University of Bergen
Department of Informatics

Generic programming using Higher Kinded Data

Author: Kathryn Frid
Supervisor: Jaakko Timo Henrik Järvi

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

June, 2023

Abstract

This thesis describes datatype-generic programming, what it is, and how it is done in Scala. The thesis covers ways of thinking about datatype-generic programming and today's tools and libraries for datatype-generic programming in Scala and summarize how to use them.

A new library called perspective for datatype-generic programming with higher kinded data is presented. How these libraries, including perspective, work together with sum types is be covered. Benchmarks are presented on both runtime and compile time performance of perspective and other libraries. perspective manages to keep up in performance with other libraries while offering faster code at the expense of longer compile time if a developer opts into this.

Finally, an example language-integrated query library built on perspective is described.

Acknowledgements

First of all, major thanks to Jaakko Järvi, my supervisor. Roughly four years ago I showed him what would eventually become DataPrism, he offered feedback, and suggested I write my thesis about it, and in a way, I did. Later on he also accepted being my supervisor for this thesis. Without him, this thesis would not be as good as it is.

I would also like to thank Uwe Wolter. It was through his class in Cateogry Theory and tips in the class that I understood how to handle sum types in perspective, and roughly what this handling means theoretically.

I would also like to thank a group of friends that have helped me along with functional programming more generally, and offered feedback where possible, while at the same time understanding somewhat what I am doing. Among these there is @JamesGallicchio, @rcano, Jade, and others. Github ids are given instead of names in some cases here.

Next, I would also like to thank my girlfriend for being there along the way, putting up with me working a bit too much on the thesis at times, and for coming up with the name DataPrism.

More generally, I would also like to thank Miles Sabin for creating shapeless, which is what initially got me interested in datatype-generic programming. Next, the TypeLevel team for being a wonderful functional programming organization, and making solid libraries. And lastly, Martin Ordersky for creating Scala and setting me upon the track of functional programming.

`Kathryn Frid`
Thursday 1st June, 2023

Contents

Chapter 1

Motivation

A software library within programming is a piece of code serving as a solution to a problem. For developers, libraries can be pretty nice, as they are code they do not have to write themselves. However, sometimes a specific library for a developer's needs might not exist. Other times such a library might exist, but it does not work well for the developer. Maybe it is slow, outdated, requires too much boilerplate to be written by the developer, or does not fit for some other reason. In these cases, the developer can write their own library to solve their problem. As the developer does so, they can create a solution that works better for how they want to use the library.

This is the situation I found myself in. I was working with an SQL query construction library and wanted more control over how the library generated the query. I had some ideas for how a library could allow this additional control, without losing functionality. The idea was to use generics at the type level, indicating what operations were valid to perform on the columns being handled. I tested if my library would typecheck with a few simple queries, and it did.

I began writing an implementation for my library and expanding on the idea. As I did, I started encountering problems with both the implementation and the types. Type inference was limited, and handling different valid types like single values, tuples, and case classes started to become difficult. While not realizing the problem then, I was missing the right "language" to express the library's operations. Eventually, I caught a glimpse of this "language" and continued working on the library. As time passed, I realized this "language" was not just for making SQL queries but for more general generic programming.

1.1 What is generic programming?

There are many different definitions of generic programming, and many languages have a feature that they might call generics, for example, Java [30], C# [42] and TypeScript [10]. Many of these languages talk about the same idea, parameterization. In the above languages, it is parameterization of types. Parameterization turns a hardcoded fact of a function or a program into a parameter. While the typical way of looking at generics is the parameterization of types, it is not the only way. For example, another kind of parameterization is taking programs and code as parameters. This mechanism can realize metaprogramming and macros.

What I will talk about in this thesis is datatype-generic programming. Datatype-generic programming inspects and manipulates the shape of data structures, which is what the code is parameterized by [19]. This type of generic programming allows one to generate useful functions that would require a lot of boilerplate code to write manually. Examples of such code could be encoding and decoding datatypes to other formats (like JSON) or showing these values as strings, but also more complex code, such as implementing a set of REST endpoints for manipulating a data structure.

1.2 Why datatype-generic programming?

A lot of programming is about defining some datatype and then implementing some functions using this datatype. Often these functions need to be implemented for many different types in the same way. Examples include verifying that the values conform to some restriction, encoding and decoding the data to a different format, or transforming some of the fields of a type. Without datatype-generic programming, each datatype has to reimplement this functionality. For example, if all the string fields of a datatype should be lowercased, this functionality has to be implemented with a new function specifically for this datatype. Typically, there are no features within normal generics (type parameterization) that would give access to all the fields of a type, access to the types of these fields, and allow functions to modify the values in these fields. Datatype-generic programming provides a way to do exactly this. With datatype-generic programming, one could implement a single `lowercase` function that one could use on all datatypes. In some ways, a library for doing generic programming can be said to be a library for building libraries.

```scala
case class RawThreadMetadata(
    archived: Boolean,
    autoArchiveDuration: Int,
    archiveTimestamp: OffsetDateTime,
    locked: Boolean,
    invitable: Option[Boolean],
    createTimestamp: Option[OffsetDateTime]
)

// Encoding by hand
implicit val rawThreadMetadataEncoderNoGeneric: Encoder[RawThreadMetadata] = obj =>
  Json.obj(
    "archived" := obj.archived,
    "auto_archive_duration" := obj.autoArchiveDuration,
    "archive_timestamp" := obj.archiveTimestamp,
    "locked" := obj.locked,
    "invitable" := obj.invitable,
    "create_timestamp" := obj.createTimestamp
  )

// With datatype-generic programming
implicit val rawThreadMetadataEncoder: Encoder[RawThreadMetadata] =
  derivation.deriveEncoder(derivation.renaming.snakeCase, None)
```

Figure 1.1: An AckCord definition and encoder definition written by hand and with a macro.

1.2.1 AckCord and datatype definitions

As an example of the power of datatype-generic programming, I describe a library I regularly work on: AckCord [14]. It is a library for building bots on the chat platform Discord. There are three core modules of AckCord. The first is a module for making requests to Discord in a ratelimit-compliant way. The second is a module to help establish a WebSocket connection to Discord, listening to events and sending them to a user to act on. The last core module is just datatype definitions used by the gateway and the requests modules. Of these modules, the requests module is the biggest in terms of lines of code (5164 lines), the datatype module a close second (4325 lines) and the smallest is the gateway module (1609 lines).

When I say the last module is datatype definitions, I mean it literally; some of the datatype definitions have a few one-line functions within them, but there is very little code other than the datatype definitions. The code to encode and decode datatypes to and from JSON, the most common operations performed on the datatypes, is all generated with generic programming. Figure 1.1 shows an example of data defined by AckCord, the code to encode the object to JSON by hand, and code that does the same using a macro.

By my rough guess, around 60% of AckCord is datatype definitions, 15% is algorithms, 15% is code passing datatypes around and transforming them in simple ways, and the last 10% is boilerplate. AckCord can get away with so much of the code being datatype definitions because generic programming handles a lot of code that would otherwise be needed. While this is where AckCord stands today, there is still room for improvement with more generic programming. For example, the generic programming AckCord uses for JSON encoding and decoding does not differentiate between a missing field and the field being `null`. As such, the code to handle this is today written by hand, but it could instead be avoided with more extensive use of generic programming.

1.3 In this thesis

In this thesis, I will go over the current generic programming facilities in Scala, my library for generic programming and optimizations applied to it. I will show how to use my library and generic programming to make the SQL query construction library I initially wanted. I will start by discussing generic programming and its alternatives, like macros. I will then talk about how to think of generic programming in general, before covering shapeless 2 and 3, popular libraries for generic programming in Scala. I will then explain the ideas behind my library, *perspective*, and how it supports effective and simple generic programming. I will then cover various performance boosts and optimizations perspective can perform while keeping the user-facing code mostly the same. Some of these optimizations can result in code that is in some cases faster than handwritten idiomatic code. I will then go over generic programming with sum types and how perspective's approach works pretty seamlessly with them. Lastly, I will cover DataPrism, the SQL library I wanted to make when I started this journey. DataPrism is built on top of perspective and the ideas it brings.

1.4 Contributions

The main thing this thesis provides is the library perspective. perspective is library for doing datatype-generic programming using higher kinded data. perspective defines a set of typeclasses that provides the language to operate on this higher kinded data. These operations are not unlike what a developer might use on a `List`, operations like `map`, `map2`, `foldLeft` and `traverse`.

By default, these operations can only be executed on higher kinded data. To allow using these operations on normal data, perspective defines a set of typeclasses (`HKDGeneric[A]` and its children) that allows conversion to and from a higher kinded data type where these operations can be performed.

perspective also exposes a typeclass to more efficiently operate on higher kinded data by indexing and tabulating over the data. The advantage of indexing and tabulation over the data is that most datatype-generic programming transformations can be done with a single iteration over the data.

For Scala 3, perspective also defines a new set of typeclasses (`InlineHKDGeneric[A]` and its children) to allow for more efficient bytecode generation using macros that generate while loops. While somewhat experimental, these while loops can then be gotten rid of using loop unrolling.

perspective's scheme for datatype-generic programming also extends quite seamlessly to sum types, and most operations available using product types are also available with sum types.

Chapter 2

Scala primer and background

This chapter will cover some Scala and related background needed to make sense of the later chapters. Readers who know Scala well can safely skip this chapter.

2.1 Algebraic data types

Generic programming generally deals specifically with what is called algebraic data types (ADTs). ADTs are built from two different kinds of types, product types and sum types. Algebraic data types give a way to represent data with "and"s and "or"s [21].

2.1.1 Product types

Product types represent Cartesian products of values. For example, a product type of size two could contain two independent integers. Product types represent "and"s. A pair could be said to be the simplest product type and can also represent all other product types through nesting. It is called a product type because the total values the type can represent is the product of the values the types that make up the product can represent. For example, a pair of two bytes could represent $256 * 256 = 65\ 536$ different values. Like with normal multiplication, there is also a type equivalent to the multiplication's identity element (1) for product types: the unit type `()` [29].

In Scala, a tuple could be said to be the simplest case of a product type. Adding `case` before a class definition allows Scala to recognize this definition as a product type. Here is an example of a product type in Scala and the equivalent tuple representation:

```scala
case class Foo(a: String, b: Int, c: Double, d: Boolean)

type FooTuple = (String, Int, Double, Boolean)
```

The Scala compiler will change how it generates `case classes` slightly to make them behave more like values instead of objects. Some ways of doing generic programming backed by `Mirror` (covered in Section 3.2) require `case class` on the data type they work with to work. Approaches to generic programming that use macros are not affected by this, as the macro needs to determine what it considers a product type. Making a type a `case class` might automatically fulfill these restrictions, but they can also be fulfilled in other ways. Figure 2.1 is an example to show some of what a `case class` generates automatically. This is not a complete list of what `case class` generates.

2.1.2 Sum types

Sum types, also sometimes called coproduct types, represent disjoint unions. A sum type is made up of a choice of several different types. Each of these types can then be lifted into the sum type. Sum types represent "or"s. Just like with product types, sum types also get their names from the number of possible values they can store. For sum types, this is the sum of the different types they are made up of. Like with product types, there is an equivalent type for the addition's identity element (0) with sum types, the uninhabited type. In Scala the uninhabited type is `Nothing` [29].

In Scala, `Either` could be said to be the simplest case of a sum type. Disjointness here is important, as Scala's union types are not sum types. All the different cases of a sum type should be known statically. With Scala 3, there are two ways to define sum types: sealed hierarchies and enums. Sealed hierarchies came first and are more flexible, while enums were introduced in Scala 3 and are more specialized in what they can do but they offer nicer syntax. Figure 2.2 is an example of sealed hierarchies and enums as sum types and the equivalent nested eithers representation. All of these data types represent the same sum type.

```scala
class FooClass(
  val a: String,
  val b: Int,
  val c: Double,
  val d: Boolean
) extends Product:

  override def hashCode: Int = ??? // Runtime specific implementation
  override def equals(other: AnyRef): Boolean = ??? // Runtime specific implementation
  override def toString: String = scala.runtime.ScalaRunTime._toString(this)

  override def productArity: Int = 4
  override def productPrefix: String = "FooClass"

  override def productElement(n: Int): AnyRef = (n: @scala.annotation.switch) match
    case 0 => a
    case 1 => b
    case 2 => c
    case 3 => d
    case _ => throw new IndexOutOfBoundsException(n.toString)

  override def productElementName(n: Int): String = (n: @scala.annotation.switch) match
    case 0 => "a"
    case 1 => "b"
    case 2 => "c"
    case 3 => "d"
    case _ => throw new IndexOutOfBoundsException(n.toString)

  def copy(a: String = a, b: Int = b, c: Double = c, d: Boolean = d): FooClass =
    FooClass(a, b, c, d)

  def _1: String = a
  def _2: Int = b
  def _3: Double = c
  def _4: Boolean = d

object FooClass extends scala.deriving.Mirror.Product:
  def apply(a: String, b: Int, c: Double, d: Boolean): FooClass =
    new FooClass(a, b, c, d)

  def unapply(fooClass: FooClass): FooClass = fooClass

  def toString: String = "FooClass"

  override def fromProduct(product: Product): FooClass =
    new FooClass(
      product.productElement(0).asInstanceOf[String],
      product.productElement(1).asInstanceOf[Int],
      product.productElement(2).asInstanceOf[Double],
      product.productElement(3).asInstanceOf[Boolean],
    )
```

Figure 2.1: What a case class generates.

```scala
sealed trait FooSealed
object FooSealed:
  case class BarSealed(i: Int) extends FooSealed
  case class BazSealed(s: String) extends FooSealed
  case class BinSealed(b: Boolean) extends FooSealed

enum FooEnum:
  case BarEnum(i: Int)
  case BazEnum(s: String)
  case BinEnum(b: Boolean)

type FooEither = Either[Int, Either[String, Boolean]]
```

Figure 2.2: Scala's sum types defined with a sealed hierarchy and and enum.

2.2 Companion objects

In some of the code shown so far, both a datatype and an object is defined with the same name. This object is called the companion object, and is what Scala uses instead of `static` instances found in Java. The companion object is special in a few ways. The Scala compiler will look for implicit instances (discussed below) for a type in the type's companion object. The companion object also has access to private members of a type it is the companion of.

2.3 Implicits: Given and Using

Scala implicits are a way of passing around function parameters implicitly. Usually, when one calls a function, one has to define all the function's parameters (unless they have default values). However, with implicit parameters if a parameter is not specified explicitly, the Scala compiler will try to look for a value in the calling context, based on specific lookup rules.

There are two steps for making use of implicit parameter passing. First, a function must mark one or more lists of parameters with `using`. Parameters in a parameter list marked with `using` can be anonymous, they do not need to be named. The second step is telling Scala what values are eligible to be passed implicitly, which is done with a `given` definition. A `given` definition looks like a normal `def` definition but there are some differences. All parameter lists of a `given` definition must be marked as `using`. A `given` definition can be anonymous, and the Scala compiler will generate a name for

the definition if this is the case. Parameters passed to a function implicitly can also be passed to other functions implicitly within the function's body.

The Scala compiler has a list of rules for where it should look for values to pass as implicit arguments. These rules include but are not limited to the following:

Enclosing scope: Implicit values defined in an enclosing scope can also be passed to function calls within that scope. For example, implicit parameters handed to a class can also be passed to code in the function bodies of this class.

Companion object of what is being looked for: For example, if some code needs a List implicitly, the Scala compiler could look for an instance in the companion object of List.

Companion object of types related to the type being looked for: In the same list example, the compiler would also look for instances of the list in the companion object of the type of the list's contents.

Imported Implicits can be imported and be found that way.

Note that Scala defines an ordering for where to look for implicits first. If the compiler finds multiple instances that are not ordered, it produces a compiler error. Figure 2.3 shows some examples of implicits, where values can be found and where they cannot. The example also shows how to pass values to parameters in a parameter list marked as `using` explicitly. `summon` is a function that returns an instance of the type passed to it and will be used here to show that implicit search succeeds. It is defined like this: `transparent inline def summon[A](using v: A): A = v`. Because of `inline` the function disappears in generated code, and because of `transparent`, the return type can be more specific than what was asked for.

2.3.1 Why use implicit parameters

There are myriads of different reasons and ways to use implicits. They are simply a tool. Here are some ways to use implicits.

Implicits can be used to pass around values through several levels of frames in the call stack without passing them explicitly and where most functions do not need to concern themselves with them. An object carrying configuration information or anything else one might use a reader monad for is a good example. The common alternative to parameter

```scala
package a1:
  given Int = 5
  package b1:
    //Enclosing scope
    summon[Int]

package a1.b2:
  //Fails. b2 is not enclosing b1
  //summon[Int]

class A2(given Int):
  //Enclosing scope
  def givenInt: Int = summon[Int]


class A3[A]
object A3:
  given A3[Int] = new A3

//Companion object
summon[A3[Int]]

class A4
object A4:
  given A3[A4] = new A3

//Companion object of "related" type
summon[A3[A4]]

object A5:
  given A3[Double] = new A3

import A5.given
//Imported
summon[A3[Double]]

object A6:
  given a3Boolean: A3[Boolean] = new A3

//Fails. Not imported
//summon[A3[Boolean]]

//Passing the value explicitly
summon[A3[Boolean]](using A6.a3Boolean)
```

Figure 2.3: Examples of ways to use implicits and where values are found.

passing is global variables, but parameter passing has advantages: they are more easily testable, they can work with multiple different values at the same time and they expose less mutable state.

Implicits can also be used to pass around capabilities. Scala has done this for ages with `Future` and `ExecutionContext`, with `ExecutionContext` being like a thread pool passed around to indicate where to perform work. More recently, the Scala 3 compiler team has been exploring further ideas around capabilities [24][25].

Implicits can also act as proofs about types. For example, the `=:=[A, B]` type can only be summoned if `A` and `B` are known to be the same type. There is also a weaker variant that only checks for subtyping, `<:<[A, B]`.

Implicits are also used for Scala's typeclasses. More on those in Section 2.7.

Lastly, they can be harnessed for logic programming.

2.3.2 Implicit search and logic programming

Implicits can, if taken far enough, interface directly with the type system, similarly to logic programing. Implicits can infer new types from existing ones or perform general programming. [27]

Using Prolog as an analog, normal values like atoms and numbers correspond to types. Singleton types can also be used here. A compound term in Prolog is like a type taking type arguments in Scala. A variable in Prolog can be represented as a type parameter in Scala.

A fact in Prolog is equivalent to a `given` definition in Scala. A rule, meanwhile, is equivalent to a `given` definition itself having `using` parameters. Negation is encoded with the special `NotGiven` type.

Figure 2.4 shows a set of Prolog definitions, and Figure 2.5 shows a direct translation into Scala.

Several key distinctions between Prolog and Scala can be seen here.

First of all, Scala requires code to use subtyping and always define terms before using them. In this example, I need to define the predicates, such as `Friend`, the values to use, such as `A1`, and the variables to use, the type arguments of the given definition.

```
friend(a, b).
friend(b, c).
friend(c, d).
friend(d, e).

symmetry(F, A, B) :- call(F, A, B).
symmetry(F, A, B) :- call(F, B, A).

friend_of_friend(A, B) :- friend(A, C), friend(C, B).

degrees_of_seperation(A, A, 0).
degrees_of_seperation(A, B, 1) :- friend(A, B).
degrees_of_seperation(A, B, M) :-
    friend(A, C), C \= B,
    degrees_of_seperation(C, B, N),
    succ(N, M).
```

Figure 2.4: An exaple program in Prolog.

The definitions also need to be named. Scala has some rules for naming `given` definitions automatically, but sometimes the compiler encounters conflicts when following those rules, and a user has to step in.

While not required to, the code also uses path-dependent types instead of type parameters, in this case with `DegreesOfSeperation` on line 9. Using path-dependent types makes it clearer what are the "return values" of the computation. The Aux pattern can be used to get back the "all type parameters" definition.

One last significant aspect to note about Scala is that it does not like ambiguity. If, for example, there was also a fact in the above definitions which stated that `a` and `d` were friends and the degrees of separation between `a` and `e` was queried, Prolog would give both answers, one after the other. On the other hand, Scala would refuse to give either, returning a compile error instead. Scala's dislike of ambiguity has gotten especially pronounced in Scala 3, where ambiguities are global errors instead of local ones, meaning that they are unrecoverable errors.

2.4 (Linked) lists

Linked lists, also known as just `List` in Scala are simple data structures for carrying around multiple values of a type. Lists are often used with functional programming for their simple definition, constant time prepend operation and the ease of matching on

```scala
import scala.compiletime.ops.int.S
import scala.util.NotGiven

// Abstract compound terms
trait Friend[A, B]
trait FriendOfFriend[A, B]
trait Symmetry[F[_, _], A, B]

trait DegreesOfSeperation[A, B]:
  type N <: Int

object DegreesOfSeperation:
  type Aux[A, B, N1 <: Int] = DegreesOfSeperation[A, B] { type N = N1 }

// Defining Succ here for more literal translation
// S is a type returning the successor of an integer. It works more like a function
// and less like a typical logic programming predicate.
// Succ as defined here operates as a more traditional logic programming predicate
trait Succ[N <: Int, M <: Int]
given [N <: Int]: Succ[N, S[N]] with {}

// Atoms
trait a; trait b; trait c; trait d; trait e

// Facts
given Friend[a, b] with {}
given Friend[b, c] with {}
given Friend[c, d] with {}
given Friend[d, e] with {}

// Rules
given symA[F[_, _], A, B](using F[A, B]): Symmetry[F, A, B] with {}
given symB[F[_, _], A, B](using F[B, A]): Symmetry[F, A, B] with {}

given [A, B, C](using Friend[C, B], Friend[A, C]): FriendOfFriend[A, B] with {}

given deg0[A]: DegreesOfSeperation[A, A] with {type N = 0}
given deg1[A, B](using Friend[A, B]): DegreesOfSeperation[A, B] with {type N = 1}

given degN[A, B, C, N1 <: Int, M <: Int](using
  Friend[A, C], NotGiven[C =:= B],
  DegreesOfSeperation.Aux[C, B, N1], Succ[N1, M]
): DegreesOfSeperation[A, B] with {type N = M}
```

Figure 2.5: A direct translation of the Prolog program into Scala using logic programming.

```scala
enum List[+A]:
  case ::(head: A, tail: List[A])
  case Nil extends List[Nothing]

object List:
  extension [A](head: A) def ::[AA <: A](tail: List[A]): List[A] = new ::(head, tail)

export List.{::, Nil}

val value: List[Int] = 2 :: 7 :: 5 :: Nil // How to define values of List

extension [A](l: List[A]) def foldLeft[B](b: B)(f: (B, A) => B): B =
  l match
    case h :: t => t.foldLeft(f(b, h))(f)
    case Nil    => b
```

Figure 2.6: Definition and use of List.

them. Figure 2.6 shows an example of how `List` can be defined, how to create values of `List` and how they can easily be matched on.

A `List` can either be a `::` (pronounced cons) holding an element or a `Nil` signaling the end of the list. For the rest of this thesis, instead of the definition in Figure 2.6, I will use Scala's built in lists. The essence of the definitions and their use remains the same.

2.5 Type definitions

Scala has many ways of expressing types. We use several of these constructions in the thesis.

2.5.1 Type "aliases"

Type aliases are simple definitions that can refer to other types. I say "aliases" as even though they are called aliases, they are more like type functions. Type "aliases" can take types as arguments and return other types based on the type arguments passed in. Anything that is a type can appear on the right-hand side of a type alias.

Here is a simple example of a type alias for a pair where both types are identical.

```scala
type Both[A] = (A, A)
```

2.5.2 Higher kinded types

Higher kinded types are types that themselves take types. `List`, for example, is a type still expecting a type. Scala allows higher kinded types to be passed as a type parameter and used with generics, like any other type. To do this, one indicates the "holes" in the type with underscores.

Here are two simple examples of higher kinded types and a type alias to apply them to some other types:

```
1 type Apply1[F[_], A] = F[A]
2 type Apply2[F[_, _], A, B] = F[A, B]
3
4 type Foo = Apply1[List, Int]
5 type Bar = Apply2[Function1, Int, String]
```

`Apply1` and `Apply2` without any other arguments are also higher kinded types. `Apply1` has the kind `F[_[_], _]` and `Apply2` has the kind `F[_[_, _], _, _]`.

2.5.3 Type lambdas

Type lambdas allow one to construct an inline type alias. Sometimes the kind of the type a developer wants to pass to some definition does not quite fit. Say the developer wanted to pass `Either` to `Apply1` defined above. That will not work, as `Either` has a different kind than what `Apply1` expects of its type parameter. Furthermore, there is no clear definition of what applying `Either` to `Apply1` would return either. The developer can, however, make the application work if they fix one of the types on `Either`. They could, for example, do this with a type alias like so:

```
1 type IntEither[A] = Either[Int, A]
2 type AppliedIntEither = Apply1[IntEither, String]
```

Defining an alias like this anytime the developer needs an "intermediary" type to construct well-kinded types can get tiring. In the past, people avoided having to define a type alias like this by abusing several obscure language features together to create an inline alias, essentially a lambda, but for types. Scala 3 added official support for type lambdas with a much nicer syntax. Here is a snippet showing both the old and new way of making a type lambda. Note the `=>>` arrow in the new example. It is important to distinguish the type lambda from a polymorphic function, which will be covered in the next section (2.6).

```
type AppliedIntEitherOld = Apply1[({type L[A] = Either[Int, A]})#L, String] // Old
type AppliedIntEitherNew = Apply1[[A] =>> Either[Int, A], String] // New
```

2.5.4 Match types

Type aliases can normally not inspect the type of an argument. *Match types* allow for this. They also allow types to refer to themselves recursively. Here is an example of a match type that extracts the elements of potentially nested lists:

```
type Contents[A] = A match
  case List[a] => Contents[a]
  case a       => a
```

Match types are Turing complete. This fact can be seen by building a Turing machine with match types, something I did once when I was bored [12].

2.6 Polymorphic functions

In Scala 2, methods had more power than functions. A function could take values as parameters but no types or implicit parameters, and their return values could not be dependently typed on the input. Scala 3 makes this gap smaller. Of note to this thesis are polymorphic functions, functions that take types as arguments. Here is a simple example of defining a `head` method and a similar function:

```scala
def headM[A](xs: List[A]): A = xs.head
val headF: [A] => List[A] => A = [A] => (xs: List[A]) => xs.head
```

While the type of `headF` is still optional, everything on the right after `=` is mandatory. There is little type inference for polymorphic functions. Polymorphic functions are also why type lambdas are written with the `=>>` arrow, to disambiguate them. Otherwise they would use the same syntax for different things.

2.7 Typeclasses

A typeclass is a container of sorts defined in terms of one or more type parameters and contains a collection of abstract functions. A typeclass can have many different instances with different types and function implementations. Sometimes, this can also include different function implementations for the same underlying type. A function can then use typeclasses instead of being hardcoded to a concrete type, making them more general [1].

For example, a function that sorts a list could work for lists of any type `A` provided there existed a typeclass for `A` which contained functions to compare instances of `A` against each other, and say which one is larger.

When compiling code, the compiler has a set of rules it follows to find typeclass instances for a certain type. Unlike with classes and inheritance, the behavior is defined separately from the definition of the type. The advantage of typeclasses is that they can adapt a type for new uses without the "owner" of the type knowing about the new use case. For example, in Java, and consequently Scala, all types with an ordering needs to extend `Comparable`. A library defines a type `A` for which there exists an ordering, but the type `A` does not inherit `Comparable` as the developer of the library had no need to compare values of `A`. A user of the library then comes along and needs to compare values of `A`. The user has two choices: wrap `A` in a new type which does extend `Comparable`, or use a typeclass. The typeclass `Ordering` already exists in Scala for cases like this.

Some typeclasses can have laws associated with them that instances have to fulfill to ensure that they behave as consumers of the typeclass might expect. For example, `Ordering` requires that a type has a partial ordering, that is to say, it requires reflexivity,

anti-symmetry, and transitivity over the `<=` operation, and that all values can be compared with each other [9, 8].

In Scala, typeclasses are modeled using implicits. Here is an example of a typeclass called Monoid:

```
1 trait Monoid[A]:
2   def empty: A
3   extension (lhs: A) def |+|(rhs: A): A
```

Monoid has the laws of associativity and identity, essentially stating that `a |+| (b |+| c) = (a |+| b) |+| c` and `a |+| empty = a` hold for all values. Numbers are one example of valid monoid instances, either with addition as the binary operation and 0 as the constant or, respectively, multiplication and 1. Strings are another example of a type for which there exists a valid monoid instance.

Another often-used typeclass is `Show`. It provides the functionality of `toString`, except as a typeclass. The `Show` typeclass is important, as it will be used as an example for typeclass derivation throughout this thesis. It is defined like this:

```
1 trait Show[A]:
2   extension (a: A) def show: String
```

Chapter 3

How to do generic programming

This chapter will cover different approaches to generic programming in Scala. It will discuss common ideas behind the different ways of doing generic programming, and also advantages and shortcomings of each method.

A common use for datatype-generic programming is to automatically make instances of various typeclasses for data types. This is called deriving typeclasses. For each method, an example will be provided showing how to derive a `Show` typeclass for a product type. Note that these examples only gives an idea of what programming using a particular method might look like. Each section here will focus on the "ideal" of the method, and less on the nitty-gritty that might sometimes be needed to make it tolerant towards errors, make it more general, or even make it work.

3.1 Macros

There are two main ways to access information about the fields and the types of these fields of a type in Scala 3. The first way is through macros.

Macros are special functions that execute at compile time. They give access to all the information the compiler is willing to expose to the macro function. The macro function then generates an abstract syntax that the compiler will insert in place of the call to the macro. The macro function can execute whatever code it wants while generating this abstract syntax tree. If it finds something it did not expect, it can cancel the compilation with a normal compile error.

In general, macros are the most powerful tool for generic programming in Scala. Some information, like annotations, is only accessible in macros, which makes it necessary for users to build layers on top of macros if they want access to this information outside of macros.

Macros are often dense, and the documentation might talk a lot about the "what", but less about the "why". For example, the type of a symbol can be obtained in multiple different ways, and it is not always clear to the developer which way to use. Some information, like a method's default values, requires knowledge about how the compiler encodes this information in the final classfiles. In the case of default values, they are encoded as additional functions named based on the name of the original function, and the index of the parameter. A user wanting to use these default values must call these functions themselves.

For typical generic programming with macros, a bunch of lists of fields or expressions is often used. Figure 3.1 shows an example of code deriving `Show` using macros.

3.2 Scala 3's Mirror

The second way to access information about the fields of a type and the types of these fields is through what Scala 3 calls `Mirror`. `Mirror` is a trait containing path-dependent types of tuples and singleton types. These types contain the field names and types of the type the `Mirror` is for. A function can request a `Mirror` for a type using normal Scala implicits.

For a product type `A`, `Mirror[A]` also provides a function to convert a `Product` into a `A`. This function is unsafe, as it assumes the size is correct. The `Mirror` does not provide any functions to access the fields of the product type. Instead, users can access this from functions in `Product` that all product types inherit from. For sum types, `Mirror` provides a function giving the ordinal of the type.

Scala 2 did not have `Mirror`, and all generic programming in Scala 2 use macros or is built on pieces that use macros.

Figure 3.2 shows an example of how to derive `Show` using `Mirror`.

```scala
import scala.quoted.*

inline def deriveShow[A]: Show[A] = ${ deriveShowImpl[A] }

def deriveShowImpl[A: Type](using q: Quotes): Expr[Show[A]] =
  val aStr = Expr(Type.show[A])
  // One could also summon a Mirror here and use information gotten from that,
  // but this shows better what it is like to write macros
  import q.reflect.*
  val aRepr = TypeRepr.of[A]
  val aSym = aRepr.classSymbol.getOrElse(
    report.errorAndAbort(s"${Type.show[A]} is not a class type")
  )
  val aFieldsShowedOpt = aSym.declaredFields.map { f =>
    val fieldType = aRepr.select(f).widenTermRefByName
    fieldType.asType match
      case '[a] =>
        Expr
          .summon[Show[a]]
          .toRight(s"Instance of ${Type.show[Show[a]]} not found")
          .map { instance =>
            (e: Expr[A]) => '{$instance.show(${Select(e.asTerm, f).asExprOf[a]})}
          }
  }

  val aFieldsShowed =
    if aFieldsShowedOpt.exists(_.isLeft)
    then
      report.errorAndAbort(aFieldsShowedOpt.collectFirst { case Left(e) => e }.get)
    else aFieldsShowedOpt.map(_.right.get)

  val aFieldsStr = (e: Expr[A]) =>
    aFieldsShowed
      .map(f => f(e))
      .foldLeft(Expr(""))((acc, s) => '{$acc + ", " + $s })

  '{ new Show[A] {
       extension (a: A) def show: String = s"${$aStr}(${${ aFieldsStr('{ a }) }})"
     }
  }
```

Figure 3.1: A macro deriving an instance of Show for a product type.

```scala
import scala.deriving.Mirror
import scala.compiletime.{erasedValue, summonInline}
import scala.reflect.ClassTag

inline def summonShowInstances[A <: Tuple]: List[Show[Any]] =
  inline erasedValue[A] match
    case _: (h *: t) =>
      summonInline[Show[h]].asInstanceOf[Show[Any]] :: summonShowInstances[t]
    case _: EmptyTuple => Nil


inline def deriveShow[A <: Product](
  using m: Mirror.ProductOf[A], c: ClassTag[A]
): Show[A] =
  val instances = summonShowInstances[m.MirroredElemTypes]

  new Show[A]:
    extension (value: A) def show: String =
      val fields =
        value
          .productIterator
          .zip(instances)
          .map((f, instance) => instance.show(f))
          .mkString(", ")
      s"${c.runtimeClass.getSimpleName}($fields)"
```

Figure 3.2: Code deriving instances of Show using Mirror.

3.3 List-based generic programming

For the simplest kind of generic programming, one does not need to concern oneself with macros or `Mirror`, just lists.

When all the field types of a product type are the same, one can store the values of this product type in a list. For example, consider a 3D vector, defined as such, with functions to convert to and from the list form, and another list containing the names of the fields of the type.

```
1 case class Vector3(x: Double, y: Double, z: Double)
2 object Vector3:
3   def toList(vector: Vector3): List[Double] = List(vector.x, vector.y, vector.z)
4   def fromList(vector: List[Double]): Vector3 =
5     Vector3(vector(0), vector(1), vector(2))
6
7   val fieldNames: List[String] = List("x", "y", "z")
```

Using this information, one can transform `Vector3`s in various ways. For example, Figure 3.3 shows a definition of `Show` using the defined members of `Vector3`. `Show` and some instances of this typeclass are also shown and will be used in further examples. In the example, the values of `Vector3` are converted to the list form, and zipped together with the field names. This list of field names and values is then folded over and combined field by field. The resulting code would for the value `Vector3(1, 2, 3)` return `Vector3(, x = 1, y = 2, z = 3)`. While the extra comma and space after the left parenthesis are undesirable, they are kept as it makes the code easier to explain. Future code examples will also have this bug.

It might have been easier to define an instance for `Show[Vector3]` by hand instead of using members of `Vector3` as in Figure 3.3. The advantage of the approach in Figure 3.3 is that the code never relies on anything intrinsic about `Vector3` other than that all of its fields were of the same type.

The code in Figure 3.3 shows the basic structure for some approaches of generic programming, which will be covered in further sections. There are two functions that convert to and from an intermediary representation, called the generic representation [21]. One can perform operations on this intermediary representation and convert it back to

```scala
// Definition of show
trait Show[A]:
  extension (a: A) def show: String

object Show:
  given showDouble: Show[Double] = (d: Double) => d.toString
  given showInt: Show[Int] = (i: Int) => i.toString
  given showString: Show[String] = (s: String) => s
  given showBoolean: Show[Boolean] = (b: Boolean) => b.toString

// Defining an instance of Show for Vector3
given vector3ShowGen: Show[Vector3] with
  extension (a: Vector3) def show: String =
    val fieldsWithNames =
      Vector3.toList(a).zip(Vector3.fieldNames)
        .foldLeft("") { case (acc, (value, fieldName)) =>
          s"$acc, $fieldName = $value"
        }

    s"Vector3($fieldsWithNames)"
```

Figure 3.3: Deriving an instance of Show using members in Vector3's companion object.

the type one is interested in. How one operates on this intermediary representation and what operations one can do is what generally set the different techniques for generic programming apart from each other.

The functions to convert to and from the generic representation, together with some other members that depend on the approach, can be packaged together into a typeclass to encode these operations. I will call these typeclasses and their instances generic typeclasses and generic instances.

Mirror could be said to be a generic typeclass in the loosest sense of the idea. It does not have a function to convert a value to the generic representation, but otherwise it works mostly like a generic typeclass.

A generic typeclass can also be defined for the `Vector3` example shown above. Figure 3.4 shows such a typeclass called `ListGeneric`, in addition to an implementation for `Vector3`. In `ListGeneric`, the type `A` is the type being abstracted over. The type `Inner` is the type of the fields. In the case of `Vector3`, `Inner` is `Double`. `Inner` is a type member as it is not generally something a consumer of `ListGeneric` needs to specify. `ListGeneric` contains two methods to convert to and from lists. Lastly, `ListGeneric` contains members with the names of the fields and the name of the type being worked on.

```scala
trait ListGeneric[A]:
  type Inner
  def to(a: A): List[Inner]
  def from(list: List[Inner]): A

  def fieldNames: List[String]
  def typeName: String

given ListGeneric[Vector3] with
  type Inner = Double
  def to(a: Vector3): List[Double] = Vector3.toList(a)
  def from(list: List[Double]): Vector3 = Vector3.fromList(list)

  val fieldNames: List[String] = Vector3.fieldNames
  val typeName: String = "Vector3"
```

Figure 3.4: ListGeneric, a generic typeclass using List.

```scala
def deriveShow[A](
  using gen: ListGeneric[A], innerShow: Show[gen.Inner]
): Show[A] = new Show[A]:
  extension (a: A) def show: String =
    val fieldsWithNames =
      gen.to(a).zip(gen.fieldNames)
        .foldLeft("") { case (acc, (value, fieldName)) =>
              s"$acc, $fieldName = ${value.show}"
        }

    s"${gen.typeName}($fieldsWithNames)"
end deriveShow
```

Figure 3.5: Deriving show using ListGeneric.

Figure 3.5 shows `deriveShow`, an example of how `ListGeneric` can be used to derive `Show`. The logic of what is happening is the same as with `vector3ShowGen`, but using `ListGeneric` instead of referring to the members in the companion object. `deriveShow` also requires an instance of `Show` for the inner type the code operates with, as calling `toString` on values might be undesirable.

3.3.1 Summary of generic programming using Lists

Generic programming as list manipulation has both advantages and disadvantages. Clear advantages of this form of generic programming are its simplicity and the range of operations one can perform. Program code deals with one or more lists of values and can operate on these lists just like on any other list of values. One can zip two lists together,

flatmap the elements in the lists, fold the list, and more. Regarding the disadvantages, the biggest one is that all list elements must have the same type. The second disadvantage of this solution is that it is not safe. Nothing stops one from passing in a list that is too long or short in `ListGeneric#from`. For this implementation for `Vector3`, an exception is thrown if the list is too short, and elements are ignored if the list is too long.

3.4 HList based generic programming and shapeless 2

The above described approach to generic programming works decently for types like `Vector3` where the types of all the fields are the same, but not for types where fields have different types. The approach is also unsafe as it does not check the length of lists until runtime. To solve these problems, this section looks at how to represent product types at the type level. The generic programming library shapeless 2 [33] uses this representation. A more detailed account on HList based generic programming and shapeless 2 can be found in [21].

3.4.1 HLists

To allow different types, and to check the list lengths at compile time, shapeless 2 uses heterogeneous lists (HLists). These special lists can contain multiple different types of elements and store the type of each element of the list at the type level. Because of this fact, HLists also implicitly store their length at the type level. The code below shows how one can define an HList. While the code shows the idea of HLists, the code deviates a bit from shapeless 2 for simplicity.

```
1 sealed trait HList
2 case class ::[H, T <: HList](head: H, tail: T) extends HList:
3   def ::[H2](h: H2): H2 :: H :: T = new ::(h, this)
4
5 case object HNil extends HList:
6   def ::[H](h: H): H :: HNil = new ::(h, this)
7 type HNil = HNil.type
```

Like normal lists, Hlists have a cons case to store an element and a nil case to indicate the end of the list. Unlike in a normal list, where cons and nil do not appear at the type level, here they do. The base `HList` type also contains no type information to indicate what types it might contain. The type alias for `HNil` is for convenience, to remove the need to write `HNil.type`.

Here is a demonstration of converting between an `HList` and a normal case class.

```
case class ISB(i: Int, s: String, b: Boolean)
object ISB:
  type HListRep = Int :: String :: Boolean :: HNil

  def toHList(isb: ISB): HListRep = isb.i :: isb.s :: isb.b :: HNil

  def fromHList(rep: HListRep): ISB =
    val i :: s :: b :: HNil = rep
    ISB(i, s, b)
```

Note how the type of the HList representation and the value of the HList mirror each other. One could not accidentally flip `s` and `b` without also flipping their types. One could also not forget to include one of the fields at the value level unless it was also forgotten at the type level.

shapeless 2 uses HLists as its generic representation, and its generic typeclass is called `Generic`, defined like this:

```
trait Generic[A]:
  type Repr
  def to(a: A): Repr
  def from(repr: Repr): A
```

There is no upper bound on the `Repr` type, so that `Generic` can also work for sum types. Sum types and shapeless 2 is covered in Section 5.1. shapeless 2 does not include the name of the type in `Generic`. The name of a type is instead contained in a type called `Typeable`. This type also contains a method to cast an arbitrary value to the given type if the value is of a type that matches. `Typeable` is defined like this:

```scala
trait Typeable[A]:
  def describe: String
  def cast(any: Any): Option[A]
```

Field names are handled separately and can be found in a different trait, which will be covered later.

3.4.2 Type level programming with HLists

Any operation on HLists is implemented with type level programming and induction. One defines the base case on `HNil` and the recursion step on `H :: T` as `given` instances.

When deriving typeclasses, there is also another instance that ties the `HList` based generic representation to the actual type using a `Generic` and a typeclass instance for the generic representation. The Scala compiler will then perform the induction steps until completion. Here is an example for the `Show` type:

```scala
given hnilShow: Show[HNil]:
  extension (a: HNil.type) def show: String = ""

given hconsShow: [H, T <: HList](using Show[H], Show[T]): Show[H :: T]:
  extension (a: H :: T) def show: String = s"${a.head.show}, ${a.tail.show}"

def deriveShowGeneric[A](
  using gen: Generic[A], typeable: Typeable[A], instance: Show[gen.Repr]
): Show[A] = new Show[A]:
  extension (a: A) def show: String = s"${typeable.describe}(${gen.to(a).show})"
```

Figure 3.6 An example of how a call to `deriveShowGeneric` using `ISB` could be expanded.

The Scala compiler automatically constructs the `given` instance needed. The `Generic` and `Typeable` would be generated from Scala 3's Mirror or with a macro.

```scala
given gen: Generic[ISB] with
  type Repr = Int :: String :: Boolean :: HNil
  def to(a: ISB): Repr = a.i :: a.s :: a.b :: HNil
  def from(repr: Repr): ISB = repr match
    case i :: s :: b :: HNil => ISB(i, s, b)

given typeable: Typeable[ISB] with
  def describe: String = "ISB"
  def cast(any: Any): Option[A] = any match
    case isb: ISB => Some(isb)
    case _ => None

deriveShowGeneric[ISB](
  using gen,
  typeable,
  hconsShow[Int, String :: Boolean :: HNil](
    using Show.showInt,
    hconsShow[String, Boolean :: HNil](
      using Show.showString,
      hconsShow[Boolean, HNil](using Show.showBoolean, hnilShow)
    )
  )
)
```

Figure 3.6: An expansion of deriveShowGeneric using ISB.

3.4.3 Labels

In the previous example, the Show derivation is not labeled. The field values are included, but not their names. To allow for dealing with the field names of a type, shapeless 2 defines the generic typeclass `LabelledGeneric`. This type works almost like `Generic`, except that the type of the HList elements includes the field names in a tagged type. Its definition is shown below:

```scala
trait LabelledGeneric[A]:
  type Repr
  def to(a: A): Repr
  def from(repr: Repr): A

object LabelledGeneric:
  opaque type Labelled[A, S <: String] <: A = A
  object Labelled:
    inline def of[A, S <: String](a: A): Labelled[A, S] = a

export LabelledGeneric.Labelled
```

`Labelled` is an opaque type that allows one to tag a type with a string at the type level. The opaque type has a bound, so one can still treat it as the unlabelled type. The intention with the label is to use singleton types. Singleton types are types inhabited by only a single value, like `"foo"` or `5`. The type `ISB` from above would have the following `LabelledGeneric#Repr` type.

```
type ISBLabelledRepr =
  Labelled[Int, "i"] :: Labelled[String, "s"] :: Labelled[Boolean, "b"] :: HNil
```

The `ValueOf` typeclass, provided by the Scala standard library, can be used to get the value a singleton type represents. This typeclass can be used to access the tagged value at runtime.

Putting the above tools together, the code below shows how to derive instances of `Show` using `LabelledGeneric`. There is no need to define a new instance for `HNil`, as it does not contain label information.

```
given hconsLabelledShow: [H, S <: String, T <: HList](
  using Show[H], Show[T]
)(using s: ValueOf[S]): Show[Labelled[H, S] :: T]:
  extension (a: Labelled[H, S] :: T) def show: String =
    s"${s.value} = ${a.head.show}, ${a.tail.show}"

def deriveShowLabelledGeneric[A](
  using gen: LabelledGeneric[A], typeable: Typeable[A], instance: Show[gen.Repr]
): Show[A] = new Show[A]:
  extension (a: A) def show: String = s"${typeable.describe}(${gen.to(a).show})"
```

Figure 3.7 An example of how a call to `deriveShowLabelledGeneric` using `ISB` could be expanded.

3.4.4 The hidden fold

In Section 3.3 that discussed orinary lists, there was a nice `foldLeft` function call in the function `deriveShow`, while there is nothing like that visible in `deriveShowGeneric` or `deriveShowLabelledGeneric`. Instead, the Hlist traversing manually matches over the

```scala
given gen: LabelledGeneric[ISB] with
  type Repr =
    Labelled[Int, "i"] :: Labelled[String, "s"] :: Labelled[Boolean, "b"] :: HNil

  def to(a: ISB): Repr =
    Labelled.of[Int, "i"](a.i)
      :: Labelled.of[String, "s"](a.s)
      :: Labelled.of[Boolean, "b"](a.b)
      :: HNil

  def from(repr: Repr): ISB = repr match
    case i :: s :: b :: HNil => ISB(i, s, b)

given typeable: Typeable[ISB] with
  def describe: String = "ISB"
  def cast(any: Any): Option[A] = any match
    case isb: ISB => Some(isb)
    case _ => None

deriveShowLabelledGeneric[ISB](
  using gen,
  typeable,
  hconsLabelledShow[
    Int,
    "i",
    Labelled[String, "s"] :: Labelled[Boolean, "b"] :: HNil
  ](
    using Show.showInt,
    hconsLabelledShow[String, "s", Labelled[Boolean, "b"] :: HNil](
      using Show.showString,
      hconsLabelledShow[Boolean, "b", HNil](
        Show.showBoolean, hnilShow
      )(new ValueOf["b"]("b"))
    )(new ValueOf["s"]("s"))
  )(new ValueOf["i"]("i"))
)
```

Figure 3.7: An expansion of deriveShowLabelledGeneric using ISB.

```scala
import scala.compiletime.{erasedValue, summonInline}

inline def deriveShowHListInline[Repr <: HList]: Show[Repr] =
  inline erasedValue[Repr] match
    case _: h :: t =>
      val headShow = summonInline[Show[h]]
      val tailShow = deriveShowHListInline[t]
      val sh = new Show[h :: t]:
        extension (a: h :: t) def show: String =
          s"${headShow.show(a.head)}, ${tailShow.show(a.tail)}"

      sh.asInstanceOf[Show[Repr]]

    case _: HNil   => hnilShow.asInstanceOf[Show[Repr]]

inline def deriveShowGenericInline[A](
  using gen: Generic[A], typeable: Typeable[A]
): Show[A] =
  given Show[gen.Repr] = inline erasedValue[gen.Repr] match
    case _: HList =>
      deriveShowHListInline[gen.Repr & HList].asInstanceOf[Show[gen.Repr]]

  new Show[A]:
    extension (a: A) def show: String = s"${typeable.describe}(${gen.to(a).show})"
```

Figure 3.8: A rewriting of deriveShowGeneric using Scala 3's inline features.

types. If one rewrites `deriveShowGeneric` using Scala 3's inline features, as was done in Figure 3.8, the logic becomes more apparent, making it easier to understand what is happening.

While there is no `foldLeft` function call in Figure 3.8 either, the resulting code looks something like a `foldRight` expansion. That is to say (with a bit of code that would not actually be valid), the code could be rephrased as a `foldRight` function and a call to this function, as was done in Figure 3.9. The reason the code in Figure 3.9 would not be valid is that `summonInline` would be expanded too early and with the wrong type. The code would try to look for a `Show[Tpe]`. Such an instance does not exist as `Tpe` would vary from invocation to invocation.

3.4.5 Summary of generic programming using HLists

Metaprogramming with `HLists` solves all the problems that manifested with generic programming using ordinary lists. The size of `HLists` are checked at compile time, and they

```scala
//FoldRight function on HLists over typeclasses
extension [L <: HList](l: L)
  def foldRightTC[TC[_]](base: TC[HNil])(
    f: [Tpe, Acc <: HList] => (Tpe, TC[Acc]) => TC[Tpe :: Acc]
  ): TC[L]

def deriveShowHListInline[Repr <: HList](repr: Repr): Show[Repr] =
  repr.foldRightTC(hnilShow) { [Tpe, Acc <: HList] => (tpe: Tpe, acc: Show[Acc]) =>
    val headShow = summonInline[Show[Tpe]]
    new Show[Tpe :: Acc]:
      extension (a: Tpe :: Acc) def show: String =
        s"${headShow.show(a.head)}, ${acc.show(a.tail)}"
  }
```

Figure 3.9: A further rewriting of deriveShowGeneric using Scala 3's inline features and some illegal code to expose a foldRight function.

can contain arbitrary types. Sadly, `HLists` are not without problems either, most of which they inherit from the logic programming approach they take.

First, the logic programming style is generally less familiar to many Scala programmers, requiring them to learn another style of programming besides functional and object oriented programming. Scala is also poorly suited for logic programming: it requires tons of boilerplate and extra definitions that would not be needed in, e.g., Prolog. Scala 3 has improved this situation, but there is still room to improve.

Second, code written with this approach can take a long time to compile. Scala is already often said to be a language that takes a long time to compile, most time is usually spent typechecking code. This is a space shapeless can often make worse. There are guides and tools [2] to help to write programs in ways that are faster to compile, but that is yet another thing developers need to know about.

Lastly, debugging implicits can often be confusing and, not so infrequently, the compiler's diagnostics boils down to "something went wrong," leaving the developer to figure out what and why.

3.5 Generic programming in shapeless 3

With Scala 3 came new features for generic programming like polymorphic functions and `Mirror`, that unlocked new possibilities. While the old techniques for generic programming still worked (with some exceptions involving compiler bugs [13]), the macros that

they were built upon did not. From this situation, shapeless 3 [34] was born, taking advantage of these new features.

3.5.1 Polymorphic functions

A significant new feature for generic programming was polymorphic functions. Scala 2 had no explicit syntax to express functions like `List#head` as values. Before Scala 3, one had to define a trait with a polymorphic apply method and construct instances of this trait. There is a compiler plugin called Kind Projector [32] that, among other things, made this slightly easier to write with dedicated syntax. Scala 3 added its own syntax for polymorphic functions to the language. This syntax is more expressive than Kind Projector's syntax was.

```
1  //As a function
2  def head[A](xs: List[A]): A = xs.head
3
4  //Old way to get a value
5  trait FunctionK[A[_], B[_]]:
6    def apply[Z](z: A[Z]): B[Z]
7  type Id[A] = A
8  val headOld: FunctionK[List, Id] = new FunctionK[List, Id]:
9    def apply[Z](z: List[Z]): Z = z.head
10
11 //Old way using Kind projector
12 //There is no way to refer to the type A here
13 val headOldKindProjector: FunctionK[List, Id] =
14   Lambda[FunctionK[List, Id]](xs => xs.head)
15
16 //New way
17 val headNewWay: [A] => List[A] => A = [A] => (xs: List[A]) => xs.head
```

3.5.2 ProductInstances

The shapeless 3 library uses Scala's new polymorphic functions to add a new higher level interface for generic programming for simpler use cases called `ProductInstances`. The basic idea seems to be to take the `foldRight` function that could be seen in shapeless 2 with enough rewriting, simplification, and ignoring some language restrictions, and make

```scala
type CompleteOr[T] = T | Complete[T]
case class Complete[T](t: T)

extension [F[_], T](inst: ProductInstances[F, T])
  inline def construct(f: [t] => F[t] => t): T

  inline def map(x: T)(f: [t] => (F[t], t) => t): T
  inline def map2(x: T, y: T)(f: [t] => (F[t], t, t) => t): T

  inline def foldLeft[Acc](x: T)(i: Acc)(
    f: [t] => (Acc, F[t], t) => CompleteOr[Acc]
  ): Acc
  inline def foldLeft2[Acc](x: T, y: T)(i: Acc)(
    f: [t] => (Acc, F[t], t, t) => CompleteOr[Acc]
  ): Acc

  inline def foldRight[Acc](x: T)(i: Acc)(
    f: [t] => (F[t], t, Acc) => CompleteOr[Acc]
  ): Acc
  inline def foldRight2[Acc](x: T, y: T)(i: Acc)(
    f: [t] => (F[t], t, t, Acc) => CompleteOr[Acc]
  ): Acc

  inline def project[R](t: T)(p: Int)(f: [t] => (F[t], t) => R): R
```

Figure 3.10: API exposed by ProductInstances

it a real thing. shapeless 3 does this with several different functions. shapeless 3 achieves this by specifying the typeclass to work with beforehand and then provides instances of that typeclass for each field in the polymorphic functions found in shapeless 3's API.

?? shows some of the operations offered by `ProductInstances`. `construct` can be used to construct each of the fields of a type from the typeclass of that field, and from those fields, construct the type `T`. `map` and `map2` allow transforming each of the fields of one or two values together with a typeclass of the field into a new value of that field, and constructing a new value from that. `foldLeft`, `foldLeft2`, `foldRight` and `foldRight2` allow folding over the fields of one or two values together with the typeclass for the field. Iteration can be finished early by wrapping the accumulated value in `Complete`. `project` allows indexing into the fields of a value, and performing some value on that field and the typeclass of that field.

`ProductInstances` is very much a black box to the user, with little information on how it does the operations it exposes and little room for adding new functions.

Figure 3.11 shows an example of how to use `ProductInstances` to derive a `Show` instance. The fields of the values are left folded. The labels are converted to an iterator

```scala
def deriveShowProductInstances[A](
  using inst: ProductInstances[Show, A],
  labels: Labelling[A]
): Show[A] = new Show[A]:
  extension (a: A) def show: String =
    val elemLabelsIt = labels.elemLabels.iterator

    val elems = inst.foldRight(a)("")(
      [t] => (sh: Show[t], value: t, acc: String) =>
        s"${elemLabelsIt.next} = ${sh.show(value)}, $acc"
    )
    s"${labels.label}($elems)"
```

Figure 3.11: Deriving instances of Show using ProductInstances.

and accessed in the function with `next`. Care must be taken to ensure the iterator and iteration over the value does not get out of sync.

In shapeless 3, the type name and field names are contained in the type `Labelling`. This type contains a `Seq[String]`, which in turn contains the element labels.

3.5.3 Summary of generic programming using shapeless 3

The shapeless 3 library solves the significant problems that shapeless 2 has, namely obscure code, slow compile times and poor support for debugging. For simple cases with shapeless 3, there is no logic programming to obscure what is happening, slow down compile times, and to make debugging harder. A call to something like a `foldRight` that got lost in the transition from `Lists` to `HLists` in shapeless 2 has returned.

shapeless 3 also has the advantage of exposing an API for performing generic programming, and not a datastructure like shapeless 2 did. Because of this, shapeless 3 can optimize the functions found in `ProductInstances` while user code continues to work without recompilation. For example, the value used for the labels can go from a `List` to a `Vector` without breaking any programs.

Lastly, shapeless 3 also allows for expressing generic programming on more kinds with little extra code. shapeless 3 has definitions and functions not just for deriving typeclasses with kind `F[_]`, but also `F[_[_]]` and `F[_, _]`.

One of the problems with the `List` approach that HLists solved, has come back: type safety. `Labelling` exposes a `Seq[String]` as the labels. This type does not interact

in a type safe way with `ProductInstances`. In `deriveShowProductInstances` above, an interator is used to access each label in turn. Care must be taken to make sure the iterator is in lockstep with the iteration over the data structure `ProductInstances` offers functions to operate on. Another option here is to rely on `ProductInstances#project` together with the indices obtained from the labels, but this is also unsafe, as `project` operates with `Int`. There is nothing stopping code from accidentially supplying an index that is out of bounds for what is expected.

Another problem with the approach `ProductInstances` takes is that the types of operations a developer can do is restricted. First off, only a single call to the functions on `ProductInstances` can be used. These functions cannot be chained together in a way similar to how a developer might chain together `flatMap`, `zip`, `foldLeft` and more. Next, the number of values the functions exposed by `ProductInstances` can operate on is limited. This fact becomes clear as there are both functions `foldLeft` and `foldLeft2` that operate on one or two values. If a user wants to operate on three values, they can't. Because of this restriction, the library developer hass to think of use cases a user might have, and move to cover these use cases explicitly. There are workarounds for this restriction using auxillary typeclasses, and using multiple instances of `ProductInstances`, but I will not cover those workarounds here.

Chapter 4

perspective

perspective [17] is a new library for generic programming in Scala. perspective is written for both Scala 2 and Scala 3, but has more features in Scala 3.

Like shapeless 3, perspective wants clear and simple higher order functions visible in the code that make use of polymorphic functions. Unlike shapeless 3, perspective does not wish to hide the data operated on in a black box. perspective instead looks to where these higher order functions come from, namely typeclasses describing these functions, like Functor, Applicative, Foldable, Traversable, and more. By encoding higher kinded versions of these typeclasses, perspective establishes a language to express generic programming in a way that is easy to understand and expressible. perspective does this by using higher kinded types, higher kinded data, and higher kinded typeclasses to operate on the data it abstracts over.

4.1 perspective's typeclasses

perspective is built on typeclasses. This section will cover the ones perspective uses, how they relate to one another, what they are useful for, and their laws. Most of the discussion of these typeclasses will relate to how they are useful for generic programming. I will also show inductive proofs for defining instances of these typeclasses for product types of arbitrary sizes.

4.1.1 Foldable

One class of fundamental operations in datatype-generic programming are folds over the elements of different data types. This is what the thesis has focused on so far, so folds will be covered first. Folds come from the typeclass `Foldable`, looking something like this:

```
1 trait Foldable[F[_]]:
2   extension [A](fa: F[A]) def foldLeft[B](b: B)(f: (B, A) => B): B
```

While lacking functions like `foldRight` and `foldMap`, which can be found in, for example, Cats [38], the core functionality is the same. You could, for example, easily define an instance like `Foldable[List]`. This definition would, however, not work for perspective's use case. In this definition, `F` is a type taking a single type argument, which means that all the values to fold over must be of that type. This is the same problem that came up with `List` based generic programming. The container's element types had to all be of the same type.

To be able to fold over heterogeneous data, one needs to parameterize the data to work on with a higher kinded type, as in the example below.

```
1 //No parameterizing
2 case class Isb(i: Int, s: String, b: Boolean)
3
4 //Higher kinded type parameterized
5 case class IsbK[F[_]](i: F[Int], s: F[String], b: F[Boolean])
```

This type of definition, where each value in a type is applied with the same higher kinded type, is called higher kinded data.

Here, `Isb` is monomorphic data, and `IsbK` is higher kinded. One can represent the values in `Isb` with a special type called `Id`, defined like this: `type Id[A] = A`. The type `Id` maps all types to themselves. It is like the identity function at the type level. `IsbK[Id]` then is a type where the fields are just normal values, containing `Int`, `String` and `Boolean`. Because of the above facts, `Isb` and `IsbK[Id]` are isomorphic. One can convert between them without loss. Whenever higher kinded data needs to represent values without anything else, `Id` is used.

```
def isbToIsbK(isb: Isb): IsbK[Id] = IsbK(isb.i, isb.s, isb.b)
def isbKToIsb(isb: isbK[Id]): Isb = Isb(isb.i, isb.s, isb.b)
```

A new `Foldable` type that handles higher kinded data can be defined like this:

```
trait FoldableK[F[_[]]]:
  extension [A[_]](fa: F[A]) def foldLeftK[B](b: B)(f: [Z] => (B, A[Z]) => B): B
```

`F` here gained another kind. `A` also became higher kinded. A polymorphic function was also employed instead of a normal one. These are the general rules when converting a typeclass to operate on higher kinds. Types become kinds (`A -> A[_]`), kinds become higher kinded kinds (`A[_] -> A[_[_]]`), and functions are lifted to operate on the higher kinded types. `(A => B) -> ([Z] => A[Z] => B[Z])`.

Conventions in perspective

In perspective, higher kinded variants of typeclasses and functions are declared with a `K` suffix. perspective also defines several type aliases (most infix) to allow easier writing of higher kinded typeclasses. These aliases can be seen in Figure 4.1.

`FunctionK` is a polymorphic function from one higher kinded type to another, with `~>:` being an inline syntax for such functions. `Const` is a special type constructor that always has the value passed in. It allows one to convert higher kinded data into fixed sized lists of a specific type. For example, `IsbK[Const[String]]` is like a fixed size list with three string elements. perspective uses special arrows with `#` to indicate `FunctionK`s where that side of the arrow is `Const`. As an example `A ~>#: B` is equal to `A ~>: Const[B]`. This function takes values of types taking a type, like `F[_]` and return values of types not taking a type. `A #~>#: B` and `A => B` are isomorphic.

The perspective library also defines higher kinded typeclasses with kind `F[_[_], _]`, not `F[_[_]]` as seen above. There are arguments for both forms, but perspective goes with the first as it makes some typeclasses easier to understand. To work with types like `IsbK` above, perspective defines aliases for the typeclasses that do not use the second type argument using the type `IgnoreC`, defined like this: `type IgnoreC[F[_[_]]] = [A[_], _] =>> F[A]`. Typeclasses applied with this type have the

```scala
/** A higher kinded type that ignores its second type parameter. */
type Const[A] = [_] =>> A

type FunctionK[A[_], B[_]] = [Z] => A[Z] => B[Z]
object FunctionK:
  def identity[A[_]]: A ~>: A = [Z] => (a: A[Z]) => a

infix type ~>:[A[_], B[_]] = FunctionK[A, B]

/** A FunctionK returning a [[Const]] type. */
infix type ~>#:[F[_], R] = F ~>: Const[R]

/** A FunctionK taking a [[Const]] type. */
infix type #~>:[T, F[_]] = Const[T] ~>: F

/** A FunctionK taking and returning [[Const]] types. */
infix type #~>#:[T, R] = Const[T] ~>: Const[R]
```

Figure 4.1: Special types used by perspective.

`C` suffix. In the end, the choice between `F[_[_], _]` and `F[_[_]]` is a fairly meaningless one, especially because a lot of generic derivation using perspective currently only makes use of typeclasses with the `C` suffix.

Sometimes Scala's type inference is not powerfuil enough for perspective's definitions. In these cases, perspective provides special preapplied functions that make the expression's types more explicit to help type inference. This is mainly seen when `Id` and `Const` are involved.

Foldable in perspective

With the above building blocks and conventions in place, here is how perspective defines `FoldableK`:

```scala
trait FoldableK[F[_[_], _]]:
  extension [A[_], C](fa: F[A, C])
    def foldLeftK[B](b: B)(f: B => A ~>#: B): B

    def foldMapK[B](f: A ~>#: B)(using B: Monoid[B]): B =
      foldLeftK(B.empty)(b => [Z] => (az: A[Z]) => b.combine(f(az)))

  extension [A, C](fa: F[Const[A], C])
    def toListK: List[A] = fa.foldMapK(FunctionK.liftConst(List(_: A)))

type FoldableKC[F[_[_]]] = FoldableK[IgnoreC[F]]
```

Foldable's laws

Foldable's laws basically ensure that the above functions are consistent. The laws are not particularly strong. Here they are [39].

FoldLeft consistency:

```
val m = summon[Monoid[B]]
fa.foldMapK(f) ==
  fa.foldLeftK(m.empty)(b => [Z] => (a: A[Z]) => b.combine(b, f(a)))
```

ToList consistent:

```
fa.toListK ==
  fa.foldLeftK(
    mutable.ListBuffer.empty[A]
  )(buf => [Z] => (a: A[Z]) => buf += a).toList
```

Order consistent:

```
if fa == fb then fa.toListK == fb.toListK
else true
```

As an example of an instance of the `FoldableK` typeclass, here is an inductive definition for foldable over products of arbitrary size.

```
1  //Base case
2  given [X]: FoldableKC[[A[_]] =>> A[X]] with
3    extension [A[_], C](fa: A[X]) def foldLeftK[B](b: B)(f: B => A ~>#: B): B =
4      f(b)(fa)
5
6  //Induction step
7  given [X1[_[_]]: FoldableKC, X2]: FoldableKC[[A[_]] =>> (X1[A], A[X2])] with
8    extension [A[_], C](fa: (X1[A], A[X2]))
9      def foldLeftK[B](b: B)(f: B => A ~>#: B): B =
10       val b1 = fa._1.foldLeftK(b)(f)
11       f(b1)(fa._2)
```

To prove that this definition obeys the above laws, we can first notice that foldLeft consistency is by definition obeyed as we have not redefined `foldMapK`. We then analyze the base case and the induction step.

For the base case, we get:

- ```
  fa.toListK ==
    fa.foldLeftK(
      mutable.ListBuffer.empty[A])(buf => [Z] => (a: A[Z]) => buf += a
    ).toList
  ```
- Unwrapping toListK

  ```
  fa.foldMapK(FunctionK.liftConst(List(_: A))) ==
    fa.foldLeftK(
      mutable.ListBuffer.empty[A])(buf => [Z] => (a: A[Z]) => buf += a
    ).toList
  ```
- Unwrapping foldMapK

  ```
  fa.foldLeftK(Nil: List[A])(b => [Z] => (az: A[Z]) => b.combine(List(az: A))) ==
    fa.foldLeftK(
      mutable.ListBuffer.empty[A])(buf => [Z] => (a: A[Z]) => buf += a
    ).toList
  ```
- Unwrapping foldLeftK

  ```
  Nil.combine(List(fa: A)) ==
    (mutable.ListBuffer.empty[A] += fa).toList
  ```
- Simplifying list construction

  ```
  List(fa) == List(fa)
  ```

From this, it follows that if there are two values `fa` and `fb` and `fa == fb` holds, then `fa.toListK == fb.toListK`, which is equivalent to `List(fa) == List(fb)`, must also hold.

For the induction step, we get:

- ```
  fa.toListK ==
    fa.foldLeftK(
      mutable.ListBuffer.empty[A]
    )(buf => [Z] => (a: A[Z]) => buf += a).toList
  ```
- Unwrapping toListK

  ```
  fa.foldMapK(FunctionK.liftConst(List(_: A))) ==
    fa.foldLeftK(
      mutable.ListBuffer.empty[A]
    )(buf => [Z] => (a: A[Z]) => buf += a).toList
  ```
- Unwrapping foldMapK

```
fa.foldLeftK(Nil: List[A])(b => [Z] => (az: A[Z]) => b.combine(List(az: A))) ==
  fa.foldLeftK(
    mutable.ListBuffer.empty[A]
  )(buf => [Z] => (a: A[Z]) => buf += a).toList
```

- Unwrapping foldLeftK

```
fa._1.foldLeftK(Nil: List[A])(
  b => [Z] => (az: A[Z]) => b.combine(List(az: A))
).combine(List(fa._2: A)) ==
  (fa._1.foldLeftK(mutable.ListBuffer.empty[A])(
    buf => [Z] => (a: A[Z]) => buf += a
  ) += fa._2).toList
```

- Induction hypothesis and wrapping so everything still makes sense

```
fa._1.foldLeftK(Nil: List[A])(
  b => [Z] => (az: A[Z]) => b.combine(List(az: A))
).combine(List(fa._2: A)) ==
  (ListBuffer.from(
    fa._1.foldLeftK(Nil: List[A])(
        b => [Z] => (az: A[Z]) => b.combine(List(az: A))
    )
  ) += fa._2).toList
```

- `(ListBuffer.from(a) += b).toList` is equivalent to `a.combine(List(b))`

```
fa._1.foldLeftK(Nil: List[A])(
  b => [Z] => (az: A[Z]) => b.combine(List(az: A))
).combine(List(fa._2: A)) ==
  fa._1.foldLeftK(Nil: List[A])(
    b => [Z] => (az: A[Z]) => b.combine(List(az: A))
  ).combine(List(fa._2: A))
```

If we have two tuples `fa` and `fb`, and `fa == fb` holds, then we also know that `fa._1.toListK == fb._1.toListK` holds (induction hypothesis). From this, it also follows that `fa._2 == fb._2`, `List(fa._2) == List(fb._2)` and `fa._1.toListK.combine(List(fa._2)) == fb._1.toListK.combine(List(fb._2))` hold.

4.1.2 Functor

A `Functor` exposes a function to map over a structure. `Functor` is the simplest typeclass we cover. It is defined in Figure 4.2

```scala
trait FunctorK[F[_[_], _]]:
  extension [A[_], C](fa: F[A, C])
    def mapK[B[_]](f: A ~>: B): F[B, C]

    inline def mapConst[B](f: A ~>#: B): F[Const[B], C] =
      mapK(f)

type FunctorKC[F[_[_]]] = FunctorK[IgnoreC[F]]
```

Figure 4.2: perspective's FunctorK.

Functor's laws

All functors must obey the laws of composition and identity [40].

Composition: `fa.mapK(f).mapK(g) = fa.map([Z] => (a: A[Z]) => g(f(a)))`

Identity: `fa.mapK(FunctionK.identity) = fa`

Here is an inductive definition for functors over products of arbitrary size.

```scala
//Base case
given [X]: FunctorKC[[A[_]] =>> A[X]] with
  extension [A[_], C](fa: A[X])
    def mapK[B[_]](f: A ~>: B): B[X] = f(fa)

//Induction step
given [X1[_[_]]: FunctorKC, X2]: FunctorKC[[A[_]] =>> (X1[A], A[X2])] with
  extension [A[_], C](fa: (X1[A], A[X2]))
    def mapK[B[_]](f: A ~>: B): (X1[B], B[X2]) = (fa._1.mapK(f), f(fa._2))
```

The proof that this definition obeys the functor laws is inductive. The base case, requires two subcases, one for each law.

Composition:

- `fa.mapK(f).mapK(g) = fa.mapK([Z] => (a: A[Z]) => g(f(fa)))`
- Unwrapping right mapK:
  `fa.mapK(f).mapK(g) = g(f(fa))`
- Unwrapping first left mapK:
  `f(fa).mapK(g) = g(f(fa))`

- Unwrapping second left mapK:
  `g(f(fa)) = g(f(fa)))`

Identity:

- `fa.mapK(FunctionK.identity) = fa`
- Unwrapping FunctionK.identity
  `fa.mapK([Z] => (a: A[Z]) => a) = fa`
- Unwrapping mapK:
  `fa = fa`

For the induction step, we get:

- `fa.mapK(f).mapK(g) = fa.mapK([Z] => (a: A[Z]) => g(f(fa)))`
- Showing tuple:
  `(x1, x2).mapK(f).mapK(g) = (x1, x2).mapK([Z] => (a: A[Z]) => g(f(fa)))`
- Unwrapping first left mapK:
  `(x1.mapK(f), f(x2)).mapK(g) = (x1, x2).mapK([Z] => (a: A[Z]) => g(f(fa)))`
- Unwrapping second left mapK:
  `(x1.mapK(f).mapK(g), g(f(x2))) = (x1, x2).mapK([Z] => (a: A[Z]) => g(f(fa)))`
- Unwrapping right mapK:
  `(x1.mapK(f).mapK(g), g(f(x2))) = (x1.mapK([Z] => (a: A[Z]) => g(f(fa))), g(f(x2)))`
- Induction hypothesis:
  `(x1.mapK(f).mapK(g), g(f(x2))) = (x1.mapK(f).mapK(g), g(f(x2)))`

Identity:

- `fa.mapK([Z] => (a: A[Z]) => a) = fa`
- Showing tuple:
  `(x1, x2).mapK([Z] => (a: A[Z]) => a) = (x1, x2)`
- Unwrapping mapK: `(x1.mapK([Z] => (a: A[Z]) => a), x2) = (x1, x2)`
- Induction hypothesis:
  `(x1, x2) = (x1, x2)`

```scala
trait ApplyK[F[_[_], _]] extends FunctorK[F]:
  extension [A[_], B[_], C](ff: F[[D] =>> A[D] => B[D], C])
    def ap(fa: F[A, C]): F[B, C] =
      ff.map2K(fa)([Z] => (f: A[Z] => B[Z], a: A[Z]) => f(a))

  extension [A[_], C](fa: F[A, C])
    def map2K[B[_], Z[_]](fb: F[B, C])(f: [X] => (A[X], B[X]) => Z[X]): F[Z, C]

    inline def map2Const[B[_], Z](fb: F[B, C])(
      f: [X] => (A[X], B[X]) => Z
    ): F[Const[Z], C] = fa.map2K(fb)(f)

    def tupledK[B[_]](fb: F[B, C]): F[[Z] =>> (A[Z], B[Z]), C] =
      fa.map2K(fb)([Z] => (fa: A[Z], fb: B[Z]) => (fa, fb))

type ApplyKC[F[_[_]]] = ApplyK[IgnoreC[F]]
```

Figure 4.3: perspective's ApplyK.

4.1.3 Apply

The `Apply` typeclass allows combining several different values in a context into a single value of that context. `Apply` is a useful tool for many generic programming tasks.

Apply is not a well-established typeclass, but rather a simplification of the typeclass called `Applicative`. All instances of `Applicative` are also `Functors`. `Apply` is an `Applicative` without the function `pure`. I separate the typeclasses like this because `Apply` is generally more useful than `Applicative`. Not all types with instances of `Apply` have instances for `Applicative`. Figure 4.3 shows perspective's definition of `ApplyK`.

Deriving Show with Apply and Foldable

At this point, I have gone over everything needed to derive some typeclasses like `Show`. I will show how to derive the typeclass `Show` similarly to how a user might first approach perspective and reason about it. `F[Id]` is always the type used to store values, and will be used as such in this example. For the field names, `F[Const[String]]` will be used, and this is also what is generally used with perspective. `F[Const[List[String]]]` is more general and is used when deriving typeclasses deeply, but perspective shies away from this style of generic programming when doing typeclass derivation. Finally, generally instances of the typeclass being derived also need to exist for the fields of the type the typeclass is being derived for. `F[TC]` is used to store these instances, where `TC` is the

typeclass being derived. In this example, it would be `F[Show]`. As one can see, `F` is used as a container for everything the code touches and manipulates, values, field names, and typeclasses. In other languages like Haskell or Rust, passing around typeclasses as values is less trivial. In Scala, however, typeclasses are first-class citizens. With that in mind, here is the code to derive instances of `Show`:

```
def deriveShowPerspective[F[_]: ApplyK: FoldableK](
  typeName: String, names: F[Const[String]], instances: F[Show]
): Show[F[Id]] =
  new Show[F[Id]]:
    extension (a: F[Id]) def show: String =
      val elems = instances
        .map2Const(a)([Z] => (instance: Show[Z], value: Z) => instance.show(value))
        .map2Const(names)([Z] => (str: String, name: String) => s"$name = $str")
        .foldLeftK("")(acc => [Z] => (str: String) => s"$acc,  $str")

      s"$typeName($elems)"
end deriveShowPerspective
```

As can be seen, the values are operated on like one might operate on values in a list, evem though we are working on arbitrary data structures, not lists. First the instances and the values are mapped together to convert the field values into values of `String`. The string values are then mapped together with the field names to add the field name to the string. Finally everything is folded and combined together. Note that `map2Const` is used instead of `map2K` as the type `String` does not take another type.

It should be noted that while this style of generic programming is easy to understand, it is not particularly efficient as it requires multiple iteration passes over the same data. As such more experienced users of perspective would not derive typeclass instances like this. An alternative is presented in subsection 4.1.8.

Apply's laws

Apply's laws are all the Applicative laws that do not mention `pure`. The only law this involves is associativity [37].

Associativity:

```
fa.tupledK(fb).tupledK(fc) =
  fa.tupledK(fb.tupledK(fc)).mapK{ [Z] => (ft: (A[Z], (B[Z], C[Z]))) =>
          ((ft._1, ft._2._1), ft._2._2)
  }
```

Here is an inductive definition for `Apply` over products of arbitrary size:

```
1  //Base step
2  given [X]: ApplyKC[[A[_]] =>> A[X]] with
3    extension [A[_], C](fa: A[X])
4      def map2K[B[_], Z[_]](fb: B[X])(f: [Y] => (A[Y], B[Y]) => Z[Y]): Z[X] =
5        f(fa, fb)
6
7  //Induction step
8  given [X1[_[_]]: ApplyKC, X2]: ApplyKC[[A[_]] =>> (X1[A], A[X2])] with
9    extension [A[_], C](fa: (X1[A], A[X2]))
10     def map2K[B[_], Z[_]](fb: (X1[B], B[X2]))(
11       f: [X] => (A[X], B[X]) => Z[X]
12     ): (X1[Z], Z[X2]) = (fa._1.map2K(fb._1)(f), f(fa._2, fb._2))
```

The proof of correctness of this definition can be found in Section A.1.

4.1.4 Applicative

Applicative exposes a function to construct a value of a higher kinded data type from a unit function called `ValueK`. Figure 4.4 shows perspective's definition of `ApplicativeK`.

Generally, there are three ways to get a `ValueK`. The first way is to use a constant value, for example, `Unit`. The second way is to use a covariant value, for example, `None` together with the type `Option`. The third way is to use a contravariant value, for example, `(a: Any) => a.toString` together with the type `[A] =>> A => String`.

In general, `Applicative` is not used much and it provides little value as part of perspective. It is presented here primarily for completeness' sake.

```scala
type ValueK[A[_]] = [Z] => () => A[Z]
object ValueK:

  /** Construct a [[ValueK]] from a covariant higher kinded type. */
  def co[A[+_]](covariant: A[Nothing]): ValueK[A] = [Z] => () => covariant

  /** Construct a [[ValueK]] from a contravariant higher kinded type. */
  def contra[A[-_]](contravariant: A[Any]): ValueK[A] = [Z] => () => contravariant

  /** Construct a [[ValueK]] of a constant. */
  def const[A](a: A): ValueK[Const[A]] = [Z] => () => a

trait ApplicativeK[F[_[_], _]] extends ApplyK[F]:
  extension [A[_]](a: ValueK[A]) def pure[C]: F[A, C]

  def unitK[C]: F[Const[Unit], C] = ValueK.const(()).pure

type ApplicativeKC[F[_[_]]] = ApplicativeK[IgnoreC[F]]
```

Figure 4.4: perspective's ApplicativeK and ValueK.

Applicative's laws

All applicatives must obey the laws of associativity, left identity, and right identity. Associativity was covered with Apply, so only left and right identity will be covered here [37].

Left identity:

```scala
ValueK.const(()).pure.tupledK(fa).mapK([Z] => (ft: (Unit, A[Z])) => ft._2) = fa
```

Right identity:

```scala
fa.tupledK(ValueK.const(()).pure).mapK([Z] => (ft: (A[Z], Unit)) => ft._1) = fa
```

Here is an inductive definition for applicative over products of arbitrary size.

```scala
//Base case
given [X]: ApplicativeKC[[A[_]] =>> A[X]] with
  extension [A[_]](a: ValueK[A]) def pure[C]: A[X] = a()

//Induction step
given [X1[_[_]]: ApplicativeKC, X2]: ApplicativeKC[[A[_]] =>> (X1[A], A[X2])] with
  extension [A[_]](a: ValueK[A]) def pure[C]: (X1[A], A[X2]) = (a.pure, a())
```

The proof of correctness of this definition can be found in Section A.2.

```scala
/** The composition of two higher kinded types. */
type Compose2[A[_], B[_]] = [Z] =>> A[B[Z]]

trait TraverseK[F[_[_], _]] extends FunctorK[F], FoldableK[F]:
  extension [A[_], C](fa: F[A, C])
    def traverseK[G[_]: Applicative, B[_]](f: A ~>: Compose2[G, B]): G[F[B, C]]

    inline def traverseConst[G[_]: Applicative, B](
      f: A ~>#: G[B]
    ): G[F[Const[B], C]] = traverseK(f)

    inline def traverseIdK[G[_]: Applicative](f: A ~>: G): G[F[Id, C]] =
      traverseK(f)

    inline def sequenceIdK(using Applicative[A]): A[F[Id, C]] =
      fa.sequenceK

  extension [G[_]: Applicative, A[_], C](fga: F[Compose2[G, A], C])
    def sequenceK: G[F[A, C]] =
      fga.traverseK(FunctionK.identity[Compose2[G, A]])

type TraverseKC[F[_[_]]] = TraverseK[IgnoreC[F]]
```

Figure 4.5: perspective's TraverseK.

4.1.5 Traverse

As used typically, traverse allows one to convert from `F[G[A]]` to `G[F[A]]`. As an example, a developer has a value of type `List[A]`, maps over this values producing `List[G[A]]`, and need `G[List[A]]`. `G` in this example might be `IO`, or maybe `Either`. In the `IO` case, `Traverse` would allow a user to convert many stateful values into a single stateful value. The `Either` case might be used when parsing something, and the developer is only interested in the first error value.

In perspective, many of these same use cases apply, but for dealing with a generic data structure instead of for example a `List`. The parsing example is the most common one, and allows one to convert a value of type `F[[Z] =>> Either[String, Z]]` to `Either[String, F[Id]]`, where `Z` is the concrete type being parsed. Figure 4.5 shows perspective's `TraverseK`.

Traverse's laws

Here are the laws for `Traverse` instances [36].

```
val fa: F[A, D] = ???
val f: A ~>: Compose2[M, B] = ???
val g: B ~>: Compose2[N, C] = ???

val N = summon[Applicative[N]]
val M = summon[Applicative[M]]

val lhs = Nested(M.map(fa.traverseK(f))(fb => fb.traverseK(g)))
val rhs = fa.traverseK[[Z] => Nested[M, N, Z], C](
  [Z] => (a: A[Z]) => Nested(M.map(f(a))(b => g(b)))
)

lhs == rhs
```

Figure 4.6: Sequential composition traverse law.

Identity: `fa.traverseK[Id, B](f) = fa.mapK(f)`

Sequential composition: See Figure 4.6.

Parallel composition: See Figure 4.7.

Here is an inductive definition for traverse over products of arbitrary size.

```
1  //Base case
2  given [X]: TraverseKC[[A[_]] =>> A[X]] with
3    extension [A[_], C](fa: A[X])
4      def traverseK[G[_]: Applicative, B[_]](f: A ~>: Compose2[G, B]): G[B[X]] =
5        f(fa)
6
7  //Induction step
8  given [X1[_[_]]: TraverseKC, X2]: TraverseKC[[A[_]] =>> (X1[A], A[X2])] with
9    extension [A[_], C](fa: (X1[A], A[X2]))
10     def traverseK[G[_]: Applicative, B[_]](
11       f: A ~>: Compose2[G, B]
12     ): G[(X1[B], B[X2])] = fa._1.traverseK(f).product(f(fa._2))
```

The proof of correctness of this definition can be found in Section A.3.

4.1.6 Distributive

Distributive is the dual of Traverse. Distributive allows one to convert from `G[F[A]]` to `F[[Z] =>> G[A[Z]]]`. All Distributives are functors. Figure 4.8 shows perspective's `DistributiveK`.

I will not go over Distributive's laws or instances for it, as it is isomorphic with another class, Representable, which will be covered in subsection 4.1.8 [23].

```
val fa: F[A, D] = ???
val f: A ~>: Compose2[M, B] = ???
val g: B ~>: Compose2[N, B] = ???

val N = summon[Applicative[N]]
val M = summon[Applicative[M]]

type MN[Z] = (M[Z], N[Z])

given Applicative[MN] with:
  def pure[X](x: X): MN[X] = (M.pure(x), N.pure(x))

  def pure[X, Y](f: MN[X => Y])(fa: MN[X]): MN[Y] =
    val (fam, fan) = fa
    val (fm, fn) = f
    (M.ap(fm)(fam), N.ap(fn)(fan))

  override def map[X, Y](fx: MN[X])(f: X => Y): MN[Y] =
    val (mx, nx) = fx
    (M.map(mx)(f), M.map(nx)(f))

  override def product[X, Y](fx: MN[X], fy: MN[Y]): MN[(X, Y)] =
    val (mx, nx) = fx
    val (my, ny) = fy
    (M.product(mx, my), N.product(nx, ny))
end given

val lhs: MN[F[B]] = fa.traverseK[MN, B]([Z] => (a: A[Z]) => (f(a), g(a)))
val rhs: MN[F[B]] = (fa.traverseK(f), fa.traverseK(g))

lhs == rhs
```

Figure 4.7: Parallel composition traverse law.

```
1  trait DistributiveK[F[_[_], _]] extends FunctorK[F]:
2    extension [G[_]: Functor, A[_], C](gfa: G[F[A, C]])
3      def distributeK[B[_]](f: Compose2[G, A] ~>: B): F[B, C] =
4        gfa.cosequenceK.mapK(f)
5
6      inline def distributeConst[B](f: Compose2[G, A] ~>#: B): F[Const[B], C] =
7        distributeK[Const[B]](f)
8
9      def cosequenceK: F[Compose2[G, A], C]
10
11   extension [G[_]: Functor, A](ga: G[A])
12     def collectK[B[_], C](f: A => F[B, C]): F[Compose2[G, B], C] =
13       ga.map(f).cosequenceK
14
15 type DistributiveKC[F[_[_]]] = DistributiveK[IgnoreC[F]]
```

Figure 4.8: perspective's DistributiveK.

```scala
trait MonadK[F[_[_], _]] extends ApplicativeK[F]:
  extension [A[_], C](ffa: F[[Z] =>> F[A, Z], C]) def flattenK: F[A, C] =
    ffa.flatMapK(FunctionK.identity)

  extension [A[_], C](fa: F[A, C])
    def flatMapK[B[_]](f: A ~>: ([Z] =>> F[B, Z])): F[B, C]

type MonadKC[F[_[_]]] = MonadK[IgnoreC[F]]
```

Figure 4.9: perspective's MonadK.

4.1.7 Monad

Explaining `MonadK` in simple terms is probably as hard as explaining `Monad` to someone for the first time. I have not seen any actual uses for `MonadK`, and it is here mainly for completeness' sake. All instances of Monad are also Applicative. Figure 4.9 shows perspective's `MonadK`.

The flatten function is probably the easier one to explain. If `F[A, C]` has n values arranged as a list, `F[[Z] =>> F[A, Z], C]` has n^2 values arranged as a table. `flattenK` grabs the values along the diagonal to get back to n elements. The first column of the first row is taken for the first field. The second column of the second row is taken for the second field. This goes on until all fields have a value. Using the kind `F[_[_], _]` instead of `F[_[_]]` also makes the types of the functions in `MonadK` simpler. For `MonadKC`, the type of `flattenK` is instead `F[Const[F[A]]] => F[A]`. Figure 4.10 shows an example of how to implement a `Monad` instance over a tuple of size three where all the values have the same type.

Monad's laws

Left identity:

```scala
ValueK.const(a).pure.flatMapK(f) = f(a)
```

Right identity:

```scala
fa.flatMapK([Z] => (a: A[Z]) => ValueK.const(a).pure) = fa
```

Associativity:

```scala
fa.flatMapK(f).flatMapK(g) = fa.flatMapK([Z] => (a: A[Z]) => f(a).flatMapK(g))
```

Here is an inductive definition for monad over products of arbitrary size.

```scala
trait Monad[F[_]]:
  extension [A](ffa: F[F[A]]) def flatten: F[A]

given Monad[[Z] =>> (Z, Z, Z)]:
  extension [A](
    ffa: (
      (A, A, A),
      (A, A, A),
      (A, A, A)
    )
  ) def flatten: (A, A, A) =
    // The tuple is deconstructed to better show the "table" explanation
    val (
      (a11, a12, a13),
      (a21, a22, a23),
      (a31, a32, a33)
    ) = ffa
    (a11, a22, a33)
```

Figure 4.10: Example instance of Monad for Tuple3.

```scala
//Base case
given [X]: MonadKC[[A[_]] =>> A[X]] with
  extension [A[_], C](fa: A[X])
    def flatMapK[B[_]](f: A ~>#: B[X]): B[X] = f(fa)

//Induction step
given [X1[_[_]]: MonadKC, X2]: MonadKC[[A[_]] =>> (X1[A], A[X2])] with
  extension [A[_], C](fa: (X1[A], A[X2]))
    def flatMapK[B[_]](f: A ~>#: (X1[B], B[X2])): (X1[B], B[X2]) =
      (fa._1.flatMapK([Z] => (a: A[Z]) => f(a)._1), f(fa._2)._2)
```

The proof of correctness of this definition can be found in Section A.4.

One thing to note about this proof is that it cannot be typed in normal Scala. The reason for this is that there is no function like `def pureK[A](a: A): F[A]`. This code does not typecheck, as `F` expects a higher kinded type. Likewise, the function `def pureK[A[_], Z](a: A[Z]): F[A]` does not exist. If it did, it could blow up at runtime. Say `F` is `A[_] => A[Int]`, and one runs `pureK(Some("string"))`. The code somehow needs a way to turn an arbitrary type, in this case, `String`, into `Int`. Outside of constant functions, such a function does not exist. There is, however, something like the function `def pureConst[A](a: A): F[Const[A]]`. This function safely lifts a value into `F`. For the right identity law, however, this type conflicts with the wanted type. The code wants to

call `flatMapK[A]` but ends up with values of type `F[Const[A[Z]]]`. This type cannot be used with `flatMapK`, as the type `Z` would escape outside of the scope it is defined in.

4.1.8 Representable

The typeclasses defined above provide functions a user might use to work on a fixed number of values at a time. They are also the functions a user might use when working with lists and other containers like them. With perspective, these operations are a simplified way to operate on a structure, but they are more constrained in what they can do. For example, `map2K` allows combining two different structures, but only two. If combining three different structures is needed, a user must first combine two of them, then combine the result with the third.

This problem and many more is what Representable solves. Representable offers a way to index and tabulate over the data structure being worked on, similar to `List#apply` and `List.tabulate` do for ordinary lists. It also offers more efficiency, since, users do not need to iterate over the same structure multiple times. It also makes code cleaner by reducing the number of lambdas.

The way `Representable` works is that it has an index type (called `RepresentationK` in perspective), and the values of this type is used to index into the type being worked on. What `RepresentationK` is backed by does not matter for generic programming. The important part here is an isomorphism between the fields of the type being worked with and the values of the representation. That means that this index function needs to be total. A list is, for example, not representable unless the size of the list is known at compile time and a fixed size type is used to index into the list. RepresentableK's definition in perspective can be seen in Figure 4.11.

All functions found in `FunctorK`, `ApplyK`, `ApplicativeK`, `MonadK` and `DistributiveK` can be implemented using functions found in `RepresentableK`, and should as such be avoided and replaced with functions found in `RepresentableK` in performant generic programming code if possible. Functions found in `TraverseK` and `FoldableK` cannot be implemented using `RepresentableK`, but can still be performed more efficiently in many cases by using the functions of `RepresentableK#indices` instead.

Figure 4.12 shows and example of how `RepresentableK` can be implemented for higher kinded data. Essentially, `RepresentableK` turns the fields of some higher kinded data

```scala
trait RepresentableK[F[_[_], _]] extends MonadK[F] with DistributiveK[F]:
  type RepresentationK[_]

  def tabulateK[A[_], C](f: RepresentationK ~>: A): F[A, C]

  def indicesK[C]: F[RepresentationK, C] = tabulateK(FunctionK.identity)

  extension [A[_], C](fa: F[A, C])
    def indexK[Z](i: RepresentationK[Z]): A[Z]

type RepresentableKC[F[_[_]]] = RepresentableK[IgnoreC[F]]
```

Figure 4.11: perspective's RepresentableK.

```scala
case class IsbK[F[_]](
  intField: F[Int], stringField: F[String], booleanField: F[Boolean]
)

enum IsbKRepresentation[A]:
  case IntField extends IsbKRepresentation[Int]
  case StringField extends IsbKRepresentation[String]
  case BooleanField extends IsbKRepresentation[Boolean]

given RepresentableKC[IsbK] with
  type RepresentationK[A] = IsbKRepresentation[A]
  import IsbKRepresentation.*

  def tabulateK[A[_], C](f: RepresentationK ~>: A): IsbK[A] =
    IsbK(f(IntField), f(StringField), f(BooleanField))

  extension [A[_], C](fa: IsbK[A])
    def indexK[Z](i: RepresentationK[Z]): A[Z] = i match
      case IntField => i.intField
      case StringField => i.stringField
      case BooleanField => i.booleanField
```

Figure 4.12: Defining an instance of RepresentableK.

```scala
def deriveShowPerspectiveRepresentable[F[_]: FoldableKC](
  typeName: String, names: F[Const[String]], instances: F[Show]
)(using F: RepresentableKC[F]): Show[F[Id]] = new Show[F[Id]]:
  extension (a: F[Id]) def show: String =
    val elems = F.indicesK.foldLeftK("") { acc => [Z] => (i: F.RepresentationK[Z]) =>
      val value = a.indexK(i)
      val name = names.indexK(i)
      val instance = instances.indexK(i)

      s"$acc, $name = ${instance.show(value)}"
    }

    s"$typeName($elems)"
end deriveShowPerspective
```

Figure 4.13: Deriving Show using RepresentableK and FoldableK.

into a sum type, and allows functions to match over and pass the fields to functions using this sum type.

Figure 4.13 shows an example of how to derive instances of `Show` using `RepresentableK` and `FoldableK`. The `foldLeftK` call is done on the indices found in `RepresentableK`, giving access to these indices in the body of the fold. The value, name, and typeclass instance for each field are then obtained by indexing into the structure containing the appropriate values. The values are then combined as seen in previous examples.

Representable's laws

Representable has very simple laws. The only thing it requires is that the representation is a true isomorphism. These two equalities can express that [35].

Tabulate index:

```
tabulateK([Z] => (i: RepK[Z]) => fa.indexK(i)) = fa
```

Index tabulate:

```
tabulateK(f).indexK(i) = f(i)
```

Here is an inductive definition for representable over products of arbitrary size.

```scala
//Base case
given [X]: RepresentableKC[[A[_]] =>> A[X]] with
  type RepresentationK[_] = Unit

  def tabulateK[A[_], C](f: RepresentationK ~>: A): A[X] = f(())

  extension [A[_], C](fa: A[X])
    def indexK[Z](i: RepresentationK[Z]): A[Z] =
      // This case is safe. There is no way to get an instance
      // of the representation with the wrong type
      fa.asInstanceOf[Z]

//Induction step
given [X1[_[_]], R1[_], X2](
  using X1: RepresentableKC.Aux[X1, R1]
): RepresentableKC[[A[_]] =>> (X1[A], A[X2])] with
  type RepresentationK[RepA] = Either[R1[RepA], Unit]

  def tabulateK[A[_], C](f: RepresentationK ~>: A): (X1[A], A[X2]) =
    (X1.tabulateK([Z] => (r: R1[Z]) => f(Left(r))), f(Right(())))

  extension [A[_], C](fa: (X1[A], A[X2]))
    def indexK[Z](i: RepresentationK[Z]): A[Z] = i match
      case Left(rep) => fa._1.indexK(rep)
      case Right(()) =>
        // This case is safe. There is no way to get an instance
        // of the representation with the wrong type
        fa._2.asInstanceOf[Z]
```

The proof of correctness of this definition can be found in Section A.5.

4.2 Exotic Higher Kinded Data

Whenever one has a normal case class with no unapplied type arguments, one can create a higher kinded version of this case class by parameterizing the case class with a higher kinded type and wrapping all the types of the fields in this higher kinded type. One can also define functions that, for any case class, exposes a type that is higher kinded. This can be done using tuples and `Mirror`, as follows:

```scala
def toHigherKinded[A](a: A)(
  using m: Mirror.ProductOf[A]
): Tuple.Map[m.MirroredElemTypes, Id] =
  Tuple.fromProduct(product).asInstanceOf[Tuple.Map[m.MirroredElemTypes, Id]]

def fromHigherKinded[A, T <: Tuple](t: Tuple.Map[m.MirroredElemTypes, Id])(
  using m: Mirror.ProductOf[A]
): A = n.fromTuple(t.asInstanceOf[m.MirroredElemTypes])
```

Any higher kinded data created this way will have all the typeclasses we have introduced. This fact is, however, not true for all higher kinded data. I will call such higher kinded data exotic. A simple example of such higher kinded data is anything containing `List[F[A]]`, which is not the same as `F[List[A]]`. In the latter case, the list is wrapped in the higher kinded type, like usual. In the former case, there is instead an unknown number of elements wrapped in the higher kinded type. As the number of elements is unknown, the type is no longer representable.

Another example of exotic higher kinded data is `Either[F[A], F[A]]`. While it is still possible to implement `FunctorK` and `FoldableK` for this type, implementing more requires the type to be biased to one side. Doing so might not be wanted when doing generic programming, as treating everything equally is desirable. This particular kind of exotic higher kinded data will be discussed more in the chapter on sum types.

4.3 perspective-derivation

With the typeclasses and their operations out of the way, let us go over how perspective puts them together in a nice box to help with deriving typeclasses. Note that what I discuss in this section is a slight simplification of perspective's features.

Like shapeless 2's `HList` and the `ListGeneric`'s `List`, perspective too has a generic representation that the operations will work on. Unlike shapeless 2 and `ListGeneric`, however, this generic representation does not need to be exposed as much to the user, as all the operations that interact with the representation come from typeclasses. For a representation `Gen[_[_]]`, it needs to be simple to get values of `Gen[TC]` for some typeclass `TC[_]`. This is done by taking `Gen[TC]` as an implicit parameter. For example, if the generic representation represents a type with fields `Int`, `String` and `Boolean`, and

there exist instances of `Show` for all these types, then there must also exist an implicit value of `Gen[Show]`. To allow for this, the type `Gen[_[_]]` cannot be completely opaque (meaning it must be observable from the outside).

The index type used by `Representable` should also be easily accessible as it is the most performant way to work with higher kinded data.

perspective puts all of these ideas together, in addition to some extra functions that provide more flexibility or performance, and gets its own generic typeclass called `HKDProductGeneric`. This typeclass can be seen in Figure 4.14

The type is a union of all the field types contained within `A`. It is used more with sum types, but for product types it only provides a bound when converting a string to an index. Both the types of the field names and the type name are stored both on the value level and on the type level. The information is stored at the type level for instances where a user might want to manipulate this information at the type level. For example, using this information, one could create a function (shown below) that lets one access a field with a string, but only if the string is known at compile time to be a valid field name.

```
1 def access[A, FNames <: String, K <: FNames & Singleton](a: A)(field: K)(
2   using gen: HKDGeneric[A] { type Names = FNames}
3 ): gen.FieldOf[K] = a.productElementId(gen.nameToIndex(field))
```

The `productElementId` function is a small optimization that lets one index into `A` without converting `A` into the generic representation first. `productElementId` internally calls `Product#productElement` which is used to index into anything that extends `Product`. All case classes extends `Product`.

`TupleRep` together with `genToTuple` and `tupleToGen` provides easy conversion to and from tuples. This is useful for interoperability with other datatype-generic code. In addition to that, some operations are more easily done with tuples.

The functions `tabulateFoldLeft`, `tabulateTraverseK`, `tabulateTraverseKOption` and `tabulateTraverseKEither` are all optimized and fused versions of calls of the form `representable.indices.<operation>`. These functions are more efficient as they do not have to construct the `F[Index]` value. The traverse function also has specializations provided for option and either. Traverse is a relatively slow operation generally, so these are provided when `G` is known to be `Option` or `Either`.

```scala
trait HKDProductGeneric[A]:
  type Gen[_[_]]
  type Index[_]
  type ElemTop

  type TypeName <: String
  def typeName: TypeName

  type Names <: String
  def names: Gen[Const[Names]]

  def stringToName(s: String): Option[Names]

  type FieldOf[Name <: Names] <: ElemTop
  def nameToIndex[Name <: Names](name: Name): Index[FieldOf[Name]]

  def to(a: A): Gen[Id]
  def from(gen: Gen[Id]): A

  extension (a: A) def productElementId[X](index: Index[X]): X

  type TupleRep <: Tuple
  def genToTuple[F[_]](gen: Gen[F]): Tuple.Map[TupleRep, F]
  def tupleToGen[F[_]](tuple: Tuple.Map[TupleRep, F]): Gen[F]

  lazy val representable: RepresentableKC.Aux[Gen, Index]
  lazy val traverse: TraverseKC[Gen]
  given RepresentableKC.Aux[Gen, Index] = representable
  given TraverseKC[Gen]                 = traverse

  def tabulateFoldLeft[B](start: B)(f: B => [X] => Index[X] => B): B

  def tabulateTraverseK[G[_], B[_]](f: [X] => Index[X] => G[B[X]])(
    using Applicative[G]
  ): G[Gen[B]]

  def tabulateTraverseKOption[B[_]](
    f: [X] => Index[X] => Option[B[X]]
  ): Option[Gen[B]]

  def tabulateTraverseKEither[E, B[_]](
    f: [X] => Index[X] => Either[E, B[X]]
  ): Either[E, Gen[B]]
```

Figure 4.14: A simplified version of perspective's HKDProductGeneric typeclass.

4.3.1 Using HKDProductGeneric

One uses HKDProductGeneric not too unlike what has already been covered with `Representable` in Figure 4.13. Unlike with typeclasses, everything that is needed is housed in the generic instance or can be summoned as typeclasses. HKDProductGeneric also prioritizes working with indices over other functions supplied by the typeclasses. The functions in the typeclasses are more to let new developers understand perspective quicker and not require them to understand how indices work at first.

Here is an example of deriving instances of `Show` using HKDProductGeneric. The code is very similar to what was covered with `Representable`, but here it uses `HKDProductGeneric` instead. It provides the functions `tabulateFoldLeft` and `productElementId` which are used instead where possible.

```
def deriveShow[A](
  using gen: HKDProductGeneric[A], instances: gen.Gen[Show]
): Show[A] = new Show[A]:
  import gen.given
  private val names = gen.names

  extension (a: A) def show: String =
    val elems = gen.tabulateFoldLeft("") { acc =>
      [Z] => (idx: gen.Index[Z]) =>
        val value = a.productElementId(idx)
        val name = names.indexK(idx)
        val instance = instances.indexK(idx)

        s"$acc, $name = ${instance.show(value)}"
    }
    s"${gen.typeName}($elems)"
```

4.3.2 Implementation

The perspective library currently uses `Product` as the generic representation. As the type being abstracted over is also a `Product`, calling `HKDProductGeneric#to` is a NO-OP. In all other cases, a wrapped array is used, which inherits from `Product`.

The optimized tabulate functions are all implemented with while loops and early returns where possible.

4.4 Inline perspective-derivation

Sometimes normal generic derivation is not quite fast enough for a developer, which is when they would typically turn to macros instead. perspective, however, offers an alternative to this. perspective provides a separate generic typeclass called `InlineHKDProductGeneric`. This typeclass provides mostly the same user interface as `HKDProductGeneric` but changing all the implementations to inline functions and macros. This feature is only available with Scala 3.

`InlineHKDProductGeneric`'s tools are entirely separate from everything else in perspective, as inline requires reimplementing everything. A typeclass based approach was tried, but Scala did not manage to resolve the functions when needed, so typeclasses were dropped. Instead, all the functions found within the typeclasses were moved directly to the same generic instance.

`InlineHKDProductGeneric` uses a lot of beta reduction to simplify code and avoid lambdas in the generated code. At the time of writing `InlineHKDProductGeneric`, this beta reduction did not work with polymorphic functions (it now does [16]). perspective got around this problem by representing the `Index` type differently using path dependent types and aggressively recommending users to only use the tabulate functions. Non-tabulate functions are still left in the API but might not generate as good bytecode. The new `Index` type looks like this:

```
1 type Index <: Any { type X <: ElemTop }
2 type IndexAux[X0 <: ElemTop] = Index { type X = X0 }
3
4 class IdxWrapper[X](val idx: IndexAux[X & ElemTop])
5
6 given [X]: Conversion[IdxWrapper[X], IndexAux[X & ElemTop]] = _.idx
7 given [X]: Conversion[IndexAux[X & ElemTop], IdxWrapper[X]] = new IdxWrapper(_)
```

The `Any` bound is to allow for implementing `Index` using `Int` later. I also bound `Index#X` to `ElemTop` as I control all the code `Index` will interact with. In some cases, adding a bound to `Index#X` might pose problems with type checking, which `IdxWrapper` exists to solve. For example, `indicesK` must be typed as `Gen[IdxWrapper]` as otherwise `Gen` would have to be defined as `Gen[_[_ <: ElemTop]]`. Doing so would conflict with working with most typeclasses.

This representation of the index type also coincidentally leads to less boilerplate. Compare the non-inline and inline versions of an implementation of `mapK` in terms of `tabulate`.

```
// Non-inline
def mapK[B[_]](f: A ~>: B): Gen[B] =
  representable.tabulateK(fa)([Z] => (i: Index[Z]) => f(i))

// Inline
def mapK[B[_]](f: A ~>: B): Gen[B] =
  gen.tabulateK(i => f(i))
```

`InlineHKDGeneric` also provides a function `inline def summonInstances[F[_]]: Gen[F]` that will generate code containing instances of a certain typeclass for each field in the type being worked on. This function replaces implicit parameters of instances (`Gen[TC]` for some typeclass `TC`), which was used with `HKDGeneric`. As a result, the generic representation can remain completely opaque to the user.

4.4.1 Using InlineHKDProductGeneric

Usage of `InlineHKDProductGeneric` does not differ significantly from usage of `HKDProductGeneric` except that there are no polymorphic function calls and implicit instances are found differently. To showcase the similarity, here is an example of deriving instances of `Show` using `InlineHKDProductGeneric`.

```
inline def deriveShow[A](
  using gen: InlineHKDProductGeneric[A]
): Show[A] = new Show[A]:
  private val names = gen.names
  private val instances = gen.summonInstances[Show]

  extension (a: A) def show: String =
    val elems = gen.tabulateFoldLeft("") { (acc, idx) =>
      val value = a.productElementId(idx)
      val name = names.indexK(idx)
      val instance = instances.indexK(idx)

      s"$acc, $name = ${instance.show(value)}"
    }
    s"${gen.typeName}($elems)"
```

4.4.2 Implementation

In `InlineHKDGeneric`, the generic representation is an `IArray`. The index type is an `Int` with an extra type member added to it. Functions are implemented with inline and while loops or macros that generate while loops. These while loops are, in some cases, specialized depending on the types involved. This specialization is especially important for traverse, removing the need for individually specialized method calls. This specialization is done by inspecting the implicit instances of `Applicative` passed in. If the instance matches a known one, a specialized implementation is used.

4.5 Unrolling perspective-derivation

`InlineHKDProductGeneric` has some support for loop unrolling for specific pieces of code, so in some cases it can remove the while loops the code would otherwise generate. A developer can enable unrolling by passing in `unrolling = true` to functions that support unrolling. At the time of writing, these functions are `tabulateK`, `tabulateFoldLeft`, and `tabulateTraverseK`.

Loop unrolling has two main use cases. The first is simpler and sometimes more compact bytecode. Even when the code is not more compact, it might still run quicker in benchmarks. Here is an example of what the generated code for a `foldLeft` function call might look like compared to the unrolled version.

```
1  def notUnrolled(arr: Array[String]): String =
2    var acc: String = ""
3    var i: Int = 0
4    while(i < arr.length) do
5      acc = acc + arr(i)
6      i = i + 1
7    acc
8
9  def unrolled(arr: Array[String]): String =
10   "" + arr(0) + arr(1) + arr(2)
```

The other usecase is avoiding boxing. When one has a `Gen[Id]` and a call to `indexK(idx)` on it, one is given a value of type `idx.X` with the bound `ElemTop`. If

`ElemTop` is equal to `Int` because the type being abstracted over only contains integers, then there is no problem. If, however, `ElemTop` contains anything else, one will have to deal with the JVM's boxing. In most cases, this boxing is inevitable. Boxing happens everywhere. For example, all generic functions box their arguments and result (some exceptions exist where the Scala compiler will specialize a small subset of functions).

In a loop, boxing is inevitable, as the value needs to take on many different types. For example, if one is dealing with a case class defined as
`case class ISB(i: Int, s: String, b: Boolean)` and call `indexK` on a generic representation of the case class, the call will return different types on different iterations. Here is an example.

```
1  inline def boxingHappensHere[A](a: A)(using gen: InlineHKDProductGeneric[A]): String =
2    val instances = gen.summonInstances[Show]
3    gen.tabulateFoldLeft("") { (acc, i) =>
4      val value: i.Idx = a.productElementId(i)
5      val instance = instances.indexK(i)
6      acc + instance.show(value)
7    }
```

On the first iteration here, `value` will contain an `Int`. On the next, it will be a `String`. On the last, it will be a `Boolean`. As such, `value` will be boxed.

Even if one can avoid boxing the value when getting it using `indexK`, care must still be taken not to box the value later. Any typeclass used here would, for example, lead to boxing as a generic method would be involved. In the case of `Show`, calling `show` on an `Int` will box the int when passing it to the typeclass. A specialized method is also needed to do the desired operation, which will not box. Instead of `Show[Int]`, one could, for example, use `java.lang.Integer.toString`.

With both obstacles out of the way, here is how to prevent boxing by unrolling code. First, the field must be accessed in a way that does not box. `indexK` and `productElementId` will both box. Instead, `productElementIdExact` will be used. This function is like `productElementId`, but it will access the field directly using its name instead of using `Product#productElement`. `productElementIdExact` can only be called in unrolling code.

The next step is to match on the type and choose what to do based on that. Scala 3's `inline match` looks perfect to do this. Sadly it expands too quickly when

```scala
inline def noBoxingHere[A](a: A)(using gen: InlineHKDProductGeneric[A]): String =
  val instances = gen.summonInstances[Show]
  gen.tabulateFoldLeft("", unrolling = true) { (acc, i) =>
    val str = gen.lateInlineMatch {
      a.productElementIdExact(i) match
        case p: Byte    => java.lang.Byte.toString(p)
        case p: Short   => java.lang.Byte.toString(p)
        case p: Char    => java.lang.Character.toString(p)
        case p: Int     => java.lang.Integer.toString(p)
        case p: Long    => java.lang.Long.toString(p)
        case p: Float   => java.lang.Float.toString(p)
        case p: Double  => java.lang.Double.toString(p)
        case p: Boolean => java.lang.Boolean.toString(p)
        case other      =>  instances.indexK(i).show(other)
    }

    acc + str
  }
```

Figure 4.15: A non-boxing function that folds over a data structure, converts the fields to strings, and concatenates them.

the type information has not yet been refined. As a workaround, perspective provides `lateInlineMatch`, which tries to do something similar. The argument of the function must be a match expression and it must always be used within an unrolled function. Figure 4.15 shows what a non-boxing version of the above code snippet might look like by putting all of the ideas discussed together.

4.5.1 Limitations of perspective's unrolling

perspective's unrolling is more a preview of what is possible than an implementation of true unrolling. It covers areas that are easy to implement and use but is very limited. For example, the array that is the intermediary representation will box anything put into it. That means that calls to `tabulateK` and `tabulateTraverseK` will box primitives. A proper implementation of unrolling that tries to avoid boxing in these instances might require a completely different approach, not unlike how `InlineHKDProductGeneric` and `HKDProductGeneric` are entirely different.

I can see two methods to implement a completely unrolling version of `InlineHKDProductGeneric`: one without an intermediary representation and one with a miniboxed array intermediary representation. I do not foresee perspective implementing either of these methods in the near future, but nevertheless document preliminary ideas about both approaches.

No intermediary representation

The first approach would involve no intermediary representation. The API would still include one in a generic typeclass, but no trace of it would remain in the code. Instead, each iteration call would be converted into many different variables, one for each field. Each intermediary representation would be expanded into a set of variables. The tricky part here is to avoid naming issues and ensure everything works together. Scoping would likely be a problem here. The implementation would probably rely on a single "god" macro to implement all the needed logic. The reason this would likely be needed is to make sure everything flowed together nicely. The alternative would be to somehow pass information from one macro to another, which might become clunky. Erased definitions [26] would possibly make this approach easier.

Miniboxing

Miniboxing is a technique for specializing code yet avoiding excessive bytecode duplication. The idea is to identify disjoint sets of types that can all be stored in the same manner and only emit one specialization per set [41]. Applying this technique to `InlineHKDProductGeneric` would lead to each disjoint set getting its own backing array. This method is far less involved than removing the intermediary representation at the cost of doing the miniboxing.

4.6 Summary of generic programming using perspective

The perspective library brings many new ideas and ways of doing things to generic programming that have not been seen much before. Like shapeless 3, perspective does not require any type level programming and works on the value level instead of the type level. Unlike shapeless 3, it is not restricted to a single operation or set of typeclasses to work with. In this area, perspective is more similar to shapeless 2.

On the other hand, perspective also requires the developer to be completely comfortable with working with higher kinded data. shapeless 3 is both simpler from an API standpoint and from the standpoint of what it asks of the developer. There is generally

one way to use shapeless 3, while perspective offers many different ways of achieving the same result. A developer might use simpler typeclass functions like `mapK`, `map2K` and `traverseK`, they might use index based functions, they might use inline functions or they might also be unrolling and then have to think about how everything boxes and what the resulting class file will look like.

The manner perspective uses Scala's type system is new and it has shown to stretch Scala's and the Scala compiler's limits. For example, perspective does not work well with Scala 2.12, as the type inference breaks in most places. This is why I dropped Scala 2.12 support early in the development. perspective for Scala 3 also uses many new features like polymorphic functions that are still error prone in specific configurations.

Chapter 5

Dealing with sum types

Up until this point, I have only talked about product types. However, sum types are also important, and will be covered in this chapter.

5.1 Sum types in shapeless 2

The shapeless 2 library encodes sum types in a type it calls `Coproduct`. `Coproduct` is like `HList` but for sum types instead of product types, storing only one value among a list of possible values instead of a list of values. It is like a bunch of nested `Either`s growing to the right. It is defined as something like this.

```
1 sealed trait Coproduct
2 sealed trait :+:[H, T <: Coproduct] extends Coproduct
3 case class Inl[H, T <: Coproduct](head: H) extends :+:[H, T]
4 case class Inr[H, T <: Coproduct](tail T) extends :+:[H, T]
5
6 sealed trait CNil extends Coproduct:
7   def impossible: Nothing
```

Note that CNil is not inhabited and only exists to end a Coproduct. Much of the same that applies to HList, like type level programming and how to deal with labels, also applies to Coproduct. The example below shows how a Coproduct could represent a sum type.

```
enum Foo:
  case Bar(i: Int)
  case Baz(s: String)
  case Bin(b: Boolean)

type FooRepr = Foo.Bar :+: Foo.Baz :+: Foo.Bin :+: CNil
```

Here is an example of how to derive an instance of `Show` for sum types:

```
given cnilShow: Show[CNil]:
  extension (a: CNil) def show: String = a.impossible

given cconsShow: [H, T <: HList](using Show[H], Show[T]): Show[H :+: T]:
  extension (a: H :+: T) def show: String = a match
    case Inl(h) => h.show
    case Inr(t) => t.show

def deriveShowGeneric[A](
  using gen: Generic[A], instance: Show[gen.Repr]
): Show[A] = new Show[A]:
extension (a: A) def show: String = gen.to(a).show
```

For the `CNil` case, the code calls `impossible` found on `CNil`. This function returns `Nothing`, the uninhabited type. The cons case meanwhile matches on the `:+:` and shows the head or the tail depending on what the value is.

5.2 Sum types in shapeless 3

The shapeless 3 library uses the same approach for both sum and product types. That means hiding the intermediary representation and offering a few functions to operate on the data instead.

```
extension [F[_], T](inst: CoproductInstances[F, T])
  inline def map(x: T)(f: [t] => (F[t], t) => t): T
  inline def inject[R](p: Int)(f: [t <: T] => F[t] => R): R
  inline def fold[R](x: T)(f: [t <: T] => (F[t], t) => R): R
  inline def fold2[R](x: T, y: T)(a: => R)(f: [t <: T] => (F[t], t, t) => R): R
  inline def fold2[R](x: T, y: T)(g: (Int, Int) => R)(
    f: [t <: T] => (F[t], t, t) => R
  ): R
```

In this case, there are almost just fold functions. `map` applies the function to the present case and returns it. `fold` works almost the same as `map` but the return value is allowed to be anything, not just the type being abstracted over. `fold2` works similarly but with two values. If the two values are not of the same case, the value `a` or the function `g` applied to the ordinals of the two cases is returned instead. Finally `inject` allows a user to construct a value using the typeclass at the specified ordinal. This ordinal could for example be obtained by checking the index of a string in `Labelling`'s `Seq[String]`.

Here is how to derive `Show` using shapeless 3.

```
1 def deriveShow[A](using inst: CoproductInstances[Show, A]): Show[A] = new Show[A]:
2   extension (a: A) def show: String =
3     inst.fold(a)([t <: A] => (sh: Show[t], v: t) => sh.show(v))
```

5.3 Sum types in perspective

Now for how to work with sum types in perspective. As was covered earlier, `Either[F[A], F[A]]` can be considered an exotic higher kinded type, as only folds and map can easily be defined on it without introducing a bias. As `Either` is the traditionally simplest sum type, things are not looking great. This difficulty in defining typeclass instances over sum types is potentially why shapeless 3 mostly offers folds for dealing with sum types.

perspective's perspective is more open-minded. What happens if I as the developer of perspective try to define an instance of `ApplyKC` for `Either` as shown in Figure 5.1? Where does the code go wrong, and is there anything that could be done to cover these holes? I will mark these holes where things break down with `???`.

The problem cases are the ones where values are present on different sides. One solution to this problem is to change the definition of Either to allow for this trivially, by introducing a value that represents neither side. Figure 5.2 shows the result of doing this.

The code in Figure 5.2 works, but the approach cannot also be used with `ApplicativeK`, implementing `pure` as returning `NeitherK`. Doing so breaks all applicative laws that interact with `pure`. In this case, the opposite of what was done with `NeitherK` is needed. That is to say, we need a case where both values are present. Such a type exists and is

```scala
enum EitherK[F[_], A, B]:
  case LeftK(a: F[A])
  case RightK(b: F[B])

given [L, R]: ApplyKC[[F[_]] =>> EitherK[F, L, R]] with
  extension [A[_], C](fa: EitherK[A, L, R])
    def map2K[B[_], Z[_]](fb: EitherK[B, L, R])(
      f: [X] => (A[X], B[X]) => Z[X]
    ): Ior[Z, L, R] =
      import EitherK.*
      (fa, fb) match
        case (LeftK(la), LeftK(lb))    => LeftK(f(la, lb))
        case (LeftK(la), RightK(_))    => ???

        case (RightK(ra), RightK(rb))   => RightK(f(ra, rb))
        case (RightK(ra), LeftK(_))     => ???
```

Figure 5.1: Trying to define ApplyKC for EitherK.

```scala
enum OptEitherK[F[_], A, B]:
  case LeftK(a: F[A])
  case RightK(b: F[B])
  case NeitherK

given [L, R]: ApplyKC[[F[_]] =>> OptEitherK[F, L, R]] with
  extension [A[_], C](fa: OptEitherK[A, L, R])
    def map2K[B[_], Z[_]](fb: OptEitherK[B, L, R])(
      f: [X] => (A[X], B[X]) => Z[X]
    ): OptEitherK[Z, L, R] =
      import OptEitherK.*
      (fa, fb) match
        case (LeftK(la), LeftK(lb)) => LeftK(f(la, lb))
        case (LeftK(la), RightK(_)) => NeitherK
        case (LeftK(la), NeitherK)  => NeitherK

        case (RightK(ra), RightK(rb)) => RightK(f(ra, rb))
        case (RightK(ra), LeftK(_))   => NeitherK
        case (RightK(ra), NeitherK)   => NeitherK
```

Figure 5.2: Trying to define ApplyKC for OptEitherK.

```scala
enum Ior[A, B]:
  case Left(a: A)
  case Right(b: B)
  case Both(a: A, b: B)
```

Figure 5.3: Definition of Ior.

```scala
case class OptIorKProd[F[_], A, B](left: Option[F[A]], right: Option[F[B]])
enum OptIorKSum[F[_], A, B]:
  case LeftK(a: F[A])
  case RightK(b: F[B])
  case BothK(left: F[A], right: F[B])
  case NeitherK

def sumToProd[F[_], A, B](sum: OptIorKSum[F, A, B]): OptIorKProd[F, A, B] =
  sum match
    case LeftK(a)    => OptIorKProd(Some(a), None)
    case RightK(b)   => OptIorKProd(None, Some(b))
    case BothK(a, b) => OptIorKProd(Some(a), Some(b))
    case NeitherK    => OptIorKProd(None, None)

def prodToSum[F[_], A, B](prod: OptIorKProd[F, A, B]): OptIorKSum[F, A, B] =
  (prod.left, prod.right) match
    case (Some(a), None)    => OptIorKSum.LeftK(a)
    case (None, Some(b))    => OptIorKSum.RightK(b)
    case (Some(a), Some(b)) => OptIorKSum.BothK(a, b)
    case (None, None)       => OptIorKSum.NeitherK
```

Figure 5.4: A sum and product definition of OptIorK, and isomorphisms between them.

called `Ior`. `Ior` is an inclusive or type and contrasts `Either`, which is exclusive. That is to say, `Either` can only contain `Left` or `Right`, not both simultaneously. `Ior` meanwhile can contain both at the same time. Figure 5.3 shows how `Ior` can be defined.

Putting these ideas together, one gets something with an `Applicative` instance. In fact, this new type is not at all exotic, as it also has a `Representable` instance. That is because this new type is both a sum type and a product type simultaneously. Figure 5.4 shows both ways of defining it, with functions to convert between the different representations.

This scheme also grows well to arbitrary sum types, and is the idea perspective uses for the type providing mappings between the normal and the generic representation.

5.3.1 HKDSumGeneric

perspective's generic typeclass for working with sum types, `HKDSumGeneric` (defined in Figure 5.5), reuses many of the ideas found in `HKDProductGeneric`. So similar is the handling of sum and product types in perspective that they share most of their code in a type called `HKDGeneric`. There are however some differences and extra functions `HKDSumGeneric` has access to.

```scala
trait HKDSumGeneric[A]:
  type Gen[_[_]]

  type ElemTop <: A

  def indexOf[X <: ElemTop](x: X): Index[X]
  def indexOfA(a: A): IdxWrapper[_ <: ElemTop] = indexOf(a.asInstanceOf[ElemTop])
  def indexOfACasting(a: A): HKDSumGeneric.IndexOfACasting[Index, ElemTop]

  inline def widenConst[F[+_]](gen: Gen[F]): Gen[Const[F[A]]] =
    // This is safe. We can't use the widen method as it can't know about the
    // contents of Gen, we do
    gen.asInstanceOf[Gen[Const[F[A]]]]

  def to(a: A): Gen[Option] =
    val index = indexOf(a.asInstanceOf[ElemTop])
    // This cast is safe as we know A = Z
    representable.tabulateK(
      [Z] => (i: Index[Z]) =>
        if i == index.idx then Some(a.asInstanceOf[Z]) else None
    )

  def from(a: Gen[Option]): Option[A] =
    traverse.toListK(widenConst(a)).flatten match
      case Nil      => None    // No values present
      case a :: Nil => Some(a) // One value present
      case _        => None    // More than one value present

  ...
```

Figure 5.5: perspective's HKDSumGeneric.

First off, the functions `to` and `from` still exist, but have different types. While `HKDProductGeneric` uses `A => Gen[Id]` and `Gen[Id] => A`, `HKDSumGeneric` uses `A => Gen[Option]` and `Gen[Option] => Option[Id]`. The function `to` creates a `Gen[Option]` where only one case is `Some`, and the rest are `None`. The function `from` only returns a `Some` if exactly one case in `Gen[Option]` is `Some`. Sadly this is not what a user will likely want and will require the user to call `get` on the `Option`, verifying that the call should always be safe. This is the only confirmed case of type unsafeness in perspective.

Unlike with `HKDProductGeneric`, `to` and `from` are also no longer the fundamental functions that everything is built on, and can instead be implemented from other functions provided only for sum types. If there is such a thing as a fundamental function for sum types, it is `indexOf`. Given a subtype of the type being worked on, it gives a correctly typed index to this type. It is important that the equality `gen.to(a).indexK(gen.indexOf(a)).get == a` always holds for all values.

The `indexOfA` function is a small utility that allows values of type `A` to be passed to `indexOf`. Values of type `A` cannot be directly passed to `indexOf` as this would not be type safe. The type `ElemTop` is a union of all the types that make up `A`, but it is not equal to `A`. Here is a small example of why `indexOf` would be unsafe with the wrong bound.

```
1  def deriveUnsafe[A](a: A)(
2    using gen: HKDSumGeneric[A], showInstances: gen.Gen[Show]
3  ): Show[A] =
4    showInstances.indexK(gen.indexOf(a))
5
6  enum Foo:
7    case FooString(s: String)
8    case FooInt(i: Int)
9
10 object Foo:
11   given Show[Foo.FooString] = (foo: Foo.FooString) => foo.s
12   given Show[Foo.FooInt] = (foo: Foo.FooInt) => foo.i.toString
13   given Show[Foo] = deriveUnsafe(Foo.FooString("")) //Unsafe
```

Within this code, the call to `deriveUnsafe` would also pass a value along to the derivation function together with the generic instance and show instances. The function then chooses a show instance based on the value passed in. If `indexOf` allowed types of `A`, it would return a value typed as `Index[A]`. This value can then be used to cast something that is a subtype of `A` to `A`. The function `indexOfA` bypasses this restriction by

"forgetting" the type it returns to the user. Note that this type unsoundness has not been fixed for the Scala 2 version of perspective. The reason for this is that Scala 2 does not have union types. `indexOf` is currently defined as `def indexOf[X <: A](v: X): Index[X]`. Because the lack of union types, the only real option to fix the unsoundness would be to only keep something like `indexOfA`, and lose functionality in the process. The Scala 2 version of perspective is also not particularly well maintained and feature-complete anyway, and is not where the focus of perspective lies.

The function `indexOfA` can sometimes be cumbersome to use as Scala does not play well with existential types. The function `indexOfACasting` helps with this by both returning index type as a dependent type but also casts the value to this dependent type. This function requires the equalities `gen.indexOfACasting(a).value == a` and `gen.indexOfACasting(a).index == gen.indexOfA(a)` to work.

Lastly, there is the function `widenConst`, which casts all the values in the generic representation to the type `A`. This operation can only be done if the type `F` is covariant. One example of this function is its use in the `from` function implementation. `Traverse#toListK` is a function that can be called on any value with the type `F[Const[Z]]` for some type `Z`. It then returns a `List[Z]`. The `widenConst` call here turns a value of type `Gen[Option]` into a value of type `Gen[Const[Option[A]]]`. After the call to `toListK`, this value is then a `List[Option[A]]`, which is then flattened down to a `List[A]`. The code then matches over the list, returning the first element as long as there are no other elements.

5.3.2 Using HKDSumGeneric

Using `HKDSumGeneric` is mostly like using `HKDProductGeneric`, with one extra complication. The code needs to also handle the subtypes of the type the derivation is being done for. The reason for this is that one way users might use typeclass derivation is with the `derives` keyword on Scala enums. This keyword creates a typeclass by calling a function named `derived` in the companion object of the typeclass being derived. Scala does nothing special for the subtypes of the enum, so the function doing the derivation needs to handle them instead. I have tried numerous ways to reduce the boilerplate required for this but have not found a way around it. Figure 5.6 shows how to derive a `Show` instance for sum types with this extra complication accounted for. The example also demonstrates what deriving typeclasses for algebraic data types more generally looks like.

There are three different functions to this snippet of code: `derived`, `caseShows`, and `deriveSumShow`.

```scala
import scala.compiletime.{erasedValue, summonFrom, summonInline}

inline def derived[A](using gen: HKDGeneric[A]): Show[A] = inline gen match
  case gen: HKDProductGeneric.Aux[A, gen.Gen] =>
    val shows = summonInline[gen.Gen[Show]]
    derivedProductShow
  case gen: HKDSumGeneric.Aux[A, gen.Gen] =>
    summonFrom {
      case given gen.Gen[Show] => deriveSumShow
      case _ =>
        given gen.Gen[Show] = gen.tupleToGen(
          caseShows[gen.TupleRep, Helpers.TupleMap[gen.TupleRep, Show]](
            Helpers.TupleBuilder.mkFor
          )
        )
        deriveSumShow
    }

private inline def caseShows[T <: Tuple, R <: Tuple](
  builder: Helpers.TupleBuilder[R]
): R = inline erasedValue[T] match
  case _: (h *: t) =>
    builder += summonFrom {
      case sh: Show[`h`] => sh
      case given HKDGeneric[`h`] => derived[h]
    }
    caseShows[t, R](builder)
  case _: EmptyTuple => builder.result

def deriveSumShow[A](
  using gen: HKDSumGeneric[A], instances: gen.Gen[Show]
): Show[A] = new Show[A]:
  extension (a: A) def show: String =
    import gen.given
    val casted = gen.indexOfACasting(a)
    instances.indexK(casted.index).show(casted.value)
```

Figure 5.6: Deriving data structures using perspective (derivedProductShow not shown).

The function `deriveSumShow` is the function that does the actual derivation of the typeclass for sum types. It gets the index of the case passed in using `indexOfACasting` to get both the index and the value cast to an easier to work with type. Using this index, the code then grabs a `Show` instance for the case passed in, and calls `show` using the casted value.

The function `derived` is the entry point to where the generic programming happens. The first thing the function does is to determine if it will derive an instance for a product or a sum type. For product types, the function then summons the instances it needs and calls `derivedProductShow` (not shown here) or `deriveSumShow` to do the actual typeclass derivation. For sum types, the code uses `summonFrom` to try and summon all the instances it needs for the subtypes. If the code finds an instance of `ge.Gen[Show]`, things go on as usual. If the code does not find such an instance, it calls `caseShows` that generates a tuple representation of these instances. This tuple instance is then converted to the generic representation, and the typeclass derivation goes on as usual.

The function `caseShows` has the job of, for each case in a sum type, finding or making the typeclass needed. First, it matches over the tuple representation. If the tuple forms a cons (`*:`), it looks for a typeclass for that type. If it does not find a typeclass, it finds a `HKDGeneric` instance instead and makes a typeclass instance for the case. This instance is then added to a tuple builder, which perspective provides. Once the entire tuple has been matched over, the finished tuple is returned. A tuple builder is used here, as building the tuple with cons (`*:`) has a time complexity of $O(n^2)$.

Chapter 6

Performance

This chapter will go over the performance of various ways of deriving typeclasses. It will go over how long the various methods take to compile code and how efficient that code is when run. While perspective has a much larger feature set in Scala 3, it also has a Scala 2 version, which will be tested as well.

The benchmarking was run on an Acer Aspire 7 laptop with a Ryzen 7 5700U and 16 GB of RAM using Java 17 (output from `java -version` is
`OpenJDK Runtime Environment Temurin-17+35 (build 17+35)` on the second line). A new Windows user was created to minimize possible interference with the benchmarking. All the startup programs were also disabled. A separate power plan was created to allow the laptop to always stay at high performance.

The benchmarks themselves were written using JMH [31]. JMH is a benchmarking harness for the JVM to get more accurate benchmarking results. Benchmarking on the JVM is tricky as it can do various things to optimize the code, which developers might not anticipate. JMH helps combat some of these optimizations that the JVM might perform and makes benchmarks more accurate.

This chapter will move away from the `Show` typeclass, which has been used as an example so far, and will instead use the JSON encoding and decoding typeclasses from the Circe [28] library.

6.1 The contestants

Here are the different types of typeclass derivation which were benchmarked.

circe Generic (Scala 2): In Scala 2, CirceGeneric is a derivation scheme offered by Circe using shapeless under the hood. It uses a macro, but only to unpack the HList shapeless gives it.

circe Generic (Scala 3): In Scala 3, CirceGeneric is implemented using Scala 3's Mirror.

circe Derivation (Scala 2): A custom macro provided by Circe which has less functionality but performs and compiles quicker. This is the gold standard within all the Scala 2 benchmarks.

Handwritten: Idiomatically handwritten code.

shapeless 2: A custom solution using shapeless 2 and type level programming. The Scala 3 code uses an incomplete port of shapeless 2 to Scala 3 which I worked on [11]. While the entire library does not quite work for Scala 3 yet, the parts used (HLists and similar) do. For encoding products, a typeclass that first gathers all the fields up in a list is used. This is to give shapeless 2 a fair chance, as the naive solution is far less performant.

shapeless 3 (Scala 3): A custom solution built on the functionality shapeless 3 exposes.

perspective: perspective as an inexperienced developer would use it. No usage of indices of any type.

perspective faster: perspective used with indices.

perspective inline (Scala 3): perspective with its inline functionality.

perspective unrolling (Scala 3): perspective inline with unrolling enabled, and attempts made to avoid boxing.

The decompiled code for CirceGeneric, CirceDerivation, PerspectiveInline and PerspectiveUnrolling can be found in Appendix B.

For sum types, n cases with a single value were made. The benchmark then tried to encode and decode the last case. Ideally encoding and decoding sum cases should take a constant amount of time, and not grow with the amount of cases. Benchmarks that tried to encode and decode the first and middle case were also run, but are not discussed in this chapter. The results can be found in Appendix C.

Allocation rate is a useful metric to track in the benchmarks, but it turned out that this closely followed the execution time of a given benchmark, with a few exceptions. As such these benchmarks were not included in this chapter. Alloation benchmarks can be found in Appendix C.

The benchmarks all have data points at both size 22 and size 23 case classes because Scala 2 stops making tuples after size 22. Scala 3 still supports tuples bigger than 22 fields but with an array backing them instead. perspective for Scala 2 also handles products bigger than 22 differently.

The benchmark results shown here show what I got when benchmarking and in some cases writing encoders and decoders with the given tools. An experienced user with these tools might be able to get more performance out of them.

Compile time benchmarks compile a circe product encoder and decoder of the given size.

All charts have the Y-axis in log scale to better show the difference in performance along each step.

The benchmark code and the raw results obtained from running the benchmarks can be found in the *perspective-derivation-performance* [18] Github repository at commit `d60a797247a566b1bbb242d6ce675a9cffae0111`.

6.2 Scala 2

6.2.1 Runtime

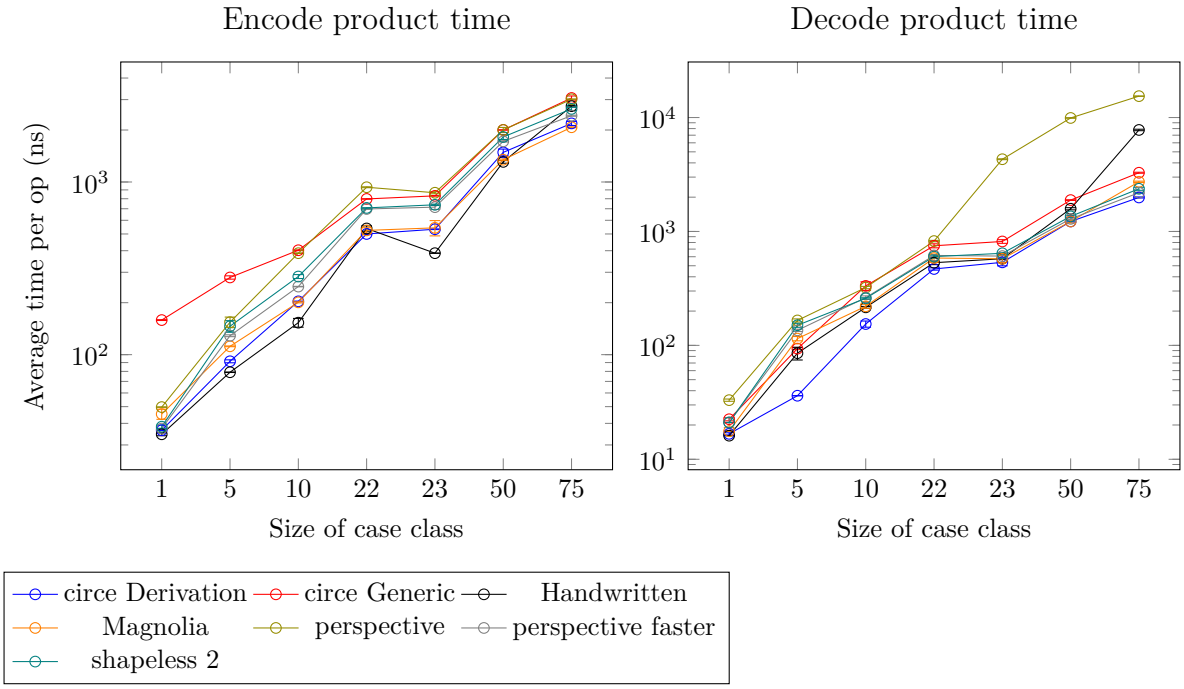

Figure 6.1: Product type Scala 2 performance.

Figure 6.1 shows the time taken to encode and decode a product type to and from circe's abstract syntax tree for JSON with Scala 2.

Most methods perform about the same. perspective is a bit behind, but this is to be expected as it does not use indices. perspective with indieces (perspective faster) performs more as expected. circe derivation is also quite fast in both of these benchmarks, with Magnolia following closely in encoding. What I am more surprised by is that circe generic is so close to shapeless, so I wonder why it uses a macro to unpack the HList. The handwritten code performs quite well for small cases, but as the size of the case class grows past 50, it starts performing a lot worse. This is a pattern that will return with other derivation methods as well in future benchmarks.

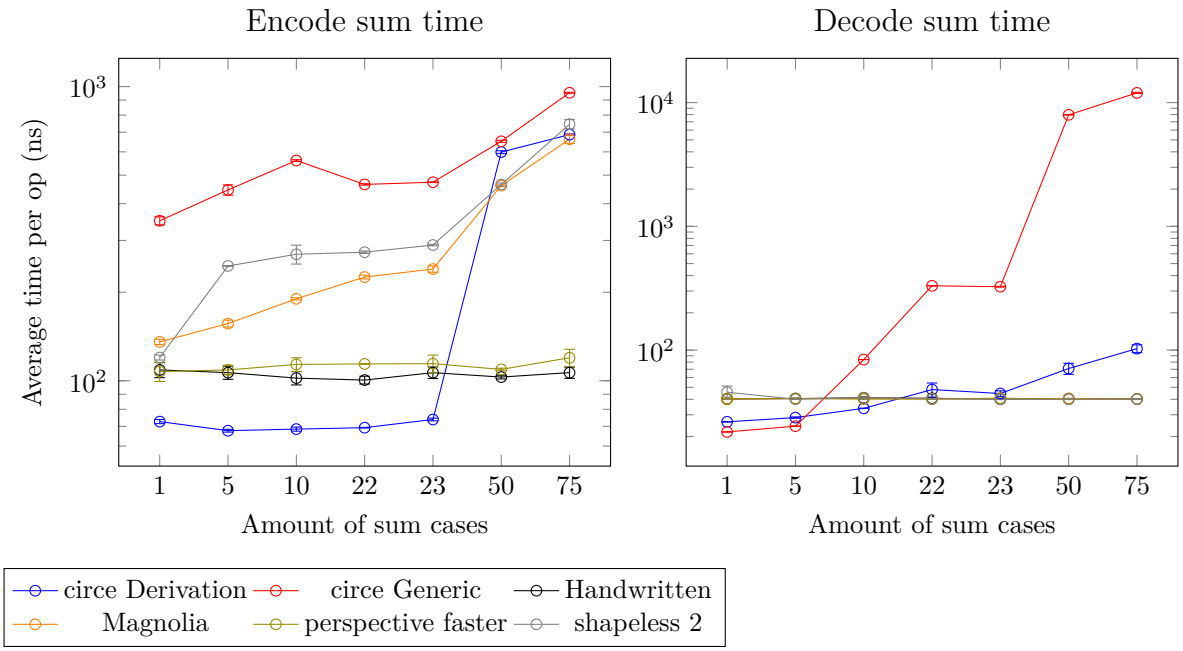

Figure 6.2: Last sum case Scala 2 performance.

Figure 6.2 shows the time taken to encode and decode a sum type's last case to and from circe's abstract syntax tree for JSON with Scala 2.

Ideally, everything here should be constant time, or near constant time. For encoding, only perspective, handwritten, and circe derivation below 22 fulfills this criteria. For decoding, everything except circe generic fulfills this criteria. While circe-derivation is not completely constant, it grows slowly enough that it does not matter too much.

6.2.2 Compile time

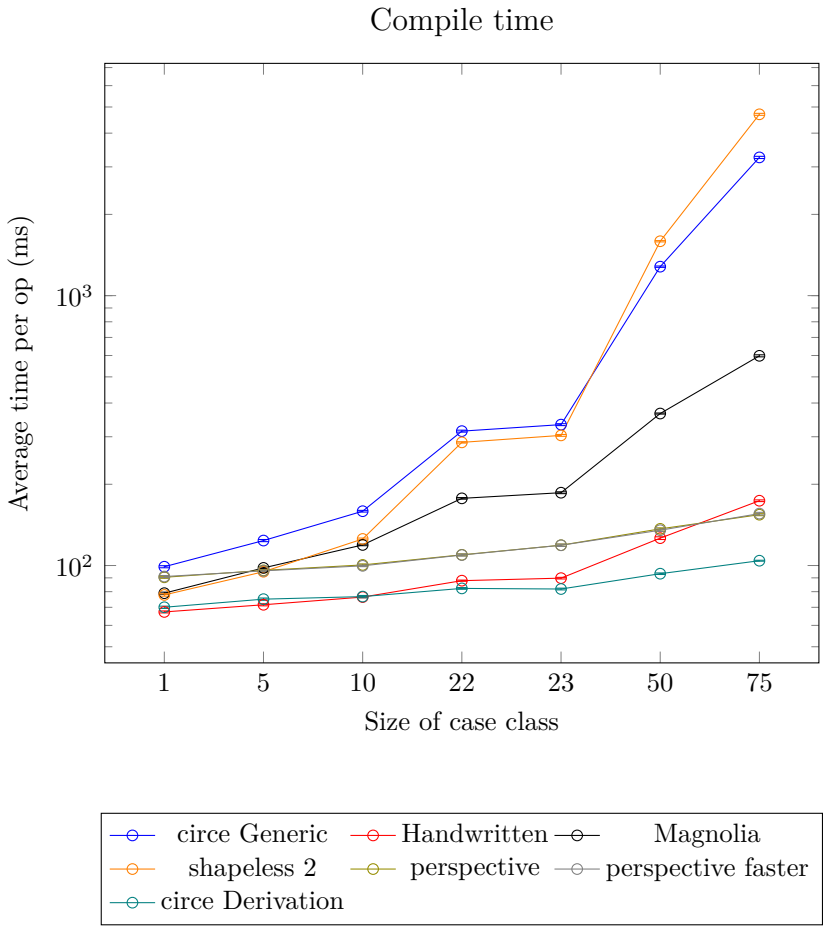

Figure 6.3: Scala 2 Compile performance.

Figure 6.3 shows the time it takes to compile a set of JSON encoder and decoder typeclasses derived in various ways with Scala 2.

When it comes to compile time, the different methods of typeclass derivation differentiate themselves a bit more. circe generic and shapeless are the slowest, as circe depends on shapeless. Magnolia not too far behind, meaning that while it offered nice performance at runtime, compile time leaves something to be desired. perspective sits a bit below that, with both variants taking the same amount of time to compile. Fastest to compile is circe derivation and handwritten code, although as the case class gets large enough, handwritten code grows a bit faster than the other methods.

6.3 Scala 3

6.3.1 Runtime

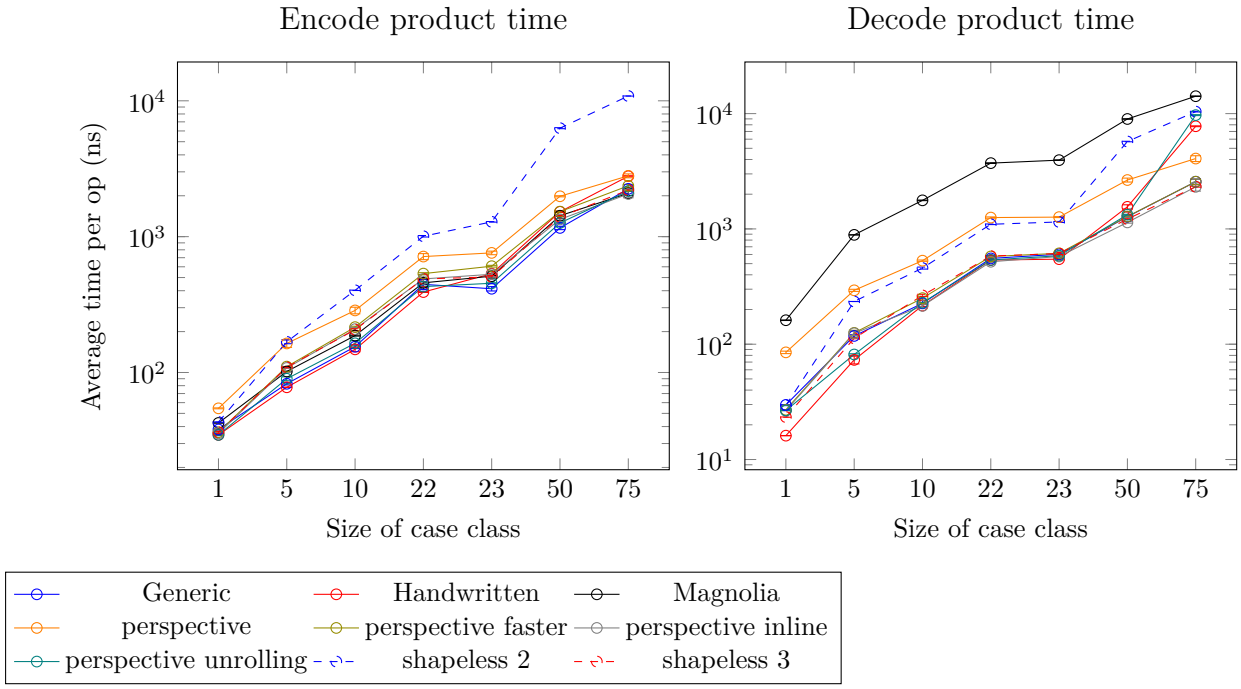

Figure 6.4: Product type Scala 3 performance.

Figure 6.4 shows the time taken to encode and decode a product type to and from circe's abstract syntax tree for JSON with Scala 3.

Next is running with Scala 3. Here things are much more equal, although some patterns do repeat them selves, like perspective without indices being slow. To note here again is that the version of shapeless 2 running in the Scala 3 benchmarks is not released, and performance might change on release. I do not really know why Shapeless 2 runs so badly with Scala 3. Interestingly, Magnolia also performs far worse on decoding using Scala 3 than it did with Scala 2.

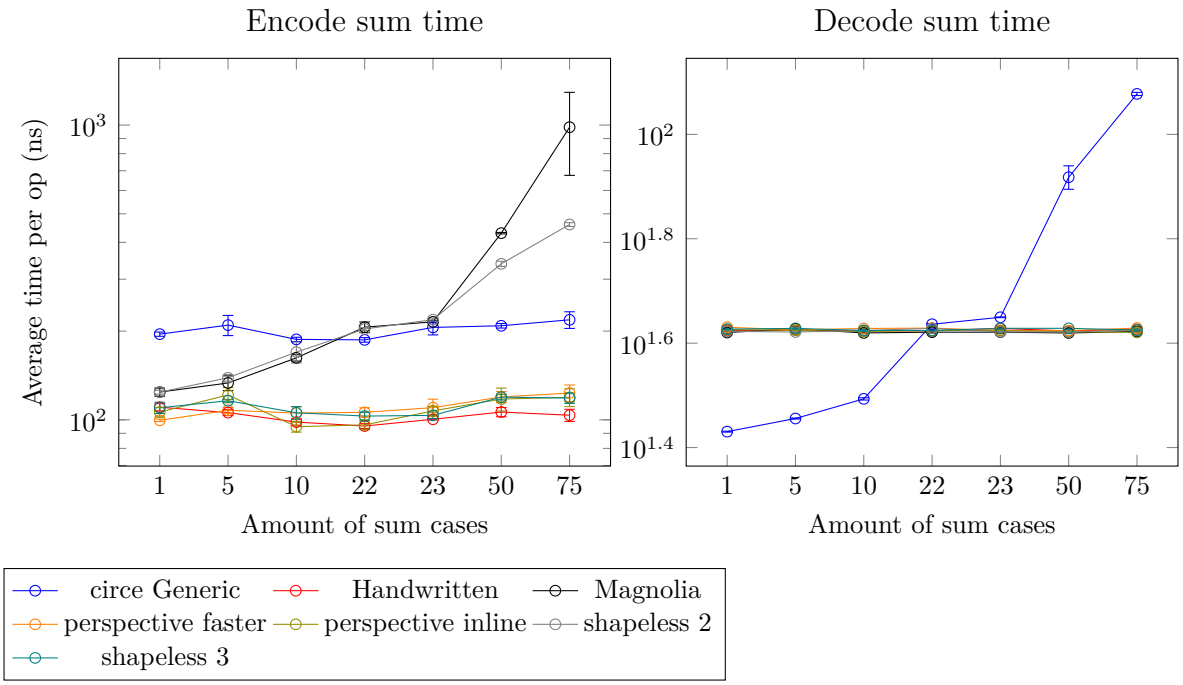

Figure 6.5: Last sum case Scala 3 performance.

Figure 6.5 shows the time taken to encode and decode a sum type's last case to and from circe's abstract syntax tree for JSON with Scala 3.

Unlike with encoding and decoding sum types with Scala 2, using Scala 3 things are more equal. shapeless 2 and Magnolia are still not quite constant time. circe generic is also not constant time when decoding, and has a significant overhead when encoding. The reason most libraries have a roughly equal runtime might be because Scala 3 offers `Mirror.Sum` which provides a function to get the ordinal of a sum type case.

6.3.2 Compile time

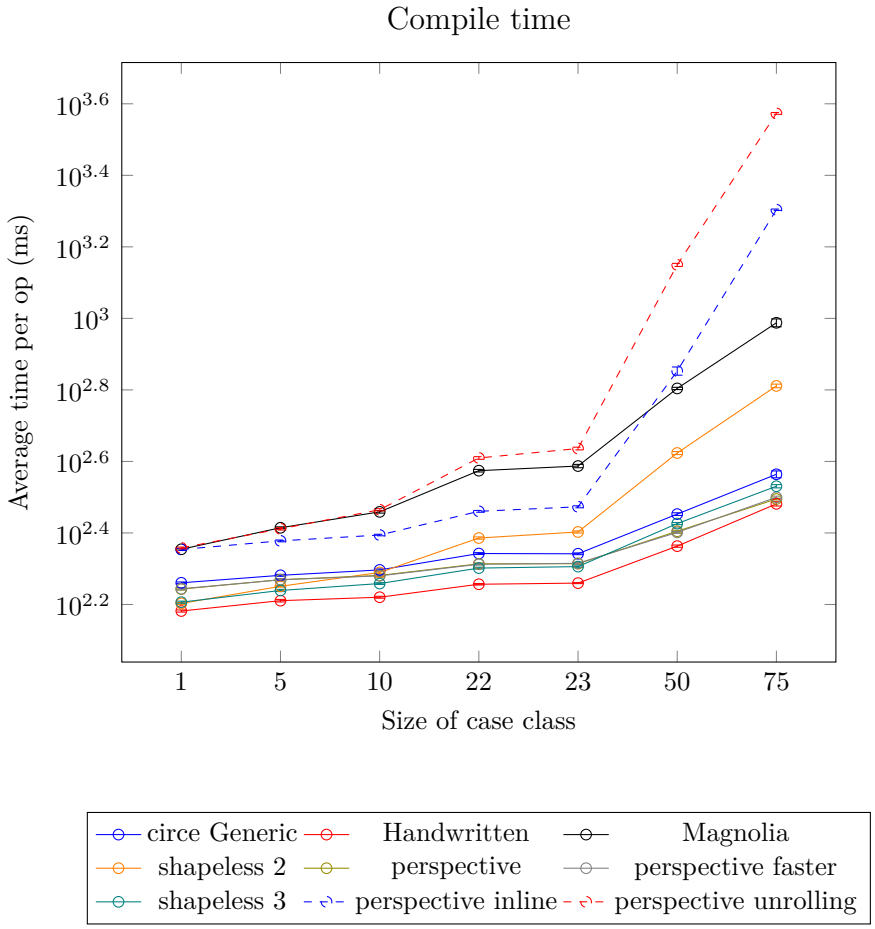

Figure 6.6: Scala 3 Compile performance.

Figure 6.6 shows the time it takes to compile a set of JSON encoder and decoder typeclasses derived in various ways with Scala 3.

Finally we get to the compile time using Scala 3. Something that becomes clear at once is that perspective inline and unrolling join Magnolia in terms of having slow compile times. Derivation methods using `Mirror` with few to no macros like perspective, shapeless 3 and circe generic all perform fairly similarly. Handwritten code however remains the clear winner in terms of compile time.

I should also note other annoyances I encountered while compiling large case classes with the various derivation schemes. circe generic and shapeless 3 require me to pass `-Xmax-inlines 128` to the compiler. Setting it to around `75`, the size of the largest types, might have been sufficient. Magnolia also ran out of stack space while compiling

the code. I tried to 2 MiB of stack size, which was not enough. 8 MiB seems to have been enough.

6.4 Discussion on performance

The first thing to point out here is that while there were differences in the benchmark results, they were quite small, with a few exceptions. This means that unless performance is really important, it is probably smarter for developers to use a typeclass generation scheme that is otherwise best for their needs.

As mentioned earlier, a lot of things change around 22-23 number of elements in sum and product types. If code for 23 elements is somehow faster than code for 22 elements, it might indicate lost performance potential. Circe generic and handwritten code are the only schemes where this is prominent.

The perspective library has two variants which use macros, inlining and unrolling. For the time it takes to encode product types, inlining and unrolling is slightly faster than many other alternatives. This might be because they unnecessary object creation whenever possible. For decoding however, unrolling is only faster when the case classes it deals with are themselves small. This might be because it can not prevent boxing here, and has to rely on the JVM preferring the bytecode it generates. There is little performance difference between inlining and non-inlining.

A future goal for perspective would be to reduce the compile times for inlining and unrolling while keeping the performance advantage these derivation schemes have. This could for example be done by dropping less important features meant to be used at the type level.

Chapter 7

Example application: DataPrism

The chapters so far have all talked about typeclass derivation together with generic programming. While this is a common use case, it is far from the only one. This chapter will go over a prototype language integrated query library called DataPrism [15] created using perspective. It will be a very different use case than the typeclass derivation we have covered so far, but the underlying ideas are essentially the same. The code shown here represents the fundamental ideas of DataPrism. There is far more than shown here, and not everything will be discussed. I will not go much into the implementation of things here, and as such, I will mostly only show the abstract definition for things.

7.1 Two different worlds

When dealing with a language integrated query library, one is working with two different worlds at the same time. The first world is the "normal" world, with "normal" values. These values have "normal" unwrapped types. An integer is an `Int`, and a string is a `String`. The second world is the wrapped world, where one is handling data that exists elsewhere. In this case, that elsewhere will be a database server. In this world, an integer is no longer just an `Int`, as one cannot, e.g., pass it to functions in a codebase that accept an `Int`. DataPrism does this by giving wrapped values new types. For `Int` this wrapped type is `DbValue[Int]`. In DataPrism, all values in the wrapped world are wrapped in `DbValue`, while values in the normal world could be said to be wrapped in `Id`. This distinction works nicely with the idea of higher kinded data, which DataPrism uses a lot.

```scala
case class Table[A[_[_]]](
  tableName: String,
  columns: A[Column]
)(using val FA: ApplyKC[A], val FT: TraverseKC[A])

case class Column[A](name: String, tpe: DbType[A])

case class DbType[A](
  name: String,
  get: (ResultSet, Int) => A,
  set: (PreparedStatement, Int, A) => Unit
)
object DbType:
  val int32: DbType[Int]    = DbType("INT32", _.getInt(_), _.setInt(_, _))
  val int64: DbType[Long]   = DbType("INT64", _.getLong(_), _.setLong(_, _))
  val float: DbType[Float] = DbType("REAL", _.getFloat(_), _.setFloat(_, _))
  val text: DbType[String] = DbType("TEXT", _.getString(_), _.setString(_, _))
  ...

  def array[A: ClassTag](inner: DbType[A]): DbType[Seq[A]] = DbType(
    "ARRAY" + inner.name,
    _.getObject(_).asInstanceOf[Array[A]].toSeq,
    (a, b, c) => a.setObject(b, c.toArray)
  )

  def nullable[A](inner: DbType[A])(
    using NotGiven[A <:< Option[_]]
  ): DbType[Option[A]] = DbType(
    inner.name,
    (a, b) => Option(inner.get(a, b)),
    (a, b, c) => inner.set(a, b, c.orNull)
  )

```

Figure 7.1: DataPrism's Table, Column and DbType.

7.2 Tables

Let start with defining tables in DataPrism. A table consists of two things, the table name, and the columns as higher kinded data with the `Column` type. A column consist of the column name and a value indicating the SQL type of the column. The values indicating the SQL types give information on how to get the specified types from the result and how to set a value of the given type in a prepared statement. The value indicating the SQL type of the column are essentially typeclasses, except that they are passed explicitly because I think it is important to be explicit in what types are being handled. Tables also require an apply and traverse instance for this higher kinded data. These can be generated using macros. Definitions for all of these can be seen in Figure 7.1.

7.3 Simple selects

With a table defined, the next step is to perform some select queries using this table. For this, a query type and a function that lifts the table into this query type is needed. A few functions that run queries is also needed. Note that running queries is done quite differently in the actual library. The functions presented here are simplifications that do not exist in the way shown here. In the actual library, operations are built using a series of functions to ensure the operations are valid. There is also just a single run function, which returns an object where the developer then specifies how many rows they were expecting. In these examples, those two steps have been merged.

```
type Query[A[_[_]]]

type QueryCompanion
val Query: QueryCompanion

extension (q: QueryCompanion) def from[A[_[_]]](table: Table[A]): Query[A]

extension [A[_[_]]](q: Query[A])
  def runMany: Future[Seq[A[Id]]]
  def runSingle: Future[A[Id]]
  def runOptional: Future[Option[A[Id]]]
```

We have a few different run functions here: `runMany` runs the query and returns all rows; `runSingle` grabs only the first row, failing if one does not exist; and `runOptional` tries to grab the first row, returning None if no results were returned. A question that might crop up is why the code returns `F[A[Id]]` instead of `A[F]` where `F` is a type like `Option` or `Seq`. The reason is that for `F[A[Id]]`, all the results are correlated. For example, all the columns are missing, or none of them are. In `A[Option]`, just because the first column is missing does not mean the second column is. This problem becomes even more visible with `Seq`, where there is no guarantee that all of the columns contain the same number of values. `A[F]` also becomes problematic when dealing with nullable columns.

7.4 filter, map, and groupBy

The next step would be to define filter, map, and groupBy operators for the query type. These correspond to the SQL operators `WHERE`, `SELECT`, and `GROUP BY`.

```
extension [A[_[_]]](q: Query[A])
  def filter(f: A[DbValue] => DbValue[Boolean]): Query[A]

  def map[B[_[_]]: ApplicativeK: TraverseK](f: A[DbValue] => B[DbValue]): Query[B]

  def groupMap[B[_[_]]: TraverseKC, C[_[_]]: ApplicativeK: TraverseK](
    group: A[DbValue] => B[DbValue]
  )(map: (B[DbValue], A[Many]) => C[DbValue]): C[DbValue]
```

The filter function works mostly as one might expect. It turns the current columns in the query into a wrapped boolean. The map function allows one to change the "shape" of the query and the columns the query is dealing with. DataPrism does not define a `groupBy` function but instead defines a `groupMap` function. Providing a `groupMap` function instead of a `groupBy` function allows for both better ergonomics and allows DataPrism to take advantage of its higher kinded data again. In the group function, the user must construct wrapped values that will be grouped against. The map call then allows access to these grouped values while also giving access to the original values of the query, but typed as `Many` instead of `DbValue`. The user must then use aggregation functions to regain `DbValue` values. Because of this, queries cannot fail when using SQLs `GROUP BY`.

7.5 Joins

Joins will be covered next and are shown in Figure 7.2. These work mostly like the filter function but with two queries. A match type is also used to avoid multiple wrappings of `Option`. Higher kinded types are again used to annotate fields as nullable when needed.

7.6 Insert and update

DataPrism also allows for using higher kinded types when doing updates and inserts. If one wants to set or update all the rows that a table has defined, one does not need to do anything fancy. Given an `A[Id]`, one can insert it with little problem. Sometimes, however, one wants to avoid setting some rows when updating and inserting data. Updating and inserting rows in this way can be done using `Option`. One can leave any row one does not

```
type Nullable[A] = A match {
  case Option[b] => Option[b]
  case _         => Option[A]
}

type Compose2[A[_], B[_]] = [Z] =>> A[B[Z]]

type InnerJoin[A[_[_]], B[_[_]]] = [F[_]] =>> (A[F], B[F])
type LeftJoin[A[_[_]], B[_[_]]] = [F[_]] =>> (A[F], B[Compose2[F, Nullable]])
type RightJoin[A[_[_]], B[_[_]]]  = [F[_]] =>> (A[Compose2[F, Nullable]], B[F])
type FullJoin[A[_[_]], B[_[_]]]  =
  [F[_]] =>> (A[Compose2[F, Nullable]], B[Compose2[F, Nullable]])

extension [A[_[_]]](lhs: Query[A])
  def join[B[_[_]]](rhs: Query[B])(
    f: (A[DbValue], B[DbValue]) => DbValue[Boolean]
  ): Query[InnerJoin[A, B]]

  def leftJoin[B[_[_]]](rhs: Query[B])(
    f: (A[DbValue], B[DbValue]) => DbValue[Boolean]
  ): Query[LeftJoin[A, B]]

  def rightJoin[B[_[_]]](rhs: Query[B])(
    f: (A[DbValue], B[DbValue]) => DbValue[Boolean]
  ): Query[RightJoin[A, B]]

  def fullJoin[B[_[_]]](rhs: Query[B])(
    f: (A[DbValue], B[DbValue]) => DbValue[Boolean]
  ): Query[FullJoin[A, B]]
```

Figure 7.2: DataPrism's joins.

```
extension [A[_[_]]](q: Table[A])
  def insert(value: A[Option]): Future[Int]

  def insertQuery(value: Query[[F[_]] => A[Compose2[Option, F]]])

  def update(where: A[DbValue] => DbValue[Boolean])(
    setValues: A[DbValue] => A[Compose2[Option, DbValue]]
  ): Future[Int]
```

Figure 7.3: DataPrism's insert and update.

want to set as `None` instead. Figure 7.3 shows how insert and update could look, using this idea.

Because the code in Figure 7.3 uses the type `Option[DbValue[Z]]` for some type `Z`, the code can determine which columns to set and which to ignore when the query is being generated. If the code instead used `DbValue[Option[Z]]`, this would not be possible. Entire queries can also be inserted this way using exotic higher kinded data.

7.7 Comparision with existing libraries

Two existing prominent libraries in the Scala ecosystem within the same space are Slick [22] and Quill [7]. There is also a Haskell SQL library called Beam [6], which is also built on higher kinded data. Here we focus on Scala libraries and will not make comparisons between Beam and DataPrism.

Slick uses a lot of tuples and advanced typed programming to deal with the database. It generates the queries from the given operations at runtime. It requires that the user defines the wrapped representation and how the wrapped representation maps to the unwrapped representation. If wanted, this can be done with code generation, where the code generation reads an SQL schema and generates wrapped and unwrapped representations.

Quill does most of its work at compile time using macros. The unwrapped representation is also the wrapped representation. Unless configured otherwise, it directly gets the name of the columns and tables from the case class. It gets around the problems associated with sharing the representation by verifying everything at compile time.

DataPrism sits somewhere between these two. Like Slick, DataPrism does most of its operations at runtime. It, however, shares the intermediary representation with the unwrapped representation but keeps their types different with higher kinded types.
DataPrism requires the user to explicitly supply the names of the columns and table, like Slick, but technically nothing stops DataPrism from inferring this data automatically. Requiring that the user supplies the column name, type, and table name is an intentional choice and not a choice made because of technological limitations, like with Slick.

At this point in time, DataPrism is also very light, sitting under 3000 lines of code. This count might increase a bit as more functions are implemented and more databases are supported, but it is unlikely that the codebase will grow significantly.

7.7.1 Exotic higher kinded data and DataPrism

Something that sets DataPrism apart from Slick and Quill is its leniency with nested data.

Slick supports nested data as long as it has been told how this data should be represented when wrapping and unwrapping. In practice, this means that anything other than a simple case class as the unwrapped representation can quickly become cumbersome. Quill supports nested fields in case classes where those fields are themselves case classes.

DataPrism meanwhile accepts anything that has instances for `ApplyKC` and `TraverseKC`. That means that a query like `table.map(t => List(t.foo, t.bar, t.baz))` can be perfectly legal. Slick might be able to "learn" how to do this with enough code supporting this feature, while for Quill, this is simply impossible, as Quill cannot differentiate between `List[F[A]]` and `F[List[A]]`. That is to say, Quill cannot know if the list is a wrapped value or if it only contains wrapped values.

A clear example where DataPrism takes advantage of exotic higher kinded data is for update and insert queries (values of type `Query[Option[A]]` are also generated in the implementation of insert).

DataPrism's use and allowance of exotic higher kinded data questions what it means for a data type to be flat. Much of the literature mentions flat vs. nested results [3], but a clear definition of what flat and nested means is not given. If flat means that the size of the data is known at compile time, then it does seem like DataPrism allows nested data. However, if one only requires that the data being worked on can be converted into flat data, then yes, DataPrism works with flat data like most other SQL query generation libraries.

Chapter 8

Conclusion

This thesis has discussed datatype-generic programming and its manifestation in Scala. Most of the focus has been on deriving typeclasses with datatype-generic programming. It has explained how to think about datatype-generic programming, the tools Scala exposes for this kind of programming, and introduced some libraries intended for making it easier.

The main new contribution this thesis presents is perspective, a library for datatype-generic programming using higher kinded data. perspective defines higher kinded versions of normal typeclasses like `Functor`, `Applicative`, `Foldable`, `Traverse` and more. With these tools, the programmer can apply datatype-generic programming to arbitrary data types in much the same way the programmer might work on lists. Furthermore, for more advanced datatype-generic programing perspective also includes a higher kinded version of `Representable`. This allows a developer to index and tabulate over a data structure, performing some operation on each of the fields of the data structure.

The perspective library further takes these ideas and exposes the type `HKDGeneric` that functions can use to perform datatype-generic programming with normal, non-higher kinded data. It is these typeclasses that most users will use when performing typeclass derivation. perspective also provides a different typeclass, `InlineHKDGeneric`, which can be used to generate compact and fast bytecode using inline functions. `InlineHKDGeneric` also provides a limited form of unrolling for even faster bytecode and less boxing in certain cases.

The thesis also discusses how datatype-generic programming is applied on sum types and what different libraries do differently with sum types. It also shows how perspective's

Title	Issue id	Comment
Type lambda don't work with implicit search in Dotty, does in Scala 2	7452	From the SQL library that started it all. In DataPrism, the type Abstract is today called MapRes, and works fairly similarly.
Polymorphic function types and normal types don't mash	9663	Perspective's foldLeftK function did not initially play well with polymorphic function
Assertion failed when not adding prefix to extension method in method body	11318	Problems with Representable
Poly functions are not beta reduced	15968	Scala 3 offers some tools to beta reduce functions in macros. This did not at the time work for polymorphic functions, forcing perspective to user another encoding of polymorphic functions instead which can still be found in `InlineHKDGeneric` today.
Crash with dependent types and match type	15983	perspective often combines types in different ways, and sometimes the Scala 3 compiler does not like dealing with some of these combinations.
asExprOf fails for tuples larger than 22	17257	In an attempt to make the macro code in perspective safer, many casts were changed to `asExprOf` instead to let the Scala compiler check them at compile time. This did not work for larger case classes. This bug was found while optimizing compile time performance for the benchmarks.
Performance of compiletime tuple construction	15988	Constructing tuples with cons (`*:`) currently has a runtime time complexity of $O(n^2)$. As such perspective avoids doing this in as many cases as possible, and offers a type `TupleBuilder` to do this in user code.

Figure 8.1: Scala 3 compiler bugs encountered while making perspective.

approach fits mostly seamlessly with sum types by using higher kinded types where each field is of the type `Option` instead of `Id` that is used for product types.

The different ways of doing typeclass derivation are then benchmarked in terms of runtime and compile time performance. perspective does quite well in terms of runtime performance but has room to grow in terms of compile time performance. This is especially true for `InlineHKDGeneric`.

Lastly, an example SQL query construction library called DataPrism built on perspective is demonstrated. This library makes plenty use of higher kinded data and can do some things other libraries in this space cannot, like updating a specific selection of columns with one operation. DataPrism can also handle exotic higher kinded data, which means it is less strict with the queries it generates.

My hope with perspective is that it can push forward new ideas surrounding datatype-generic programming in Scala. perspective has already had some impact in the Scala community. While developing perspective, I ran into many bugs in the Scala 3 compiler, many of which have been fixed. A list and explanation of these bugs can be found in Figure 8.1.

perspective is also part of Scala 3's community build, which means that the Scala 3 compiler will regularly try to compile perspective. It should be noted that the version of perspective in the community build is an older version, but it has still managed to have some impact [20, 4, 5].

Bibliography

[1] V. Bragilevsky. *Haskell in Depth*. Manning, 2021. ISBN 9781617295409.

[2] Jorge Vicente Cantero. Speeding up compilation time with scalac-profiling, 2018. **URL:** `https://www.scala-lang.org/blog/2018/06/04/scalac-profiling.html`. Accessed: 2023-05–13.

[3] James Cheney, Sam Lindley, and Philip Wadler. A practical theory of language-integrated query. *ACM SIGPLAN Notices*, 48(9):403–416, 2013.

[4] Jan Chyb. Regression in katrix/perspective (stack overflow with match types), 2023. **URL:** `https://github.com/lampepfl/dotty/issues/16706`. Accessed: 2023-04-25.

[5] Jan Chyb. Regression in katrix/perspective (non-reducible tuple type in match types), 2023. **URL:** `https://github.com/lampepfl/dotty/issues/16707`. Accessed: 2023-04-25.

[6] Beam contributors. Beam: a type-safe, non-TH Haskell relational database library and ORM, 2023. **URL:** `https://github.com/haskell-beam/beam/`. Accessed: 2023-04-25.

[7] Quill contributors. Quill, 2023. **URL:** `https://github.com/zio/zio-quill`. Accessed: 2023-04-25.

[8] Scala contributors. PartialOrdering, 2020. **URL:** `https://dotty.epfl.ch/api/scala/math/PartialOrdering.html`. Accessed: 2023-05-13.

[9] Scala contributors. Ordering, 2021. **URL:** `https://dotty.epfl.ch/api/scala/math/Ordering.html`. Accessed: 2023-05-13.

[10] TypeScript documentation contributors. Generics, 2023. **URL:** `https://www.typescriptlang.org/docs/handbook/2/generics.html`. Accessed: 2023-04-17.

[11] Kathryn Frid. Scala 3 port, 2021.
URL: https://github.com/milessabin/shapeless/pull/1200. A port of shapeless 2 to Scala 3. Accessed: 2023-04-25.

[12] Kathryn Frid. stupidIdea.scala, 2021.
URL: https://gist.github.com/Katrix/b68cd01305c466292ff99d94c4c89674. Accessed: 2023-05-13. An example of a Turing machine in Scala using match types.

[13] Kathryn Frid. Math using implicits no longer works, 2022.
URL: https://github.com/lampepfl/dotty/issues/15692. Accessed: 2023-05-16.

[14] Kathryn Frid. AckCord, 2023.
URL: https://github.com/Katrix/AckCord. Accessed: 2023-04-17.

[15] Kathryn Frid. DataPrism, 2023.
URL: https://github.com/Katrix/DataPrism. Accessed: 2023-04-26.

[16] Kathryn Frid. Poly functions are not beta reduced, 2023.
URL: https://github.com/lampepfl/dotty/issues/15968. Accessed: 2023-04-25.

[17] Kathryn Frid. perspective, 2023.
URL: https://github.com/Katrix/perspective. Accessed: 2023-04-26.

[18] Kathryn Frid. perspective-derivation-performance, 2023.
URL: https://github.com/Katrix/perspective-derivation-performance. Accessed: 2023-05-27.

[19] Jeremy Gibbons. Datatype-generic programming. *Lecture Notes in Computer Science*, 4719:1, 2007.

[20] Tom Grigg. Pickling crash compiling perspective from community build, 2021.
URL: https://github.com/lampepfl/dotty/issues/13660. Accessed: 2023-04-25.

[21] Dave Gurnell. *The Type Astronaut's Guide to Shapeless*. Underscore Consulting LLP, 2017.

[22] Lightbend Inc. and Slick contributors. Slick, 2023.
URL: https://github.com/slick/slick. Accessed: 2023-04-25.

[23] Edward Kmett. Data.Distributive, 2016.
URL: https://hackage.haskell.org/package/distributive-0.6.2.1/docs/Data-Distributive.html. Accessed: 2023-04-25.

[24] LAMP/EPFL and Dotty contributors. CanThrow capabilities, 2022.
URL: http://dotty.epfl.ch/docs/reference/experimental/canthrow.html. Accessed: 2023-04-19.

[25] LAMP/EPFL and Dotty contributors. Capture checking, 2022.
URL: http://dotty.epfl.ch/docs/reference/experimental/cc.html. Accessed: 2023-04-19.

[26] LAMP/EPFL and Dotty contributors. Erased definitions, 2022.
URL: https://docs.scala-lang.org/scala3/reference/experimental/erased-defs.html. Accessed: 2023-05-23.

[27] George Leontiev. There's a prolog in your scala, 2014.
URL: https://www.youtube.com/watch?v=iYCR2wzfdUs. Accessed: 2023-04-19.

[28] Ephox Pty Ltd, Mark Hibberd, Sean Parsons, Travis Brown, and Circe contributors. circe, 2023.
URL: https://github.com/circe/circe. Accessed: 2023-04-25.

[29] Bartosz Milewski. *Category theory for programmers*. Blurb, 2018. ISBN 9780464243878.

[30] Oracle. Lesson: Generics (updated).
URL: https://docs.oracle.com/javase/tutorial/java/generics/index.html. Accessed: 2023-04-17.

[31] Oracle and JMH contributors. Java Microbenchmark Harness (JMH), 2023.
URL: https://github.com/openjdk/jmh. Accessed: 2023-04-25.

[32] Erik Osheim and Typelevel. Kind Projector, 2023.
URL: https://github.com/typelevel/kind-projector. Accessed: 2023-04-16.

[33] Miles Sabin and shapeless contributors. shapeless: generic programming for scala, 2023.
URL: https://github.com/milessabin/shapeless. shapeless 2 version. Accessed: 2023-04-25.

[34] Miles Sabin and shapeless contributors. shapeless: generic programming for scala, 2023.
URL: https://github.com/typelevel/shapeless-3. shapeless 3 version. Accessed: 2023-04-25.

[35] Typelevel and Cats contributors. RepresentableLaws.scala, 2018.
URL: https://github.com/typelevel/cats/blob/main/laws/src/main/scala/cats/laws/RepresentableLaws.scala. Accessed: 2023-04-25.

[36] Typelevel and Cats contributors. TraverseLaws.scala, 2022.
URL: https://github.com/typelevel/cats/blob/main/laws/src/main/scala/cats/laws/TraverseLaws.scala. Accessed: 2023-04-25.

[37] Typelevel and Cats contributors. Applicative, 2023.
URL: https://typelevel.org/cats/typeclasses/applicative.html. Accessed: 2023-04-25.

[38] Typelevel and Cats contributors. Foldable, 2023.
URL: https://typelevel.org/cats/typeclasses/foldable.html. Accessed: 2023-05-25.

[39] Typelevel and Cats contributors. FoldableLaws.scala, 2023.
URL: https://github.com/typelevel/cats/blob/main/laws/src/main/scala/cats/laws/FoldableLaws.scala.

[40] Typelevel and Cats contributors. Functor, 2023.
URL: https://typelevel.org/cats/typeclasses/functor.html. Accessed: 2023-04-25.

[41] Vlad Ureche, Cristian Talau, and Martin Odersky. Miniboxing: improving the speed to code size tradeoff in parametric polymorphism translations. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 73–92, 2013.

[42] Bill Wagner and Youssef Victor. Generic classes and methods, 2022.
URL: https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/types/generics. Accessed: 2023-04-17.

Appendix A

Proof for typeclass instances for products of arbitrary size

A.1 Apply

A.1.1 Base Step

- Start:

```
fa.tupledK(fb).tupledK(fc) =
  fa.tupledK(fb.tupledK(fc)).mapK{ [Z] => (ft: (A[Z], (B[Z], C[Z]))) =>
    ((ft._1, ft._2._1), ft._2._2)
  }
```

- Unwrapping the first left tupledK:

```
(fa, fb).tupledK(fc) =
  fa.tupledK(fb.tupledK(fc)).mapK{ [Z] => (ft: (A[Z], (B[Z], C[Z]))) =>
    ((ft._1, ft._2._1), ft._2._2)
  }
```

- Unwrapping the second left tupledK:

```
((fa, fb), fc) =
  fa.tupledK(fb.tupledK(fc)).mapK{ [Z] => (ft: (A[Z], (B[Z], C[Z]))) =>
    ((ft._1, ft._2._1), ft._2._2)
  }
```

- Unwrapping the second right tupledK:

```
((fa, fb), fc) =
  fa.tupledK((fb, fc)).mapK{ [Z] => (ft: (A[Z], (B[Z], C[Z]))) =>
    ((ft._1, ft._2._1), ft._2._2)
  }
```

- Unwrapping the first right tupledK:

```
((fa, fb), fc) =
  (fa, (fb, fc)).mapK{ [Z] => (ft: (A[Z], (B[Z], C[Z]))) =>
    ((ft._1, ft._2._1), ft._2._2)
  }
```

- Unwrapping the mapK:

```
((fa, fb), fc) = ((fa, fb), fc)
```

A.1.2 Induction step

- Start:

```
fa.tupledK(fb).tupledK(fc) =
  fa.tupledK(fb.tupledK(fc)).mapK{ [Z] => (ft: (A[Z], (B[Z], C[Z]))) =>
    ((ft._1, ft._2._1), ft._2._2)
  }
```

- Expanding tuples:

```
(x1a, x2a).tupledK((x1b, x2b)).tupledK((x1c, x2c)) =
  (x1a, x2a).tupledK((x1b, x2b).tupledK((x1c, x2c))).mapK {
    [Z] => (ft: (A[Z], (B[Z], C[Z]))) => ((ft._1, ft._2._1), ft._2._2)
  }
```

- Unwrapping the first left tupledK:

```
(x1a.tupledK(x1b), (x2a, x2b)).tupledK((x1c, x2c)) =
  (x1a, x2a).tupledK((x1b, x2b).tupledK((x1c, x2c))).mapK {
    [Z] => (ft: (A[Z], (B[Z], C[Z]))) => ((ft._1, ft._2._1), ft._2._2)
  }
```

- Unwrapping the second left tupledK:

```
(x1a.tupledK(x1b).tupledK(x1c), ((x2a, x2b), x2c)) =
  (x1a, x2a).tupledK((x1b, x2b).tupledK((x1c, x2c))).mapK {
    [Z] => (ft: (A[Z], (B[Z], C[Z]))) => ((ft._1, ft._2._1), ft._2._2)
  }
```

- Unwrapping the second right tupledK:

```
(x1a.tupledK(x1b).tupledK(x1c), ((x2a, x2b), x2c)) =
  (x1a, x2a).tupledK((x1b.tupledK(x1c), (x2b, x2c))).mapK {
    [Z] => (ft: (A[Z], (B[Z], C[Z]))) => ((ft._1, ft._2._1), ft._2._2)
  }
```

- Unwrapping the first right tupledK:

```
(x1a.tupledK(x1b).tupledK(x1c), ((x2a, x2b), x2c)) =
  (x1a.tupledRight(x1b.tupledK(x1c)), (x2a, (x2b, x2c))).mapK {
    [Z] => (ft: (A[Z], (B[Z], C[Z]))) => ((ft._1, ft._2._1), ft._2._2)
  }
```

- Unwrapping mapK:

```
(x1a.tupledK(x1b).tupledK(x1c), ((x2a, x2b), x2c)) =
  (
    x1a.tupledRight(x1b.tupledK(x1c)).mapK {
      [Z] => (ft: (A[Z], (B[Z], C[Z]))) => ((ft._1, ft._2._1), ft._2._2)
    },
    (x2a, (x2b, x2c)).mapK {
      [Z] => (ft: (A[Z], (B[Z], C[Z]))) => ((ft._1, ft._2._1), ft._2._2)
    }
  )
```

- Induction hypothesis:

```
(x1a.tupledK(x1b).tupledK(x1c), ((x2a, x2b), x2c)) =
  (
    x1a.tupledK(x1b).tupledK(x1c),
    (x2a, (x2b, x2c)).mapK {
      [Z] => (ft: (A[Z], (B[Z], C[Z]))) => ((ft._1, ft._2._1), ft._2._2)
    }
  )
```

- Unwrapping mapK:

```
(x1a.tupledK(x1b).tupledK(x1c), ((x2a, x2b), x2c)) =
  (x1a.tupledK(x1b).tupledK(x1c), ((x2a, x2b), x2c))
```

A.2 Applicative

A.2.1 Base step

Left identity

- Start:

```
ValueK.const(()).pure.tupledK(fa).mapK([Z] => (ft: (Unit, A[Z])) => ft._2) = fa
```

- Unwrapping pure:

```
().tupledK(fa).mapK([Z] => (ft: (Unit, A[Z])) => ft._2) = fa
```

- Unwrapping tupledK:

```
((), fa).mapK([Z] => (ft: (Unit, A[Z])) => ft._2) = fa
```

- Unwrapping mapK:

```
fa = fa
```

Right identity

- Start:

```
fa.tupledK(ValueK.const(()).pure).mapK([Z] => (ft: (A[Z], Unit)) => ft._1) = fa
```

- Unwrapping pure:

```
fa.tupledK(()).mapK([Z] => (ft: (A[Z], Unit)) => ft._1) = fa
```

- Unwrapping tupledK:

```
(fa, ()).mapK([Z] => (ft: (Unit, A[Z])) => ft._1) = fa
```

- Unwrapping mapK:

```
fa = fa
```

A.2.2 Induction step

Left identity

- Start:

```
ValueK.const(()).pure.tupledK(fa).mapK {
  [Z] => (ft: (Unit, A[Z])) => ft._2
} = fa
```

- Showing tuple:

```
ValueK.const(()).pure.tupledK((x1, x2)).mapK {
  [Z] => (ft: (Unit, A[Z])) => ft._2
} = (x1, x2)
```

- Unwrapping pure:

```
(ValueK.const(()).pure, ()).tupledK((x1, x2)).mapK {
  [Z] => (ft: (Unit, A[Z])) => ft._2
} = (x1, x2)
```

- Unwrapping tupledK:

```
(ValueK.const(()).pure.tupledK(x1), ((), x2)).mapK {
  [Z] => (ft: (Unit, A[Z])) => ft._2
} = (x1, x2)
```

- Unwrapping mapK:

```
(ValueK.const(()).pure.tupledK(x1).mapK {
  [Z] => (ft: (Unit, A[Z])) => ft._2
}, x2) = (x1, x2)
```

- Induction hypothesis:

```
(x1, x2) = (x1, x2)
```

Right identity

- Start:

```
fa.tupledK(ValueK.const(()).pure).mapK {
  [Z] => (ft: (A[Z], Unit)) => ft._1
} = fa
```

- Showing tuple:

```
(x1, x2).tupledK(ValueK.const(()).pure).mapK {
  [Z] => (ft: (A[Z], Unit)) => ft._1
} = (x1, x2)
```

- Unwrapping pure:

```
(x1, x2).tupledK((ValueK.const(()).pure, ())).mapK {
  [Z] => (ft: (A[Z], Unit)) => ft._1
} = (x1, x2)
```

- Unwrapping tupledK:

```
(x1.tupledK(ValueK.const(()).pure), (x2, ())).mapK {
  [Z] => (ft: (A[Z], Unit)) => ft._1
} = (x1, x2)
```

- Unwrapping mapK:

```
(x1.tupledK(ValueK.const(()).pure).mapK {
  [Z] => (ft: (A[Z], Unit)) => ft._1
}, x2) = (x1, x2)
```

- Induction hypothesis:

```
(x1, x2) = (x1, x2)
```

A.3 Traverse

The extra definitions and such will only be shown once.

A.3.1 Base step

Sequential composition

Extra definitions:

```
val fa: F[A, D] = ???
val f: A ~>: Compose2[M, B] = ???
val g: B ~>: Compose2[N, C] = ???

val N = summon[Applicative[N]]
val M = summon[Applicative[M]]
```

- Start:

  ```
  Nested(M.map(fa.traverseK(f))(fb => fb.traverseK(g))) ==
    fa.traverseK[[Z] => Nested[M, N, Z], C](
      [Z] => (a: A[Z]) => Nested(M.map(f(a))(b => g(b)))
    )
  ```

- Unwrapping traverseK (and renaming b to fb):

  ```
  Nested(M.map(f(fa))(fb => g(fb))) == Nested(M.map(f(fa))(fb => g(fb)))
  ```

Parallel composition

Extra definitions:

```
val fa: F[A, D] = ???
val f: A ~>: Compose2[M, B] = ???
val g: B ~>: Compose2[N, B] = ???

val N = summon[Applicative[N]]
val M = summon[Applicative[M]]

type MN[Z] = (M[Z], N[Z])

given Applicative[MN] with:
  def pure[X](x: X): MN[X] = (M.pure(x), N.pure(x))

  def pure[X, Y](f: MN[X => Y])(fa: MN[X]): MN[Y] =
    val (fam, fan) = fa
    val (fm, fn) = f
    (M.ap(fm)(fam), N.ap(fn)(fan))

  override def map[X, Y](fx: MN[X])(f: X => Y): MN[Y] =
    val (mx, nx) = fx
    (M.map(mx)(f), M.map(nx)(f))

  override def product[X, Y](fx: MN[X], fy: MN[Y]): MN[(X, Y)] =
    val (mx, nx) = fx
    val (my, ny) = fy
    (M.product(mx, my), N.product(nx, ny))
end given
```

- Start:

  ```
  fa.traverseK[MN, B]([Z] => (a: A[Z]) => (f(a), g(a))) ==
    (fa.traverseK(f), fa.traverseK(g))
  ```

- Unwrapping traverseK

  ```
  (f(fa), g(fa)) == (f(fa), g(fa))
  ```

A.3.2 Induction step

Sequential composition

- Start:

```
Nested(M.map(fa.traverseK(f))(fb => fb.traverseK(g))) ==
  fa.traverseK[[Z] => Nested[M, N, Z], C](
    [Z] => (a: A[Z]) => Nested(M.map(f(a))(b => g(b)))
  )
```

- Unwrapping traverseK:

```
Nested(
  M.map(fa._1.traverseK(f).product(f(fa._2)))(
    fb => fb._1.traverseK(g).product(g(fb._2))
  )
) ==
  fa._1.traverseK[[Z] => Nested[M, N, Z], C](
    [Z] => (a: A[Z]) => Nested(M.map(f(a))(b => g(b)))
  ).product(Nested(M.map(f(fa._2))(b => g(b)))
```

- Induction hypothesis

```
Nested(
  M.map(fa._1.traverseK(f).product(f(fa._2)))(
    fb => fb._1.traverseK(g).product(g(fb._2))
  )
) ==
  Nested(M.map(fa._1.traverseK(f))(fb => fb.traverseK(g)))
    .product(Nested(M.map(f(fa._2))(b => g(b)))
```

- Unwrapping Nested.product

```
//Nested.product can be unwrapped like so
Nested(mna).product(Nested(mnb)) ==
  Nested(M.map(M.product(mna, mnb))((na, nb) => N.product(na, nb)))

Nested(
  M.map(fa._1.traverseK(f).product(f(fa._2)))(
    fb => fb._1.traverseK(g).product(g(fb._2))
  )
) ==
  Nested(
    M.map(
      M.product(
        M.map(fa._1.traverseK(f))(fb => fb.traverseK(g)),
        M.map(f(fa._2))(b => g(b)
      )
    )((n1, n2) => N.product(n1, n2))
  )
```

- Removing nested and rewriting product to always use instances

```
M.map(
  M.product(
    fa._1.traverseK(f),
    f(fa._2)
  )
)(fb => N.product(fb._1.traverseK(g), g(fb._2))) ==
  M.map(
    M.product(
      M.map(fa._1.traverseK(f))(fb => fb.traverseK(g)),
      M.map(f(fa._2))(b => g(b)
    )
  )((n1, n2) => N.product(n1, n2))
```

- Functor law of composition together with applicative law of associativity.

```
M.map(
  M.product(
    M.map(fa._1.traverseK(f))(fb => fb.traverseK(g)),
    M.map(f(fa._2))(b => g(b))
  )
)((n1, n2) => N.product(n1, n2)) ==
  M.map(
    M.product(
      M.map(fa._1.traverseK(f))(fb => fb.traverseK(g)),
      M.map(f(fa._2))(b => g(b))
    )
  )((n1, n2) => N.product(n1, n2))
```

Parallel composition

- Start:

```
fa.traverseK[MN, B]([Z] => (a: A[Z]) => (f(a), g(a))) ==
  (fa.traverseK(f), fa.traverseK(g))
```

- Unwrapping traverseK:

```
fa._1.traverseK[MN, B](
  [Z] => (a: A[Z]) => (f(a), g(a))
).product((f(fa._2), g(fa._2))) ==
  (
    fa._1.traverseK(f).product(f(fa._2)),
    fa._2.traverseK(g).product(g(fa._2))
  )
```

- Unwrapping NM.product

```
(
  fa._1.traverseK[MN, B](
    [Z] => (a: A[Z]) => (f(a), g(a))
  )._1.product(f(fa._2)),
  fa._1.traverseK[MN, B](
    [Z] => (a: A[Z]) => (f(a), g(a))
  )._2.product(g(fa._2))
) ==
  (
    fa._1.traverseK(f).product(f(fa._2)),
    fa._2.traverseK(g).product(g(fa._2))
  )
```

- Induction hypothesis:

```
(
  (fa._1.traverseK(f), fa._1.traverseK(g))._1.product(f(fa._2)),
  (fa._2.traverseK(f), fa._2.traverseK(g))._2.product(g(fa._2))
) ==
  (
    fa._1.traverseK(f).product(f(fa._2)),
    fa._2.traverseK(g).product(g(fa._2))
  )
```

- Unwrapping tuple access:

```
(
  fa._1.traverseK(f).product(f(fa._2)),
  fa._2.traverseK(g)._2.product(g(fa._2))
) ==
  (
    fa._1.traverseK(f).product(f(fa._2)),
    fa._2.traverseK(g).product(g(fa._2))
  )
```

A.4 Monad

A.4.1 Base step

Left identity

- Start:

```
ValueK.const(a).pure.flatMapK(f) = f(a)
```

- Unwrapping pure:

  ```
  a.flatMapK(f) = f(a)
  ```

- Unwrapping flatMapK:

  ```
  f(a) = f(a)
  ```

Right identity

- Start:

  ```
  fa.flatMapK([Z] => (a: A[Z]) => ValueK.const(a).pure) = fa
  ```

- Unwrapping pure:

  ```
  fa.flatMapK([Z] => (a: A[Z]) => a) = fa
  ```

- Unwrapping flatMapK:

  ```
  fa = fa
  ```

Associativity

- Start:

  ```
  fa.flatMapK(f).flatMapK(g) =
    fa.flatMapK([Z] => (a: A[Z]) => f(a).flatMapK(g))
  ```

- Unwrapping first left flatMapK:

  ```
  f(fa).flatMapK(g) =
    fa.flatMapK([Z] => (a: A[Z]) => f(a).flatMapK(g))
  ```

- Unwrapping first right flatMapK:

  ```
  f(fa).flatMapK(g) = f(fa).flatMapK(g)
  ```

A.4.2 Induction step

Left identity

- Start:

```
ValueK.const(a).pure.flatMapK(f) = f(a)
```

- Unwrapping pure:

```
(ValueK.const(a).pure, a).flatMapK(f) = f(a)
```

- Unwrapping flatMapK:

```
(
  ValueK.const(a).pure.flatMapK([Z] => (a: A[Z]) => f(a)._1),
  f(a)._2
) = f(a)
```

- Induction hypothesis with the function inside flatMapK:

```
(([Z] => (a: A[Z]) => f(a)._1)(a), f(a)._2) = f(a)
```

- Applying function:

```
(f(a)._1, f(a)._2) = f(a)
```

- The function f returns a tuple, so constructing a tuple from the return values of the function is a no-op

```
f(a) = f(a)
```

Right identity

- Start:

```
fa.flatMapK([Z] => (a: A[Z]) => ValueK.const(a).pure) = fa
```

- Expanding tuple:

```
(x1, x2).flatMapK([Z] => (a: A[Z]) => ValueK.const(a).pure) = (x1, x2)
```

- Expanding pure:

```
(x1, x2).flatMapK([Z] => (a: A[Z]) => (ValueK.const(a).pure, a)) = (x1, x2)
```

- Expanding flatMapK:

```
(
  x1.flatMapK {
    [Z] => (a: A[Z]) => (ValueK.const(a).pure, a)._1
  },
  (ValueK.const(x2).pure, x2)._2
) = (x1, x2)
```

- Removing tuples where possible:

```
(x1.flatMapK([Z] => (a: A[Z]) => ValueK.const(a).pure), x2) = (x1, x2)
```

- Induction hypothesis:

```
(x1, x2) = (x1, x2)
```

Associativity

- Start:

```
fa.flatMapK(f).flatMapK(g) =
  fa.flatMapK([Z] => (a: A[Z]) => f(a).flatMapK(g))
```

- Expanding tuples:

```
(x1, x2).flatMapK(f).flatMapK(g) =
  (x1, x2).flatMapK([Z] => (a: A[Z]) => f(a).flatMapK(g))
```

- Expanding first left flatMapK:

```
(x1.flatMapK([Z] => (a: A[Z]) => f(a)._1), f(x2)._2).flatMapK(g) =
  (x1, x2).flatMapK([Z] => (a: A[Z]) => f(a).flatMapK(g))
```

- Expanding second left flatMapK:

```
(
  x1.flatMapK([Z] => (a: A[Z]) => f(a)._1).flatMapK([Z] => (b: B[Z]) => g(b)._1),
  g(f(x2)._2)._2
) = (x1, x2).flatMapK([Z] => (a: A[Z]) => f(a).flatMapK(g))
```

- Expanding first right flatMapK:

```
(
  x1.flatMapK([Z] => (a: A[Z]) => f(a)._1).flatMapK([Z] => (b: B[Z]) => g(b)._1),
  g(f(x2)._2)._2
) = (
  x1.flatMapK([Z] => (a: A[Z]) => f(a).flatMapK(g)._1),
  f(x2).flatMapK(g)._2
)
```

- Expanding right f application into tuples:

```
(
  x1.flatMapK([Z] => (a: A[Z]) => f(a)._1).flatMapK([Z] => (b: B[Z]) => g(b)._1),
  g(f(x2)._2)._2
) = (
  x1.flatMapK([Z] => (a: A[Z]) => (f(a)._1, f(a)._2).flatMapK(g)._1),
  (f(x2)._1, f(x2)._2).flatMapK(g)._2
)
```

- Expanding second right flatMapK:

```
(
  x1.flatMapK([Z] => (a: A[Z]) => f(a)._1).flatMapK([Z] => (b: B[Z]) => g(b)._1),
  g(f(x2)._2)._2
) = (
  x1.flatMapK {
    [Z] => (a: A[Z]) => (f(a)._1.flatMapK([Z] => (b: B[Z]) => g(b)._1), g(f(a)._2)._2)._1
  },
  (f(x2)._1.flatMapK([Z] => (b: B[Z]) => g(b)._1), g(f(x2)._2)._2)._2
)
```

- Removing tuples where possible:

```
(
  x1.flatMapK([Z] => (a: A[Z]) => f(a)._1).flatMapK([Z] => (b: B[Z]) => g(b)._1),
  g(f(x2)._2)._2
) = (
  x1.flatMapK([Z] => (a: A[Z]) => f(a)._1.flatMapK([Z] => (b: B[Z]) => g(b)._1)),
  g(f(x2)._2)._2
)

\item Induction hypothesis:
\begin{minted}[linenos=false,frame=none]{scala}
(
  x1.flatMapK([Z] => (a: A[Z]) => f(a)._1).flatMapK([Z] => (b: B[Z]) => g(b)._1),
  g(f(x2)._2)._2
) = (
  x1.flatMapK([Z] => (a: A[Z]) => f(a)._1.flatMapK([Z] => (b: B[Z]) => g(b)._1)),
  g(f(x2)._2)._2
)
```

A.5 Representable

A.5.1 Base step

Tabulate index

- Start:

```
tabulateK([Z] => (i: RepK[Z]) => fa.indexK(i)) = fa
```

- Expanding tabulateK:

```
fa.indexK(()) = fa
```

- Expanding indexK:

```
fa = fa
```

Index tabulate

- Start:

```
tabulateK(f).indexK(i) = f(i)
```

- Expanding tabulateK:

```
f(()).indexK(i) = f(i)
```

- There exists only one index value, Unit. As such i must also be Unit:

```
f(()).indexK(()) = f(())
```

- Expanding indexK:

```
f(()) = f(())
```

A.5.2 Induction step

Tabulate index

- Start:

```
tabulateK([Z] => (i: RepK[Z]) => fa.indexK(i)) = fa
```

- Expanding tuple:

```
tabulateK([Z] => (i: RepK[Z]) => (x1, x2).indexK(i)) = (x1, x2)
```

- Expanding tabulateK:

```
(
  tabulateK([Z] => (r: R1[Z]) => (x1, x2).indexK(Left(r))),
  (x1, x2).indexK(Right(()))
) = (x1, x2)
```

- Expanding indexK:

```
(tabulateK([Z] => (r: R1[Z]) => x1.indexK(r)), x2) = (x1, x2)
```

- Induction hypothesis:

```
(x1, x2) = (x1, x2)
```

Index tabulate

- Start:

```
tabulateK(f).indexK(i) = f(i)
```

- Expanding tabulateK:

```
(tabulateK([Z] => (r: R1[Z]) => f(Left(r))), f(Right(()))).indexK(i) = f(i)
```

- From here on, i can take two values. If it is Right(()) we end up here

```
(tabulateK([Z] => (r: R1[Z]) => f(Left(r))), f(Right(()))).indexK(Right(()))
  = f(Right(()))
```

  Expanding indexK

```
f(Right(())) = f(Right(()))
```

- If i is Left(r) we end up here

```
(tabulateK([Z] => (r: R1[Z]) => f(Left(r))), f(Right(()))).indexK(Left(r))
  = f(Left(r))
```

  Expanding indexK

```
tabulateK([Z] => (r: R1[Z]) => f(Left(r))).indexK(r) = f(Left(r))
```

  Abstracting

```
val g = [Z] => (r: R1[Z]) => f(Left(r))
tabulateK(g).indexK(r) = g(r)
```

  Induction hypothesis

```
val g = [Z] => (r: R1[Z]) => f(Left(r))
g(r) = g(r)
```

Appendix B

Decompiledd class files

All the class files here have been decompiled with CFR 0.152 and were for a case class of five fields. Forwarders have been removed. The code has been formated with IntelliJ and line breaks have been added to fit on the pages.

B.1 Encoders

B.1.1 CirceDerivation

```
1  public final class CirceDerivationDefs$BenchmarkCaseClass5$.anon.3
2      implements Encoder.AsObject<CirceDerivationDefs.BenchmarkCaseClass5> {
3      private final Encoder<Object> encoder0;
4      private final Encoder<String> encoder1;
5      private final Encoder<Object> encoder2;
6      private final Encoder<Object> encoder3;
7      private final Encoder<Json> encoder4;
8
9      public final JsonObject encodeObject(
10         CirceDerivationDefs.BenchmarkCaseClass5 a) {
11         return JsonObject$.MODULE$.fromIterable((Iterable) new.colon.colon(
12             (Object) new Tuple2((Object) "f0", (Object) this.encoder0.apply(
13                 (Object) BoxesRunTime.boxToInteger((int) a.f0()))),
14             (List) new.colon.colon((Object) new Tuple2((Object) "f1",
15                     (Object) this.encoder1.apply((Object) a.f1())),
16                 (List) new.colon.colon((Object) new Tuple2((Object) "f2",
17                         (Object) this.encoder2.apply(
18                             (Object) BoxesRunTime.boxToDouble((double) a.f2()))),
```

```
                    (List) new.colon.colon((Object) new Tuple2((Object) "f3",
                        (Object) this.encoder3.apply(
                            (Object) BoxesRunTime.boxToBoolean(
                                (boolean) a.f3()))), (List) new.colon.colon(
                        (Object) new Tuple2((Object) "f4",
                            (Object) this.encoder4.apply((Object) a.f4())),
                        (List) Nil$.MODULE$))))));
    }

    public CirceDerivationDefs$BenchmarkCaseClass5$.anon.3() {
        Encoder.$init$((Encoder) this);
        Encoder.AsObject.$init$((Encoder.AsObject) this);
        this.encoder0 = Encoder$.MODULE$.encodeInt();
        this.encoder1 = Encoder$.MODULE$.encodeString();
        this.encoder2 = Encoder$.MODULE$.encodeDouble();
        this.encoder3 = Encoder$.MODULE$.encodeBoolean();
        this.encoder4 = Encoder$.MODULE$.encodeJson();
    }
}
```

B.1.2 CirceGeneric (Scala 2)

```
public final class CirceGenericDefs$BenchmarkCaseClass5$anon$lazy$macro$23$1$.anon.3
    extends
    ReprAsObjectEncoder<
        .colon.colon<Object,
            .colon.colon<String,
                .colon.colon<Object,
                    .colon.colon<Object, .colon.colon<Json, HNil>>>>>> {
    private final Encoder<Object> circeGenericEncoderForf0 =
        Encoder$.MODULE$.encodeInt();
    private final Encoder<String> circeGenericEncoderForf1 =
        Encoder$.MODULE$.encodeString();
    private final Encoder<Object> circeGenericEncoderForf2 =
        Encoder$.MODULE$.encodeDouble();
    private final Encoder<Object> circeGenericEncoderForf3 =
        Encoder$.MODULE$.encodeBoolean();
    private final Encoder<Json> circeGenericEncoderForf4 =
        Encoder$.MODULE$.encodeJson();

    public final JsonObject encodeObject(
        .colon.colon<Object,
```

```
        .colon.colon<String,
            .colon.colon<Object,
                .colon.colon<Object, .colon.colon<Json, HNil>>>>> a) {
    .colon.colon<Object,
        .colon.colon<String,
            .colon.colon<Object,
                .colon.colon<Object, .colon.colon<Json, HNil>>>>>
        colon2 = a;
    if (colon2 == null) throw new MatchError(colon2);
    int circeGenericHListBindingForf0 =
        BoxesRunTime.unboxToInt((Object) colon2.head());
    .colon.colon colon3 = (.colon.colon) colon2.tail();
    if (colon3 == null) throw new MatchError(colon2);
    String circeGenericHListBindingForf1 = (String) colon3.head();
    .colon.colon colon4 = (.colon.colon) colon3.tail();
    if (colon4 == null) throw new MatchError(colon2);
    double circeGenericHListBindingForf2 =
        BoxesRunTime.unboxToDouble((Object) colon4.head());
    .colon.colon colon5 = (.colon.colon) colon4.tail();
    if (colon5 == null) throw new MatchError(colon2);
    boolean circeGenericHListBindingForf3 =
        BoxesRunTime.unboxToBoolean((Object) colon5.head());
    .colon.colon colon6 = (.colon.colon) colon5.tail();
    if (colon6 == null) throw new MatchError(colon2);
    Json circeGenericHListBindingForf4 = (Json) colon6.head();
    HNil hNil = (HNil) colon6.tail();
    if (!HNil$.MODULE$.equals(hNil)) throw new MatchError(colon2);
    return JsonObject$.MODULE$.fromIterable(
        (Iterable) Vector$.MODULE$.apply(
            (Seq) ScalaRunTime$.MODULE$.wrapRefArray(
                (Object[]) new Tuple2[]{new Tuple2((Object) "f0",
                    (Object) this.circeGenericEncoderForf0.apply(
                        (Object) BoxesRunTime.boxToInteger(
                            (int) circeGenericHListBindingForf0))),
                    new Tuple2((Object) "f1",
                        (Object) this.circeGenericEncoderForf1.apply(
                            (Object) circeGenericHListBindingForf1)),
                    new Tuple2((Object) "f2",
                        (Object) this.circeGenericEncoderForf2.apply(
                            (Object) BoxesRunTime.boxToDouble(
                                (double) circeGenericHListBindingForf2))),
                    new Tuple2((Object) "f3",
                        (Object) this.circeGenericEncoderForf3.apply(
                            (Object) BoxesRunTime.boxToBoolean(
                                (boolean) circeGenericHListBindingForf3))),
```

```
                        new Tuple2((Object) "f4",
                            (Object) this.circeGenericEncoderForf4.apply(
                                (Object) circeGenericHListBindingForf4))})));
    }

    public CirceGenericDefs$BenchmarkCaseClass5$anon$lazy$macro$23$1$.anon.3(
        CirceGenericDefs.BenchmarkCaseClass5.anon.lazy.macro.23.1$outer) {
    }
}

public final class CirceGenericDefs.BenchmarkCaseClass5.anon.lazy.macro.23.1
    implements Serializable {
    private DerivedAsObjectEncoder<CirceGenericDefs.BenchmarkCaseClass5>
        inst$macro$1;
    private ReprAsObjectEncoder<
        .colon.colon<Object,
            .colon.colon<String,
                .colon.colon<Object,
                    .colon.colon<Object, .colon.colon<Json, HNil>>>>>>
        inst$macro$22;
    private volatile byte bitmap$0;

    private DerivedAsObjectEncoder<CirceGenericDefs.BenchmarkCaseClass5>
        inst$macro$1$lzycompute() {
        CirceGenericDefs.BenchmarkCaseClass5.anon.lazy.macro231 var1_1 = this;
        synchronized (var1_1) {
            if ((byte) (this.bitmap$0 & 1) != 0) return this.inst$macro$1;
            this.inst$macro$1 = DerivedAsObjectEncoder$.MODULE$.deriveEncoder(
                LabelledGeneric$.MODULE$.materializeProduct(
                    DefaultSymbolicLabelling$.MODULE$.instance(
                        (HList) new .colon.colon(
                            (Object) SymbolLiteral.bootstrap("apply", "f0"),
                            (HList) new .colon.colon(
                                (Object) SymbolLiteral.bootstrap("apply", "f1"),
                                (HList) new .colon.colon(
                                    (Object) SymbolLiteral.bootstrap("apply",
                                        "f2"), (HList) new .colon.colon(
                                    (Object) SymbolLiteral.bootstrap("apply",
                                        "f3"), (HList) new .colon.colon(
                                    (Object) SymbolLiteral.bootstrap("apply",
                                        "f4"), (HList) HNil$.MODULE$))))))),
                    Generic$.MODULE$.instance(
                        (Function1 & Serializable) x0$3 -> {
                            CirceGenericDefs.BenchmarkCaseClass5
                                benchmarkCaseClass5 = x0$3;
```

```
                    if (benchmarkCaseClass5 == null)
                        throw new MatchError(
                            (Object) benchmarkCaseClass5);
                    int f0$macro$17 = benchmarkCaseClass5.f0();
                    String f1$macro$18 = benchmarkCaseClass5.f1();
                    double f2$macro$19 = benchmarkCaseClass5.f2();
                    boolean f3$macro$20 = benchmarkCaseClass5.f3();
                    Json f4$macro$21 = benchmarkCaseClass5.f4();
                    return new .colon.colon(
                        (Object) BoxesRunTime.boxToInteger(
                            (int) f0$macro$17),
                        (HList) new .colon.colon((Object) f1$macro$18,
                            (HList) new .colon.colon(
                                (Object) BoxesRunTime.boxToDouble(
                                    (double) f2$macro$19),
                                (HList) new .colon.colon(
                                    (Object) BoxesRunTime.boxToBoolean(
                                        (boolean) f3$macro$20),
                                    (HList) new .colon.colon(
                                        (Object) f4$macro$21,
                                        (HList) HNil$.MODULE$)))));
                }, (Function1 & Serializable) x0$4 -> {
                    .colon.colon colon2 = x0$4;
                    if (colon2 == null)
                        throw new MatchError((Object) colon2);
                    int f0$macro$12 =
                        BoxesRunTime.unboxToInt((Object) colon2.head());
                    .colon.colon colon3 =
                        (.colon.colon) colon2.tail();
                    if (colon3 == null)
                        throw new MatchError((Object) colon2);
                    String f1$macro$13 = (String) colon3.head();
                    .colon.colon colon4 =
                        (.colon.colon) colon3.tail();
                    if (colon4 == null)
                        throw new MatchError((Object) colon2);
                    double f2$macro$14 = BoxesRunTime.unboxToDouble(
                        (Object) colon4.head());
                    .colon.colon colon5 =
                        (.colon.colon) colon4.tail();
                    if (colon5 == null)
                        throw new MatchError((Object) colon2);
                    boolean f3$macro$15 = BoxesRunTime.unboxToBoolean(
                        (Object) colon5.head());
                    .colon.colon colon6 =
```

```
                                (.colon.colon) colon5.tail();
                            if (colon6 == null)
                                throw new MatchError((Object) colon2);
                            Json f4$macro$16 = (Json) colon6.head();
                            HNil hNil = (HNil) colon6.tail();
                            if (!HNil$.MODULE$.equals(hNil))
                                throw new MatchError((Object) colon2);
                            return new CirceGenericDefs.BenchmarkCaseClass5(
                                f0$macro$12, f1$macro$13, f2$macro$14,
                                f3$macro$15, f4$macro$16);
                        }), hlist.ZipWithKeys$.MODULE$.hconsZipWithKeys(
                        hlist.ZipWithKeys$.MODULE$.hconsZipWithKeys(
                            hlist.ZipWithKeys$.MODULE$.hconsZipWithKeys(
                                hlist.ZipWithKeys$.MODULE$.hconsZipWithKeys(
                                    hlist.ZipWithKeys$.MODULE$.hconsZipWithKeys(
                                        hlist.ZipWithKeys$.MODULE$.hnilZipWithKeys(),
                                        Witness$.MODULE$.mkWitness(
                                            (Object) SymbolLiteral.bootstrap(
                                                "apply", "f4"))),
                                    Witness$.MODULE$.mkWitness(
                                        (Object) SymbolLiteral.bootstrap(
                                            "apply", "f3"))),
                                Witness$.MODULE$.mkWitness(
                                    (Object) SymbolLiteral.bootstrap("apply",
                                        "f2"))), Witness$.MODULE$.mkWitness(
                                (Object) SymbolLiteral.bootstrap("apply",
                                    "f1"))), Witness$.MODULE$.mkWitness(
                            (Object) SymbolLiteral.bootstrap("apply", "f0"))),
                    (.less.colon.less)
                $less$colon$less$.MODULE$.refl()), Lazy$.MODULE$.apply(
                (Function0 & Serializable) () -> this.inst$macro$22()));
            this.bitmap$0 = (byte) (this.bitmap$0 | 1);
        }
        return this.inst$macro$1;
    }

    public DerivedAsObjectEncoder<
        CirceGenericDefs.BenchmarkCaseClass5> inst$macro$1() {
        if ((byte) (this.bitmap$0 & 1) != 0) return this.inst$macro$1;
        return this.inst$macro$1$lzycompute();
    }

    private ReprAsObjectEncoder<
        .colon.colon<Object,
            .colon.colon<String,
```

```
                .colon.colon<Object, .
                    colon.colon<Object, .colon.colon<Json, HNil>>>>>>
        inst$macro$22$lzycompute() {
        CirceGenericDefs.BenchmarkCaseClass5.anon.lazy.macro.23.1
        var1_1 = this;
        synchronized (var1_1) {
            if ((byte) (this.bitmap$0 & 2) != 0) return this.inst$macro$22;
            this.inst$macro$22 = new /* Unavailable Anonymous Inner Class!! */;
            this.bitmap$0 = (byte) (this.bitmap$0 | 2);
        }
        return this.inst$macro$22;
    }

    public ReprAsObjectEncoder<
        .colon.colon<Object,
            .colon.colon<String,
                .colon.colon<Object,
                    .colon.colon<Object, .colon.colon<Json, HNil>>>>>>
        inst$macro$22() {
        if ((byte) (this.bitmap$0 & 2) != 0) return this.inst$macro$22;
        return this.inst$macro$22$lzycompute();
    }
}
```

B.1.3 CirceGeneric (Scala 3)

```
public static final class CirceGenericDefs$BenchmarkCaseClass5$.anon.3
    implements DerivedEncoder<CirceGenericDefs.BenchmarkCaseClass5> {
    public static final long OFFSET$0 = LazyVals$.MODULE$.getOffsetStatic(
        CirceGenericDefs$BenchmarkCaseClass5$.anon.3.class.getDeclaredField(
            "0bitmap$3"));
    private final String[] elemLabels;
    public long 0bitmap$3;
    public Encoder[] elemEncoders$lzy2;

    public CirceGenericDefs$BenchmarkCaseClass5$.anon.3() {
        String string = "f0";
        String string2 = "f1";
        String string3 = "f2";
        String string4 = "f3";
        String string5 = "f4";
        this.elemLabels =
```

```
            (String[]) package$.MODULE$.Nil().$colon$colon((Object) string5)
                .$colon$colon((Object) string4).$colon$colon((Object) string3)
                .$colon$colon((Object) string2).$colon$colon((Object) string)
                .toArray(ClassTag$.MODULE$.apply(String.class));
    }

    public final String name() {
        return "BenchmarkCaseClass5";
    }

    public final String[] elemLabels() {
        return this.elemLabels;
    }

    public Encoder[] elemEncoders() {
        long l;
        long l2;
        while ((l2 = LazyVals$.MODULE$.STATE(
            l = LazyVals$.MODULE$.get((Object) this, OFFSET$0), 0)) != 3L) {
            if (l2 == 0L) {
                if (!LazyVals$.MODULE$.CAS((Object) this, OFFSET$0, l, 1, 0))
                    continue;
                try {
                    Encoder encodeA;
                    Encoder encodeA2;
                    Encoder encodeA3;
                    Encoder encodeA4;
                    Encoder encodeA5;
                    Encoder encoder = encodeA5 = Encoder$.MODULE$.encodeInt();
                    Encoder encoder2 =
                        encodeA4 = Encoder$.MODULE$.encodeString();
                    Encoder encoder3 =
                        encodeA3 = Encoder$.MODULE$.encodeDouble();
                    Encoder encoder4 =
                        encodeA2 = Encoder$.MODULE$.encodeBoolean();
                    Encoder encoder5 = encodeA = Encoder$.MODULE$.encodeJson();
                    Encoder[] encoderArray = (Encoder[]) package$.MODULE$.Nil()
                        .$colon$colon((Object) encoder5)
                        .$colon$colon((Object) encoder4)
                        .$colon$colon((Object) encoder3)
                        .$colon$colon((Object) encoder2)
                        .$colon$colon((Object) encoder)
                        .toArray(ClassTag$.MODULE$.apply(Encoder.class));
                    this.elemEncoders$lzy2 = encoderArray;
                    LazyVals$.MODULE$.setFlag((Object) this, OFFSET$0, 3, 0);
```

```
                    return encoderArray;
                } catch (Throwable throwable) {
                    LazyVals$.MODULE$.setFlag((Object) this, OFFSET$0, 0, 0);
                    throw throwable;
                }
            }
            LazyVals$.MODULE$.wait4Notification((Object) this, OFFSET$0, 1, 0);
        }
        return this.elemEncoders$lzy2;
    }

    public final JsonObject encodeObject(
        CirceGenericDefs.BenchmarkCaseClass5 a) {
        CirceGenericDefs$BenchmarkCaseClass5$
            circeGenericDefs$BenchmarkCaseClass5$ =
            CirceGenericDefs$BenchmarkCaseClass5$.MODULE$;
        return JsonObject$.MODULE$.fromIterable(
            this.encodedIterable((Product) a));
    }
}
```

B.1.4 PerspectiveInlining

```
public static final class PerspectiveInlineDefs$BenchmarkCaseClass5$.anon.3
    implements Encoder<PerspectiveInlineDefs.BenchmarkCaseClass5> {
    private final InlineHKDProductGeneric.DerivedImpl gen$proxy3$2;
    private final Encoder[] encoders;
    private final String[] names;

    public PerspectiveInlineDefs$BenchmarkCaseClass5$.anon.3(
        InlineHKDProductGeneric.DerivedImpl gen$proxy3$1) {
        this.gen$proxy3$2 = gen$proxy3$1;
        InlineHKDProductGeneric.DerivedImpl DerivedImpl_this = gen$proxy3$1;
        this.encoders = new Encoder[]{Encoder$.MODULE$.encodeInt(),
            Encoder$.MODULE$.encodeString(), Encoder$.MODULE$.encodeDouble(),
            Encoder$.MODULE$.encodeBoolean(), Encoder$.MODULE$.encodeJson()};
        InlineHKDProductGeneric.DerivedImpl DerivedImpl_this2 = gen$proxy3$1;
        this.names = new String[]{"f0", "f1", "f2", "f3", "f4"};
    }

    public Json apply(PerspectiveInlineDefs.BenchmarkCaseClass5 a) {
        InlineHKDProductGeneric.DerivedImpl InlineHKDGenericTypeclassOps_this =
```

```
            this.gen$proxy3$2;
        Nil$ res = package$.MODULE$.Nil();
        int i = 0;
        while (true) {
            if (i >= 5) {
                Nil$ list = res;
                return Json$.MODULE$.obj((Seq) list);
            }
            Nil$ nil$ = res;
            int n = i++;
            InlineHKDProductGeneric.DerivedImpl
                InlineHKDGenericTypeclassOps_this2 = this.gen$proxy3$2;
            Tuple2 tuple2 = Tuple2$.MODULE$.apply((Object) this.names[n],
                (Object) this.encoders[n].apply(a.productElement(n)));
            res = nil$.$colon$colon((Object) tuple2);
        }
    }
}
```

B.1.5 PerspectiveUnrolling

```
public static final class PerspectiveUnrollingDefs$BenchmarkCaseClass5$.anon.3
    implements Encoder<PerspectiveUnrollingDefs.BenchmarkCaseClass5> {
    private final InlineHKDProductGeneric.DerivedImpl gen$proxy3$2;
    private final Encoder[] encoders;
    private final String[] names;

    public PerspectiveUnrollingDefs$BenchmarkCaseClass5$.anon.3(
        InlineHKDProductGeneric.DerivedImpl gen$proxy3$1) {
        this.gen$proxy3$2 = gen$proxy3$1;
        InlineHKDProductGeneric.DerivedImpl DerivedImpl_this = gen$proxy3$1;
        this.encoders = new Encoder[]{Encoder$.MODULE$.encodeInt(),
            Encoder$.MODULE$.encodeString(), Encoder$.MODULE$.encodeDouble(),
            Encoder$.MODULE$.encodeBoolean(), Encoder$.MODULE$.encodeJson()};
        InlineHKDProductGeneric.DerivedImpl DerivedImpl_this2 = gen$proxy3$1;
        this.names = new String[]{"f0", "f1", "f2", "f3", "f4"};
    }

    public Json apply(PerspectiveUnrollingDefs.BenchmarkCaseClass5 a) {
        InlineHKDProductGeneric.DerivedImpl InlineHKDGenericTypeclassOps_this =
            this.gen$proxy3$2;
        Nil$ nil$ = package$.MODULE$.Nil();
```

```
        int n = 0;
        InlineHKDProductGeneric.DerivedImpl DerivedImpl_this =
            this.gen$proxy3$2;
        int n2 = a.f0();
        Json json = Json$.MODULE$.fromInt(n2);
        Tuple2 tuple2 =
            Tuple2$.MODULE$.apply((Object) this.names[n], (Object) json);
        List list = nil$.$colon$colon((Object) tuple2);
        int n3 = 1;
        InlineHKDProductGeneric.DerivedImpl DerivedImpl_this2 =
            this.gen$proxy3$2;
        String string = a.f1();
        Json json2 = Json$.MODULE$.fromString(string);
        Tuple2 tuple22 =
            Tuple2$.MODULE$.apply((Object) this.names[n3], (Object) json2);
        List list2 = list.$colon$colon((Object) tuple22);
        int n4 = 2;
        InlineHKDProductGeneric.DerivedImpl DerivedImpl_this3 =
            this.gen$proxy3$2;
        double d = a.f2();
        Json json3 = Json$.MODULE$.fromDoubleOrString(d);
        Tuple2 tuple23 =
            Tuple2$.MODULE$.apply((Object) this.names[n4], (Object) json3);
        List list3 = list2.$colon$colon((Object) tuple23);
        int n5 = 3;
        InlineHKDProductGeneric.DerivedImpl DerivedImpl_this4 =
            this.gen$proxy3$2;
        boolean bl = a.f3();
        Json json4 = Json$.MODULE$.fromBoolean(bl);
        Tuple2 tuple24 =
            Tuple2$.MODULE$.apply((Object) this.names[n5], (Object) json4);
        List list4 = list3.$colon$colon((Object) tuple24);
        int n6 = 4;
        InlineHKDProductGeneric.DerivedImpl DerivedImpl_this5 =
            this.gen$proxy3$2;
        Json json5 = a.f4();
        InlineHKDProductGeneric.DerivedImpl InlineHKDGenericTypeclassOps_this2 =
            this.gen$proxy3$2;
        Json json6 = this.encoders[n6].apply((Object) json5);
        Tuple2 tuple25 =
            Tuple2$.MODULE$.apply((Object) this.names[n6], (Object) json6);
        List list5 = list4.$colon$colon((Object) tuple25);
        return Json$.MODULE$.obj((Seq) list5);
    }
```

```
}
```

B.2 Decoders

B.2.1 CirceDerivation

```
public final class CirceDerivationDefs$BenchmarkCaseClass5$.anon.4
    implements Decoder<CirceDerivationDefs.BenchmarkCaseClass5> {
    private final Decoder<Object> decoder0;
    private final Decoder<String> decoder1;
    private final Decoder<Object> decoder2;
    private final Decoder<Object> decoder3;
    private final Decoder<Json> decoder4;

    public final Either<
            DecodingFailure, CirceDerivationDefs.BenchmarkCaseClass5> apply(
        HCursor c) {
        Either res = this.decoder0.tryDecode(c.downField("f0"));
        if (!res.isRight()) return res;
        int res1 = BoxesRunTime.unboxToInt((Object) ((Right) res).value());
        Either res2 = this.decoder1.tryDecode(c.downField("f1"));
        if (!res2.isRight()) return res2;
        String res22 = (String) ((Right) res2).value();
        Either res3 = this.decoder2.tryDecode(c.downField("f2"));
        if (!res3.isRight()) return res3;
        double res32 =
            BoxesRunTime.unboxToDouble((Object) ((Right) res3).value());
        Either res4 = this.decoder3.tryDecode(c.downField("f3"));
        if (!res4.isRight()) return res4;
        boolean res42 =
            BoxesRunTime.unboxToBoolean((Object) ((Right) res4).value());
        Either res5 = this.decoder4.tryDecode(c.downField("f4"));
        if (!res5.isRight()) return res5;
        Json res52 = (Json) ((Right) res5).value();
        return new Right(
            (Object) new CirceDerivationDefs.BenchmarkCaseClass5(res1, res22,
                res32, res42, res52));
    }

    private List<DecodingFailure> errors(
```

```
        Validated<NonEmptyList<DecodingFailure>, Object> result) {
        Validated<NonEmptyList<DecodingFailure>, Object> validated = result;
        if (validated instanceof Validated.Valid) {
            return Nil$.MODULE$;
        }
        if (!(validated instanceof Validated.Invalid))
            throw new MatchError(validated);
        Validated.Invalid invalid = (Validated.Invalid) validated;
        NonEmptyList e = (NonEmptyList) invalid.e();
        return e.toList();
    }

    public final Validated<
            NonEmptyList<DecodingFailure>, CirceDerivationDefs.BenchmarkCaseClass5>
            decodeAccumulating(
        HCursor c) {
        Validated res1 = this.decoder0.tryDecodeAccumulating(c.downField("f0"));
        Validated res2 = this.decoder1.tryDecodeAccumulating(c.downField("f1"));
        Validated res3 = this.decoder2.tryDecodeAccumulating(c.downField("f2"));
        Validated res4 = this.decoder3.tryDecodeAccumulating(c.downField("f3"));
        Validated res5 = this.decoder4.tryDecodeAccumulating(c.downField("f4"));
        List dfs =
            (List) new.colon.colon(this.errors(
                (Validated<NonEmptyList<DecodingFailure>, Object>) res1),
                (List) new.colon.colon(this.errors(
                    (Validated<NonEmptyList<DecodingFailure>, Object>) res2),
                    (List) new.colon.colon(this.errors(
                        (Validated<NonEmptyList<DecodingFailure>, Object>) res3),
                        (List) new.colon.colon(this.errors(
                            (Validated<NonEmptyList<DecodingFailure>, Object>) res4),
                            (List) new.colon.colon(this.errors(
                                (Validated<NonEmptyList<DecodingFailure>, Object>) res5),
                                (List) Nil$.MODULE$)))))
            .flatten(Predef$.MODULE$.$conforms());
        if (!dfs.isEmpty()) return Validated$.MODULE$.invalid(
            (Object) NonEmptyList$.MODULE$.fromListUnsafe(dfs));
        return Validated$.MODULE$.valid(
            (Object) new CirceDerivationDefs.BenchmarkCaseClass5(
                BoxesRunTime.unboxToInt((Object) ((Validated.Valid) res1).a()),
                (String) ((Validated.Valid) res2).a(),
                BoxesRunTime.unboxToDouble(
                    (Object) ((Validated.Valid) res3).a()),
                BoxesRunTime.unboxToBoolean(
                    (Object) ((Validated.Valid) res4).a()),
                (Json) ((Validated.Valid) res5).a()));
```

```
    }

    public CirceDerivationDefs$BenchmarkCaseClass5$.anon.4() {
        Decoder.$init$((Decoder) this);
        this.decoder0 = Decoder$.MODULE$.decodeInt();
        this.decoder1 = Decoder$.MODULE$.decodeString();
        this.decoder2 = Decoder$.MODULE$.decodeDouble();
        this.decoder3 = Decoder$.MODULE$.decodeBoolean();
        this.decoder4 = Decoder$.MODULE$.decodeJson();
    }
}
```

B.2.2 CirceGeneric (Scala 2)

```
public final class CirceGenericDefs$BenchmarkCaseClass5$anon$lazy$macro$47$1$.anon.4
    extends
    ReprDecoder<
        .colon.colon<Object,
            .colon.colon<String,
                .colon.colon<Object,
                    .colon.colon<Object, .colon.colon<Json, HNil>>>>>> {
    private final Decoder<Object> circeGenericDecoderForf0 =
        Decoder$.MODULE$.decodeInt();
    private final Decoder<String> circeGenericDecoderForf1 =
        Decoder$.MODULE$.decodeString();
    private final Decoder<Object> circeGenericDecoderForf2 =
        Decoder$.MODULE$.decodeDouble();
    private final Decoder<Object> circeGenericDecoderForf3 =
        Decoder$.MODULE$.decodeBoolean();
    private final Decoder<Json> circeGenericDecoderForf4 =
        Decoder$.MODULE$.decodeJson();

    public final Either<DecodingFailure,
        .colon.colon<Object,
            .colon.colon<String,
                .colon.colon<Object,
                    .colon.colon<Object, .colon.colon<Json, HNil>>>>>>
        apply(
        HCursor c) {
        return (Either) ReprDecoder$.MODULE$.consResults(
            (Object) this.circeGenericDecoderForf0.tryDecode(c.downField("f0")),
            ReprDecoder$.MODULE$.consResults(
```

```
                (Object) this.circeGenericDecoderForf1.tryDecode(
                    c.downField("f1")), ReprDecoder$.MODULE$.consResults(
                    (Object) this.circeGenericDecoderForf2.tryDecode(
                        c.downField("f2")), ReprDecoder$.MODULE$.consResults(
                        (Object) this.circeGenericDecoderForf3.tryDecode(
                            c.downField("f3")),
                        ReprDecoder$.MODULE$.consResults(
                            (Object) this.circeGenericDecoderForf4.tryDecode(
                                c.downField("f4")),
                            (Object) ReprDecoder$.MODULE$.hnilResult(),
                            (Apply) Decoder$.MODULE$.resultInstance()),
                        (Apply) Decoder$.MODULE$.resultInstance()),
                    (Apply) Decoder$.MODULE$.resultInstance()),
                (Apply) Decoder$.MODULE$.resultInstance()),
            (Apply) Decoder$.MODULE$.resultInstance());
    }

    public final Validated<NonEmptyList<DecodingFailure>,
        .colon.colon<Object,
            .colon.colon<String,
                .colon.colon<Object,
                    .colon.colon<Object, .colon.colon<Json, HNil>>>>>>
        decodeAccumulating(
        HCursor c) {
        return (Validated) ReprDecoder$.MODULE$.consResults(
            (Object) this.circeGenericDecoderForf0.tryDecodeAccumulating(
                c.downField("f0")), ReprDecoder$.MODULE$.consResults(
                (Object) this.circeGenericDecoderForf1.tryDecodeAccumulating(
                    c.downField("f1")), ReprDecoder$.MODULE$.consResults(
                    (Object) this.circeGenericDecoderForf2.tryDecodeAccumulating(
                        c.downField("f2")), ReprDecoder$.MODULE$.consResults(
                        (Object) this.circeGenericDecoderForf3.tryDecodeAccumulating(
                            c.downField("f3")),
                        ReprDecoder$.MODULE$.consResults(
                            (Object)this
                                .circeGenericDecoderForf4
                                .tryDecodeAccumulating(
                                    c.downField("f4")),
                            (Object) ReprDecoder$.MODULE$.hnilResultAccumulating(),
                            (Apply) Decoder$.MODULE$.accumulatingResultInstance()),
                        (Apply) Decoder$.MODULE$.accumulatingResultInstance()),
                    (Apply) Decoder$.MODULE$.accumulatingResultInstance()),
                (Apply) Decoder$.MODULE$.accumulatingResultInstance()),
            (Apply) Decoder$.MODULE$.accumulatingResultInstance());
    }
```

```
74
75      public CirceGenericDefs$BenchmarkCaseClass5$anon$lazy$macro$47$1$.anon.4(
76          CirceGenericDefsBenchmarkCaseClass5.anon.lazy.macro.47.1 $outer) {
77      }
78  }
79
80
81  public final class CirceGenericDefsl.BenchmarkCaseClass5.anon.lazy.macro.47.1
82      implements Serializable {
83      private DerivedDecoder<CirceGenericDefs.BenchmarkCaseClass5> inst$macro$25;
84      private ReprDecoder<
85          .colon.colon<Object,
86              .colon.colon<String,
87                  .colon.colon<Object,
88                      .colon.colon<Object, .colon.colon<Json, HNil>>>>>>
89          inst$macro$46;
90      private volatile byte bitmap$0;
91
92      private DerivedDecoder<CirceGenericDefs.BenchmarkCaseClass5>
93          inst$macro$25$lzycompute() {
94          CirceGenericDefs.BenchmarkCaseClass5.anon.lazy.macro.47.1 var1_1 = this;
95          synchronized (var1_1) {
96              if ((byte) (this.bitmap$0 & 1) != 0) return this.inst$macro$25;
97              this.inst$macro$25 = DerivedDecoder$.MODULE$.deriveDecoder(
98                  LabelledGeneric$.MODULE$.materializeProduct(
99                      DefaultSymbolicLabelling$.MODULE$.instance(
100                         (HList) new .colon.colon(
101                             (Object) SymbolLiteral.bootstrap("apply", "f0"),
102                             (HList) new .colon.colon(
103                                 (Object) SymbolLiteral.bootstrap("apply", "f1"),
104                                 (HList) new .colon.colon(
105                                     (Object) SymbolLiteral.bootstrap("apply",
106                                         "f2"), (HList) new .colon.colon(
107                                     (Object) SymbolLiteral.bootstrap("apply",
108                                         "f3"), (HList) new .colon.colon(
109                                     (Object) SymbolLiteral.bootstrap("apply",
110                                         "f4"), (HList) HNil$.MODULE$))))),
111                     Generic$.MODULE$.instance(
112                         (Function1 & Serializable) x0$7 -> {
113                             CirceGenericDefs.BenchmarkCaseClass5
114                                 benchmarkCaseClass5 = x0$7;
115                             if (benchmarkCaseClass5 == null)
116                                 throw new MatchError(
117                                     (Object) benchmarkCaseClass5);
118                             int f0$macro$41 = benchmarkCaseClass5.f0();
```

```
                String f1$macro$42 = benchmarkCaseClass5.f1();
                double f2$macro$43 = benchmarkCaseClass5.f2();
                boolean f3$macro$44 = benchmarkCaseClass5.f3();
                Json f4$macro$45 = benchmarkCaseClass5.f4();
                return new .colon.colon(
                    (Object) BoxesRunTime.boxToInteger(
                        (int) f0$macro$41),
                    (HList) new .colon.colon((Object) f1$macro$42,
                        (HList) new .colon.colon(
                            (Object) BoxesRunTime.boxToDouble(
                                (double) f2$macro$43),
                            (HList) new .colon.colon(
                                (Object) BoxesRunTime.boxToBoolean(
                                    (boolean) f3$macro$44),
                                (HList) new .colon.colon(
                                    (Object) f4$macro$45,
                                    (HList) HNil$.MODULE$)))));
            }, (Function1 & Serializable) x0$8 -> {
                .colon.colon colon2 = x0$8;
                if (colon2 == null)
                    throw new MatchError((Object) colon2);
                int f0$macro$36 =
                    BoxesRunTime.unboxToInt((Object) colon2.head());
                .colon.colon colon3 =
                    (.colon.colon) colon2.tail();
                if (colon3 == null)
                    throw new MatchError((Object) colon2);
                String f1$macro$37 = (String) colon3.head();
                .colon.colon colon4 =
                    (.colon.colon) colon3.tail();
                if (colon4 == null)
                    throw new MatchError((Object) colon2);
                double f2$macro$38 = BoxesRunTime.unboxToDouble(
                    (Object) colon4.head());
                .colon.colon colon5 =
                    (.colon.colon) colon4.tail();
                if (colon5 == null)
                    throw new MatchError((Object) colon2);
                boolean f3$macro$39 = BoxesRunTime.unboxToBoolean(
                    (Object) colon5.head());
                .colon.colon colon6 =
                    (.colon.colon) colon5.tail();
                if (colon6 == null)
                    throw new MatchError((Object) colon2);
                Json f4$macro$40 = (Json) colon6.head();
```

```
                            HNil hNil = (HNil) colon6.tail();
                            if (!HNil$.MODULE$.equals(hNil))
                                throw new MatchError((Object) colon2);
                            return new CirceGenericDefs.BenchmarkCaseClass5(
                                f0$macro$36, f1$macro$37, f2$macro$38,
                                f3$macro$39, f4$macro$40);
                        }), hlist.ZipWithKeys$.MODULE$.hconsZipWithKeys(
                        hlist.ZipWithKeys$.MODULE$.hconsZipWithKeys(
                            hlist.ZipWithKeys$.MODULE$.hconsZipWithKeys(
                                hlist.ZipWithKeys$.MODULE$.hconsZipWithKeys(
                                    hlist.ZipWithKeys$.MODULE$.hconsZipWithKeys(
                                        hlist.ZipWithKeys$.MODULE$.hnilZipWithKeys(),
                                        Witness$.MODULE$.mkWitness(
                                            (Object) SymbolLiteral.bootstrap(
                                                "apply", "f4"))),
                                    Witness$.MODULE$.mkWitness(
                                        (Object) SymbolLiteral.bootstrap(
                                            "apply", "f3"))),
                                Witness$.MODULE$.mkWitness(
                                    (Object) SymbolLiteral.bootstrap("apply",
                                        "f2"))), Witness$.MODULE$.mkWitness(
                                (Object) SymbolLiteral.bootstrap("apply",
                                    "f1"))), Witness$.MODULE$.mkWitness(
                            (Object) SymbolLiteral.bootstrap("apply", "f0"))),
                    (.less.colon.less)
                $less$colon$less$.MODULE$.refl()), Lazy$.MODULE$.apply(
                (Function0 & Serializable) () -> this.inst$macro$46()));
            this.bitmap$0 = (byte) (this.bitmap$0 | 1);
        }
        return this.inst$macro$25;
    }

    public DerivedDecoder<CirceGenericDefs.BenchmarkCaseClass5> inst$macro$25() {
        if ((byte) (this.bitmap$0 & 1) != 0) return this.inst$macro$25;
        return this.inst$macro$25$lzycompute();
    }

    private ReprDecoder<
        .colon.colon<Object,
            .colon.colon<String,
                .colon.colon<Object,
                    .colon.colon<Object, .colon.colon<Json, HNil>>>>>>
        inst$macro$46$lzycompute() {
        CirceGenericDefs.BenchmarkCaseClass5.anon.lazy.macro.47.1
        var1_1 = this;
```

```
        synchronized (var1_1) {
            if ((byte) (this.bitmap$0 & 2) != 0) return this.inst$macro$46;
            this.inst$macro$46 = new /* Unavailable Anonymous Inner Class!! */;
            this.bitmap$0 = (byte) (this.bitmap$0 | 2);
        }
        return this.inst$macro$46;
    }

    public ReprDecoder<
        .colon.colon<Object,
            .colon.colon<String,
                .colon.colon<Object,
                    .colon.colon<Object, .colon.colon<Json, HNil>>>>>>
        inst$macro$46() {
        if ((byte) (this.bitmap$0 & 2) != 0) return this.inst$macro$46;
        return this.inst$macro$46$lzycompute();
    }
}
```

B.2.3 CirceGeneric (Scala 3)

```
public static final class CirceGenericDefs$BenchmarkCaseClass5$.anon.4
    implements DerivedDecoder<CirceGenericDefs.BenchmarkCaseClass5> {
    public static final long OFFSET$0 = LazyVals$.MODULE$.getOffsetStatic(
        CirceGenericDefs$BenchmarkCaseClass5$.anon.4.class.getDeclaredField(
            "0bitmap$4"));
    private final String[] elemLabels;
    public long 0bitmap$4;
    public Decoder[] elemDecoders$lzy2;

    public CirceGenericDefs$BenchmarkCaseClass5$.anon.4() {
        String string = "f0";
        String string2 = "f1";
        String string3 = "f2";
        String string4 = "f3";
        String string5 = "f4";
        this.elemLabels =
            (String[]) package$.MODULE$.Nil().$colon$colon((Object) string5)
                .$colon$colon((Object) string4).$colon$colon((Object) string3)
                .$colon$colon((Object) string2).$colon$colon((Object) string)
                .toArray(ClassTag$.MODULE$.apply(String.class));
    }
```

```
22
23      public final String name() {
24          return "BenchmarkCaseClass5";
25      }
26
27      public final String[] elemLabels() {
28          return this.elemLabels;
29      }
30
31      public Decoder[] elemDecoders() {
32          long l;
33          long l2;
34          while ((l2 = LazyVals$.MODULE$.STATE(
35              l = LazyVals$.MODULE$.get((Object) this, OFFSET$0), 0)) != 3L) {
36              if (l2 == 0L) {
37                  if (!LazyVals$.MODULE$.CAS((Object) this, OFFSET$0, l, 1, 0))
38                      continue;
39                  try {
40                      Decoder decodeA;
41                      Decoder decodeA2;
42                      Decoder decodeA3;
43                      Decoder decodeA4;
44                      Decoder decodeA5;
45                      Decoder decoder = decodeA5 = Decoder$.MODULE$.decodeInt();
46                      Decoder decoder2 =
47                          decodeA4 = Decoder$.MODULE$.decodeString();
48                      Decoder decoder3 =
49                          decodeA3 = Decoder$.MODULE$.decodeDouble();
50                      Decoder decoder4 =
51                          decodeA2 = Decoder$.MODULE$.decodeBoolean();
52                      Decoder decoder5 = decodeA = Decoder$.MODULE$.decodeJson();
53                      Decoder[] decoderArray = (Decoder[]) package$.MODULE$.Nil()
54                          .$colon$colon((Object) decoder5)
55                          .$colon$colon((Object) decoder4)
56                          .$colon$colon((Object) decoder3)
57                          .$colon$colon((Object) decoder2)
58                          .$colon$colon((Object) decoder)
59                          .toArray(ClassTag$.MODULE$.apply(Decoder.class));
60                      this.elemDecoders$lzy2 = decoderArray;
61                      LazyVals$.MODULE$.setFlag((Object) this, OFFSET$0, 3, 0);
62                      return decoderArray;
63                  } catch (Throwable throwable) {
64                      LazyVals$.MODULE$.setFlag((Object) this, OFFSET$0, 0, 0);
65                      throw throwable;
66                  }
```

```
            }
            LazyVals$.MODULE$.wait4Notification((Object) this, OFFSET$0, l, 0);
        }
        return this.elemDecoders$lzy2;
    }

    public final Either apply(HCursor c) {
        CirceGenericDefs$BenchmarkCaseClass5$
            circeGenericDefs$BenchmarkCaseClass5$;
        CirceGenericDefs$BenchmarkCaseClass5$ m =
            circeGenericDefs$BenchmarkCaseClass5$ =
                CirceGenericDefs$BenchmarkCaseClass5$.MODULE$;
        if (!c.value().isObject()) return package$.MODULE$.Left().apply(
            (Object) DecodingFailure$.MODULE$.apply(this.name(),
                () ->
    CirceGenericDefs$.
        perspective$circederivation$CirceGenericDefs$BenchmarkCaseClass5$$
        anon$4$$_$apply$$anonfun$2(
            (HCursor) c)));
        Iterator iter = this.resultIterator(c);
        Object[] res = new Object[this.elemCount()];
        Left failed = null;
        int i = 0;
        while (iter.hasNext() && failed == null) {
            Either either = (Either) iter.next();
            if (either instanceof Right) {
                Object value;
                res[i] = value = ((Right) either).value();
            } else {
                Left l;
                if (!(either instanceof Left))
                    throw new MatchError((Object) either);
                failed = l = (Left) either;
            }
            ++i;
        }
        if (failed != null) return (Either) failed;
        return package$.MODULE$.Right()
            .apply(m.fromProduct(Tuple$.MODULE$.fromArray((Object) res)));
    }

    public final Validated decodeAccumulating(HCursor c) {
        CirceGenericDefs$BenchmarkCaseClass5$
            circeGenericDefs$BenchmarkCaseClass5$;
        CirceGenericDefs$BenchmarkCaseClass5$ m =
```

```
            circeGenericDefs$BenchmarkCaseClass5$ =
                CirceGenericDefs$BenchmarkCaseClass5$.MODULE$;
        if (!c.value().isObject()) return Validated$.MODULE$.invalidNel(
            (Object) DecodingFailure$.MODULE$.apply(this.name(),
                () ->
    CirceGenericDefs$.
        perspective$circederivation$CirceGenericDefs$BenchmarkCaseClass5$$
        anon$4$$_$decodeAccumulating$$anonfun$2(
                    (HCursor) c)));
        Iterator iter = this.resultAccumulatingIterator(c);
        Object[] res = new Object[this.elemCount()];
        Builder failed = package$.MODULE$.List().newBuilder();
        int i = 0;
        while (true) {
            BoxedUnit boxedUnit;
            if (!iter.hasNext()) {
                List failures = (List) failed.result();
                if (!failures.isEmpty()) return Validated$.MODULE$.invalid(
                    (Object) NonEmptyList$.MODULE$.fromListUnsafe(failures));
                return Validated$.MODULE$.valid(
                    m.fromProduct(Tuple$.MODULE$.fromArray((Object) res)));
            }
            Validated validated = (Validated) iter.next();
            if (validated instanceof Validated.Valid) {
                Object object;
                Object value;
                Validated.Valid valid = Validated.Valid$.MODULE$.unapply(
                    (Validated.Valid) validated);
                res[i] = value = (object = valid._1());
                boxedUnit = BoxedUnit.UNIT;
            } else {
                NonEmptyList nonEmptyList;
                if (!(validated instanceof Validated.Invalid))
                    throw new MatchError((Object) validated);
                Validated.Invalid invalid = Validated.Invalid$.MODULE$.unapply(
                    (Validated.Invalid) validated);
                NonEmptyList failures =
                    nonEmptyList = (NonEmptyList) invalid._1();
                boxedUnit =
                    failed.$plus$plus$eq((IterableOnce) failures.toList());
            }
            ++i;
        }
    }
```

```
}
```

B.2.4 PerspectiveInlining

```
public static final class PerspectiveInlineDefs$BenchmarkCaseClass5$.anon.4
    implements Decoder<PerspectiveInlineDefs.BenchmarkCaseClass5> {
    private final InlineHKDProductGeneric.DerivedImpl gen$proxy4$2;
    private final Decoder[] decoders;
    private final String[] names;

    public PerspectiveInlineDefs$BenchmarkCaseClass5$.anon.4(
        InlineHKDProductGeneric.DerivedImpl gen$proxy4$1) {
        this.gen$proxy4$2 = gen$proxy4$1;
        InlineHKDProductGeneric.DerivedImpl DerivedImpl_this = gen$proxy4$1;
        this.decoders = new Decoder[]{Decoder$.MODULE$.decodeInt(),
            Decoder$.MODULE$.decodeString(), Decoder$.MODULE$.decodeDouble(),
            Decoder$.MODULE$.decodeBoolean(), Decoder$.MODULE$.decodeJson()};
        InlineHKDProductGeneric.DerivedImpl DerivedImpl_this2 = gen$proxy4$1;
        this.names = new String[]{"f0", "f1", "f2", "f3", "f4"};
    }

    public Either apply(HCursor cursor) {
        InlineHKDProductGeneric.DerivedImpl InlineHKDGeneric_this;
        InlineHKDProductGeneric.DerivedImpl InlineHKDGenericTypeclassOps_this =
            InlineHKDGeneric_this = this.gen$proxy4$2;
        LazyRef lazyRef = new LazyRef();
        Object default_ = null;
        DecodingFailure error = default_;
        boolean gotError = false;
        Object[] arr = new Object[5];
        for (int i = 0; i < 5 && !gotError; ++i) {
            int n = i;
            InlineHKDProductGeneric.DerivedImpl
                InlineHKDGenericTypeclassOps_this2 = this.gen$proxy4$2;
            Either res =
                this.decoders[n].tryDecode(cursor.downField(this.names[n]));
            Either either = res;
            if (either instanceof Right) {
                Object v;
                arr[i] = v = ((Right) either).value();
                continue;
            }
```

```
            if (!(either instanceof Left))
                throw new MatchError((Object) either);
            DecodingFailure e = (DecodingFailure) ((Left) either).value();
            gotError = true;
            error = e;
        }
        Object ret = gotError ? package$.MODULE$.Left().apply((Object) error) :
            package$.MODULE$.Right().apply((Object) arr);
        Left left = ret;
        if (left instanceof Right) {
            Object value = ((Right) left).value();
            InlineHKDProductGeneric.DerivedImpl DerivedImpl_this =
                this.gen$proxy4$2;
            return package$.MODULE$.Right().apply(
                (Object) ((PerspectiveInlineDefs.BenchmarkCaseClass5)
                    DerivedImpl_this.m().fromProduct(
                        (Product) ArrayProduct$.MODULE$.apply((Object[]) IArray.package.
            IArray$.MODULE$.map(value,
    PerspectiveInlineDefs$::
        perspective$circederivation$PerspectiveInlineDefs$BenchmarkCaseClass5$$
        anon$4$$_$apply$$anonfun$2,
                ClassTag$.MODULE$.apply(Object.class))))));
        }
        if (!(left instanceof Left)) throw new MatchError((Object) left);
        DecodingFailure e = (DecodingFailure) left.value();
        return package$.MODULE$.Left().apply((Object) e);
    }
}
```

B.2.5 PerspectiveUnrolling

```
public static final class PerspectiveUnrollingDefs$BenchmarkCaseClass5$.anon.4
    implements Decoder<PerspectiveUnrollingDefs.BenchmarkCaseClass5> {
    private final InlineHKDProductGeneric.DerivedImpl gen$proxy4$2;
    private final Decoder[] decoders;
    private final String[] names;

    public PerspectiveUnrollingDefs$BenchmarkCaseClass5$.anon.4(
        InlineHKDProductGeneric.DerivedImpl gen$proxy4$1) {
        this.gen$proxy4$2 = gen$proxy4$1;
        InlineHKDProductGeneric.DerivedImpl DerivedImpl_this = gen$proxy4$1;
        this.decoders = new Decoder[]{Decoder$.MODULE$.decodeInt(),
```

```
            Decoder$.MODULE$.decodeString(), Decoder$.MODULE$.decodeDouble(),
            Decoder$.MODULE$.decodeBoolean(), Decoder$.MODULE$.decodeJson()};
        InlineHKDProductGeneric.DerivedImpl DerivedImpl_this2 = gen$proxy4$1;
        this.names = new String[]{"f0", "f1", "f2", "f3", "f4"};
    }

    public Either apply(HCursor cursor) {
        Left ret;
        Left left;
        InlineHKDProductGeneric.DerivedImpl InlineHKDGenericTypeclassOps_this =
            this.gen$proxy4$2;
        Object[] arr = new Object[5];
        int n = 0;
        InlineHKDProductGeneric.DerivedImpl InlineHKDGenericTypeclassOps_this2 =
            this.gen$proxy4$2;
        Either either =
            this.decoders[n].tryDecode(cursor.downField(this.names[n]));
        if (either instanceof Right) {
            Object v;
            arr[0] = v = ((Right) either).value();
            int n2 = 1;
            InlineHKDProductGeneric.DerivedImpl
                InlineHKDGenericTypeclassOps_this3 = this.gen$proxy4$2;
            Either either2 =
                this.decoders[n2].tryDecode(cursor.downField(this.names[n2]));
            if (either2 instanceof Right) {
                Object v2;
                arr[1] = v2 = ((Right) either2).value();
                int n3 = 2;
                InlineHKDProductGeneric.DerivedImpl
                    InlineHKDGenericTypeclassOps_this4 = this.gen$proxy4$2;
                Either either3 = this.decoders[n3].tryDecode(
                    cursor.downField(this.names[n3]));
                if (either3 instanceof Right) {
                    Object v3;
                    arr[2] = v3 = ((Right) either3).value();
                    int n4 = 3;
                    InlineHKDProductGeneric.DerivedImpl
                        InlineHKDGenericTypeclassOps_this5 = this.gen$proxy4$2;
                    Either either4 = this.decoders[n4].tryDecode(
                        cursor.downField(this.names[n4]));
                    if (either4 instanceof Right) {
                        Object v4;
                        arr[3] = v4 = ((Right) either4).value();
                        int n5 = 4;
```

```
                        InlineHKDProductGeneric.DerivedImpl
                            InlineHKDGenericTypeclassOps_this6 =
                            this.gen$proxy4$2;
                        Either either5 = this.decoders[n5].tryDecode(
                            cursor.downField(this.names[n5]));
                        if (either5 instanceof Right) {
                            Object v5;
                            arr[4] = v5 = ((Right) either5).value();
                            left = package$.MODULE$.Right().apply((Object) arr);
                        } else {
                            if (!(either5 instanceof Left))
                                throw new MatchError((Object) either5);
                            DecodingFailure e =
                                (DecodingFailure) ((Left) either5).value();
                            left = package$.MODULE$.Left().apply((Object) e);
                        }
                    } else {
                        if (!(either4 instanceof Left))
                            throw new MatchError((Object) either4);
                        DecodingFailure e =
                            (DecodingFailure) ((Left) either4).value();
                        left = package$.MODULE$.Left().apply((Object) e);
                    }
                } else {
                    if (!(either3 instanceof Left))
                        throw new MatchError((Object) either3);
                    DecodingFailure e =
                        (DecodingFailure) ((Left) either3).value();
                    left = package$.MODULE$.Left().apply((Object) e);
                }
            } else {
                if (!(either2 instanceof Left))
                    throw new MatchError((Object) either2);
                DecodingFailure e = (DecodingFailure) ((Left) either2).value();
                left = package$.MODULE$.Left().apply((Object) e);
            }
        } else {
            if (!(either instanceof Left))
                throw new MatchError((Object) either);
            DecodingFailure e = (DecodingFailure) ((Left) either).value();
            left = package$.MODULE$.Left().apply((Object) e);
        }
        Left left2 = ret = left;
        if (left2 instanceof Right) {
            Object value = ((Right) left2).value();
```

```
            InlineHKDProductGeneric.DerivedImpl DerivedImpl_this =
                this.gen$proxy4$2;
            return package$.MODULE$.Right().apply(
                (Object) ((PerspectiveUnrollingDefs.BenchmarkCaseClass5)
                    DerivedImpl_this.m().fromProduct(
                        (Product) ArrayProduct$.MODULE$.apply((Object[]) IArray.package.
            IArray$.MODULE$.map(value,
    PerspectiveUnrollingDefs$::
        perspective$circederivation$PerspectiveUnrollingDefs$BenchmarkCaseClass5$$
        anon$4$$_$apply$$anonfun$2,
                ClassTag$.MODULE$.apply(Object.class))))));
        }
        if (!(left2 instanceof Left)) throw new MatchError((Object) left2);
        DecodingFailure e = (DecodingFailure) left2.value();
        return package$.MODULE$.Left().apply((Object) e);
    }
}
```

Appendix C

Further benchmark results

C.1 Allocations

C.1.1 Scala 2

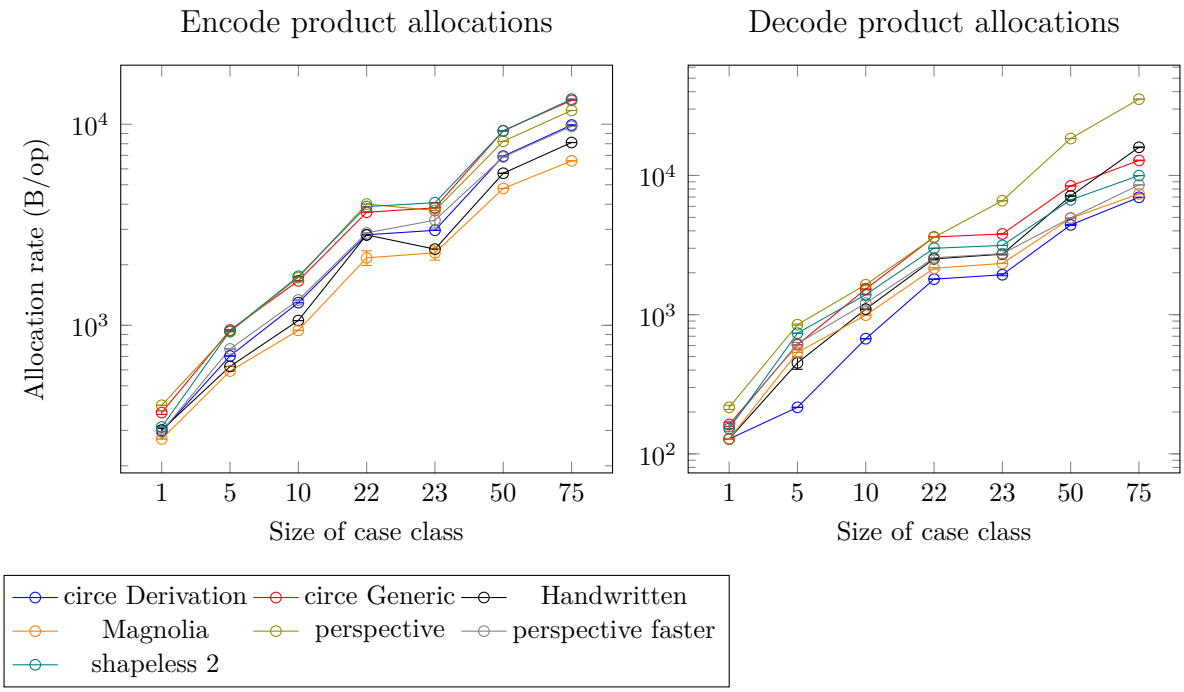

Figure C.1: Product type Scala 2 allocations.

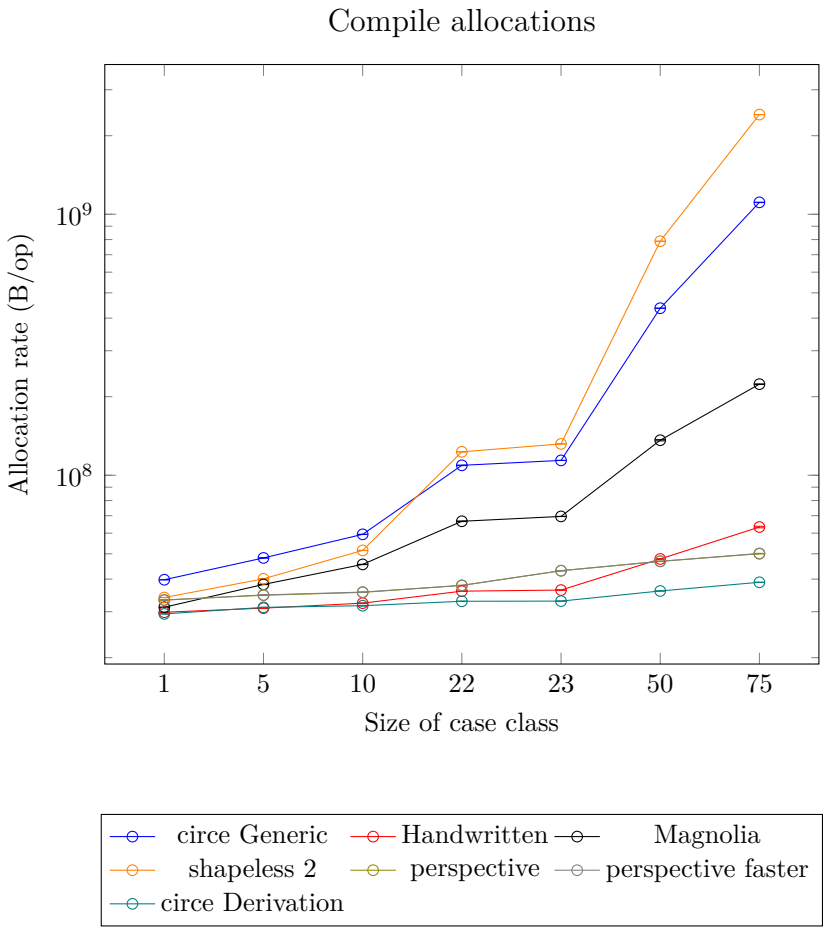

Figure C.2: Scala 2 Compile allocations.

C.1.2 Scala 3

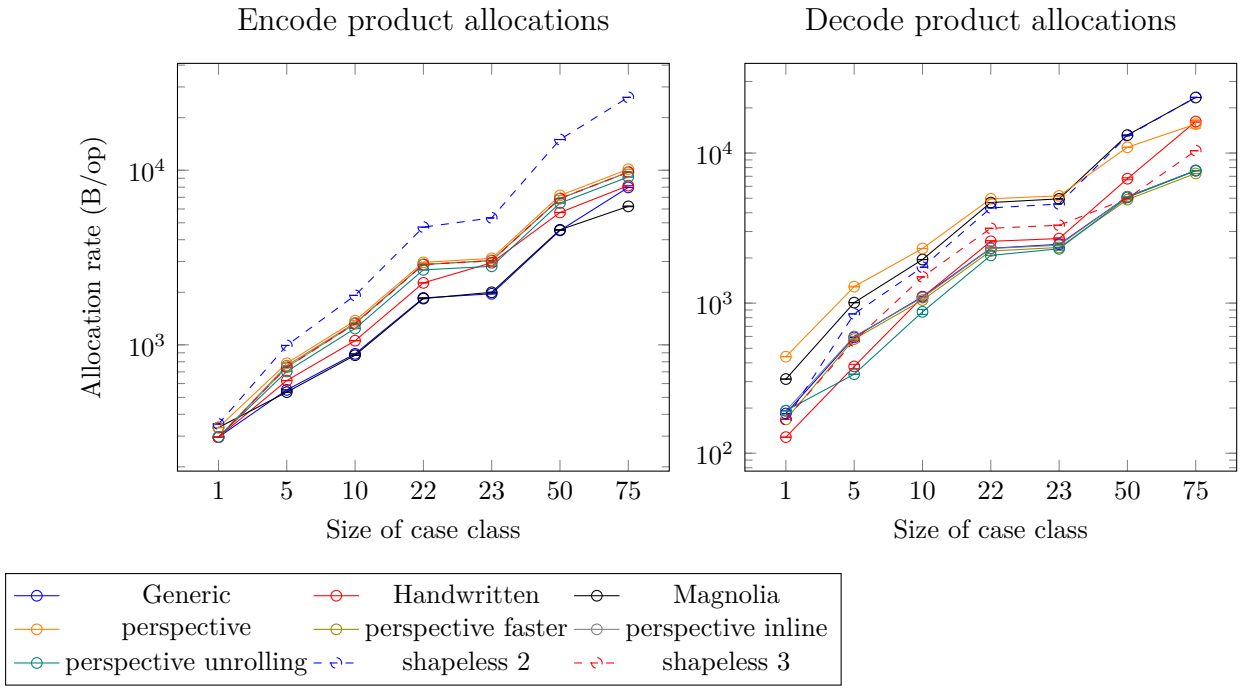

Figure C.3: Product type Scala 3 allocations.

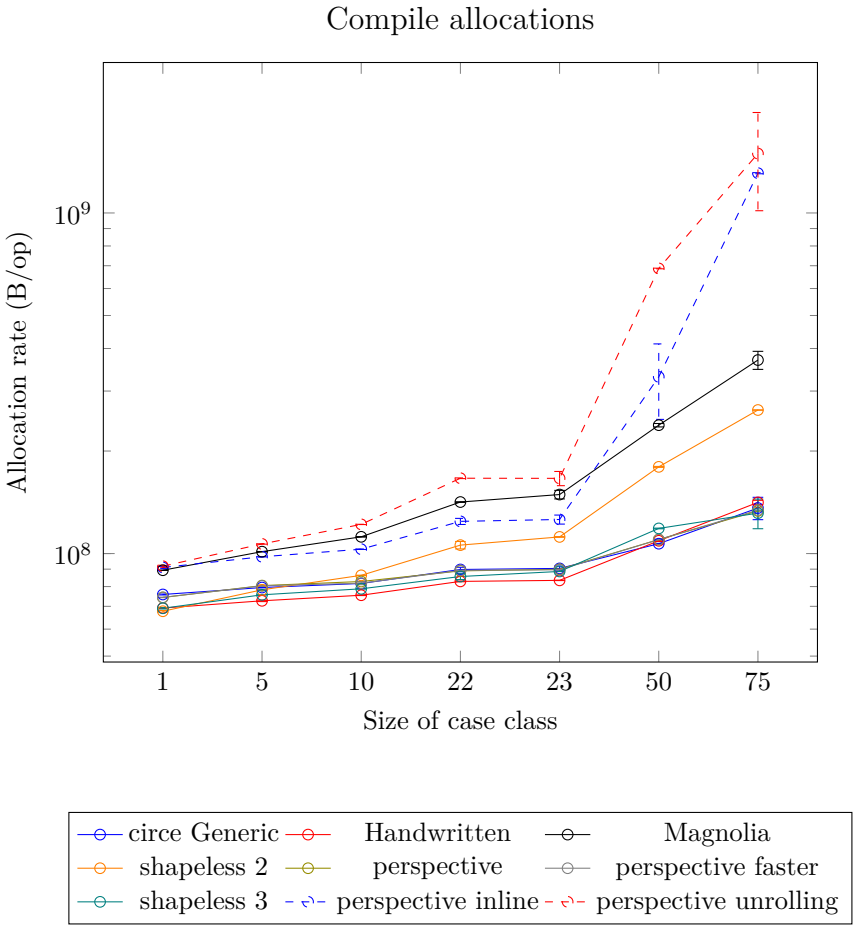

Figure C.4: Scala 3 Compile allocations.

C.2 Sum cases

C.2.1 Scala 2

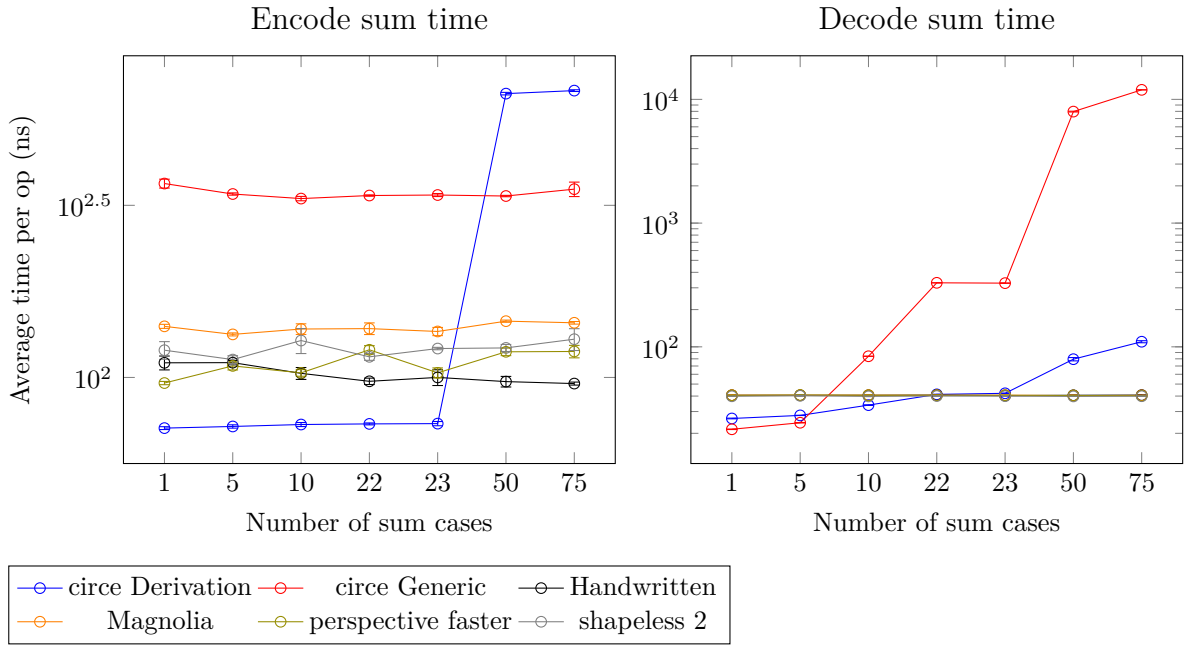

Figure C.5: First sum case Scala 2 performance.

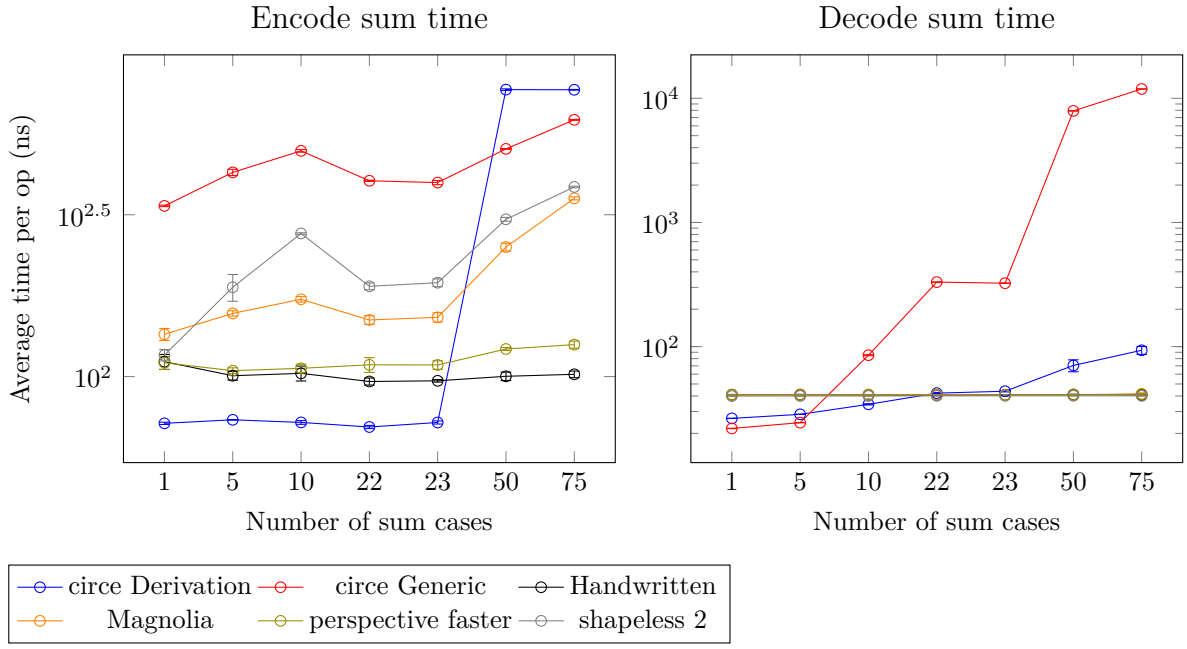

Figure C.6: Middle sum case Scala 2 performance.

C.2.2 Scala 3

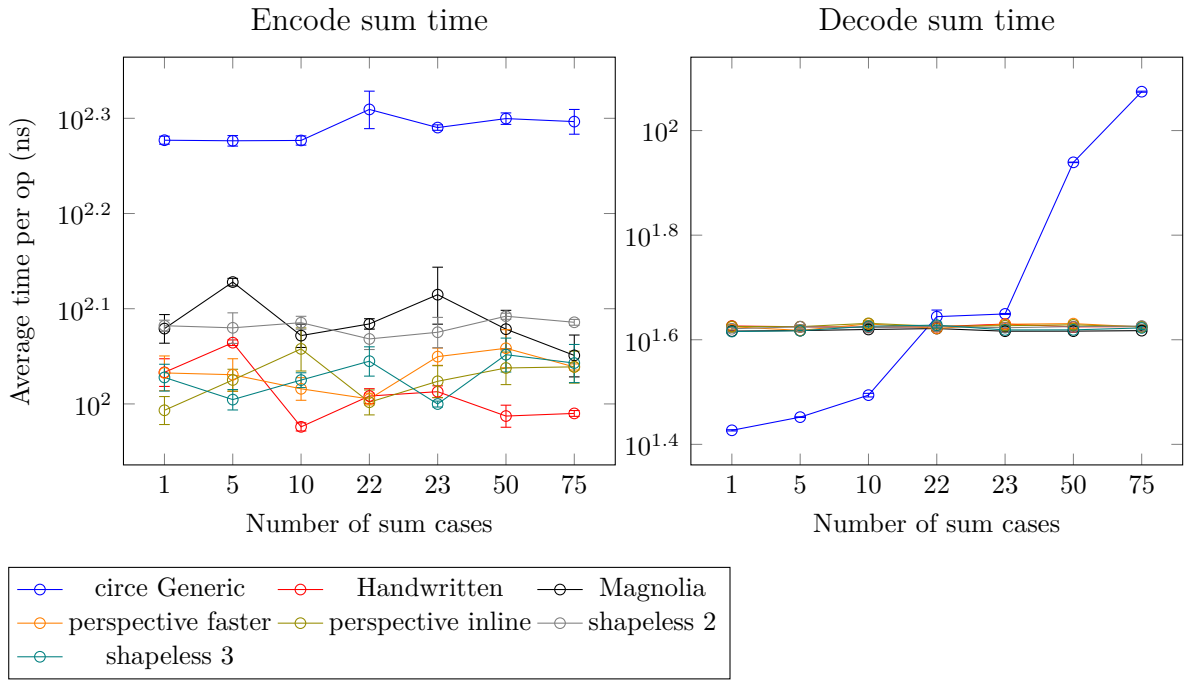

Figure C.7: First sum case Scala 3 performance.

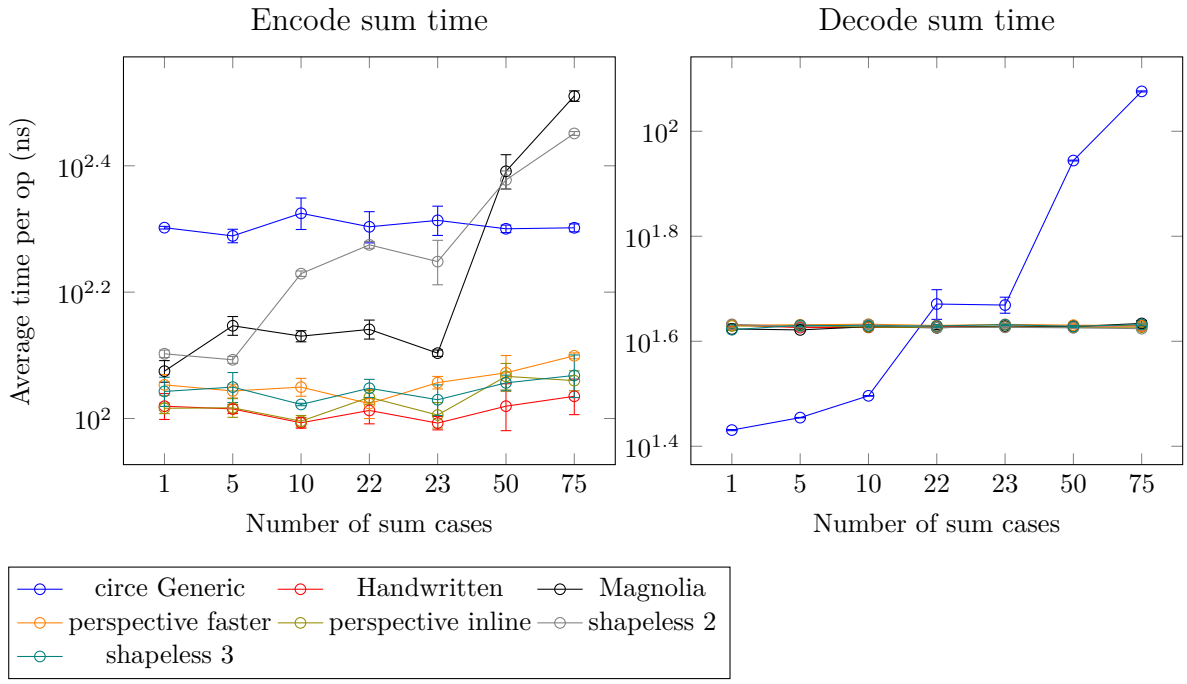

Figure C.8: Middle sum case Scala 3 performance.

C.3 Sum cases allocations

C.3.1 Scala 2

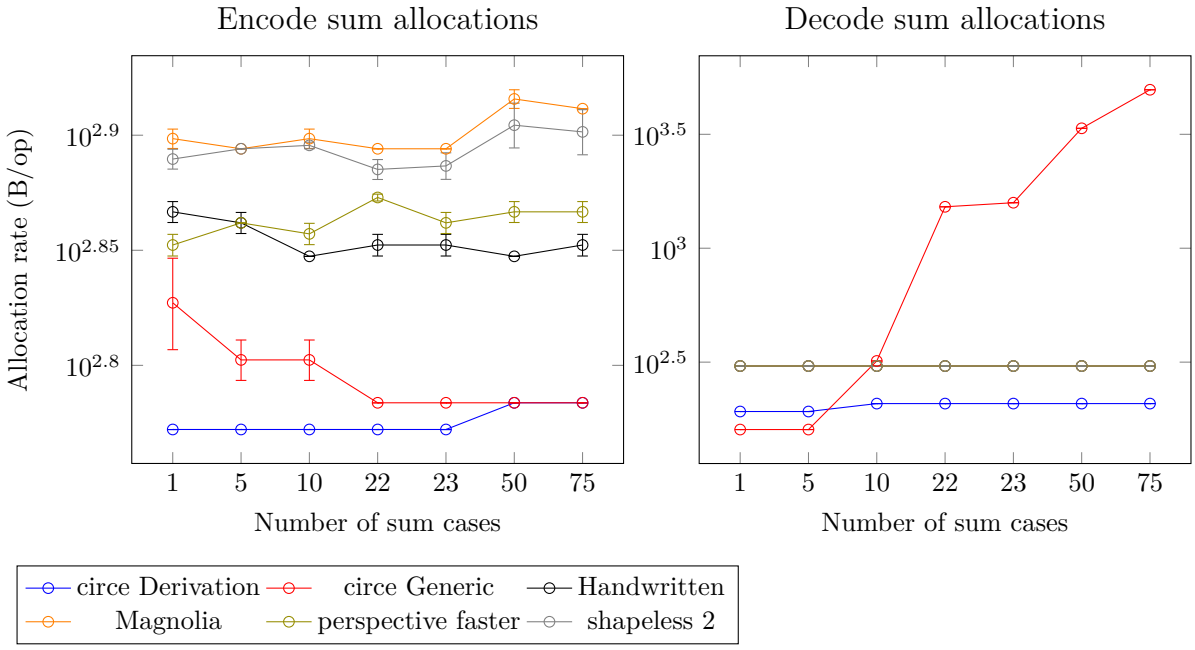

Figure C.9: First sum case Scala 2 allocations.

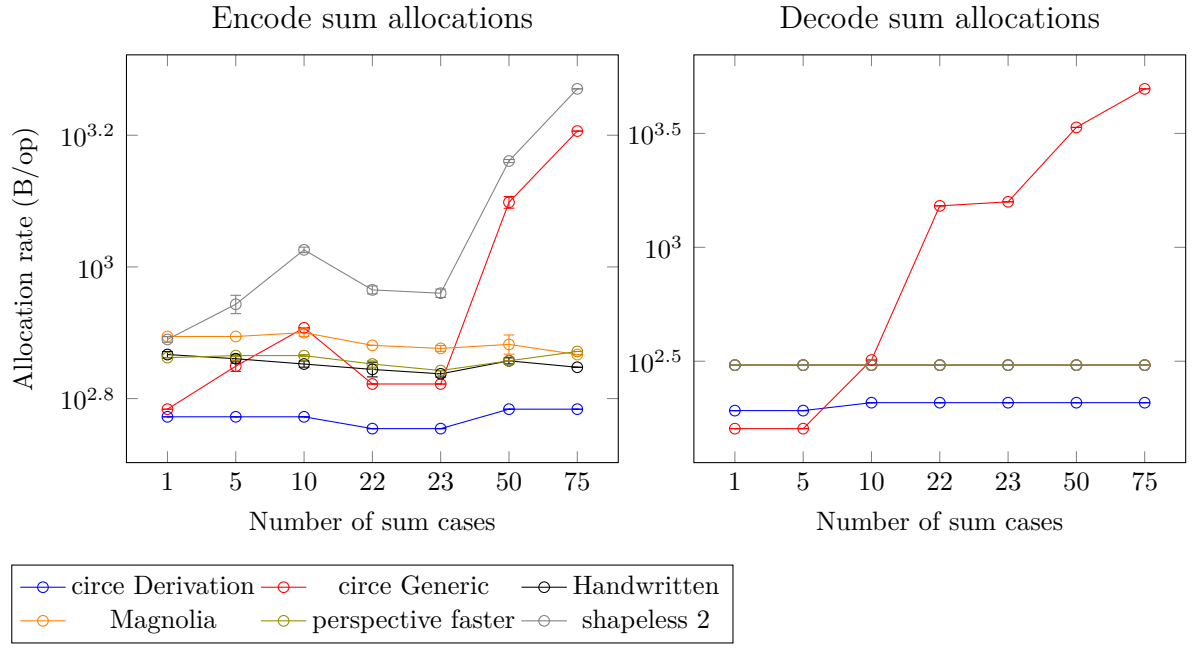

Figure C.10: Middle sum case Scala 2 allocations.

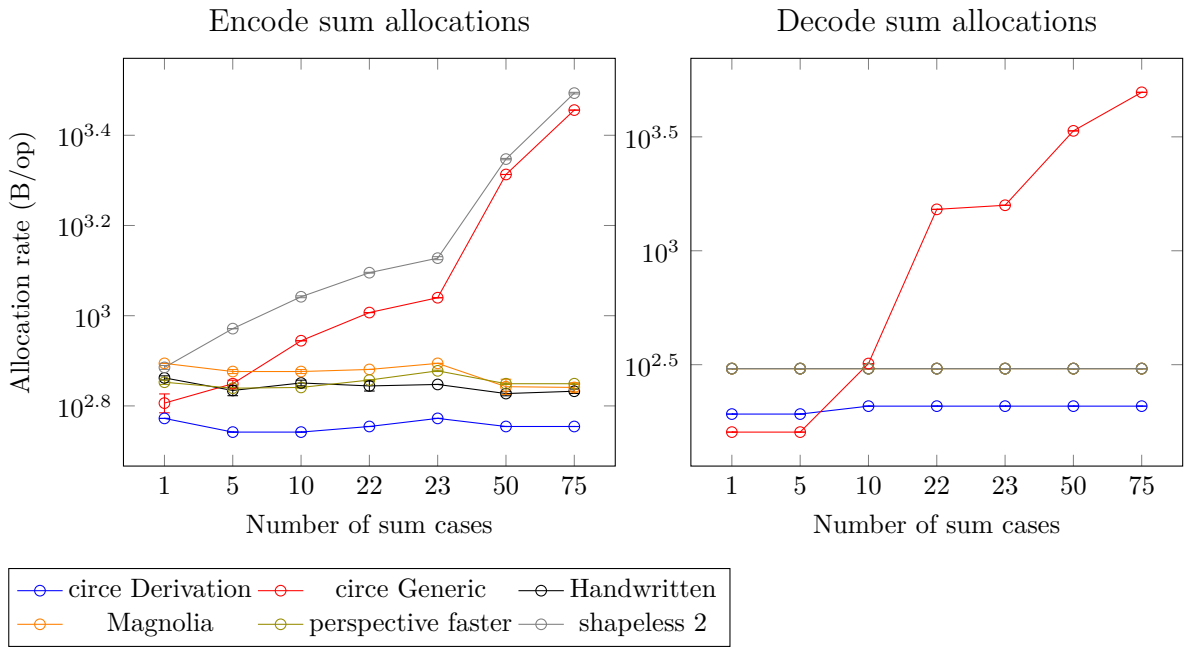

Figure C.11: Last sum case Scala 2 allocations..

C.3.2 Scala 3

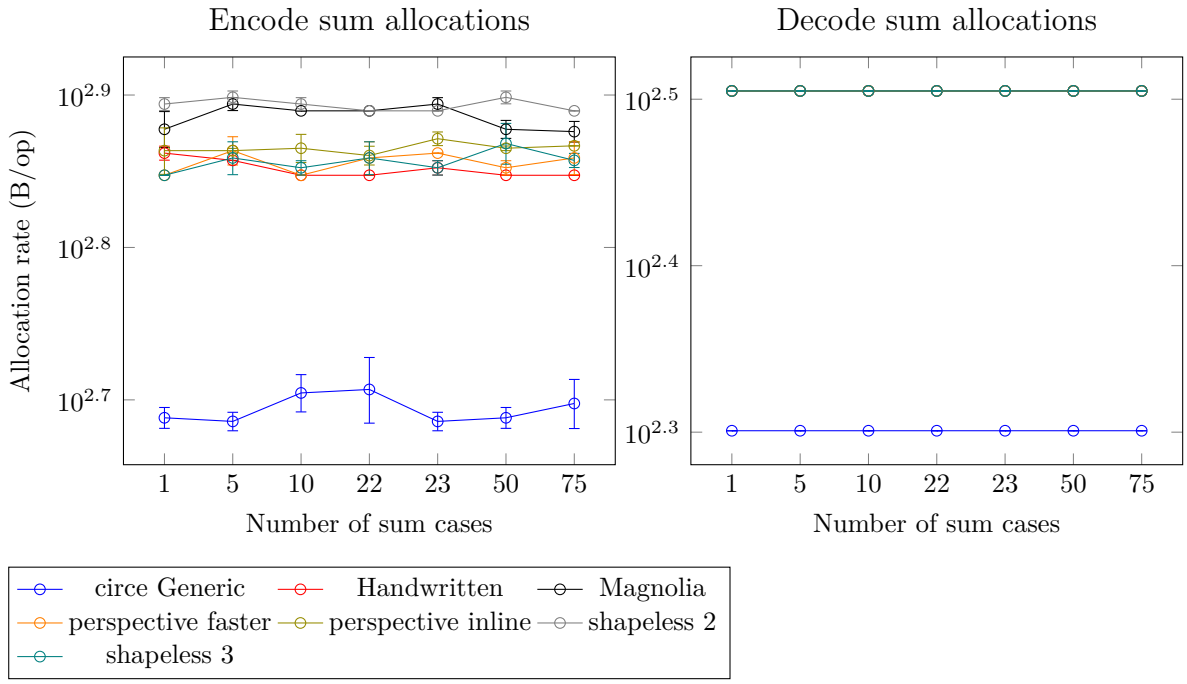

Figure C.12: First sum case Scala 3 allocations.

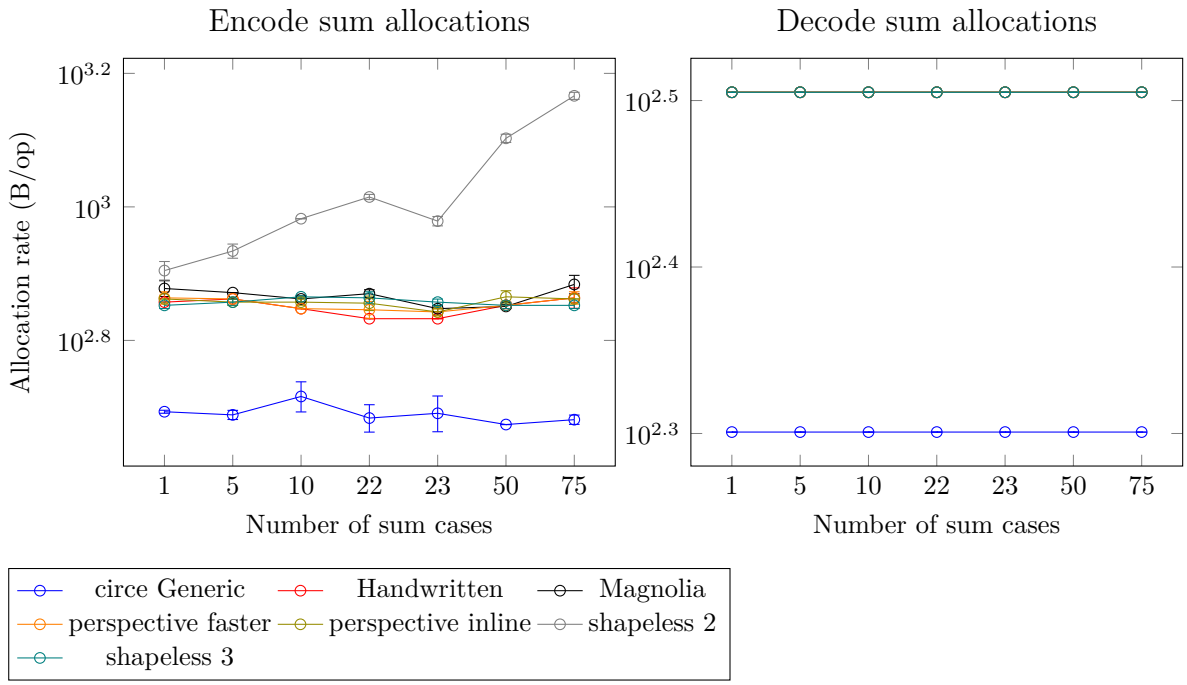

Figure C.13: Middle sum case Scala 3 allocations.

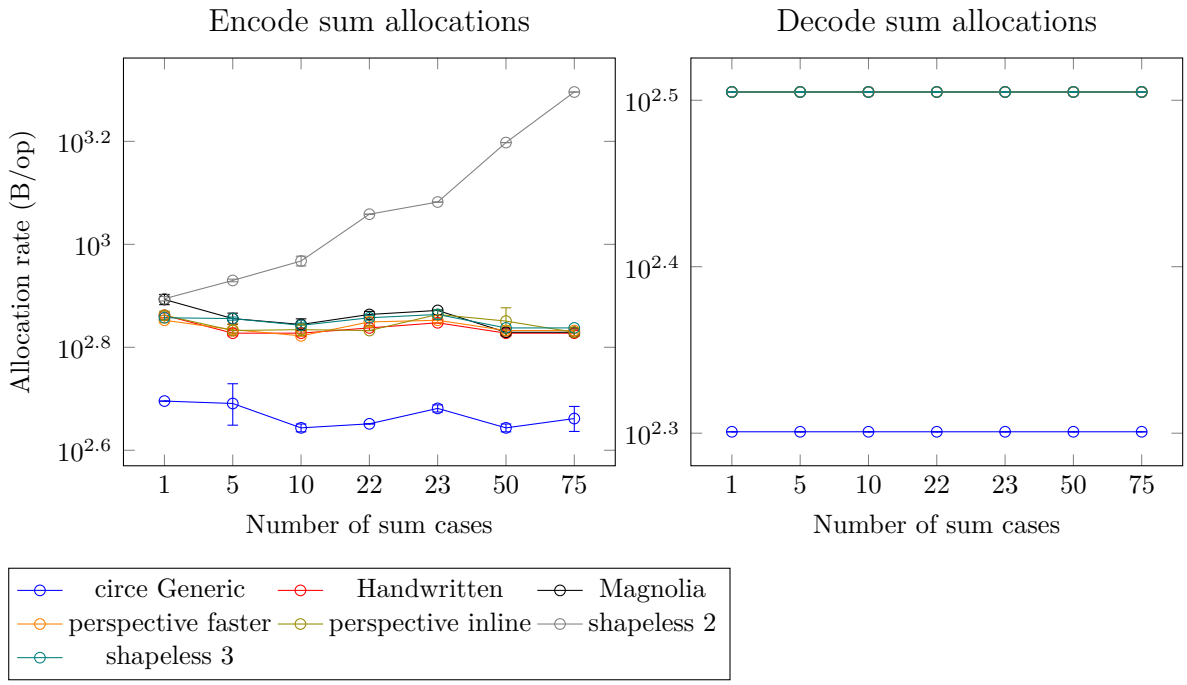

Figure C.14: Last sum case Scala 3 allocations.