Representation of Algebraic Expressions

Khizar Siddiqui

October 21, 2020

Contents

1 Introduction

This documentation was created for Assignment 1 of 3MI3, it's purpose is to help understand the fundamentals behind how (and why) my program runs. Expressions are the most basic combinations of mathematical symbols to achieve a result. We will attempt to create an expression interpreter using both Scala and Prolog to help us solve these expressions to the scope of our operators.

2 Basic expressions

We will start off with expressions consisting of basic integers that have operations performed on them. The operators that we currently account for are **const, neg, abs, plus, times, minus, exp** representing an integer constant, negation, absolute value, addition, multiplication, subtraction and exponential respectively.

2.1 Scala

For Scala, we will classify our "expression" as a trait and our operators as classes that extend this trait. Our constant operator will only take an integer input since a constant can only be an integer but the rest of the operators will take inputs that match the type expression.

```
sealed trait Expr
case class Const(x: Int) extends Expr
case class Neg(x: Expr) extends Expr
case class Abs(x: Expr) extends Expr
case class Plus(x: Expr, y: Expr) extends Expr
case class Times(x: Expr, y: Expr) extends Expr
case class Minus(x: Expr, y: Expr) extends Expr
case class Exp(x: Expr, y: Expr) extends Expr
```

Now that our operators are defined, we can work on how to interpret them using a function named interpretExpr. Since each operator (except for Const) can have a detailed expression as an input, we attempt to solve the input before solving for our operator. This recursive strategy ends with a base case that is defined to be our Const, once our Const is interpreted we can attempt to solve for the remaining of our expression using the operator cases and their respective definitions.

```
def interpretExpr(e: Expr): Int = e match {
    case Const(x) => x
    case Neg(x) => interpretExpr(x) * -1
    case Abs(x) => if (interpretExpr(x) < 0) {
        interpretExpr(x) * -1
    } else {
        interpretExpr(x)
    }
    case Plus(x, y) => interpretExpr(x) + interpretExpr(y)
    case Times(x, y) => interpretExpr(x) * interpretExpr(y)
    case Minus(x, y) => interpretExpr(x) - interpretExpr(y)
    case Exp(x, y) => Math.pow(interpretExpr(x), interpretExpr(y)).toInt
}
```

2.2 Prolog

For Prolog, we define a predicate isExpr to recognise trees that represent our expressions. We do this by defining a base case for a constant value and recursively calling our operator inputs, which are represented by labels, to ensure all inputs are valid.

```
isExpr(constE(_)).
isExpr(negE(X)) :- isExpr(X).
isExpr(absE(X)) :- isExpr(X).
isExpr(plusE(X, Y)) :-
    isExpr(X),
    isExpr(Y).
isExpr(timesE(X, Y)) :-
    isExpr(X),
    isExpr(Y).
isExpr(minusE(X, Y)) :-
    isExpr(X),
    isExpr(Y).
isExpr(expE(X, Y)) :-
    isExpr(X),
    isExpr(Y).
```

We also define a predicate that helps us solve our expressions, much like what we created in the Scala part above. Our predicate behaves similar to isExpr in the sense that we have a base case for our constant value and recursively call the same function. But for this predicate, it solves for our result according to what our label, which defines an operator, states.

```
interpretExpr(constE(X), X).
interpretExpr(negE(E), X) :-
    interpretExpr(E, V),
    X is V * -1.
interpretExpr(absE(E), X) :-
    interpretExpr(E, V),
    X is abs(V).
interpretExpr(plusE(E1, E2), X) :-
    interpretExpr(E1, V1),
    interpretExpr(E2, V2),
    X is V1 + V2.
interpretExpr(timesE(E1, E2), X) :-
    interpretExpr(E1, V1),
    interpretExpr(E2, V2),
    X is V1 * V2.
interpretExpr(minusE(E1, E2), X) :-
    interpretExpr(E1, V1),
    interpretExpr(E2, V2),
    X is V1 - V2.
interpretExpr(expE(E1, E2), X) :-
    interpretExpr(E1, V1),
    interpretExpr(E2, V2),
    X is V1 ** V2.
```

3 Variable expressions

Now that we have defined our basic expressions, we can take a step forward and work on variable expressions for when we have more complicated expressions. Since we are introducing variables, it is only right we introduce substitution as well so that our variables have meaning.
We assume we do not intend to interpret just variables.

3.1 Scala

We define a new trait VarExpr for our implementation, where we can simply copy over our previous operators defined for basic expressions with the addition of our new operators. For variables, we use the type **Symbol** to denote a variable. In substitution we have 3 inputs: our expression to substitute on, the variable to substitute and the expression to substitute the variable with.

```
case class Var(x: Symbol) extends VarExpr
case class Subst(x: VarExpr, y: Symbol, z: VarExpr) extends VarExpr
```

And just like the definition for our classes above, the only changes for our interpreter are the addition of our new operators. Since only having a variable in our expression is not evaluate-able, we only define our substitution operator.

```
case Subst(x, y, z) => interpretVarExpr(substitute(x, y, z))
```

Notice the function substitute from above, it is a helper function that recursively substitutes all instances of the variable (if any) in our expression, with the base case being either a constant (nothing to substitute) or the variable (which is substituted). The reason for implementing this helper function was to bypass the strict return type of interpretVarExpr (which was Int) and allow us to have a return type of VarExpr.

```
def substitute(x: VarExpr, y: Symbol, z: VarExpr): VarExpr = x match {
    case Const(c) => Const(c)
    case Neg(c) => Neg(substitute(c, y, z))
    case Abs(c) => Abs(substitute(c, y, z))
    case Plus(c1, c2) => Plus(substitute(c1, y, z), substitute(c2, y, z))
    case Times(c1, c2) => Times(substitute(c1, y, z), substitute(c2, y, z))
    case Minus(c1, c2) => Minus(substitute(c1, y, z), substitute(c2, y, z))
    case Exp(c1, c2) => Exp(substitute(c1, y, z), substitute(c2, y, z))
    case Var(c) => {
        if (c == y) {
            z
        } else {
            Var(c)
        }
    }
    case Subst(a, b, c) => substitute(substitute(a, b, c), y, z)
}
```

3.2 Prolog

In Prolog, we define a predicate isVarExpr very much like the predicate for basic expressions with the addition of our 2 new operators, however we now have an additional base case that is our variable. Unlike Scala, we don't need a separate type for our variables as they can be denoted by atoms.

```
isVarExpr(var(_)).
isVarExpr(subst(X, Y, Z)) :-
    isVarExpr(X),
    isVarExpr(var(Y)),
    isVarExpr(Z).
```

We also define a predicate to interpret our substitution operator along with our typical basic expressions that we can just reuse from before. Note, we don't need a predicate for variables since variables alone cannot be interpreted. Our new predicate is:

```
interpretVarExpr(subst(X1, Y, Z), X2) :-
    substitute(X1, Y, Z, X3),
    interpretVarExpr(X3, X2).
```

Similar to the implementation in Scala, we define helper predicates that substitute all instances of the variable with the given expression, leaving us with an expression that is ready to be interpreted.

```
substitute(constE(C), _, _, constE(C)).
substitute(absE(C), V, E, X) :-
    substitute(C, V, E, X1),
    X = absE(X1).
substitute(negE(C), V, E, X) :-
    substitute(C, V, E, X1),
    X = negE(X1).
substitute(var(V), V, E2, E2).
substitute(var(V), _, _, var(V)).
substitute(plusE(C1, C2), V, E2, X) :-
    substitute(C1, V, E2, X1),
    substitute(C2, V, E2, X2),
    X = plusE(X1, X2).
substitute(timesE(C1, C2), V, E2, X) :-
    substitute(C1, V, E2, X1),
    substitute(C2, V, E2, X2),
    X = timesE(X1, X2).
substitute(minusE(C1, C2), V, E2, X) :-
    substitute(C1, V, E2, X1),
    substitute(C2, V, E2, X2),
    X = minusE(X1, X2).
substitute(expE(C1, C2), V, E2, X) :-
    substitute(C1, V, E2, X1),
    substitute(C2, V, E2, X2),
    X = expE(X1, X2).
substitute(subst(E1, V1, E2), V2, E3, X) :-
    substitute(E1, V1, E2, X1),
    substitute(X1, V2, E3, X).
```

4 Boolean expressions

We will now build more onto basic expressions by introducing Boolean expressions and operators such as true, false, and, or and not represented by **TT, FF, Band, Bor, Bnot** respectively.

4.1 Scala

In Scala, this is implemented by creating a new trait MixedExpr that inherits all the classes from basic expressions with the addition of our new operators. Since we intend to call true/false with TT/FF instead of TT()/FF() we will define those as objects rather than classes to skip the parenthesis.

```
case object TT extends MixedExpr
case object FF extends MixedExpr
case class Bnot(x: MixedExpr) extends MixedExpr
case class Bor(x1: MixedExpr, x2: MixedExpr) extends MixedExpr
case class Band(x1: MixedExpr, x2: MixedExpr) extends MixedExpr
```

For our interpreter, since we have 2 potential return types, we reuse our cases from basic expressions with a little modification in addition to the cases for our new operators. We also set our return type to the form Some(Either(Int, Boolean)) where Either accommodates 2 possible return types and Some accommodates instances of expression mixing between Boolean and Basic.

```
def interpretMixedExpr(e: MixedExpr): Option[Either[Int, Boolean]] = e match {
    case Const(x) => Some(Left(x))
    case Neg(x) => interpretMixedExpr(x) match {
        case Some(Left(c)) => Some(Left((c * -1)))
        case Some(Right(c)) => None
        case None => None
    }
    case Abs(x) => interpretMixedExpr(x) match {
        case Some(Left(c)) => {
            if (c < 0) {
                Some(Left(c * -1))
            } else {
                Some(Left(c))
            }
        }
        case Some(Right(c)) => None
        case None => None
    }
    case Plus(x, y) => interpretMixedExpr(x) match {
        case Some(Left(c1)) => interpretMixedExpr(y) match {
            case Some(Left(c2)) => Some(Left(c1 + c2))
            case Some(Right(c2)) => None
            case None => None
        }
        case Some(Right(c1)) => None
        case None => None
    }
    case Times(x, y) => interpretMixedExpr(x) match {
        case Some(Left(c1)) => interpretMixedExpr(y) match {
            case Some(Left(c2)) => Some(Left(c1 * c2))
            case Some(Right(c2)) => None
            case None => None
        }
```

```
            case Some(Right(c1)) => None
            case None => None
        }
        case Minus(x, y) => interpretMixedExpr(x) match {
            case Some(Left(c1)) => interpretMixedExpr(y) match {
                case Some(Left(c2)) => Some(Left(c1 - c2))
                case Some(Right(c2)) => None
                case None => None
            }
            case Some(Right(c1)) => None
            case None => None
        }
        case Exp(x, y) => interpretMixedExpr(x) match {
            case Some(Left(c1)) => interpretMixedExpr(y) match {
                case Some(Left(c2)) => Some(Left(Math.pow(c1, c2).toInt))
                case Some(Right(c2)) => None
                case None => None
            }
            case Some(Right(c1)) => None
            case None => None
        }
        case TT => Some(Right(true))
        case FF => Some(Right(false))
        case Bnot(x) => interpretMixedExpr(x) match {
            case Some(Right(true)) => Some(Right(false))
            case Some(Right(false)) => Some(Right(true))
            case Some(Left(x)) => None
            case None => None
        }
        case Bor(x1, x2) => interpretMixedExpr(x1) match {
            case Some(Right(true)) => Some(Right(true))
            case Some(Right(false)) => interpretMixedExpr(x2) match {
                case Some(Right(true)) => Some(Right(true))
                case Some(Right(false)) => Some(Right(false))
                case Some(Left(x)) => None
                case None => None
            }
            case Some(Left(x)) => None
            case None => None
        }
        case Band(x1, x2) => interpretMixedExpr(x1) match {
            case Some(Right(false)) => Some(Right(false))
            case Some(Right(true)) => interpretMixedExpr(x2) match {
                case Some(Right(true)) => Some(Right(true))
                case Some(Right(false)) => Some(Right(false))
                case Some(Left(x)) => None
                case None => None
            }
            case Some(Left(x)) => None
            case None => None
        }
}
```

Every case checks to see if it's input is of the other return type or not. If it is, it will simply return None.

4.2 Prolog

For Prolog, we can define a predicate isMixedExpr that builds upon basic expressions by reusing predicates and creating predicates for our new operators. The labels representing our new operators are:

```
isMixedExpr(tt).
isMixedExpr(ff).
isMixedExpr(bnot(X)) :- isMixedExpr(X).
isMixedExpr(band(X, Y)) :-
    isMixedExpr(X),
    isMixedExpr(Y).
isMixedExpr(bor(X, Y)) :-
    isMixedExpr(X),
    isMixedExpr(Y).
```

We also define a predicate interpretMixedExpr to help us interpret our Boolean expressions along with our basic expressions. For basic expressions, we can reuse our old predicates and rename them. Our new predicates are:

```
interpretMixedExpr(tt, true).
interpretMixedExpr(ff, false).
interpretMixedExpr(bnot(B), X1) :-
    interpretMixedExpr(B, X2),
    negate(X2, X1).
interpretMixedExpr(band(B1, B2), X1) :-
    interpretMixedExpr(B1, X2),
    interpretMixedExpr(B2, X3),
    and(X2, X3, X1).
interpretMixedExpr(bor(B1, B2), X1) :-
    interpretMixedExpr(B1, X2),
    interpretMixedExpr(B2, X3),
    or(X2, X3, X1).
```

We have again defined some helper predicates (negate, and and or) that take a Boolean input/inputs and return the appropriate result:

```
negate(true, false).
negate(false, true).

and(true, true, true).
and(_, _, false).

or(false, false, false).
or(_, _, true).
```

In the case of an expression that has a mix of basic and Boolean expressions, it will simply return False since Prolog predicates won't have an instance for it.