Free vs Final Tagless

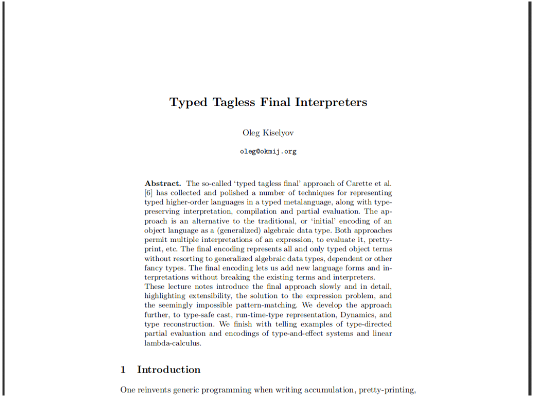
Markus Hauck (@markus1189)

codecentric

Free vs Tagless

Content

- start with the basics from Oleg's excellent paper
- Typed tagless-final interpretations: Lecture notes
- clarify tagged, tagless, initial and final
- extensibility
- inspection and transformation of programs
- Free-X and MTL vs initial and final
- comparison of both approaches
- uses Scala, but the idea is independent of the language



Typed Tagless Final Interpreters

Oleg Kiselyov

oleg@okmij.org

Abstract. The so-called ‘typed tagless final’ approach of Carette et al. [6] has collected and polished a number of techniques for representing typed higher-order languages in a typed metalanguage, along with type-preserving interpretation, compilation and partial evaluation. The approach is an alternative to the traditional, or ‘initial’ encoding of an object language as a (generalized) algebraic data type. Both approaches permit multiple interpretations of an expression, to evaluate it, pretty-print, etc. The final encoding represents all and only typed object terms without resorting to generalized algebraic data types, dependent or other fancy types. The final encoding lets us add new language forms and interpretations without breaking the existing terms and interpreters.
These lecture notes introduce the final approach slowly and in detail, highlighting extensibility, the solution to the expression problem, and the seemingly impossible pattern-matching. We develop the approach further, to type-safe cast, run-time-type representation, Dynamics, and type reconstruction. We finish with telling examples of type-directed partial evaluation and encodings of type-and-effect systems and linear lambda-calculus.

1 Introduction

One reinvents generic programming when writing accumulation, pretty-printing,

http://okmij.org/ftp/tagless-final/course/index.html

The Toy Language

- toy language with operations:
- integer constants: `Int(42)`
- integer addition: `Int(21) + Int(21)`
- string constants: `Str("4")`
- string concatenation: `Str("4") + Str("2")`
- conversion from string to integer: `StrToInt("42")`

Initial vs Final vs Tagged

Terminology

directly from the paper by Oleg Kiselyov:

> *There are **two basic approaches** to embedding languages and writing their interpreters, which we shall call, somewhat informally, **initial** and **final**.*
>
> *The **initial** approach represents a term of an object language **as a value** of an algebraic data type in the metalanguage; interpreters recursively traverse the values de-constructing them by **pattern-matching**.*
>
> *In the **final** approach, object language terms are represented as expressions built from a small set of **combinators**, which are **ordinary functions** rather than data constructors.*

Interpreter: Initial Tagged

- small language: addition, concatenation, literals and conversion from string to int

```
1 + 1
"hello," + " world"
"42".toInt
```

Interpreter: Initial Tagged

```scala
sealed abstract class Expr extends Product with Serializable
final case class IntLit(value: Int) extends Expr
final case class Add(e1: Expr, e2: Expr) extends Expr
final case class StrLit(value: String) extends Expr
final case class Concat(e1: Expr, e2: Expr) extends Expr
final case class StrToInt(e: Expr) extends Expr
```

Interpreter: Initial Tagged

```scala
def sampleProgram: Expr = StrToInt(Concat(StrLit("4"), StrLit("2")))
//Scala equivalent: ("4" + "2").toInt

def problematic: Expr = StrToInt(IntLit(42))
//Scala equivalent: 42.toInt
```

Interpreter: Initial Tagged

```
def interp(e: Expr): Any = e match {
  case IntLit(value: Int)    => value
  case StrLit(value: String) => value
  case Add(StrLit(_), e2)    => ???
  case _                     => ???
}
```

Interpreter: Initial Tagged

```scala
sealed abstract class Result
final case class IntResult(value: Int) extends Result
final case class StrResult(value: String) extends Result
```

```scala
def interp(e: Expr): Either[String, Result] = e match {
  case IntLit(value)  => IntResult(value).asRight
  case StrLit(value)  => StrResult(value).asRight
  case Add(e1, e2)    => handleAdd(e1, e2)
  case Concat(e1, e2) => handleConcat(e1, e2)
  case StrToInt(e_)   => handleStrToInt(e_)
}
```

Interpreter: Initial Tagged

```
private[this] def handleAdd(e1: Expr, e2: Expr): Either[String, Result] =
  for {
    r1 <- interp(e1)
    r2 <- interp(e2)
    result <- (r1, r2) match {
      case (IntResult(v1), IntResult(v2)) => IntResult(v1 + v2).asRight
      case _                              => s"Could not add $r1 and $r2".asLeft
    }
  } yield result
```

Interpreter: Initial Tagged

- problems: have to handle errors in interpreter
- should: don't allow invalid programs at all
- btw: this is a very nice criteria for any DSL

Tagged Initial Encoding

- initial encoding = data structure + `Result` union + error handling
- “tagged union” a.k.a. sum types in type theory
- this “tag” is used for pattern matching
- programs are first class, store as data etc

Tagless Initial Encoding

- we saw the problem of invalid programs
- next step: make invalid programs impossible
- Use GADTs: ADTs that refine the type parameter

Interpreter: Initial Tagless

```scala
sealed abstract class Expr[A] extends Product with Serializable
final case class IntLit(value: Int) extends Expr[Int]
final case class Add(e1: Expr[Int], e2: Expr[Int]) extends Expr[Int]
final case class StrLit(value: String) extends Expr[String]
final case class Concat(e1: Expr[String], e2: Expr[String]) extends Expr[String]
final case class StrToInt(e: Expr[String]) extends Expr[Int]
```

- Expr has a type parameter that is refined in subclasses
- when pattern matching, we can recover this refinement

Interpreter: Initial Tagless

```
def sampleProgram: Expr[Int] = StrToInt(Concat(StrLit("4"), StrLit("2")))

// does no longer compile:
// def problematic = StrToInt(IntLit(42))
```

Interpreter: Initial Tagless

```
def interp[A](e: Expr[A]): A = e match {
  case IntLit(value)  => value
  case StrLit(value)  => value
  case Add(e1, e2)    => handleAdd(e1, e2)
  case Concat(e1, e2) => handleConcat(e1, e2)
  case StrToInt(e_)   => handleStrToInt(e_)
}
```

Interpreter: Initial Tagless

```scala
def interp[A](e: Expr[A]): A = e match {
  case IntLit(value)  => value
  case StrLit(value)  => value
  case Add(e1, e2)    => handleAdd(e1, e2)
  case Concat(e1, e2) => handleConcat(e1, e2)
  case StrToInt(e_)   => handleStrToInt(e_)
}
```

```scala
private[this] def handleAdd(e1: Expr[Int], e2: Expr[Int]): Int =
  interp(e1) + interp(e2)
```

Interpreter: Initial Tagless

- use GADTs and make incorrect programs impossible
- gets rid of Either in the interpretation
- in summary, no reason to choose initial tagged if you have GADTs

Tagless Final Encoding

- totally different idea: avoid data structure
- use typeclass for operations and instances for the concrete implementation
- interpreter are just instances
- you get type safety out of the box (no tagged encoding)

Interpreter: Final Tagless

```scala
trait ExprSym[Expr[_]] {
  def intLit(value: Int): Expr[Int]
  def add(e1: Expr[Int], e2: Expr[Int]): Expr[Int]

  def strLit(value: String): Expr[String]
  def concat(e1: Expr[String], e2: Expr[String]): Expr[String]

  def strToInt(e: Expr[String]): Expr[Int]
}
```

Interpreter: Final Tagless

```scala
def sampleProgram[F[_]](implicit expr: ExprSym[F]): F[Int] = {
  import expr._
  strToInt(concat(strLit("4"), strLit("2")))
}
```

Interpreter: Final Tagless

```scala
case class Interp[A](value: A) extends AnyVal

implicit val exprSymInterp: ExprSym[Interp] = new ExprSym[Interp] {
  override def intLit(value: Int): Interp[Int] = Interp(value)

  override def add(e1: Interp[Int], e2: Interp[Int]): Interp[Int] =
    Interp(e1.value + e2.value)

  override def strLit(value: String): Interp[String] = Interp(value)

  override def concat(e1: Interp[String], e2: Interp[String]): Interp[String] =
    Interp(e1.value + e2.value)

  override def strToInt(e: Interp[String]): Interp[Int] =
    Interp(e.value.toInt)
}
```

First Summary

- we saw: initial tagged, initial tagless and final tagless
- implemented simple language
- next: compose languages

The Expression Problem

Philip Wadler on 12. November 1998

The Expression Problem is a new name for an old problem. The goal is to define a datatype by cases, where one can **add new cases to the datatype** and **new functions over the datatype**, without recompiling existing code, and while retaining static type safety (e.g., no casts).

- 1) add new cases to datatype
- 2) add new function over datatype
- no recompilation + static type safety

Case Study: Adding a pretty printer

(expression problem: add function over datatype)

Case Study: Adding a pretty printer

- instead of evaluating a program, pretty print it
- by adding a special interpreter
- corresponds to the second case of the expression problem (new function over the datatype)
- start with initial encoding
- then do final encoding

Initial Encoding: Pretty Printer

- add a new function
- pattern match on our `Expr` type

Initial Encoding: Pretty Printer

```scala
def prettyPrint[A](e: Expr[A]): String = e match {
  case IntLit(value)  => s"Int($value)"
  case Add(e1, e2)    => s"${prettyPrint(e1)} + ${prettyPrint(e2)}"
  case StrLit(value)  => s"Str($value)"
  case Concat(e1, e2) => s"${prettyPrint(e1)} + ${prettyPrint(e2)}"
  case StrToInt(e)    => s"str2int(${prettyPrint(e)})"
}
```

```scala
def sampleProgram: Expr[Int] = StrToInt(Concat(StrLit("4"), StrLit("2")))
prettyPrint(sampleProgram) // => "str2int(Str(4) + Str(2))"
```

Initial Encoding: Pretty Printer

- in general, adding interpreters is easy in the initial encoding
- just define a new function and use pattern matching
- what about final?

Final Encoding: Pretty Printer

- adding a pretty printer means defining a new `ExprSym` instance
- as usual: define a new type to attach the instance to

Final Encoding: Pretty Printer

```scala
case class PP[A](value: String)

implicit val expSymPrint: ExprSym[PP] = new ExprSym[PP] {
  override def intLit(value: Int): PP[Int] = PP(s"Int($value)")

  override def add(e1: PP[Int], e2: PP[Int]): PP[Int] =
    PP(s"(${e1.value} + ${e2.value})")

  override def strLit(value: String): PP[String] = PP(s"Str($value)")

  override def concat(e1: PP[String], e2: PP[String]): PP[String] =
    PP(s"(${e1.value} + ${e2.value})")

  override def strToInt(e: PP[String]): PP[Int] =
    PP(s"str2int(${e.value})")
}
```

Final Encoding: Pretty Printer

```scala
def sampleProgram[F[_]](implicit expr: ExprSym[F]): F[Int] = {
  import expr._
  strToInt(concat(strLit("4"), strLit("2")))
}

sampleProgram[PP].value // => str2int((Str(4) + Str(2)))
```

Final Encoding: Pretty Printer

- add pretty printer using newtype and typeclass instance
- no need to touch any existing code
- final tagless supports the introduction of new functions over the datatype

Case Study: Adding “if”

(expression problem: add cases to datatype)

Case Study: Adding “if”

- the goal is to add the “if” to our language
- needed parts:
  - a way to introduce booleans
  - the actual if construct

Case Study: If with Initial Tagless

```scala
sealed abstract class Expr[A] extends Product with Serializable
final case class IntLit(value: Int) extends Expr[Int]
final case class Add(e1: Expr[Int], e2: Expr[Int]) extends Expr[Int]
final case class StrLit(value: String) extends Expr[String]
final case class Concat(e1: Expr[String], e2: Expr[String]) extends Expr[String]
final case class StrToInt(e: Expr[String]) extends Expr[Int]

final case class BoolLit(value: Boolean) extends Expr[Boolean]
final case class If[A](
    condition: Expr[Boolean],
    ifTrue: () => Expr[A],
    ifFalse: () => Expr[A]
) extends Expr[A]
```

Case Study: If with Initial Tagless

```
def sampleProgram: Expr[Int] =
  If(BoolLit(true), { () =>
    IntLit(42)
  }, { () =>
    IntLit(21)
  })
```

Case Study: If with Initial Tagless

```scala
def interp[A](e: Expr[A]): A = e match {
  case IntLit(value)  => value
  case Add(e1, e2)    => handleAdd(e1, e2)
  case StrLit(value)  => value
  case Concat(e1, e2) => handleConcat(e1, e2)
  case StrToInt(e_)   => handleStrToInt(e_)
  case BoolLit(value) => value
  case If(c, t, f)    => handleIf(c, t, f)
}
```

Case Study: If with Initial Tagless

```scala
  private[this] def handleIf[A](
      condition: Expr[Boolean],
      ifTrue: () => Expr[A],
      ifFalse: () => Expr[A]
  ): A =
    if (interp(condition)) interp(ifTrue()) else interp(ifFalse())
```

Case Study: If with Initial Tagless

- this is what the expression problem is all about
- had to touch the language and **all** interpreters
- problem: what if we regularly extend the language?
- better: if we could compose languages instead of changing
- initial solution: datatypes à la carte

Datatypes à la carte

- Wouter Swiestra: Data types à la carte
- demonstrating the use of fixed point and parameterized expressions over a type constructor
- use a typeclass to inject languages into a coproduct
- in cats: **InjectK** and `Inject`
- but in summary: big pain, you probably don't want to go there
- instead: let's look at final tagless version

Case Study: If with Final Tagless

```
trait ExprIf[Expr[_]] {
  def boolLit(value: Boolean): Expr[Boolean]
  def intToBool(e: Expr[Int]): Expr[Boolean]
  def ifExpr[A](c: Expr[Boolean])(ifTrue: () => Expr[A])(
      ifFalse: () => Expr[A]
  ): Expr[A]
}
```

Case Study: If with Final Tagless

```scala
def sampleProgram[F[_]](
    implicit exprSym: ExprSym[F],
    exprIf: ExprIf[F]
): F[String] = {
  import exprIf._
  import exprSym._

  val condition: F[Boolean] = intToBool(strToInt(strLit("42")))
  ifExpr(condition)(() => strLit("it was true"))(() => strLit("it was false"))
}
```

Case Study: If with Final Tagless

```scala
// NO Interp class! Re-use it!

implicit val exprIfInterp: ExprIf[Interp] = new ExprIf[Interp] {
  override def boolLit(value: Boolean): Interp[Boolean] = Interp(value)

  override def intToBool(e: Interp[Int]): Interp[Boolean] =
    Interp(e.value == 0)

  override def ifExpr[A](
      c: Interp[Boolean]
  )(ifTrue: () => Interp[A])(ifFalse: () => Interp[A]): Interp[A] =
    if (c.value) ifTrue() else ifFalse()
}
```

Case Study: If with Final Tagless

- no need to touch `Interp` class, just add an instance
- we are able to re-use the **ExprSym** in programs
- this solves the expression problem, we did not have to change existing things
- and without all the hassle of `Inject` and datatypes à la carte
- this is the big advantage of the final tagless encoding

Recap: Case Studies

- initial encoding: easy to add new functions, hard to extend datatype
- datatypes à la carte can remedy this, at some cost
- final tagless encoding: easy to extend language **and** easy add new functions
- if you need extensibility in both dimensions, use datatypes à la carte or final encoding

Working With Programs

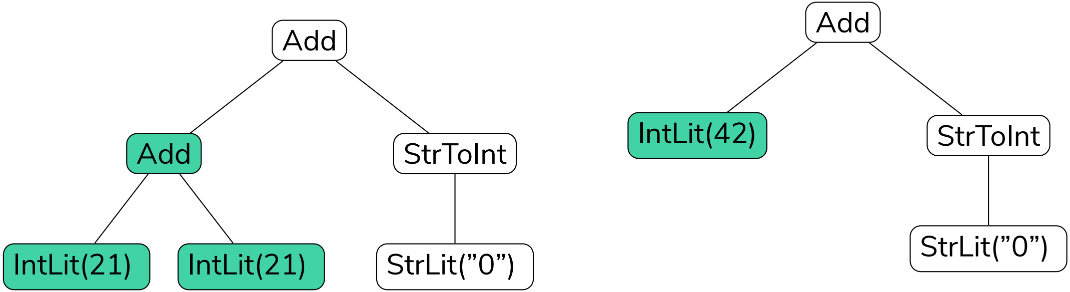
Optimization

- time to talk about program optimization and transformation
- DSL: program is written once, interpreted many times
- myth: inspection/transformation impossible in finally tagless

Optimization: Inlining of Addition

- goal: inline addition with literals
- i.e. all `Add` with only `IntLit` children

Optimization: Inlining of Addition



Initial Encoding: Inlining of Addition

- initial encoding: we are building the tree and use pattern matching
- the program tree looks like this in the DSL

```
Add(Add(IntLit(21), IntLit(21)), StrToInt(StrLit("0")))
```

Initial Encoding: Inlining of Addition

```scala
  Add(Add(IntLit(21), IntLit(21)), StrToInt(StrLit("0")))
```

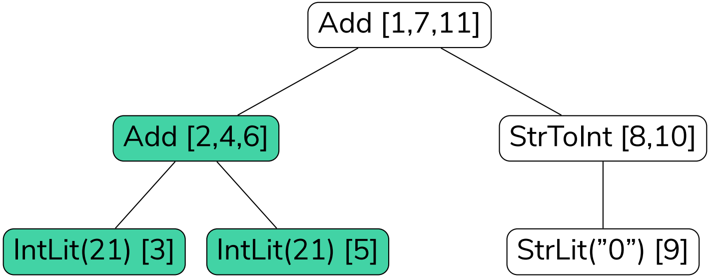
```scala
def inlineAddition1[A](program: Expr[A]): Expr[A] = program match {
  case Add(IntLit(lhs), IntLit(rhs)) => IntLit(lhs + rhs)
  case Add(lhs, rhs)                 => Add(inlineAddition1(lhs), inlineAddition1(rhs))
  case _                             => program // why can we cheat here?
}

def inlineAddition[A](program: Expr[A]): Expr[A] =
  fixpoint[Expr[A]](program)(inlineAddition1)
```

Final Encoding: Inlining of Addition

- with the final encoding there is no program
- no pattern matching on the AST
- trick: explicate the necessary context using a special instance
- keep track of predecessors during traversal

Final Encoding: Inlining of Addition



1	2	3	4	5	6	7	8	9	10	11
Add	Add	IntLit(21)	Add	IntLit(21)	Add	Add	StrToInt	StrLit(0)	StrToInt	Add

Final Encoding: Inlining of Addition

```scala
sealed abstract class Ctx // explicit context needed
object Ctx {
  case class CtxInt(value: Int) extends Ctx
  case object CtxAdd extends Ctx
  case object CtxOther extends Ctx
}
```

```scala
case class Opt[F[_], A](run: List[Ctx] => (List[Ctx], F[A]))
```

```scala
// Using kind-projector
implicit def inlineAdditionExprSym[F[_]](
    implicit base: ExprSym[F]
): ExprSym[Opt[F, ?]] = ???
```

Final Encoding: Inlining of Addition

```scala
// def inlineAdditionExprSym[F[_]](...) = {
  override def intLit(value: Int): Opt[F, Int] =
    Opt(ctx => (CtxInt(value) :: ctx, base.intLit(value)))

  override def add(e1: Opt[F, Int], e2: Opt[F, Int]): Opt[F, Int] = Opt { ctx0 =>
    val (ctx1, v1) = e1.run(CtxAdd :: ctx0)
    val (ctx2, v2) = e2.run(CtxAdd :: ctx1)

    ctx2 match {
      case CtxInt(lhs) +: CtxAdd +: CtxInt(rhs) +: CtxAdd +: ctxs =>
        (CtxInt(lhs + rhs) :: ctxs, base.intLit(lhs + rhs))
      case _ => (CtxAdd :: ctx2, base.add(v1, v2))
    }
  }

  // more overrides...
// }
```

Final Encoding: Inlining of Addition

- making the context explicit is non-mechanic
- you lose the pattern matching language from initial
- in a nutshell, this is the big trade-off
- still, you can do every optimization in final **and** initial
- some things are just really hard (like de-/serialization)
- every inspection = run the program (overhead)

Final Encoding: Inlining of Addition

- making the context explicit is non-mechanic
- you lose the pattern matching language from initial
- in a nutshell, this is the big trade-off
- still, you can do every optimization in final **and** initial
- some things are just really hard (like de-/serialization)
- every inspection = run the program (overhead)
- State Monad anyone?

```scala
case class Opt[F[_], A](run: List[Ctx] => (List[Ctx], F[A]))
```

Final Encoding: Inlining of Addition

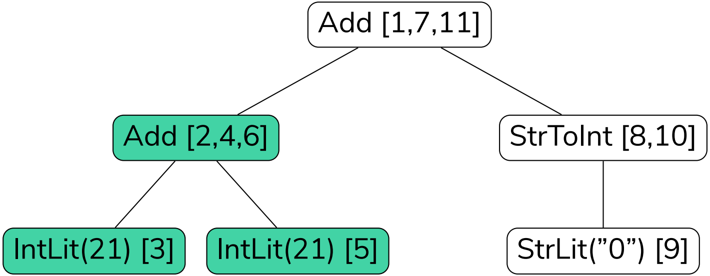
- making the context explicit is non-mechanic
- you lose the pattern matching language from initial
- in a nutshell, this is the big trade-off
- still, you can do every optimization in final **and** initial
- some things are just really hard (like de-/serialization)
- every inspection = run the program (overhead)
- State Monad anyone?

```
case class Opt[F[_], A](run: List[Ctx] => (List[Ctx], F[A]))
```

- Quiz: what is the problem with this interpreter?

Final Encoding: Inlining of Addition

- we **always** traverse the left branch



- we should instead look ahead before traversing down

Final Encoding: Inlining of Addition

- we can do this by writing a more clever instance
- same idea of wrapping a base interpreter
- use a tuple of the actual interpreter (that is delayed using a thunk) and our look ahead interpreter
- for `add`, first peek at the two branches, only go down if no optimization applies
- full code is on github in the `Lookahead` object (link at the end)

Recap: Working With Programs

- initial encoding: programs exists as data
- cheap to inspect and transform using pattern matching
- final encoding: programs are functions
- inspection and transformation is possible, but more work

Free And MTL

Free As Initial Encoding

- Free is an initial encoding
- but a Free X is associated to a **typeclass** + laws
- the minimal **initially encoded** structure satisfying the laws
- Free Monad, Free Applicative, Free Monoid
- initial encoding + DSL based on typeclass

Interpreter: Free Monad

```scala
sealed abstract class ExprF[A] extends Product with Serializable
final case class IntLit(value: Int) extends ExprF[Int]
final case class Add(e1: Int, e2: Int) extends ExprF[Int]
final case class StrLit(value: String) extends ExprF[String]
final case class Concat(e1: String, e2: String) extends ExprF[String]
final case class StrToInt(e: String) extends ExprF[Int]
```

Interpreter: Free Monad — Hide Injects

```scala
type Expr[A] = Free[ExprF, A]

def intLit(value: Int): Expr[Int] = Free.inject(IntLit(value))
def add(e1: Int, e2: Int): Expr[Int] = Free.inject(Add(e1, e2))

def strLit(value: String): Expr[String] = Free.inject(StrLit(value))
def concat(e1: String, e2: String): Expr[String] = Free.inject(Concat(e1, e2))

def strToInt(e: String): Expr[Int] = Free.inject(StrToInt(e))
```

Interpreter: Free Monad

```scala
def sampleProgram: Expr[Int] =
  for {
    four <- strLit("4")
    two <- strLit("2")
    concatenated <- concat(four, two)
    result <- strToInt(concatenated)
  } yield result
```

Interpreter: Free Monad

```scala
def interp[A, M[_]: Monad](expr: Expr[A]): M[A] =
  expr.foldMap(new (ExprF ~> M) {
    override def apply[X](fa: ExprF[X]): M[X] = fa match {
      case IntLit(value)  => Monad[M].pure(value)
      case Add(e1, e2)    => Monad[M].pure(e1 + e2)
      case StrLit(value)  => Monad[M].pure(value)
      case Concat(e1, e2) => Monad[M].pure(e1 + e2)
      case StrToInt(e)    => Monad[M].pure(e.toInt)
    }
  })
```

Interpreter: Free Monad

- we use Monad to embed our language
- extract the real value at every step
- nice: interop with standard Scala
- nice: lots of combinators for monads exist already
- bad: interpreter fixes sequential evaluation
- but just an example, could've used `FreeApplicative`

MTL as Final Encoding

- MTL: define type class for "additional" operations of a special Monad
- e.g. accessing state (`MonadState`), reading environment (`MonadReader`), etc.
- implemented in the `mtl` package in Haskell
- commonly referred to as "mtl-style"
- = final tagless encoding that additionally has typeclass ops

Interpreter: MTL

without mtl:

```
trait ExprSym[Expr[_]] {
  def intLit(value: Int): Expr[Int]
  def add(e1: Expr[Int], e2: Expr[Int]): Expr[Int]

  def strLit(value: String): Expr[String]
  def concat(e1: Expr[String], e2: Expr[String]): Expr[String]

  def strToInt(e: Expr[String]): Expr[Int]
}
```

Interpreter: MTL

with mtl:

```scala
trait ExprMSym[F[_]] {
  def intLit(value: Int): F[Int]
  def add(e1: Int, e2: Int): F[Int]

  def strLit(value: String): F[String]
  def concat(e1: String, e2: String): F[String]

  def strToInt(e: String): F[Int]
}
```

Interpreter: MTL

```scala
def sampleProgram[F[_]: Monad](implicit expr: ExprMSym[F]): F[Int] = {
  import expr._

  for {
    four <- strLit("4")
    two <- strLit("2")
    concatenated <- concat(four, two)
    result <- strToInt(concatenated)
  } yield result
}
```

Interpreter: MTL

```scala
case class Interp[A](value: A) extends AnyVal // add Monad instance (Identity)

implicit val exprSymInterp: ExprMSym[Interp] = new ExprMSym[Interp] {
  override def intLit(value: Int): Interp[Int] = Interp(value)

  override def add(e1: Int, e2: Int): Interp[Int] = Interp(e1 + e2)

  override def strLit(value: String): Interp[String] = Interp(value)

  override def concat(e1: String, e2: String): Interp[String] =
    Interp(e1 + e2)

  override def strToInt(e: String): Interp[Int] = Interp(e.toInt)
}
```

Interpreter: MTL

- sequencing no longer part of our language (Monad)
- flexible choice of target Monad for pluggable effects
- we can choose between type classes easily (vs. Free-X)
- for example try to combine program using `Applicative` and another using `Monad` (hard using Free constructions!)

Recap: Free vs MTL

- Free X = initial encoding + typeclass X
- MTL = final tagless + typeclass
- level of introspection of programs greatly dependent on typeclass constraints
- for example with `Monad`, severely crippled
- advice: if you need `Monad` just go with final tagless every time

Comparison

Free vs Finally Tagless

- seen both approaches
- but important question: when to use which
- spoiler: it depends
- but first: can I have my cake and eat it, too?
- turns out: yes

Final vs Initial: Final2Initial

```scala
def finalToInitial[F[_]: ExprSym, A](p: F[A]): ExprSym[Expr] =
  new ExprSym[Expr] {
    override def intLit(value: Int): Expr[Int] = IntLit(value)
    override def add(e1: Expr[Int], e2: Expr[Int]): Expr[Int] = Add(e1, e2)
    override def strLit(value: String): Expr[String] = StrLit(value)
    override def concat(e1: Expr[String], e2: Expr[String]): Expr[String] =
      Concat(e1, e2)
    override def strToInt(e: Expr[String]): Expr[Int] = StrToInt(e)
  }
```

Final vs Initial: Initial2Final

```scala
def initialToFinal[F[_], A](p: Expr[A])(implicit interp: ExprSym[F]): F[A] =
  p match {
    case IntLit(value)  => interp.intLit(value)
    case Add(e1, e2)    => interp.add(initialToFinal(e1), initialToFinal(e2))
    case StrLit(value)  => interp.strLit(value)
    case Concat(e1, e2) => interp.concat(initialToFinal(e1), initialToFinal(e2))
    case StrToInt(e)    => interp.strToInt(initialToFinal(e))
  }
```

Final vs Initial

- by going back and forth, you can enjoy the advantages of both approaches
- you don’t have to commit to either in your implementations
- example: write complex transformations using initial encoding, “compile” back to final encoding for repeated execution
- or: go from final tagless to initial for complex program transformation, then back again

Initial Tagless Encoding / Free X

- programs are first class
- **easy** to add interpreters, **hard** to change language
- hard to compose languages
- easy to optimize/transform/serialize/partially evaluate
- need **Monad**, use **final tagless**!
- structure built and torn down again
- requires GADTs to get type safety
- **inspect** more often than evaluate

Final Tagless Encoding / MTL

- passing around programs as arguments can get tricky
- easy to add interpreters **and** extend language and compose
- harder to optimize/transform/serialize/partially evaluate
- flexible choice of typeclass constraint
- execution is faster because no structure is built
- it's only typeclasses and instances
- **evaluate** more often than inspect

Conclusion

- overview of final and tagless encoding
- going from tagged to tagless
- going from initial tagless to final tagless
- relation between initial / final and free
- optimization / introspection of programs
- guidelines when to choose what

The End

THANKS!

(@markus1189)

References

- Typed tagless-final interpretations, lecture notes: http://okmij.org/ftp/tagless-final/course/index.html
- Datatypes à la carte: https://www.cs.ru.nl/ W.Swierstra/Publications/DataTypesALaCarte.pdf
- Source Code: https://github.com/markus1189/free-vs-tagless

Questions?