Semester project report
Capture checking examples

Simon Hayoz
s.hayoz@epfl.ch

Supervisor: Anatolii Kmetiuk
Professor: Martin Odersky
Programming Methods Laboratory LAMP, EPFL
Lausanne, Switzerland

***Abstract*—This project adapts Scala examples to the capture checking system to show use-cases, test the system, show some general ideas about how to port a Scala project to the capture checking system, and finally show some bugs and limitations of this system.**

I. Introduction

Capture checking is a research project modifying the Scala type system to track references to capabilities in values introduced in Odersky et al. [1]. It can be a key part of the solutions to many long-standing problems in programming languages, such as:

- How to do a safe try-with-resources pattern?
- How to have a simple and flexible system for checked exceptions?
- How to solve the "what color is your function?" problem of mixing synchronous and asynchronous computations?
- How to do region-based allocation, safely?
- How to reason about capabilities associated with memory locations?

Below is an example of how such system can help to create safer try-with-resources patterns. First, the following function using a `FileOutputStream` is defined:

```
def usingLogFile[T](op: FileOutputStream => T): T =
  val logFile = FileOutputStream("log")
  val result = op(logFile)
  logFile.close()
  result
```

Then without capture checking, the following will crash on runtime:

```
val later = usingLogFile { file => () =>
    file.write(0) }
later() // crash
```

This runtime crash can be prevented by using capture checking and redefining the signature of the function above as follows:

```
def usingLogFile[T](op: ({*} FileOutputStream) =>
    T): T =
  // ...
```

Then the runtime crash code will now fail on compilation with the following error:

```
|  val later = usingLogFile { f => () => f.write(0) }
|              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
|The expression's type {*} () -> Unit is not
    allowed to capture the root capability `*`.
|This usually means that a capability persists
    longer than its allowed lifetime.
```

This is the intended behavior as the `FileOutputStream` should indeed not be used after being closed.

A capability in the capture checking system is syntactically a method- or class-parameter, a local variable, or the **`this`** of an enclosing class. The type of a capability must be a capturing set with a non-empty capture set. The most general capability from which all others are derived is written $*$ and is called the universal capability. Capabilities can only be used when they are explicitly defined in the type definition. This behavior is enforced by the compiler.

A safer exceptions system can then be created as well by using the `CanThrow` capability for a clean and fully safe system for checked exceptions in Scala.

This project aims at adapting some Scala examples to the capture checking system to show use-cases, test the system, show some general ideas about how to port a Scala project to the capture checking system, and finally show some bugs and limitations of this system. All the examples will adapt in some way a pattern for an object with open/close type of elements. This will ensure that with the use of capability, the use of these objects is limited to predefined places (i.e. the objects have a lifetime) and enforced on compile time.

II. Examples

All presented examples are adapted from the Hands-on Scala book by Li Haoyi [2]. This book serves as an introduction to the Scala language, mainly through examples. It brings up interesting cases to be tested and ported to the capture checking system. The following type of patterns are treated as examples below: SQL connections, HTTP requests, WebSocket connections, etc.

To complete these examples, most of the collections have been replaced by the *CollectionStrawManCC5_1* [3] which is a collection with capture checking.

All adapted examples have been gathered in a single repository [4].

A. External links

The example, adapted from [5], can be found in the *external-links* folder [4].

This example uses jsoup [6], a Java library for parsing HTML files to get a website page and extract and manipulate the HTML data. It is used in that example to extract all external links from a certain web page. It then repeats the same extraction for links it has previously found until all links have been crawled once. The getting and parsing are done as follows:

```
val doc = Jsoup.connect(current).get()
```

This call will block until the HTML file is retrieved and parsed to the correct jsoup type. Then links are selected as a list, using:

```
doc.select("a").asScala.map(_.attr("href"))
```

As this example is straightforward and the connection is synchronous and does not have an open/close kind of statement, the adaptation to capture checking did not require changing the code too much. The only adaptation was to use the Collection Strawman instead of the normal Scala collection. This required to also create a new function `withFilter` for `ListBuffer` from the Strawman collection, as this type is used in a for-loop with an if.

B. Scraping docs

The example, adapted from [7], can be found in the *scraping-docs* folder [4].

This example, very similar to the previous one, will get a certain web page using jsoup and then for each found URL, it will get the page at that URL to get a summary of its content. The getting of the summary is done in a list of closures that are then applied. The type of the closures in the list is `Unit =>Tuples`. Where `Tuples` is a tuple of interesting elements to retrieve in the page, such as the URL, the summary, the name of the page, etc. This type is equivalent to a function capturing the universal capability: `{*} Unit -> Tuples` in the capture checking system.

The goal of this example was to correctly handle exceptions in a closure using the new `language.experimental.saferExceptions` that uses capabilities to throw "safer" exception in Scala. To do so, the following statement was used:

```
Jsoup.connect(...).get()
```

This statement throws a checked `IOException`.

As the call to jsoup is done in Java, it will not be tracked by the safer exception system. Indeed, capture checking and the safer exception system cannot detect Java's checked exceptions. This can be tested by using the following statement:

```
for (closure <- closures) closure(())
```

Indeed, even without enclosing this statement in a try/catch block, it will not fail as required. To prevent such behavior, a generic function has been created to be able to track and catch correctly Java exception:

```
def javaExceptionWrapper[A, E <: Exception](f: Unit
    => A): {f} A throws E =
  try {
    f(())
  } catch {case (e: E) => throw e}
```

Then the connect statement can be wrapped as follows:

```
javaExceptionWrapper(_ => Jsoup.connect(...).get())
```

This will ensure that Java functions throw in Scala by catching and throwing the exception again from Scala this time. Thus, the checked exceptions in Java can be tracked with this wrapper. Except that it loses the compile-time error advantages. Indeed, as you have to be aware of the Java code throwing errors and wrap it manually with the function defined above, you may miss checked exceptions that will only throw on runtime.

As the throwing block is in a closure, and this closure won't be the one handling the error, the following had to be added to the type of the closure: `throws IOException` to delegate the handling of this error to the part applying the closures:

```
List[Unit => Tuples throws IOException]
```

Now the statement applying closures will fail if not enclosed in a try/catch block. As it should for checked exceptions.

C. SQL queries

The example, adapted from [8], can be found in the *quill-sql-queries* and *standalone-sql-queries* folders [4].

This example creates a connection to a PostgreSQL database and run some SQL queries on it. The initial version was using Quill [9] (`io.getquill`), which was a problem due to the use of macros in Quill. This problem is depicted in Section III-A. The failing code is in the following line:

```
ctx.run(query[Country]).take(1)
```

As the `run` function is using macros with code that would fail to compile on the capture checking system. In this case, the error is that a type cannot be inferred in the capture checking system, even though it was correctly inferred without capture checking.

To adapt to this problem, this example has been rewritten as a standalone using only `java.sql` and the PostgreSQL driver for Java. To create the standalone, the following abstractions have been used for the different types of queries:

```
case class DataTable[A](tableName: String,
   fromResultSet: ResultSet => A,
   columns: List[String],
   toPreparedStatement: (A, PreparedStatement, Int)
       => Unit)

abstract class SQLQuery[A, B](dataTable:
    DataTable[A]) {
  def run(ctx: {*} Connection): B
  def getDataTable: DataTable[A] = dataTable
}

abstract class PipelinedQuery[A](dataTable:
    DataTable[A]) extends SQLQuery[A,
    List[A]](dataTable) { // ... }

abstract class NonPipelinedQuery[A](dataTable:
    DataTable[A]) extends SQLQuery[A,
    List[A]](dataTable) { // ... }
```

Using the abstraction above, the following queries can be defined as a `PipelinedQuery` using a `FilterQueryFromDataTable`, followed by `MapQuery`:

```
query[City].filter(_.id == cityId).map(_.name)
```

Then calling run `ctx.run(...)` on this query will execute the query using the run method defined in `SQLQuery`.

With these basic queries, the connection element of type `java.sql.Connection` was turned into a capability as it uses the notion of open and close. It can be seen as a try-with-resources pattern. To change it into a capability, the run method and multiple `SQLQuery` classes had to be adapted to accept a capability as a connection. Making it explicit makes it so that the query explicitly accepts that it can run this connection in its scope. This transforms the queries into capabilities as well as their capture set will contain the connection. For instance, a `PipelinedQuery` can be chained with a filter and it will have the following type/capabilities:

```
def filter(pred: A => Boolean): {pred, this}
    PipelinedQuery[A] = FilterQuery(pred, this,
    dataTable)
```

Where **`this`** becomes the previous query in the new `FilterQuery`. This means that `PipelinedQuery` always have a non-empty set of capabilities containing the previous query. This makes sense, as for a `PipelinedQuery` to be able to run, the previous query should still ”be in scope” and active as well.

Using the connection as a capability ensured that the connection would not leak to unwanted places and/or would not be used after `ctx.close()` has been called.

D. WebSockets

The example, adapted from [10], can be found in the *cask-websockets*, *http4s-websockets* and *standalone-websockets* folders [4].

This example shows the use of WebSockets by creating an in-browser chat using WebSockets. The initial example uses Cask [11] for the following connections/HTTP requests:

- HTTP Get: to get the HTML page on which the chat and form to post messages are displayed
- HTTP Post: to post a new message to the chat
- WebSocket connection: which is initiated by the client's browser after getting the HTML page and which the server will use to update the list of messages on a new message from any client

Once again this example could not work with Cask which is using annotations and macros. This problem is depicted in Section III-A.

Then to make this example work, http4s, a cats effect-based library using fs2 (functional streams), was used. This library was chosen among a lot of others since it does not rely on annotations and macros for requests and WebSockets but rather on `PartialFunction` for its routes. Another reason is that http4s is one of the few libraries that already have a Scala 3 version.

The difference between Cask and http4s is mainly in the way to handle WebSockets. For http4s, the WebSockets are handled as follows:

```
val toClient: Stream[F, WebSocketFrame] =
  Stream.fromQueueNoneTerminated(clientQueue)...
val fromClient: Pipe[F, WebSocketFrame, Unit] = ...
wsb.build(toClient, fromClient)
```

Thus, the `toClient` is a stream of elements and adding a `WebSocketFrame` to its underlying queue (from cats effect) will cause http4s to send the `WebSocketFrame` to the client. On the opposite the `fromClient`'s `Pipe` serves as the receiving end of messages sent by the client.

To use this behavior, the outgoing queue of each connected client is saved in a global list and each time the POST method is called with a new chat message, the update can be pushed to every client queue by just looping on the list and adding a new message to each underlying queue.

With this way of handling connections and WebSockets, it makes sense to transform the client's underlying queue into a capability. Indeed, this queue should only be used while the client is connected and should not be used elsewhere or after the WebSockets connection is closed. This is typical behavior for ”lifetime” variables. Moreover, queues should not have the universal capability, as it is never a good idea to use such general capabilities in a list. This means that the list of queues has the following types in the method `routes`:

```
val queueCapability: {*} Queue[F, Option[String]] =
    null
val openConnectionQueues:
    ListBuffer[{queueCapability} Queue[F,
    Option[String]]]
```

A capability `queueCapability` is ”artificially” created in the `routes` method and used in the capture set of the queues in the list. This ensure that queues are capabilities as well and that they have a limited lifetime in the `routes` method or when explicitly typed.

Due to some problems with the interaction between the capture checking system and the http4s library depicted in

Section III-B, it is not possible to pass a queue with capability to some http4s methods. There are multiple solutions to this, one would be to use the `Escaper` describes in Section III-C. Another solution would be to create the queue without capability and then transform it to be passed with capability to the list:

```
val queueAsCapability: {queueCapability} Queue[F,
    Option[String]] = newQueue
openConnectionQueues += queueAsCapability
```

To prevent this problem and to show how such examples of cats effect could be adapted in the capture checking system, this example has been adapted as a standalone using `java.net`. The same structure has then been used. Incoming requests are matched using a partial function that returns a response that is either a direct HTTP response or a WebSocket connection that will handle upgrading the HTTP connection to a WebSocket protocol and will keep it open and transmit information both ways. The WebSocket handler signature is as follows:

```
class WebSocketServerHandler(request: Request,
    client: Socket, in: InputStream, out:
    OutputStream, toClient: {*}
    ConcurrentLinkedQueue[WebSocketFrame],
    fromClient: {*} Pipe[WebSocketFrame, Unit])
```

It will upgrade the connection and then wait asynchronously for new messages from the client and messages to be sent to the client as messages added to the queue `ConcurrentLinkedQueue`.

The partial function of the routes is also adapted and has the type `PartialFunction[Request, {queueCapability} Response]`.

This means that the response, which can either be treated directly as a simple HTTP response or can keep a connection open for the WebSocket connection, is a capability. This makes sense in the case of a WebSocket connection, it makes it a lifetime element as it should handle the WebSocket connection as long as it is opened. One problem encountered here was the use of the partial function, for instance, the following does not work, even though it should:

```
class TestTemp(val capString: {*} String = "") {
  def testFunction: PartialFunction[Int,
      {capString} String] = {
    case t if t == 3 =>
      val v: {capString} String = "3"
      v
    case t if t == 5 =>
      val v: {capString} String = "5"
      v
  }
}
```

It will fail with the following error:

```
|Found: {FlakyPartialFunctionBug.this.capString} B1
|Required: B1
```

Whereas, this would work, even though the returning type of the partial function is the same:

```
class TestTemp(val capString: {*} String = "") {
  def testFunction: PartialFunction[Int,
      {capString} String] = {
    case t if t == 3 =>
      capString + "3"
    case t if t == 5 =>
      capString + "5"
  }
}
```

This example shows that capture tunneling for partial functions is flaky and not fully reliable. Thus, to prevent this flakiness, the standalone example could be adapted by using a full function of type `Request =>{queueCapability} Response` and using pattern matching on the request instead.

Finally, the WebSocket example was completed with the safer checked exceptions system. This system was used on receiving and sending new messages from the `WebSocketServerHandler`. Indeed some operations from the WebSocket protocol have not been implemented for simplicity and they were not useful for this example. Thus, these methods should throw an exception if such parts of the protocol are used. The two methods then have the following signatures:

```
def sendToClient(wsf: WebSocketFrame): Unit throws
    UnknownWebSocketFrameException = ...
def receiveFromClient: WebSocketFrame throws
    UnknownWebSocketFrameException |
    UnsupportedWebSocketOperationException = ...
```

As these methods have `throws` in their signature, the compiler with safer exceptions will enforce that the calling method `handle()` should either catch the exceptions or add a `throws` statement to its signature as well. The second solution was chosen as the `WebSocketServerHandler` cannot know how to handle these exceptions. Thus the server, which is calling `handle()` will be the one handling the exceptions. It will do so the following way:

```
Future.unit.map(_ =>
  val wssh: {toClient, fromClient}
      WebSocketServerHandler =
      WebSocketServerHandler(...)
  try {
    wssh.handle()
    queue.add(0)
  } catch {
    case _: UnknownWebSocketFrameException =>
      wssh.closeAndFreeResources()
      queue.add(1)
    case _: UnsupportedWebSocketOperationException =>
      wssh.closeAndFreeResources()
      queue.add(2)
  }
)
```

Two interesting observations can be made about the above code. The first one is that all this code is wrapped in a `Future` call, this ensures that the server can handle other requests while connected to one or multiple clients using the WebSocket protocol by handling the WebSocket protocol asynchronously. The second interesting element is that it had to catch exceptions, otherwise the code would fail on

compilation.

III. General observations

A. Macros

One problem that was encountered with the use of that kind of system is macros. Indeed, macros are code that are parsed on compile-time inside some other code using `macro` or `@annotation`. This causes a problem for the capture checking system as this code will then be analyzed by the type system with capture checking after being parsed. Even though the code of the macros might come from a library that was compiled without capture checking and thus, causing an error that was not detected before while compiling the library. This error cannot be fixed without actually changing the library which is the problem as well. This is because the failing code is on the library side and not on the side that can be modified.

This problem has no solution yet but is a problem that does not only appears with capture checking, but it is a rather known problem of using macros in Scala (and in general in programming languages).

B. Interaction with Scala libraries

There are multiple interactions with Scala libraries that can cause problems in the capture checking system, the first one being the macros explained in Section III-A. A second problem that may arise is to try to put a capability as an argument in a function of an external library. This case arose while using http4s:

```
(newQueue: {queueCapability} Queue[F,
    Option[String]]) => {
  // ...
  val toClient: Stream[F, WebSocketFrame] =
    Stream.fromQueueNoneTerminated(newQueue).map(s
        => Text(s))
  val fromClient: Pipe[F, WebSocketFrame, Unit] =
      _.evalMap {
    case Text(t, _) => F.delay(println(t))
    case Close(_) => F.delay(openConnectionQueues -=
        newQueue)
    case f => F.delay(println(s"Unknown type: $f"))
  }
  // ...
}
```

The code above would fail with the following error:

```
|Stream.fromQueueNoneTerminated(newQueue).map(s =>
    Text(s))
|Found:    (newQueue : {queueCapability}
    cats.effect.std.Queue[F, Option[String]])
|Required: cats.effect.std.QueueSource[F, Option[?
    String]]
// ...
|case Close(_) => F.delay(openConnectionQueues -=
    newQueue)
|Found:    {newQueue} () ?-> Unit
|Required: () ?-> Unit
```

Where the signature of both method are respectively:

```
def fromQueueNoneTerminated[F[_]: Functor, A](
  queue: Queue[F, Option[A]]
): Stream[F, A]
```

and

```
def delay[A](thunk: => A): IO[A]
```

Thus, the first error is due to the fact that the method expects a pure variable of type queue and received a capability with a capture set of `{queueCapability}`. The second error is a bit more tricky: `delay` expects a variable passed by-name which is transformed to a context function, for instance `thunk: =>Unit` is transformed to `thunk: ()?=>Unit`. Then on the capture checking system this context function is turned into a capability capturing the universal capability: `thunk: {*} ?-> Unit`. From there the following equations can be made:

Listing 1: Subtyping rules in CC

```
A => B ≡ {*}A -> B (General rule for functions)

{c} <: {*} ⟹ {c}A <: {*}A (Subtyping)

(B1<:A1  A2<:B2) / (A1=>A2<:B1=>B2)  (General subtyping rule functions)
```

Then using all the equations defined in Listing 1 together gives us:

```
⟹ {*}A -> B <: {c}A -> B
```

Then by replacing by the right types:

```
⟺ {*}() -> Unit <: {c}() -> Unit
```

Thus the given type with a capture set containing a specific element is a supertype of the expected type, which is not possible as the argument of a function in Scala is contravariant. This proves that this works as intended and can cause problems for all function and by-name arguments sent to a library call with capability.

The same problems is happening with: `Future {...}` from the standard Scala library as Future is expecting a by-name parameter. Moreover, the following works:

```
Future.unit.map(_ => {
  // expr with capabilities
})
```

Even though it is said in the documentation of `Future.apply` that

```
* The following expressions are equivalent:
* {{{
* val f1 = Future(expr)
* val f2 = Future.unit.map(_ => expr)
* val f3 = Future.unit.transform(_ => Success(expr))
* }}}
```

This is because the map function takes a function and not a by-name parameter, which breaks the Future's documentation promise.

C. Interaction with Java libraries

As the capture checking system is complex and fully based on Scala, the interaction with Java libraries and Java code has been adapted. Thus, Java code is not checked in the capture checking system. This helps the interaction with Java libraries but has one downside which is that Java could be used to escape a variable from capture checking. This problem can be shown as follows:

If the following class is defined in Java:

```
class Escaper {
  public static Object escape(Object el) {
    return el;
  }
}
```

Then the following would be working:

```
object Main {
  def takeWithoutCC(s: String): String =
    s + " should fail"
  def main(args: Array[String]): Unit =
    val s: {*} String = "a string with capability"
    println(takeWithoutCC(Escaper.escape(s).
        asInstanceOf[String]))
}
```

Even though, without the Escaper: `takeWithoutCC(s)` would fail on compilation. This is a small hack that could both be abused to escape valid capabilities or be used for instance to escape library calls that cannot be modified and would fail with capabilities as explained in Section III-B.

As explained in Section II-B, another problem that may arise from not checking and tracking Java code is safer exceptions. Indeed, Java code that throws checked exceptions will not be tracked in the safer exception system and will require adding the `javaExceptionWrapper` function as a wrapper to catch and throw the exception again, so that the safer exception system can track it.

IV. General strategies to port

A general strategy to port Scala code to the capture checking system is to directly compile with capture checking. Indeed, doing so will let you have direct feedback. As the capture checking system is more strongly typed than Scala without it, it will fail where errors appear. This means that compiling will give first-hand errors about the code. This gives good insights into what will have to be fixed. This is also the goal of the capture checking system, to fail on compile-time rather than on runtime. It is then possible to fix errors by adding capabilities to some variables and also seeing detected code as possible leaks/errors.

Once all such errors have been fixed, the method- or class-parameters and local variables in the code should be analyzed. Variables or parameters that make sense to be turned into a capability should be transformed as such and code should be compiled again, compile-time errors should be fixed again, and so on. This should be done until all variables that make sense to be capabilities are transformed.

Another general strategy to detect possible candidates for capabilities is to detect try-with-resources patterns and more generally recurring elements and asynchronous parts in the code. To detect recurring elements, the logical graph of the program can be used.

For instance using the standalone WebSockets example presented in Section II-D, the following (simplified) logical graph can be generated:

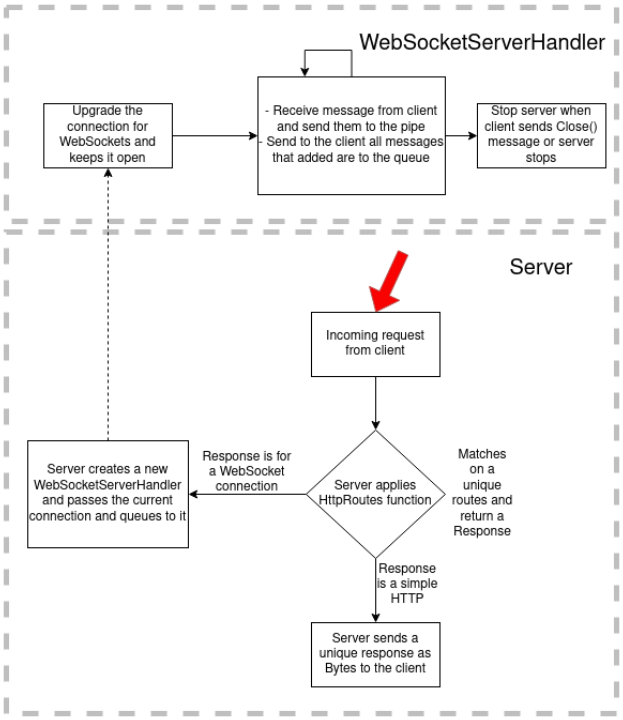

Fig. 1: Logical graph of the standalone WebSockets example

As it can be seen in the graph above, the middle block of the *WebSocketServerHandler* loops on itself and exits only on certain conditions. This can be seen as a sign of a possible capability. In that case, the underlying WebSocket handler's input/output streams can be transformed into capabilities. Such graphs can help detect this kind of behavior.

V. Bugs

While porting the examples presented in Section II, multiple bugs have been found in the dotty system with capture checking. The first one was a bug where for some cases `A =>B` was not equals to `{*} A -> B` and was actually crashing for `A =>B` and compiling correctly for `{*} A -> B`. This bug was fixed after the issue was reported on dotty [12].

Another bug was about for-comprehension and their desugared version behaving differently while using the capture checking system. Indeed, the following would fail:

```
for {
  a <- Some(1)
  optFunction: OptFunc[Int, String] =
      _.map(_.toString)
} yield toOptional(a, optFunction)
```

Whereas the following compiles correctly:

```
Some(1).flatMap(a => {
  val optFunction: OptFunc[Int, String] =
      _.map(_.toString)
  toOptional(a, optFunction)
})
```

This bug is currently not fixed in dotty, even though it was reported [13].

A third found bug is about partial functions being flaky while returning a capability as explained in Section II-D. This was not reported to the dotty system, as it is hard to define a clear case of what is working and what is not working.

VI. Conclusion

In this project, it was shown through example how the capture checking system can be used to simplify and improve Scala code. It was also shown that problems may arise while porting Scala code to the capture checking system. Overall, bugs and problems are very specific in the capture checking system but may arise in every code and library with this system.

References

[1] Martin Odersky et al. "Safer exceptions for Scala". In: *Proceedings of the 12th ACM SIGPLAN International Symposium on Scala*. 2021, pp. 1–11.

[2] Li Haoyi. *Hands-on Scala*. Haoyi, Li, 2020.

[3] *Capture checking Collection Strawman*. https://github.com/lampepfl/dotty/blob/cc-experiment/tests/run-custom-args/captures/colltest5/CollectionStrawManCC5_1.scala.

[4] Simon Hayoz. *Capture checking examples*. https://github.com/simhayoz/capture-checking-examples.

[5] Li Haoyi. *External links example, chapter 11.4*. https://github.com/handsonscala/handsonscala/blob/v1/examples/11.4%20-%20ExternalLinks/ExternalLinks.sc.

[6] *jsoup*. https://jsoup.org/.

[7] Li Haoyi. *Scraping docs example, chapter 11.2*. https://github.com/handsonscala/handsonscala/blob/v1/examples/11.2%20-%20ScrapingDocs/ScrapingDocs.sc.

[8] Li Haoyi. *SQL queries example, chapter 15.1*. https://github.com/handsonscala/handsonscala/blob/v1/examples/15.1%20-%20Queries/Queries.sc.

[9] *Quill*. https://getquill.io/.

[10] Li Haoyi. *WebSockets example, chapter 14.4*. https://github.com/handsonscala/handsonscala/blob/v1/examples/14.4%20-%20Websockets/app/src/MinimalApplication.scala.

[11] *Cask*. https://com-lihaoyi.github.io/cask/.

[12] *First reported bug*. https://github.com/lampepfl/dotty/issues/14782.

[13] *Second reported bug*. https://github.com/lampepfl/dotty/issues/15005.