Investigations About Multi-Parameter Type Classes for the Possible Use in General Type Safe Data Processing in Frege

Thibault Gagnaux
Supervisors: Prof. Dr. Dierk König
University of Applied Sciences and Arts Northwestern Switzerland
School of Engineering

Abstract

Processing a data source line by line is a common data processing operation. This operation can be supported by the type system for statically-typed languages such as Haskell or Frege by only allowing matching line producer-consumer type pairs to reduce programming errors. A type classes approach in Frege provides a way to abstract the source of the data but is only able to support monomorphic text type line producer-consumer pairs. This article shows that the use of a free monad transformer data type can successfully extend the monomorphic limitation to arbitrary line producer-consumer pairs. It concludes by uncovering that free monads are part of multiple ongoing research areas.

Index Terms

monad transformer, coroutines, stream processing

I. Introduction

Reading or streaming data line by line is a common data processing operation, especially if the data is too big to fit into your machine's memory. The programming language Groovy has the function `eachLine()` [1] for this purpose. Listing 1 shows a simple example, which uses the `eachLine()` function.

```
def data = '''\
first line,
second line
'''

data.eachLine { line, count ->
  if (count == 0) {
    println "line $count: $line" // Output: line 0: first line,
  }
}
```

Listing 1: A simple `eachLine()` example in Groovy.

The simple example in Listing 1 uses the more general stream processing model. The stream processing model consists of two main actors: The producer generates lines of data which the consumer processes line by line. This producer-consumer pair only works if they both speak the same language. For example, if the producer generates text data, but the consumer can only process numbers, we have a mismatch. This can lead to easy programming errors with undesired and unexpected behaviour as can be seen in Listing 2.

```
def sum = 0
data.eachLine { line, count ->
  sum += line
}
println sum // Output : 0first line,second line
```

Listing 2: A simple programming error resulting from a producer-consumer mismatch.

One approach to detect such producer-consumer mismatches early is to switch to a statically-typed language, such as Haskell or Frege, which leverage the type system to report these type mismatches at compile time. This composability property prevents the developer from writing incorrect programs early with a quick feedback loop.

In order to make full use of the type system and its composability property, the `eachLine()` function should be applicable to multiple types of line producers, for example, a file, URL or string line producer. In other words, the `eachLine()` function needs to work differently for each line producer type. This property is known as *ad-hoc polymorphism* coined by Strachey [2].

Kaes extended the form of ad-hoc polymorphism to the concept of *parametric overloading* [3] which Wadler and Blott independently introduced as *type classes* [4] a year later in Haskell. It is therefore a natural idea to implement the `eachLine()` function as a type class.

Prof. König followed that path in Frege, a Haskell for the Java Virtual Machine (JVM), and defined the `LineProducer` [5] type class. The approach turned out fruitful at first because the `LineProducer` type class allows to write generic code instances for the different native Java types `BufferedReader`, `File`, `String` and `URL`. However, the design has the surprising and major drawback that it is limited to a single `String` type line consumer. As a result, it only supports the composition of line producers which generate `String` values with `String` line consumers.

One well-known language extension to type classes are *multi-parameter type classes* [6]. The original uninvestigated hypothesis was that multi-parameter type classes could overcome this single type limitation and generalise the `LineProducer` type class to support arbitrary line producer-consumer type pairs. However, Frege does not support multi-parameter type classes which led to investigations about alternative solutions.

As a result of these investigations, this article shows that the simple application of the free monad transformer *Coroutine* [7] introduced by Blazevic supports not only composable arbitrary line producer-consumer type pairs in Frege but also an alternative approach to handle effects.

The rest of the article is structured as follows: Section II describes the intuition behind using coroutines and free monad transformers for the line producer-consumer problem and explains why the generalisation works on a concept level. Section III shows the generalisation with code examples and Section IV concludes with a summary and names some areas for future work.

II. The Way to Coroutines and Free Monad Transformers

The `LineProducer` type class abstracts over the origin of the line data. In contrast, abstracting over composing only matching line producers with matching line consumers means the correct match must be determined. Or in other words: it must be *interpreted* as a correct line producer-consumer pair.

The idea of an interpreter is to separate the syntax or structure from its meaning. The general structure of the line producer and line consumer is the ability to either run a computation or suspend. The line producer either produces a line value or suspends whereas the line consumer either consumes a line value or suspends. Blazevic calls this notion of alternating computations and pauses *trampolines* and generalises it to the `Coroutine` [7] data type with a helper function `suspend` shown in Listing 3.

```
data Coroutine suspendFun m res =
  Coroutine
  { resume :: m (Either (suspendFun (Coroutine suspendFun m res)) res)
  }

suspend :: (Functor f, Monad m) => f (Coroutine f m res) -> Coroutine f m res
suspend s = Coroutine $ pure $ Left s
```

Listing 3: The `Coroutine` data type and `suspend` function from Blazevic in Frege.

The secret behind the `Coroutine` data type is realizing that it is *isomorphic* [8] to the free monad transformer data type `FreeT` [9]. A free monad transformer allows composing different effects with the concept of a *free monad* [10]. A free monad allows to transform any functor into a monad. Hence, the `Coroutine suspendFun m res` data type can be read as taking any functor `suspendFun`, interleaved with effects from the monad `m` and with return type `res`.

Blazevic [7] uses the ability of the free monad to define the two new data types `Generator` and `Iteratee` as depicted in Listing 4. `Generator` is simply a `Coroutine` using the `((,) a)` pair functor and the `Iteratee` is a `Coroutine` using the `((->) a)` function functor.

```
type Generator a m x = Coroutine ((,) a)  m x
type Iteratee  a m x = Coroutine ((->) a) m x
```

Listing 4: The `Generator` and `Iteratee` data type from Blazevic in Haskell.

The `Generator` and `Iteratee` are just synonyms for the introduced line producer and line consumers. Therefore, they are defined as type synonyms as shown in Listing 5 and will be used interchangeably from now own.

```
type Producer a m b = Generator a m b
type Consumer a m b = Iteratee  a m b
```

Listing 5: The `Producer` and `Consumer` data type in Frege.

In summary, the line producer and line consumer are just two `Coroutine` types with different suspension functors. Since they both share the same `Coroutine` structure by the underlying free monad concept, they can be interpreted by the same interpreter. This is the final connection to the interpreter abstraction.

III. Examples

A. Producer/Consumer

The **Producer** and **Consumer** data type are free monads on their own. Therefore, Blazevic [7] goes on and defines the `yield` and `await` smart constructor functions to generate producers and consumers respectively and the `runGenerator` and `runIteratee` functions to run them as shown in Listing 6.

```
yield :: Monad m => a -> Generator a m ()
yield a = suspend (a, pure ())

await :: Monad m => Iteratee a m a
await = suspend pure

runGenerator :: Monad m => Generator a m res -> m ([a], res)
runGenerator = go id
  where
    go f gen = do
      eitherGen <- resume gen
      case eitherGen of
        (Left (x, newGen)) -> go (f . (x:)) newGen
        (Right x)   -> pure (f [], x)

runIteratee :: Monad m => [a] -> Iteratee a m res -> m res
runIteratee xs iter = do
  eitherCo <- resume iter
  case eitherCo of
    (Left co) ->
      case xs of
          [] -> runIteratee [] $ co $ error "No more values to process"
          (x:xs') -> runIteratee xs' $ co x
    (Right x) -> pure x
```

Listing 6: The `yield`, `await`, `runGenerator` and `runIteratee` functions from Blazevic in Frege.

The `runGenerator` function simply collects the generated values to a list. But note that it returns both the collected list and a return value of the `runGenerator` as a monadic pair. The `runIteratee` function, on the other hand, takes a list and an **Iteratee** as arguments and returns the final monadic value of the **Iteratee** computation.

Now it is finally time to run the first simple **Producer** and **Consumer** examples which can be seen in Listing 7.

```
main :: IO ()
main =
 do
   gen =
     do
       yield 1
       yield 2
       pure 42
   iter =
     do
       lift $ print "Enter two numbers: "
       a <- await
       b <- await
       lift $ println $ "Sum is " ++ show (a + b)
  (xs, res) <- runGenerator gen
 println $ show xs ++ " " ++  show res
 runIteratee xs iter
 -- Output: [1, 2] 42
 --         Enter two numbers: Sum is 3
```

Listing 7: A simple **Producer** combined with a **Consumer** in Frege.

B. Interpreter/Pipe

The example in Listing 7 already shows that producers and consumers can be combined. However, Blazevic generalises the communication between producers and consumers with the introduction of the pipe2 [7] function as depicted in Listing 8.

```
bindM2 :: Monad m => (a -> b -> m c) -> m a -> m b -> m c
bindM2 f ma mb = do
    a <- ma
    b <- mb
    f a b

pipe2 :: Monad m => Generator a m x -> Iteratee (Maybe a) m y -> m (x, y)
pipe2 gen iter =
  bindM2 go (resume gen) (resume iter) where
    go (Left (x, gen')) (Left funIter) = pipe2 gen' (funIter $ Just x)
    go (Right x) (Right y)             = pure (x, y)
    go (Left (x, gen')) (Right y)      = pipe2 gen' $ return y
    go (Right x) (Left funIter)        = pipe2 (return x) (funIter Nothing)
```

Listing 8: The pipe2 function which runs the producer-consumer pair in a synchronized manner using the bindM2 helper function.

The pipe2 function takes a **Generator** a m x and **Iteratee** (**Maybe** a) m y as argument and returns both the generator's and the iteratee's return values as a pair in the monad m. The first case distinction in the helper function go handles the case when both producer and consumer are suspended. The second case handles the situation when both producer and consumer return and the remaining two deal with only one of the two returning and the other suspending. Note that **Nothing** is returned in the last case distinction to signal that the producer has already finished providing values. In other words, pipe2 synchronises the producer-consumer pair because it is triggered for each suspension and resumption. As a result, no deadlocks or race conditions are possible according to Blazevic.

Finally, note that pipe2 serves as the interpreter because it allows only matching values of type a for the producer and consumer.

C. eachLine() with Producer and Consumer

With the pipe2 function defined it is finally possible to transform the eachLine() function from Groovy to Frege. But first the line producer and the line consumer need to be defined. The line producer function produceLinesFromFile is shown in Listing 9. It just reads the file line by line using the native BufferedReader Java type.

```
produceLinesFromBufferedReader :: Mutable s BufferedReader -> Producer String (ST s)
 ↪  ()
produceLinesFromBufferedReader bufferedReader =
  do
    line <- lift $ bufferedReader.readLine
    case line of
      (Just x) ->
        do
          yield x
          produceLinesFromBufferedReader bufferedReader
      (Nothing) ->
        do
          lift $ bufferedReader.close
          pure ()

produceLinesFromFilePath :: String -> Producer String IO ()
produceLinesFromFilePath filePath =
  do
   bufferedReader <- lift $ openReader filePath
   produceLinesFromBufferedReader bufferedReader

produceLinesFromFile :: File -> Producer String IO ()
produceLinesFromFile file = produceLinesFromFilePath file.getPath
```

Listing 9: The produceLinesFromFile function which reads a file line by line returning them as String.

The line consumer shown in in Listing 10 counts the number of lines and prints them as the function name suggests. An interesting detail to note is that the countAndPrintLinesConsumer makes use of the Show type class constraint and it can thus be combined with any *showable* type producer. With this approach, type classes and their benefits can still be used.

```
countAndPrintLinesConsumer :: Show a => Int -> Consumer (Maybe a) IO Int
countAndPrintLinesConsumer count =
  do
    line <- await
    case line of
      (Just l) ->
        do
          lift $ println $ show (count + 1) ++ ": " ++ (show l)
          countAndPrintLinesConsumer (count + 1)
      (Nothing) -> pure count
```

Listing 10: The produceLinesFromFile function which reads a file line by line returning them as String.

What remains is to interpret the producer-consumer pair using the pipe2 function as depicted in Listing 11.

```
main :: IO ()
main =
 do
   testFile = File.new "TestFile.txt"
   writeFile testFile.getPath $ unlines ["first line","second line","third line"]
   (xs, count) <- pipe2 (produceLinesFromFilePath testFile.getPath)
    ↪  (countAndPrintLinesConsumer 0)
   println $ "total number of lines: " ++ show count
   -- Output: 1: "first line"
   --         2: "second line"
   --         3: "third line"
   -- total number of lines: 3
```

Listing 11: The eachLine() function from Groovy transformed to Frege using a line producer-consumer pair.

D. Arbitrary Line Producer-Consumer Pairs

The acute reader may have noticed that the support of arbitrary producer-consumer pairs has not yet been clearly demonstrated. That is why Listing 12 shows a Person producer-consumer pair.

```
data Person = Person {
    name :: String,
    age  :: Int
}

producePerson :: Producer Person IO ()
producePerson =
  do
    yield $ Person "Elon Musk" 49
    yield $ Person "Jeff Bezos" 57
    yield $ Person "Bill Gates" 65
    pure ()

personConsumer :: Consumer (Maybe Person) IO ()
personConsumer =
  do
    maybePerson <- await
    case maybePerson of
      (Just p) ->
        do
          lift $ println $ "Processing " ++ p.name
          personConsumer
      (Nothing) -> pure ()
main :: IO ()
main =
  do
    (_, _) <- pipe2 (producePerson) (personConsumer)
    pure ()
    -- Output: Processing Elon Musk
    --         Processing Jeff Bezos
    --         Processing Bill Gates
```

Listing 12: A simple producer-consumer example using a custom data type `Person`.

The reason behind the general type support lies in the producer's `((,) a)` and consumer's `((->) a` suspension functor. They both support arbitrary values a.

Even more examples, including the already mentioned, are documented in the *github* [11] repository.

IV. Conclusion

The coroutine free monad transformer data type by Blazevic provides the necessary basis to build a processing model where computations and pauses alternate. The carefully chosen functors `((,) a)` and `((->) a)` define the producer and consumer monad respectively and due to the general type a they further extend the producer-consumer data processing operation to arbitrary types. As a final step, the `pipe2` function connects the producer and the consumer by interpreting the line producer-consumer pair in a synchronized way.

However, Blazevic's paper does not stop with simple producer-consumer pairs. He also introducers the *transducer* [7] data type, which is also known as an enumeratee. The transducer data type allows to transform the output of a producer or consumer and pass it on to another consumer resulting in a complete map reduce pipeline. His ideas have strongly influenced the two popular streaming libraries *Pipes* [12] and *Conduit* [13] and have recently sparked more research on how to *optimize coroutine pipelines* [14] [15] even further.

On the other side, the free monad transformer concept exposes an alternative approach to monad transformers on how to compose monads. These *different approaches* [16] [17] to the monad composition problem are an actively researched topic as well. This topic is of special interest to Frege, which is still missing Haskell's central monad transformer library *mtl* [18]. Finally, an especially interesting and still uninvestigated question resides if the introduced producer-consumer problem could be solved by the multi-parameter type classes extension in Haskell.

References

[1] Apache Software Foundation, “eachline.” [Online]. Available: https://docs.groovy-lang.org/3.0.7/html/groovy-jdk/java/io/Reader.html

[2] C. Strachey, “Fundamental concepts in programming languages,” *Higher-order and symbolic computation*, vol. 13, no. 1, pp. 11–49, 2000.

[3] S. Kaes, “Parametric overloading in polymorphic programming languages,” in *European Symposium on Programming*. Springer, 1988, pp. 131–144.

[4] P. Wadler and S. Blott, “How to make ad-hoc polymorphism less ad hoc,” in *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1989, pp. 60–76.

[5] Dierk König, “LineProducer in Frege,” https://github.com/Dierk/fregeTutorial/blob/master/src/main/frege/suggest/LineProducer.fr, 2020, Accessed: 2021-02-05.

[6] S. P. Jones, M. Jones, and E. Meijer, “Type classes: an exploration of the design space,” in *Haskell workshop*. Citeseer, 1997, pp. 1–16.

[7] M. Blazevic, “Coroutine pipelines,” *The Monad Reader*, vol. 19, pp. 29–50, 2011.

[8] illabout, “what is the relationship between Haskell’s FreeT and Coroutine type,” https://stackoverflow.com/questions/45192066/what-is-the-relationship-between-haskells-freet-and-coroutine-type, 2017, Accessed: 2021-02-05.

[9] Edward Kmett, “free: Monads for free.” [Online]. Available: https://hackage.haskell.org/package/free

[10] W. Swierstra, “Data types à la carte,” *Journal of functional programming*, vol. 18, no. 4, p. 423, 2008.

[11] Thibault Gagnaux, “line-producer,” https://github.com/tricktron/line-producer, 2021, Accessed: 2021-02-05.

[12] Gabriel Gonzales, “pipes: Compositional pipelines.” [Online]. Available: https://hackage.haskell.org/package/pipes

[13] Michael Snoyman, “conduit: Streaming data processing library.” [Online]. Available: https://hackage.haskell.org/package/pipes

[14] M. Spivey, “Faster coroutine pipelines,” *Proceedings of the ACM on Programming Languages*, vol. 1, no. ICFP, pp. 1–23, 2017.

[15] R. P. Pieters and T. Schrijvers, “Faster coroutine pipelines: A reconstruction,” in *International Symposium on Practical Aspects of Declarative Languages*. Springer, 2019, pp. 133–149.

[16] O. Kiselyov, A. Sabry, and C. Swords, “Extensible effects: an alternative to monad transformers,” *ACM SIGPLAN Notices*, vol. 48, no. 12, pp. 59–70, 2013.

[17] O. Kiselyov and H. Ishii, “Freer monads, more extensible effects,” *ACM SIGPLAN Notices*, vol. 50, no. 12, pp. 94–105, 2015.

[18] Andy Gill, “Monad classes, using functional dependencies.” [Online]. Available: https://hackage.haskell.org/package/mtl