# Second-Class Values for Safer Effect Handlers

Jonas Benn

August 26, 2019

# Second-Class Values for Safer Capabilities

Jonas Benn

August 26, 2019

## Table of contents

http://tiny.cc/dotty-escape

## Introduction

## Why talk about capabilities instead of effect handlers?

- Admittedly not the exciting part of *Scala Effekt*
- But capability-passing makes *Effekt* unsafe
- When capabilities with prompts leak, it's a problem (Brachthäuser and Schuster 2017)

## Programming with (leaking) Capabilities

```scala
trait ReadIO {
  def readLine(): String
}

def readLine(cap: ReadIO): String =
  cap.readLine()

def withStdRead[A](f: ReadIO => A): A =
  f(new ReadIO {
      def readLine(): String =
        scala.io.StdIn.readLine()
    })
```

```scala
trait WriteIO {
 def writeLine(s: String): Unit
}

def writeLine(s: String)(cap: WriteIO): Unit =
 cap.writeLine(s)

def withStdWrite[A](f: WriteIO => A): A =
 f(new WriteIO {
     def writeLine(s: String): Unit =
       println(s)
   })
```

Let’s use this:

```
def welcome(writeCap: WriteIO, readCap: ReadIO) = {
  writeLine("What's your name?")(writeCap)
  val name = readLine(readCap)
  writeLine(name + ", that's a nice
      name!")(writeCap)
}

withStdWrite { writeCap: WriteIO =>
  withStdRead { readCap: ReadIO =>
    welcome(writeCap, readCap)
  }
}
```

Let’s make this more ergonomic

```scala
trait ReadIO {
  def readLine(): String
}

def readLine(implicit cap: ReadIO): String =
    cap.readLine()

def withStdRead[A](f: implicit ReadIO => A): A =
  f(new ReadIO {
      def readLine(): String =
        scala.io.StdIn.readLine()
    })
```

This looks better:

```
def welcome(implicit writeCap: WriteIO
                   , readCap: ReadIO) = {
  writeLine("What's your name?")
  val name = readLine
  writeLine(name + ", that's a nice name!")

}

withStdWrite { implicit writeCap: WriteIO =>
  withStdRead { implicit readCap: ReadIO =>
    welcome
  }
}
```

For comparison:

```
def welcome(writeCap: WriteIO
          , readCap: ReadIO) = {
  writeLine("What's your name?")(writeCap)
  val name = readLine(readCap)
  writeLine(name + ", that's a nice name!")
            (writeCap)
}

withStdWrite { writeCap: WriteIO =>
  withStdRead { readCap: ReadIO =>
    welcome(writeCap, readCap)
  }
}
```

- **Good**
  Implicit capability passing
- **Bad**
  Explicit declaration and naming of capabilities

Let's use context bounds!

More ergonomic type class encoding for Scala:

```scala
trait Showable[T] {
  def show(t: T): String
}
implicit val showInt: Showable[Int]
  = new Showable[Int] {
      def show(i: Int) = i.toString
}

def show[T: Showable](t: T) =
  implicitly[Showable[T]].show(t)
// ... desugar ...
def show[T](t: T)(implicit ev: Showable[T]) =
  implicitly[Showable[T]].show(t)
```

Context bounds for context queries:

```
type GetWriteIO[T] = WriteIO
type GetReadIO = [T] => ReadIO

def welcome[T: GetWriteIO : GetReadIO] = ...
// ... desugars to ...
def welcome [T]
  (implicit ev$1: GetWriteIO[T]
          , ev$2: GetReadIO[T]): Unit = ...
```

Context bounds lack abstraction:

```
def genFeed[M[_]: Monad:
  Logging: UserDatabase:
  ProfileDatabase: RedisCache:
  GeoIPService: AuthService:
  SessionManager: Localization:
  Config: EventQueue: Concurrent:
  Async: MetricsManager]: M[Feed] = ???
```

(From: http://degoes.net/articles/zio-environment)

*Ideally, if we have two methods with the same set of type class constraints, we'd like to be able to create something to represent that set of constraints, and then use it to remove the duplication across the two methods:*

```scala
def method1[F[_]: AllConstraints] = ???

def method2[F[_]: AllConstraints] = ???
```

(From: http://degoes.net/articles/zio-environment)

## Implicit Function Types

## Implicit function types

Introduced by Odersky et al. in “Simplicitly: Foundations and Applications of Implicit Function Types”

```scala
type GetWriteIO[T] = implicit WriteIO => T
type GetReadIO[T] = implicit ReadIO => T

def welcome: GetReadIO[GetWriteIO[Unit]] = body
// ... dealias ...
def welcome:
  implicit ReadIO =>
    implicit WriteIO => T = body
// ... desugar ...
def welcome =
  implicit (ev$1: ReadIO) =>
    implicit (ev$2: WriteIO) => body
```

```
type using[T,Cap] = implicit Cap => T
type and[T,Cap] = implicit Cap => T

def welcome: Unit using WriteIO and ReadIO = ...
// ... desugars to ...
def welcome =
  implicit (ev$1: ReadIO) =>
    implicit (ev$2: WriteIO) => ...
```

```
type using[T,Cap] = implicit Cap => T
type and[T,Cap] = implicit Cap => T

// Power of abstraction
type ReadWriteIO[T] = T using WriteIO and ReadIO

def welcome: ReadWriteIO[Unit] = ...
// ... desugars to ...
def welcome =
  implicit (ev$1: ReadIO) =>
    implicit (ev$2: WriteIO) => ...
```

```
withStdWrite {
  withStdRead {
    welcome(writeCap, readCap)
  }
}
// ... desugar ...
withStdWrite { ev$1: WriteIO =>
  withStdRead { ev$2: ReadIO =>
    welcome(ev$1)(ev$2)
  }
}
```

Can we break our safe API?

```
withStdRead { implicit readCap: ReadIO =>
  // How do we get the capability out of here?
}
def easyRead() = readLine()(???)
```

```
var myReadCap: ReadIO = _

withStdRead { implicit readCap: ReadIO =>
 myReadCap = readCap
}

def easyRead() = readLine(myReadCap)
```

Leakage might also happen accidentally...

```
val greeter = withStdWrite {
  { (name: String) =>
     writeLine("Good day " + name) }
}

> greeter("Jonas")
Good day Jonas
```

## Keeping Capabilities in Check

There are at least three approaches to make capabilities safe:

- **Lightweight monadic regions** (Kiselyov and Shan 2009)
  Uses Rank2Types in combination with a phantom type
- **Stoic functions** (Liu 2016)
  Introduces a *variable-capturing discipline*
- **Second-class values** (Osvald et al. 2016)
  Will be explained in the following

## Second-class values

Introduced by Osvald et al. in “Gentrification Gone Too Far? Affordable 2nd-class Values for Fun and (Co-)Effect”

*Most modern languages have abolished restrictions and admit functions [...] as first-class citizens alongside integers and real numbers. Even conservative languages, like Java and C++, have added closures, albeit with some limitations. But uniformly replacing second-class with first-class constructs is a process not unlike gentrification in urban development, where inexpensive living space is transformed into posh condos in an effort of modernisation, but ultimately leading to an undesirable situation where inexpensive and restricted "second-class" constructs are no longer available.*

Let’s build some social housing in Scala!

## Informal definition

(1) First-class functions may not refer to second-class values through free variables.

(2) All functions must return first-class values.

(3) Every parameter of a function is marked as either first- or second-class.
First-class values can be passed as second-class arguments but not the other way around.

(4) Only first-class values may be stored in object fields or mutable variables

Second-class values in action:
\begin{liveCoding}

\end{liveCoding}

## Formal definition $\lambda^{1/2}$

**Syntax**

$$\begin{array}{lcll}
n & ::= & 1 \mid 2 & \text{1st/2nd class} \\
t & ::= & c \mid x^n \mid \lambda x^n.t \mid t\ t & \text{Terms} \\
v & ::= & c \mid \langle H, \lambda x^n.t \rangle & \text{Values} \\
T & ::= & B \mid T_1^n \to T_2 & \text{Types} \\
\\
G & ::= & \emptyset \mid G, x^n : T & \text{Type Envs} \\
H & ::= & \emptyset \mid H, x^n : v & \text{Value Envs}
\end{array}$$

$$G/H^{[\leq n]} = \{x^m : \_ \in G/H \mid m \leq n\}$$

**Operational Semantics** $\boxed{H \vdash t \Downarrow^n v}$

$$H \vdash c \Downarrow^n c \qquad \text{(ECST)}$$

$$\frac{x^m : v \in H^{[\leq n]}}{H \vdash x \Downarrow^n v} \qquad \text{(EVAR)}$$

## Formal definition $\lambda^{1/2}$

**Operational Semantics** $\boxed{H \vdash t \Downarrow^n v}$

$$H \vdash c \Downarrow^n c \qquad \text{(ECST)}$$

$$\frac{x^m : v \in H^{[\leq n]}}{H \vdash x \Downarrow^n v} \qquad \text{(EVAR)}$$

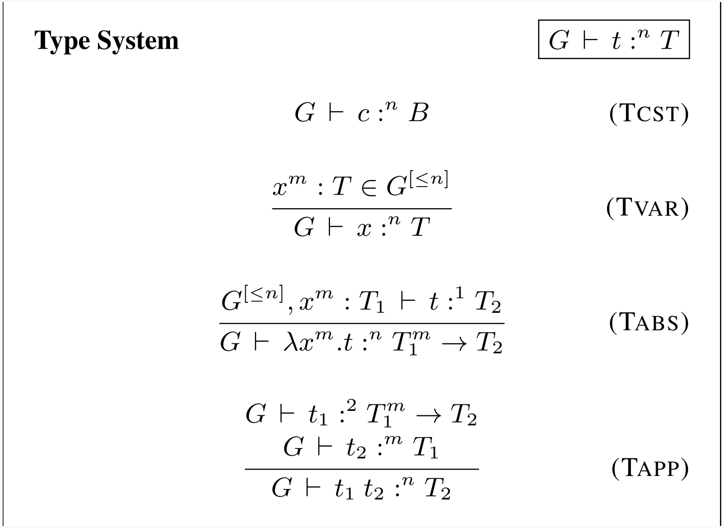
$$H \vdash \lambda x^m.t \ \Downarrow^n \left\langle H^{[\leq n]}, \lambda x^m.t \right\rangle \qquad \text{(EABS)}$$

$$\frac{H \vdash t_1 \Downarrow^2 \langle H', \lambda x^m.t_3 \rangle \qquad H \vdash t_2 \Downarrow^m v_2 \qquad H', x^m : v_2 \vdash t_3 \Downarrow^1 v_3}{H \vdash t_1\ t_2 \Downarrow^n v_3} \qquad \text{(EAPP)}$$

**Type System** $\boxed{G \vdash t :^n T}$

$$G \vdash c :^n B \qquad \text{(TCST)}$$

## Formal definition $\lambda^{1/2}$



**Type System** $\boxed{G \vdash t :^n T}$

$$G \vdash c :^n B \qquad \text{(TCST)}$$

$$\frac{x^m : T \in G^{[\leq n]}}{G \vdash x :^n T} \qquad \text{(TVAR)}$$

$$\frac{G^{[\leq n]}, x^m : T_1 \vdash t :^1 T_2}{G \vdash \lambda x^m . t :^n T_1^m \to T_2} \qquad \text{(TABS)}$$

$$\frac{\begin{array}{c} G \vdash t_1 :^2 T_1^m \to T_2 \\ G \vdash t_2 :^m T_1 \end{array}}{G \vdash t_1\ t_2 :^n T_2} \qquad \text{(TAPP)}$$

**Figure 1:** Osvald et al. 2016, page 237 or page 4

- Second-class values were only available for Scala 2
- Used annotation based system

```
@local val x = ...
def f(@local x: Int, f: Int -> Int) = ...
```

Add support in Dotty for second-class values:

- Modify parser to support new keyword `local`
- Add phase that type checks program according to $\lambda^{1/2}$
- Add support for *local implicit function types*

Combining implicit function types & second-class values:

```
type using[A,Cap] = implicit local A => Cap
type and[T,Cap] = implicit local Cap => T

def f: Int using ReadIO =
  readLine().toInt
// ... dealias ...
def f: implicit local ReadIO => Int =
  readLine().toInt
// ... desugars to ...
def f = implicit local (ev$0: ReadIO) =>
  readLine(ev$0).toInt
```

\begin{liveCoding}

\end{liveCoding}

## Summary

The cake is a lie!

## The root of the problem

- Only 2nd-class functions are allowed to close over 2nd-class values
- But 2nd-class functions must not be returned!

## Making second-class values useful

$\Rightarrow$ Future work?
Return policy has to be less restrictive

$\Rightarrow$ Until then, use the next best thing:
*lightweight monadic regions*

## Slight semantic difference

- **Second-class values**
  Prevent capabilities from escaping their defining scope
- **Lightweight monadic regions**
  Make capabilities unusable outside their defining scope

Probably just relevant for error reporting

## Discussion time

## Bibliography

Brachthäuser, Jonathan Immanuel and Philipp Schuster (2017). "Effekt: Extensible Algebraic Effects in Scala (Short Paper)". In: *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*. SCALA 2017. Vancouver, BC, Canada: ACM, pp. 67–72. ISBN: 978-1-4503-5529-2. DOI: `10.1145/3136000.3136007`. URL: `http://doi.acm.org/10.1145/3136000.3136007`.

Kiselyov, Oleg and Chung-chieh Shan (Jan. 2009). "Lightweight monadic regions". In: *ACM SIGPLAN Notices* 44.2, p. 1. ISSN: 0362-1340. DOI: `10.1145/1543134.1411288`. URL: `http://dx.doi.org/10.1145/1543134.1411288`.

Liu, Fengyun (2016). *A Study of Capability-Based Effect Systems*. Tech. rep.

Odersky, Martin et al. (Dec. 2017). "Simplicitly: Foundations and Applications of Implicit Function Types". In: *Proc. ACM Program. Lang.* 2.POPL, 42:1–42:29. ISSN: 2475-1421. DOI: 10.1145/3158130. URL: http://doi.acm.org/10.1145/3158130.

Osvald, Leo et al. (Oct. 2016). "Gentrification Gone Too Far? Affordable 2nd-class Values for Fun and (Co-)Effect". In: *SIGPLAN Not.* 51.10, pp. 234–251. ISSN: 0362-1340. DOI: 10.1145/3022671.2984009. URL: http://doi.acm.org/10.1145/3022671.2984009.