Pamphlet 6, INF222, Spring 2020

6.1 Calculator language structure

Looking at some of the previous pamphlets, we notice that there are important difference between the structural aspects of the calculator:

- p1: `CalculatorAST` – `CalculatorInterpreter` – has no storage.
- p2: `CalculatorRegisterAST` – `CalculatorRegisterInterpreter` – has a fixed set of registers.
- p4: `CalculatorVariableAST` – `CalculatorVariableInterpreter` – has an unbounded set of variables and requires declaration before use.

However our recent upgrade from pamphlet 4 to pamphlet 5:

- p4: `CalculatorVariableAST` – `CalculatorVariableInterpreter` – has an unbounded set of variables and requires declaration before use, has 4 primitive numerical operations.
- p5: `CalculatorVariableIdivremAST` – `CalculatorVariableIdivremInterpreter` – has an unbounded set of variables and requires declaration before use, has 6 primitive numerical operations.

There is no qualitative difference between these two languages, just a pragmatical difference between the selection of numerical operations.

So let us ask ourselves: can we structure the calculator language tools such that:

- Changing the structural aspects of the language implies modifying the interpreter.
- Changing selection of primitive numerical operations leaves the AST & interpreter unmodified.

6.2 Designing reusable calculator tools

Our goal here is to build reuseable tools linked to the calculator language structure, yet independent of the selection of primitive numerical operations.

6.2.1 Designing a reusable AST

Looking at the p5 AST we observe that calling a primitive numerical operation is identifying the operation, and getting access to the operation's list of arguments. Making the AST reuseable across an open ended set of primitive numerical operations, we need to use an open ended set of operation names, e.g., Haskell integers or strings, rather than AST constructors. Likewise, we need a list of arguments to the function, since the actual number of arguments for a function depends on the function. Here we use `String` to name the function.

```
  Fun String [CalcExprAST]
```

Below we have replaced the specific listing of numerical operations with the general form.

```haskell
-- | AST for variable based integer calculator with open ended set of primitive functions.
--
-- Author Magne Haveraaen
-- Since 2020-03-19

module CalculatorExternalPrimitivesAST where


-----------------
-- | Expressions for a calculator with variables.
-- The calculator supports integer literals Lit,
-- an open ended set of primitive functions Fun, and
-- an open ended set of variables Var.
data CalcExprAST
  = Lit Integer
  | Fun String [CalcExprAST]
  | Var String
  deriving (Eq, Read, Show)


-- | Statement for declaring (setting) and changing (assigning) a variable
data CalcStmtAST
  = SetVar String CalcExprAST
  | AssVar String CalcExprAST
  deriving (Eq, Read, Show)



-----------------
-- | A primitive function declaration defines
-- a function name (String), a list of parameter types (Strings), and a return type (String).
type Primitive = (String ,[ String ], String )

-- | The semantics of a call of a primitive function is a mapping
-- from the function name (String) and argument list of integers to the resulting integer.
type PrimitiveSemantics = String -> [Integer ] -> Integer



-----------------
-- | A few ASTs for variable based CalcExprAST.
-- Note that primitive function names in AST2 and AST3 are
-- chosen for historical reasons (the same as in pamphlet 1).
-- These examples show the notation, but are not practically useful.
calculatorExprVPAST1
  = Lit 4
calculatorExprVPAST2
  = Fun "Neg" [Fun "Mult" [Fun "Add" [( Lit 3) ,( Fun "Sub" [( Lit 7) ,( Lit 13) ]) ],( Lit 19) ]]
calculatorExprVPAST3
  = Fun "Add" [(Var "Reg1"), (Var "Reg4")]
calculatorExprVPAST4
  = Var "Reg2"
```

```
-- | A few CalcStmtASTs for setting and assigning variables.
-- These examples are not practically useful.
-- They only show that the AST notation is understood.
calculatorStmtVEPAST1
  = SetVar "Reg4" calculatorExprVPAST1
calculatorStmtVEPAST2
  = SetVar "Reg1" calculatorExprVPAST2
calculatorStmtVEPAST3
  = AssVar "Reg2" calculatorExprVPAST3
calculatorStmtVEPAST4
  = AssVar "Reg1" calculatorExprVPAST4


-------------------
```

Here we introduced two new types to help with writing the calculator tools.

- `Primitive` – a declaration of a primitive function for the calulator. It is a triple with a function name (a string), a list of argument types (for now a list of strings `"Integer"` since this is our calculator's domain), and the return type (for now the string `"Integer"`). See examples below.

- `PrimitiveSemantics` – the type for functions providing the semantics of the calculator's primitive functions. This will take the primitive function name (string), a list of integers (arguments to the function call), and return an integer (the effect of applying the primitive function to the arguments). This is what a semantic function needs to do for a `Fun` call in the AST.

The code examples `calculatorExprVPAST1`, ... are not very useful this time, since the actual names of primitive functions are not known here. However, the examples show the syntax of the AST, which may be of help to understand it.

Note the similarity between what we are doing here and the discussion of "Less stringent presentation of BTL in ESL" in section 2.4 of Note 1 (Abstract Syntax). In both cases we are making the AST less specific to open up for more reusable tools.

6.2.2 Primitive Operations

The primitive operations for a calculator are shown below. These are the 6 functions provided in pamphlet 5.

The code is organised in four groups.

- The declaration of the primitives `integeroperations :: [Primitive]` – name, parameter list, return type for the 6 primitives.

- The semantic function `integersemantics :: PrimitiveSemantics` – defines for every primitive function and appropriate argument list what result to compute.

  Note how we can use pattern matching to give appropriate error messages when dividing/doing remainder by 0.

  Here we risk the AST has some improper function call (wrong function name or number of arguments), so the last alternative captures this and prints an error message.

- Unit test `unittestCalculatorIntegerPrimitives` – checks that each of the declared functions in `integeroperations` are handled by the semantic function `integersemantics`.
- A main function `main` – an interactive calculator with the primitives above. This function prints the declarations of the primitives using Haskell declaration style before it calls the actual interactive calculator.

```haskell
-- | A selection of integer function declarations and their semantics.
--
-- Author Magne Haveraaen
-- Since 2020-03-21

module Calculator6IntegerPrimitives where

-- Use the AST format for calculators with an open set of operations.
import CalculatorExternalPrimitivesAST

-- Use the interpreter for calculators with an open set of operations.
import CalculatorExternalPrimitivesInterpreter

-- Use calculator state for variables and store.
import CalculatorState


------------------------------
-- | Declaration of operations and their argument list & return type.
integeroperations :: [Primitive]
integeroperations
  = [
      ("Add", ["Integer","Integer"], "Integer"),
      ("Mult", ["Integer","Integer"], "Integer"),
      ("Sub", ["Integer","Integer"], "Integer"),
      ("Neg", ["Integer"], "Integer"),
      ("Idiv", ["Integer","Integer"], "Integer"),
      ("Rem", ["Integer","Integer"], "Integer")
    ]

------------------------------
-- | Semantics of chosen integer operations.
integersemantics :: PrimitiveSemantics
integersemantics "Add" [i1,i2] = i1 + i2
integersemantics "Mult" [i1,i2] = i1 * i2
integersemantics "Sub" [i1,i2] = i1 - i2
integersemantics "Neg" [i] = - i
integersemantics "Idiv" [i1,0] = error $ "Cannot do integer division of " ++ (show i1)
    ++ " by 0."
integersemantics "Idiv" [i1,i2] = quot i1 i2
integersemantics "Rem" [i1,0] = error $ "Cannot do remainder of " ++ (show i1) ++ " 
    by 0."
integersemantics "Rem" [i1,i2] = rem i1 i2
integersemantics fname alist = error $ "Unknown function name/arg list " ++ fname ++
    (show alist)
```

```haskell
-- ---------------
-- | Unit test of the integeroperations:
-- for each declaration in integeroperations checks that there is a corresponding integersemantics.
unittestCalculatorIntegerPrimitives = do
  print $ "-- unittestCalculatorIntegerPrimitives "
  -- print $ show $ map testfunction integeroperations
  print $
    -- Expected result of calling the declared functions on relevant argument lists.
    if (map testfunction integeroperations) == [21,110,-1,-10,0,10]
    then "Unit tests hold"
    else "Tests failed"


-- | Turns a Primitive declaration into a call of the primitive function with an argument list
-- and calls integersemantics to compute the resulting value of the function with arguments.
testfunction :: Primitive -> Integer
testfunction (fname, plist, res) = integersemantics fname (testdatalist plist)

-- | Creates test data (integer numbers) from a declared parameter list: one integer per argument
testdatalist :: [String] -> [Integer]
testdatalist plist = [10..9+ toInteger (length plist)]



-- ---------------
-- | Interactive calculator with variables and given selection of integer operations.
{- | Run the following commands in sequence at the prompt
SetVar "Reg4" (Lit 4)
SetVar "Reg1" (Fun "Neg" [Fun "Mult" [Fun "Add" [Lit 3,Fun "Sub" [Lit 7, Lit 13]], Lit
    19]])
SetVar "Reg2" (Fun "Add" [Var "Reg1",Var "Reg4"])
AssVar "Reg1" (Var "Reg2")
AssVar "Reg1" (Fun "Add" [Var "Reg1",Var "Reg4"])
AssVar "Reg1" (Fun "Add" [Var "Reg1",Var "Reg4"])
SetVar "v9" (Fun "Add" [Fun "Mult" [Fun "Idiv" [Var "Reg1",Var "Reg4"], Var "Reg4"], Fun
    "Rem" [Var "Reg1",Var "Reg4"]])
show

-}
main = do
  putStrLn $ "-- Interactive calulator with variables and open ended set of
      operations."
  putStrLn $ "The set of primitive functions (and their arguments):"
  -- putStrLn $ " " ++ (show integeroperations)
  putStr $ foldl (++) ""
    (map
      (\(fn, args, res) -> ("  " ++ fn ++ " :: " ++ (foldr listargcomma "" args) ++ "
          -> " ++ res ++ "\n"))
      integeroperations
    )
  mainCalculatorVariableAsk integersemantics newstate
```

```haskell
-- | Function for inserting a comma between two nonempty strings.
listargcomma str1 str2 =
  if str1 == "" && str2 == ""
  then ""
  else
    if str1 == ""
    then str2
    else
      if str2 == ""
      then str1
      else str1 ++ ", " ++ str2
```

6.3 Reusable Calculator Interpreter

The reuseable interpreter will look quite similar to our previous interpreters.

The core difference is that the semantics of `Fun` will use the external semantic function for the primitives.

```haskell
evaluate :: PrimitiveSemantics -> CalcExprAST -> State -> Integer
evaluate primsem (Fun str args) state = primsem str ( evaluatelist primsem args state )
```

Note how the semanics of the primitive functions `primsem` has to be a parameter to the evaluation function. We also use a support function to evaluate the argument list to the primitive functions.

```haskell
evaluatelist :: PrimitiveSemantics -> [CalcExprAST] -> State -> [Integer]
evaluatelist primsem (arg:args) state
  = (evaluate primsem arg state ):( evaluatelist primsem args state )
evaluatelist primsem [] state = []
```

Alternatively it is possible to express this using a map function directly in the evalution of the primitive function.

Similarly, we will need `primsem` to be a parameter to the execute function.

```haskell
execute :: PrimitiveSemantics -> CalcStmtAST -> State -> State
```

Here `primsem` is passed on to the evaluate function. The handling of variables is independent of the primitives of the language.

6.4 Task

6.4.1 Implement an interpreter with external primitive functions

Implement an interpreter with external primitive functions as discussed above.

Write a unit test for the interpreter, i.e., a unit test checking the declaration and modification of variables. The unit test for this interpreter does not know anything about the primitive functions discussed above. It will only be able to test for handling of variables and literals: the part of this calculator that is independent of the choice of primitives. This is a simpler unit test than earlier where we also checked evaluation of expressions. This simpler unit test function still needs to call `evaluate` and will thus need some ad hoc semantic function created just for testing purposes.

Implement the `mainCalculatorVariable` function for the interactive calculator (as called by `main` in `Calculator6IntegerPrimitives` above). This function requires only minor tweaks to the previously implemented `mainCalculatorVariable`* functions from pamphlet 5. The main changes will be renaming the interpreter module's `main` to `mainCalculatorVariable`, then giving the primitive semantics function as argument to all of the functions implementing the interactive calculator.

```haskell
mainCalculatorVariable :: PrimitiveSemantics -> IO ()
mainCalculatorVariableAsk :: PrimitiveSemantics -> State -> IO ()
mainCalculatorVariableShowstate :: PrimitiveSemantics -> State -> IO ()
mainCalculatorVariableExc :: PrimitiveSemantics -> String -> State -> IO ()
```

6.4.2 Extend `Calculator6IntegerPrimitives` with more primitives

Add new primitive operations like *greatest common divisor* or *least common multiple* to the set of operations declared in the `Calculator6IntegerPrimitives` module. Such functions are also defined in Haskell's library so are easy to use for the semantic functions.

If you first add the declarations to `integeroperations` you can check that the unit test `unittestCalculatorIntegerPrimitives` discovers that these are not implemented, before you extend the semantic function `integersemantics` with the new primitives.