Exercises week 11
Monday 12 November 2012

2012-11-12

Goal of the exercises

The goal of this week's exercises is to get acquainted with some core features of the Scala programming language.

Do exercises 11.1, 11.2, 11.3, 11.4, and 11.5. Hand in the solutions.

Do this first

Download Scala from `http://www.scala-lang.org/downloads` and unpack it for your platform (Windows, MacOS, Linux). To run Scala, you must have a Java Virtual Machine installed too.

Exercise 11.1 Recall the very simple expression language shown in the lecture:

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

def eval0(expr: Expr): Int =
  expr match {
    case CstI(i)          => i
    case Prim(op, e1, e2) =>
      val v1 = eval0(e1)
      val v2 = eval0(e2)
      op match {
        case "+" => v1 + v2
        case "-" => v1 - v2
        case "*" => v1 * v2
      }
  }
```

(i) Extend the evaluator `eval0` with new cases to handle also operators for division (`/`), less than (`<`), greater than (`>`) and logical and (`&`). In the evaluator, use 1 to represent true and 0 to represent false. Try the new evaluator on several examples, for instance `7 * (10 - 6 + 5)` and `7 * (10 < 6 + 5)`.

(ii) Extend the abstract syntax for expressions with a new case class `Var` to represent named variables. Extend the evaluator `eval0` to create a new evaluator `eval1(expr: Expr, env: Map[String,Int]): Int` that can evaluate expressions involving variables. The variable environment `env` is represented by an instance of Scala's immutable Map class.

An environment that maps variable `x` to 27 and `y` to 42 can be written `Map("x" -> 27, "y" -> 42)`.

Exercise 11.2 Write a Scala function `simplify(expr: Expr): Expr` to perform expression simplification on the extended `Expr` type from exercise 11.1. For instance, it should simplify $(x + 0)$ to x, and simplify $(1 + 0)$ to 1. The more ambitious student may want to simplify $(1 + 0) * (x + 0)$ to x. Hint 1: Pattern matching is your friend. Hint 2: Don't forget the case where you cannot simplify anything.

You might consider the following simplifications, plus any others you find useful and correct. The last one below is trickier than the others:

$0 + e$	$\longrightarrow$	e
$e + 0$	$\longrightarrow$	e
$e - 0$	$\longrightarrow$	e
$1 * e$	$\longrightarrow$	e
$e * 1$	$\longrightarrow$	e
$0 * e$	$\longrightarrow$	0
$e * 0$	$\longrightarrow$	0
$e - e$	$\longrightarrow$	0

Exercise 11.3 Extend the evaluator `eval1` from exercise 11.1 to create a new evaluator `eval2(expr: Expr, env: Map[String,Int]): Option[Int]` that returns `Some(r)` if the evaluation succeeds with result `r`, and returns `None` if the evaluation fails.

The evaluation may fail because it divides by zero or uses a variable that is not in the environment `env`.

As in Java, you can check whether `x` is a key in the map `env` by evaluating `env.contains(x)`, or more Scala-ish, `env contains x`. Alternatively, calling `env.get(x)` returns `Some(v)` if `x` maps to `v` in `env`, or `None` if `x` is not contained in `env`.

Hint: Use pattern matching on the results returned by the recursive calls to `eval2`.

Exercise 11.4 In exercise 11.3 you end up with nested pattern matching on the results of `eval2`. Create a new evaluator `eval3(expr: Expr, env: Map[String,Int]): Option[Int]` that works exactly as `eval2` but uses Scala's `for` expressions instead of pattern matching on `None` and `Some`.

Hint 1: When `v1` and `v2` have type `Option[Int]`, then

```
for (i1 <- v1;
     i2 <- v2)
yield i1+i2
```

has the same meaning as the more verbose

```
v1 match {
  case None     => None
  case Some(i1) => {
    v2 match {
      case None     => None
      case Some(i2) => Some(i1+i2)
    }
  }
}
```

If `v1` has value `None`, the `for` expression has value `None`; and if `v1` has value `Some(17)`, then `i1` will be bound to 17, and the next clauses in the `for` expression will be evaluated in a similar manner.

Hint 2: Your solution will probably need three bindings inside the `for` expression.

Exercise 11.5 Let us return to the simplest expression language at the beginning of exercise 11.1 (i). That language is untyped, because it is possible to write expressions corresponding to `7 * (2 < 4)` which mixes numbers and truth values in an unhealthy way.

In Scala, it is possible for a subclass to inherit from a superclass that has one or more type parameters instantiated. Using this, one can define the abstract syntax so that it can represent only strongly typed expressions. Types such as `Expr` are sometimes called algebraic datatypes, and types such as `TypedExpr` below that use this Scala feature are called *generalized algebraic datatypes*.

```
sealed abstract class TypedExpr[T]
case class CstI(n: Int) extends TypedExpr[Int]
case class Plus(e1: TypedExpr[Int], e2: TypedExpr[Int]) extends TypedExpr[Int]
case class Times(e1: TypedExpr[Int], e2: TypedExpr[Int]) extends TypedExpr[Int]

case class LessThanEq(e1: TypedExpr[Int], e2: TypedExpr[Int])
    extends TypedExpr[Boolean]
case class IfThenElse[T](cond: TypedExpr[Boolean], e1: TypedExpr[T], e2: TypedExpr[T])
    extends TypedExpr[T]
```

The type parameter `T` in `TypedExpr[T]` represents the type that the expression should return when evaluated. By filling in specific types for `T`, we guarantee that `eval` will return the correct type.

In fact, the datatypes contain the same information as the typing rules on page 69 of *Programming Language Concepts*. For example, `CstI` corresponds to the rule:

$$\frac{}{\rho \vdash i : \texttt{int}}$$

and `IfThenElse` corresponds to the rule

$$\frac{\rho \vdash e_1 : \texttt{bool} \quad \rho \vdash e_2 : t \quad \rho \vdash e_3 : t}{\rho \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : t}$$

where the type t corresponds to the Scala type parameter `T`.

The evaluation function takes a type parameter `T`, which represents the type of the desired result. Then, the argument is constrained to be an expression that evaluates to the correct type (that is, `TypedExpr[T]`):

```
def eval[T](e: TypedExpr[T]): T = e match {
  case CstI(n) => n
  case Plus(e1, e2) => eval(e1) + eval(e2)
  case Times(e1, e2) => eval(e1) * eval(e2)
  case LessThanEq(e1, e2) => eval(e1) <= eval(e2)
  case IfThenElse(cond, e1, e2) => if (eval(cond)) eval(e1) else eval(e2)
}
```

(i) Construct the following expressions in abstract syntax (that is, as `val`s). Note that type errors occur at Scala's compile-time. Comment out those that do not type check.

- `2 + 13 * 4`
- `if 23 <= 17 then 7 else 3 + 21`
- `if 0 then 1 else 2`
- `if 12 <= 3 then 42 else 3 <= 7`
- `if 12 <= 3 then 42 <= 1000 else 3 <= 7`

(ii) Add constant Booleans, subtraction, the remaining comparison operators, and short-circuiting AND and OR to the abstract syntax. Modify the evaluator to evaluate these correctly. It should be a Scala type error to attempt to construct expressions that do not type check.

(iii) Replace the separate abstract syntax for integer and Boolean constants with a single generic constant class that takes a type parameter and a value of that type. Update the evaluator accordingly. Leave the original data type commented out for your submission.

Hint: begin with `case class Const[T](...)`.