>
</tr>
<tr>
<td>C</td>
<td>50</td>
<td>Imperative</td>
<td>static</td>
<td>sys. prog., OSes, much more</td>
</tr>
<tr>
<td>Rust</td>
<td>11</td>
<td>Imp., func.</td>
<td>static</td>
<td>sys. prog., utilities, Firefox</td>
</tr>
<tr>
<td>Haskell</td>
<td>32</td>
<td>Pure functional</td>
<td>static</td>
<td>academics, utilities</td>
</tr>
<tr>
<td>Java</td>
<td>26</td>
<td>Imp., OOP</td>
<td>static</td>
<td>general purpose, Android apps</td>
</tr>
<tr>
<td>Elm</td>
<td>10</td>
<td>Pure functional</td>
<td>static</td>
<td>web apps</td>
</tr>
<tr>
<td>Perl</td>
<td>24</td>
<td>Imp., OOP, func.</td>
<td>dynamic</td>
<td>web</td>
</tr>
</tbody>
</table>

The age of each language is given in years since its initial release. “Imperative” is abbreviated as “Imp.,” and “functional” is abbreviated as “func.”

2.1 Language selection

I aimed to survey a diverse collection of languages. The languages I picked represent different paradigms (imperative, functional), different uses (systems programming, scripting, Web), different type systems (dynamic, static), and different eras. The languages I chose to survey are in Table 1.

This selection was ultimately influenced by personal preference, familiarity, and convenience, but it still manages to capture decent diversity in programming languages.

2.2 Three error handling paradigms

In surveying this selection of languages, I found three overall error handling paradigms, where each language’s approach generally fell into one category. These three categories are Special Return Value, Try-Catch with exceptions, and Type System Approaches.

2.2.1 Special Return Value. Perhaps the most basic way to handle errors is by encoding exceptional status in return values. A function’s documentation can specify that some particular return values indicate error conditions. It is then up to the callers of the function to check the return value to determine whether or not an error has occurred.

For example, the `write` function from the C POSIX library (in `unistd.h`) returns the number of bytes that were written. If an error occurred, it instead returns −1. Callers of the `write` function might ignore this fact and just blindly assume that it succeeds. For the `write` function in particular, there’s an additional consideration, which is that even if it does not encounter an error condition, the function may still write fewer bytes than were requested. Depending on the semantics of each particular program, this may also be considered an error condition. Once again, callers must check the return value if they want to detect this condition.

The Special Return Value pattern is commonplace in C. The C language has limited other built-in error handling approaches (see section 2.3.4), so many standard library functions indicate success or error based on their return value. Of course, C is the oldest language examined in this survey, so it isn’t particularly surprising that it uses this basic error handling approach.

Special Return Value also appears in Go, though in a more structured way than it does in C. Go’s `error` type is an interface with a function `Error` returning a string description of the error. Go
functions can return the \texttt{error} type, or more commonly, a tuple containing some return value and the \texttt{error} type. The error value returned by the function is \texttt{nil} if no error occurred, and non-\texttt{nil} if there was an error. Then, callers of such a function can simply check whether or not the \texttt{error} value in the tuple is \texttt{nil}. For example, see Listing 1.

```go
sqrt, err := Sqrt(x)
if err != nil {
    return 0, err
}
// Else, continue...
```

Listing 1. Example usage of a \texttt{Sqrt} function, which might return an error

Unlike C’s approach, Go’s approach separates the (potential) error value from the useful return value of the function. In C’s scheme, an incorrectly written program might use a returned error value as if it is a valid result. In Go, because the potential error value is a different type and a different tuple member, it is difficult to interpret one as the other.

Still, Go’s model allows for the possibility of neglecting to check the return value for error conditions, because a program could use the result item from the returned tuple and ignore the error. I find it peculiar that, due to the design of Go’s type system, functions that return an error must still return some (possibly invalid) value for the section of the returned tuple that contains some result.

2.2.2 Try-Catch. Another paradigm I found is Try-Catch. This pattern involves wrapping segments of code that could lead to exceptions with Try blocks, and then using Catch blocks to specify how to handle specific types of exceptions that may arise.

For example, consider the Java snippet in Listing 2.

```java
try {
    loadNetworkResource();
    calculateResponse();
    flushCache();
} catch (NetworkException e) {
    System.out.println("Received network exception!");
} catch (CacheException e) {
    // do nothing
}
```

Listing 2. Handling possible exceptions in Java

In this example, the code within the \texttt{try} block calls three methods, some of which might raise an exception. Then, the two \texttt{catch} blocks specify what to do in the case that particular exception types are raised.

Most languages that follow the Try-Catch paradigm allow multiple Catch blocks for different exception types. In addition, most Try-Catch languages also feature a "Finally" block which executes no matter what. The Finally block is generally used to contain code that needs to run regardless of whether an exception was raised in the Try block, such as resource cleanup/deallocation.
The Try-Catch paradigm tends to be accompanied by Object-Oriented exception features, where exceptions are represented as objects that have methods and an inheritance tree. This allows programmers to define handling for an exception type and have it also apply to any child classes of that type.

Try-Catch is used in Java, Python, JavaScript/TypeScript, Ruby, and D. Try-Catch appears in Perl as an experimental feature, though a similar feature appears as part of Perl’s `eval`, which allows the program to detect failure in a block of code.

Languages in the Try-Catch paradigm generally allow exceptions to rise through the call stack until they are caught and handled (or if not handled, until they reach the top level and terminate the program). This means that the source of a caught exception could be arbitrarily far in the call stack from where it is handled. This is a difference between Try-Catch and Special Return Value, because in the latter, an error will be “caught” and handled by the immediate caller of the function that encountered the error (if it is handled at all).

2.2.3 Type System Approaches. In the Type System Approaches pattern, languages use their type system to model and capture ways that actions could fail, and to ensure that possible failures are handled.

The fundamental components of type system error handling are two parametric types, which we’ll call `Option` and `Result` (borrowing these names from Rust).

A value of the `Option<T>` type either contains a value of type T, or nothing at all. For example, an `Option<u32>` might contain nothing, or it might contain some (unsigned 32-bit) integer, such as 42. The `Option` type, in a certain sense, allows a variable to be nullable, something which is allowed for variables of all types in some other languages (for example, Java). See Listing 3 for an example.

A value of the type `Result<T, E>` contains either a successful value of type T, or an error of type E. For instance, the type `Result<i64, String>` will either be a (signed 64-bit) integer in the case of success, or a string representing an error, in the case of failure. This type allows the success or failure of an operation to be captured by the type system, in contrast to the Special Return Value and Try-Catch approaches.

The type systems of languages using this approach distinguish between T and `Option<T>` and `Result<T, E>`, meaning that a value of type `Option<T>` cannot be used as a plain T without first considering the possibility that there is no value at all. In programming languages that have nullability, neglecting to check whether a value is `null` before using it can lead to bugs and sometimes cause errors, such as Java’s `NullPointerException`.

The Type System Approach to error handling is used in Rust, Haskell, and Elm. All three of these languages already have robust and full-featured type systems, which is what allows them to use their type systems for error handling.

There are also a number of languages that do not idiomatically use types like `Option` and `Result`, but have type systems powerful enough that users of the language could implement their own versions of these types and use them in their own code. These languages include TypeScript, D, Java, and Go.

Of the three paradigms, Type System Approaches seem the most robust. By using the type system to explicitly model failure modes, languages with this approach allow static guarantees that possible failures are definitely handled, and missing values will be checked.

2.3 Other Related Features

2.3.1 Stack traces. A stack trace, also sometimes known as a traceback or a backtrace, is a list of the stack of function calls that led to an error. It’s generally displayed with the lines of source code that were executed along the way to arrive at the error.
Stack traces are a common feature in languages using the Try-Catch paradigm, but beyond this they are fairly ubiquitous in today’s popular programming languages, regardless of error handling paradigm. They can be useful during development and debugging because they allow the programmer to diagnose the cause of a particular error and its location in the source code.

In Try-Catch languages, stack traces tend to be attached to exception objects when they are raised. This could theoretically be used at runtime to properly handle the error, but this was not something I came across in my survey.

Stack traces are not central or essential to any of the error handling approaches I looked at, but they are useful for development nonetheless.

2.3.2 Checked Exceptions. Checked exceptions are a Java feature that augments Java’s Try-Catch approach. To raise certain types of exceptions, methods must declare in their type signature that they may raise those errors. Then, callers of those methods must either use Try-Catch to handle the exceptions that could be raised, or declare those exceptions in their own signatures.

Checked exceptions are, in my view, vaguely reminiscent of Type System Approaches to error handling, because they do use the type system to tag potential failures. But checked exceptions fall short in their strictness; not all types of exceptions must be declared, so in particular, checked exceptions do not help with null pointer exceptions and other errors that result from unhandled null values.

2.3.3 Automatically Closing Resources. This pattern refers to language features that ensure a resource is closed when exiting a certain scope. It appears in Python and Java.

Python’s with blocks take a context manager (or more than one) which performs some action (such as resource creation) when the block is entered, and another (such as cleanup) when the block is exited. For example usage, see Listing 4, which uses with to open two files and ensure they are closed.

```python
with open('src.txt') as src, open('dest.txt', 'w') as dest:
    dest.write(src.read())
```

Listing 4. Opening two files in Python and writing the contents of one to the other
If both files are successfully opened, they get assigned to the src and dest variables, respectively. Then, the body of the with block is executed. Whether the body runs to completion or encounters an error, the with block ensures that both files get closed before execution continues.

In Python, custom context managers can be defined to perform arbitrary setup and cleanup actions by taking advantage of the with blocks.

Java’s try-with-resources statement offers this feature through an augmentation to the Try-Catch blocks. It allows try blocks to initialize resources that will then be closed at the end of the statement. Any object that implements a specific interface is supported in these special try blocks. The example following in Listing 5 (from Java documentation) [5] demonstrates this feature.

```java
static String readFirstLineFromFile(String path) throws IOException {
    try (FileReader fr = new FileReader(path);
         BufferedReader br = new BufferedReader(fr)) {
        return br.readLine();
    }
}
```

Listing 5. Example from Java documentation [5] of try-with-resources

In the example, the argument to the try block initializes a FileReader and a BufferedReader. Then, Java guarantees that these resources will be cleaned up when the try block is exited.

I believe that the Automatically Closing Resources pattern helps to create more correct programs. The language can be trusted to guarantee that all necessary cleanup actions take place regardless of the (potentially complicated) control flow. And this feature allows more explicit descriptions of the programmer’s intent, rather than asking the programmer to translate their intent into something like a Try-Catch-Finally.

2.3.4 Setjmp. C has a pair of functions called setjmp and longjmp which allow marking a particular place in a program to return to, and then jumping back to that place along with an integer value. It can be used for rudimentary error handling, but this can also make it hard to properly free resources [2].

2.3.5 Defer. Defer is a feature in Go that allows the programmer to specify a function that should run upon leaving the current scope. Deferred actions are collected on a stack and eventually run in last-in-first-out order. Defer can be used for cleanup actions such as closing files and network connections, or committing database transactions.

An example of this feature from the Go Blog [4] can be found in Listing 6.
func CopyFile(dstName, srcName string) (written int64, err error) {
    src, err := os.Open(srcName)
    if err != nil {
        return
    }
    defer src.Close()

    dst, err := os.Create(dstName)
    if err != nil {
        return
    }
    defer dst.Close()

    return io.Copy(dst, src)
}


Once a file is successfully opened, this code ensures that it will be closed, regardless of how
function execution proceeds or where we exit from.

The Defer feature also appears in D, under the name scope(exit). D’s version goes further, with
scope(failure), which only runs the action if the scope is exited due to an exception being raised,
and scope(success) which only runs the action is the scope is exited not due to an exception.

An alternative to Defer when it comes to cleanup actions is the finally block of a Try-Catch
language, because it similarly allows code to be run at a certain point regardless of how scope was
exited. In contrast to this, Defer allows the cleanup action to be located close to resource allocation
in source code, because the deferred action is not run at the time it is registered. This can help with
code readability, because it keeps resource initialization and cleanup right next to one another.

3 COMPENSATIONS

The 2004 paper Finding and preventing run-time error handling mistakes by Westley Weimer and
George C. Necula [6] proposed a programming language feature it called “compensations.” In
explaining the motivations for this feature, the paper points out that Java programs sometimes
contain “paths along which programs forget to discharge obligations in the presence of run-time
errors.” Weimer and Necula analyzed control flow in Java programs to find such paths.

For example, consider the Java snippet in Listing 7, which uses a hypothetical ServerConnection
class to connect to two remote hosts and transfer data between them.
ServerConnection host1 = new ServerConnection("host1.example");
ServerConnection host2 = new ServerConnection("host2.example");
try {
    String data = host1.receive();
    host2.send(data);
} finally {
    host1.disconnect();
    host2.disconnect();
}

Listing 7. Incorrect usage of finally in an attempt to properly close resources

The programmer intends to ensure that whatever server connections are opened will be disconnected at the end of this snippet.

At first glance, it seems like this snippet is probably correct, since it uses a finally block to make sure that even if some exception is raised during the communication with the servers, both connections will be closed.

But this is not correct. Something could go wrong in the process of connecting to the second host, and that would throw an exception which would cause the connection with the first host to never be closed. In addition, if host1.disconnect() happened to throw an exception, then we would never close the second connection.

Weimer and Necula used their analysis to find and characterize mistakes like these in the projects they examined.

They then propose their idea of compensations. Compensations are closures that accumulate on a stack. When the stack’s compensations are run, they run in last-in-first-out order. The scheme proposed in the paper "allows programmers to link actions with compensations, and guarantees that if an action is taken, the program cannot terminate without executing the associated compensation," [6]. This means that, even in the presence of errors, any necessary resource cleanup will happen.

As an example, the paper gives a method that sets up a database connection and uses compensations to make sure that it will be closed, seen in Listing 8.

```java
public Connection getConnection(CompensationStack S) throws SQLException {
    compensate { /* ... do work ... */ }
    with { this.close(); } using (S);
}
```

Listing 8. A method that takes a CompensationStack, from Weimer and Necula [6]

Listing 9 contains code that calls this method (also from the paper).
Connection cn; PreparedStatement ps; ResultSet rs;
CompensationStack S = new CompensationStack();
try {
    cn = ConnectionFactory.getConnection(S, /* ... */);
    StringBuffer qry = ...; // do some work
    ps = cn.prepareStatement(S, qry.toString());
    rs = ps.executeQuery(S);
    ... // do I/O-related work with rs
} finally {
    S.run();
}


In this example, the various helper methods all take the compensation stack as a parameter, so that they can add their own actions that need to eventually happen. Then, a finally block is used to ensure that the compensations run, whether everything succeeded or an exception occurred.

Weimer and Necula looked at two programs and found that they could be modified to use compensation stacks, and that after doing so the programs handled their resources correctly and were even faster.

3.1 Similarity to Defer

The concept of a compensation — an arbitrary closure of code (often having to do with resource cleanup) that is guaranteed to run at a specific point later than it is defined — should sound familiar. This closely resembles the Defer feature found in Go and D from the language survey (section 2.3.5).

Go was developed a number of years after this paper was published, so although this was merely an academic suggestion at the time, in the intervening years it became a real feature in what would be a mainstream language.

Note that Compensations are not completely identical to Defer. For example, the Compensations proposal allows explicitly passing a compensation stack as a function argument, which allows functions that allocate resources to also push cleanup actions onto the compensation stack. In contrast, the stack that collects deferrals in Go is implicit and based on scope, and cannot be passed as an argument or directly modified.

The implicit deferral stack in Go is arguably simpler, because deferrals that run in a particular scope will be visible in the source code of that scope itself. On the other hand, this seems to allow Compensations to be slightly more powerful, as compensations can be automatically created in more places.

I believe that Defer in Go is easier to learn than Compensations would be, because it is more obvious just from inspecting program code where deferrals are created. I think that Defer is the simplest version of this concept, because it captures the idea of “run this code, later” without any extra concepts to reason about. And by making the compensation stack implicit, Defer also becomes more approachable.

3.2 Comparison to try-with-resources

Compensations accomplish something along the lines of what Java’s try-with-resources statement does (which I called an example of the Automatically Closing Resources pattern in section 2.3.3). The try-with-resources statement, like Compensations, ensures that certain actions run when
exiting current scope. Compensations can be used for resource cleanup just like try-with-resources, but Compensations are far more general.

A compensation can run arbitrary closures, and could be used for actions not related to resource cleanup, as necessary. In contrast, I believe that because try-with-resources is more focused on resource cleanup, it can do that job better. It allows resources to specify their own cleanup, which reduces code duplication and also reduces the possibility of cleanup bugs when compared to Compensations, which involves putting cleanup code in individual closures every time a resource is created.

### 3.3 Implementing Compensations in Ruby

Ruby is a flexible language that can be easily extended with Ruby code. This is commonly used to create domain-specific languages, which introduce new syntax and methods to accomplish a specific task. In addition, Ruby supports arbitrary closures as a language feature. I took advantage of these strengths to implement the Compensations concept in Ruby.

In my implementation, my `new_scope` method takes a block (closure) in which a new compensate method is available. The `compensate` method takes a block which it remembers until the scope is exited, whether by exception or normal execution, and then executes the block.

```ruby
new_scope do
  puts 'setup'
  compensate { puts 'cleanup' }
  intermediate_work
  puts 'finished intermediate work'
end
```

Listing 10. Using my Ruby implementation of Compensations to ensure cleanup code will run

The example in Listing 10 demonstrates how my implementation of compensations in Ruby works. After doing some setup work (demonstrated here by printing text), we put a cleanup action on our compensation stack. Then, we do some intermediate work (that might raise an exception), and finally we print a message about finishing the work. Once we leave this scope, the compensation will be run. When this example is run, the output (assuming `intermediate_work` doesn’t throw an exception) is setup, followed by `finished intermediate work`, followed finally by cleanup. If `intermediate_work` does throw an exception, then we won’t get the output of `finished intermediate work`, but the compensation will still run, so we will still see `cleanup` in the output.

The version I built is straightforward: it collects closures on a stack and runs them in reverse order once scope is exited. It handles cases where exceptions or unusual control flow are present. The extensibility of Ruby made this implementation quite simple.

My Ruby implementation of Compensations (Listing 11) is not identical to the feature as proposed by Weimer and Necula. I chose not to expose the compensation stack to the calling code, and I chose to automatically handle creating the compensation stack and then later running it. This was inspired by Go’s `defer` and D’s `scope` (exit), which both do not allow direct access to such a stack.
class CompensationScope
  def initialize
    @compensations = []
  end

  def compensate(&block)
    @compensations << block
  end

  def run_compensations
    @compensations.reverse_each do |compensation|
      compensation.call
      rescue
    end
  end
end

module Kernel
  def new_scope(&block)
    scope = CompensationScope.new
    begin
      scope.instance_eval &block
    ensure
      scope.run_compensations
    end
  end
end

Listing 11. The full implementation of Compensations in Ruby

4 DISCUSSION
There’s a remarkable diversity of error handling approaches across the languages I surveyed. Each language seems to use an approach that makes the most sense for the language overall — for example, using Type System Approaches requires having a robust type system in the first place. And the approach used by each language can reinforce the language’s strengths by capturing error concepts in a way that makes sense for the language’s overall design.

I suspect that most users of programming languages don’t really take error handling style into account when choosing a language. Although proper error handling is important to all projects (except maybe prototype/toy projects), there are usually more important reasons to choose a particular language than error handling style, such as the type system or the abstractions it offers.

I hypothesize that certain error handling paradigms are more effective and correct, if not in general, then at least for specific types of applications. Personally, I’m partial to Type System Approaches, because I think that the guarantees of a robust type system allow us to write programs that handle all possible error cases and avoid running into unexpected problems. But simultaneously, I’m aware that this strictness might hamper development by requiring explicit handling of errors that, for whatever reason, the programmer does not want to handle.
A feature of Try-Catch that is both a strength and a weakness in my eyes is the fact that it can intercept exceptions that occur at an arbitrarily deep call depth from the actual try block. This can make it very easy to catch and handle errors in large sections of a program all at once, but it can also mean not being able to tailor the recovery to the specific cause of the exception.

In most programming languages, the types of events that errors represent are quite diverse. Some errors, such as a failure to allocate memory, are often unrecoverable and it’s likely desirable for the program to just crash. Other errors, like failing to connect to an analytics server, are technically an erroneous condition but can be ignored without seriously affecting program functionality. In the middle, for errors like failing to read a file, we would generally want to detect and handle these so that the program can continue. When discussing error handling, we are discussing all sorts of errors across this spectrum, and what may be appropriate for one type of error may actually be wrong for another kind. On the whole, language designers don’t seem to differentiate between different kinds of errors when it comes to error handling. Java’s Checked Exceptions buck this trend, because certain types of exceptions are considered unchecked, meaning they don’t have to be declared even if they might occur. These unchecked exceptions are things that could arise almost anywhere, like ArithmeticException and NullPointerException, as well as errors that are essentially unrecoverable, such as SecurityException.

Overall, it is my strong suspicion that error handling can and should be matched with the type of program it is being used in. I doubt that there exists one “best” error handling strategy, and instead believe that different strategies are each most appropriate for different applications. This is a complicated question without a clear answer, but I think that in most cases, the typical applications of a programming language are well matched with its error handling style. For instance, JavaScript’s Try-Catch style is not particularly strict or robust, which is acceptable for most web applications (a common use of JavaScript). On the other hand, Rust programs, for which performance and reliability are often crucial, benefit from its stricter Type System Approaches.

Error handling is a problem with room for new solutions. Having traced the history of Compensations, it is instructional and inspirational to see the path that new error handling ideas take from theory to practice. Especially since error handling should be well matched with program types, there must be new or as-of-yet undiscovered concepts that will make certain types of programs more resilient and reliable. When these new concepts come up, it will take time and refinement to develop them into intuitive and effective language features, as happened when Compensations became Defer.

Of course, not every novel error handling idea turns out to be viable in a mainstream language. One idea I came across was that of Alerts [2]. This idea, while fascinating, has not made it into any mainstream language, as far as I can tell.

5 FUTURE WORK

I am curious to learn not just which error handling features exist in programming languages but how they are used.

Overall, as touched on above, I’m curious about how the different error handling patterns compare to one another when it comes to writing effective programs. If different paradigms encourage handling errors in different ways, or at different distances from the point of failure, then I’d also like to know how this plays into the error handling effectiveness.

5.1 Quantifying error handling in Go

I did preliminary work on a tool that traverses Go ASTs with the goal of examining error handling practices by finding how often a given program handles errors, and how often it passes them up the call stack. I would like to look at some popular open-source Go projects to obtain data about
how frequently they handle errors. In addition, I think it would be fascinating to try to quantify how far from the source errors are handled, in terms of the call stack. I believe this could be done using a graph of control flow to find potential code paths through which errors might rise until eventually being handled, though this would certainly be complicated to develop.

It would also be interesting to try to quantify usage of `defer`. What purposes is it used for? How often does it appear in source code, and in which functions?

### 5.2 Error handling and program correctness

I’d like to see more qualitative research about how different error handling features affect program correctness. This would require human analysis of selected codebases to detect program mistakes around error handling and control flow. As part of this, I would also like to learn more about developers’ perceptions of the various error handling features.

The 2018 paper *How Swift Developers Handle Errors* [3] did some of these things for error handling in Swift. The paper included both codebase static analysis and interviews with ten developers about their experiences. They found that less experienced developers often shied away from error handling, while more advanced developers were sometimes frustrated by limitations of Swift’s error handling features. They did not really consider correctness of the codebases, instead focusing on whether they followed a set of best practices. In their codebase analysis, they found that most codebases did follow at least some best practices, although some of the more complicated recommendations were followed far less often. This paper exemplifies the type of work I think would be valuable to further pursue.

### 5.3 Compensations followup

It would also be interesting to try to use Compensations in some codebase that currently uses a different error handling mechanism. When they were proposed in the 2004 paper, Compensations were conceptual, not actually part of Java. As I said in that section, I was able to implement Compensations in Ruby, and with this implementation I could try to use Compensations in an existing Ruby codebase. This would allow us to qualify how the Compensations feature affects program structure and possibly even correctness. I think this would be interesting to try in other languages as well.

### 6 ACKNOWLEDGEMENTS

Thanks to my Honors advisor, Stephen Checkoway, and to the other Computer Science faculty at Oberlin College.

### REFERENCES


