Safer Exceptions for Scala

Martin Odersky
Aleksander Boruch-Gruszecki
Jonathan Immanuel Brachthäuser
École polytechnique fédérale de Lausanne
Switzerland
{martin.odersky,aleksander.boruch-gruszecki}@epfl.ch
jonathan.brachthaeuser@uni-tuebingen.de

Edward Lee
Ondřej Lhoták
University of Waterloo
Canada
{e45lee,olhotak}@uwaterloo.ca

Abstract

We describe a scheme for reflecting exceptions as capabilities in the Scala type system that keeps notational overhead to a minimum and avoids well-known problems with Java's checked exceptions framework. The scheme makes exceptions *safer* but not fully *safe* since the capability for throwing an exception may still yet *escape* its enclosing **try** block. To address this limitation, we also propose a type system which prevents capabilities from escaping.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Theory of computation** → **Rewrite systems**.

Keywords: capture, references, exceptions, capabilities, subtyping, types, effects, Scala

ACM Reference Format:
Martin Odersky, Aleksander Boruch-Gruszecki, Jonathan Immanuel Brachthäuser, Edward Lee, and Ondřej Lhoták. 2021. Safer Exceptions for Scala. In *Proceedings of the 12th ACM SIGPLAN International Scala Symposium (SCALA '21), October 17, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3486610.3486893

1 Introduction

Exceptions have been a controversial feature in Scala and other programming languages. On one hand, exceptions are ideal for error handling in many situations; they propagate error conditions with minimum boilerplate code. In many runtimes, including the JVM, they cause zero overhead for the "happy path", which means they are very efficient as long as errors arise infrequently. Exceptions are also easy to debug, since they produce handler-site stack traces; that is, one never has to guess where an erroneous condition originated.

On the other hand, exceptions in Scala and many other languages are problematic since they are not reflected in the type system. This means that an essential part of a function's contract – i.e. what exceptions can it produce? – is not statically checked. This is widely acknowledged to be a problem, but so far the alternative of checked exceptions was shunned because of its bad ergonomics. Java's checked exceptions are a case in point – they 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 are built on the JVM have copied this feature. See for example Anders Hejlsberg's statement [Venners and Eckel 2003] on why C# does not have checked exceptions.

This paper describes a scheme for reflecting exceptions in Scala types that keeps the notational overhead low through scoping and effect polymorphism. The scheme is based on the *effects as implicit capabilities* [Odersky 2015] pattern. It supports effect masking in try expressions through a new type system that tracks variables captured by values.

The paper is organized as follows. Section 2 provides some motivation why a new approach to statically typed exceptions is needed, by discussing the respective disadvantages of checked exceptions in Java on the one hand, and the alternative of monadic effect abstractions on the other hand. Section 3 presents a scheme for tracking thrown exceptions using implicit capabilities. The scheme has been fully implemented and is slated to be released soon in version 3.1 of Scala. The described scheme makes exceptions *safer* than the status quo but not fully *safe* in that it does not guard against capabilities that escape the scope in which they are defined. Section 4 proposes a new type system that addresses this limitation by keeping track of free variables in closures. Section 5 explores the expressiveness and usability of that type system in programming languages like Scala. Section 6 discusses related work. Section 7 concludes and maps out future work.

SCALA '21, October 17, 2021, Chicago, IL, USA
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9113-9/21/10…$15.00
https://doi.org/10.1145/3486610.3486893

2 Motivation

The main problem with Java's checked exception model is the inflexibility caused by the lack of polymorphism. Consider the map method which is declared on List[A]:

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

In the Java model, the 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 LimitExceeded in an unchecked RuntimeException that is caught and unwrapped at the callsite:

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

This greatly complicates and obscures the original call syntax. Generally, monomorphic checked exceptions do not play well with programming patterns involving higher-order functions.

2.1 Polymorphic Exceptions

One can make exception handling polymorphic by allowing exception types as type parameters. In this setting, we could express map as follows:

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

In fact, the parametric approach to effect polymorphism [Lucassen and Gifford 1988] is also supported in Java, but it is rarely used, due to the unappealing overhead of the additional type parameters needed.

Languages like Koka [Leijen 2017] and Frank [Lindley et al. 2017] try to reduce the syntactic overhead by introducing syntactic sugar to hide effect parameters in some cases. This is at best a partial solution since the underlying complexity has a tendency to resurface, for instance when resolving complicated type error messages.

2.2 Monadic Effects

So the dilemma is that exceptions are easy to use only as long as we forgo static type checking. This has caused many Scala developers to abandon exceptions altogether and to use monads [Wadler 1998] instead, specifically an error monad like Try or Either. This approach can often work but is not without its downsides either. It makes code more complicated and harder to refactor. It means one is quickly confronted with the problem how to work with several monads. Dealing with one monad at a time is straightforward but dealing with several monads together is much less pleasant since monads do not compose well. A great number of techniques have been proposed, implemented, and promoted to deal with this, from monad transformers [Liang et al. 1995], to free monads [Sheard and Pasalic 2004], to tagless final [Carette et al. 2009]. None of these techniques is universally liked, however; each introduces a complicated DSL with runtime overhead that is both hard to understand for non-experts and hard to debug. In the end, many developers prefer to work instead in a single "super-monad" like ZIO[1] that has error propagation built-in alongside other aspects. This one-size-fits-all approach can work nicely for frameworks that offer a fixed set of capabilities, but its fixed overhead and lack of flexibility make it unsuitable as the only provided solution for a general purpose programming language.

3 From Effects to Capabilities

In this paper we propose a different approach to polymorphic statically checked exceptions that does away with 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 types. It turns out that this can make a big difference!

3.1 CanThrow as a Capability

In the *effects as capabilities* model, an effect is expressed as an (implicit) parameter of a certain type. For checked 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 makes use of a new (currently experimental) Scala feature: *erased definitions* [Stucki and Odersky 2020]. 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[2], or produce a compile-time error otherwise.

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:

[1]https://zio.dev/

[2]Given instances and using parameters are a new feature of Scala 3.

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

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

$$A \texttt{ throws } E \longrightarrow A \texttt{ \$throws } E$$
$$A \texttt{ throws } E_1 \mid \ldots \mid E_i \longrightarrow A \texttt{ \$throws } E_1 \ \ldots \texttt{ \$throws } E_i$$

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
```

Multiple `CanThrow` capabilities can be combined in a single throws clause. For instance, the method

```
def m2(x: T)
  (using CanThrow[E1], CanThrow[E2]): U
```

can alternatively be expressed like this:

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

Aside: One could have used `throws` without the leading `$` as the infix type, except that that name is already taken as an annotation name in Scala's standard library. So the current solution treats `throws` as a soft keyword instead, and translates to `$throws` when used as an infix type.

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 capabilities for `CanThrow[Ex1]`, ..., `CanThrow[ExN]` that are in scope as givens in body. It does this by augmenting the `try` as follows:

```
try
  given CanThrow[ExN] = ???
  ...
  { given CanThrow[Ex1] = ???
    body
  }
catch ...
```

The nesting of the given capabilities mirrors the precedence of exception catching, where earlier catch clauses shadow later ones.

Note that the right-hand side of all givens is ??? (undefined). This is OK since these givens are erased; they will not be executed at runtime.

Note also that only checked exceptions (in the Java sense) are tracked with capabilities. Unchecked exceptions that derive from `java.lang.RuntimeException` or `java.lang.Error` are not statically represented.

3.2 An Implementation

The presented scheme has been implemented for Scala 3. It is enabled through the language import

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

The implementation consists of the following deltas relative to the standard Scala 3 compiler and library:

1. It adds in the class `CanThrow` and the type `$throws` as they were described above, and the `unsafeExceptions` object described later.
2. It adds the described desugaring rules to rewrite `throws` types to cascaded `$throws` types.
3. It augments the type checking of `throw` by *demanding* a `CanThrow` capability for the thrown exception.
4. It augments the type checking of `try` by *providing* `CanThrow` capabilities for every caught exception.

All additions are either library code or pre-typechecking desugarings. So far, no changes to the type system are needed for exception checking. All that's needed are regular givens and context functions. Any runtime overhead is eliminated by using `erased` types.

3.3 Usage Example

Here is an example program that throws an exception without a matching capability:

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

Compiling this program produces the following 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:
| - A using clause `(using CanThrow[LimitExceeded])`
| - A `throws` clause in a result type
| - an enclosing `try` that catches LimitExceeded
|
|The following import might fix the problem:
|
|  import unsafeExceptions.canThrowAny
```

The message explains 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 let's 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")
```

We can 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. Note that we have called the unmodified map of Scala's standard library without any ceremony. How did that work? Here is how the compiler expands the test function:

```
// compiler-generated code
@main def test(xs: Double*) =
  try
    given ctl: CanThrow[LimitExceeded] = ???
    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.

3.4 Gradual Typing via Imports

Another advantage of the capability model is that it allows a gradual migration from current unchecked exceptions to safer exceptions. Assume for a moment that experimental.saferExceptions is turned on everywhere. Lots of code would break, since functions have not yet been properly annotated with throws. However, there is an easy escape hatch that lets us ignore the breakages for a while by providing globally the CanThrow capability for any exception. Simply import:

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

Here is 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.

It remains an open question whether back-doors like canThrowAny should be better regulated. In the current implementation, erased values of any type, including capability types, can be created by using "undefined" (???) as the right hand side. One could simply decree that such constructs are to be considered as escape hatches that undermine capability safety, just like asInstanceOf is an escape hatch that undermines type safety. An alternative would be to look for ways to control the creation of such escape hatches more tightly.

3.5 Limitations

The effects as capabilities model explored so far allows one 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 pureMap, a variant of map that should enforce that its argument does not throw exceptions or have any other effects (maybe because we want it 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 as a method on List with the following signature:

```
def pureMap[B](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 one tries 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.

Even so, exception checking is arguably already useful as it is. It gives a clear path forward to make exception-using code safer, better documented, and easier to refactor.

The only loophole arises for ephemeral 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.

Nevertheless, it would be great if we could close the loophole. In the rest of this paper, we report on current work that attempts to do this.

4 Tracking Captured Capabilities

We now develop a new type system that supports purity and ephemeral capabilities by tracking the free variables of values. The core idea is to add a form of *capturing type* $\{x_1, \ldots, x_n\}\ T$ which represents the type T that may capture variables $x_1, \ldots, x_n$ in its *capture set*. Capture sets are finite sets of program variables. The type of a lambda abstraction summarizes in its capture set the free variables of the lambda.

The system presented in Figure 1 is dependently typed. In application, the arguments replace the parameters in the capture sets of the result type. Similarly to $D_{<:}$ and DOT [Amin et al. 2016], the terms in the calculus are in ANF.

It may look like such a system will lead to very verbose types. But we will demonstrate that the notational overhead can be kept quite reasonable by using a combination of selective tracking, subtyping, and type inference.

The rest of this section presents the *capture calculus* $\mathsf{CF}_{<:\Box}$ as an extension of System $\mathsf{F}_{<:}$, formulated in ANF. We pick $\mathsf{F}_{<:}$ as a basis, since is a standard, small calculus reflecting the fundamental concepts of subtyping and universal polymorphism. We make crucial use of subtyping to reflect specificity of capture sets. Universal polymorphism causes some interesting challenges, addressed by boxed types. We pick ANF since it directly supports variable dependencies.

4.1 Syntax

The syntax of types consists of the usual $\mathsf{F}_{<:}$ types, plus a capturing type $\{x_1, \ldots, x_i\}\ R$ and a boxed type $\Box\ T$. All type forms except capturing types are classified as *pure types*, ranged over by the letter R. We assume the structural equivalence

$$\{\}\ R \equiv R$$

That is, a pure type can be regarded as a capturing type with an empty capture set. Note that type variable bounds and arguments must be pure types. All other types can have capture sets.

Capture sets are finite sets of variables. We write $C\backslash x$ for the capture set C without the variable x. We assume a special variable $*$ that is not bound in the environment Γ, but can be part of capture sets. $*$ represents the root capability, from which all other capabilities are derived.

Unlike in $\mathsf{F}_{<:}$, functions in $\mathsf{CF}_{<:\Box}$ are dependent - the parameter may occur in the (capture sets of) the result. Therefore function types are written $\forall(x : S)T$ so that the parameter can be named. We retain the syntax $S \rightarrow T$ as a shorthand for the function type $\forall(x : S)T$ where T does not contain x as a free variable. Function types are pure; they only retain variables that are explicitly mentioned in a capture set prefix. Capturing binds more weakly than function type arrows, so $\{x, y\}\ A \rightarrow B$ parses as $\{x, y\}\ (A \rightarrow B)$.

The syntax of values and terms is what one would expect in an ANF version of System $\mathsf{F}_{<:}$, except for boxing and unboxing. $\Box\ x$ is a value that represents x with a boxed type. $C \multimapinv x$ is a term that takes a reference x (referring to a boxed value) and accesses the underlying unboxed value by presenting the capture set of its type. Boxing and unboxing operations are needed as type variables in $\mathsf{CF}_{<:\Box}$ range only over pure types. Boxing is a way to "hide" capture sets, turning a type into a pure type so that a type variable can be instantiated with it. Conversely, unboxing "recovers" the capture set by presenting it explicitly as a key with which the boxed term is opened (the symbol $\multimapinv$ is intentionally chosen to resemble a key). Boxed types and boxing/unboxing operations would usually be inferred, so they do not need to be written explicitly in source (and they can probably be safely elided in error messages as well).

The *captured variables* $\mathrm{cv}(t)$ of a term t are defined as follows:

$$\begin{array}{rcll}
\mathrm{cv}(\lambda(x : S)t) & = & \mathrm{cv}(t)\backslash x & \\
\mathrm{cv}(\lambda[X <: R]t) & = & \mathrm{cv}(t) & \\
\mathrm{cv}(x) & = & \{x\} & \\
\mathrm{cv}(\textbf{let}\ x = v\ \textbf{in}\ t) & = & \mathrm{cv}(t) & \textbf{if}\ x \notin \mathrm{cv}(t) \\
\mathrm{cv}(\textbf{let}\ x = s\ \textbf{in}\ t) & = & \mathrm{cv}(s) \cup \mathrm{cv}(t)\backslash x & \\
\mathrm{cv}(x\ y) & = & \{x, y\} & \\
\mathrm{cv}(x[R]) & = & \{x\} & \\
\mathrm{cv}(\Box\ x) & = & \{\} & \\
\mathrm{cv}(C \multimapinv x) & = & C \cup \{x\} &
\end{array}$$

The captured variables of a term are closely related to its free variables, except for the following three differences:

1. A box operation $\Box\ x$ "forgets" x as a free variable.
2. Dually, an unbox operation $C \multimapinv x$ counts the variables in C as free variables.
3. In an evaluated let binding $\textbf{let}\ x = v\ \textbf{in}\ t$, the free variables of v are counted only if x is a captured variable of t.

The first two rules encapsulate the essence of (un)box pairs: Boxing forgets about free variables in the boxed term, but these need to be presented then instead by the unbox operation that needs to be applied before a boxed value can be accessed.

4.2 Evaluation

Figure 1 defines a small step evaluation relation using two kinds of contexts: *store contexts* σ and *evaluation contexts*

Syntax

Value	v, w	$::=$	$\lambda(x : T)t$	abstraction
		$\mid$	$\lambda[X <: R]t$	type abstraction
		$\mid$	$\Box\, x$	boxing
Answer	a	$::=$	v	
		$\mid$	x	variable
Term	s, t	$::=$	a	
		$\mid$	$x\, y$	application
		$\mid$	$x[R]$	type application
		$\mid$	$C \multimap x$	unboxing
		$\mid$	$\mathbf{let}\, x = s\, \mathbf{in}\, t$	let

Pure Type	R	$::=$	X	type variable
		$\mid$	$\top$	top type
		$\mid$	$\forall(x : S)T$	term function
		$\mid$	$\forall[X <: R]T$	type function
		$\mid$	$\Box\, T$	boxed type
Type	S, T	$::=$	R	
		$\mid$	$C\, R$	capturing type
Capture set	C	$::=$	$\{x_1, \ldots, x_n\}$	
Store context	σ	$::=$	$[\,] \mid \mathbf{let}\, x = v\, \mathbf{in}\, \sigma$	
Eval context	e	$::=$	$\mathbf{let}\, x = [\,]\, \mathbf{in}\, t \mid \mathbf{let}\, x = e\, \mathbf{in}\, t$	

Subcapturing $\boxed{\Gamma \vdash C <: C}$

$$\frac{\Gamma \vdash \{x_1\} <: C \quad \ldots \quad \Gamma \vdash \{x_n\} <: C}{\Gamma \vdash \{x_1, \ldots, x_n\} <: C} \;(\textsc{sc-set}) \qquad \frac{x \in C}{\Gamma \vdash \{x\} <: C} \;(\textsc{sc-elem}) \qquad \frac{\Gamma \vdash x : C\, R}{\Gamma \vdash \{x\} <: C} \;(\textsc{sc-var})$$

Subtyping $\boxed{\Gamma \vdash T <: T}$

$$\Gamma \vdash T <: T \;(\textsc{refl}) \qquad \frac{\Gamma \vdash T_1 <: T_2 \qquad \Gamma \vdash T_2 <: T_3}{\Gamma \vdash T_1 <: T_3} \;(\textsc{trans})$$

$$\frac{X <: T \in \Gamma}{\Gamma \vdash X <: T} \;(\textsc{tvar}) \qquad \Gamma \vdash R <: \top \;(\textsc{top})$$

$$\frac{\Gamma \vdash S_2 <: S_1 \qquad \Gamma, x : S_2 \vdash T_1 <: T_2}{\Gamma \vdash \forall(x : S_1)T_1 <: \forall(x : S_2)T_2} \;(\textsc{fun}) \qquad \frac{\Gamma \vdash R_2 <: R_1 \qquad \Gamma, x : R_2 \vdash T_1 <: T_2}{\Gamma \vdash \forall[X <: R_1]T_1 <: \forall[X <: R_2]T_2} \;(\textsc{tfun})$$

$$\frac{\Gamma \vdash C_1 <: C_2 \qquad \Gamma \vdash R_1 <: R_2}{\Gamma \vdash C_1\, R_1 <: C_2\, R_2} \;(\textsc{capt}) \qquad \frac{\Gamma \vdash T_1 <: T_2}{\Gamma \vdash \Box\, T_1 <: \Box\, T_2} \;(\textsc{boxed})$$

Typing $\boxed{\Gamma \vdash t : T}$

$$\frac{x : C\, R \in \Gamma}{\Gamma \vdash x : \{x\}\, R} \;(\textsc{var}) \qquad \frac{\Gamma \vdash t : S \qquad \Gamma \vdash S <: T \qquad \Gamma \vdash T\ \mathbf{wf}}{\Gamma \vdash t : T} \;(\textsc{sub})$$

$$\frac{\Gamma, x : S \vdash t : T}{\Gamma \vdash \lambda(x : S)t : \mathrm{cv}(t) \backslash x\ \forall(x : S)T} \;(\textsc{abs}) \qquad \frac{\Gamma, X <: R \vdash t : T}{\Gamma \vdash \lambda[X <: R]t : \mathrm{cv}(t)\ \forall[X <: R]T} \;(\textsc{tabs})$$

$$\frac{\Gamma \vdash x : C\ \forall(z : S)T \qquad \Gamma \vdash y : S}{\Gamma \vdash x\, y : [z := y]T} \;(\textsc{app}) \qquad \frac{\Gamma \vdash x : C\ \forall[X <: R]T}{\Gamma \vdash x[R] : [X := R]T} \;(\textsc{tapp})$$

$$\frac{\Gamma \vdash x : C\, R \qquad C \subseteq \mathrm{dom}(\Gamma)}{\Gamma \vdash \Box\, x : \Box\, C\, R} \;(\textsc{box}) \qquad \frac{\Gamma \vdash x : \Box\, C\, R \qquad C \subseteq \mathrm{dom}(\Gamma)}{\Gamma \vdash C \multimap x : C\, R} \;(\textsc{unbox})$$

$$\frac{\Gamma \vdash s : S \qquad \Gamma, x : S \vdash t : T \qquad x \notin \mathrm{fv}(T)}{\Gamma \vdash \mathbf{let}\, x = s\, \mathbf{in}\, t : T} \;(\textsc{let})$$

Evaluation $\boxed{\Gamma \vdash t \longrightarrow t'}$

$\sigma[\, x\, y\,]$	$\longrightarrow$	$\sigma[\, [z := y]t\,]$	**if** $\sigma(x) = \lambda(z : T)t$	($\textsc{apply}$)
$\sigma[\, x[R]\,]$	$\longrightarrow$	$\sigma[\, [X := R]t\,]$	**if** $\sigma(x) = \lambda[X <: R']t$	($\textsc{tapply}$)
$\sigma[\, C \multimap x\,]$	$\longrightarrow$	$\sigma[\, y\,]$	**if** $\sigma(x) = \Box\, y$	($\textsc{unbox}$)
$\sigma[\, e[\, \mathbf{let}\, x = y\, \mathbf{in}\, t\,]\,]$	$\longrightarrow$	$\sigma[\, e[\, [x := y]t\,]\,]$		($\textsc{rename}$)
$\sigma[\, e[\, \mathbf{let}\, x = v\, \mathbf{in}\, t\,]\,]$	$\longrightarrow$	$\sigma[\, \mathbf{let}\, x = v\, \mathbf{in}\, e[\, t\,]\,]$		($\textsc{lift}$)
$\sigma[\, e[\, t\,]\,]$	$\longrightarrow$	$\sigma[\, e[\, t'\,]\,]$	**if** $\sigma[\, t\,] \longrightarrow \sigma[\, t'\,]$	($\textsc{context}$)

Figure 1. System $\mathrm{CF}_{<:\Box}$

e. These are defined as orthogonal decompositions of **let**. We write $\sigma(x) = v$ if the store context σ contains a binding $x = v$.

The first three reduction rules(APLY), (TAPPLY), and (UNBOX) rewrite a toplevel redex in a store context. The next two rules are administrative in nature. Rule (RENAME) eliminates aliases $x = y$ by renaming. Rule (LIFT) lifts evaluated let bindings $x = v$ out into the store context. Finally, rule (CONTEXT) allows reduction anywhere in an inner evaluation context embedded in an outer store context. A straightforward inspection of reduction rules establishes:

Proposition. $\longrightarrow$ is deterministic.

4.3 Type System

As in $\mathsf{F}_{<:}$, the type system of $\mathsf{CF}_{<:\square}$ is defined by well-formedness, subtyping, and type assignment rules. Subtyping uses an auxiliary *subcapturing* relation on capture sets.

4.3.1 Subcapturing. Subcapturing judgments $\Gamma \vdash C_1 <: C_2$ relate two capture sets in an environment. The relation takes variable types into account. E.g. under a variable binding `x: {y} Transform` this relation would validate that `{x} <: {y}`, using rule (SC-VAR).

The motivation is that, since the only variable that instances of `x` could capture is `y`, it makes sense to declare the capture set `{x}` to be more precise than the capture set `{y}`. Note that the reverse is not true, as `x` may be instantiated with values which do not capture anything.

Rule (SC-VAR) also allows pure variables to be dropped from capture sets.

$\{*\}$ is the top capture set in the sense that $\Gamma \vdash C <: \{*\}$ is admissible, provided that Γ satisfies a well-formedness condition. We will prove this formally below in Proposition (Top CaptureSet).

4.3.2 Subtyping. The subtyping relation $\Gamma \vdash T_1 <: T_2$ is lifted directly from $\mathsf{F}_{<:}$, augmented with two additional rules for dealing with boxed types and capture sets. Both rules are natural; rule (CAPT) lifts the natural notion of subcapturing to types, and (BOXED) just extends subtyping to boxed types. Note that $\top$ covers only pure types; $\{*\}\ \top$ is the real top type.

4.3.3 Typing. Again, typing builds upon the standard notion of typing from $\mathsf{F}_{<:}$, here, we illustrate the small differences by highligthing the rules that change.

Rule (VAR) introduces term dependencies via capture sets. If the type of a variable x in the environment consists of a pure type R and a (possibly empty) capture set C, then the type of the variable reference x is $\{x\}\ R$. The capture set C of the declared type is recovered by subsumption. Indeed, subcapturing gives us $\{x\} <: C$ in this case, and therefore we get $\Gamma \vdash x : C\ R$ by subtyping rule (CAPT).

This also holds if the declared type of x is pure. If $x : R \in \Gamma$ then we still have $x : \{x\}\ R$. However, subcapturing gives us $\{x\} <: \{\}$ in this case so we get again $\Gamma \vdash x : R$ by subsumption. The root capability $*$ is needed for bootstrapping. Without it, all variable references would have empty capture sets.

Compared to System $\mathsf{F}_{<:}$, the (ABS) and (TABS) rules augment the abstraction's type with all variables that are free in the abstracted term. Applying subsumption and rule (SC-VAR), we can immediately drop untracked variables from that capture set.

In rule (APP), the result of application is the result type of the function with the parameter z substituted with the argument y. This is the same as function application for dependent function types except that the dependencies are restricted to variable tracking. The capture set C of the function t itself is discarded in an application, c.f. rules (APP) and (TAPP).

Aside. A more conventional version of (TAPP) would be

$$\frac{\Gamma \vdash x : C\ \forall[X <: R']T \qquad \Gamma \vdash R <: R'}{\Gamma \vdash x[R] : [X := R]T} \quad \text{(TAPP')}$$

That formulation is equivalent to (TAPP) in the sense that either rule is derivable from the other, using subsumption and contravariance of type bounds.

4.3.4 Well-Formedness. Well-formedness $\Gamma \vdash T$ **wf** is equivalent to well-formedness in System $\mathsf{F}_{<:}$ in that free variables in types and terms must be defined in the environment, except that capturing types may mention $*$ in their capture sets:

$$\frac{\Gamma \vdash R\ \textbf{wf} \qquad C \subseteq dom(\Gamma) \cup \{*\}}{\Gamma \vdash C\ R\ \textbf{wf}} \quad \text{(CAPT-WF)}$$

4.3.5 Properties.

Definition. (Well-formed Environment) An environment Γ is well-formed, if for every decomposition of Γ as $\Gamma = \Gamma_1, x : T, \Gamma_2$ it holds that $\Gamma_1 \vdash T$ **wf**.

Proposition. (Top CaptureSet) For every capture set C and well-formed environment Γ, if $C \subseteq \mathrm{dom}(\Gamma) \cup \{*\}$ then $\Gamma \vdash C <: \{*\}$.

Proof. By induction on Γ and a distinction by subcapturing cases.

Corollary. (Top Type) For every well-formed environment Γ, if $\Gamma \vdash T$ **wf** then $\Gamma \vdash T <: \{*\}\top$.

The following two main propositions are at present conjectures. Related systems have been shown correct, including similar work by Boruch-Gruszecki et al. [2021] and an ANF extension of System $\mathsf{F}_{<:}$ [Amin et al. 2016].

Proposition. (Preservation) If $\vdash t : T$ and $t \longrightarrow t'$, then $\vdash t' : T$.

Proposition. (Progress). If $\vdash\ t : T$ then either t is of the form $\sigma[a]$ for some store context σ and answer a, or there exists a term t' such that $t \longrightarrow t'$.

4.3.6 Algorithmic Typing. The main issue facing algorithmic typing is the side condition $x \notin fv(T)$ in the typing rule (LET). An algorithm has to *avoid* x in the inferred type of the **let** if it appears in the inferred type of the body of the **let**. This requires widening the type and applying subsumption. In the case of $\mathsf{CF}_{<:\Box}$, widening can be achieved by replacing covariant occurrences of x in T with the capture set of x while replacing contravariant occurrences with the empty set. In a full programming language like Scala we also have to deal with non-variant occurrences which can arise for instance through type abstraction. The Scala 3 compiler already has a sophisticated scheme for avoidance in place that can be re-used for capture sets.

5 Usage Scenarios

We present two usage scenarios that show that the $\mathsf{CF}_{<:\Box}$ typing discipline is expressive and that it leads to reasonably concise programs. The first scenario shows higher-order operations over collections, and the second scenario shows that try expressions can be made fully safe.

Mapping over a Collection. We showcase the expressiveness and usability of $\mathsf{CF}_{<:\Box}$ by looking at map operations. For simplicity we use a continuation encoding of single-element "cells" as the underlying data structure on which the map operations are defined. We could equally well define them on a Böhm-Berarducci encoding [Böhm and Berarducci 1985] of lists, but this would make the example far larger. Most key insights are already demonstrable using single element collections.

We use Scala 3 notation throughout, but we assume that `A => B` is now the type of functions that can capture arbitrary tracked variables and that a new function type `A -> B` represents pure functions that do not capture any tracked variables. That is, => can be thought of being defined in terms of -> like this:

```
infix type => [A, B] = {*} A -> B
```

We consider collections of the following type:

```
type Cell[+T] = [K] -> (T => K) -> K
```

That is, a `Cell` is a polymorphic function that takes a result type `K` and a (possibly side-effecting) continuation of type `T => K` and that returns the result of applying the continuation to the element of the cell. Here's the constructor to create a cell with a given element `x`:

```
def cell[T](x: T): Cell[T] =
  [K] => (k: T => K) => k(x)
```

Note that on the term level we still use => as a constructor for both pure and impure lambdas. No capture annotations are needed on the result type of `cell` because, first, `x` is of type `T` where the type variable `T` is pure, hence `x` is not tracked, and, second, `k` is consumed in the body `k(x)`, so it is not captured either.

Here is an operation to retrieve the element of a cell:

```
def get[T](c: Cell[T]): T = c[T](identity)
```

We can define a strict map operation on cells like this:

```
def map[A, B](c: Cell[A])(f: A => B)
  : Cell[B]
  = c[Cell[B]]((x: A) => cell(f(x)))
```

No capture annotations are needed since `c` and `f` are consumed in the body of `map`.

We can also require that the mapping function is pure, simply by replacing the => in the argument type with `->` :

```
def pureMap[A, B](c: Cell[A])(f: A -> B)
  : Cell[B]
  = c[Cell[B]]((x: A) => cell(f(x)))
```

We now define a lazy variant of `map`. To keep the presentation short, we let `lazyMap` return a closure instead of changing the data structure to support lazy cells.

```
def lazyMap[A, B](c: Cell[A])(f: A => B)
  : {f} () -> Cell[B]
  = () => c[Cell[B]]((x: A) => cell(f(x)))
```

`lazyMap` returns a closure with free variables `c` and `f`. The variable `c` is of the pure type `Cell[A]` and therefore untracked. The variable `f` is of the capturing function type `A => B`, which means that `f` does appear in the capture set of the result type of `lazyMap`.

Here are some examples of how map operations are used. Let's first define a side effecting function that captures a global capability `io` of type `IO`:

```
val loggedOne: {io} () -> Int
  = () => { io.print("1"); 1 }
```

We can encapsulate `loggedOne` in a cell like this:

```
val c: Cell[box {io} () -> Int]
  = cell[box {io} () -> Int](box loggedOne)
```

Note that `loggedOne` needs to be boxed since its type is represented by the type variable `T` of `Cell`, which is instantiated to the boxed type $\Box$ `{io} () -> Int`. Recall that type variables cannot be instantiated with capturing types such as `{io} () -> Int`.

We'd like to map a function over `c` that takes the function `f` stored in `c`, applies it twice and prints the sum of the results. This is written as follows:

```
val g = (f: {io} () -> Int) =>
  val x = f(); io.print("␣+␣")
  val y = f(); io.print(s"␣=␣${x␣+␣y}")

val r = lazyMap[box {io} () -> Int, Unit]
  (c)(box (f => g(unbox {io} f)))
r() // prints 1 + 1 = 2
```

The function value g has type `{io} ({io} () -> Int)) -> Unit`. It takes a function with `io` capability as parameter but also uses `io` independently, so `{io}` appears in two positions in the type.

The example shows that creating data structures with polymorphic type constructors generally does not require capture annotations since elements are of boxed types. On the other hand, access operations over such data structures need unboxing, which requires a capture set. E.g. the function passed to lazyMap has to unbox the element function `f` stored in the cell with the capture set `{io}` as key and has to re-box the result of g.

Boxed types on access operations result in smaller types than capture tracking on construction (which would have to track type variables as well [Boruch-Gruszecki et al. 2021]). While the (un)box operations are a notational burden, they can probably be inferred everywhere by applying the following rules:

- We assume a box on a type with non-empty capture set if the type is an argument to (or a bound of) a type variable.
- We infer a box operation if a value's type is constrained from above by a type variable.
- We infer an unbox operation if a variable's type is constrained from below by a type variable. (Alternatively: if a variable x of boxed type is accessed)

For instance, in the last code line above:

```
lazyMap[box {io} () => Int, Unit] // boxed type arguments assumed
(box (  // box inferred since expression's type <: B in lazyMap
   f => g(
     unbox {io} f // unbox inferred since
   )))            // g's parameter type >: A in lazyMap
```

By inferring box types and (un)box operations, we get

```
lazyMap[{io} () => Int, Unit](c)(g)
```

or, with inferred type arguments

```
lazyMap(c)(g)
```

We are working on a type inferencer that will test this hypothesis. The type inferencer is constructed as a separate pass after regular type checking. It annotates types with inferred capture sets, using a global propagation constraint solver for subcapturing constraints.

Safe Exceptions. The type discipline of $\mathsf{CF}_{<:\square}$ is sufficient for safe exception handling where `CanThrow` capabilities cannot escape in closures. Comparing to Section 3, two modifications are needed in the expansion of `try` expressions: (1) the generated capabilities are variables of type `{*} CanThrow[Ex]` and (2) the body of the `try` is boxed.

For example, take the erroneous definition of `escaped`. Its new compiler generated expansion is:

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

The principal type of the closure returned from try's body is `{ctl} () -> Int`. The variable `ctl` is local to the enclosing block, so it cannot be mentioned in the type of that block. The best possible type of that block is `{*} () -> Int`. But now the box operation is not well-typed since (box) requires that all captured variables in its argument are bound in the environment.

The type discipline enforced by `try` can in its essence also be abstracted in a user-defined function. Here is the signature of a user-defined `_try` function that admits handlers of a single exception of arbitrary type `E`:

```
def _try[E, A]
  (body: (ct: {*} CanThrow[E]) ?=> A)
  (handler: E => A): A =
  try body catch case ex: E => handler(e)
```

The two restrictions enforced by the compiler-generated `try` are implicitly also present in the signature of `_try`: First, the required capability `ct` has type `{*} CanThrow[E]` and is therefore tracked. Second, the body argument of `_try` has a type variable as type, so any non-empty capture set must be boxed. This shows that $\mathsf{CF}_{<:\square}$ can express not only specific handlers but also polymorphic abstractions over them.

6 Related Work

Effects as Capabilities. Marino and Millstein [2009] propose an effect framework that treats effects as sets of capabilities. Following this approach, Liu et al. [2016; 2020] propose to distinguish *stoic* functions, which do not close over capabilities from regular functions.

The problem of how to prevent capabilities from escaping in closures is also addressed by *second-class values* that can only be passed as arguments but not be returned in results or stored in mutable fields. Siek et al. [2012] enforce second-class function arguments using a classical polymorphic effect discipline whereas Osvald et al. [2016] present a specialized type discipline for this task. Second-class values can be encoded in $\mathsf{CF}_{<:\square}$ [Boruch-Gruszecki et al. 2021]. Second-class values are more restrictive than $\mathsf{CF}_{<:\square}$ in that they don't support returning a second-class capability.

Brachthäuser et al. [2020] build on the work by Osvald et al. [2016] to present the *Effekt* language, supporting algebraic effects and handlers. By restricting *all* functions to be second class, Effekt supports a lightweight form of effect polymorphism. Effekt's semantics are given by a translation into explicit capability passing. This is in spirit similar to using context functions that are automatically η-expanded.

Choudhury and Krishnaswami [2020] introduce a type system mechanism to distinguish between pure and impure terms in an impure-by-default calculus. A special "purity" type $\square\ T$ witnesses that the term cannot close over any impure bindings, that is: over potentially effectful resources. Purity types are similar to $\mathsf{CF}_{<:\Box}$'s boxed types. The main difference is that boxed types *can* capture impure variables, but that the captured variables have to be presented again as capabilities when a value of a boxed type is accessed.

Boruch-Gruszecki et al. [2021] present a calculus similar to $\mathsf{CF}_{<:\Box}$, but not restricted to ANF and without boxing. They prove soundness of their calculus, and demonstrate how to extend it to support non-local returns, regions and effect handlers. Unlike $\mathsf{CF}_{<:\Box}$, their calculus cannot support polymorphic data structures containing impure data.

Lightweight Polymorphic Exceptions. Zhang and Myers describe a scheme for abstraction-safe exception handlers [Zhang et al. 2016] and algebraic effects [Zhang and Myers 2019] in which functions are annotated with the set of captured handler values. Handler values can be regarded as special-cased capabilities. By contrast, $\mathsf{CF}_{<:\Box}$ has a more foundational focus in that arbitrary variables can be tracked. Tunneled exceptions as described in their work could be built on top of System $\mathsf{CF}_{<:\Box}$ if `CanThrow` capabilities are passed as regular, non-erased values, using a de-aliasing technique similar to Scala's handling of non-local returns.

Rytz et al. [2013] describe a polymorphic effect system that can express thrown exceptions by annotating results of functions. Annotations can refer to function parameters. This is similar to the way dependent function types are defined in $\mathsf{CF}_{<:\Box}$, except that they track executed effects where $\mathsf{CF}_{<:\Box}$ tracks captured capabilities.

An alternative to source-level type systems for exception tracking is a static analysis such as the one by Pessaux and Leroy [1999] for OCaml. Their analysis consists of a unification-based type inference for a non-standard type system without the need or support of source annotations.

Internal Type Systems. The issue of capturing also comes up in type systems that are used internally for optimization and code generation. Typed closure conversion [Minamide et al. 1996] makes free variables of closures explicit in the types. The most interesting difference between that system and ours is what happens when a bound variable goes out of scope. Typed closure conversion still represents the variable under an existential whereas in our system the type has to be widened so that it does not mention the bound variable at all. The usual way to achieve that is replacing the variable by the set of tracked variables it references. This is also the approach followed by Scherer and Hoffmann [2013] who develop a simply typed system that can track variables in closures for enabling dataflow analyses on curried functions.

7 Conclusion

We presented two contributions that together make it possible to check exceptions statically in Scala. First, we represent effects as implicit capabilities, allowing a low-overhead declaration of exceptions thrown from a block or its result. Second, we introduce a new type system $\mathsf{CF}_{<:\Box}$ to track references of values, which makes it possible to exclude capabilities that escape the scope in which they are defined as parts of closures. We demonstrated the usability of these schemes in program examples. The required annotations are almost surprisingly light-weight, due to several aspects of the system.

Effects as implicit capabilities are concise since (1) implicit parameters can be abstracted in `throws` clauses, (2) throws clauses scope over possibly large bodies of code, so less repetition is needed, and (3) higher-order functions do not need separate variables for thrown exceptions.

The key principles that keep notational overhead for capture tracking low are as follows. First, variables are tracked only if their types have non-empty capture sets. In practice the majority of variables are untracked and thus do not need to be mentioned at all. Second, subcapturing, subtyping and subsumption mean that more detailed capture sets can be subsumed by coarser ones. Finally, boxed types stop propagation of capture information in enclosing types and demand instead the capture set to be presented when a value of a boxed type is accessed. This often reduces overhead, in particular when (un)box operations can be inferred.

Future Work. While exception tracking with implicit capabilities is fully implemented, capture tracking cannot yet be integrated in Scala. Looking first at foundations, it is important to have mechanized proofs for the metatheory of $\mathsf{CF}_{<:\Box}$. It would also be interesting to explore capturing types and boxed types in System $D_{<:}$ and DOT, which can formalize more core features of Scala. Finally, we need to extend rules for variable tracking to other Scala constructs including assignment, classes and traits.

On the implementation side, we need to complete a type inference system for capture sets, study its soundness, and test its ergonomics. Afterwards, we need to work out how to migrate Scala libraries including the standard collection library to support capture tracking. The import escape hatch discussed in Section 2 will surely help, but maybe other techniques will be needed as well.

This work could bring great benefits. Capture tracking has broad applicability - it could also be an essential part of algebraic effect systems, fine-grained resource management, or type systems to control ownership [Clarke et al. 1998; Hogg 1991; Noble et al. 1998], uniqueness [Barendsen and Smetsers 1996], [Haller and Odersky 2010] or interference [Reynolds 1978].

Acknowledgments

This research was supported by the Natural Sciences and Engineering Research Council of Canada.

References

Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The Essence of Dependent Object Types. In *A List of Successes That Can Change the World*. Springer, 249–272. https://doi.org/10.1007/978-3-319-30936-1_14

Erik Barendsen and Sjaak Smetsers. 1996. Uniqueness Typing for Functional Languages with Graph Rewriting Semantics. *Mathematical Structures in Computer Science* 6, 6 (Dec. 1996), 579–612. https://doi.org/10.1017/S0960129500070109

Corrado Böhm and Alessandro Berarducci. 1985. Automatic Synthesis of Typed λ-Programs on Term Algebras. *Theoretical Computer Science* 39 (1985), 135–154. https://doi.org/10.1016/0304-3975(85)90135-5

Aleksander Boruch-Gruszecki, Jonathan Immanuel Brachthäuser, Edward Lee, Ondřej Lhoták, and Martin Odersky. 2021. Tracking Captured Variables in Types. *arXiv:2105.11896 [cs]* (May 2021). arXiv:cs/2105.11896

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as Capabilities: Effect Handlers and Lightweight Effect Polymorphism. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 126 (Nov. 2020). https://doi.org/10.1145/3428194

Jacques Carette, Oleg Kiselyov, and Chung-Chieh Shan. 2009. Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. *Journal of Functional Programming* 19, 5 (Sept. 2009), 509–543. https://doi.org/10.1017/S0956796809007205

Vikraman Choudhury and Neel Krishnaswami. 2020. Recovering Purity with Comonads and Capabilities. *Proc. ACM Program. Lang.* 4, ICFP, Article 111 (Aug. 2020), 28 pages. https://doi.org/10.1145/3408993

David G. Clarke, John M. Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '98)*. Association for Computing Machinery, New York, NY, USA, 48–64. https://doi.org/10.1145/286936.286947

Philipp Haller and Martin Odersky. 2010. Capabilities for Uniqueness and Borrowing. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings (Lecture Notes in Computer Science)*, Theo D'Hondt (Ed.), Vol. 6183. Springer, 354–378. https://doi.org/10.1007/978-3-642-14107-2_17

John Hogg. 1991. Islands: Aliasing Protection in Object-Oriented Languages. In *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*. 271–285. https://doi.org/10.1145/117954.117975

Daan Leijen. 2017. Type directed compilation of row-typed algebraic effects. In *Proceedings of the Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 486–499. https://doi.org/10.1145/3009837.3009872

Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. Association for Computing Machinery, New York, NY, USA, 333–343. https://doi.org/10.1145/199448.199528

Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do Be Do Be Do. *SIGPLAN Not.* 52, 1 (Jan. 2017), 500–514. https://doi.org/10.1145/3093333.3009897

Fengyun Liu. 2016. A Study of Capability-Based Effect Systems. Master's thesis. infoscience.epfl.ch/record/219173

Fengyun Liu, Sandro Stucki, Nada Amin, Paolo Giarruso, and Martin Odersky. 2020. *Disciplined Capabilities.* Technical Report. EPFL. infoscience.epfl.ch/record/273642

J. M. Lucassen and D. K. Gifford. 1988. Polymorphic Effect Systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '88)*. Association for Computing Machinery, New York, NY, USA, 47–57. https://doi.org/10.1145/73560.73564

Daniel Marino and Todd D. Millstein. 2009. A Generic Type-and-Effect System. In *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*, Andrew Kennedy and Amal Ahmed (Eds.). ACM, 39–50. https://doi.org/10.1145/1481861.1481868

Yasuhiko Minamide, Greg Morrisett, and Robert Harper. 1996. Typed Closure Conversion. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*. Association for Computing Machinery, New York, NY, USA, 271–283. https://doi.org/10.1145/237721.237791

James Noble, Jan Vitek, and John Potter. 1998. Flexible Alias Protection. In *ECOOP'98 — Object-Oriented Programming (Lecture Notes in Computer Science)*, Eric Jul (Ed.). Springer, Berlin, Heidelberg, 158–185. https://doi.org/10.1007/BFb0054091

Martin Odersky. 2015. Effects as Implicit Capabilities. Swiss National Fund research project. https://infoscience.epfl.ch/record/287464

Leo Osvald, Grégory M. Essertel, Xilun Wu, Lilliam I. González Alayón, and Tiark Rompf. 2016. Gentrification gone too far? affordable 2nd-class values for fun and (co-)effect. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 234–251. https://doi.org/10.1145/2983990.2984009

François Pessaux and Xavier Leroy. 1999. Type-Based Analysis of Uncaught Exceptions. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, Andrew W. Appel and Alex Aiken (Eds.). ACM, 276–290. https://doi.org/10.1145/292540.292565

John C. Reynolds. 1978. Syntactic Control of Interference. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski (Eds.). ACM Press, 39–46. https://doi.org/10.1145/512760.512766

Lukas Rytz, Nada Amin, and Martin Odersky. 2013. A flow-insensitive, modular effect system for purity. In *Proceedings of the 15th Workshop on Formal Techniques for Java-like Programs, FTfJP 2013, Montpellier, France, July 1, 2013*, Werner Dietl (Ed.). ACM, 4:1–4:7. https://doi.org/10.1145/2489804.2489808

Gabriel Scherer and Jan Hoffmann. 2013. Tracking Data-Flow with Open Closure Types. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 710–726. https://doi.org/10.1007/978-3-319-30936-1_14

Tim Sheard and Emir Pasalic. 2004. Two-Level Types and Parameterized Modules. *Journal of Functional Programming* 14, 5 (Sept. 2004), 547–587. https://doi.org/10.1017/S095679680300488X

Jeremy G. Siek, Michael M. Vitousek, and Jonathan D. Turner. 2012. Effects for Funargs. *CoRR* abs/1201.0023 (2012). arXiv:1201.0023 http://arxiv.org/abs/1201.0023

Nicolas Stucki and Martin Odersky. 2020. Erased Definitions. Scala 3 Language Reference Page. https://dotty.epfl.ch/docs/reference/experimental/erased-defs.html

Bill Venners and Bruce Eckel. 2003. The Trouble With Checked Exceptions - A Conversation with Anders Hejlsberg Part 2. www.artima.com/articles/the-trouble-with-checked-exceptions

Philip Wadler. 1998. The Marriage of Effects and Monads. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998*, Matthias Felleisen, Paul Hudak, and Christian Queinnec (Eds.). ACM, 63–74. https://doi.org/10.1145/289423.289429

Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-safe Effect Handlers via Tunneling. *Proc. ACM Program. Lang.* 3, POPL, Article 5 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290318

Yizhou Zhang, Guido Salvaneschi, Quinn Beightol, Barbara Liskov, and Andrew C. Myers. 2016. Accepting Blame for Safe Tunneled Exceptions. In *Proceedings of the Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA). ACM, New York, NY, USA, 281–295. https://doi.org/10.1145/2980983.2908086