Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

Bachelor Thesis Computer Science

Second-Class Values for Safer Effect Handlers

Jonas Benn

May 2, 2019

Reviewer

Prof. Dr. Klaus Ostermann
Department of Computer Science
University of Tübingen

Supervisor

Jonathan Immanuel Brachthäuser
Department of Computer Science
University of Tübingen

Benn, Jonas:
Second-Class Values for Safer Effect Handlers
Bachelor Thesis Computer Science
Eberhard Karls Universität Tübingen
Period: Januar 9, 2019 - April 11, 2019

Abstract

In *Effekt: Extensible Algebraic Effects in Scala* Brachthäuser and Schuster (2017) introduce a ergonomic way to program with effect handlers in Scala 3. For this approach they use multi-prompt delimited continuations. The prompts introduced by handlers might escape their defining scope. This might lead to runtime errors for users of Effekt. Brachthäuser & Schuster theorize that the concept of second-class values (Osvald, Essertel, Wu, Alayón, & Rompf, 2016) might make the usage of those prompts save.

By implementing second-class values for Scala 3 this thesis provides an empirical test whether the combination of second-class values and the Effekt library proves to be fruitful.

Abstract (German)

In *Effekt: Extensible Algebraic Effects in Scala* stellen Brachthäuser and Schuster (2017) einen Ansatz vor, wie Programme mit sogenannten *Effect Handlern* in Scala 3 definiert werden können. Der vorgestellter Ansatz basiert vor allem auf *Multi-Prompt delimited continuations*. Die *Prompts* die durch *Handler* bereitgestellt werden, können jedoch aus ihrem Scope entkommen. Dies kann zu Laufzeitfehlern für Benutzer der Bibliothek *Effekt* führen. Brachthäuser & Schuster vermuten, dass *Second-Class Values* (Osvald et al., 2016) hier Abhilfe schaffen könnten.

In dieser Arbeit testen wir empirisch, ob die Kombination aus *Second-Class Values* und der Bibliothek *Effekt* nützlich ist, indem wir *Second-Class Values* für Scala 3 implementieren.

Acknowledgements

First of all, I would like to thank Jonathan Brachthäuser for his wise guidance and for providing the idea for this thesis. Most of all, I want to thank my parents who made this long, exciting odyssey some people call “bachelor” possible.

Contents

1 Introduction

In recent years effectful programming with effect handlers (Plotkin & Pretnar, 2009) gains more and more in popularity. Nowadays there are languages with native support for them available (Bauer & Pretnar, 2012; Leijen, 2016), and already existing languages are getting support for them through libraries (Brady, 2013; Kammar, Lindley, & Oury, 2013; Leijen, 2017).

One example for the latter is Scala. In *Effekt: Extensible Algebraic Effects in Scala* Brachthäuser and Schuster (2017) introduce an embedding for effect handlers which is mainly based on *mult-prompt delimited continuations* (Dybvig, Peyton Jones, & Sabry, 2007). Effekt relies on capabilities not escaping their defining scope. This is especially important since those capabilities carry a reference to the prompt introduced by their corresponding handler. In vanilla Scala this cannot be ensured statically though and capabilities might leak. Brachthäuser and Schuster (2017) theorize that the concept of *second-class values* (Osvald et al., 2016) might fix that.

In the near future Scala 2 will be superseded by Scala 3. Version 3 of the language will be compiled by a new compiler called Dotty (Odersky, 2018b). We will use Dotty and Scala 3 interchangeably in the remainder of this text.

Effekt is available for Dotty. It makes good use of the new feature called *implicit function types* (Odersky et al., 2017):

```
// Effekt without implicit function types
run {
  ambList { implicit ambCap =>
    eitherExc { implicit excCap =>
      weirdCoinFlip(ambCap, excCap)
}}}

// Effekt with implicit function types
run { ambList { eitherExc { weirdCoinFlip }}}
```

second-class values are only available for Scala 2 trough the compiler plug-in `scala-escape`. To get the best of both worlds — second-class values and implicit function types — we port the notion of second-class values for Dotty.

- In the first chapter we provide the background for Effekt, implicit function types and second class values.
- The second chapter contains our main contribution, namely the implementation of second-class values for Dotty. We call this implementation `dotty-escape`. We also discuss open questions regarding the interaction between second-class values and object-oriented concepts present in Scala.
- In the last chapter we combine `dotty-escape`, implicit function types and Effekt for the purpose of creating safe effect handlers.

2 Background

2.1 Effekt

In this section we introduce the library Effekt (Brachthäuser & Schuster, 2017; Brachthäuser, Schuster, & Ostermann, 2019) and in the next section we will see how a feature of Dotty allows for a more ergonomic interface of Effekt.

Effekt makes it possible to write effectful code with effect handlers similar to the notation that Koka offers (Leijen, 2016). The library is using a *multi-prompt delimited continuation monad* (Dybvig et al., 2007), called `Control` and *capability-passing* to encode effect handlers.

When working with effect handlers, users can define effects in a modular way. First they define an effect signature specifying all operations of the effect. In Koka, where effect handlers a supported natively, such a signature looks the following way:

```
effect amb {
  flip() : bool
}
```

In Effekt the signature looks almost the same with the exception that the return type of an operation is wrapped inside the type constructor `Op[_]`. We explain later on why this is needed.

```
trait Amb extends Eff {
  def flip(): Op[Boolean]
}
```

In similar fashion we can define further effects:

```
trait Exc extends Eff {
  def raise[A](s: String): Op[A]
}
```

As shown above, operations are allowed to accept arguments and may use abstract types in their signature — just like normal functions.

After defining an effect's signature, it can already be used:

```
def getRandomNumber(amb: Capability[Amb]): Control[Int] =
  (amb.handler.flip()).flatMap{
    (b: Boolean) => if (b) pure(4) else pure(42)
  }
```

There are two important observations to make here.

First, every time we want to use an operation of the effect signature `E`, we have to provide a handler instance of type `E`. These instances are also called *capabilities*.

Secondly, when using effects, we are programming within the `Control` monad. This is a monad for multi-prompt delimited continuations (Dybvig et al., 2007). This is part of the reason why all return types of operations are wrapped inside `Op[_]`.

Since all effects of Effekt are embedded in the `Control` monad, multiple effects can be composed easily:

```
def weirdCoinFlip(amb: Capability[Amb], exc: Capability[Exc]):
    Control[String] =
  for {
    gravityWorks <- amb.handler.flip()
    isHeads <- if (gravityWorks)
                 /*then*/ amb.handler.flip()
                 else exc.handler.raise("Coin flipping doesn't work
    without gravity")
  } yield if (isHeads) "Heads!" else "Tails!"
```

To run those effectful programs, we have to define handlers corresponding to the effect signatures. In Effekt a handler extends the interfaces of the effect signatures it handles and the `Handler[_/*From*/,_/*Into*/]` trait:

```
trait AmbList[A] extends Amb with Handler[A, List[A]] {

  def unit[A](a: A): List[A] = List(a)

  def flip(): Op[Boolean] =
    (resume: Boolean => Control[List[A]]) =>
      for {
        continueTrue  <- resume(true)
        continueFalse <- resume(false)
      } yield continueTrue ++ continueFalse
}
```

Instances of `AmbList` will handle the operation defined by `Amb`, namely `flip` and also provide in implementation for it. Another thing becomes apparent: `Op[B]` is just an alias for `(B => Control[A]) => Control[A]` which is a value of type `B` written in continuation-passing style or a type raised `B` for readers more familiar with categorical grammar. We can also infer that the continuation delimited by the handler is always of type `Control[A]`.

Every handler can provide its own implementation for effect operations, this makes effectful programming with effect handlers extensible with respect to different interpretations. We can also define another, more trivial, handler for the same effect:

```
trait SoTrue[A] extends Amb with Handler[A, A] {

  def unit[A](a: A): A = a

  def flip(): Op[Boolean] =
    (resume: Boolean => Control[A]) => resume(true)
}
```

The function `unit` makes sure that results of the applied continuation fit the effect domain.

There is still a handler for `Exc` left to be written:

```
trait EitherExc[A] extends Exc with Handler[A, Either[String,A]] {

  def unit[A](a: A): A = Right(a)

  def raise(s: String): Op[Boolean] = resume =>
    pure(Left(s))
}
```

Interpreting effectful code

After defining the effect signatures and defining handlers to handle these operations, we are very close to interpreting the effectful programs we wrote. We still have one problem: our programs are expecting `Capability[Amb]` but we are only able to create instances of `Amb with Handler[_,_]`.

Effekt provides the creation of capabilities by offering the function `handleWith` [1] taking a handler as first and an effectful program as its second argument:

[1] Brachthäuser and Schuster (2017) call this function `handle`

```
def handleWith[From,Into]
  (h: Handler[From,Into])
  (program: Capability[h.type] => Control[From]): Control[Into] = ...
```

`handleWith` will create a capability corresponding to the provided handler and will provide this capability to `program` so it can be interpreted. The result will be a program of type `Control[Into]` where all operations corresponding to the provided capability will be handled. Only `handleWith` can create these capabilities.

After all effects are handled, which means all needed capabilities were provided, it is safe to interpret the program. This can be done with the function `run` which returns the result of type `A` after it interpreted the program of type `Control[A]`:

```
run { handleWith(new SoTrue {}) { ambCap => getRandomNumber(ambCap)
    } }
// > 4
```

We can also define helper functions for more concise handling:

```
def ambList[A](program: Capability[Amb] => Control[A]):
    Control[List[A]] =
  handleWith(new ambList {})(program)
// Analogous for all handlers...
```

Let us observe some example code:

```
run { soTrue { ambCap => getRandomNumber(ambCap) } }
// > 4
run { ambList { ambCap => getRandomNumber(ambCap) } }
// > List(4,42)
run { ambList { ambCap =>
  eitherExc { excCap =>
    weirdCoinFlip(ambCap, excCap)
  }
}}
// > List(Right("Heads!"),Right("Tails!"),Left("Coin flipping
    doesn't work without gravity"))
```

Capabilities not only contain the implementation for effectful operations but they also carry a prompt. This prompt marks a special position on the stack that is maintained by Effekt: it is where `handleWith` was invoked. This delimits the continuation that can be called with the use of `resume`.

Capabilities passing with style

The helper functions defined above already create a more readable syntax but this can be improved further since the passing of capabilities can be automated:

```
def flip(implicit amb: Capability[Amb]) = amb.handler.flip()
// Analogous for other effectful operations

def weirdCoinFlip(implicit amb: Capability[Amb], exc:
    Capability[Exc]): Control[String] =
  for {
    gravityWorks <- flip()
    isHeads <- if (gravityWorks)
                 /*then*/ flip()
                 else raise("Coin flipping doesn't work without
    gravity")
  } yield if (isHeads) "Heads!" else "Tails!"

run { ambList { implicit ambCap =>
  eitherExc { implicit excCap =>
    weirdCoinFlip
  }
}}
```

As we can see declaring Handlers as implicit parameters makes the effectful code much more readable. Nevertheless the user has to write a list of effects used in a function *explicitly* as *implicit parameters*. Also the capabilities created by the handler function also have to be introduced explicitly.

In contrast, this is what the type signature of `weirdCoinFlip` would look like in Koka:

```
fun weirdCoinFlip() : <amb,exc> string { ... }

// we can also alias the usage of multiple effects:
alias ambex = <amb,exc>

fun anotherCoinFlip() : ambex string { ... }
```

Handling of effects is also looking more ergonomic in Koka:

```
ambList { eitherExc { weirdCoinFlip } }
```

In the next chapter we explore a feature that was introduced by Odersky et al. (2017) and will be available with the new Scala compiler *Dotty*. Equipped with this feature Brachthäuser and Schuster (2017) offer a version of Effekt

that looks almost indistinguishable from Koka's syntax.

2.2 Implicit Function Types

In this section we introduce *implicit function types*. We explain why they are a useful concept in general and then proceed to take a closer look how Brachthäuser and Schuster (2017) use it to come close to providing ergonomic effect handlers for the Scala language.

2.2.1 Better abstraction for implicit parameters

The problem with dependency injection can be solved in Scala with implicit types. This is similar to the approach utilizing the *reader monad* in Haskell.

Both approaches build on the widely accepted notion in the functional programming community that better testable and more modular code can be written with the extensive use of simple function parameters. However with a growing number of parameters writing code can get tedious. Especially with parameters that only carry contextual information and are simply passed on to subroutines, one might wish for an abstraction. And indeed there is.

Implicit parameters relieve the programmer of passing parameters *explicitly*. This approach can still become quite costly since every implicit parameter that a function needs has to be explicitly enumerated and named when defining that function. With a growing number of implicit parameters this is still tedious. Also it seems redundant to name parameters that are passed on automatically anyway:

```
def createUser(name: String)
  (implicit dbConfig: DBConfig, vLevel: VerbosityLevel, ...): Unit =
    ...
```

By using the reader monad the programmer is relieved of explicit enumeration of parameters at the call **and** definition site:

```
type MonadConfig m = (MonadReader DBConfig m, MonadReader
   VerbosityLevel m, ...)

createUser :: MonadConfig m => String -> m ()
createUser name = do ...
```

With the introduction of *implicit function types* Odersky et al. (2017) provide

similar ergonomics for implicit parameters:

```
type With[A,Config] = implicit Config => A
type WithAppConfig[A] = Unit With DBConfig With VerbosityLevel With
    ...

def createUser(name: String): WithAppConfig[Int] = ...
```

Implicit function types also have an advantage over the reader monad since the programmer is not forced to write their programs in *monadic style*.

Odersky et al. (2017) also show in their paper that passing parameters with implicit function types is much more performant compared to available implementations of the reader monad in Scala.

But how do implicit function types work?

2.2.2 First taste of implicit function types

The way implicit function types work is quite straight-forward: if the return type of an expression is defined as `implicit A => B` but the inferred type is `B` instead, the expression will be wrapped in an implicit closure that has one parameter of type `A`. Let us consider a concrete example :

```
def f(): implicit Int => Int = implicitly[Int]
// ... desugares to ...
def f(): implicit Int => Int = { implicit ev$n:Int =>
    implicitly(ev$n) }
```

The implicit parameter will be assigned a fresh unique name. As we can see we are able to get hold of this implicit parameter without knowing its name by using `implicitly[DesiredType]`. This implicit function is defined in Dotty's standard library in the following way:

```
@inline final def implicitly[T](implicit ev: T): T = ev
```

So it is basically just an identity function where the compiler inserts the argument automatically.

2.2.3 More Ergonomic Effect Handlers

Brachthäuser and Schuster (2017) combine their library with *implicit function types* in their definitions the following way:

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

The user's code becomes the following:

```
val prog: Boolean using Amb = flip()
// ... desugars to ...
val prog: implicit Cap[Amb] => Boolean = { ev$1: Cap[Amb] =>
  flip()(ev$1)
}
```

The expansion for implicit function types does also work recursively:

```
def prog(x: Int): Boolean using Amb and Exc =
  if(x <= 0) flip() else raise("Panic!")
// ... desugars to ...
def prog(x:Int): implicit Exc => implicit Cap[Amb] => Boolean =
  { ev$1: Cap[Exc] =>
    { ev$2: Cap[Amb] =>
      if(x <= 0) flip()(ev$2) else raise("Panic!""Panic!")(ev$1)
    }
  }
```

2.2.4 Everything in its right place?

With the help of implicit function types, Effekt is an easy to use embedding of effect handlers. So usability is great but what about safety?

As Brachthäuser and Schuster (2017) point out, Effekt relies on capabilities not escaping their defining handler's dynamic scope. Consider:

```
trait Console extends Eff {
  def print(s: String): Op[Unit]
}

def print(s: String): Unit using Console = implicit c =>
  use(c)(c.handler.print(s))

// Prepare the escape
val c: Console = null

run {
  colorfulConsole { implicit cap =>
    c = cap // Leak capability
  }
```

```
}

// What does this do?
run { print("Hello World?")(c) }
```

Interestingly, the user will not be able to use this escaped capability for printing to the console, instead they will encounter a runtime exception. Why is this? Capabilities of Effekt carry the prompt that was introduced by the handler. The way Effekt works is that this prompt disappears from the stack maintained by the library when we are outside the corresponding handler's scope. If an effectful operation, that uses the continuation, is then run outside the handlers scope, Effekt will throw a runtime exception.

Brachthäuser and Schuster (2017) warn about this safety issue but they also hint that there might be a solution at hand, namely *second class values*.

2.3 Second-class values

Second-class values were introduced in *Gentrification gone too far?* (Osvald et al., 2016). The lifetimes of second-class values follow a stack-discipline and these values cannot escape their defining scope. Considering the safety issue we had with Effekt caused by capabilities escaping their defining scope, second-class values and their stack-based lifetimes might prove to be a very useful concept.

Second-class values receive a proper formal definition by Osvald et al. (2016) — the $\lambda^{1/2}$ calculus. For our purposes we can summarize the restrictions of second-class values with the following informal rules where the first two are quoted from Osvald et al., 2016, page 235.

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

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

(3) Every argument 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.

2.3.1 Second-class values in Scala

In Scala we mark second-class value and function definitions with the `local` keyword:

```
local val x: Int = 42
local val f1 = (y: Int) => body
local def f2(y: Int) = body
```

And local parameters are annotated analogously:

```
def f(local x: Int) = body
val anonDef = local (x: Int) => body
```

Every definition that is not annotated with `local` is assumed to be first-class.

Osvald et al. have an additional informal rule regarding Scala:

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

2.3.2 Simple Capabilities

This section tries to provide the reader with a better intuition for second-class values in Scala:

We will define a simple capability `NukeLauncher` with the function `pushTheButton()`.

```
class NukeLauncher {
  def pushTheButton() = ... // Trigger launch sequence
}
```

Now let us try to leak the second-class value through the `trusted` function in various ways.

We will begin with a straight-forward attempt:

```
def trusted(local launcher: NukeLauncher) = {
  return launcher
}
```

This code violates rule (1). Since `launcher` is a second-class value it cannot be returned.

A similarly direct approach would be to store the reference in a variable that is defined in an outer scope or to store it in an object's field:

```
case class Box[A](content: A)
```

```
var x: NukeLauncher = FakeLauncher
def trusted(local launcher: NukeLauncher): Box[NukeLauncher] = {
  x = launcher;
  return new Box(launcher)
}
```

Both actions are disallowed by rule (4).

What if we hid the assignment in another function?

```
var x: NukeLauncher = FakeLauncher

def assignToX(launcher: NukeLauncher) = { x = launcher; }

def trusted(local launcher: NukeLauncher): Box[NukeLauncher] = {
  assignToX(launcher)
}
```

Well, this is not allowed due to rule (3) since we would pass a second-class value as a first-class parameter. Marking the parameter of `assignToX` as second-class would make the assignment to `x` in its body invalid.

Our next approach of smuggling `launcher` out of its defining scope is a bit more subtle:

```
def trusted(local launcher: NukeLauncher) : Int => Unit = {
  def shadyFunction(y: Int): Unit = {
    startCountdown(y);
    launcher.pushTheButton();
  }
  return shadyFunction
}
```

Here a closure is returned by `trusted` and the closure's environment would contain a reference to the launcher. Therefore rule (1) prohibits the first-class closure `shadyFunction` to refer to the second-class value `launcher`.

If we defined `shadyFunction` as second-class instead, its reference to `launcher` would be legal. But then `shadyFunction` could not be returned by `trusted` anymore.

As we can see the rules provided are effective in preventing second-class values from escaping their defining scope.

In this section we have shown how second-class values provide a way to prevent

capabilities from escaping their defining scope. Unfortunately second class values are only available in for Scala 2 and implicit function types only for Dotty. In order to make our safe and ergonomic effect handlers in Dotty a reality, we implemented second-class values for Dotty.

3 Dotty Escape

For the reimplementation of second class values in Dotty, we did not want to reinvent the wheel compared to the original implementation, namely `scala-escape`. Nevertheless, some design decisions were made differently. First we introduce the most obvious difference in our implementation — the syntax — and afterwards we discuss the semantics of `dotty-escape`.

3.1 Syntax

`scala-escape` is implemented as a non-invasive plugin for Scala 2.11. Instead of introducing any new syntax simply annotations are used. They provide the following definitions as a library:

```
class local extends StaticAnnotation with TypeConstraint

trait `->`[-A,+B] extends Function1[A,B] {
  def apply(@local y: A): B
}
```

Those definitions can then be used in the following way:

```
@local val x = 42
def localFun(@local x: Int) = ...

@local implicit val y = 42
def localImplicitFun(@local implicit x: Int) = ...
def higherLocalFun(f: Int -> Int) = ...
```

`Function1[A,B]` is a class that is provided by the standard library and represents functions with one parameter in Scala.

If we would adopt the syntax of `scala-escape` in Dotty, *implicit local function types* would look the following way:

```
type using[A,E] = (implicit Cap[E] @local) => C[A]
```

Instead, we decided to make `local` a keyword similar to `implicit` which leads to the following type signatures:

```
type using[A,E] = implicit local Cap[E] => C[A]
// The order of the keywords does not matter
type using[A,E] = local implicit Cap[E] => C[A]
```

In our introductory chapter we used this keyword-based syntax already so the reader should not be too surprised by this revelation. This syntax is also akin to the combination of `implicit` with `erased` (Stucki, 2018) .

In comparison to the annotation based syntax, our implementation requires changes to the parser and the introduction of `local` as a reserved keyword with a corresponding token.

3.2 Semantics

In order to decide whether the program is well typed with respect to the rules of second-class values, we add an extra phase to the compiler. In this phase we traverse the program's abstract syntax tree once. For every violation of our rules an error gets emitted, informing the user which part of their program is invalid.

The way we go about this is quite close to the implementation of `scala-escape` whose source-code is luckily open source. Unfortunately its design decisions are fairly undocumented. Therefore we will provide a rough overview ourselves. The implementation in figure 3.1 on page 18 is a simplified version of the abstract syntax tree traverser that we implemented. In the following we describe the reasoning behind the code.

3.2.1 How to type check

Osvald et al. (2016) hint at the fact that the implementation of the compiler-plugin is a little different from the formal model because Scala 2 makes heavy use of a symbol table and does not commonly maintain a list of the current environment while traversing the program's AST. The same is true for Dotty.

So instead we keep track of two pieces of information while traversing:

1. `localMode`

   This value tells us whether we are currently in a first- or second-class position. Considering expressions of lambda calculus only, first-class positions are:

   - the expression on the right side of first-class function application and
   - the expression that is the body of a lambda abstraction

   The notion of positions is modeled after $:^1$ and $:^2$ in $\lambda^{1/2}$ (Osvald et al., 2016, page 238). In this model a function is typed as first-class when it is in a first-class position.

2. `enclosingFunctions`

   In the type system rule (TAbs) of $\lambda^{1/2}$, the environment gets immediately filtered depending on a closure being typed as first- or second-class. Since we do not keep track of such a flat environment, a different strategy is needed.

   Before we type-check the body of a lambda abstraction, we add the symbol of this function to a list which keeps track of all the enclosing functions, hence the name `EnclosingFunctions`.

   When a variable is encountered later on, we check whether it is a bound variable inside all the first-class functions in this list. In the case of the variable being free and second-class, we report a violation of the type system.

3.2.2 Implicit local function types

As we have shown, a combination of implicit function types and second-class values is available — implicit local function types. We added support for automatic closure generation analogous to ordinary implicit function types:

```
type using[A,E] = implicit local Cap[E] => C[A]

val stayOrGo: String using Amb =
  flip().flatMap {
    b => if (b) pure("Stay") else pure("Go")
  }
```

```
def traverse( tree: Ast
            , localMode: LocalMode
            , enclosingFunctions: List[Symbols]
            ): List[Errors] =
  tree match {
    // Function Application
    case Apply(fun, arg) => {
      val errorsFun = traverse(fun, SecondClass, enclosingFunctions)
      val errorsArg = traverse(arg, FirstClass,  enclosingFunctions)

      return errorsLeft ++ errorsRight
    }
    // Lambda Abstraction / Anonymous Function
    case Lambda(_, body) => {
      val errors =
        traverse(body,
                 FirstClass,
                 tree.symbol :: enclosingFunctions)
      return errors
    }
    // Variable / Identifier
    case Ident(s) => {

      if (s.isSecondClass && localMode == FirstClass) {

          return Error("Second-class variable in first-class
   position")

      } else if (s.isSecondClass && localMode == SecondClass) {

          for(enclosingFunction <- enclosingFunctions) {
            if (s.isDefinedIn(enclosingFunction))
              return List() // No error emitted
            else
              return Error("Second-class variable captured by
   first-class closure")
          }

      } else if (s.isFirstClass)

        return List() // No error emitted
      }
  }
```

Figure 3.1: Simplified version of the traverser

```
// ... desugars to ...
val stayOrGo: implicit local Cap[Amb] => C[String] =
  implicit local ev$1 =>
    flip()(ev$1).flatMap {
      b => if (b) pure("Stay") else pure("Go")
    }
```

3.3 Open design decisions regarding OOP

So far we have discussed the treatment of constructions that constitute lambda calculus but the Scala language is richer than this and while traversing a program's AST we encounter a variety of nodes. Osvald et al. (2016) provide an extended version of the DOT-calculus but they admit that it is still an open question how to treat the concept of classes and objects in regard to second-class values.

Like the original implementation, ours is still in its infancy when it comes to the treatment of OOP constructs. One way to go about this is to treat OOP constructs extremely restrictive. But even than, one has to be carefully consider what restrictions should be put in place. That is why we think that more work should be done to extend the concept of second-class values to cover constructs of OOP in a principled way. In this section we provide some thoughts on which challenges have to be considered.

Let us start with a low-hanging fruit that was already picked by Osvald et al. (2016, page 244):

> Class definitions are treated like first-class functions and cannot access second-class values from their defining scope. The following code is thus illegal,
>
> ```
> local val log = ...
> class Handler {
>   def func() = log.println("A") // error
> }
> val a = new Handler; a.func()
> ```
>
> but the same functionality can be implemented like this:
>
> ```
> @local val log = ...
> class Handler {
> ```

```
  def func(@local val log: File) = log.println("A")
}
val a = new Handler; a.func(log)
```

There is at least one more problem though when it comes to the interplay of OOP and second-class values.

3.3.1 *this* is a problem

In general we would like to allow every class to have first- and second-class instances. Unfortunately, there is a simple case where Scala's classes conflict with second-class values:

```
class ConsiderThis {
  def returnMyself() = this
  def captureMyself() = () => this
}
```

Every second-class instance of `ConsiderThis` would be invalid since `returnMyself` would return a second-class value and `captureMyself` would return a first-class closure capturing a second-class value. An extremely conservative implementation would type `this` always as second-class. Contrarily, a more pragmatic approach would be to treat only specially annotated classes this rigorously and allow second-class instances of these classes exclusively. Though one has to be cautious: not only the implementation of the marked class and its children have to be checked but also the implementations they inherit. This might be tricky to do considering separate compilation.

Fortunately, there is precedence for something like this in Scala: *value classes* (Odersky, Olson, Phillips, & Suereth, 2012) are classes that have exactly one field and are represented *unboxed* at runtime in most cases. They are used as cheap wrapper classes and in that sense they are a similar to *newtype* in Haskell. The definitions of value classes have to obey a set of restrictions, similar to our second-class class definitions. To make checking of those restrictions possible, value classes can only extend so called *universal traits* and have to extend explicitly the class `AnyVal`. `AnyVal` represents primitive values and is opposed to `AnyRef` which all user-defined classes and traits extend if they don't extend `Any` or `AnyVal` explicitly.

A design similar to this could be used for second-class class definitions:

```
class ConsiderThis extends AnySecondClassVal {
  def returnMyself() = this // error: this is typed as second class
    and cannot be returned
  def captureMyself() = () => this // error: ...
}
```

If one takes a deeper dive into the semantics of Scala's class system, some other issues in addition to the one with `this` might come to light, too. Conceptually there seems to be already some dissonance considering ordinary objects and second-class values since the latter obey stack-discipline and objects are heap-allocated entities whose lifetime is determined by the garbage collector. We think there are some shared goals when we compare second-class values, value classes and the newly introduced *opaque type aliases* (Odersky, 2018a) which are closely related to value classes.

3.3.2 Capabilities as Second-Class Values

For the sake of the argument, let us assume that above design decisions have been settled and we had a sound implementation of second-class values in Dotty available. Can we then use the combination of second-class values and implicit function types for a safe and ergonomic implementation of Effekt? We will discuss that in our last chapter.

4 Will it blend?

In this chapter we will shed light on the question whether the combination of Effekt, implicit function types and second-class values results in an ergonomic and safe library for effect handlers.

4.1 Safe capabilities

In this section we will see that the concept of second-class values has a severe limitation that makes it unusable for our purposes. To make it easier to see what the root cause of this problem is, we will consider only simple capabilities augmented with implicit function types in an analogous manner to Effekt in the last chapter.

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

trait ReadIO {
  def readLine(): String =
    scala.io.StdIn.readLine()
}
object ReadIO {
  def readLine(): String using IO =
    implicit local cap => cap.readLine(s)
}

def withReadIO[A](f: A using ReadIO): A = f(new ReadIO {})

import ReadIO.readLine

trait Console {
  def print(s: String): String =
    print(s)
}
object Console {
  def print(s: String): Unit using Console =
    implicit local cap => cap.print(s)
  }

def withConsole[A](f: A using Console): A = f(new Console {})
```

Using the combination makes the user's code more concise:

```
def readLines(): Stream[String] using ReadIO =
  readLine() #:: readLines()
// ... desugares to ...
def readLines(): implicit local ReadIO => Stream[String] =
 { implicit local (c: ReadIO) => readLine()(c) #:: readLines()(c) }
```

This is exactly what we have wanted and for one capability this works as well as the version without implicit function types.

Now we will consider a function that uses two capabilities:

```
def readLinesVerbose(): Stream[String] using ReadIO and Console = {
  Console.print("Read a line!")
  readLine() #:: readLines()
}
```

When we type-check this code, our version of Dotty will report the following error:

```
2 |  Console.print("Read a line!")
  |                               ^
  |          Found local value evidence$1 inside first class method
   $anonfun.
```

What does this mean? Let us have a look at the desugared code:

```
def readLinesVerbose(): implicit local Console => implicit local
    ReadIO => Stream[String] =
  { implicit local ev$1: Console =>
    { implicit local ev$2: ReadIO
        Console.print("Read a line!")(ev$1)
        readLine()(ev$2) #:: readLines()(ev$2)
    }
  }
```

The compiler wrapped two implicit local functions around the original body of the function, one for each capability. This is also in line with the unaliased type. Unfortunately, using the `Console` capability `ev$1` inside the inner closure is illegal. This is the case because the inner closure is returned after the first capability is applied. Therefore the inner function is first class ergo it is not allowed to close over the second-class value `ev$1`.

This leads to a far-reaching limitation of second-class values: second-class

values can only be supplied in the last parameter list of a function. This leads to another far more serious problem: the concept of currying does not work with second-class values. Consider a function `f1(local x: Int, y: Int) = x + y` where the curried version would be `f2(local x: Int)(local y: Int) = x + y`. Whereas `f1` is a completely valid function definition, `f2` is **invalid**. And rightly so, otherwise the following would be possible:

```
{
  local val i = 41
  def f2 = local x => local y => x + y
  val gotcha: local Int => Int = f2(i) // i captured in function
    gotcha
  return gotcha // i escapes the scope through gotcha
}
```

Note, while `gotcha` is a function that takes a second-class value as a parameter, it is itself first-class. Therefore the above code is clearly illegal.

One might argue the above code would not be problematic if `gotcha` was to be typed as second-class. Then `gotcha` could not escape the defining scope of `i`.

```
{
  local val i = 41
  def f3 = local x => {
    return { local y => x + y }
  }
  local val gotcha: local Int => Int = f3(i) // i captured in
    function gotcha
  return gotcha // but gotcha is not allowed to be returned this time
}
```

For that to be possible, one might propose an extension of $\lambda^{1/2}$ where second-class closures are allowed to be returned. Currying would then be possible: after partial application of `f3` to `i`, `local y => i + y` would be returned and typed as second-class. But this leads to the question how to differentiate the well behaved implementation of `f3` from the one of `f4`:

```
def f4 = local x => {
  local val innerValue = 43
  return { local y => innerValue } // innerValue escapes through
    closure
}
```

As before, the inner function closes over a second-class value but this time it should not be allowed to do so — otherwise `innerValue` would escape its

defining scope.

We see no straight-forward way to extend second-class values that would remove the currying-limitation. Osvald et al. also developed a second-class value calculus with a *privilege lattice* that has some interesting properties. But even with this extension, we do not see how this limitation can be overcome.

4.2 Another path forward

Unfortunately, the missing support for currying makes second-class values not usable in combination with implicit function types for automatic capability passing.

Stoic functions (Liu, 2016) are another approach to keep capabilities from escaping their scope. At the moment it seems unclear though whether something like stoic functions will be added to Dotty.

Luckily Scala supports the next best thing already: *Lightweight Monadic Regions* (Kiselyov & Shan, 2009). Unlike second-class values, the approach of Kiselyov and Shan does **not** prevent capabilities from leaking. Instead it makes them unusable outside their defining scope using phantom and rank-2 types. Albeit this might not be perfect for error reporting, it is sufficient to make the usage of capabilities safe.

The version of Effekt presented in this thesis is similar to the one introduced in *Effekt: Extensible Algebraic effects in Scala* (Brachthäuser & Schuster, 2017). There is also a more recent and slightly different version of the library presented in *Effekt: Type- and Effect-safe, Extensible Effect Handlers in Scala* (Brachthäuser et al., 2019). This version utilizes a variation of lightweight monadic regions — instead of rank-2 types, singleton types are used. With this and intersection types, they are able to do what we, equipped with second-class values, were not: implement safe and ergonomic effect handlers for Dotty.

Bibliography

Bauer, A., & Pretnar, M. (2012). Programming with algebraic effects and handlers. *arXiv preprint arXiv:1203.1539*.

Brachthäuser, J. I., & Schuster, P. (2017). Effekt: Extensible algebraic effects in scala (short paper). In *Proceedings of the 8th acm sigplan international symposium on scala* (pp. 67–72). SCALA 2017. doi:10.1145/3136000.3136007

Brachthäuser, J. I., Schuster, P., & Ostermann, K. (2019). *Effekt: Type- and effect-safe, extensible effect handlers in scala*.

Brady, E. (2013). Programming and reasoning with algebraic effects and dependent types. *SIGPLAN Not. 48*(9), 133–144. doi:10.1145/2544174.2500581

Dybvig, R. K., Peyton Jones, S., & Sabry, A. (2007). A monadic framework for delimited continuations. *J. Funct. Program. 17*(6), 687–730. doi:10.1017/S0956796807006259

Kammar, O., Lindley, S., & Oury, N. (2013). Handlers in action. *SIGPLAN Not. 48*(9), 145–158. doi:10.1145/2544174.2500590

Kiselyov, O., & Shan, C.-c. (2009). Lightweight monadic regions. *ACM SIGPLAN Notices*, *44*(2), 1. doi:10.1145/1543134.1411288

Leijen, D. (2016). *Algebraic effects for functional programming*. Retrieved from https://www.microsoft.com/en-us/research/publication/algebraic-effects-for-functional-programming/

Leijen, D. (2017). *Implementing algebraic effects in c*. Retrieved from https://www.microsoft.com/en-us/research/publication/implementing-algebraic-effects-c/

Liu, F. (2016). *A study of capability-based effect systems*.

Odersky, M. (2018a). Opaque type aliases. Retrieved April 6, 2019, from https://github.com/lampepfl/dotty/blob/c8a5cd93b1245d6702e4c2c695f759e2c315c52d/docs/docs/reference/opaques.md

Odersky, M. (2018b). Towards scala 3.

Odersky, M., Blanvillain, O., Liu, F., Biboudis, A., Miller, H., & Stucki, S. (2017). Simplicitly: Foundations and applications of implicit function types. *Proc. ACM Program. Lang. 2*(POPL), 42:1–42:29. doi:10.1145/3158130

Odersky, M., Olson, J., Phillips, P., & Suereth, J. (2012). Sip-15 - value classes. Retrieved April 6, 2019, from https://docs.scala-lang.org/sips/completed/value-classes.html

Osvald, L., Essertel, G., Wu, X., Alayón, L. I. G., & Rompf, T. (2016). Gentrification gone too far? affordable 2nd-class values for fun and (co-)effect. *SIGPLAN Not. 51*(10), 234–251. doi:10.1145/3022671.2984009

Plotkin, G., & Pretnar, M. (2009). Handlers of algebraic effects. In *Proceedings of the 18th european symposium on programming languages and systems: Held as part of the joint european conferences on theory and practice of software, etaps 2009* (pp. 80–94). ESOP '09. doi:10.1007/978-3-642-00590-9_7

Stucki, N. (2018). Erased terms. Retrieved April 6, 2019, from https://github.com/lampepfl/dotty/blob/5e1fafd319f6bbc2e192fa6ceb6478a2380ea461/docs/docs/reference/erased-terms.md

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Bachelorarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Ort, Datum Unterschrift