Pamphlet 8, INF222, Spring 2021

8.1 Calculator for more domains

Our calculator so far has been computing with integers. It could just as well compute with reals or complex numbers. That would also give us a larger reportoir of functions we can calculate with, such as trigonometri and exponentiation/logarithms.

Instead of making a real or complex version of our calculator, we will do this in steps.

1. Generalise the calculator so that it no longer is tied to a specific type of data.
2. Provide versions of the calculator for several types of data.

8.2 Make calculator type independent

So far we have separated the primitive set of functions from the calculator infrastructure. This makes it easy for us to change primitive functions. The calculator is still tied to integer literals (see the AST from previous pamphlets).

Here we will break this dependency by introducing a type parameter `valuedomain` for the calculator's value domain.

```
data CalcExprAST valuedomain
  = Lit valuedomain
  | Fun String [CalcExprAST valuedomain]
  | Var String
  deriving (Eq, Read, Show)
data CalcStmtAST valuedomain
  = SetVar String (CalcExprAST valuedomain)
  | AssVar String (CalcExprAST valuedomain)
  deriving (Eq, Read, Show)
```

This has repercussions for most of our other support modules.

- State: The `State` and `Store` must be given a `valuedomain` parameter so that they can store values for an arbitrary value domain. Many of the support functions need a value domain parameter, and for those functions that show a related error message it will be necessary to declare the `valuedomain` type parameter to be showable.

```
type State valuedomain = (Environment,Store valuedomain)
newstate :: State valuedomain
getvalue :: String -> State valuedomain -> valuedomain
addvariable :: String -> valuedomain -> State valuedomain -> State valuedomain
changevalue :: Show valuedomain => String -> valuedomain -> State valuedomain
    -> State valuedomain

type Environment = [( String , Integer )]
emptyenvironment :: Environment
addenv :: String -> Integer -> Environment -> Environment

type Store valuedomain = Array Integer valuedomain
emptystore :: Store valuedomain
getstore :: Store valuedomain -> Integer -> valuedomain
setstore :: Show valuedomain => Integer -> valuedomain -> Store valuedomain ->
    Store valuedomain
enlargestore :: Store valuedomain -> valuedomain -> (Integer,Store valuedomain)
```

- Interpreter (separated from the interactive calculator): The same kind of changes as for state, i.e., introduce a showable `valuedomain` type parameter to the evaluation functions.
- Interactive calculator: Mainly the same changes as above. This is a good time to separate the interactive calculator from the interpreter, and also gather the print documentation functions from the calculator's primitive function module.

8.2.1 Tasks

1. Upgrade the support modules as described above.
2. Upgrade the integer calculator to use the upgraded support modules. Extend the list of primitive functions to include `Abs` `Sqr` `Succ` `Pred` using Pascal semantics.

This should remove the calculator's dependency on integers, yet provide us with a more flexible calculator than before.

8.2.2 Implement a calculator for reals

TASK: Provide a calculator for reals using the value domain independent support modules.

You can choose the following list of primitive functions. Using Haskell's **`Double`** as the semantics gives easy support for most of these functions.

```
realoperations :: [Primitive]
realoperations
 = [
     ("Add", ["Real","Real"],"Real","Add two reals"),
     ("Mult", ["Real","Real"],"Real","Multiply two reals"),
     ("Sub", ["Real","Real"],"Real","Subtract two reals"),
     ("Neg", ["Real"],"Real","Negate (change sign of) a real"),
     ("Slash", ["Real","Real"],"Real","Division of two reals"),
     ("Abs", ["Real"],"Real","The absolute value ( positive ) a real"),
     ("Sqr", ["Real"],"Real","Squaring a real"),
     ("Sin", ["Real"],"Real","Sine of a real ( radians )"),
     ("Cos", ["Real"],"Real","Cosine of a real ( radians )"),
     ("Exp", ["Real"],"Real","Exponent of a real ( natural exponent)"),
     ("Ln", ["Real"],"Real","Natural logarithm of a real"),
     ("Sqrt", ["Real"],"Real","Square root of a non-negative real"),
     ("Arctan", ["Real"],"Real",
       "Arc tangent ( inverse tangent ) of a real ( radians ) \ nfirst and fourth quadrant"),
     ("Idiv", ["Real","Real"],"Real","Integer division rounded towards zero"),
     ("Rem", ["Real","Real"],"Real","Remainder when doing integer division"),
     ("Succ", ["Real"],"Real","Successor , adding 1 to a number"),
     ("Pred", ["Real"],"Real","Predessor , subtracting 1 from a number")
   ]
```

Note that this list includes `"Idiv"` and `"Rem"`. If we define the former for arguments `i1` and `i2` by **`fromIntegral`** `(`**`truncate`** `(i1 / i2))` we get a function on the reals that rounds division to the nearest integer towards 0. This is similar to our integer version of `"Idiv"`. Then `"Rem"` should be the corresponding remainder function, i.e., for arguments `i1` and `i2` it yields `i1 -` **`fromInteger`** `(`**`truncate`** `(i1 / i2)) * i2`. Such a remainder function is available in the C/C++ libraries as `fmod`.