Contents

1 Meeting 18.09

1.1 static methods

A static method is a function associated with a class itself, rather than with any specific instance (object) of that class. Static method are called directly using the class name without needing to create an object first. They are often used for utility functions or calculations that are logically grouped within a class but do not depend on the state of any particular object.

A UPL example of static and non-static method calls:

```
theory some_theory {
    static_method = ...
    non_static_method: ...
}
// static method call on class/theory name without
   needing to create an object/instance first
some_theory.static_method()

some_instance = some_theory {
    non_static_methods = ...
}
// non static method call on object/instance name
some_instance.non_static_method()
```

In UPL static methods are concrete methods that are fully defined in the theory while non-static methods are abstract in the theory and only made concrete in instantiations.

Also methods can be defined inside theories or outside theories in the scope of the module. (non-static vs static return)

```
theory M {
    bind: ...
}
```

```
module M {
    return: ...
}
```

```
theory M {
    bind: ...
    return: ...
}
```

Technically a concrete method in an abstract theory is more similar to a Java static method than a method defined for a module. We can add a concrete method to a theory without worrying about breaking code in instantiations of the theory.

1.2 Monads in UPL

Theories for monad and associated OptionMonad, ListMonad, SetMonad for the built-in types option, list, set. We would wish to formalize a theory graph like this:

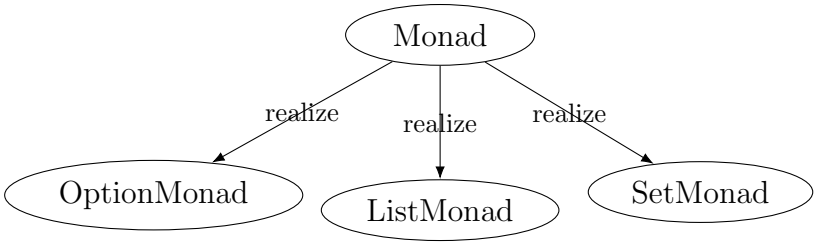

Figure 1: Theory graph (realize hierarchy) of functional programming concepts. Nodes are theories and edges are realize relations. Root theory monad with higher-kinded type constructor variable M, return and bind operations, and the monad laws. The monad theory can only be implemented with a - as of yet unavailable - *higher-kinded type variable* of kind `* -> *`. The less abstract theories OptionMonad, ListMonad, SetMonad can be implemented with the available *higher-kinded types* option, list, set.

```
theory Monad {
    type A
    type B
    type M[_]: * -> *
    return: A -> M[A]
    bind:
        M[A] -> (A -> M[B])
            -> M[B]
}
```

```
OptionMonad = Monad {
    type M[_] = option[_]
    return =
        (x: A) -> option[x]
    bind =
        (ox: option[A]) ->
        (f: A -> option[B])
        -> {...}
}
```

Can successfully write theories for Option-, List-, SetMonad, because built-in higher-kinded types (collection types) exist.

```
type A
type MA1 = option[A]
type MA2 = list[A]
type MA3 = set[A]
```

But cannot write a generalized Monad theory with a higher-kinded type variable.

```
type A
type B
type M[_] // need both M[A] and M[B], wildcard type _
```

Implemented OptionMonad and ListMonad (...) and they work.

```
theory OptionMonad {
    type X
    type Y

    return: X -> option[X] = (x: X) -> [x]

    bind: option[X] -> (X -> option[Y]) -> option[Y]
    bind = (ox: option[X]) -> (f: X -> option[Y]) -> {
        ox match {
            [] -> []
            [x] -> f(x)
        }
    }
}
```

```
theory ListMonad {
    type X
    type Y

    return: X -> list[X] = (x: X) -> [x]

    // need append for bind
    append: (list[Y], list[Y]) -> list[Y]
    append = (xs, ys) -> {
        xs match {
            [] -> ys
            x -: xs2 -> x -: append(xs2, ys)
        }
    }

    // lists as nondeterministic computations
    bind: list[X] -> (X -> list[Y]) -> list[Y]
    bind = (lx: list[X]) -> (f: X -> list[Y]) -> {
        lx match {
            [] -> []
            x -: xs -> append(f(x), bind(xs)(f))
        }
    }
}
```

But we cannot generalize to a monad theory without higher-kinded type variables.

1.3 Higher-kinded type operator M

As yet we cannot generalize to a Monad theory with a higher-kinded type variable `M` in UPL

```
type A
type B
type M[_] // need both M[A] and M[B], wildcard type _
```

It is also not possible to write

```
type A
type M[A]
//
//
```

```
type A
type B
type M[A]
type M[B]
```

UPL has types of kind `*` (int, string)
and collection types of kind `* -> *` (option[int], list[X], set[Y]).

It is possible to write

```
type X
type Y = option[X]
```

but Y is only simple/nullary and not a higher-kinded type variable.

1.3.1 Scala 2 & Scalaz

Scala 2.13 natively enables higher-kinded type variables with the wildcard type:

```
trait Monad[M[_]] {
  def pure[A](x: A): M[A]
  def bind[A, B](mx: M[A], f: (A => M[B])): M[B]
}
object Monad {
  implicit val MonadOption: Monad[Option] = new Monad[
     Option] {
    def pure[A](x: A) = Some(x)
    def bind[A, B](mx: Option[A], f: (A => Option[B])) =
        mx match {
      case None => None
      case Some(x) => f(x)
    }
  }
  implicit val MonadList: Monad[List] = new Monad[List]
     {
    def pure[A](x: A) = List(x)

    def bind[A, B](mx: List[A], f: (A => List[B])) = mx
       match {
      case Nil => Nil
      case x :: xs => f(x).mappend(bind(xs, f))
    }
  }
}
def pure_outside[M[_]: Monad, A](x:A) = {
  val m = implicitly[Monad[M]]
  m.pure(x)
} // pure_outside(1), Error: Ambiguous given instance
def pure_option[A](x:A) = {
  val m = implicitly[Monad[Option]]
  m.pure(x)
} // pure_option(1) // Some(1)
```

Scalaz packages this and enables Monad functionality by importing:

```
import scalaz.Monad
import scalaz.Scalaz.optionInstance
val O = Monad[Option]
O.pure(1) // Some(1)
```

1.3.2 Specification for UPL extension

Where do would we add and change code? How would a specification look like for a higher-kinded type variable like `M[_]` look like?

1. Parser: need syntax for

```
type A
type M[A]
```

```
type M[_]
```

2. Type checker
3. Interpreter?

1.4 do notation

Want to implement *do notation* with left-arrows `<-` which is internally replaced with calls to the bind method.

To give some context and frame of reference we will first look at the Haskell do notation, Scala for expressions, Scalaz notation

These are the types for the following examples:

```
mx: m a
my: m b
mz: m (a -> b)
mw: m c
return: a -> m a
bind: m a -> (a -> m b) -> m b
```

where `m` is a monad type of kind `* -> *`.

First, have a look at the Haskell do notation

```
do
    x <- mx
    y <- my
    f <- mz
    mw
    return (f x y)
```

```
-- >>= is the infix bind operator
-- >>= :: m a -> (a -> m b) -> m b
mx >>= (\x ->
    my >>= (\y ->
    mz >>= (\f ->
    mw >>= (_ ->
    return (f x y)))))
```

Scala for expressions with lists do something similar
(cf. Haskell list comprehensions):

```
for {
      x <- List(1, 2)
      y <- List('a', 'b')
      f <- List((x:Int) => (y:Char) => (x,y))
      _ <- List("some side effect")
    } yield f(x)(y)
```

Scalaz:

Alternative do notations: no keyword/only {}-block, do keyword, return keyword outside

```
{
  x <- mx
  f <- mf
  return(f(x))
}
```

```
do {
   x <- mx
   f <- mf
   return(f(x))
}
```

```
do {
   x <- mx
   f <- mf
   //
} return f(x)
```

Internally `do notation` is replaced with calls to bind method.

```
do {
    x: a <- mx
    y: b <- my
    f: (a -> b) <- mz
    mw
    return(f(x,y))
}
=>
mx.bind(x ->
my.bind(y ->
mz.bind(f ->
mw.bind(_ ->
return(f(x,y))
```

Built-in do notation where each `do` keyword associated with exactly one left arrow:

```
// x: a, mx: m a, ...: mb
do x <- mx {
    ...
}
=>
mx.bind(x -> ...)
```

$\leftarrow$ *notation* as syntactic sugar for *do notation*, process of desugaring:

```
do {
    x <- mx
    y <- my
    ...
}
=>
do x <- mx {
    y <- my
    ...
}
=>
do x <- mx {
    do y <- my {
        ...
    }
}
```

1.5 Specification: built-in monads

1.6 Specification: do notation for built-in monads

Can make all collection types (option, list, set, bound, interval collections, ...) built-in monads. This means they have return and bind methods and can be used in do blocks.

1.6.1 Parsing

Part of AST:

```
case class Do(m: Expression, fun: Expression) extends
   Expression
case class Return(e: Expression) extends Expression
```

1.6.2 Type checking

Parts of type-checking:

```
getMonadObject(tp: Type) = tp match {
  case CollectionType(_, k) => k match {
     case CollectionKind.List => ...
     case CollectionKind.Option => ...
   }
   case _ => error("not a monadic type")
}
```

while doing type checking of Expression

```
checkExpression(gc: GlobalContext, exp: Expression, tp:
   Type) = exp match {

  case Do(m,f) =>
     val (mC,mI) = inferExpression(gc, m)
     val monad = getMonadObject(mI)
     checkExpression(gc, OwnedExpr(monad, Application(
        ClosedRef("bind"), List(mC,f)), tp)

  case Return(e) =>
    val monad = getMonadObject(tp)
    ...

}
```

1.6.3

1.7 Specification: user-defined monads

1.8 Specification: do notation for user-defined monad types

User defines a monad by following a certain template, e.g. defining return and bind or realizing Monad theory.

```
theory Monad {
    type A
    type B
    type M[_]: * -> *
    return: A -> M[A]
    bind:
        M[A] -> (A -> M[B])
            -> M[B]
}
```

```
OptionMonad = Monad {
    type M[_] = option[_]
    return =
        (x: A) -> option[x]
    bind =
        (ox: option[A]) ->
        (f: A -> option[B])
        -> {...}
}
```

Now we want that UPL automatically makes the do notation available for the new monad type.

With the above style of defining monads, we cannot go from do(o, f) where o: option[int] to o.bind(f) because the bind method is defined in OptionMonad