CanThrow Capabilities

Edit this page on GitHub

This page describes experimental support for exception checking in Scala 3. It is enabled by the language import

```
import language.experimental.saferExceptions
```

The reason for publishing this extension now is to get feedback on its usability. We are working on more advanced type systems that build on the general ideas put forward in the extension. Those type systems have application areas beyond checked exceptions. Exception checking is a useful starting point since exceptions are familiar to all Scala programmers and their current treatment leaves room for improvement.

Why Exceptions?

Exceptions are an ideal mechanism for error handling in many situations. They serve the intended purpose of propagating error conditions with a minimum of boilerplate. They cause zero overhead for the "happy path", which means they are very efficient as long as errors arise infrequently. Exceptions are also debug friendly, since they produce stack traces that can be inspected at the handler site. So one never has to guess where an erroneous condition originated.

Why Not Exceptions?

However, exceptions in current Scala and many other languages are not reflected in the type system. This means that an essential part of the contract of a function - i.e. what exceptions can it produce? - is not statically checked. Most people acknowledge that this is a problem, but that so far the alternative of checked exceptions was just too painful to be considered. A good example are Java checked exceptions, which do the right thing in principle, but are widely regarded as a mistake since they are so difficult to deal with. So far, none of the successor languages that are modeled after Java or that build on the JVM has copied this feature. See for example Anders

Java or that build on the JVM has copied this feature. See for example Anders Hejlsberg's statement on why C# does not have checked exceptions.

The Problem With Java's Checked Exceptions

The main problem with Java's checked exception model is its inflexibility, which is due to lack of polymorphism. Consider for instance the `map` function which is declared on `List[A]` like this:

```
def map[B](f: A => B): List[B]
```

In the Java model, function `f` is not allowed to throw a checked exception. So the following call would be invalid:

```
xs.map(x => if x < limit then x * x else throw LimitExceeded())
```

The only way around this would be to wrap the checked exception `LimitExceeded` in an unchecked `java.lang.RuntimeException` that is caught at the callsite and unwrapped again. Something like this:

```
try
    xs.map(x => if x < limit then x * x else throw Wrapper(LimitExceeded()))
  catch case Wrapper(ex) => throw ex
```

Ugh! No wonder checked exceptions in Java are not very popular.

Monadic Effects

So the dilemma is that exceptions are easy to use only as long as we forget static type checking. This has caused many people working with Scala to abandon exceptions altogether and to use an error monad like `Either` instead. This can work in many situations but is not without its downsides either. It makes code a lot more complicated and harder to refactor. It means one is quickly confronted with the problem how to work with several monads. In general, dealing with one monad at a time in Scala is straightforward but dealing with several monads together is much less pleasant since monads don't compose. A great number of techniques have been proposed, implemented, and promoted to deal with this, from monad transformers, to free monads, to tagless final. But none of these techniques is universally liked; each introduces a complicated DSL that's hard to understand for non-experts, introduces runtime overheads, and makes debugging difficult. In the end, quite a few developers prefer to work instead with a single "super-monad" like `ZIO` that has error propagation built in alongside other aspects. This one-size fits all approach can work very nicely, even though (or is it because?) it represents an all-encompassing framework.

However, a programming language is not a framework; it has to cater also for those applications that do not fit the framework's use cases. So there's still a strong motivation for getting exception checking right.

From Effects To Capabilities

Why does `map` work so poorly with Java's checked exception model? It's because `map`'s signature limits function arguments to not throw checked exceptions. We could try to come up with a more polymorphic formulation of `map`. For instance, it could look like this:

```
def map[B, E](f: A => B throws E): List[B] throws E
```

This assumes a type `A throws E` to indicate computations of type `A` that can throw an exception of type `E`. But in practice the overhead of the additional type parameters makes this approach unappealing as well. Note in particular that we'd have to parameterize *every method* that takes a function argument that way, so the added overhead of declaring all these exception types looks just like a sort of ceremony we would like to avoid.

But there is a way to avoid the ceremony. Instead of concentrating on possible *effects* such as "this code might throw an exception", concentrate on *capabilities* such as "this code needs the capability to throw an exception". From a standpoint of expressiveness this is quite similar. But capabilities can be expressed as parameters whereas traditionally effects are expressed as some addition to result values. It turns out that this can make a big difference!

The `CanThrow` Capability

In the *effects as capabilities* model, an effect is expressed as an (implicit) parameter of a certain type. For exceptions we would expect parameters of type `CanThrow[E]` where `E` stands for the exception that can be thrown. Here is the definition of `CanThrow`:

```
erased class CanThrow[-E <: Exception]
```

This shows another experimental Scala feature: erased definitions. Roughly speaking, values of an erased class do not generate runtime code; they are erased before code generation. This means that all `CanThrow` capabilities are compile-time only artifacts; they do not have a runtime footprint.

Now, if the compiler sees a `throw Exc()` construct where `Exc` is a checked exception, it will check that there is a capability of type `CanThrow[Exc]` that can be summoned as a given. It's a compile-time error if that's not the case.

How can the capability be produced? There are several possibilities:

Most often, the capability is produced by having a using clause `(using CanThrow[Exc])` in some enclosing scope. This roughly corresponds to a `throws` clause in Java. The analogy is even stronger since alongside `CanThrow` there is also the following type alias defined in the `scala` package:

```
infix type A = Int
```

```
infix type $throws[R, +E <: Exception] = CanThrow[E] ?=> R
```

That is, `R $throws E` is a context function type that takes an implicit `CanThrow[E]` parameter and that returns a value of type `R`. What's more, the compiler will translate an infix types with `throws` as the operator to `$throws` applications according to the rules

```
             A throws E  -->  A $throws E
    A throws E₁ | ... | Eᵢ  -->  A $throws E₁ ... $throws Eᵢ
```

Therefore, a method written like this:

```
def m(x: T)(using CanThrow[E]): U
```

can alternatively be expressed like this:

```
def m(x: T): U throws E
```

Also the capability to throw multiple types of exceptions can be expressed in a few ways as shown in the examples below:

```
def m(x: T): U throws E1 | E2
def m(x: T): U throws E1 throws E2
```

```
def m(x: T)(using CanThrow[E1], CanThrow[E2]): U
def m(x: T)(using CanThrow[E1])(using CanThrow[E2]): U
def m(x: T)(using CanThrow[E1]): U throws E2
```

Note 1: A signature like

```
def m(x: T)(using CanThrow[E1 | E2]): U
```

would also allow throwing `E1` or `E2` inside the method's body but might cause problems when someone tried to call this method from another method declaring its `CanThrow` capabilities like in the earlier examples. This is because `CanThrow` has a contravariant type parameter so `CanThrow[E1 | E2]` is a subtype of both `CanThrow[E1]` and `CanThrow[E2]`. Hence the presence of a given instance of `CanThrow[E1 | E2]` in scope satisfies the requirement for `CanThrow[E1]` and `CanThrow[E2]` but given instances of `CanThrow[E1]` and `CanThrow[E2]` cannot be combined to provide and instance of `CanThrow[E1 | E2]`.

Note 2: One should keep in mind that `|` binds its left and right arguments more tightly than `throws` so `A | B throws E1 | E2` means `(A | B) throws (Ex1 | Ex2)`, not `A | (B throws E1) | E2`.

The `CanThrow` / `throws` combo essentially propagates the `CanThrow` requirement outwards. But where are these capabilities created in the first place? That's in the `try` expression. Given a `try` like this:

```
try
  body
catch
  case ex1: Ex1 => handler1
  ...
  case exN: ExN => handlerN
```

the compiler generates an accumulated capability of type `CanThrow[Ex1 | ... | Ex2]` that is available as a given in the scope of `body`. It does this by augmenting the `try` roughly as follows:

```
try
  erased given CanThrow[Ex1 | ... | ExN] = compiletime.erasedValue
  body
catch ...
```

Note that the right-hand side of the synthesized given is `???` (undefined). This is OK since this given is erased; it will not be executed at runtime.

Note 1: The `saferExceptions` feature is designed to work only with checked exceptions. An exception type is *checked* if it is a subtype of `Exception` but not of `RuntimeException`. The signature of `CanThrow` still admits `RuntimeException`s since `RuntimeException` is a proper subtype of its bound, `Exception`. But no capabilities will be generated for `RuntimeException`s. Furthermore, `throws` clauses also may not refer to `RuntimeException`s.

Note 2: To keep things simple, the compiler will currently only generate capabilities for catch clauses of the form

```
case ex: Ex =>
```

where `ex` is an arbitrary variable name (`_` is also allowed), and `Ex` is an arbitrary checked exception type. Constructor patterns such as `Ex( ... )` or patterns with guards are not allowed. The compiler will issue an error if one of these is used to catch a checked exception and `saferExceptions` is enabled.

Example

That's it. Let's see it in action in an example. First, add an import

```
import language.experimental.saferExceptions
```

to enable exception checking. Now, define an exception `LimitExceeded` and a function `f` like this:

```
val limit = 10e9
class LimitExceeded extends Exception
def f(x: Double): Double =
  if x < limit then x * x else throw LimitExceeded()
```

You'll get this error message:

```
  if x < limit then x * x else throw LimitExceeded()
                               ^^^^^^^^^^^^^^^^^^^^^

The capability to throw exception LimitExceeded is missing.
```

The capability can be provided by one of the following:

- Adding a using clause `(using CanThrow[LimitExceeded])` to the definition of the enclosing method
- Adding `throws LimitExceeded` clause after the result type of the enclosing

method
- Wrapping this piece of code with a `try` block that catches `LimitExceeded`

The following import might fix the problem:

```
import unsafeExceptions.canThrowAny
```

As the error message implies, you have to declare that `f` needs the capability to throw a `LimitExceeded` exception. The most concise way to do so is to add a `throws` clause:

```
def f(x: Double): Double throws LimitExceeded =
  if x < limit then x * x else throw LimitExceeded()
```

Now put a call to `f` in a `try` that catches `LimitExceeded`:

```
@main def test(xs: Double*) =
  try println(xs.map(f).sum)
  catch case ex: LimitExceeded => println("too large")
```

Run the program with some inputs:

```
> scala test 1 2 3
14.0
> scala test
0.0
> scala test 1 2 3 100000000000
too large
```

Everything typechecks and works as expected. But wait - we have called `map` without any ceremony! How did that work? Here's how the compiler expands the `test` function:

```
// compiler-generated code
@main def test(xs: Double*) =
  try
    erased given ctl: CanThrow[LimitExceeded] = compiletime.erasedValue
    println(xs.map(x => f(x)(using ctl)).sum)
  catch case ex: LimitExceeded => println("too large")
```

The `CanThrow[LimitExceeded]` capability is passed in a synthesized `using` clause to `f`, since `f` requires it. Then the resulting closure is passed to `map`. The signature of `map` does not have to account for effects. It takes a closure as always, but that closure may refer to capabilities in its free variables. This means that `map` is already effect polymorphic even though we did not change its signature at all. So the takeaway is that the effects as capabilities model naturally provides for effect polymorphism whereas this is something that other approaches struggle with.

Gradual Typing Via Imports

Another advantage is that the model allows a gradual migration from current unchecked exceptions to safer exceptions. Imagine for a moment that `experimental.saferExceptions` is turned on everywhere. There would be lots of code that breaks since functions have not yet been properly annotated with `throws`. But it's easy to create an escape hatch that lets us ignore the breakages for a while: simply add the import

```
import scala.unsafeExceptions.canThrowAny
```

This will provide the `CanThrow` capability for any exception, and thereby allow all throws and all other calls, no matter what the current state of `throws` declarations is. Here's the definition of `canThrowAny`:

```
package scala
object unsafeExceptions:
  given canThrowAny: CanThrow[Exception] = ???
```

Of course, defining a global capability like this amounts to cheating. But the cheating is useful for gradual typing. The import could be used to migrate existing code, or to enable more fluid explorations of code without regard for complete exception safety. At the end of these migrations or explorations the import should be removed.

Scope Of the Extension

To summarize, the extension for safer exception checking consists of the following elements:

- It adds to the standard library the class `scala.CanThrow`, the type `scala.$throws`, and the `scala.unsafeExceptions` object, as they were described above.
- It adds some desugaring rules ro rewrite `throws` types to cascaded `$throws` types.
- It augments the type checking of `throw` by *demanding* a `CanThrow` capability or the thrown exception.

- It augments the type checking of `try` by *providing* `CanThrow` capabilities for every caught exception.

That's all. It's quite remarkable that one can do exception checking in this way without any special additions to the type system. We just need regular givens and context functions. Any runtime overhead is eliminated using `erased`.

Caveats

Our capability model allows to declare and check the thrown exceptions of first-order code. But as it stands, it does not give us enough mechanism to enforce the *absence* of capabilities for arguments to higher-order functions. Consider a variant `pureMap` of `map` that should enforce that its argument does not throw exceptions or have any other effects (maybe because wants to reorder computations transparently). Right now we cannot enforce that since the function argument to `pureMap` can capture arbitrary capabilities in its free variables without them showing up in its type. One possible way to address this would be to introduce a pure function type (maybe written `A → B`). Pure functions are not allowed to close over capabilities. Then `pureMap` could be written like this:

```
def pureMap(f: A -> B): List[B]
```

Another area where the lack of purity requirements shows up is when capabilities escape from bounded scopes. Consider the following function

```
def escaped(xs: Double*): () => Int =
  try () => xs.map(f).sum
  catch case ex: LimitExceeded => -1
```

With the system presented here, this function typechecks, with expansion

```
// compiler-generated code
def escaped(xs: Double*): () => Int =
  try
    given ctl: CanThrow[LimitExceeded] = ???
    () => xs.map(x => f(x)(using ctl)).sum
  catch case ex: LimitExceeded => -1
```

But if you try to call `escaped` like this

```
val g = escaped(1, 2, 1000000000)
g()
```

the result will be a `LimitExceeded` exception thrown at the second line where `g` is called. What's missing is that `try` should enforce that the capabilities it generates do not escape as free variables in the result of its body. It makes sense to describe such scoped effects as *ephemeral capabilities* - they have lifetimes that cannot be extended to delayed code in a lambda.

Outlook

We are working on a new class of type system that supports ephemeral capabilities by tracking the free variables of values. Once that research matures, it will hopefully be possible to augment the Scala language so that we can enforce the missing properties.

And it would have many other applications besides: Exceptions are a special case of *algebraic effects*, which has been a very active research area over the last 20 years and is finding its way into programming languages (e.g. Koka, Eff, Multicore OCaml, Unison). In fact, algebraic effects have been characterized as being equivalent to exceptions with an additional *resume* operation. The techniques developed here for exceptions can probably be generalized to other classes of algebraic effects.

But even without these additional mechanisms, exception checking is already useful as it is. It gives a clear path forward to make code that uses exceptions safer, better documented, and easier to refactor. The only loophole arises for scoped capabilities - here we have to verify manually that these capabilities do not escape. Specifically, a `try` always has to be placed in the same computation stage as the throws that it enables.

Put another way: If the status quo is 0% static checking since 100% is too painful, then an alternative that gives you 95% static checking with great ergonomics looks like a win. And we might still get to 100% in the future.

For more info, see also our paper at the ACM Scala Symposium 2021.

‹ Fewer ... Erased ... ›