ScalaSyd

Episode 60 (8 November 2017)

Conditional contexts

Mark Hopkins

CBA

@antiselfdual | mjhopkins | mjhopkins.github.io

Problem

I've got a generic (polymorphic) method

```
def method[A](a: A)
```

but I want to do something different
when `A` has some *additional property*.

Additional property = instance of some type class

Examples

numeric stability

```
def sum[N](l: List[N])(implicit num: Numeric[N]): N =
  l.foldLeft(num.zero)(num.plus)
```

Doesn't have behave well for floating point numbers (numerical stability issues).

Want to us a different algorithm when `N` represents a floating point number

e.g. the *Kahan summation algorithm*

distinct elements in a list

if all you can do is test for equality, $O(n^2)$
if you can sort, $O(n \log n)$

Well actually...

- linking two events
- query planning based on presence or absence of an index
- different kinds of printing
  e.g. mask out parts of sensitive data.
- ...

If the *different property* was encoded as a super-class,

we could just **type-test**:

```
def method[A](a: A) = a match {
  case x: SomeClass => doSomethingSpecial(x)
  case _            => defaultCase(a)
}
```

What's the type class equivalent to this?

N.B. Good reasons for encoding as type classes rather than inheritance.

- don't need to change original code
- flexible
- can make different choices in different contexts
- datatype generic programming
  e.g. instances automatically created for classes
  with certain structure

I can "test" if an instance exists

```
implicitly[SomeTypeClass[A]]
```

```
def implicitly[T](implicit t: T): T = t
```

But this fails at compile-time if `A` doesn't have an instance for `SomeTypeClass`.

We want to move this to a (safe) run-time check instead.

What we want

replace

```
implicitly[T]: T
```

with

```
maybeImplicitly[T]: Option[T]
```

Workaround

Two separate methods

```
// N.B. Don't use for floating point!!!!! Bad!!1!
def sum[N: Numeric]
// N.B. Don't use for Int - too slow!
def sumFP[N: Fractional]
```

- fragile
- original problem still exists - still have to choose

Haskell! ♥♥♥♥♥♥

A general purpose pure functional programming language.

https://www.haskell.org/

Mike Izbicki has written a package called `ifcxt`.

> constraint level if statements

> put if statements within type constraints

> `ifcxt` proposes an extension to Haskell's type system to allow constraints to include if statements.
> This lets us write faster, more generic code

https://hackage.haskell.org/package/ifcxt

Constraints and type classes

```
class Show a where
  show :: a -> String
```

```
λ> :kind Show
Show :: * -> Constraint

λ> :kind Show Int
Show Int :: Constraint

λ> :t show
show :: Show a => a -> String

λ> show 4
"4"
```

```
class IfCxt (cxt :: Constraint) where
  ifCxt :: proxy cxt -> (cxt => a) -> a -> a
                |                |        |    |
specify cxt---|                |        |    |
                                 |        |    |
                                 |        |    |
           fn to use if y ---|        |    |
                                 |        |    |
                                 |        |    |
                 fn to use if n -----|    |
                                          |
                                          |
                                result -----|
```

Usage:

```haskell
cxtSum :: forall n. (Num n, IfCxt (Floating n)) => [n] -> n
cxtSum = ifCxt p sumKahan sum
 where p = Proxy :: Proxy (Floating n)
```

```
{-# LANGUAGE ConstraintKinds   #-}
{-# LANGUAGE FlexibleContexts  #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE Rank2Types        #-}
{-# LANGUAGE TemplateHaskell   #-}
```

Implementation

Template Haskell!

First of all, user has to explicitly ask for `IfCxt` instances for the type classes they want

```
mkIfCxtInstances ''Ord
mkIfCxtInstances ''Show
mkIfCxtInstances ''Typeable
```

There's an overlappable instance that always says "no"

```
instance {-# OVERLAPPABLE #-} IfCxt cxt where ifCxt _ t f = f
```

TH will produce the overlapping "yes" instances as relevant.

Pseudocode of simplified algorithm

```
mkIfCxtInstances n:
  if n not a type class, abort.
  for each instance i of n:
    if there's already an instance of IfCxt (n i), do nothing.
    else make a new one.

THE END.
```

Complications:

- instances may have their own contexts (preconditions).
- could be recursive.

e.g. from

```
Show a => Show [a]
```

we should be able to derive

```
IfCxt (Show a) => IfCxt(Show [a])
```

Code

https://github.com/mikeizbicki/ifcxt

Let's port this to Scala

Template Haskell

$\Longrightarrow$ Scala macros

Explicitly asking for instances to be created

$\Longrightarrow$ implicit materializer

Complications

$\Longrightarrow$ *do they even make sense for Scala??*

out-of-scope for now

Code

https://github.com/mjhopkins/ifcxt-scala

See IfCxtMacros.scala and IfCxt.scala.

STOP!

Could there be an easier way?*

After all, "constraints" are just values in Scala...

* Thanks to Laurence Rouesnel for pointing this out!

Version 2

```
trait IfCxt[T] {
  def ifCxt[A](t: T => A, f: A): A =
    maybeCxt map t getOrElse f
  def maybeCxt: Option[T]
}

object IfCxt {
  implicit def instance[T](implicit c: T = null): IfCxt[T] =
    new IfCxt[T] {
      def maybeCxt = Option(c)
    }
}
```

This works! 😀

"Nulls? default parameters? Urgh!"

Version 3

```
sealed trait IfCxt[T] {
  def ifCxt[A](t: T => A, f: A): A =
    maybeCxt map t getOrElse f
  def maybeCxt: Option[T]
}

trait LowPriorityIfCxt {
  implicit def absent[T] =
    new IfCxt[T] {
      def maybeCxt = None
    }
}

object IfCxt extends LowPriorityIfCxt {
  implicit def present[T](implicit c: T) =
    new IfCxt[T] {
      def maybeCxt = Some(c)
    }
}
```

This works too. 😄

Hang on...

Can we make this even simpler?

```
def maybeImplicitly[T](implicit t: T = null) = Option(t)
```

```
scala> maybeImplicitly[Show[Int]]
          map (_ show 3) getOrElse "Can't show"
res0: String = 3

scala> maybeImplicitly[Show[Int => Int]]
          map (_ show (_ + 1)) getOrElse "Can't show"
res1: String = Can't show
```

This **doesn't** work

```
def maybeImplicitly[T](implicit t: T = null) = Option(t)


def show[A](a: A) =
  maybeImplicitly[Show[A]] map (_ show a) getOrElse "Can't show
```

`show` will always return `"Can't show"`.

Summary

We've successfully lifted `Option` to the type class level.

Limitations

The Haskell `ifcxt` package can handle instances with contexts

e.g. deriving

`IfCxt (Show a) => IfCxt(Show [a])` from `Show a => Show [a]`

Not done in Scala.

Scala vs Haskell

It seems in this case that Scala has some advantages over Haskell

END