Type-Directed Language Extension for Effectful Computations

Evgeny Kotelnikov

Chalmers University of Technology

Scala 2014

July 28, Uppsala

Example: asynchronous computations

```
def F(): Int = slow computation

def G(x: Int): Int = slow computation


2 * G(F() + 1)
```

How to evaluate it asynchronously?

Example: asynchronous computations

```
def f(): Future[Int] = future { F() }

def g(x: Int): Future[Int] = future { G(x) }
```

Example: for-comprehension

```
def f(): Future[Int] = future { F() }
def g(x: Int): Future[Int] = future { G(x) }

for {
  x <- f()
  y <- g(x + 1)
} yield 2 * y
```

Example: async

```
def f(): Future[Int] = future { F() }
def g(x: Int): Future[Int] = future { G(x) }

import scala.async.Async._
async {
  2 * await(g(await(f()) + 1))
}
```

Example: scala-workflow

```
def f(): Future[Int] = future { F() }

def g(x: Int): Future[Int] = future { G(x) }


import scala.workflow._

workflow[Future] {
  2 * g(f() + 1)
}
```

Not only Future

```
def divide(x: Double, y: Double): Option[Double] =
  if (y == 0) None else Some(x / y)

workflow[Option] {
  divide(x + divide(1, y),
         divide(1, z) + t)
}
```

$$\frac{x + \frac{1}{y}}{\frac{1}{z} + t}$$

Not only monads

```
val xs: Stream[Int] = ...
val ys: Stream[Int] = ...
val zs: Stream[Int] = ...

workflow(zipStream) {
  xs + ys + zs
}
```

Related work

F# computation expressions

Special keywords + computation builders

```
let getLength url = async {
  let! html = fetchAsync url
  do! Async.Sleep 1000
  return html.Length
}
```

Syntax

Syntax of scala-workflow

```
import scala.workflow.{workflow, context, $}
```

```
workflow[F] {                    context[F] {
   ???                    =        $(???)
}                                }
```

Syntax of scala-workflow

```
import scala.workflow.{workflow, context, $}
```

```
workflow(obj) {
  
}
```

=

```
context(obj) {
  $( )
}
```

```
obj: Workflow[F]
```

Workflow

Stackable interface

```
trait Workflow[F[_]] {}
```

```
trait Pointing[F[_]]
trait Mapping[F[_]]
trait Applying[F[_]]
trait Binding[F[_]]
```

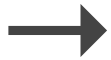
} `extends Workflow[F]`

Each trait enables a language feature within the $

Pointing

```
trait Pointing[F[_]] extends Workflow[F] {
  def point[A](a: => A): F[A]
}
```

```
workflow(option) {
  2 + 3
}
```

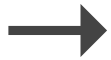


```
option.point(2 + 3)
```

Mapping

```
trait Mapping[F[_]] extends Workflow[F] {
  def map[A, B](f: A => B): F[A] => F[B]
}
```

```
workflow(option) {
  1 + divide(2, 3)
}
```



```
option.map(
  (x$1: Double) => 1 + x$1
)(divide(2, 3))
```

Applying

```
trait Applying[F[_]] extends Workflow[F] with Mapping[F] {
  def app[A, B](f: F[A => B]): F[A] => F[B]
}
```

```
workflow(option) {
  divide(1, 2) +
    divide(3, 4)
}
```

```
option.app(option.map(
  (x$1: Double) => (x$2: Double) =>
    x$1 + x$2
)(divide(1, 2)))(divide(3, 4))
```

Binding

```
trait Binding[F[_]] extends Workflow[F] {
  def bind[A, B](f: A => F[B]): F[A] => F[B]
}
```

```
workflow(option) {
  divide(divide(1, 2), 3)
}
```

```
option.bind(
  (x$1: Double) =>
    divide(x$1, 3)
)(divide(1, 2))
```

Aliases

```
trait Functor[F[_]] extends Mapping[F]

trait SemiIdiom[F[_]] extends Functor[F] with Applying[F]

trait Idiom[F[_]] extends SemiIdiom[F] with Pointing[F] {
  def map[A, B](f: A => B) = app(point(f))
}
```

Aliases

```
trait SemiMonad[F[_]] extends SemiIdiom[F] with Binding[F]


trait Monad[F[_]] extends Idiom[F] with Binding[F] {
  def app[A, B](f: F[A => B]) =
    bind(a => bind((g: A => B) => point(g(a)))(f))
}
```

Option workflow

```
val option = new Monad[Option] {
  def point[A](a: => A) = Option(a)
  def bind[A, B](f: A => Option[B]) = {
    case Some(a) => f(a)
    case None => None
  }
}
```

Rewriting

Expression rewriting

1. Traverse AST in post-order
2. Type check a node
3. If its type corresponds to the type of the workflow, rewrite with an effectful binding
4. Carry the scope of effectful bindings
5. Insert method calls based on the dependencies

Expression rewriting

```
workflow[Option] {
  2 * divide(divide(3, 4), 5)
}
```

Expression rewriting

```
workflow[Option] {
  2 * divide(divide(3, 4), 5)
}


2 : Int
```

Expression rewriting

```
workflow[Option] {
  2 * divide(divide(3, 4), 5)
}
```

```
3 : Int
```

Expression rewriting

```
workflow[Option] {
  2 * divide(divide(3, 4), 5)
}


4 : Int
```

Expression rewriting

```
workflow[Option] {
  2 * divide( divide (3, 4), 5)
}

divide : (Double, Double) => Option[Double]
```

Expression rewriting

```
workflow[Option] {
  2 * divide(divide(3, 4), 5)
}

divide(3, 4) : Option[Double]
```

Expression rewriting

```
workflow[Option] {
  2 * divide( divide(3, 4) , 5)
}

divide(3, 4) : Option[Double]
```

Expression rewriting

```
workflow[Option] {
  2 * divide( x$1 , 5)
}

x$1 : Double <- divide(3, 4)
```

Expression rewriting

```
workflow[Option] {
  2 * divide(x$1, 5)
}

5 : Int

x$1 : Double <- divide(3, 4)
```

Expression rewriting

```
workflow[Option] {
  2 * divide(x$1, 5)
}

divide : (Double, Double) => Option[Double]

x$1 : Double <- divide(3, 4)
```

Expression rewriting

```
workflow[Option] {
  2 * divide(x$1, 5)
}

divide(x$1, 5) : Option[Double]

x$1 : Double <- divide(3, 4)
```

Expression rewriting

```
workflow[Option] {
  2 * divide(x$1, 5)
}

divide(x$1, 5) : Option[Double]

x$1 : Double <- divide(3, 4)
```

Expression rewriting

```
workflow[Option] {
  2 * x$2
}

x$2 : Double <- divide(x$1, 5)

x$1 : Double <- divide(3, 4)
```

Expression rewriting

```
workflow[Option] {
  2 * x$2
}

* : (Double, Double) => Double (or something)

x$2 : Double <- divide(x$1, 5)
x$1 : Double <- divide(3, 4)
```

Expression rewriting

```
workflow[Option] {
  2 * x$2
}

2 * x$2 : Double

x$2 : Double <- divide(x$1, 5)
x$1 : Double <- divide(3, 4)
```

Expression rewriting

```
option.map(
  (x$2: Double) => 2 * x$2
)(          )
```

```
x$2 : Double <- divide(x$1, 5)
x$1 : Double <- divide(3, 4)
```

Expression rewriting

```
option.map(
  (x$2: Double) => 2 * x$2
)(option.bind(
  (x$1: Double) => divide(x$1, 5)
)(      ))

x$1 : Double <- divide(3, 4)
```

Expression rewriting

```
option.map(
  (x$2: Double) => 2 * x$2
)(option.bind(
  (x$1: Double) => divide(x$1, 5)
)(divide(3, 4)))
```

Examples

Example: evaluator of expressions

```
sealed trait Expr

case class Var(id: String) extends Expr

case class Val(value: Int) extends Expr

case class Add(lhs: Expr, rhs: Expr) extends Expr


type Env = Map[String, Int]

def lookup(id: String)(env: Env) = env.get(id)
```

Example: evaluator of expressions

```
def eval(expr: Expr)(env: Env): Option[Int] =
  expr match {
    case Var(x) => lookup(x)(env)
    case Val(n) => Some(n)
    case Add(x, y) => for { lhs <- eval(x)(env)
                            rhs <- eval(y)(env) }
                      yield lhs + rhs
}
```

Example: evaluator of expressions

```
def eval(expr: Expr)(env: Env): Option[Int] =
  context(option) {
    expr match {
      case Var(x) => lookup(x)(env)
      case Val(n) => $(n)
      case Add(x, y) => $(eval(x)(env) + eval(y)(env))
    }
  }
```

Example: evaluator of expressions

```
def eval: Expr => Env => Option[Int] =
  context(option) {
    case Var(x) => (env: Env) => lookup(x)(env)
    case Val(n) => (env: Env) => $(n)
    case Add(x, y) =>
      (env: Env) => $(eval(x)(env) + eval(y)(env))
  }
```

Example: evaluator of expressions

```
def eval: Expr => Env => Option[Int] =
  context(function[Env] $ option) { // Env => Option[_]
    case Var(x) => lookup(x)
    case Val(n) => $(n)
    case Add(x, y) => $(eval(x) + eval(y))
  }
```

Example: exception handling

```
def fetchXML(address: String): XML = {
  val url: URL = URL.fromString(address)
  val page: Page = Page.fetch(url)
  val contents: String = page.getContents
  XML.fromString(contents)
}
```

Example: exception handling

```
def fetchXML(address: String): Try[XML] =
  for {
    url <- URL.fromString(address)
    page <- Page.fetch(url)
    contents = page.getContents
    xml <- XML.fromString(contents)
  } yield xml
```

Example: exception handling

```
def fetchXML(address: String): Try[XML] =
  workflow[Try] {
    val url = URL.fromString(address)
    val page = Page.fetch(url)
    val contents = page.getContents
    XML.fromString(contents)
  }
```

Conclusions

Strenghts

- Effectful computations in direct style
- Uniform syntax for a hierarchy of computation types
- Implemented as a macro-based library
- Use cases:
    1. Incapsulation of the treatment of effects
    2. Syntactic support for DSLs
    3. Boilerplate elimination

Drawbacks

- Ill-typed code inside the brackets
- Not a full set of Scala is supported
- Untyped macros

Future work

Macro annotations to the rescue!

```
@workflow[Future] val x = 2 * g(f + 1)


@context(function[Env] $ option)
def eval: Expr => Env => Option[Int] = {
  case Var(x) => fetch(x)
  case Val(n) => $(n)
  case Add(x, y) => $(eval(x) + eval(y))
}
```

github.com/aztek/scala-workflow

```
scala> workflow[Option] { "2" * divide(divide(3, 4), 5) }
<console>:12: error: type mismatch;
 found   : Double
 required: Int

   "2".$times(x$2)

 where

   x$1: Double <- divide(3, 4)
   x$2: Double <- divide(x$1, 5)

Type error during rewriting of expression within Option context
             workflow[Option] {
                              ^
```