Programs as Data

The Scala language: kinds, monads, implicits

Peter Sestoft

2012-11-19

Agenda

- More Scala
  - More about for-yield expressions
  - Higher-kinded types: functions on types
  - Monads
  - Implicit arguments

- Other new languages
  - Go, Clojure, Ceylon, Dart, ...
  - Coffeescript, Typescript, ...

Map, filter and flatMap on lists

- `[x1...xn] map f` equals `[f(x1)...f(xn)]`
- `[x1...xn] filter p` equals subsequence `[xi1...xim]` where `p(xij)==true`
- `[x1...xn] flatMap f` equals `[y11...y1m ... yn1...ynk]` where `[yi1...yim] = f(xi)`
- Exactly the Select, Where, SelectMany extension methods on .NET's IEnumerable<T> interface
- And `xs flatMap f == (xs map f).flatten`

More on for-expressions

```
for (x <- primes; if x*x < 100) yield 3*x
```

- A for-yield expression is translated into
  - `map`
  - `filter`
  - `flatMap`

```
primes
    filter (x => x*x < 100)
    map (x => 3*x)
```

Translation of for-yield, part 1

```
for (x <- e1) yield er
```

```
e1 map (x => er)
```

```
for (x <- e1; if p) yield er
```

```
for (x <- e1 filter (x => p)) yield er
```

```
for (x <- e1; if p; more) yield er
```

```
for (x <- e1 filter (x => p); more) yield er
```

Translation of for-yield, part 2

```
for (x <- e1; y <- e2; more) yield er
```

```
e1 flatMap
   (x => for (y <- f(x); more) yield er)
```

```
for ((x1,x2) <- e1) yield er
```

```
e1 map { case (x1, x2) => er }
```

```
for (pat <- e1) yield er
```

```
e1 filter { ... } map { case pat => er }
```

For-yield works on list, set, option ...

- In fact, for-yield works on anything that supports
  - map
  - filter
  - flatMap
- Provided the types are correct, like so:

```
abstract class C[A] {
  def map[B](f: A => B): C[B]
  def filter(p: A => Boolean): C[A]
  def flatMap[B](f: A => C[B]): C[B]
  def foreach(b: A => Unit): Unit
}
```

Think of C[A] as a collection of A's

Higher-kinded types

- A *type* represents many similar values
  - Eg: Int represents 0, 1, 2, ..., -1, -2, ...
  - Eg: List[Int] represents List(2,3), List(13,21), ...
- A *kind* represents many similar types
  - Eg: * represents Int, String, Double, ...
  - Eg: * -> * represents List[_], Tree[_], ...

- Sounds cool, but why bother?
- Because even generic types lead to some code duplication (and maintenance trouble)

Motivation: filter and remove

Examples due to David Christiansen

```
trait Iterable[T] {
  def filter(p: T => Boolean): Iterable[T]
  def remove(p: T => Boolean): Iterable[T]
      = this.filter(x => !p(x))
}

trait List[T] extends Iterable[T] {
  def filter(p: T => Boolean): List[T] = ...
  override def remove(p: T => Boolean): List[T]
      = this.filter(x => !p(x))
}
```

Neat enough

Not neat: silly to have to redefine

- If remove were just inherited, it would have result type Iterable[T] not List[T]
- But the result of removing from a list should be a list, not an iterable

Type constructors to the rescue

- A *type constructor* is a function from types to a type. A generic class is a type constructor.
- Idea: Allow type constructors as arguments to types.

Takes a type and returns a type

```
trait Iterable[T, Result[_]] {
  def filter(p: T => Boolean): Result[T]
  def remove(p: T => Boolean): Result[T]
      = this.filter(x => !p(x))
}
trait List[T] extends Iterable[T, List] {
  def filter(p: T => Boolean): List[T] = ...
}
```

The List type constructor

- In List[T] the inherited **`remove`** has return type Result[T] = List[T] as desired
- No need to redefine **`remove`**

The need for kinds

- Some uses of Iterable would be meaningless: Iterable[Int, Double]
- Double is a type (*), not type function (*->*)
- We can use *kinds* to describe legal uses of List, Iterable and so on:
  - Int: *
  - List: * -> *
  - List[Int]: *
  - Result: * -> *
  - Iterable: * -> (* -> *) -> *
- See Moors, Piessens, Odersky: *Generics of a higher kind*, OOPSLA2008

Extensions of the eval function

- Consider (again) very simple expressions:

File ExpressionsMonads.scala

```
sealed abstract class Expr
case class CstI(value: Int)
     extends Expr
case class Prim(op: String, e1: Expr, e2: Expr)
     extends Expr
```

- We shall make a series of evaluators:

```
eval:       Expr -> Int                    standard
optionEval: Expr -> Option[Int]            may fail
setEval:    Expr -> Set[Int]               nondeterministic
traceEval:  Expr -> (List[String], Int)    tracing operators
```

- And use *monads* to make them look alike

First, split the eval function

```
def eval1(e: Expr): Int = {
  e match {
    case CstI(i)            => i
    case Prim(op, e1, e2) =>
      val v1 = eval1(e1)
      val v2 = eval1(e2)
      op match {
        case "+" => v1 + v2
        case "*" => v1 * v2
        case "/" => v1 / v2
      }
    }
}
```

```
def eval2(e: Expr): Int = {
  e match {
    case CstI(i)            => i
    case Prim(op, e1, e2) =>
      val v1 = eval2(e1)
      val v2 = eval2(e2)
      opEval(op, v1, v2)
    }
}
```

7+9*10

97

```
def opEval(op: String, v1: Int, v2: Int): Int =
  op match {
    case "+" => v1 + v2
    case "*" => v1 * v2
    case "/" => v1 / v2
  }
```

Expressions that may fail: Option

```
def optionEval1(e: Expr): Option[Int] = {
  e match {
    case CstI(i)           => Some(i)
    case Prim(op, e1, e2) =>
      optionEval1(e1) match {
        case None      => None
        case Some(v1) =>
          optionEval1(e2) match {
            case None      => None
            case Some(v2) => opEvalOpt(op, v1, v2)
          }
} } }
```

```
7+9*10
```

```
Some(97)
```

```
def opEvalOpt(op: String, v1: Int, v2: Int): Option[Int] =
  op match {
    case "+" => Some(v1 + v2)
    case "*" => Some(v1 * v2)
    case "/" => if (v2==0) None else Some(v1/v2)
  }
```

Expressions that may have multiple results: Set

```
def setEval1(e: Expr): Set[Int] = {
  e match {
    case CstI(i)            => Set(i)
    case Prim(op, e1, e2) =>
      (setEval1(e1) map { (v1: Int) =>
        (setEval1(e2) map { (v2: Int) => opEvalSet(op, v1, v2)
          }) . flatten
      }) . flatten
  }
}
```

```
7+choose(9,10)
```

```
Set(16, 17)
```

```
def opEvalSet(op: String, v1: Int, v2: Int): Set[Int] =
  op match {
    case "+" => Set(v1 + v2)
    case "*" => Set(v1 * v2)
    case "/" => if (v2==0) Set() else Set(v1 / v2)
    case "choose" => Set(v1, v2)
  }
```

Expressions with tracing of operators: Trace

```
def traceEval1(e: Expr): Trace[Int] = {
  e match {
    case CstI(i)            => (List(), i)
    case Prim(op, e1, e2) => {
      val (trace1, v1) = traceEval1(e1)
      val (trace2, v2) = traceEval1(e2)
      val (traceOp, valOp) = opEvalTrace(op, v1, v2)
      (trace1 ++ trace2 ++ traceOp, valOp)
    }
  }
}
```

```
type Trace[A] = (List[String], A)
```

```
7+9*10
```

```
(List(*, +),97)
```

```
def opEvalTrace(op: String, v1: Int, v2: Int): Trace[Int] =
  op match {
    case "+" => (List(op), v1 + v2)
    case "*" => (List(op), v1 * v2)
    case "/" => (List(op), v1 / v2)
  }
```

Commonality of the four evaluators

- To evaluate an integer constant CstI(n)
  - Convert n to the result type:
    Int, Option[Int], Set[Int], Trace[Int]
- To evaluate a two-operand Prim(op,e1,e2)
  - evaluate e1 to get some result(s)
  - build on them, evaluate e2 to get some result(s)
  - build on the combination of results, using an opEval, to produce final result(s)

Monads

- A monad consists of
  - A type constructor **`M[_]`**
  - A **`unit`** function: **`A -> M[A]`**
  - A **`bind`** function: **`M[A] -> (A -> M[B]) -> M[B]`**
- It must satisfy:
  - **`bind(unit(a),f) = f(a)`**
  - **`bind(a, unit) = a`**
  - **`bind(bind(a, f), g) = bind(a, x -> bind(f(x), g))`**
- Example, a set is a monad, in mathematics:
  - M[A] = all sets of A's
  - unit(x) = { x }
  - bind(xs, f) = U { f(x) | x ε xs }
- Example, a list is a monad, in Scala (or F#):
  - M[t] = List[t]
  - unit(x) = List(x)
  - bind(xs, f) = (xs map f).flatten = xs flatMap f

Monads in Scala

- A monad consists of
  - A type constructor **`M[_]`**
  - A **`unit`** function: **`A -> M[A]`**
  - A **`bind`** function: **`M[A] -> (A -> M[B]) -> M[B]`**

```
trait Monad[M[_]] {
  def unit[A](a: A): M[A]
  def bind[A,B](m: M[A])(f: A => M[B]): M[B]
}
```

- The Monad trait is generic of higher kind:

  Monad: (* -> *) -> *
- The argument to Monad is a type constructor
  Such as Option, Set, List, ...

The option monad

```
object OptionMonad extends Monad[Option] {
  def unit[A](a: A) = Some(a)
  def bind[A,B](a: Option[A])(fb: A => Option[B]) =
    a match {
      case None    => None
      case Some(v) => fb(v)
    }
}
```

- In Scala, **`bind`** on **`Option`** is called **`flatMap`**
- Useful in for-comprehensions

The set monad

```
object SetMonad extends Monad[Set] {
  def unit[A](a: A) = Set(a)
  def bind[A,B](a: Set[A])(fb: A => Set[B]) =
    (a map fb) . flatten
}
```

- In Scala, **`bind`** on **`Set`** is called **`flatMap`**
- Useful in for-comprehensions

The trace monad

```
type Trace[A] = (List[String], A)

object TraceMonad extends Monad[Trace] {
  def unit[A](a: A) = (List(), a)
  def bind[A,B](a: Trace[A])(fb: A => Trace[B]) = {
    val (aTrace, aVal) = a
    val (fTrace, fVal) = fb(aVal)
    (aTrace ++ fTrace, fVal)
  }
}
```

The identity monad

```
type Identity[A] = A

implicit object IdentityMonad extends Monad[Identity] {
  def unit[A](a: A) = a
  def bind[A,B](a: Identity[A])(fb: A => Identity[B]) =
    fb(a)
}
```

- Why, would you think, do we need this?

A general, monadic evaluator

```
def monadEval[M[_]](e: Expr)(monad: Monad[M],
     opEval: (String,Int,Int) => M[Int]): M[Int] = {
  e match {
    case CstI(i)           => monad.unit(i)
    case Prim(op, e1, e2) =>
      monad.bind(monadEval(e1)(monad, opEval)) { v1 =>
        monad.bind(monadEval(e2)(monad, opEval)) { v2 =>
          opEval(op, v1, v2) }
      }
  }
}
```

- Can now define all four evaluators:

```
def eval4(e: Expr)       = monadEval[Identity](e)(IdentityMonad, opEval)

def optionEval4(e: Expr) = monadEval(e)(OptionMonad, opEvalOpt)

def setEval4(e: Expr)    = monadEval(e)(SetMonad, opEvalSet)

def traceEval4(e: Expr)  = monadEval(e)(TraceMonad, opEvalTrace)
```

Implicit arguments

- The Scala compiler infers type parameters
  - Eg it infers the type parameters [Option], [Set] and [Trace] in the previous three examples
  - Based on the types of ordinary parameters
- The Scala compiler can also do the opposite
  - Infer an ordinary parameter ...
  - Based on type parameters or ordinary parameters

- To do this
  - Declare the formal parameter to be **`implicit`**
  - Declare the inferrable objects to be **`implicit`**

Implicits in the monadic evaluators

```
def monadEval[M[_]](e: Expr)(implicit monad: Monad[M],
     implicit opEval: (String,Int,Int)=> M[Int]): M[Int] =
{ ... }
```

```
implicit object IdentityMonad extends Monad[Identity] { ... }
implicit object OptionMonad extends Monad[Option] { ... }
implicit object SetMonad extends Monad[Set] { ... }
implicit object TraceMonad extends Monad[Trace] { ... }
```

```
implicit def opEval(op: String, v1: Int, v2: Int): Int = ...
implicit def opEvalOpt(op: String, v1: Int, v2: Int): Option[Int] ...
implicit def opEvalSet(op: String, v1: Int, v2: Int): Set[Int] = ...
implicit def opEvalTrace(op: String, v1: Int, v2: Int): Trace[Int] ..
```

Implicits in the monadic evaluators

- Now even neater evaluator definitions

```
def eval5(e: Expr)       = monadEval[Identity](e)

def optionEval5(e: Expr) = monadEval[Option](e)

def setEval5(e: Expr)    = monadEval[Set](e)

def traceEval5(e: Expr)  = monadEval[Trace](e)
```

- Given the evaluator's expected result type, the Scala compiler can infer:
  - The monad to use (optionMonad, setMonad, ...)
  - The operator evaluator (optionEval, setEval, ...)
- The Haskell language has very similar ideas
  - Allows for concise and very general programs

New (?) languages all the time

Features of some prominent languages					sestoft@itu.dk 2012-09-20								
	Lisp	**C**	**C++**	**ML/Caml**	**Java**	**C#**	**Scala**	**F#**	**Clojure**	**Go**	**Ceylon**	**Dart**	**Objective C**
First created	1958	1972	1984	1978	1994	1999	2003	2009	2009	2009	2011	2011	1986
Chief designers	McCarthy	Kernighan, Rit	Stroustrup	Milner	Arnold, Goslin	Hejlsberg	Odersky	Syme	Rich Hickey	Pike	King	Bracha, Bak	Cox
Company	MIT	Bell Labs	Bell Labs	Edinburgh U	Sun (Oracle)	Microsoft	EPFL (universi	Microsoft	(OSS?)	Google	Redhat	Google	Nextstep (App
Runtime	Own	(None)	(None)	Own	JVM	.NET/CLI	JVM	.NET/CLI	JVM	Own	JVM, Javascrip	Javascript	Own
Garbage collection	Yes	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	(Yes)
References are safe	Yes	No	No	Yes	No	No	No	No	Yes?	No	Yes	(exercise)	(exercise)
Object-oriented	No	No	Yes	No	Yes	Yes	Yes	(Yes)	Yes	Yes	Yes	(exercise)	(exercise)
Method overloading	-	-	Yes	-	Yes	Yes	Yes	Yes	No	No	No	(exercise)	(exercise)
Virtual methods	-	-	Yes	-	Yes	Yes	Yes	Yes	Yes	Yes?	Yes	(exercise)	(exercise)
Inheritance	-	-	Multiple	-	Single	Single	Single	Single	Single	No?	Single	(exercise)	(exercise)
Interfaces/traits	-	-	No	-	Interfaces	Interfaces	Traits	?	Protocols	Interfaces	Traits (called i	(exercise)	(exercise)
Functional	(Yes)	No	No	Yes	No	Yes	Yes	Yes	Yes	No	Yes	(exercise)	(exercise)
Closures capture	-	-	-	rvalue	rvalue	lvalue	lvalue	lvalue	lvalue?	-	lvalue?	(exercise)	(exercise)
Static types	No	(Yes)	Yes	Yes	Yes	Yes	Yes	Yes	Optional	Yes	Yes	(exercise)	(exercise)
Generic types, parametric	No	No	(Yes)	Yes	Yes	Yes	Yes	Yes	-	No	Yes	(exercise)	(exercise)
Type parameter bounds	-	-	No	No	Yes	Yes	Yes	Yes	-	-	Yes	(exercise)	(exercise)
Co/contravariance	-	-	No	No	No	Yes	Yes	No	-	-	Yes	(exercise)	(exercise)
Higher-kinded types	-	-	No	No	No	No	Yes	No	-	out	No	(exercise)	(exercise)
Pattern matching	No	No	No	Yes	No	No	Yes	Yes	No	No	(Yes)	(exercise)	(exercise)
Dynamic typing	(Yes)	No	No	No	No	Yes	Yes	No	Yes	No?	No	(exercise)	(exercise)
Exceptions	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	(exercise)	(exercise)
Checked exceptions	-	-	No	No	Yes	No	No	No	No	-	No	(exercise)	(exercise)
Reflection	No	No	No	No	(Yes)	Yes	Yes	(Yes)	(Not quite)	Yes	Yes	(exercise)	(exercise)
Concurrency	No	OS processes	OS processes	OS processes	Threads	Threads, asyn	Actors	Async	Transactional r	"Goroutines"	Threads??	(exercise)	(exercise)
Comm/sync	No	(Library)	(Library)	(Library)	Locks	Locks, await	Messages	Messages	Atomic sectior	Channels	Locks??	(exercise)	(exercise)

File languagecomparison.xls

What's next

- Monday 26 November:
  - Partial evaluation: Automatic program specialization, in the Scheme language
- No real exercises for Monday 26 November
- Voluntary exercise:
  - Complete the Dart and Objective-C columns
  - Possibly add another language of your choice

- Spørgetime: Onsdag 2. januar 2013 kl 0900
  - Room unknown at this time
- Eksamen: Torsdag 3. januar 2013