Programs as Data

The Scala language: kinds, implicits, monads, GADTs

Peter Sestoft

2011-11-21

Agenda

- More Scala
  - Higher-kinded types: functions on types
  - Monads, introduced
  - Implicit arguments
  - Generalized algebraic data types
- Other new languages
  - Go
  - Clojure
  - Ceylon
  - Dart

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 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 SimpleExpressions.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]               nondetermistic
traceEval:  Expr -> (List[String], Int)    operator trace
```

- 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)
    }
  }
}
```

```
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: Option, Set, Trace ...
- 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, 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 F# (or Scala):
  - M[t] = t list
  - unit(x) = [x]
  - bind(xs, f) = List.concat (List.map f xs)

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 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, do 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, ...)
- Very similar ideas in the Haskell language

New languages?

Features of some prominent languages			ITU BPRD 2011, sestoft@itu.dk 2011-11-20								
	C	**C++**	**Java**	**C#**	**Scala**	**F#**	**Clojure**	**Go**	**Ceylon**	**Dart**	**Objective C**
First created	1972	1984	1994	1999	2000?	2009	2009	2009	2011	2011	?
Chief designers	Kernighan, Rit	Stroustrup	Arnold, Goslin	Hejlsberg	Odersky	Syme	?	Pike	King	Bracha, Bak	?
Company	Bell Labs	Bell Labs	Sun (Oracle)	Microsoft	EPFL	Microsoft	?	Google	Redhat	Google	(Apple)
Runtime	(None)	(None)	JVM	.NET/CLI	JVM	.NET/CLI	JVM	Own	JVM	Javascript	Own
Garbage collection	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	(Yes)
Object-oriented	No	Yes	Yes	Yes	Yes	(Yes)	Yes	Yes	Yes	(exercise)	?
Method overloading	-	Yes	Yes	Yes	Yes	Yes	No	No	No	(exercise)	?
Virtual methods	-	Yes	Yes	Yes	Yes	Yes	?	Yes?	Yes	(exercise)	?
Inheritance	-	Multiple	Single	Single	Single	Single	Single	No?		(exercise)	?
Interfaces/traits	-	No	Interfaces	Interfaces	Traits	?	?	Interfaces	Traits (called i	(exercise)	?
Functional	No	No	No	(Yes)	Yes	Yes	Yes	No	Yes?	(exercise)	?
Static typing	(Yes)	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	(exercise)	?
Generic types, parame	No	(Yes)	Yes	Yes	Yes	Yes	-	No	Yes	(exercise)	?
Type parameter bounds	-	No	Yes	Yes	Yes	Yes	-	-	Yes	(exercise)	?
Co/contravariance declar	-	No	No	Yes	Yes	No	-	-	Yes	(exercise)	?
Higher-kinded types	-	No	No	No	Yes	No	-	-	?	(exercise)	?
Pattern matching	No	No	No	No	Yes	Yes	No	No	(Not quite)	(exercise)	?
Dynamic typing	No	No	No	Yes	No	No	Yes	No?	No	(exercise)	?
Exceptions	No	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	(exercise)	?
Checked exceptions	-	No	Yes	No	No	No	No	-	No	(exercise)	?
Reflection	No	No	Yes/No (not fo	Yes	Yes?	Yes?	(Not quite)	Yes	Yes	(exercise)	?
Concurrency	OS processes	OS processes	Threads	Threads, asyn	Actors	Async	Transactional i	"Goroutines"	Threads??	(exercise)	?
Synchronization	(Library)	(Library)	Locks	Locks, await	Messages	Messages	Atomic sectior	Channels	Locks??	(exercise)	?

What's next

- Monday 28 November: trial exam
- Spørgetime: Torsdag 5. januar 2012 kl 1000
- Eksamen: Mandag 9. januar 2012 kl 1300